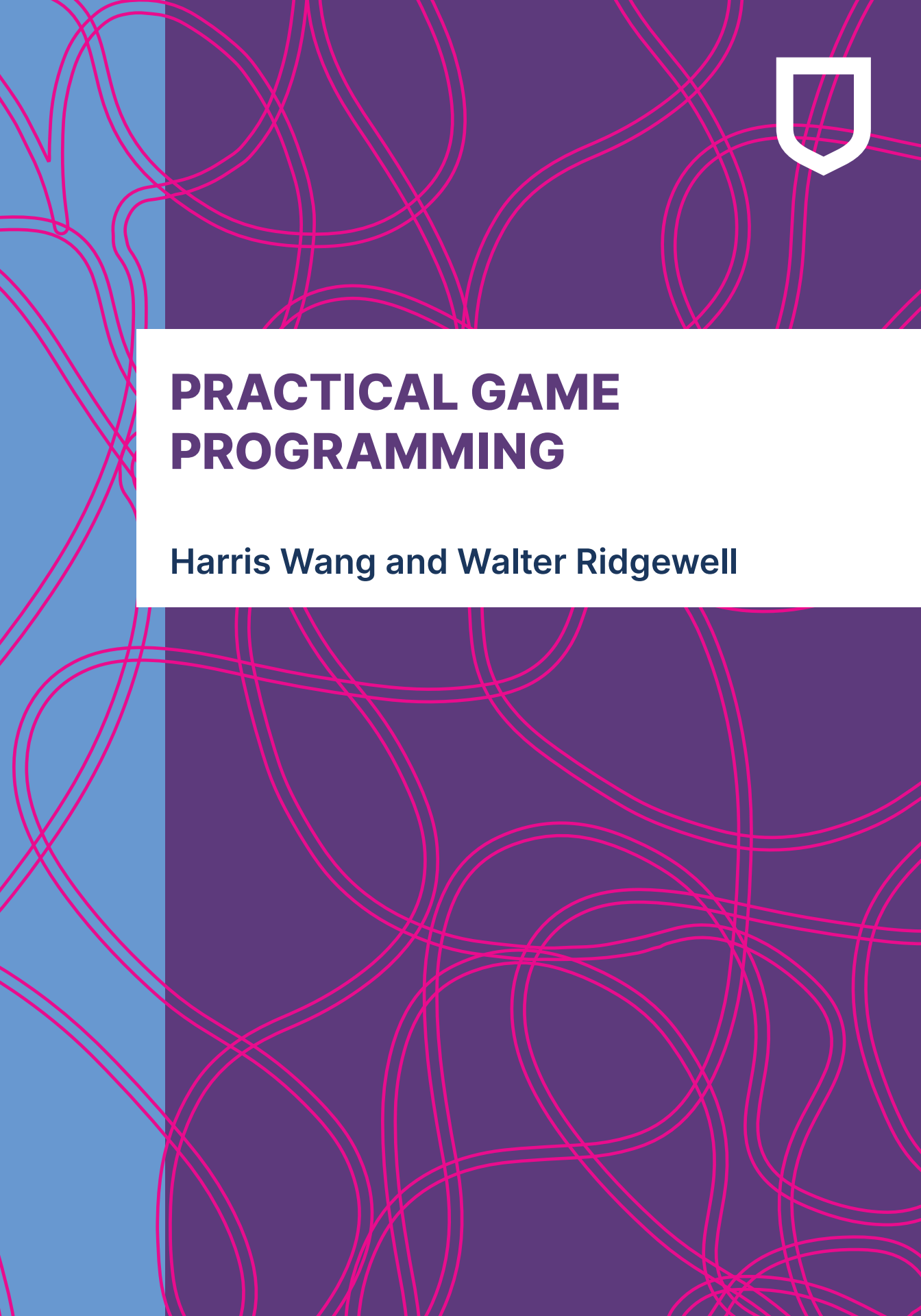

PRACTICAL GAME PROGRAMMING

Harris Wang and Walter Ridgewell

Practical Game Programming

This page intentionally left blank

Practical Game Programming

Harris Wang

Walter Ridgewell

Remix
Athabasca University

Published in 2026 by Remix, an imprint of Athabasca University Press
1 University Drive, Athabasca, AB T9S 3A3

DOI: https://doi.org/10.15215/remix/9781998944224.01
ISBN: 9781998944217 (paper) | 9781998944224 (pdf) | 9781998944231 (epub)

Cover design by Lisa Mentz and Kyle Flemmer
Illustrations by Walter Ridgewell and Kyle Flemmer

For more information, please visit aupress.ca or email OERpublishing@athabascau.ca.

Contents

This page intentionally left blank

Acknowledgements

The development of *Practical Game Programming* has benefited from a blend of human expertise and emerging technologies. Portions of the content were generated with the assistance of artificial intelligence tools and have been thoroughly reviewed and edited by the OER authors to ensure clarity, accuracy, and pedagogical integrity.

We gratefully acknowledge the support of the Athabasca University OER Publishing Support team, especially Kyle Flemmer and Dan Cockcroft, for their invaluable advice, editorial contributions, procedural guidance, and in-kind funding. Their commitment to open educational resources and accessible learning has been instrumental in the creation and publication of this textbook.

https://doi.org/10.15215/remix/9781998944224.01

This page intentionally left blank

1

Introduction

1.1. Introduction to Video Games and Game Programming

Video games have become one of the most influential forms of digital media, shaping entertainment, culture, technology, and even education. What began decades ago as simple experiments on early computers has grown into a global industry that blends art, science, mathematics, storytelling, and interactive design. Today, billions of people play games across a vast spectrum of genres and platforms—from arcade cabinets and home consoles to PCs, mobile devices, and virtual reality systems.

This textbook, *Practical Game Programming*, introduces the foundational concepts, techniques, and technologies behind modern video games. It is designed as an open educational resource (OER) to support learners, educators, and independent developers who wish to understand not only how games work but how they are created.

1.1.1. Why Learn Game Programming?

Game programming is a unique intersection of creativity and computation. A single game integrates:

Graphics and animation to render characters, scenes, and visual effects

Input handling for keyboards, mice, joysticks, touchscreens, and controllers

Sound and music to enhance atmosphere and player feedback

Physics and simulation to model movement, collisions, and interactions

Artificial intelligence that drives nonplayer characters (NPCs)

https://doi.org/10.15215/remix/9781998944224.01

User interface design to support interaction, navigation, and control
Software engineering principles for maintainable, scalable game code

Learning how to build games provides hands-on experience with many core areas of computer science. It also fosters problem-solving, creativity, algorithmic thinking, and iterative design.

1.1.2. Our Focus: A Practical, Hands-On Approach

To ensure you can apply what you learn immediately, this textbook focuses on practical game programming using C/C++ and the Allegro 5 library. Allegro 5 is an open-source, cross-platform game programming library that supports:

- 2D graphics
- Event handling
- Keyboard, mouse, and joystick input
- Sound and audio streaming
- Timers and animation
- File operations
- Add-ons for images, fonts, primitives, and more

You will also learn to set up your development environment using:

- Visual Studio Code
- GCC (MinGW or system compiler)
- CMake
- Ninja build system

This toolchain works consistently across Windows, Linux, and macOS, making it ideal for students and open-source developers.

1.1.3. What You Will Learn in This Textbook

Throughout the upcoming chapters, you will explore:

- The history and evolution of video games
- Major genres and game design principles
- The structure of game programs
- Handling input devices
- Displaying graphics and text
- Using timers and game loops
- Working with sound and music
- Implementing physics and collision detection

https://doi.org/10.15215/remix/9781998944224.01

- Building user interfaces
- Applying AI techniques
- Developing complete games step-by-step

Examples and exercises in each chapter help reinforce these concepts and gradually build toward more complex projects.

1.1.4. Before We Begin: Understanding Where Games Come From

To build games effectively, it helps to understand the origins of the medium—the technologies, ideas, innovations, and pioneers that shaped the games we play today. The evolution of video games—from early mainframe experiments to modern 3D, online, and mobile experiences—is not just a historical timeline; it is a road map of technological advancement that directly influences how we design and program games today.

With this context in mind, let us explore how video games began and how they have transformed over the decades.

1.2. History of Video Games

The history of video games is fascinating, and its long journey has spanned several decades.

1.2.1. Early Beginnings Between the 1950s and 1960s

Computer scientists and engineers started designing simple games and simulations on minicomputers and mainframes. Some important games developed during this period of time include:

***OXO* (1952):** Created by British professor A. S. Douglas at the University of Cambridge, this was a tic-tac-toe game developed as part of his doctoral dissertation.

***Tennis for Two* (1958):** Developed by William Higinbotham at the Brookhaven National Laboratory, this game simulated a tennis match on an oscilloscope screen.

***Spacewar!* (1962):** Created by Steve Russell and his colleagues at the Massachusetts Institute of Technology (MIT), this was one of the first interactive video games, featuring space combat between two ships.

***The Brown Box* (1967):** Developed by Ralph Baer and his team at Sanders Associates, this was the prototype for the first multiplayer,

https://doi.org/10.15215/remix/9781998944224.01

multiprogram video game system. It was later licensed to Magnavox and released as the Magnavox Odyssey in 1972.

1.2.2. Birth of Consumer Video Game Hardware in the 1970s

The 1970s were a transformative decade for video games, marking the transition from experimental game projects to mainstream entertainment. Some key developments during the time include the following.

Early Arcade Games

***Computer Space* (1971):** Created by Nolan Bushnell and Ted Dabney, this was the first commercially sold arcade video game.

***Pong* (1972):** Developed by Atari, Pong became the first commercially successful arcade game, popularizing video games in arcades.

Home Consoles

Magnavox Odyssey (1972): The first home video game console, developed by Ralph Baer and his team, featured simple games like table tennis and shooting games.

Atari 2600 (1977): This console introduced interchangeable game cartridges, allowing players to switch games easily. It became one of the most popular consoles of the era.

Iconic Games

***Space Invaders* (1978):** Developed by Tomohiro Nishikado, this game became a cultural phenomenon and significantly boosted the arcade industry.

***Asteroids* (1979):** Created by Atari, this game featured vector graphics and became one of the best-selling arcade games of all time.

Technological Advancements

Microprocessors: The introduction of microprocessors in the late 1970s allowed for more complex games and better graphics.

ROM Cartridges: These enabled consoles like the Atari 2600 to offer a variety of games without needing built-in hardware for each one.

1.2.3. Boom and Bust Cycles Between the 1970s and 1980s

The period between the 1970s and 1980s was a pivotal time for the video game industry, marked by the following significant advancements and the emergence of iconic games and consoles:

https://doi.org/10.15215/remix/9781998944224.01

Arcade Boom: The late 1970s saw the rise of arcade games, with titles like Space Invaders (1978) by Tomohiro Nishikado becoming a massive hit and sparking the golden age of arcade video games.

Home Consoles: The Atari 2600 was released in 1977, revolutionizing home gaming with its use of interchangeable cartridges.

***Pac-Man* (1980):** Created by Namco, Pac-Man became a cultural phenomenon and one of the most famous arcade games of all time.

***Donkey Kong* (1981):** Developed by Nintendo, this game introduced the world to Mario and became a major success.

Video Game Crash of 1983: An oversaturation of low-quality games led to a significant downturn in the video game market, particularly in North America.

Nintendo Entertainment System (NES): Released in 1985, the NES revitalized the home console market and introduced iconic games like *Super Mario Bros.* and *The Legend of Zelda*.

Graphics and Sound: The transition from simple, blocky graphics to more detailed and colourful visuals, along with improved sound capabilities, enhanced the gaming experience.

Home Computers: The rise of home computers like the Commodore 64 and ZX Spectrum provided new platforms for game development and distribution.

1.2.4. The Video Game Crash and Rebound in 1983

The Video Game Crash of 1983 was a significant event that nearly ended the burgeoning video game industry.

Causes of the Crash

Market Saturation: By the early 1980s, the market was flooded with numerous consoles and a plethora of low-quality games. This oversaturation confused consumers and led to a decline in interest.

Poor-Quality Games: Many games released during this period were rushed and poorly made. The infamous *E.T. the Extra-Terrestrial* game for the Atari 2600 is often cited as a prime example.

Competition from Personal Computers: The rise of affordable personal computers provided an alternative platform for gaming, drawing consumers away from consoles.

https://doi.org/10.15215/remix/9781998944224.01

Impact of the Crash

Financial Losses: The industry saw a dramatic drop in revenue, from around $3.2 billion in 1982 to just $100 million by 1985.

Bankruptcies: Many companies producing video game consoles and games went bankrupt, and retailers were left with unsold inventory.

Loss of Consumer Confidence: The crash led to a significant loss of confidence in video games as a viable form of entertainment.

Rebound and Recovery

Nintendo Entertainment System (NES): The release of the NES in 1985 played a crucial role in revitalizing the industry. Nintendo's strict quality control and innovative marketing strategies helped restore consumer confidence.

New Business Models: The industry adopted new business models, including licensing agreements and third-party game development, which helped ensure higher-quality games.

Technological Advancements: Improved graphics, sound, and gameplay mechanics attracted a new generation of gamers.

The crash and subsequent recovery reshaped the video game industry, leading to the establishment of practices and standards that continue to influence game development today.

1.2.5. Late 1980s and Early 1990s

The period between the late 1980s and early 1990s was a dynamic time for the video game industry, marked by significant technological advancements and the release of many iconic games and consoles.

Late 1980s

Nintendo Entertainment System (NES): The NES continued to dominate the home console market, with games like *Super Mario Bros. 3* (1988) and *The Legend of Zelda* (1986) becoming massive hits.

Sega Genesis (1988): Known as the Mega Drive outside North America, the Sega Genesis introduced 16-bit graphics and became a strong competitor to the NES.

Handheld Gaming: The release of the Game Boy in 1989 by Nintendo revolutionized portable gaming. It came bundled with *Tetris*, which became a global phenomenon.

https://doi.org/10.15215/remix/9781998944224.01

Early 1990s

Super Nintendo Entertainment System (SNES): Released in 1990 in Japan and 1991 in North America, the SNES brought advanced graphics and sound capabilities, with iconic games like *Super Mario World* and *The Legend of Zelda: A Link to the Past*.

Sega Genesis vs. SNES: The early 1990s saw the height of the console wars between Sega and Nintendo, with both companies releasing numerous hit games and marketing campaigns.

PC Gaming: The early 1990s also saw significant growth in PC gaming, with titles like *Doom* (1993) and *Myst* (1993) pushing the boundaries of graphics and gameplay.

Technological Advancements

16-bit Graphics: The transition from 8-bit to 16-bit graphics allowed for more detailed and colourful visuals.

Sound and Music: Improved sound chips enabled more complex and immersive audio experiences in games.

CD-ROM Technology: The introduction of CD-ROMs provided greater storage capacity, allowing for more extensive and detailed games, and CD-ROMs were incorporated into consoles, impacting Sega and Nintendo.

1.2.6. Late 1990s and Beyond

The late 1990s and beyond saw tremendous growth and innovation in the video game industry.

Late 1990s

3D Graphics: The transition from 2D to 3D graphics was a major leap. Games like *Super Mario 64* (1996) and *The Legend of Zelda: Ocarina of Time* (1998) showcased the potential of 3D environments.

PlayStation (1994): Sony's PlayStation revolutionized gaming with its CD-ROM format, allowing for larger and more complex games. Titles like *Final Fantasy VII* (1997) and *Metal Gear Solid* (1998) became iconic.

PC Gaming: The late 1990s saw the rise of influential PC games like *Half-Life* (1998) and *StarCraft* (1998), which pushed the boundaries of storytelling and multiplayer gaming.

https://doi.org/10.15215/remix/9781998944224.01

Early 2000s

Online Gaming: The advent of broadband internet enabled online multiplayer games. *World of Warcraft* (2004) and *Halo 2* (2004) were pivotal in popularizing online gaming.

New Consoles: The release of the PlayStation 2 (2000), Xbox (2001), and Nintendo GameCube (2001) brought enhanced graphics and new gameplay experiences.

Mobile Gaming: The early 2000s also saw the rise of mobile gaming with devices like the Game Boy Advance (2001) and the Nintendo DS (2004).

Mid to Late 2000s

High-Definition Gaming: The Xbox 360 (2005) and PlayStation 3 (2006) introduced high-definition graphics, significantly improving visual fidelity.

Motion Controls: The Nintendo Wii (2006) popularized motion controls, making gaming more accessible to a broader audience.

Indie Games: Platforms like Steam and Xbox Live Arcade provided a space for indie developers to thrive, leading to the success of games like *Braid* (2008) and *Minecraft* (2009).

2010s and Beyond

Virtual Reality (VR): The introduction of VR headsets like the Oculus Rift (2016) and PlayStation VR (2016) opened new possibilities for immersive gaming experiences.

Streaming and Cloud Gaming: Services like Twitch allowed gamers to stream their play sessions online, while Google Stadia let users enjoy high-end games without needing powerful hardware.

Next-Gen Consoles: The PlayStation 5 (2020) and Xbox Series X (2020) brought even more advanced graphics, faster load times, and new features like ray tracing.

1.2.7. Future of the Video Game Industry

The video game industry is massive and continues to grow rapidly. As of 2024, the global video game market generated approximately US$182.7 billion in revenue, reflecting a year-over-year increase of about 3.2% compared to 2023. Industry analysts forecast steady growth, with global market revenues expected to exceed US$200 billion by 2027, corresponding to a compound annual growth rate (CAGR) of roughly 4–5% in the near term. In terms of audience size, the

https://doi.org/10.15215/remix/9781998944224.01

global gaming population reached approximately 3.32 billion players in 2024 and was projected to grow to around 3.5–3.6 billion gamers by 2025, underscoring the continued expansion of video games as a dominant form of digital entertainment.

The future of the video game industry looks incredibly promising, driven by several key trends and technological advancements:

Virtual Reality (VR) and Augmented Reality (AR): These technologies are expected to become more mainstream, offering immersive gaming experiences. The VR and AR market is projected to reach $370 billion by 2034.

Cloud Gaming: Services like Google Stadia and NVIDIA GeForce Now are making high-quality gaming accessible without the need for powerful hardware. This trend is likely to continue, with streaming becoming a major part of the gaming landscape.

Artificial Intelligence (AI): AI will play a significant role in creating more realistic and responsive game environments. This includes advanced NPC behaviors and personalized gaming experiences.

Metaverse: The concept of the metaverse, a shared virtual space where users can interact in real time, is gaining traction. Companies are investing heavily in creating these expansive virtual worlds.

Mobile Gaming: Mobile gaming remains the largest segment of the video game industry, generating approximately US$92–93 billion in revenue in 2024, accounting for nearly half of total global game revenues. Industry forecasts indicated continued growth, with mobile game revenues projected to exceed US$100 billion in 2025, driven by widespread smartphone adoption, free-to-play business models, and global accessibility. The convenience and low barrier to entry of mobile platforms position mobile gaming as a central driver of the industry's long-term growth.

E-Sports: Competitive gaming is growing rapidly, with large audiences and significant investment. E-sports tournaments are becoming mainstream entertainment events.

The video game industry is set to continue its expansion, driven by innovation and the increasing integration of gaming into everyday life.

https://doi.org/10.15215/remix/9781998944224.01

1.3. Types of Video Games

Since we will be learning how to create video games in this course, it is necessary to know what types of games we will learn to program.

1.3.1. Action Games

Action games are a genre that emphasizes physical challenges, such as hand-eye coordination and reaction time. These games often involve combat, exploration, and overcoming obstacles in dynamic and interactive environments.

Key Features of Action Games

Fast-Paced Gameplay: Action games often involve quick reflexes and fast decision-making. Players must react swiftly to in-game events to succeed.

Combat and Fighting: Many action games focus on combat, whether it's hand-to-hand fighting, shooting, or using various weapons. Examples include *Street Fighter* and *Call of Duty*.

Exploration and Adventure: Action games often feature expansive worlds to explore, filled with enemies, obstacles, and hidden secrets. Games like *The Legend of Zelda* series blend action with adventure elements.

Platforming: Platformers are a subgenre where players navigate through levels by jumping between platforms and avoiding obstacles. Classic examples include *Super Mario Bros.* and *Sonic the Hedgehog*.

Puzzle-Solving: Some action games incorporate puzzles that require players to think strategically and solve problems to progress. The *Tomb Raider* and *Uncharted* series are known for this blend.

Popular Types of Action Games

First-Person Shooters (FPS): Players experience the game through the eyes of the protagonist, focusing on shooting and combat. Examples—*Call of Duty*, *Halo*.

Third-Person Shooters: Players control a character from a third-person perspective, often with a focus on shooting and cover mechanics. Examples—*Gears of War*, *The Division*.

Beat 'Em Ups: Players fight through waves of enemies using melee combat. Examples—*Streets of Rage*, *Double Dragon*.

Hack and Slash: Focus on melee combat with swords or other weapons, often featuring combo systems. Examples—*Devil May Cry*, *God of War*.

https://doi.org/10.15215/remix/9781998944224.01

Stealth Action: Emphasize sneaking and avoiding detection rather than direct combat. Examples—*Metal Gear Solid*, *Hitman*.

Popular Action Games

***Elden Ring*:** Known for its challenging combat and expansive open world, blending action with RPG (role-playing game) elements.

***The Legend of Zelda: Breath of the Wild*:** Combines action, exploration, and puzzle-solving in a vast open world.

***Grand Theft Auto V*:** Offers a mix of action, driving, and open-world exploration with a compelling storyline.

***Doom Eternal*:** A fast-paced FPS with intense combat and a focus on movement and strategy.

1.3.2. Adventure Games

Adventure games are a genre where players take on the role of a protagonist in an interactive story, driven by exploration and puzzle-solving. These games often emphasize narrative and character development, allowing players to immerse themselves in a rich, engaging storyline.

Key Features of Adventure Games

Story-Driven Gameplay: Adventure games often feature deep, engaging narratives that drive the gameplay. Players progress through the story by solving puzzles and interacting with the game world.

Exploration: Players explore diverse environments, from fantasy realms to real-world locations. The thrill of discovery is a significant part of the experience.

Puzzle-Solving: Puzzles are a core element, requiring players to think critically and use logic to progress. These can range from simple tasks to complex, multistep challenges.

Character Interaction: Players interact with various characters, each with their own stories and personalities. Dialogue choices can influence the story and outcomes.

Inventory Management: Many adventure games involve collecting and using items to solve puzzles and advance the plot.

Popular Types of Adventure Games

Point-and-Click Adventures: Players use a mouse to interact with the environment and solve puzzles. Examples—*Monkey Island* series, *Grim Fandango*.

https://doi.org/10.15215/remix/9781998944224.01

Action-Adventure: Combines elements of action games with adventure gameplay. Examples—*The Legend of Zelda* series, *Tomb Raider* series.

Text Adventures: Players input text commands to interact with the game world. Examples—*Zork*, *Hitchhiker's Guide to the Galaxy*.

Visual Novels: Focus on narrative and character development, often with branching storylines based on player choices. Examples—*Phoenix Wright—Ace Attorney*, *Danganronpa*.

Interactive Fiction: Similar to text adventures but often more focused on storytelling and player choices. Examples—*80 Days*, *Lifeline*.

Popular Adventure Games

***The Legend of Zelda: Breath of the Wild*:** An open-world action-adventure game known for its vast exploration, puzzle-solving, and engaging story.

***The Witcher 3: Wild Hunt*:** Combines action, exploration, and narrative-driven gameplay in a richly detailed fantasy world.

***Life Is Strange*:** An episodic adventure game that focuses on narrative and player choices, with a unique time-rewinding mechanic.

***Uncharted* Series:** Action-adventure games that blend cinematic storytelling with exploration and puzzle-solving.

***Grim Fandango*:** A classic point-and-click adventure game with a unique art style and engaging story.

1.3.3. Arcade Games

Arcade games are coin-operated entertainment machines typically found in public places like restaurants, bars, and amusement arcades. These games are designed to be easy to play but challenging to master, often focusing on skill-based gameplay.

Key Features of Arcade Games

Fast-Paced Gameplay: Arcade games are designed to be quick and engaging, often requiring fast reflexes and quick decision-making.

Simple Controls: The controls are usually straightforward, making the games easy to pick up and play but challenging to master.

Increasing Difficulty: As players progress, the difficulty level increases, providing a continuous challenge.

High Scores: Many arcade games feature a high-score system, encouraging players to compete for the top spot.

https://doi.org/10.15215/remix/9781998944224.01

Short Play Sessions: Arcade games are typically designed for short play sessions, making them ideal for quick entertainment.

Popular Types of Arcade Games

Classic Arcade Games: These include iconic titles from the golden age of arcade gaming, such as *Pac-Man*, *Space Invaders*, and *Asteroids*.

Platformers: Players navigate characters through levels filled with obstacles and enemies. Examples—*Super Mario Bros.*, *Donkey Kong*.

Shoot 'Em Ups: Players control a character or vehicle and shoot at waves of enemies. Examples—*Galaga*, *Gradius*.

Beat 'Em Ups: Players fight through levels filled with enemies using hand-to-hand combat. Examples—*Double Dragon*, *Streets of Rage*.

Puzzle Games: These games challenge players with puzzles that require logic and strategy to solve. Examples—*Tetris*, *Puzzle Bobble*.

Popular Arcade Games

***Pac-Man*:** A classic maze game where players control Pac-Man, eating pellets and avoiding ghosts.

***Space Invaders*:** A pioneering shoot 'em up game where players defend against waves of descending aliens.

***Street Fighter II*:** A revolutionary fighting game that popularized the one-on-one fighting genre.

***Galaga*:** A shoot 'em up game where players control a spaceship and shoot at enemy formations.

***Donkey Kong*:** A platformer where players control Mario, climbing ladders and avoiding obstacles to rescue a damsel in distress.

***Geometry Dash*:** A rhythm-based platformer with challenging levels and fast-paced gameplay.

***Crossy Road*:** A modern take on the classic *Frogger*, where players navigate characters across busy roads and rivers.

***Fruit Ninja*:** A game where players swipe to slice fruits while avoiding bombs.

1.3.4. Board Games

Board games are traditionally defined as tabletop games in which players move pieces or tokens on a premarked surface, or *board*, according to a formal set of rules. These games vary widely in complexity, theme, and mechanics, ranging from simple abstract contests to elaborate strategy-driven systems. Board games are especially valued for fostering social interaction, communication, and shared experiences among players.

https://doi.org/10.15215/remix/9781998944224.01

Historically, board games represent one of the oldest forms of structured play. Some of the earliest known examples date back thousands of years, such as Senet from ancient Egypt and Go from China. Despite their physical origins, board games are best understood as *systems of rules and mechanics*, rather than as artifacts tied exclusively to physical components.

Importantly, the core mechanics of board games—turn-based play, discrete state transitions, resource management, probability, and strategic decision-making—lend themselves naturally to digital implementation. As a result, many board games have been successfully adapted into video games, and some modern board games are designed from the outset to integrate digital components. In these cases, video technology may replace tasks traditionally handled by human players, such as rule enforcement, state tracking, randomness generation, or narrative control. This demonstrates that board games are not fundamentally incompatible with video games; rather, they form a genre whose mechanics can be realized in either physical or digital form.

Examples of this convergence include fully digital adaptations of classic board games such as chess, Monopoly, Risk, Catan, and Ticket to Ride, as well as hybrid "video board games" like Mansions of Madness (Second Edition) and XCOM: The Board Game, which combine physical boards with companion apps that manage scenarios, timing, and game events. These examples illustrate that board game designs can function equivalently—or even more effectively—within a video game environment.

Key Features of Board Games

Social Interaction: Board games are often played with friends or family, fostering social interaction and teamwork.

Strategy and Skill: Many board games require strategic thinking, planning, and skill to win.

Well-Defined Rules: Board games typically operate under explicit and deterministic rule systems, making them suitable for algorithmic implementation.

Variety of Themes: Board games cover a wide range of themes, from fantasy and adventure to economics and history.

Replayability: Good board games offer high replay value, with different outcomes and strategies each time they are played.

Popular Types of Board Games

Classic Board Games: These are timeless games that have been enjoyed for generations. Examples—chess, checkers, backgammon, Monopoly, Scrabble.

https://doi.org/10.15215/remix/9781998944224.01

Strategy Games: These games require careful planning and strategy to outmaneuver opponents. Examples—Catan, Risk, Ticket to Ride.

Cooperative Games: Players work together to achieve a common goal. Examples—Pandemic, Forbidden Island, Gloomhaven.

Party Games: Designed for larger groups, these games are often lighthearted and focus on fun and interaction. Examples—Codenames, Cards Against Humanity, Pictionary.

Deck-Building Games: Players build their own decks of cards throughout the game to gain advantages. Examples—Dominion, Ascension, Clank!

Popular Board Games

Catan: A strategy game where players collect resources and build settlements to earn points.

Pandemic: A cooperative game where players work together to stop global outbreaks of diseases.

Ticket to Ride: A route-building strategy game in which players connect cities on a map to complete objectives.

Gloomhaven: A cooperative, campaign-based game with deep storytelling and tactical combat.

Monopoly: A classic game where players buy, trade, and develop properties to bankrupt their opponents.

Wingspan: A strategy game about bird collecting, featuring beautiful artwork and engaging gameplay.

Azul: A tile-placement game where players compete to create the most beautiful patterns.

Terraforming Mars: A strategy game where players work to terraform the planet Mars by managing resources and projects.

1.3.5. Puzzle Games

Puzzle games are a genre that challenges players' problem-solving skills, logic, and mental acuity. They come in various forms, each offering unique gameplay experiences.

Key Features of Puzzle Games

Problem-Solving: Puzzle games require players to solve problems, often through logic, pattern recognition, or strategic thinking.

Variety of Challenges: These games can include a wide range of challenges, such as jigsaw puzzles, word puzzles, number puzzles, and logic puzzles.

https://doi.org/10.15215/remix/9781998944224.01

Increasing Difficulty: Many puzzle games feature levels that increase in difficulty, providing a continuous challenge as players progress.

Casual and Relaxing: Puzzle games are often designed to be casual and relaxing, making them accessible to players of all ages and skill levels.

Popular Types of Puzzle Games

Jigsaw Puzzles: Players assemble pieces to form a complete picture. Examples—*Jigsaw Planet*.

Word Puzzles: Games that involve forming words from letters or solving crossword puzzles. Examples—*Words of Wonders*, *Word Wipe*.

Number Puzzles: Games that involve numbers. Examples—*Sudoku*, *2048*.

Logic Puzzles: Games that require logical thinking to solve. Examples—*Minesweeper*, *Flow Free*.

Match-Three Games: Players match three or more items of the same type to clear them from the board. Examples—*Candy Crush Saga*, *Bejeweled*.

Popular Puzzle Games

***Tetris*:** A classic puzzle game where players arrange falling blocks to complete lines.

***Portal*:** A first-person puzzle game that involves creating portals to solve spatial puzzles.

***The Witness*:** An open-world puzzle game with a variety of challenging puzzles to solve.

***Monument Valley*:** A visually stunning puzzle game that involves manipulating impossible architecture to guide a character through levels.

***Baba Is You*:** A unique puzzle game where players manipulate the rules of the game to solve puzzles.

1.3.6. Role-Playing Games

Role-playing games (RPGs) are a genre where players assume the roles of characters in a fictional setting. Players take control of these characters and guide them through various quests, battles, and storylines.

https://doi.org/10.15215/remix/9781998944224.01

Key Features of RPGs

Character Development: Players can customize their characters' appearance, abilities, and skills. Characters gain experience points (XP) and level up, improving their stats and unlocking new abilities.

Storylines and Quests: RPGs often feature rich, immersive storylines with multiple quests and side missions. Players make choices that can affect the outcome of the story and the world around them.

Exploration: RPGs typically offer expansive worlds to explore, filled with towns, dungeons, and hidden secrets. Exploration is often rewarded with loot, new quests, and lore.

Combat Systems: Combat can be turn-based, real time, or a hybrid of both. Players use a variety of weapons, spells, and tactics to defeat enemies.

Inventory and Equipment: Managing inventory and equipping characters with the best gear is a crucial part of RPGs. Players can find, buy, or craft weapons, armor, and items.

Popular Types of RPGs

Action RPGs: Focus on real-time combat and fast-paced gameplay. Examples of action RPGs include *The Witcher 3: Wild Hunt*, *Dark Souls*.

Turn-Based RPGs: Combat is turn-based, allowing players to plan their moves strategically. Examples of turn-based RPGs include the *Final Fantasy* series, the *Persona* series.

MMORPGs (Massively Multiplayer Online RPGs): Feature large, persistent worlds where thousands of players can interact. Examples of MMORPGs include *World of Warcraft*, *Final Fantasy XIV*.

JRPGs (Japanese RPGs): Often feature stylized art, turn-based combat, and linear storylines. Examples of JRPGs (Japanese RPGs) include *Dragon Quest* series, *Tales of* series.

Genre RPGs: Emphasize open-world exploration and player choice in a themed setting. Examples include *Red Dead Redemption* in a western setting, *The Elder Scrolls* series in the fantasy realm of Tamriel, and the *Fallout* series set in a post-nuclear-apocalypse America.

Popular RPG Games

***Elden Ring*:** Known for its challenging combat and expansive open world.

https://doi.org/10.15215/remix/9781998944224.01

***The Witcher 3: Wild Hunt*:** Praised for its storytelling and detailed world.

***Final Fantasy VII Remake*:** A modern take on a classic JRPG.

1.3.7. Sports Games

Sports games are a genre that simulates the practice of sports. These games can replicate real-life sports or incorporate elements of sports into their gameplay. They often focus on physical and tactical challenges, testing the player's precision, accuracy, and strategic thinking.

Key Features of Sports Games

Realism: Many sports games strive to accurately model the athletic characteristics required by the sport, such as speed, strength, and agility.

Variety: They cover a wide range of sports, including football, basketball, soccer, baseball, and more.

Multiplayer Options: Players can compete against computer-controlled opponents or other players, either locally or online.

Career Modes: Some games offer career modes, where players can manage a team or develop a single athlete's career over multiple seasons.

Popular Types of Sports Games

Sports games come in many forms, each emphasizing different mechanics and styles of play. These are some of the most widely recognized types:

Team Sports Simulations: These games replicate real-world team sports with a focus on authenticity, licensed teams, and realistic physics. Examples include soccer (e.g., *FIFA*, *eFootball*), basketball (e.g., *NBA 2K*), American football (e.g., *Madden NFL*), hockey (e.g., *NHL* series).

Individual Sports Games: These focus on one-on-one or solo athletic performance. Examples include tennis (*Top Spin*, *Tennis World Tour*), golf (*PGA Tour 2K*, *Everybody's Golf*), boxing/MMA (*Fight Night*, *UFC*).

Racing and Motorsports: These involve cars, motorcycles, karts, or futuristic vehicles. Examples include simulation racing (*Gran Turismo*, *Forza Motorsport*), arcade racing (*Mario Kart*, *Burnout*), rally and off-road (*Dirt*, *WRC*).

https://doi.org/10.15215/remix/9781998944224.01

Extreme and Action Sports: Games that highlight high-energy, trick-based gameplay. Examples include skateboarding (*Tony Hawk's Pro Skater*), snowboarding (*SSX*), BMX and freestyle sports (*Mat Hoffman's Pro BMX*).

Sports Management and Strategy Games: These emphasize tactics, recruitment, finances, and long-term planning rather than real-time play. Examples include *Football Manager*, *Out of the Park Baseball*, *Motorsport Manager*.

Popular Sports Games

***FIFA* Series:** Developed by EA Sports, this series is known for its realistic soccer simulation, featuring licensed teams, players, and leagues.

***NBA 2K* Series:** A basketball simulation game that offers realistic gameplay, player animations, and a deep career mode.

***Madden NFL* Series:** Another EA Sports title, this series focuses on American football, offering realistic gameplay and various modes, including Franchise and Ultimate Team.

***Tony Hawk's Pro Skater*:** A classic skateboarding game known for its arcade-style gameplay and iconic soundtrack.

***Rocket League*:** A unique blend of soccer and vehicular acrobatics, where players control rocket-powered cars to hit a ball into the opponent's goal.

1.3.8. Strategy Games

Strategy games are a genre of games that emphasizes planning, decision-making, and resource management to achieve victory. These games often require players to think critically and strategically, making choices that will impact the outcome of the game.

Key Features of Strategy Games

Resource Management: Players must gather and manage resources such as money, materials, and manpower to build and sustain their forces or civilizations.

Tactical and Strategic Planning: Players need to devise and execute plans to outmaneuver opponents, whether in battle or through economic and diplomatic means.

Turn-Based vs. Real-Time: In turn-based strategy (TBS) games, players take turns to make their moves. In real-time strategy (RTS) games, gameplay occurs in real time, requiring quick decision-making.

https://doi.org/10.15215/remix/9781998944224.01

Single-Player and Multiplayer Modes: In single-player modes, campaigns and scenarios are played against AI opponents. Multiplayer modes involve competitive or cooperative play against other human players.

Popular Types of Strategy Games

Real-Time Strategy (RTS): Players manage resources, build units, and control armies in real time. Examples include *StarCraft* and *Age of Empires*.

Turn-Based Strategy (TBS): Players take turns making decisions and moves. Examples include *Civilization* and *XCOM*.

4X (eXplore, eXpand, eXploit, eXterminate) Strategy: These involve exploring, expanding, exploiting, and exterminating. Examples include *Sid Meier's Civilization* and *Stellaris*.

Grand Strategy: Focuses on managing entire nations or empires over long periods. Examples include *Crusader Kings* and *Europa Universalis*.

Tower Defense Strategy: In such games, players build defensive structures to protect against waves of enemies. Examples include *Plants vs. Zombies*, *Kingdom Rush*.

Popular Strategy Games

These are some influential and widely played strategy titles across different subgenres:

Real-Time Strategy (RTS)

***StarCraft II*:** Known for its competitive multiplayer and asymmetric factions.

***Age of Empires IV*:** A modern evolution of classic empire-building gameplay.

***Warcraft III*:** A foundational RTS with strong hero-based mechanics.

Turn-Based Strategy (TBS)

***Civilization VI*:** Players build and expand civilizations across millennia.

***XCOM 2*:** A tactical combat strategy game with high-stakes decisions.

***Advance Wars*:** A turn-based military strategy series known for accessibility and depth.

https://doi.org/10.15215/remix/9781998944224.01

4X (eXplore, eXpand, eXploit, eXterminate) Strategy

***Stellaris*:** A space-faring empire builder with dynamic narratives.

***Sid Meier's Civilization Series*:** The gold standard for 4X historical strategy.

***Endless Legend*:** A fantasy-themed 4X with asymmetric factions.

Grand Strategy

***Crusader Kings III*:** Focuses on dynasties, politics, and medieval intrigue.

***Europa Universalis IV*:** Deep global strategy spanning centuries.

***Hearts of Iron IV*:** A WWII-focused grand strategy emphasizing logistics and warfare.

Tower Defense

***Plants vs. Zombies*:** Accessible and humorous tower defense gameplay.

***Kingdom Rush*:** Known for its polished mechanics and challenging levels.

***Bloons TD* Series:** A long-running, highly popular tower defense franchise.

1.3.9. Utilities Games

Utilities games are a unique category of software available on platforms like Steam that provide functional tools or applications rather than traditional gameplay experiences. These "games" often serve practical purposes, such as creating art, managing tasks, or enhancing the user experience on a computer.

Key Features of Utilities Games

Productivity Enhancement: Utilities games often include features that help users stay focused and organized. For example, *Virtual Cottage* provides a cozy virtual environment to set goals and work without distractions.

Creative Tools: Some utilities games are designed to foster creativity. *Aseprite* is a pixel-art tool that allows users to create 2D animations and sprites for games.

System Optimization: Utilities games can also help optimize system performance. *CPUCores* is a tool that isolates and dedicates central processing unit (CPU) resources to improve gaming performance.

https://doi.org/10.15215/remix/9781998944224.01

Customization: Many utilities games offer extensive customization options. *Wallpaper Engine* allows users to create and animate live wallpapers for their desktops.

Popular Types of Utilities Games

Creativity and Design Tools: These "games" function as creative suites for artists, designers, and content creators. Examples include *Aseprite* (pixel-art and sprite-animation tool), *Blender* (Steam distribution; 3D modeling, animation, and rendering), and *RPG Maker Tools* (create sprites, maps, and game assets).

System Enhancement and Personalization Tools: Designed to customize or optimize the user's computer experience. Examples include *Wallpaper Engine* (animated desktop backgrounds), *CPUCores* (CPU optimization for gaming performance), *Rainmeter Skins* (via Steam tools; desktop information and visualization widgets).

Productivity and Focus Tools: Applications that create calming or structured environments to support work or study. Examples include *Virtual Cottage* (a relaxing space to help users stay focused), *Lofi Room / Ambient Focus Tools* (provide sounds and visual ambience), and other task organizers distributed through Steam utility categories.

Simulation-Based Utilities: Tools that provide utility through simulated or controlled environments. Examples include *PC Building Simulator* (both a game and learning tool for PC hardware assembly) and other logic/puzzle toolkits used for teaching circuits and computing concepts.

Popular Utilities Games

***Wallpaper Engine*:** Allows users to create and use animated wallpapers on their desktops.

***Aseprite*:** A pixel-art tool for creating 2D animations and sprites.

***Blender*:** A comprehensive 3D creation suite used for modeling, animation, and rendering.

***Virtual Cottage*:** Provides a relaxing virtual environment to help users focus and be productive.

These utilities can be incredibly useful for creative projects, productivity, and personalizing your digital workspace.

https://doi.org/10.15215/remix/9781998944224.01

1.3.10. Game Demos

Game demos are trial versions of video games that allow players to experience a portion of the game before deciding to purchase the full version. They are a great way to try out new titles and get a feel for the gameplay, graphics, and overall experience.

Key Features of Game Demos

Limited Content: Demos typically include a small portion of the game, such as the first few levels, a specific mission, or a limited-time trial.

Free to Play: Most game demos are free to download and play, providing a risk-free way to try out new games.

Showcase Gameplay: Demos are designed to highlight the core mechanics, graphics, and features of the game to entice players to purchase the full version.

Feedback Opportunity: Developers can use demos to gather feedback from players, which can be valuable for making improvements before the full release.

Game demos are often distributed as shareware, demo discs, or downloadable software.

Popular Platforms for Game Demos

Steam: Steam offers a wide variety of free game demos that you can download and play. It's a great platform to explore new titles and try before you buy.

Xbox: The Microsoft Store on Xbox provides numerous game demos for players to try out. You can find demos for both new and upcoming games.

Epic Games Store: The Epic Games Store also features a selection of game demos, allowing players to experience a portion of the game before making a purchase.

Examples of Game Demos

***The First Berserker—Khazan* Demo:** A demo for an action-adventure game where players can experience the initial gameplay and story.

***SWORD ART ONLINE Fractured Daydream* Demo:** A demo for a role-playing game based on the popular anime series.

https://doi.org/10.15215/remix/9781998944224.01

***PowerWash Simulator* Demo:** A demo for a simulation game where players can try out the power washing mechanics and complete a few cleaning tasks.

Benefits of Playing Game Demos

Informed Decisions: Trying a demo helps you make an informed decision about whether to purchase the full game.

Discover New Games: Demos allow you to explore new genres and titles that you might not have considered otherwise.

Early Access: Some demos provide early access to upcoming games, giving you a sneak peek before the official release.

1.3.11. Emulators and Video Games

Emulated games are video games run on a platform other than the one they were originally designed for, using software called an emulator. An emulator lets a host device such as a PC, smartphone, or other modern platform mimic the behavior of a different, often older, gaming system so it can run software and games developed for that original system. Emulation is especially useful for preserving and playing retro games on contemporary hardware.

Emulators are also widely used for software development and testing, allowing developers to run and debug applications on multiple platforms without requiring the actual hardware. For example, developers commonly write Android apps on a PC and use an Android emulator to verify how the app functions on a simulated phone.

Key Features of Emulators

Compatibility: Emulators can run games from various classic gaming systems, including consoles like the NES, SNES, Sega Genesis, Game Boy, PlayStation, and more.

Preservation: Emulators help preserve classic games that might otherwise be lost because of outdated hardware. They allow new generations of players to experience these games.

Customization: Many emulators offer features like save states, enhanced graphics, and controller support, which can improve the gaming experience.

Accessibility: Emulators make it possible to play games on devices that were not originally designed for them, such as playing a Game Boy game on a PC.

https://doi.org/10.15215/remix/9781998944224.01

Functionality: Emulators replicate the hardware and software environment of the target system, allowing games and applications to run as if they were on the original hardware.

Types: There are emulators for various platforms, including classic consoles like the NES, SNES, Sega Genesis, PlayStation, and more.

Usage: They are popular among retro gaming enthusiasts who want to play old games that are no longer available or supported on modern hardware.

Popular Types of Emulators

RetroArch: A versatile emulator that supports a wide range of gaming systems through "cores" that can be downloaded and installed.

Dolphin: A popular emulator for Nintendo GameCube and Wii games, known for its high compatibility and performance.

PCSX2: An emulator for PlayStation 2 games, offering a range of features to enhance the gaming experience.

PPSSPP: An emulator for PlayStation Portable (PSP) games, known for its high compatibility and ability to run games at higher resolutions.

Cemu: An emulator for the Nintendo Wii U, allowing players to enjoy Wii U games on their PCs.

Popular Emulator Games

***Super Mario 64*:** Originally for the Nintendo 64, this classic platformer can be played on various emulators.

***The Legend of Zelda: Ocarina of Time*:** Another Nintendo 64 classic that is widely enjoyed through emulation.

***Pokémon FireRed*:** A Game Boy Advance game that remains popular among emulator users.

***Final Fantasy VII*:** Originally for the PlayStation, this iconic RPG is often played on emulators.

***Metroid Prime*:** A GameCube game that is frequently emulated using Dolphin.

Legal Considerations

While emulators themselves are legal, downloading and using game ROMs (the game files) can be legally complex. It's important to only use ROMs (or the software image file) for games that you own or that are legally available for download. Always respect copyright laws and the rights of game developers.

https://doi.org/10.15215/remix/9781998944224.01

1.4. Road Map of Game Development

1.4.1. Process of Game Development

In section 1.3, we explored different game genres and their unique characteristics. This section is to help you understand the game development process, including concept ideation, design, implementation, and testing.

Game development is a fascinating and complex process that involves several stages. Here's a high-level overview of the key steps:

1. **Concept and Planning**
   Idea Generation: Brainstorming and coming up with the game's concept
   Market Research: Understanding the target audience and market trends
   Game Design Document (GDD): Creating a detailed document outlining the game's mechanics, story, characters, and more
2. **Pre-Production**
   Prototyping: Developing a basic version of the game to test concepts and gameplay mechanics
   Technical Planning: Deciding on the technology stack, tools, and platforms to be used
   Team Assembly: Gathering a team of developers, artists, designers, and other necessary roles
3. **Production**
   Art and Design: Creating the visual elements, including characters, environments, and user interfaces
   Programming: Writing the code that makes the game function
   Audio Design: Developing sound effects, music, and voice acting
   Level Design: Building the game's levels and ensuring they are engaging and balanced
4. **Testing**
   Quality Assurance (QA): Identifying and fixing bugs and glitches
   Playtesting: Gathering feedback from players to improve gameplay and balance
5. **Launch**
   Marketing and Promotion: Creating buzz and awareness about the game
   Distribution: Releasing the game on various platforms (e.g., Steam, consoles, mobile)
6. **Postlaunch**
   Updates and Patches: Fixing any issues that arise after launch and adding new content

https://doi.org/10.15215/remix/9781998944224.01

Community Engagement: Interacting with players and building a community around the game

Each of these stages requires collaboration and creativity, and the process can vary depending on the size of the team and the scope of the game. Are you interested in any specific part of game development?

1.4.2. Programming Languages and Tools

There are several popular programming languages used in game development, each with its own strengths and applications. Here are some of the most commonly used ones:

C++: Known for its performance and control over system resources, C++ is widely used in game development, especially for high-performance games. It's used in many game engines like Unreal Engine.

C#: This language is popular because of its use in the Unity game engine, which is one of the most widely used game development platforms. C# is known for being relatively easy to learn and use.

Java: Java is versatile and can be used for both desktop and mobile game development. It's particularly popular for Android game development.

Python: While not as common for high-performance games, Python is great for beginners and is used in some game development frameworks like pygame.

JavaScript: Essential for web-based games, JavaScript works well with HTML5 and CSS3 to create interactive games that run within browsers.

Lua: Often used as a scripting language in game engines like Unity, Corona SDK, and Roblox, Lua is lightweight and easy to embed.

Each of these languages has its own community and resources, making it easier to find tutorials and support as you learn.

To be an effective game developer, one also needs to be familiar with integrated development environments (IDEs) and game engines.

There are several popular IDEs and game engines that developers use to create games. Here are some of the commonly used ones.

https://doi.org/10.15215/remix/9781998944224.01

Game Engines

Unity: One of the most popular game engines, Unity is known for its versatility and ease of use. It supports both 2D and 3D game development and is widely used for mobile, PC, and console games.

Unreal Engine: Renowned for its high-quality graphics and realistic physics, Unreal Engine is a favorite for AAA game development. It features a visual scripting system called Blueprints, which allows developers to create game logic without extensive coding.

Godot Engine: An open-source game engine that is gaining popularity, especially among indie developers. Godot offers a powerful 2D and 3D rendering engine and is known for its flexibility and ease of use.

GameMaker Studio: Ideal for 2D game development, GameMaker Studio is user-friendly and allows developers to create games with minimal coding. It's popular for indie and mobile games.

IDEs

Visual Studio: A powerful IDE used by many professional game developers. It supports multiple programming languages and integrates well with popular game engines like Unity and Unreal Engine.

JetBrains Rider: Known for its robust code analysis and debugging tools, Rider supports multiple languages and integrates with Unity and Unreal Engine.

MonoDevelop: An open-source IDE that is simple and easy to use. It supports multiple languages and integrates with Unity and Godot Engine.

These tools provide a range of features that help streamline the game development process, from coding and debugging to designing and testing.

1.4.3. Game Design Principles

Game design principles are essential guidelines that help create engaging and enjoyable gaming experiences. Here are some of the key ones:

Clear Goals and Objectives: Players should always know what they need to achieve. Clear goals provide direction and motivation, keeping players engaged.

https://doi.org/10.15215/remix/9781998944224.01

Engaging Core Mechanics: The core mechanics are the fundamental actions players perform in the game. These should be intuitive and enjoyable, forming the backbone of the gameplay experience.

Balanced Challenge and Skill: Games should balance difficulty to match the player's skill level. This keeps the game challenging but not frustrating, maintaining a state of "flow," where players are fully immersed.

Feedback and Rewards: Providing immediate feedback and rewards for player actions helps reinforce learning and keeps players motivated. This can include points, achievements, or visual and audio cues.

Immersive Storytelling: A compelling narrative can enhance the gaming experience by providing context and emotional engagement. Good storytelling can make players care about the characters and the world.

Player Agency: Allowing players to make meaningful choices that affect the game world and outcomes increases engagement and investment in the game.

Aesthetic and Audio Design: Visual and audio elements should complement the gameplay and enhance the overall experience. Good design can create an immersive atmosphere and evoke emotions.

These principles help ensure that a game not only is fun to play but also provides a satisfying and memorable experience.

1.4.4. Graphics and Animation in Video Games

Graphics and animation are crucial components of games; they enhance the visual appeal and immersive experience of games. These elements work together to create visually stunning and more interactive gaming experiences.

Graphics in Game Development

2D and 3D Graphics: Games can feature either 2D or 3D graphics. 2D graphics are often used in simpler, indie games, while 3D graphics are common in more complex, high-budget games. Tools like Adobe Photoshop and GIMP are popular for creating 2D assets, while Blender and Maya are widely used for 3D modeling.

Rendering: This is the process of generating the final visual output from the game's assets. Real-time rendering is crucial for games,

https://doi.org/10.15215/remix/9781998944224.01

allowing for dynamic and interactive environments. DirectX and OpenGL are common APIs used for rendering.

Shaders: Shaders are programs that run on the graphics processing unit (GPU) to control the rendering of graphics. They can create various effects like lighting, shadows, and textures, enhancing the visual quality of the game.

Animation in Game Development

Big or small, most video games have one feature in common—that is, animated graphics and figures. To use animations properly and effectively, one needs to carefully study the following:

Principles of Animation: Game animation commonly applies multiple classical animation principles, including squash and stretch, anticipation, and timing, to enhance realism and player engagement.

Rigging and Skinning: Rigging involves creating a skeleton for 3D models, while skinning attaches the model to the skeleton. This allows for realistic movement and deformation of characters.

Animation Techniques: Techniques like keyframe animation, motion capture, and procedural animation are used to create character movements. Keyframe animation involves manually setting key points of motion, while motion capture records real-life movements for more realistic animations.

Animation Systems: Modern game engines like Unity and Unreal Engine have built-in animation systems that support complex animations, including blend trees and inverse kinematics.

One may also use some specific tools to produce 2D or 3D animation, such as the following:

Unity: Offers robust tools for both 2D and 3D animation, including a powerful Animator component.

Unreal Engine: Known for its high-quality graphics and advanced animation tools, including the use of Blueprints for visual scripting.

Blender: An open-source tool for 3D modeling and animation, widely used in the industry.

https://doi.org/10.15215/remix/9781998944224.01

1.4.5. Physics and Simulation

Physics and simulation play a crucial role in creating realistic and immersive experiences in video games.

Physics in Game Development

Rigid Body Dynamics: This involves simulating the motion and interaction of solid objects. It's essential for creating realistic collisions and movements. Physics engines like Havok and PhysX are commonly used for these simulations.

Soft Body Dynamics: Unlike rigid bodies, soft bodies can deform and change shape. This is used for simulating objects like cloth, jelly, or even characters' skin.

Fluid Dynamics: Simulating liquids and gases can add a lot of realism to games. This includes water, smoke, fire, and other fluid-like behaviors. Advanced techniques and powerful hardware are often required for these simulations.

Ragdoll Physics: This technique is used to simulate the natural movement of characters' limbs when they are knocked down or killed. It enhances realism by allowing characters to react dynamically to forces.

Simulation in Game Development

Environmental Simulation: This includes weather systems, day-night cycles, and other environmental effects that make the game world feel alive and dynamic.

AI Simulation: Simulating intelligent behavior in nonplayer characters (NPCs) is crucial for creating engaging and challenging gameplay. This involves pathfinding, decision-making, and learning algorithms.

Vehicle Simulation: For games involving vehicles, realistic simulation of driving physics—including traction, suspension, and collision—is essential.

1.4.6. User Interface Design

User Interface (UI) design is a critical aspect of game development, as it directly impacts the player's experience and interaction with the game. It's not uncommon to read game comments or reviews about how much better a game would be if it weren't for the badly designed UI. Here are some key principles and tools involved in UI design for games.

https://doi.org/10.15215/remix/9781998944224.01

Key Principles of UI Design

Clarity: UI elements should be clear and easy to understand. Players should be able to quickly grasp what each element does without confusion.

Consistency: Consistent design elements help players learn and predict how the UI will behave. This includes consistent use of colours, fonts, and layouts.

Simplicity: A simple and uncluttered UI reduces cognitive load, making it easier for players to focus on the game itself.

Functionality: The UI should be designed with the game's mechanics in mind, ensuring that it supports and enhances gameplay rather than hindering it.

Feedback: Providing immediate feedback for player actions helps reinforce learning and keeps players informed about the game's state.

Tools for UI Design

Unity: Unity offers robust tools for creating UI elements, including the Unity UI system and the newer UI Toolkit. These tools allow for the creation of complex, scalable, and performant interfaces.

Unreal Engine: Unreal Engine's UMG (Unreal Motion Graphics) provides a powerful framework for designing and implementing UI. It supports both visual scripting and traditional coding.

Adobe XD: This tool is excellent for wireframing and prototyping UI designs. It allows designers to create interactive mock-ups and test navigation flows before implementation.

Sketch: Another popular tool for UI design, Sketch is known for its ease of use and powerful vector-editing capabilities. It's widely used for designing game interfaces.

Key Components in Game UI

Health Bars and HUDs (Heads-Up Displays): These elements provide essential information about the player's status and game environment.

Menus and Navigation: Intuitive menus and navigation systems help players access different parts of the game easily.

Interactive Elements: Buttons, sliders, and other interactive elements should be responsive and provide clear feedback.

https://doi.org/10.15215/remix/9781998944224.01

Effective UI design enhances the overall gaming experience by making it more intuitive and enjoyable.

1.4.7. Sound Effects with Sound and Music Integration

Sound and music integration are vital for creating immersive and engaging experiences in video games.

Key Aspects of Sound and Music Integration

Setting the Mood: Music sets the tone and atmosphere of the game, enhancing the emotional experience. For example, a suspenseful soundtrack can heighten tension during critical moments.

Enhancing Gameplay: Sound effects provide immediate feedback for player actions, making the game more interactive and responsive. This includes sounds for actions like jumping, shooting, or collecting items.

Creating Immersive Environments: Ambient sounds and background music help create a believable game world. This can include environmental sounds like wind, water, and wildlife.

Dynamic Audio: Adaptive music systems change the soundtrack based on the player's actions or game events, ensuring that the audio experience is always relevant and engaging.

Tools for Sound and Music Integration

FMOD: A popular audio middleware that allows developers to create complex audio behaviors without extensive coding. FMOD integrates well with game engines like Unity and Unreal Engine.

Wwise: Another powerful audio middleware, Wwise offers advanced features for audio integration, including real-time parameter control and interactive music systems.

Unity: Unity's built-in audio tools support a wide range of audio features, from simple sound effects to complex audio environments. Unity also supports integration with FMOD and Wwise.

Unreal Engine: Known for its high-quality audio capabilities, Unreal Engine provides robust tools for sound design and music integration, including support for spatial audio and real-time audio effects.

https://doi.org/10.15215/remix/9781998944224.01

Skills Involved

Sound Design: Creating and implementing sound effects that match the game's visual and interactive elements.

Music Composition: Composing original music that enhances the game's narrative and emotional impact.

Audio Scripting: Using scripts to control when and how sounds are played, ensuring they align with game events and player actions.

1.4.8. Gameplay Programming

Gameplay programming is a crucial aspect of game development, focusing on creating the interactive elements that make a game enjoyable and engaging.

Key Responsibilities of Gameplay Programming

Implementing Game Mechanics: Gameplay programmers bring the game design to life by coding the core mechanics, such as character movement, player controls and interaction, combat systems, and puzzle logic, as well as implementing features like scoring systems, power-ups, and enemy behavior.

Balancing and Tuning: They ensure that the game is fun and challenging by adjusting variables and fine-tuning gameplay elements. This involves playtesting and iterating based on feedback.

Collaboration: Gameplay programmers work closely with designers, artists, and other developers to integrate various game components seamlessly. Effective communication and teamwork are essential.

Bug Fixing and Optimization: They identify and fix bugs to ensure a smooth gaming experience. Optimization is also crucial to maintain performance, especially on different hardware platforms.

Tools and Technologies

Game Engines: Popular game engines like Unity and Unreal Engine provide robust frameworks for gameplay programming. These engines offer tools and libraries that simplify the development process.

Scripting Languages: Many game engines use scripting languages like C# (Unity) or Blueprints (Unreal Engine) to implement gameplay features. These languages allow for rapid prototyping and iteration.

https://doi.org/10.15215/remix/9781998944224.01

Version Control Systems: Tools like Git are essential for managing code changes and collaborating with other team members. They help track progress and revert to previous versions if needed.

Skills and Knowledge

Programming Proficiency: Strong skills in languages like C++, C#, or Python are essential. Understanding object-oriented programming and design patterns is also important.

Mathematics and Physics: A good grasp of mathematics and physics helps in implementing realistic game mechanics and simulations.

Problem-Solving: Gameplay programmers need to be adept at solving complex problems and debugging issues that arise during development.

1.4.9. Artificial Intelligence Techniques for Game Programming

Artificial Intelligence (AI) is a powerful tool in game programming, enhancing the complexity and realism of games.

Key Techniques of AI for Game Development

Finite State Machines (FSMs): FSMs are used to manage the states and transitions of game entities. They are ideal for simple AI behaviors, such as enemy patrol routes or basic decision-making.

Behavior Trees: These hierarchical models are used to create complex behaviors by organizing tasks and decisions in a tree structure. They are more flexible than FSMs and are commonly used for NPC behaviors.

Pathfinding Algorithms: Algorithms like A* (A-star) are used to navigate game environments. They help characters find the shortest path between points, avoiding obstacles and optimizing movement.

Decision Trees: These are used for making decisions based on a series of conditions. They can be trained to perform classification and regression tasks, making them useful for dynamic decision-making.

Neural Networks: Deep learning models, such as neural networks, can learn from data to perform tasks like pattern recognition and decision-making. They are used for more advanced AI behaviors, such as adaptive learning and complex strategy.

https://doi.org/10.15215/remix/9781998944224.01

Genetic Algorithms: These algorithms simulate the process of natural selection to optimize solutions. They are used for tasks like procedural content generation and evolving game strategies.

Reinforcement Learning: This technique involves training AI agents to make decisions by rewarding them for desirable actions. It is used for creating adaptive and intelligent behaviors that improve over time.

Practical Applications of AI in Game Programming

NPC Behavior: AI techniques are used to create realistic and responsive nonplayer characters (NPCs) that can interact with players in meaningful ways.

Procedural Content Generation: AI can generate game content—such as levels, maps, and quests—dynamically providing a unique experience for each player.

Adaptive Difficulty: AI can adjust the game's difficulty based on the player's skill level, ensuring a balanced and engaging experience.

Tools and Frameworks

Unity: Supports various AI techniques through built-in tools and third-party plugins.

Unreal Engine: Offers robust AI tools, including behavior trees and pathfinding systems.

Python and Pygame: Python is popular for AI game programming because of its simplicity and powerful libraries (like pygame).

AI in game programming is a dynamic field that continues to evolve, offering endless possibilities for creating immersive and engaging gaming experiences.

1.4.10. Game Optimization and Performance

Game optimization is about techniques for optimizing code, reducing memory usage, and improving frame rates, as well as profiling tools and debugging strategies.

These performance improvements are crucial aspects of game development that ensure a smooth and enjoyable experience for players. You can have the best game in the world, but if it crashes randomly or has slow/laggy graphics or responsiveness, people will stop playing it.

https://doi.org/10.15215/remix/9781998944224.01

Knowledge Areas

Understanding Game Performance Metrics

- **Frame Rate (fps):** Ensuring a high and stable frame rate for smooth gameplay
- **CPU and GPU Usage:** Balancing the workload between the CPU and GPU to avoid bottlenecks
- **Memory Consumption:** Efficiently managing memory to prevent leaks and excessive usage

Optimization Techniques

- **Memory Management:** Techniques like memory pooling and data structure optimization to reduce overhead
- **Asset Streaming:** Loading only necessary assets on-the-fly to reduce load times and memory usage
- **Frame Rate Stabilization:** Using methods like dynamic resolution scaling and optimizing rendering paths

Profiling and Debugging

- **Profiling Tools:** Using tools to analyze performance metrics and identify bottlenecks
- **Debugging Skills:** Identifying and fixing performance-related bugs and issues

Skills

Programming Skills

- **Efficient Coding:** Writing optimized code that runs efficiently on various hardware
- **Knowledge of Game Engines:** Understanding how to use and optimize popular game engines like Unity and Unreal Engine

Graphics and Rendering

- **Shader Optimization:** Writing efficient shaders to reduce rendering load
- **Level of Detail (LOD):** Implementing LOD systems to adjust the detail of models and textures based on their distance from the camera

System Design

- **Asynchronous Loading:** Implementing asynchronous loading to improve user experience by reducing perceived load times

https://doi.org/10.15215/remix/9781998944224.01

Segmentation and Prioritization: Dividing game assets into segments and prioritizing their loading based on the player's current state

Testing and Quality Assurance

Playtesting: Gathering feedback from players to identify performance issues

Automated Testing: Using automated tests to regularly check for performance regressions

Tools

Profilers: Tools like Unity Profiler, Unreal Engine Profiler, and external tools like NVIDIA Nsight

Debuggers: Tools like Visual Studio Debugger, GDB, and platform-specific debuggers

Mastering these knowledge areas and skills can significantly enhance your ability to optimize game performance and deliver a seamless gaming experience.

1.4.11. Game Programming Library vs. Game Engines

Libraries and frameworks serve specific and fundamental needs of game programming, whereas a game engine encompasses all the necessary components more directly for creating a complete game experience. Libraries, such as Allegro for game development or FMOD for audio, provide specialized functions that developers can integrate into their projects to build customized solutions. In contrast, game engines like Unity or Unreal Engine offer a comprehensive suite of tools and features, simplifying the development process and often including built-in support for various aspects such as rendering, physics, and sound. While libraries require a deeper understanding and manual integration, game engines offer an all-in-one solution, making them more accessible for rapid development.

Game Development Libraries

A library is a collection of code modules or functions that serve specific purposes. These modules can be used to accomplish tasks like math calculations, file loading, collision detection, rendering, and more. Some libraries concentrate on a specific area; others encompass a variety of functions. Allegro has a variety of game-related functions, including sound handling, while DirectSound only entails sound-specific functions.

https://doi.org/10.15215/remix/9781998944224.01

The purpose of libraries is to provide building blocks for developers to create their own custom solutions. As noted, some focus on specific functionalities and others are general, but typically they are modular.

Examples of game development libraries include the following:

- Allegro (for game development)
- Novodex (for physics)
- FMOD (for audio)
- Havok (for physics)
- Ogre3D (for rendering)
- BINK (for video playback)
- Direct3D and OpenGL (for graphics)
- DirectSound (for audio)
- XInput (for input handling)

Game Engines

A game engine is an extensive framework specifically designed for creating games. It goes beyond libraries and frameworks by providing a comprehensive solution.

Components of a game engine typically include:

Rendering Engine: Handles graphics and rendering
Physics Engine: Manages physical interactions
Audio System: Deals with sound effects and music
Input Handling: Captures user input (keyboard, mouse, controllers)
Networking: Facilitates multiplayer functionality
Resource Management: Efficiently manages assets (textures, models, sounds)

Examples of game engines include:

- Unreal Engine
- Unity Engine
- Source Engine
- Quake Engine
- Reality Engine

Generally speaking, the distinction between a library and an engine lies in the level of abstraction and the scope of functionality.

https://doi.org/10.15215/remix/9781998944224.01

1.5. Set Up Game Development IDE

Allegro game library is platform independent, and can be used on Windows, Linux, Apple iOS, and Android as well. In this section, we will explain how a game development environment can be set up on Windows, Linux, and MacOS.

The Allegro 5 library serves as a framework for game development, encompassing essential aspects like math, physics, rendering, and audio. It provides a cohesive set of tools, conventions, and patterns to simplify development, offering a solid foundation for building applications or games.

Allegro 5 offers a comprehensive suite of functions that enable developers to manage graphics, handle user input, and incorporate sound effects seamlessly. This framework ensures efficient resource management and supports flow control, providing a structured approach to game development.

We are going to learn game programming with the Allegro 5 game library in C++ in this book. In order to make our game programming more efficient, especially when working on large game projects, we need to have an integrated development environment. There are different ways of setting up an IDE on your chosen computing platform to meet our needs. In this book, however, we will be using the same set of open-source software tools for all three types of computing platforms (Windows, Linux, and MacOS) we cover in this book. The tools in the set are VS Code by Microsoft, GCC by GNU, CMake, and Ninja.

1.5.1. Visual Studio Code with MinGW GCC, CMake, and Ninja

Visual Studio Code

Visual Studio Code (VS Code) is a popular, open-source code editor developed by Microsoft. It's known for its versatility, ease of use, and extensive features that cater to a wide range of programming needs.

Those features include extensibility, with an integrated terminal, debugging tools, IntelliSense tools for efficient coding, version controls for project development, and customization options for the entire IDE, as well as Live Share, which allows you to collaborate with others across the globe, such as through GitHub.

MinGW GCC

MinGW-w64 is a popular and comprehensive development environment for building native Windows applications. It offers the following key features:

Complete Runtime Environment: MinGW-w64 provides a complete runtime environment for the GCC (GNU Compiler Collection)

https://doi.org/10.15215/remix/9781998944224.01

to support binaries native to both 32-bit and 64-bit Windows operating systems.

Wide Compatibility: It supports a wide range of Windows APIs and is compatible with various libraries and tools, making it suitable for developing a broad spectrum of applications.

Multilib Toolchains: MinGW-w64 supports multilib toolchains, allowing you to build both 32-bit and 64-bit binaries from the same environment.

Advanced Features: It includes support for native TLS (thread local storage) callbacks, wide-character start-up, and more.

Cross-Platform Development: MinGW-w64 can be used in conjunction with other projects like Cygwin, MSYS2, and Wine, facilitating cross-platform development.

CMake

CMake is a powerful and widely used open-source tool designed to manage the build process of software projects.

Key Features

Here are some key aspects of CMake:

Cross-Platform: CMake supports multiple platforms, including Windows, macOS, and Linux, allowing developers to write build scripts that work across different operating systems.

Out-of-Source Builds: It supports out-of-source builds, which means you can keep your source directory clean by generating build files in a separate directory.

Single Source Builds: CMake allows you to use a single source tree to build your project on multiple platforms, simplifying the build process.

Extensive Language Support: It supports a variety of programming languages, including C, C++, Fortran, and more.

Custom Scripting Language: CMake includes its own scripting language, which allows you to define build configurations, manage dependencies, and customize the build process.

Integration with IDEs: CMake integrates well with various integrated development environments (IDEs) like Visual Studio, CLion, and Xcode.

https://doi.org/10.15215/remix/9781998944224.01

How CMake Works

The core of a CMake project is the CMakeLists.txt file, which contains commands and instructions for building the project. This file specifies the source files, build options, and dependencies.

Configuration and Generation: CMake first configures the project by reading the CMakeLists.txt file and then generates the necessary build files for the chosen build system (e.g., Makefiles, Visual Studio project files).

Here's a very simple example of CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)
set(CMAKE_CXX_STANDARD 17)
add_executable(MyProject main.cpp)
```

Configure and Build: Open a terminal, navigate to your project directory, and run:

```
cmake .
```

to configure the project.

And then run:

```
cmake --build  .
```

to build the project.

CMake is a versatile tool that can significantly streamline the build process for complex projects.

Ninja

Ninja is a small, high-speed build system designed to handle the build process for large software projects efficiently.

Key Features

Here are some key aspects of Ninja:

Speed: Ninja is optimized for speed, making it ideal for large projects that require fast incremental builds. Using it on top of CMake

https://doi.org/10.15215/remix/9781998944224.01

minimizes the time spent on build tasks by focusing on quick dependency checking and parallel execution. It has the following attractive features:

Simplicity: Ninja uses a minimal build description language, which is not intended to be written by hand. Instead, it relies on higher-level build systems like CMake, GYP, or Meson to generate its build files.

Efficiency: Ninja is designed to be an "assembler" for build systems, meaning it takes care of the low-level details of the build process, allowing higher-level tools to handle the more complex configuration tasks.

Cross-Platform: Ninja supports multiple operating systems, including Linux, macOS, and Windows, making it a versatile tool for cross-platform development. Ninja is a powerful tool for developers looking to optimize their build processes.

How Ninja Works

Build Files: Ninja uses .ninja build files, which are typically generated by higher-level build systems. These files contain the rules and dependencies needed to build the project.

Incremental Builds: Ninja excels at performing incremental builds, where only the parts of the project that have changed are rebuilt. This significantly reduces build times compared to traditional build systems.

Parallel Execution: Ninja can execute build tasks in parallel, taking full advantage of multicore processors to speed up the build process.

Using Ninja on Top of CMake

Using Ninja on top of CMake offers several benefits, particularly for large and complex projects. Here are some key reasons why this combination is advantageous.

Speed

Fast Builds: Ninja is designed for speed, focusing on quick dependency checking and parallel execution. This makes it significantly faster than traditional build systems like Make, especially for incremental builds.

Efficient Dependency Management: Ninja efficiently handles dependencies, ensuring that only the necessary parts of the project are rebuilt, which further reduces build times.

https://doi.org/10.15215/remix/9781998944224.01

Simplicity and Automation

Generated Build Files: Ninja build files are typically generated by higher-level build systems like CMake. This means you don't have to write Ninja build scripts by hand, simplifying the build process.

Integration with CMake: CMake can generate Ninja build files, allowing you to leverage CMake's powerful configuration capabilities while benefiting from Ninja's fast build performance.

Efficient Cross-Platform Development

CMake supports multiple platforms, and by using Ninja as the build tool, you can ensure consistent and fast builds across different operating systems.

1.5.2. Windows OS: Setting Up IDE for Game Programming with Allegro 5 in C/C++

In this subsection, we will explain how to install MinGW GCC, Visual Studio Code (Code), CMake, and Ninja on Windows and make them work together as an integrated development environment (IDE) for game development.

Setting up Visual Studio Code (VS Code) for game development with MinGW GCC, CMake, and Ninja involves several steps. Here's a detailed guide to help you get started.

Install Visual Studio Code

1. Download Visual Studio Code from the official website (https://code.visualstudio.com/download).
2. Run the installer and follow the on-screen instructions to complete the installation.

Install MinGW-w64

GCC can be installed through MinGW installer:

1. Download the MinGW-w64 installer from the MSYS2 website.
2. Run the installer and follow the steps to install MSYS2.
3. Open the MSYS2 terminal and install the MinGW-w64 toolchain by running:

```
   pacman -S --needed base-devel
mingw-w64-ucrt-x86_64-toolchain
```

https://doi.org/10.15215/remix/9781998944224.01

4. Add the path of your MinGW-w64 bin folder to the Windows PATH environment variable:
   - Open Windows Settings and search for "Edit environment variables for your account."
   - In the User variables section, select the Path variable and click Edit.
   - Add the path to the MinGW-w64 bin folder (e.g., C:\msys64\ucrt64\bin).

Install CMake

1. Download CMake from the official website.
2. Run the installer and follow the on-screen instructions to complete the installation.
3. Add CMake to your system PATH during the installation process.

Install Ninja

Install Ninja on Windows

1. Download the binary from the Ninja releases page, or download Ninja from the official GitHub repository.
2. Extract the downloaded file.
3. Add it to your system PATH.

Generate Ninja Build Files with CMake

1. Create a CMakeLists.txt file for your project.
2. Run CMake with Ninja as the generator:

```
cmake -G Ninja.
```

3. Build your project using Ninja:

```
    ninja
```

Here's a simple example of a CMakeLists.txt file for a project using Ninja.

EXAMPLE CMAKELISTS.TXT

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)
set(CMAKE_CXX_STANDARD 17)
add_executable(MyProject main.cpp)
```

https://doi.org/10.15215/remix/9781998944224.01

Install Allegro 5

1. **Download Allegro 5**
   Visit the Allegro 5 download page and download the prebuilt binaries for MinGW.
2. **Extract Allegro 5**
   Extract the Allegro 5 archive to a directory of your choice—for example, C:\allegro5.
3. **Set Up Environment Variables**
   Add the path to Allegro's bin directory (e.g., C:\allegro5\bin) to your system's PATH environment variable.

1.5.3. Linux OS: Setting Up IDE for Game Programming with Allegro 5 in C/C++

Here's a step-by-step guide to set up Visual Studio Code (VS Code) with GCC, CMake, and Ninja for game programming with Allegro 5 on Linux.

Install Visual Studio Code

1. **Download VS Code**
   Go to the Visual Studio Code download page and download the appropriate package for your Linux distribution.
2. **Install VS Code**
   - For Debian-based distributions (e.g., Ubuntu), use:

```
sudo apt install ./<file>.deb
```

   - For RedHat-based distributions (e.g., Fedora), use:

```
sudo rpm -i <file>.rpm
```

Install GCC

1. **Update Package Lists**

```
sudo apt update
```

2. **Install GCC**

```
sudo apt install build-essential
```

   This command installs GCC, G++, and other essential development tools (https://www.linuxfordevices.com/tutorials/linux/install-cmake-on-linux).

https://doi.org/10.15215/remix/9781998944224.01

Install CMake

1. **Install CMake**

```
sudo apt install cmake
```

Alternatively, you can download the latest version from the CMake website and follow the installation instructions (https://linuxcapable.com/how-to-install-cmake-on-ubuntu-linux/).

Install Ninja

1. **Install Ninja**

```
sudo apt install ninja-build
```

Alternatively, you can download the latest version from the Ninja releases page (https://github.com/ninja-build/ninja.git) and follow the installation instructions.

Install Allegro 5

1. **Install Allegro 5**

```
sudo apt install liballegro5-dev
```

1.5.4. Apple MacOS: Setting Up IDE for Game Programming with Allegro 5 in C/C++

For the most up-to-date instructions on installing VS Code, GCC/G++, CMake, and Ninja on macOS, please consult an AI assistant. The following guide, validated at the time of writing, is provided to help you get started.

Install Visual Studio Code

1. **Download VS Code**
   Go to the Visual Studio Code download page (https://code.visualstudio.com/download) and download the macOS version.
2. **Install VS Code**
   Open the downloaded .zip file and drag Visual Studio Code.app to the Applications folder.

Install GCC

1. **Install Homebrew**
   Open the Terminal and run the following command to install Homebrew, a package manager for macOS:

https://doi.org/10.15215/remix/9781998944224.01

```
   /bin/bash -c "$(curl -fsSL https://raw
.githubusercontent.com/Homebrew/install/HEAD/
install.sh)"
```

2. **Install GCC**
   Once Homebrew is installed, run:

```
   brew install gcc
```

Install CMake

1. **Install CMake**
   Use Homebrew to install CMake by running:

```
   brew install cmake
```

Install Ninja

1. **Install Ninja**
   Use Homebrew to install Ninja by running:

```
   brew install ninja
```

Install Allegro 5

1. **Install Allegro 5**
   Use Homebrew to install Allegro 5 by running:

```
   brew install allegro
```

1.5.5. Configure Visual Studio Code to Make Everything Work Together

After you have all the tools and the Allegro 5 library installed on your platform, you need to start VS Code and make everything work together in the integrated development environment.

Install Extensions

1. Open VS Code and go to the Extensions view (⇧⌘X or Ctrl + Shift + X).
2. Search and install the following extensions:
   - C/C++ by Microsoft
   - CMake Tools by Microsoft

https://doi.org/10.15215/remix/9781998944224.01

Test Your IDE by Developing a Simple Project

1. Create a project folder with the following structure:

```
HelloAllegro/
├── CMakeLists.txt
└── src/
    └── main.cpp
```

2. Open your project folder in VS Code.
3. Open the CMakeLists.txt file and add the following to the file and save:

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.10)
project(HelloAllegro)

# Set C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Find Allegro 5 package
find_package(PkgConfig REQUIRED)
pkg_check_modules(ALLEGRO5 REQUIRED allegro-5
allegro_main-5 allegro_image-5 allegro_font-5
allegro_ttf-5 allegro_primitives-5 allegro_audio-5
allegro_acodec-5)

# Include Allegro headers
include_directories(${ALLEGRO5_INCLUDE_DIRS})

# Add executable
add_executable(hello_allegro src/main.cpp)

# Link Allegro libraries
target_link_libraries(hello_allegro
${ALLEGRO5_LIBRARIES})
```

4. Create a main.cpp file in the src directory with the following code:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
```

https://doi.org/10.15215/remix/9781998944224.01

```
int main() {
    if (!al_init()) {
        return-1;
    }

    al_init_font_addon();
    al_init_ttf_addon();

    ALLEGRO_DISPLAY* display = al_create_
display(640, 480);
    if (!display) {
        return-1;
    }

    ALLEGRO_FONT* font = al_load_ttf_font("arial.
ttf", 36, 0);
    if (!font) {
        al_destroy_display(display);
        return-1;
    }

    al_clear_to_color(al_map_rgb(0, 0, 0));
    al_draw_text(font, al_map_rgb(255, 255, 255),
320, 240, ALLEGRO_ALIGN_CENTER, "Hello, Allegro!");
    al_flip_display();

    al_rest(2.0);

    al_destroy_font(font);
    al_destroy_display(display);
    return 0;
}
```

5. Configure CMake in VS Code
   Within VS Code, open the Command Palette (⇧⌘P or Ctrl + Shift + P) and run CMake: Configure, then select the appropriate kit (e.g., GCC for MinGW-w64) and specify the generator as Ninja.
6. Build and Run Your Project
   There are two ways to build and run the project. The first one is within a terminal on your computer:

https://doi.org/10.15215/remix/9781998944224.01

Step 1. Open a terminal within VS Code, or independent of VS Code on your computer, if you don't have one already, and navigate to the project directory:

```
cd HelloAllegro
```

Step 2. Generate build files with CMake:

```
cmake -G Ninja -DCMAKE_BUILD_TYPE=Release
-DCMAKE_PREFIX_PATH="C:/allegro5" .
```

Step 3. Build the project with Ninja:

```
ninja
```

Step 4. Run the program.

After building, you should have an executable named hello_allegro in your project directory. Run it from the terminal by typing hello_allegro and then hitting the return key to see the "Hello, Allegro!" message displayed in a window.

Another way is to build from the VS Code Command Palette:

Step 1. Build your project.

- Open the Command Palette (⇧⌘P or Ctrl+Shift+P) and run CMake: Configure.
- Select the appropriate kit (e.g., GCC) and specify the generator as Ninja.
- Run CMake: Build.

Step 2. Run your project.

- Use the CMake: Run command or configure a launch configuration in the launch.json file.

1.6. The Allegro 5 Game Library

One nice thing about modern game libraries is that there are many online resources available to help one use them, and Allegro 5 is no different. As such, in this textbook, you'll have seen references to online materials, as they can do a fine job of explaining what needs to be known. What follows is a combination of information from the official Allegro 5 website (https://www.allegro

https://doi.org/10.15215/remix/9781998944224.01

.cc/manual/5/getting_started.html) with additional information provided by the authors of this text.

1.6.1. Overview of Allegro 5 Core Libraries and Add-Ons

Allegro 5.0 is divided into a core library and multiple add-ons. The add-ons are bundled together and built at the same time as the core, but they are distinct and kept in separate libraries. The core doesn't depend on anything in the add-ons, but add-ons may depend on the core and other add-ons and additional third-party libraries.

Implemented in a single library file named allegro.lib on Windows and allegro.so on Linux and MacOS, Allegro core supports the use of the following resources:

System: The Allegro 5 System library is a core part of the Allegro game programming library. It provides essential functions for initializing and managing the Allegro environment, handling system events, and managing resources. Defined in allegro.h so that no additional header file is needed.

Display: Used for creating and managing displays (windows) in your applications. Defined in allegro.h.

Events: Provides a comprehensive event system that allows you to handle various types of events, such as keyboard input, mouse input, timers, and more. allegro5/allegro_native_dialog.h is needed on top of allegro.h and header files for respective input devices to be used.

File I/O: Provides a robust set of functions for file input and output (I/O), allowing you to read from and write to files in a cross-platform manner. Defined in allegro.h.

Filesystem: Provides a set of functions for interacting with the filesystem, allowing you to manage files and directories in a cross-platform manner. Defined in allegro.h.

Fixed-Point Math: Provides a set of functions for working with fixed-point arithmetic, which can be useful for certain types of calculations, especially in environments where floating-point operations are less efficient. Fixed-point numbers in Allegro are represented using a 32-bit integer type called al_fixed, where the high word is used for the integer part and the low word for the fractional part. allegro5/allegro_fixed.h is needed on top of allegro.h.

https://doi.org/10.15215/remix/9781998944224.01

Graphics: Provides a robust set of functions for handling graphics, allowing you to create and manipulate images, draw shapes, and manage colours. Only basic operations are defined in allegro5/allegro.h. For operations such as drawing various shapes, allegro5/allegro_primitives.h, an add-on, is needed.

Joystick: Provides a set of functions for handling joystick input, allowing you to interact with joystick devices in your applications. allegro5/allegro_joystick.h is needed on top of allegro.h.

Keyboard: Provides a comprehensive set of functions for handling keyboard input. allegro5/allegro_keyboard.h is needed on top of allegro.h.

Mouse: Provides a comprehensive set of functions for handling mouse input and displaying a mouse cursor. allegro5/allegro_mouse.h is needed in addition to allegro.h.

Memory: Provides several functions for memory management, allowing you to allocate, reallocate, and free memory in a consistent manner across different platforms. These basic operations on memory will work with only allegro.h.

Path: Provides functions for manipulating and managing file system paths. This is useful for handling file and directory paths in a cross-platform manner. Defined in allegro.h.

State: Provides functions to save and restore the state of various Allegro objects. This is useful for managing the state of graphics, transformations, and other settings in your application. Defined in allegro.h.

Threads: Built on top of Windows threads and POSIX threads (pthreads), the library provides a simple cross-platform threading interface. Defined in allegro.h.

Time: Provides functions to manage and measure time. Defined in allegro.h.

Timer: Provides functions to create and manage timers. Defined in allegro.h, and no extra header file is needed.

Transformations: Provides functions to manipulate and apply transformations to graphics. Defined in allegro.h.

UTF-8: Provides support for UTF-8 encoded strings, which is essential for handling Unicode text in your applications. Defined in allegro.h.

Miscellaneous: Includes various utility functions and features that don't fit neatly into other specific categories. Defined in allegro.h.

https://doi.org/10.15215/remix/9781998944224.01

Platform-Specific Libraries: Includes libraries specific to Windows (Direct3D and Native dialogues), Linux (X11 integration and OpenGL integration), macOS (Cocoa integration and OpenGL ES integration), iOS (UIKit Integration and OpenGL ES integration), and Android (JNI integration and OpenGL ES integration).

Direct3D: Provides integration with Direct3D, allowing you to use Direct3D for hardware-accelerated graphics on Windows. allegro5/allegro_direct3d.h is needed on top of allegro.h.

OpenGL: Provides integration with OpenGL, allowing you to use OpenGL for hardware-accelerated graphics. allegro5/allegro_opengl.h is needed in addition to allegro.h.

The add-ons and their dependencies of libraries in the core are as follows:

allegro_main, which depends on allegro
: Ensures cross-platform compatibility for your main function. Required on Windows when using main() instead of WinMain().

allegro_image, which depends on allegro
: Provides support for loading and saving bitmap images in common formats such as PNG, JPG, BMP, and so on.

allegro_primitives, which depends on allegro
: Adds drawing functions for basic geometric shapes such as lines, rectangles, circles, polygons, and splines.

allegro_color, which depends on allegro
: Offers functions for working with colours, including conversions of colour spaces (RGB, HSV, HSL, etc.).

allegro_font, which depends on allegro
: Provides a system for loading, creating, and rendering bitmap fonts.

allegro_ttf, which depends on allegro_font
: Adds support for loading and rendering TrueType fonts via the FreeType library.

allegro_audio, which depends on allegro
: Implements an audio playback system, allowing you to play samples, streams, and manage audio voices.

allegro_acodec, which depends on allegro_audio
: Adds audio codecs for loading and saving common audio formats such as OGG, WAV, and FLAC.

https://doi.org/10.15215/remix/9781998944224.01

allegro_memfile, which depends on allegro
: Provides the ability to treat blocks of memory as if they were file streams, useful for in-memory loading.

allegro_physfs, which depends on allegro
: Integrates PhysicsFS support, allowing Allegro to read from archive formats (ZIP, 7z, etc.) as if they were directories.

allegro_native_dialog, which depends on allegro
: Provides platform-native dialogues, such as file selectors, message boxes, and text log windows.

1.6.2. Fundamental Operations of Video Games with Allegro 5 in C/C++

The header file for the core library is allegro5/allegro.h. The header files for the add-ons are named allegro5/allegro_image.h, allegro5/allegro_font.h, and so on. The allegro_main add-on does not have a header file.

Initialization

Before using Allegro, you must call al_init. Some add-ons have their own initialization: for example, al_init_image_addon, al_init_font_addon, and al_init_ttf_addon.

To receive input, you need to initialize some subsystems like al_install_keyboard, al_install_mouse, and al_install_joystick.

Opening a Window

A call to al_create_display will open a window and return an ALLEGRO_DISPLAY.

To clear the display, call al_clear_to_color. Use al_map_rgba or al_map_rgba_f to obtain an ALLEGRO_COLOR parameter.

Drawing operations are performed on a backbuffer. To make the operations visible, call al_flip_display.

Display an Image

To load an image from disk, you need to have initialized the image I/O add-on with al_init_image_addon. Then use al_load_bitmap, which returns an ALLEGRO_BITMAP.

Use al_draw_bitmap, al_draw_scaled_bitmap, or al_draw_scaled_rotated_bitmap to draw the image to the backbuffer. Remember to call al_flip_display.

https://doi.org/10.15215/remix/9781998944224.01

Changing the Drawing Target

Notice that al_clear_to_color and al_draw_bitmap didn't take destination parameters—the destination is implicit. Allegro remembers the current "target bitmap" for the current thread. To change the target bitmap, call al_set_target_bitmap.

The backbuffer of the display is also a bitmap. You can get it with al_get_backbuffer and then restore it as the target bitmap.

Other bitmaps can be created with al_create_bitmap, with options that can be adjusted with al_set_new_bitmap_flags and al_set_new_bitmap_format.

Event Queues and Input

Input comes from multiple sources—keyboard, mouse, joystick, timers, and so on. Event queues aggregate events from all these sources, then you can query the queue for events.

Create an event queue with al_create_event_queue, then tell input sources to place new events into that queue using al_register_event_source. The usual input event sources can be retrieved with al_get_keyboard_event_source, al_get_mouse_event_source, and al_get_joystick_event_source.

Events can be retrieved with al_wait_for_event or al_get_next_event. Check the event type and other fields of ALLEGRO_EVENT to react to the input.

Displays are also event sources, which emit events when they are resized. We will need to set the ALLEGRO_RESIZABLE flag with al_set_new_display_flags before creating the display, then register the display with an event queue. When you get a resize event, call al_acknowledge_resize.

Timers are event sources that "tick" periodically, causing an event to be inserted into the queues that the timer is registered with. Create some with al_create_timer.

al_get_time and al_rest are more direct ways to deal with time.

Displaying Some Text

To display some text, initialize the image and font add-ons with al_init_image_addon and al_init_font_addon, then load a bitmap font with al_load_font. Use al_draw_text or al_draw_textf.

For TrueType fonts, you'll need to initialize the TTF (TrueType font) add-on with al_init_ttf_addon and load a TTF font with al_load_ttf_font.

Drawing Primitives

The primitives add-on provides some handy routines to draw lines (al_draw_line), rectangles (al_draw_rectangle), circles (al_draw_circle), and so on.

https://doi.org/10.15215/remix/9781998944224.01

Blending

To draw translucent or tinted images or primitives, change the blender state with al_set_blender.

As with al_set_target_bitmap, this changes Allegro's internal state (for the current thread). Often, you'll want to save some part of the state and restore it later. The functions al_store_state and al_restore_state provide a convenient way to do that.

Sound

Use al_install_audio to initialize sound. To load any sample formats, you will need to initialize the acodec add-on with al_init_acodec_addon.

After that, you can simply use al_reserve_samples and pass the number of sound effects typically playing at the same time. Then load your sound effects with al_load_sample and play them with al_play_sample. To stream large pieces of music from disk, you can use al_load_audio_stream so the whole piece will not have to be preloaded into memory.

If this all sounds too simple and you can't help but think about clipping and latency issues, don't worry. Allegro gives you full control over how much or little you want its sound system to do. The al_reserve_samples function mentioned here only sets up a default mixer and a number of sample instances, but you don't need to use it.

Instead, to get a "direct connection" to the sound system, you would use an ALLEGRO_VOICE, but depending on the platform, only one such voice is guaranteed to be available, and it might require a specific format of audio data. Therefore, all sound can be first routed through an ALLEGRO_MIXER that is connected to such a voice (or another mixer) and will mix all sample data fed to it.

You can then directly stream real-time sample data to a mixer or a voice using an ALLEGRO_AUDIO_STREAM or play complete sounds using an ALLEGRO_SAMPLE_INSTANCE. The latter simply points to an ALLEGRO_SAMPLE and will stream it for you.

Code Samples

Text and Font

Include the necessary headers:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <cstdio>
```

https://doi.org/10.15215/remix/9781998944224.01

Initialize Allegro and the font add-ons:

```
if (!al_init()) {
    fprintf(stderr, "Failed to initialize Allegro!\n");
    return–;
}

if (!al_init_font_addon()) {
    fprintf(stderr, "Failed to initialize font
addon!\n");
    return–;
}

if (!al_init_ttf_addon()) {
    fprintf(stderr, "Failed to initialize TTF addon!\n");
    return–;
}
```

Create a display:

```
ALLEGRO_DISPLAY *display = al_create_display(800, 600);
if (!display) {
    fprintf(stderr, "Failed to create display!\n");
    return–;
}
```

Load a font:

```
ALLEGRO_FONT *font = al_load_ttf_font("arial.ttf", 36, 0);
if (!font) {
    fprintf(stderr, "Could not load 'arial.ttf'.\n");
    al_destroy_display(display);
    return–;
}
```

Draw text to the screen:

```
al_clear_to_color(al_map_rgb(0, 0, 0));
al_draw_text(font, al_map_rgb(255, 255, 255), 400, 300,
ALLEGRO_ALIGN_CENTRE, "Hello, Allegro!");
```

https://doi.org/10.15215/remix/9781998944224.01

```
al_flip_display();
al_rest(5.0);
```

Clean up:

```
al_destroy_font(font);
al_destroy_display(display);
```

Here's the complete code put together:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <cstdio>

int main() {
    if (!al_init()) {
        fprintf(stderr, "Failed to initialize
Allegro!\n");
        return–;
    }

    if (!al_init_font_addon()) {
        fprintf(stderr, "Failed to initialize font
addon!\n");
        return–;
    }

    if (!al_init_ttf_addon()) {
        fprintf(stderr, "Failed to initialize TTF
addon!\n");
        return–;
    }

    ALLEGRO_DISPLAY *display = al_create_display(800,
600);
    if (!display) {
        fprintf(stderr, "Failed to create display!\n");
        return–;
    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    ALLEGRO_FONT *font = al_load_ttf_font("arial.ttf",
36, 0);
    if (!font) {
        fprintf(stderr, "Could not load 'arial.ttf'.\n");
        al_destroy_display(display);
        return–;
    }

    al_clear_to_color(al_map_rgb(0, 0, 0));
    al_draw_text(font, al_map_rgb(255, 255, 255), 400,
300, ALLEGRO_ALIGN_CENTRE, "Hello, Allegro!");
    al_flip_display();
    al_rest(5.0);

    al_destroy_font(font);
    al_destroy_display(display);

    return 0;
}
```

This example initializes Allegro, sets up the font add-ons, creates a display, loads a TrueType font, and draws text to the screen. It then waits for five seconds before cleaning up and closing the display.

Graphics

Include the necessary headers:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <cstdio>
```

Initialize Allegro and the image add-on:

```
if (!al_init()) {
    fprintf(stderr, "Failed to initialize Allegro!\n");
    return–1;
}

if (!al_init_image_addon()) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
    fprintf(stderr, "Failed to initialize image
addon!\n");
    return–1;
}
```

Create a display:

```
ALLEGRO_DISPLAY *display = al_create_display(800, 600);
if (!display) {
    fprintf(stderr, "Failed to create display!\n");
    return–1;
}
```

Load an image:

```
ALLEGRO_BITMAP *image = al_load_bitmap("example.png");
if (!image) {
    fprintf(stderr, "Failed to load image!\n");
    al_destroy_display(display);
    return–1;
}
```

Draw shapes and the image to the screen:

```
al_clear_to_color(al_map_rgb(0, 0, 0)); // Clear the
screen to black

// Draw a filled rectangle
al_draw_filled_rectangle(00, 00, 200, 200, al_map_
rgb(255, 0, 0));

// Draw a circle
al_draw_filled_circle(400, 300, 50, al_map_rgb(0, 255,
0));

// Draw the loaded image
al_draw_bitmap(image, 300, 200, 0);

al_flip_display(); // Display the changes
al_rest(5.0); // Wait for 5 seconds
```

https://doi.org/10.15215/remix/9781998944224.01

Clean up:

```
al_destroy_bitmap(image);
al_destroy_display(display);
```

Here's the complete code put together:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <csdtio>

int main() {
    if (!al_init()) {
        fprintf(stderr, "Failed to initialize
Allegro!\n");
        return–1;
    }
if (!al_init_primitives_addon()) {
        fprintf(stderr, "Failed to initialize primitives
addon!\n");
        return–1;
    }
    if (!al_init_image_addon()) {
        fprintf(stderr, "Failed to initialize image
addon!\n");
        return–1;
    }

    ALLEGRO_DISPLAY *display = al_create_display(800,
600);
    if (!display) {
        fprintf(stderr, "Failed to create display!\n");
        return–1;
    }
    ALLEGRO_BITMAP *image = al_load_bitmap("example.
png");
    if (!image) {
        fprintf(stderr, "Failed to load image!\n");
        al_destroy_display(display);
```

https://doi.org/10.15215/remix/9781998944224.01

```
        return-1;
    }

    al_clear_to_color(al_map_rgb(0, 0, 0)); // Clear the
screen to black

    // Draw a filled rectangle
    al_draw_filled_rectangle(00, 00, 200, 200, al_map_
rgb(255, 0, 0));

    // Draw a circle
    al_draw_filled_circle(400, 300, 50, al_map_rgb(0,
255, 0));

    // Draw the loaded image
    al_draw_bitmap(image, 300, 200, 0);

    al_flip_display(); // Display the changes
    al_rest(5.0); // Wait for 5 seconds

    al_destroy_bitmap(image);
    al_destroy_display(display);

    return 0;
}
```

This example initializes Allegro, sets up the image add-on, creates a display, loads an image, and draws both shapes and the image to the screen. It then waits for five seconds before cleaning up and closing the display.

Using a Keyboard

```
// Include the necessary headers:
#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <cstdio>

// Initialize Allegro and the necessary add-ons:
if (!al_init()) {
     fprintf(stderr, "Failed to initialize Allegro!\n");
```

https://doi.org/10.15215/remix/9781998944224.01

```
     return-1;
 }
 if (!al_install_keyboard()) {
     fprintf(stderr, "Failed to initialize the
keyboard!\n");
     return-1;
 }
 if (!al_init_font_addon()) {
     fprintf(stderr, "Failed to initialize font
addon!\n");
     return-1;
 }
 if (!al_init_ttf_addon()) {
     fprintf(stderr, "Failed to initialize TTF
addon!\n");
     return-1;
 }

// Create a display and an event queue:
ALLEGRO_DISPLAY *display = al_create_display(800, 600);
if (!display) {
    fprintf(stderr, "Failed to create display!\n");
    return-1;
}

ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();
if (!event_queue) {
    fprintf(stderr, "Failed to create event queue!\n");
    al_destroy_display(display);
    return-1;
}

al_register_event_source(event_queue,
al_get_display_event_source(display));
al_register_event_source(event_queue,
al_get_keyboard_event_source());

// Load a font:
```

https://doi.org/10.15215/remix/9781998944224.01

```
ALLEGRO_FONT *font = al_load_ttf_font("arial.ttf", 36,
0);
if (!font) {
    fprintf(stderr, "Could not load 'arial.ttf'.\n");
    al_destroy_display(display);
    al_destroy_event_queue(event_queue);
    return-1;
}

// Main loop to handle events:
bool running = true;
ALLEGRO_EVENT ev;

while (running) {
    al_wait_for_event(event_queue, &ev);

    if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
        running = false;
    } else if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
        switch (ev.keyboard.keycode) {
            case ALLEGRO_KEY_ESCAPE:
                running = false;
                break;
            case ALLEGRO_KEY_UP:
                al_clear_to_color(al_map_rgb(0, 0, 0));
                al_draw_text(font, al_map_rgb(255, 255,
255), 400, 300, ALLEGRO_ALIGN_CENTRE, "Up Key Pressed");
                al_flip_display();
                break;
            case ALLEGRO_KEY_DOWN:
                al_clear_to_color(al_map_rgb(0, 0, 0));
                al_draw_text(font, al_map_rgb(255,
255, 255), 400, 300, ALLEGRO_ALIGN_CENTRE, "Down Key
Pressed");
                al_flip_display();
                break;
            // Add more cases for other keys as needed
        }
    }
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
// Clean up:
al_destroy_font(font);
al_destroy_display(display);
al_destroy_event_queue(event_queue);
```

Here's the complete code put together:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <csdtio>

int main() {
if (!al_init()) {
     fprintf(stderr, "Failed to initialize Allegro!\n");
     return–1;
 }
 if (!al_install_keyboard()) {
     fprintf(stderr, "Failed to initialize the
keyboard!\n");
     return–1;
 }
 if (!al_init_font_addon()) {
     fprintf(stderr, "Failed to initialize font
addon!\n");
     return–1;
 }
 if (!al_init_ttf_addon()) {
     fprintf(stderr, "Failed to initialize TTF
addon!\n");
     return–1;
 }    ALLEGRO_DISPLAY *display = al_create_display(800,
600);
    if (!display) {
        fprintf(stderr, "Failed to create display!\n");
        return–;
    }

    ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();
```

https://doi.org/10.15215/remix/9781998944224.01

```
    if (!event_queue) {
        fprintf(stderr, "Failed to create event
queue!\n");
        al_destroy_display(display);
        return-1;
    }

    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_keyboard_event_source());

    ALLEGRO_FONT *font = al_load_ttf_font("arial.ttf",
36, 0);
    if (!font) {
        fprintf(stderr, "Could not load 'arial.ttf'.\n");
        al_destroy_display(display);
        al_destroy_event_queue(event_queue);
        return-1;
    }

    bool running = true;
    ALLEGRO_EVENT ev;

    while (running) {
        al_wait_for_event(event_queue, &ev);

        if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            running = false;
        } else if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
            switch (ev.keyboard.keycode) {
                case ALLEGRO_KEY_ESCAPE:
                    running = false;
                    break;
                case ALLEGRO_KEY_UP:
                    al_clear_to_color(al_map_rgb(0, 0,
0));
                    al_draw_text(font, al_map_rgb(255,
255, 255), 400, 300, ALLEGRO_ALIGN_CENTRE, "Up Key
Pressed");
```

https://doi.org/10.15215/remix/9781998944224.01

```
                    al_flip_display();
                    break;
                case ALLEGRO_KEY_DOWN:
                    al_clear_to_color(al_map_rgb(0, 0,
0));
                    al_draw_text(font, al_map_rgb(255,
255, 255), 400, 300, ALLEGRO_ALIGN_CENTRE, "Down Key
Pressed");
                    al_flip_display();
                    break;
                // Add more cases for other keys as
needed
            }
        }
    }

    al_destroy_font(font);
    al_destroy_display(display);
    al_destroy_event_queue(event_queue);

    return 0;
}
```

This example initializes Allegro, sets up the keyboard input, creates a display, and handles key-press events to display different messages based on the key pressed. It then waits for the user to close the window or press the Esc key before cleaning up and exiting.

Using a Joystick

```
// Include the necessary headers:
#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <allegro5/allegro_joystick.h>
#include <cstdio>

// Initialize Allegro and the necessary add-ons:
if (!al_init()) {
        fprintf(stderr, "Failed to initialize
Allegro!\n");
```

https://doi.org/10.15215/remix/9781998944224.01

```
        return-1;
    }

    if (!al_install_joystick()) {
        fprintf(stderr, "Failed to initialize the
joystick!\n");
        return-1;
    }

    if (!al_init_font_addon()) {
        fprintf(stderr, "Failed to initialize font
addon!\n");
        return-1;
    }

    if (!al_init_ttf_addon()) {
        fprintf(stderr, "Failed to initialize TTF
addon!\n");
        return-1;
}

//Create a display and an event queue:
ALLEGRO_DISPLAY *display = al_create_display(800, 600);
if (!display) {
    fprintf(stderr, "Failed to create display!\n");
    return-1;
}

ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();
if (!event_queue) {
    fprintf(stderr, "Failed to create event
queue!\n");
    al_destroy_display(display);
    return-1;
}

al_register_event_source(event_queue,
al_get_display_event_source(display));
```

https://doi.org/10.15215/remix/9781998944224.01

```
al_register_event_source(event_queue,
al_get_joystick_event_source());

// Load a font:
ALLEGRO_FONT *font = al_load_ttf_font("arial.ttf", 36,
0);
if (!font) {
    fprintf(stderr, "Could not load 'arial.ttf'.\n");
    al_destroy_display(display);
    al_destroy_event_queue(event_queue);
    return-1;
}

// Main loop to handle events:
bool running = true;
ALLEGRO_EVENT ev;

while (running) {
    al_wait_for_event(event_queue, &ev);

    if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
        running = false;
    } else if (ev.type == ALLEGRO_EVENT_JOYSTICK_AXIS) {
        printf("Joystick axis %d moved to %f\n",
ev.joystick.axis, ev.joystick.pos);
    } else if (ev.type == ALLEGRO_EVENT_JOYSTICK_BUTTON_
DOWN) {
        printf("Joystick button %d pressed\n",
ev.joystick.button);
    } else if (ev.type == ALLEGRO_EVENT_JOYSTICK_BUTTON_
UP) {
        printf("Joystick button %d released\n",
ev.joystick.button);
    }
}

// Clean up:
al_destroy_font(font);
al_destroy_display(display);
al_destroy_event_queue(event_queue);
```

https://doi.org/10.15215/remix/9781998944224.01

Here's the complete code put together:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <allegro5/joystick.h>
#include <cstdio>

int main() {
    if (!al_init()) {
        fprintf(stderr, "Failed to initialize Allegro!\n");
        return-1;
    }

    if (!al_install_joystick()) {
        fprintf(stderr, "Failed to initialize the
joystick!\n");
        return-1;
    }

    if (!al_init_font_addon()) {
        fprintf(stderr, "Failed to initialize font
addon!\n");
        return-1;
    }

    if (!al_init_ttf_addon()) {
        fprintf(stderr, "Failed to initialize TTF
addon!\n");
        return-1;
    }

    ALLEGRO_DISPLAY *display = al_create_display(800, 600);
    if (!display) {
        fprintf(stderr, "Failed to create display!\n");
        return-1;
    }

    ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();
```

https://doi.org/10.15215/remix/9781998944224.01

```
    if (!event_queue) {
        fprintf(stderr, "Failed to create event
queue!\n");
        al_destroy_display(display);
        return-1;
    }

    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_joystick_event_source());

    ALLEGRO_FONT *font = al_load_ttf_font("arial.ttf",
36, 0);
    if (!font) {
        fprintf(stderr, "Could not load 'arial.ttf'.\n");
        al_destroy_event_queue(event_queue);
        al_destroy_display(display);
        return-1;
    }

    bool running = true;
    ALLEGRO_EVENT ev;

    while (running) {
        al_wait_for_event(event_queue, &ev);

        if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            running = false;
        } else if (ev.type == ALLEGRO_EVENT_JOYSTICK_
AXIS) {
            printf("Joystick axis %d moved to %f\n",
ev.joystick.axis, ev.joystick.pos);
        } else if (ev.type == ALLEGRO_EVENT_JOYSTICK_
BUTTON_DOWN) {
            printf("Joystick button %d pressed\n",
ev.joystick.button);
        } else if (ev.type == ALLEGRO_EVENT_JOYSTICK_
BUTTON_UP) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
            printf("Joystick button %d released\n",
ev.joystick.button);
        }
    }
    // Clean up resources
    al_destroy_font(font);
    al_destroy_event_queue(event_queue);
    al_destroy_display(display);

    return 0;
}
```

This example initializes Allegro, sets up joystick input, creates a display, and handles joystick events such as axis movement and button presses. It then waits for the user to close the window before cleaning up and exiting.

Using Audio

```
// Include the necessary headers:
#include <allegro5/allegro.h>
#include <allegro5/allegro_audio.h>
#include <allegro5/allegro_acodec.h>
#include <cstdio>

//Initialize Allegro and the audio add-ons:
if (!al_init()) {
    fprintf(stderr, "Failed to initialize Allegro!\n");
    return-1;
}

if (!al_install_audio()) {
    fprintf(stderr, "Failed to initialize audio!\n");
    return-1;
}

if (!al_init_acodec_addon()) {
    fprintf(stderr, "Failed to initialize audio
codecs!\n");
    return-1;
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
if (!al_reserve_samples()) {
    fprintf(stderr, "Failed to reserve samples!\n");
    return-1;
}

// Load an audio sample:
ALLEGRO_SAMPLE *sample = al_load_sample("example.wav");
if (!sample) {
    fprintf(stderr, "Failed to load sample!\n");
    return-1;
}

// Play the audio sample:
if (!al_play_sample(sample, .0, 0.0, .0, ALLEGRO_
PLAYMODE_ONCE, NULL)) {
    fprintf(stderr, "Failed to play sample!\n");
    al_destroy_sample(sample);
    return-1;
}

// Wait for the sample to finish playing:
al_rest(5.0); // Wait for 5 seconds to ensure the sample
plays completely

// Clean up:
al_destroy_sample(sample);
al_uninstall_audio();
```

Here's the complete code put together:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_audio.h>
#include <allegro5/allegro_acodec.h>
#include <cstdio>

int main() {
    if (!al_init()) {
        fprintf(stderr, "Failed to initialize Allegro!\n");
        return-1;
    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    if (!al_install_audio()) {
        fprintf(stderr, "Failed to initialize
audio!\n");
        return-1;
    }

    if (!al_init_acodec_addon()) {
        fprintf(stderr, "Failed to initialize audio
codecs!\n");
        return-1;
    }

    if (!al_reserve_samples(1)) {
        fprintf(stderr, "Failed to reserve samples!\n");
        return-1;
    }

    ALLEGRO_SAMPLE *sample = al_load_sample("example.
wav");
    if (!sample) {
        fprintf(stderr, "Failed to load sample!\n");
        return-1;
    }

    if (!al_play_sample(sample, 1.0, 0.0, 1.0, ALLEGRO_
PLAYMODE_ONCE, NULL)) {
        fprintf(stderr, "Failed to play sample!\n");
        al_destroy_sample(sample);
        return-1;
    }

    al_rest(5.0); // Wait for 5 seconds so the sound can
play

    al_destroy_sample(sample);
    al_uninstall_audio();

    return 0;
}
```

https://doi.org/10.15215/remix/9781998944224.01

This example initializes Allegro, sets up the audio add-ons, loads an audio sample, plays it, and then waits for five seconds to ensure the sample plays completely before cleaning up and exiting.

Using Files

```
// Include the necessary headers:
#include <allegro5/allegro.h>
#include <allegro5/allegro_native_dialog.h>
#include <cstring>
#include <cstdio>

// Initialize Allegro:
if (!al_init()) {
    al_show_native_message_box(NULL, "Error",
"Error", "Failed to initialize Allegro!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
    return-1;
}

// Open a file for writing:
ALLEGRO_FILE *file = al_fopen("example.txt", "w");
if (!file) {
    al_show_native_message_box(NULL, "Error",
"Error", "Failed to open file for writing!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
    return-1;
}

// Write to the file:
const char *text = "Hello, Allegro File I/O!";
al_fwrite(file, text, strlen(text));

// Close the file:
al_fclose(file);

// Open the file for reading:
file = al_fopen("example.txt", "r");
if (!file) {
    al_show_native_message_box(NULL, "Error",
"Error", "Failed to open file for reading!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    return-1;
}

// Read from the file:
char buffer;
al_fread(file, buffer, sizeof(buffer)-1);
buffer[sizeof(buffer)-1 ] = '\0'; // Ensure
null-termination
printf("Read from file-%s\n", buffer);

// Close the file:
al_fclose(file);
```

Here's the complete code put together:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_native_dialog.h>
#include <cstring>
#include <cstdio>

int main() {
    if (!al_init()) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to initialize Allegro!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        return-1;
    }

    // Initialize native dialog addon
    al_init_native_dialog_addon();

    ALLEGRO_FILE *file = al_fopen("example.txt", "w");
    if (!file) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to open file for writing!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        return-1;
    }

    const char *text = "Hello, Allegro File I/O!";
    al_fwrite(file, text, strlen(text));
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_fclose(file);

    file = al_fopen("example.txt", "r");
    if (!file) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to open file for reading!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        return-1;
    }

    // Read up to 255 bytes and null-terminate
    char buffer[256] = {0};  // initialize to zero
    size_t bytesRead = al_fread(file, buffer,
sizeof(buffer)-1);
    buffer[bytesRead] = '\0'; // ensure null
termination

    printf("Read from file-%s\n", buffer);
    al_fclose(file);

    return 0;
}
```

This example initializes Allegro, opens a file for writing, writes a string to the file, closes it, then reopens the file for reading, reads the content, and prints it to the console. It also includes error handling to display messages if any operation fails.

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Some projects in this section of each chapter, including those using axis-aligned bounding boxes (AABB), extend slightly beyond the textbook's core coverage. They are included to introduce widely used techniques in practical game development and to provide learners with opportunities for further exploration.

https://doi.org/10.15215/remix/9781998944224.01

Beginner Projects (Setup and Basics)

1. **IDE Setup Assignment**
   Set up Allegro 5 with Visual Studio Code, CMake, and Ninja on your operating system (Windows/Linux/macOS). Submit screenshots of a successful "Hello, Allegro!" program.
2. **First Game Window**
   Create a windowed application using Allegro 5 that displays a coloured background and closes when the user presses the Esc key.
3. **Event Handling**
   Write a program that prints keyboard input (e.g., "A pressed") to the console and exits when the window is closed.
4. **Simple Animation**
   Draw a bouncing ball using Allegro primitives (circle) that moves horizontally across the screen and reverses direction at the edges.

Intermediate Projects (Game Mechanics)

5. **Pong Clone**
   Implement a two-player Pong game with paddles, a moving ball, and score tracking. Use keyboard input for controls.
6. **Memory-Matching Game**
   Create a grid of cards (rectangles) that flip on click. Players must match pairs. Track time and moves.
7. **Platformer Movement**
   Design a player sprite that can jump, move left/right, and collide with platforms (rectangles). Use gravity for realistic jumps.
8. **Tile-Based Map**
   Load a CSV file representing a tile map (e.g., 0 = grass, 1 = water) and render it using Allegro bitmaps.
9. **Collision Detection**
   Implement axis-aligned bounding box (AABB) collision detection between a player sprite and obstacles. Display collision feedback (e.g., colour change).
10. **Simple RPG Inventory**
    Create an inventory system where players can collect items (displayed as icons) and toggle visibility with the I key.
11. **Alarm Clock with Graphical User Interface (GUI)**
    Build a digital clock with start/stop/reset buttons. Use Allegro's timer and font modules to display time.

https://doi.org/10.15215/remix/9781998944224.01

Advanced Projects (Complex Systems)

12. **Top-Down Shooter**
    Develop a game where a player-controlled spaceship shoots projectiles at enemies. Include health bars and sound effects.
13. **Procedural Level Generation**
    Generate random platformer levels using algorithms (e.g., Perlin noise) and render them with Allegro.
14. **Pathfinding AI**
    Implement A* pathfinding for an NPC to navigate a grid-based maze. Visualize the path with coloured tiles.
15. **Particle System**
    Create a fire or explosion effect using particles with varying velocity, size, and transparency.
16. **Multiplayer Quiz Game**
    Use networking (optional: local sockets) to let two players answer trivia questions. Track scores in real time.
17. **Save/Load System**
    Save player progress (e.g., position, inventory) to a file and reload it on start-up. Use Allegro's file I/O functions.
18. **Dynamic UI**
    Design a main menu with clickable buttons (e.g., Play, Settings, Exit). Add hover effects and transitions.
19. **Music Sequencer**
    Build a drum machine with interactive buttons that trigger sound samples. Include a tempo slider.

Theoretical and Research Assignments

20. **Game Design Document**
    Write a GDD for an original game, detailing mechanics, story, target audience, and technical requirements.
21. **History of Allegro**
    Research and present the evolution of the Allegro library. Compare versions 4 and 5.
22. **Optimization Report**
    Profile a slow Allegro project (e.g., particle system) and propose fixes (e.g., batch rendering, memory pooling).
23. **AI Behavior Analysis**
    Compare finite state machines and behavior trees. Implement an NPC patrol/attack system using one method.

https://doi.org/10.15215/remix/9781998944224.01

24. **Cross-Platform Challenges**
    Discuss the difficulties of porting Allegro 5 games between Windows, Linux, and macOS. Test a sample project on two OS.

Creative Challenges

25. **Interactive Art Tool**
    Build a pixel-art editor with tools (e.g., brush, colour picker) and save/load functionality.
26. **Procedural Music Visualizer**
    Analyze audio input (e.g., microphone) and generate real-time visual effects (e.g., bars, particles) synchronized to the beat.
27. **Physics-Based Puzzle Game**
    Create a game where players stack objects to reach a goal. Use Allegro's physics primitives or implement basic rigid body dynamics.
28. **Roguelike Prototype**
    Design a procedurally generated dungeon crawler with permadeath, loot, and enemy AI. Use turn-based movement.
29. **3D Rendering Experiment**
    Use Allegro's OpenGL integration to render a rotating 3D cube. Add texture mapping and lighting.
30. **Final Project: Complete Game**
    Develop a polished game (e.g., platformer, puzzle, RPG) with menus, sound, scoring, and at least three levels. Present a demo and submit source code.

https://doi.org/10.15215/remix/9781998944224.01

This page intentionally left blank

2

Essentials of Game Programming with Allegro

2.1. Common Structure of a C/C++ Program with Allegro 5

As a tradition and common practice, we will create a "Hello World!" program to introduce programming with Allegro 5 in C/C++.

This program will initialize Allegro, create a display window, and draw the text "Hello, World!" on the screen. This can be done in the following steps:

1. Set up Allegro 5 by including the Allegro 5 library so that the library will be installed and used when the program is compiled and linked. Please note that in order to be able to use Allegro 5, you must have set up your integrated development environment (IDE) to make the Allegro 5 library accessible as covered in section 1.5 of the textbook.
2. Initialize Allegro and its components such as display and font, as well as whatever you will use in the program.
3. Create the window to hold the content of the program by defining the dimensions of the window.
4. Load resources, such as fonts for displaying text.
5. Render the text by drawing the text "Hello, World!" on the screen and hold it for users to see.
6. Finish up by destroying the font and display to free up the resources used by the program.

The following is a sample of complete code:

https://doi.org/10.15215/remix/9781998944224.01

```
// setup Allegro, its font and TTF addons and operations
for IO
#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <iostream>

const int SCREEN_WIDTH = 800;
const int SCREEN_HEIGHT = 600;

// function to initialize Allegro.
// All user defined functions used in your program must
be defined before they are used.
void initAllegro() {
    if (!al_init()) { // initialize Allegro
        std::cerr << "Failed to initialize Allegro!" <<
std::endl;
        exit(1);
    }
    if (!al_init_font_addon()) { // initialize font addon
        std::cerr << "Failed to initialize font addon!"
<< std::endl;
        exit(1);
    }
    if (!al_init_ttf_addon()) { // initialize TTF addon
        std::cerr << "Failed to initialize TTF addon!" <<
std::endl;
        exit(1);
    }
    if (!al_install_keyboard()) { // install keyboard
        std::cerr << "Failed to install keyboard!" <<
std::endl;
        exit(1);
    }
}

int main() {
    initAllegro(); // call the defined function
    ALLEGRO_DISPLAY* display = al_create_display(SCREEN_
WIDTH, SCREEN_HEIGHT);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    // the above is to create a window with defined the
width and height
    if (!display) { // always check if the window is
successfully created.
        std::cerr << "Failed to create display!" <<
std::endl;
        return-1;
    }

    ALLEGRO_FONT* font = al_load_ttf_font("arial.ttf",
32, 0);
    if (!font) {
        std::cerr << "Failed to load font!" << std::endl;
        al_destroy_display(display);
        return-1;
    }

    al_clear_to_color(al_map_rgb(0, 0, 0));
    al_draw_text(font, al_map_rgb(255, 255, 255), SCREEN_
WIDTH / 2, SCREEN_HEIGHT / 2, ALLEGRO_ALIGN_CENTER,
"Hello, World!");
    al_flip_display();

    al_rest(5.0); // Display the text for 5 seconds

    al_destroy_font(font);
    al_destroy_display(display);
    return 0;
}
```

To understand the structure of this simple program, let's explain the code section by section.

2.1.1. Header Files

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <iostream>
```

https://doi.org/10.15215/remix/9781998944224.01

Allegro consists of a core library with various functional add-ons. Most games written with Allegro will need to make use of several add-on libraries to handle functions that Allegro's core doesn't. This includes the facility to render text, so for our "Hello World!" program, for an example, we needed to include the Allegro font add-on.

Additionally, here, as we are doing input/output (I/O) file functions, we will use an example of a proper C++ coding style, as seen in the use of std:: in our error-checking line and as in this line:

```
std::cerr << "Failed to initialize Allegro!" <<
std::endl;
```

We will use #include <iostream> rather than #include <cstdio>, as seen previously, as it simplifies the code and provides the functions for both file access and console error displays.

2.1.2. User-Defined Global Constants and Variables

As in all computer programs, all names must be defined before they are used. So in this code example, two constants are to be used, hence they are defined right after the header files:

```
const int SCREEN_WIDTH = 800; // this is used to set the
width of the screen
const int SCREEN_HEIGHT = 600; // this is used to set the
height of the screen
```

The reason for defining these two constants, instead of directly using the numbers 800 and 600, respectively, is to make the code more readable and easy to maintain. For example, the width and height may appear in many places in the program. When the constants are defined and used in place of the numbers in the program, if you want to change the screen size, you only need to change the definitions here. Otherwise, you would have to change the numbers in every individual instance.

2.1.3. User-Defined Functions

When programming to solve a problem or to implement a computing system, the well-known divide-and-conquer approach is commonly used. The first portion of the code sets up the bits of Allegro that are necessary to display the window, show the "Hello World!" text, and then quit when the user presses a

https://doi.org/10.15215/remix/9781998944224.01

key. In the following we will explain piece by piece what is needed to initialize the system for this intended purpose.

al_init();

Allegro has to set up some bare essentials before you use any of its functions in the specific Allegro library, and this is done with al_init(), except in the following cases:

- When using al_install_system() in place of al_init() to initialize the Allegro system. This function allows more control, such as specifying a custom atexit function.
- When working with shared libraries. In this case, it's generally advised not to call al_init() within the library itself. Instead, the application using the library should handle the initialization.
- When using a preinitialized system. If another part of your application or another library has already initialized Allegro, calling al_init() again is unnecessary and could cause issues.
- When using native dialogue functions provided by Allegro 5, such as:

```
al_show_native_message_box
```

al_install_keyboard();

al_install_keyboard() enables keyboard input. Believe it or not, accepting keyboard input to your program isn't mandatory!

ALLEGRO_DISPLAY disp = al_create_display(320, 200);*

al_create_display() tells Allegro to create an 800 × 600 pixel window. You may change the numbers, then recompile to view the change in size.

2.1.4. The Main Function

For the simple task of displaying "Hello World!," the main function of the program can be laid out as follows.

Initialize the display:

```
int main() {
    initAllegro(); // call the defined function
    ALLEGRO_DISPLAY* display = al_create_display(SCREEN_
WIDTH, SCREEN_HEIGHT);
    // the above is to create a window with defined the
width and height
```

https://doi.org/10.15215/remix/9781998944224.01

```
    if (!display) { // always check if the window is
successfully created.
        std::cerr << "Failed to create display!" <<
std::endl;
        return-1;
    }
```

Load the required font for the text output:

```
    ALLEGRO_FONT* font = al_load_ttf_font("arial.ttf",
32, 0);
    if (!font) {
        std::cerr << "Failed to load font!" << std::endl;
        al_destroy_display(display);
        return-1;
    }
```

Reset the screens colourmap (this is covered later in the textbook, so don't worry about it now):

```
    al_clear_to_color(al_map_rgb(0, 0, 0));
```

Draw and display the message for five seconds:

```
    al_draw_text(font, al_map_rgb(255, 255, 255), SCREEN_
WIDTH / 2, SCREEN_HEIGHT / 2, ALLEGRO_ALIGN_CENTER,
"Hello, World!");
    al_flip_display();

    al_rest(5.0); // Display the text for 5 seconds
```

Properly shut down the program by destroying the fonts and window used to release the resources for other uses. This is done with the following:

```
    al_destroy_font(font);
    al_destroy_display(display);
    return 0;
}
```

https://doi.org/10.15215/remix/9781998944224.01

2.2. Handling User Input

The "Hello World!" example doesn't take any input. In most programs, especially video games, however, handling user input is necessary.

2.2.1. Keyboard

Among the input devices, the keyboard is the most commonly used, especially for video games. Games running on computers may often use joysticks for a better gameplay experience, but the game programs must handle user input from a keyboard because almost all computers have a keyboard attached. This section covers basic keyboard input handling using Allegro 5 for games or interactive applications.

Setup for Handling Keyboard Input

Firstly, include necessary headers and initialize the keyboard subsystem. For keyboard, only allegro5/allegro.h is needed. (The same is not true of mouse and joystick, as you will see later.) Here is the example code:

```
#include <allegro5/allegro.h>

int main() {
    al_init(); // Initialize Allegro
    al_install_keyboard(); // Enable keyboard input
    ALLEGRO_EVENT_QUEUE* event_queue =
al_create_event_queue();
    al_register_event_source(event_queue,
al_get_keyboard_event_source());
    //...(create display, timers, etc.)
}
```

Event-Based Input

In Allegro 5, event-based input refers to handling input through an event-driven system. This means that instead of continuously checking the state of input devices, the program responds to events as they occur. Here's how it works:

Event Queue: An event queue is used to store events generated by various input devices (keyboard, mouse, joystick, etc.).

You create an event queue using al_create_event_queue() and register event sources (like the keyboard or display) with al_register_event_source().

https://doi.org/10.15215/remix/9781998944224.01

Event Types: Allegro defines various event types, such as ALLEGRO_EVENT_KEY_DOWN for key presses, ALLEGRO_EVENT_MOUSE_BUTTON_DOWN for mouse clicks, and ALLEGRO_EVENT_DISPLAY_CLOSE for window-close events.

Event Loop: The event loop continuously checks for events in the event queue using functions like al_get_next_event() or al_wait_for_event(). When an event is detected, the program processes it accordingly.

The following is a simple example demonstrating event-based input handling in Allegro 5. Note here we are not error checking the loading of our libraries, as you saw in some of the chapter 1 code examples. Here, keeping with the minimal concept of a demo program (it is a simple example after all . . .), we'll exit the program on an error. You'll see this in some of the other code examples in this text, as we're not creating production code, if you will, but only quick examples for the student. As a matter of fact, in the example after this one, we won't even have exit code to check for initialization; we'll leave that as an exercise for the student to determine where the appropriate code would go. In production or release code, however, note that it's *always* good practice to include error checking in your code and exit gracefully with a warning message or error display of some kind. Here is the example code:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_primitives.h>

struct Point { float x, y; };

int main() {
    if (!al_init()) return-1;
    if (!al_install_keyboard()) return-1;
    if (!al_install_mouse()) return-1;
    if (!al_init_primitives_addon()) return-1;

    ALLEGRO_DISPLAY* display = al_create_display(800,
600);
    if (!display) return-1;

    ALLEGRO_EVENT_QUEUE* queue = al_create_event_queue();
    if (!queue) { al_destroy_display(display); return-1;
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_register_event_source(queue,
al_get_display_event_source(display));
    al_register_event_source(queue,
al_get_keyboard_event_source());
    al_register_event_source(queue,
al_get_mouse_event_source());

    bool running = true;
    bool has_click = false;
    Point last_click = {0, 0};

    while (running) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(queue, &ev);

        if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            running = false;
        } else if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
            if (ev.keyboard.keycode == ALLEGRO_KEY_
ESCAPE) running = false;
        } else if (ev.type == ALLEGRO_EVENT_MOUSE_BUTTON_
DOWN) {
            if (ev.mouse.button == 1) { // left mouse
button
                last_click = { (float)ev.mouse.x, (float)
ev.mouse.y };
                has_click = true;
            }
        }

        // Draw the frame, draw a red circle where the
mouse is clicked
        al_clear_to_color(al_map_rgb(0, 0, 0));
        if (has_click) {
            al_draw_filled_circle(last_click.x, last_
click.y, 20.0f, al_map_rgb(255, 0, 0));
        }
        al_flip_display();
    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_destroy_event_queue(queue);
    al_destroy_display(display);
    return 0;
}
```

Explanation

Initialization: Initializes Allegro and its add-ons for keyboard, mouse, and primitives.

Display and Event Queue: Creates a display and an event queue, then registers the display, keyboard, and mouse as event sources.

Event Loop: Continuously waits for events and processes them. For example, it exits the loop if the display is closed or the Esc key is pressed.

Drawing: Clears the screen and draws a red circle where the mouse is clicked, then updates the display.

Event-based input handling is efficient and allows your program to respond to user actions in real time.

State-Based Input (Held Keys)

In Allegro 5, state-based input refers to a method of handling input where the current state of input devices (like the keyboard or mouse) is captured and stored at a specific point in time. This allows you to check the status of keys or buttons at any moment rather than relying on event-driven input alone.

The following code checks if a key is currently held down (e.g., for movement)—here, the left and right arrow keys—and then will move a red square left or right accordingly:

```
ALLEGRO_KEYBOARD_STATE key_state;
al_get_keyboard_state(&key_state);

// Move player if arrow keys are held
if (al_key_down(&key_state, ALLEGRO_KEY_RIGHT)) {
    player_x += 5;
}
if (al_key_down(&key_state, ALLEGRO_KEY_LEFT)) {
    player_x-= 5;
}
'''
```

https://doi.org/10.15215/remix/9781998944224.01

Table 1: Common Keycodes

Key	Allegro Constant
Arrow Keys	ALLEGRO_KEY_LEFT ALLEGRO_KEY_RIGHT ALLEGRO_KEY_UP ALLEGRO_KEY_DOWN
WASD	ALLEGRO_KEY_W ALLEGRO_KEY_A ALLEGRO_KEY_S ALLEGRO_KEY_D
Space/Enter	ALLEGRO_KEY_SPACE ALLEGRO_KEY_ENTER
Modifiers	ALLEGRO_KEY_SHIFT ALLEGRO_KEY_CTRL

COMPLETE CODING EXAMPLE

```
#include <allegro5/allegro.h>

int main() {
    al_init();
    al_install_keyboard();
    ALLEGRO_DISPLAY* display = al_create_display(800,
600);
    ALLEGRO_EVENT_QUEUE* event_queue =
al_create_event_queue();
    al_register_event_source(event_queue,
al_get_keyboard_event_source());

    bool game_running = true;
    float x = 400, y = 300;

    while (game_running) {
        ALLEGRO_EVENT event;
        al_wait_for_event(event_queue, &event);

        // Event-based exit (ESC)
        if (event.type == ALLEGRO_EVENT_KEY_DOWN) {
            if (event.keyboard.keycode == ALLEGRO_KEY_
ESCAPE) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
                game_running = false;
            }
        }

        // State-based movement (arrow keys)
        ALLEGRO_KEYBOARD_STATE state;
        al_get_keyboard_state(&state);
        if (al_key_down(&state, ALLEGRO_KEY_RIGHT)) x +=
2;
        if (al_key_down(&state, ALLEGRO_KEY_LEFT)) x-= 2;

        // Draw
        al_clear_to_color(al_map_rgb(0, 0, 0));
        al_draw_filled_rectangle(x, y, x+50, y+50, al_
map_rgb(255, 0, 0));
        al_flip_display();
    }

    al_destroy_display(display);
    return 0;
}
```

Tips for Handling User Input

- Use event-based input for single actions (e.g., menus, jumping).
- Use state-based input for continuous movement.
- Always call al_get_keyboard_state() before checking key states.
- Clean up resources with al_uninstall_keyboard() at exit.

With the knowledge we have just learned about handling keyboard input, we now can move to our next code example, which creates a bouncing ball simulation, with position displayed on the upper-right corner of the window. The ball stays within a container box and the program exits when Q is pressed. This will draw a colourful square on the screen and hold it until the key is pressed:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <allegro5/allegro_primitives.h>
#include <sstream>
```

https://doi.org/10.15215/remix/9781998944224.01

```
const int SCREEN_W = 800;
const int SCREEN_H = 600;
const int BOX_PADDING = 20;
const int BALL_RADIUS = 15;

int main() {
    // Initialize Allegro and components
    al_init();
    al_init_font_addon();
    al_init_ttf_addon();
    al_init_primitives_addon();
    al_install_keyboard();

    // Create display and set up
    ALLEGRO_DISPLAY* display = al_create_
display(SCREEN_W, SCREEN_H);
    ALLEGRO_TIMER* timer = al_create_timer(1.0 / 60);
    ALLEGRO_EVENT_QUEUE* event_queue =
al_create_event_queue();
    ALLEGRO_FONT* font = al_create_builtin_font();

    // Register event sources
    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_timer_event_source(timer));
    al_register_event_source(event_queue,
al_get_keyboard_event_source());

    // Ball properties
    float ball_x = SCREEN_W / 2;
    float ball_y = SCREEN_H / 2;
    float dx = 4, dy = 4;
    bool done = false;
    bool redraw = true;

    al_start_timer(timer);

    // Main loop
    while (!done) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
        ALLEGRO_EVENT event;
        al_wait_for_event(event_queue, &event);

        switch (event.type) {
            case ALLEGRO_EVENT_TIMER:
                // Update ball position
                ball_x += dx;
                ball_y += dy;

                // Collision with container box
                if (ball_x-BALL_RADIUS < BOX_PADDING ||
                    ball_x + BALL_RADIUS >
SCREEN_W-BOX_PADDING) {
                    dx =-dx;
                }
                if (ball_y-BALL_RADIUS < BOX_PADDING ||
                    ball_y + BALL_RADIUS >
SCREEN_H-BOX_PADDING) {
                    dy =-dy;
                }
                redraw = true;
                break;

            case ALLEGRO_EVENT_KEY_DOWN:
                if (event.keyboard.keycode == ALLEGRO_
KEY_Q) {
                    done = true;
                }
                break;

            case ALLEGRO_EVENT_DISPLAY_CLOSE:
                done = true;
                break;
        }

        if (redraw && al_is_event_queue_empty(event_
queue)) {
            redraw = false;
            // Drawing
            al_clear_to_color(al_map_rgb(0, 0, 0));
```

https://doi.org/10.15215/remix/9781998944224.01

```
            // Draw container box
            al_draw_rectangle(BOX_PADDING, BOX_PADDING,
                              SCREEN_W-BOX_PADDING,
SCREEN_H-BOX_PADDING,
                              al_map_rgb(255, 255, 255),
2);

            // Draw ball
            al_draw_filled_circle(ball_x, ball_y,
BALL_RADIUS,
                                  al_map_rgb(255, 0, 0));

            // Display position text
            al_draw_textf(font, al_map_rgb(255, 255,
255),
                          SCREEN_W-120, 10,
ALLEGRO_ALIGN_LEFT,
                          "X: %.1f Y: %.1f", ball_x,
ball_y);

            al_flip_display();
        }
    }

    // Cleanup
    al_destroy_font(font);
    al_destroy_timer(timer);
    al_destroy_display(display);
    al_destroy_event_queue(event_queue);

    return 0;
}
```

We need the fonts and primitives add-ons now. As long as your IDE has been set up by following the instructions given in the previous section, you can now directly build with VS Code, or compile the program with Allegro dependencies in a terminal:

```
g++ -o bouncing_ball bouncing_ball.cpp -lallegro
-lallegro_font -lallegro_ttf -lallegro_primitives
```

https://doi.org/10.15215/remix/9781998944224.01

Compile and run the program. You should see a screen with a bouncing object, a boundary box outline, and some text presenting the x and y position variables.

2.2.2. Mouse

Mice are essential peripherals in computer gaming, offering precision and control across various game genres. Here's a closer look at their role and features.

Types of Gaming Mice

Standard Gaming Mice: These are versatile and suitable for a wide range of games, from first-person shooter (FPS) to real-time strategy (RTS) games.

MMO Mice: Designed for massively multiplayer online games, these mice often have multiple programmable buttons to handle complex commands.

FPS Mice: These are optimized for first-person shooters, featuring high-DPI (dots per inch) settings for precise aiming and quick movements.

Ambidextrous Mice: Suitable for both left- and right-handed users, these mice offer symmetrical designs.

Key Features

DPI (Dots Per Inch): Higher DPI settings allow for more sensitive and precise movements, which is crucial in fast-paced games.

Polling Rate: This measures how often the mouse reports its position to the computer. A higher polling rate (e.g., 000 Hz) means more responsive performance.

Programmable Buttons: Extra buttons can be customized for specific in-game actions, enhancing gameplay efficiency.

Ergonomics: Comfortable design is essential for long gaming sessions to prevent strain and injury.

Popular Uses in Games

First-Person Shooter (FPS): Precision and quick response times are critical, making high-DPI and low-latency mice ideal.

Real-Time Strategy (RTS): Accurate clicking and multiple programmable buttons help manage complex commands and unit control.

Massively Multiplayer Online (MMO): Extra buttons and macros are useful for managing numerous abilities and commands.

https://doi.org/10.15215/remix/9781998944224.01

Top Gaming Mice in 2024

Logitech G Pro X Superlight: Known for its lightweight design and high performance, it's a favorite among professional gamers.

Razer DeathAdder V3 Pro: Offers excellent ergonomics and precision, making it ideal for long gaming sessions.

SteelSeries Rival 600: Features dual sensors for enhanced accuracy and customizable weights for a personalized feel.

Mouse-Only Games

There are also many games designed to be played primarily with a mouse. These include point-and-click adventures, puzzle games, and idle clickers. Websites like CrazyGames and Poki offer a variety of mouse-only games that you can enjoy.

Generally speaking, there are two ways you can choose to interpret mouse movement:

1. Moving the mouse pointer and clicking around as you normally would.
2. Using the *speed* of the mouse to control something in the game. This allows you to use the mouse a bit like a joystick; most PC-based first-person shooters do this to allow the player to speedily aim.

Steps to Use Mouse in Allegro Games

To use the mouse in Allegro games, we will need to follow a few steps to initialize and handle mouse input. Here's a basic guide to get you started:

1. **Initialize the Mouse**
   First, you need to install the mouse handler using al_install_mouse(). This function sets up the mouse for use in your game:

```
   if (!al_install_mouse()) {
       fprintf(stderr, "Failed to initialize the
mouse!\n");
       return–;
   }
```

2. **Check Mouse State**
   You can retrieve the current state of the mouse using al_get_mouse_state(). This function fills an ALLEGRO_MOUSE_STATE structure with the current mouse position and button states:

https://doi.org/10.15215/remix/9781998944224.01

```
ALLEGRO_MOUSE_STATE state;
al_get_mouse_state(&state);
```

3. **Handle Mouse Input**
   You can check the position and button states from the ALLEGRO_MOUSE_STATE structure. For example, to get the mouse coordinates and check if the left button is pressed, use this:

```
int mouse_x = state.x;
int mouse_y = state.y;
if (al_mouse_button_down(&state, )) {
    // Left mouse button is pressed
}
```

4. **Display Mouse Cursor**
   To show the mouse cursor on the screen, use al_show_mouse_cursor(display), where display is your Allegro display:

```
al_show_mouse_cursor(display);
```

2.2.3. Joystick

Joysticks are versatile input devices used in various types of games, especially those that require precise control, such as flight simulators, space shooters, and racing games. Here's a bit more about them:

Types of Joysticks

Digital Joysticks: These are the simplest form, where the joystick can be moved in four or eight directions. They are often used in arcade games.

Analog Joysticks: These provide a range of motion and are more precise, making them ideal for flight simulators and racing games.

HOTAS (Hands-On Throttle-And-Stick): These are advanced setups that include a joystick and a separate throttle control, often used in flight simulators for a more immersive experience.

Key Features of Joysticks

Buttons: Joysticks typically have multiple buttons that can be programmed for different functions in a game.

Hat Switch: A small joystick on top of the main stick used for looking around in a game.

https://doi.org/10.15215/remix/9781998944224.01

Throttle Control: Found in HOTAS setups, it allows for precise control of speed in flight simulators.

Popular Uses in Games

Flight Simulators: Joysticks provide the most realistic control for flying aircraft in games like *Microsoft Flight Simulator*.

Space Simulators: Games like *Elite Dangerous* and *Star Citizen* benefit from the precise control offered by joysticks.

Arcade Games: Classic arcade games often use digital joysticks for their simple and responsive controls.

Best Joysticks for Video Games in 2024

Thrustmaster HOTAS Warthog: Known for its build quality and precision, it's a favorite among flight sim enthusiasts.

Logitech G X56 HOTAS RGB: Offers a good balance between features and price, suitable for both beginners and experienced users.

Thrustmaster T.Flight HOTAS X: A budget-friendly option that still provides a solid experience.

Joysticks enhance the gaming experience by providing more intuitive and precise control, especially in games that simulate real-world activities.

Steps to Use a Joystick in Your Allegro 5 Games

To use a joystick in your Allegro 5 games requires the following steps:

1. **Initialize the Joystick**
   - Include the Allegro joystick header—#include <allegro5/allegro.h>
   - Install the joystick driver using al_install_joystick().
   - Check if the joystick is installed with al_is_joystick_installed().
2. **Get Joystick Information**
   - Use al_get_num_joysticks() to find out how many joysticks are connected.
   - Retrieve a joystick object with al_get_joystick().
3. **Read Joystick State**
   - Create an ALLEGRO_JOYSTICK_STATE object.
   - Use al_get_joystick_state() to update the state of the joystick.
4. **Handle Joystick Events**
   - Register the joystick event source with al_get_joystick_event_source().

https://doi.org/10.15215/remix/9781998944224.01

- Use an event queue to handle joystick events like button presses and axis movements.

Here's a simple example to get you started:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_joystick.h>
#include <stdio.h>

int main() {
    al_init();
    al_install_joystick();

    if (!al_is_joystick_installed()) {
        printf("Joystick not installed!\n");
        return-1;
    }

    ALLEGRO_JOYSTICK *joystick = al_get_joystick(0);
    if (!joystick) {
        printf("No joystick found!\n");
        return-1;
    }

    ALLEGRO_JOYSTICK_STATE state;
    al_get_joystick_state(joystick, &state);

    printf("Joystick name-%s\n",
al_get_joystick_name(joystick));
    printf("Number of sticks-%d\n",
al_get_joystick_num_sticks(joystick));
    printf("Number of buttons-%d\n",
al_get_joystick_num_buttons(joystick));

    // Example of reading joystick state
    while (true) {
        al_get_joystick_state(joystick, &state);
        if (state.button[0]) {
            printf("Button 0 pressed!\n");
```

https://doi.org/10.15215/remix/9781998944224.01

```
        }
    }

    al_uninstall_joystick();
    return 0;
}
```

2.3. Collision Detection

Collision detection is a crucial aspect of video games, ensuring that objects within the game world interact realistically. Collision detection is the computational process of determining when two or more objects in a game intersect or come into contact. This is essential for creating realistic interactions, such as a character bumping into a wall or a projectile hitting a target.

Collision detection is a cornerstone of action game development, ensuring that game objects interact logically within the game world. It's involved in constraining a player's actions in some cases (that wall you can't go through or objects you find that you can interact with); other times, it triggers cause and effect (the missile hits my ship and I go boom). From a programming perspective, it is the code that determines whether two or more objects occupy the same space at a given time, triggering responses like bouncing, damage, or other interactions. Accurate collision detection contributes to gameplay realism and is critical for engaging user experiences.

2.3.1. Types of Collision Detection

Collision detection methods can range from simple bounding boxes—an area around your sprite, like its rectangular cell borders or a circle emanating from the sprite's center—to more advanced algorithms for complex shapes, in which case edges will be the key components for collision detection.

1. **Axis-Aligned Bounding Box (AABB)**
   This method involves checking if the bounding boxes of two objects overlap. It's simple and efficient, especially for rectangular objects that are not rotated.

```
bool check_collision(AABB a, AABB b) {
    return (a.x < b.x + b.width &&
            a.x + a.width > b.x &&
            a.y < b.y + b.height &&
```

https://doi.org/10.15215/remix/9781998944224.01

```
                a.y + a.height > b.y);
}
```

This method uses rectangles or squares to approximate an object's boundaries, checking for overlap between them. This check for overlap in its most basic form is for a complete overlap—that is, the whole object—but can be modified to check for a single side only.

In figure 2.1, two cubes, labelled "Object A" and "Object B," overlap. The area of overlap is red, and an arrow points to it labelled "Collision Detected." This approach is computationally simple but can lead to false positives for irregular shapes. The context here is an area defined by the sprites' cell size. Take, for example, the flying saucer in figure 2.2. The red area indicates the bitmap image size (we deal with a lot of square and rectangular images by default in computer graphics/photos). The bounding box for that bitmap then would include all the area indicated by the green outline in the image. This would cause inconsistencies in collision or contacts: a laser beam, for example, could go right over the top dome of the saucer in the red area, but the collision detection (the hit) would be as soon as it contacted the red area surrounding the saucer.

2. **Circle Collision**
   For circular objects, collision detection can be done by checking the distance between their centers. If the distance is less than the sum of their radii, a collision has occurred.

```
bool check_circle_collision(Circle a, Circle b) {
    float dx = a.x–b.x;
    float dy = a.y–b.y;
    float distance = sqrt(dx * dx + dy * dy);
    return distance < (a.radius + b.radius);
}
```

Here a case for a circle might be better, but as you see in figure 2.3, it's still not perfect, and there would be the extra math to determine the edge of the circle. What this goes to show, though, is sometimes when creating your graphics, you might want to consider their design or their size to help with these issues. So, for example, I could have made the bitmap more rectangular, like in figure 2.4. In that case, the rectangular bounding box would be somewhat more accurate, and we could still utilize the simpler code.

https://doi.org/10.15215/remix/9781998944224.01

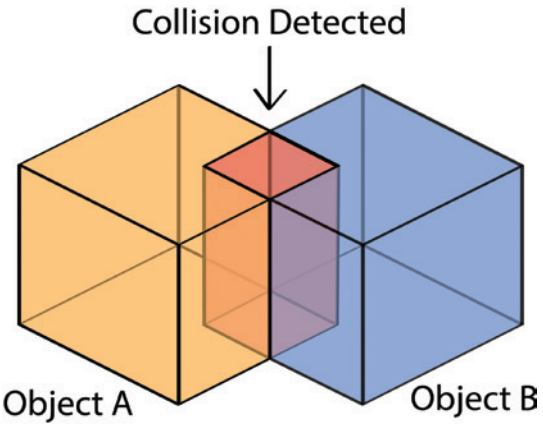

Figure 2.1: Collision detection using bounding boxes. Illustrated by Kyle Flemmer.

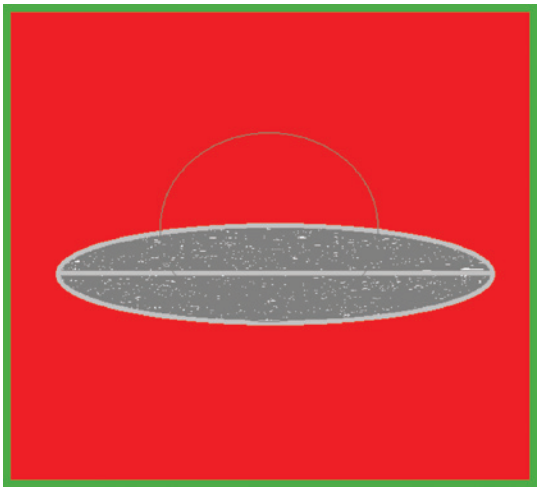

Figure 2.2: Saucer image with a green bounding box. Illustrated by Walter Ridgewell.

3. **Separating Axis Theorem (SAT)**
   This method is used for detecting collisions between convex polygons. It involves projecting the vertices of the polygons onto various axes and checking for overlaps. It's more complex but very powerful.

2.3.2. Phases of Collision Detection

1. **Broad Phase**
   The broad phase quickly eliminates pairs of objects that cannot possibly collide. Techniques like spatial partitioning (e.g., grids, quadtrees) or bounding volume hierarchies are used to reduce the number of collision checks.

https://doi.org/10.15215/remix/9781998944224.01

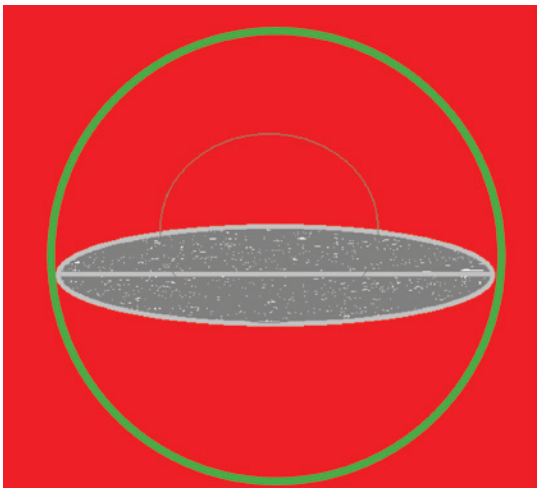

Figure 2.3: Saucer image with a round bounding box. Illustrated by Walter Ridgewell.

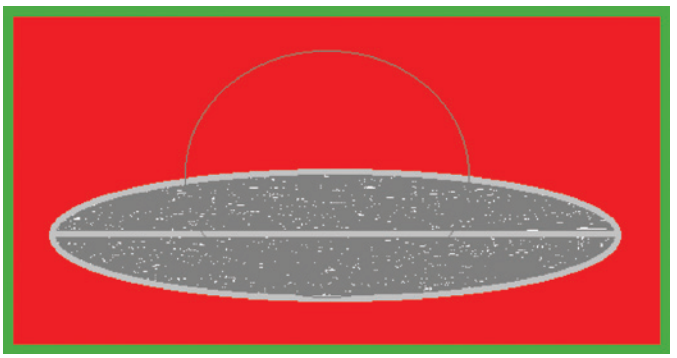

Figure 2.4: Saucer image cropped with a rectangular bounding box. Illustrated by Walter Ridgewell.

2. **Narrow Phase**
   The narrow phase performs detailed collision checks on the remaining pairs of objects. This phase uses precise algorithms to determine if and where the objects intersect.

2.3.3. Implementing Collision Detection in Allegro

Here's a simple example of implementing AABB collision detection in an Allegro game:

```
// Initialize Allegro and Create Objects
   al_init();
   al_init_image_addon();
```

https://doi.org/10.15215/remix/9781998944224.01

```
   ALLEGRO_DISPLAY *display = al_create_display(800,
600);
   if (!display) {
       fprintf(stderr, "Failed to create display!\n");
       return–;
   }

   ALLEGRO_BITMAP *sprite = al_load_bitmap("sprite.png");
   ALLEGRO_BITMAP *sprite2 = al_load_bitmap("sprite2.
png");
// Define AABB Structure and Collision Function
   typedef struct {
       float x, y, width, height;
   } AABB;

   bool check_collision(AABB a, AABB b) {
       return (a.x < b.x + b.width &&
               a.x + a.width > b.x &&
               a.y < b.y + b.height &&
               a.y + a.height > b.y);
   }
// Game Loop with Collision Detection
   AABB box = {00, 00, al_get_bitmap_width(sprite),
al_get_bitmap_height(sprite)};
   AABB box2 = {200, 200, al_get_bitmap_width(sprite2),
al_get_bitmap_height(sprite2)};

   while (true) {
       // Update positions (example)
       box.x += .0;
       box2.y–= .0;

       // Check for collision
       if (check_collision(box, box2)) {
           printf("Collision detected!\n");
       }

       // Render
       al_clear_to_color(al_map_rgb(0, 0, 0));
       al_draw_bitmap(sprite, box.x, box.y, 0);
```

https://doi.org/10.15215/remix/9781998944224.01

```
        al_draw_bitmap(sprite2, box2.x, box2.y, 0);
        al_flip_display();
        al_rest(0.06); // Approximately 60 FPS
    }
```

Since many games will need to utilize collision in some way, let's look at another example.

The CollisionTest Program

The CollisionTest program demonstrates basic collision detection principles by simulating interactions between two game objects. The setup involves:

Sprites: These are visual representations of objects; we will use simple squares.

Movement and Interaction: The program detects collisions as sprites move, triggering predefined responses such as bouncing or stopping.

As with previous code examples, we have the usual basic constructs of code: we have our setup and initialization phase: We load sprites, set initial positions, and define movement parameters.

We then have our game loop functions, which here will do the following:

- Check for collisions in every frame.
- Perform a collision response—that is, execute an appropriate reaction, such as reversing direction or displaying an effect. For our demonstration program, we will simply change the direction of the sprite objects after they collide (here a complete overlap) and print a message on the screen.

The primary element of code we need to examine here is of course the collision detection function. We begin with the Sprite struct, which stores the following information elements of the Sprite object (see table 2):

```
typedef struct Sprite {
    float x, y, width, height;
    float dx, dy; // velocity
} Sprite
```

Using these variables, we can create a function to examine the location of two sprites to see if their edges are touching or overlapping. Imagine in our

https://doi.org/10.15215/remix/9781998944224.01

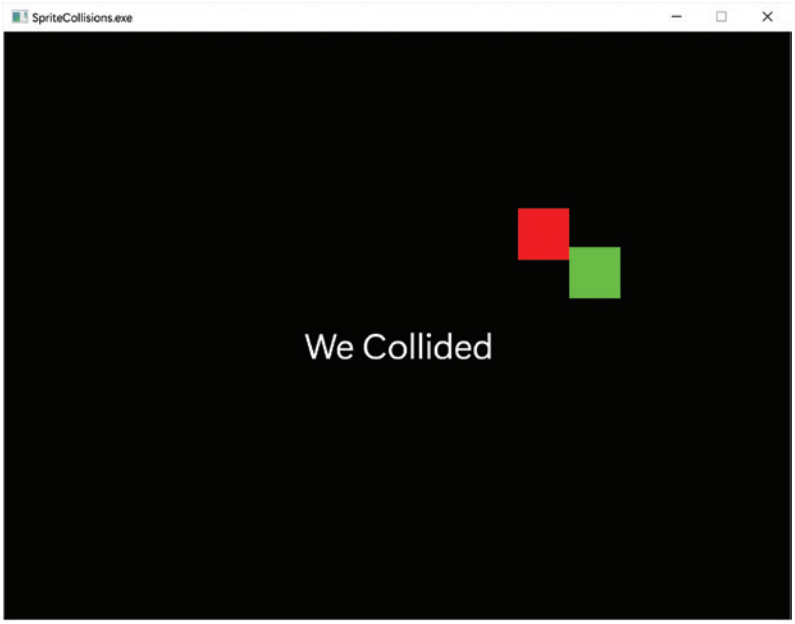

Figure 2.5: Collision response with a print message. Image by Walter Ridgewell.

Table 2: Sprite Struct Variables

Variable	Purpose
x, y	The **position** of the sprite on the screen (top-left corner).
width, height	The **size** (dimensions) of the sprite.
dx, dy	The **velocity** (change in position per frame).

game we have a sprite at position (100, 150) on our screen with size (50 × 50) and velocity (2, 3). The memory representation of that Sprite struct information is then as shown in table 3.

In each iteration of our game loop, movement-wise, this sprite—we'll call it Sprite A—will move 2 pixels right and 3 pixels down per animation frame.

Explaining the check_collision Function

We talked previously about what a bounding box collision was, and since our simple example is using basic shapes (squares), we will implement one here. Our check collision function looks like this:

https://doi.org/10.15215/remix/9781998944224.01

Table 3: Sprite Struct Properties

Property	Value
x	100
y	150
width	50
height	50
dx	2
dy	3

```
bool check_collision(Sprite *a, Sprite *b) {
    return (a->x < b->x + b->width &&
            a->x + a->width > b->x &&
            a->y < b->y + b->height &&
            a->y + a->height > b->y);
}
```

This function checks if two sprites overlap (collide) on the screen. It uses axis—aligned bounding box (AABB) collision detection, which is a fast method for detecting rectangle-based collisions and returns a simple true or false condition. Note the use of the logical AND operator (&&) in the return logic. If all four conditions are true, then the sprites overlap, and the collision detection function returns a "true" condition.

Understanding the Conditions

The function checks if the bounding boxes of two sprites overlap by looking at the edges of the two objects:

1. *a->x < b->x + b->width*
   Ensures **Sprite A's left edge** is to the **left of Sprite B's right edge**.
2. *a->x + a->width > b->x*
   Ensures **Sprite A's right edge** is to the **right of Sprite B's left edge**.
3. *a->y < b->y + b->height*
   Ensures **Sprite A's top edge** is **above Sprite B's bottom edge**.
4. *a->y + a->height > b->y*
   Ensures **Sprite A's bottom edge** is **below Sprite B's top edge**.

https://doi.org/10.15215/remix/9781998944224.01

2.3.4. Graphical Explanation Overlap Collision

Now at this point we would have, status-wise,

a->x < b->x + b->width → **True**
a->x + a->width > b->x → **False**
a->y < b->y + b->height → **True**
a->y + a->height > b->y → **False**

But our Sprite A was moving diagonally downward, and Sprite B is not moving. So let's assume the movement path creates an imminent collision event.

With a complete overlap of Sprite B by Sprite A, the conditions then change to

a->x < b->x + b->width → **True**
a->x + a->width > b->x → **True**
a->y < b->y + b->height → **True**
a->y + a->height > b->y → **True**

Since all conditions are now true, a collision is detected, and our function returns true.

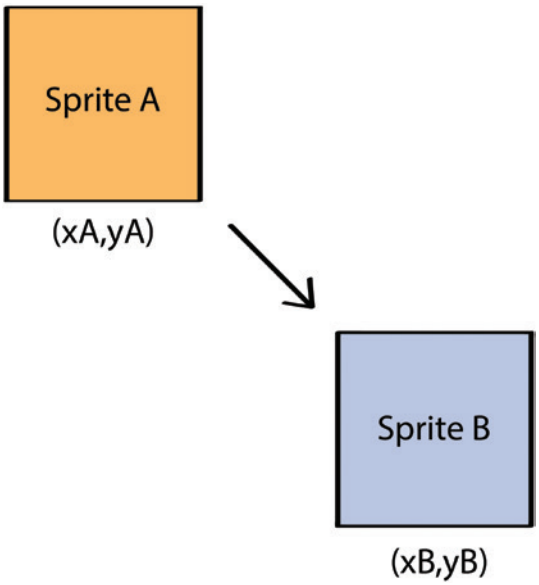

Figure 2.6: Sprites positioned apart: No collision detected. Illustrated by Kyle Flemmer.

https://doi.org/10.15215/remix/9781998944224.01

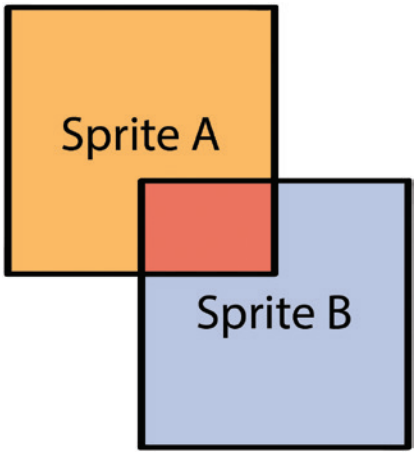

Figure 2.7: Sprites are overlapping: Collision is detected. Illustrated by Kyle Flemmer.

2.3.5. Graphical Explanation Edge Collision

A developer may want an edge overlap detected instead of a complete overlap. To do so, the function can be modified like so:

```
bool check_collision_edge(Sprite *a, Sprite *b) {
       bool left_edge = (a->x + a->width == b->x);
       bool right_edge = (a->x == b->x + b->width);
       bool top_edge = (a->y + a->height == b->y);
       bool bottom_edge = (a->y == b->y + b->height);

    return (left_edge || right_edge || top_edge ||
bottom_edge);
}
```

Here, we check the four edge detections, and we are now using OR logic for the return value, so if one of the edges of the sprite object is touching another's, we return a true. The detections are as follows:

left_edge: Right edge of Sprite A touches left edge of Sprite B.
right_edge: Left edge of Sprite A touches right edge of Sprite B.
top_edge: Bottom edge of Sprite A touches top edge of Sprite B.
bottom_edge: Top edge of Sprite A touches bottom edge of Sprite B.

To begin with, we would have the following, status-wise:

https://doi.org/10.15215/remix/9781998944224.01

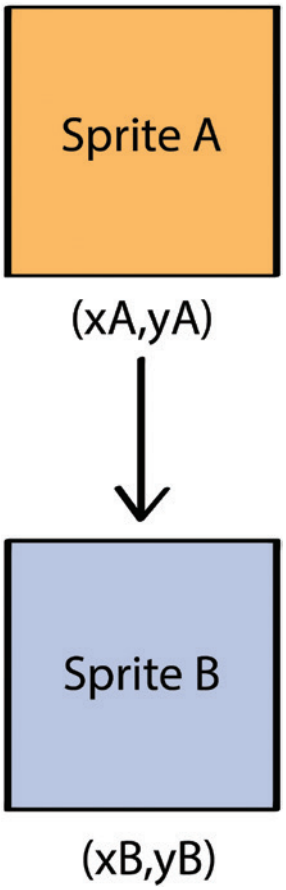

Figure 2.8: Sprites positioned apart, no collision detected. Illustrated by Kyle Flemmer.

a->x + a->width == b->x → **False**
a->x == b->x + b-> width → **False**
a->y + a->height == b->y → **False**
a->y == b->y + b->height → **False**

Our function currently returns false, but our Sprite A is moving downward, and Sprite B is stationary, so they will imminently touch.

With the contact of the top edge of Sprite B and the bottom edge Sprite A, the conditions then change to

a->x < b->x + b->width → **False**
a->x + a->width > b->x → **False**
a->y < b->y + b->height → **True**
a->y + a->height > b->y → **False**

https://doi.org/10.15215/remix/9781998944224.01

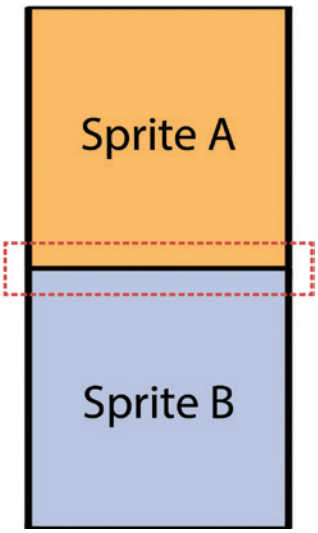

Figure 2.9: Sprites touch at edges: Collision is detected. Illustrated by Kyle Flemmer.

We now have a true condition, a collision is detected, and our function returns true.

Here's the example implementation in Allegro 5 using the overlap bounding box collision detection. This program creates two moving rectangles and detects collisions between them:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_primitives.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <stdbool.h>
#include <stdio.h>

typedef struct Sprite {
    float x, y, width, height;
    float dx, dy; // velocity
} Sprite;

bool check_collision(Sprite* a, Sprite* b) {
    return (a->x < b->x + b->width &&
        a->x + a->width > b->x &&
        a->y < b->y + b->height &&
        a->y + a->height > b->y);
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
int main() {
    al_init();
    al_init_primitives_addon();
    al_init_font_addon();
    al_init_ttf_addon();

    ALLEGRO_DISPLAY* display = al_create_display(800,
600);
    ALLEGRO_EVENT_QUEUE* event_queue =
al_create_event_queue();
    ALLEGRO_TIMER* timer = al_create_timer(1.0 / 60.0);

    ALLEGRO_FONT* font = al_load_ttf_font("arial.ttf",
36, 0);
    if (!font) {
        fprintf(stderr, "Could not load 'arial.ttf'. Make
sure the font file is available.\n");
        return-1;
    }

    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_timer_event_source(timer));

    Sprite sprite1 = { 100, 100, 50, 50, 2, 2 };
    Sprite sprite2 = { 300, 300, 50, 50,-2,-2 };

    bool collided = false;
    bool was_colliding = false;
    bool redraw = true;
    al_start_timer(timer);

    while (true) {
        ALLEGRO_EVENT event;
        al_wait_for_event(event_queue, &event);

        if (event.type == ALLEGRO_EVENT_TIMER) {
            // Update sprite positions
            sprite1.x += sprite1.dx;
```

https://doi.org/10.15215/remix/9781998944224.01

```
            sprite1.y += sprite1.dy;
            sprite2.x += sprite2.dx;
            sprite2.y += sprite2.dy;
            // Collision detection
            bool currently_colliding = check_
collision(&sprite1, &sprite2);

            if (currently_colliding && !was_colliding) {
                // Collision just started—reverse
directions once
                sprite1.dx =—sprite1.dx;
                sprite1.dy =—sprite1.dy;
                sprite2.dx =—sprite2.dx;
                sprite2.dy =—sprite2.dy;
                printf("We Collided\n");
            }

            collided = currently_colliding;
            was_colliding = currently_colliding;

            // Bounce off screen edges
            if (sprite1.x < 0 || sprite1.x + sprite1.
            width > 800) sprite1.dx =—sprite1.dx;
            if (sprite1.y < 0 || sprite1.y + sprite1.
            height > 600) sprite1.dy =—sprite1.dy;
            if (sprite2.x < 0 || sprite2.x + sprite2.
            width > 800) sprite2.dx =—sprite2.dx;
            if (sprite2.y < 0 || sprite2.y + sprite2.
            height > 600) sprite2.dy =—sprite2.dy;

            redraw = true;
        }
        else if (event.type == ALLEGRO_EVENT_DISPLAY_
CLOSE) {
            break;
        }

        if (redraw && al_is_event_queue_empty(event_
queue)) {
            redraw = false;
```

https://doi.org/10.15215/remix/9781998944224.01

```
            al_clear_to_color(al_map_rgb(0, 0, 0));

            // Change color to yellow if collision occurs
            ALLEGRO_COLOR color1 = collided ? al_map_
rgb(255, 255, 0) : al_map_rgb(255, 0, 0);
            ALLEGRO_COLOR color2 = collided ? al_map_
rgb(255, 255, 0) : al_map_rgb(0, 255, 0);

            al_draw_filled_rectangle(sprite1.x,
sprite1.y,
                sprite1.x + sprite1.width, sprite1.y +
sprite1.height, color1);
            al_draw_filled_rectangle(sprite2.x,
sprite2.y,
                sprite2.x + sprite2.width, sprite2.y +
sprite2.height, color2);

            if (collided) {
                al_draw_text(font, al_map_rgb(255, 255,
255), 400, 300,
                    ALLEGRO_ALIGN_CENTER, "We Collided");
            }

            al_flip_display();
        }
    }

    // Cleanup
    al_destroy_font(font);
    al_destroy_display(display);
    al_destroy_event_queue(event_queue);
    al_destroy_timer(timer);

    return 0;
}
```

Collision detection is vital for creating interactive and realistic game environments. By using techniques like AABB, circle collision, and SAT, you can handle various collision scenarios in your games.

https://doi.org/10.15215/remix/9781998944224.01

Collision detection can be a very complex issue. But by reducing it to component parts, we can make calculating collisions much easier. Simple overlap checks may miss certain state information but can often be "good enough" to get the job done (given small delta time between collision checks—otherwise, objects can easily pass through one another). They are simple and effective in most cases.

An improvement is intercept-based collision detection, where an actual intercept time is calculated. These intercept times can be stored in a table, and then the only time you need to update the table is when time passes or an object changes direction. This means the more collisions there are, the more calculation will be needed. But the end result is that objects behave correctly under all circumstances, and you can optimize the checking routine to only run when an object changes direction.

2.3.6. Useful Mathematical Equations for Collision Detection

Collision detection in video games involves various mathematical equations to determine if and when objects intersect. Here are some common methods and their equations:

Axis-Aligned Bounding Box (AABB)

For two axis-aligned bounding boxes, the collision detection can be determined by checking if their edges overlap. The equations are:

```
{Collision} = (A.x < B.x + B.width) and (A.x + A.width
> B.x) and (A.y < B.y + B.height) and (A.y + A.height >
B.y)
Where ( A ) and ( B ) are the two bounding boxes with
properties ( x ), ( y ), ( width ), and ( height ).
```

Circle Collision

For circular objects, the collision detection is based on the distance between their centers. The equation is:

```
{Collision} = sqrt{(A.x–B.x)^2 + (A.y–B.y)^2} < (A.radius
+ B.radius)
```

Where (A) and (B) are the circles with properties (x), (y), and (radius).

https://doi.org/10.15215/remix/9781998944224.01

Separating Axis Theorem (SAT)

The Separating Axis Theorem is used for detecting collisions between convex polygons. It involves projecting the vertices of the polygons onto various axes and checking for overlaps. The key idea is that if there is a separating axis where the projections do not overlap, then the polygons do not collide.

1. **Projection of a Point onto an Axis**
   Given a point P and an axis A, the projection P' is

$$P' = \frac{P \cdot A}{A \cdot A} \cdot A$$

2. **Overlap Check**
   For two polygons, project all vertices onto the axis and find the minimum and maximum values for each polygon. If the intervals overlap on all axes, the polygons collide.

Line Segment vs. Triangle

To check if a line segment intersects a triangle, you can use the following steps:

1. **Compute Signed Distances**
   Compute the signed distances of the segment endpoints to the plane of the triangle:

$$d = P - V_0 \cdot N$$
$$d_2 = P_2 - V_0 \cdot N$$

   Where P *and* P2 are the segment endpoints, V0 is a vertex of the triangle, and N is the normal of the triangle.
2. **Intersection Point**
   If the signs of d *and* d2 are different, compute the intersection point P:

$$P = P + \frac{d}{d - d_2} \cdot (P_2 - P)$$

3. **Point Inside Triangle**
   Check if the intersection point P is inside the triangle using barycentric coordinates or edge tests.

These equations form the basis of many collision detection algorithms used in games. For more detailed information, you can ask an AI assistant about lectures on collision detection algorithms in video games.

https://doi.org/10.15215/remix/9781998944224.01

2.4. Adding Sound Effects to Games

Sound effects play crucial roles in video games by enhancing the overall experience and creating an engaging environment. Sound and music are integral to the gaming experience, enhancing immersion, providing feedback, and supporting the narrative.

2.4.1. Sound Effects and Music

Sound effects include various sounds found in the real world as well as music.

Importance of Sound Effects

1. **Emotional Impact**
   Music sets the emotional tone of the game, influencing the player's mood and reactions. For example, a tense soundtrack can heighten the sense of danger, while a calm melody can create a peaceful atmosphere.
2. **Immersion**
   Sound effects and ambient sounds help create a believable game world. The sound of footsteps, rustling leaves, or distant explosions can make the game environment feel more real.
3. **Feedback**
   Audio cues provide feedback to the player, indicating actions or events. For instance, a sound effect might signal that a player has successfully completed a task or that an enemy is nearby.
4. **Narrative Enhancement**
   Music and sound effects can enhance the storytelling aspect of games. They can underscore dramatic moments, highlight important events, and support the narrative flow.

Components of Game Audio

1. **Music**
   Composed specifically for the game, music can vary depending on the game's genre and setting. Dynamic music systems can change the music based on the player's actions or the game's state.
2. **Sound Effects (SFX)**
   These include all the sounds made by characters, environments, and actions within the game. High-quality sound effects can significantly enhance the realism and immersion of the game.
3. **Voice Acting**
   Voice acting brings characters to life, adding depth to the narrative and making interactions more engaging. Good voice acting

https://doi.org/10.15215/remix/9781998944224.01

can greatly enhance the player's connection to the story and characters.

4. **Ambient Sounds**
   Background sounds that create the atmosphere of the game world. These can include environmental sounds like wind, water, and wildlife, which help to set the scene and make the world feel alive.

2.4.2. Implementing Sound and Music in Games

1. **Audio Middleware**
   Tools like FMOD and Wwise are commonly used to implement and manage game audio. They allow developers to create complex audio behaviors and integrate them seamlessly into the game.
2. **Dynamic Audio**
   Dynamic audio systems adjust the music and sound effects in real time based on the player's actions and the game state. This creates a more responsive and immersive audio experience.
3. **Spatial Audio**
   Techniques like 3D audio and binaural sound can create a sense of space and directionality, making it easier for players to locate sounds in the game world.

To use sound and music in Allegro games, you'll need to initialize the audio system, load audio files, and play them. Here's a step-by-step guide.

Setting Up

1. **Initialize Allegro and Audio Add-Ons**

```
if (!al_init()) {
    fprintf(stderr, "Allegro failed to init\n");
    return-1;
}
al_install_keyboard();
al_install_audio();
al_init_acodec_addon();
al_reserve_samples(8);
```

2. **Load Audio Files**
   Load your audio files into ALLEGRO_SAMPLE objects. Allegro supports various formats like WAV, OGG, and FLAC:

https://doi.org/10.15215/remix/9781998944224.01

```
ALLEGRO_SAMPLE *sfx = al_load_sample("sound.wav");
 if (!sfx) {
     fprintf(stderr, "Couldn't load sound.wav\n");
     return–1;
 }
```

3. **Play Sound Effects**
   To play a sound effect, use al_play_sample. Note that sound effects are often used in a "one-shot" type of occurrence with an event—that is, they don't repeat, and as such, we use a single play mode: ALLEGRO_PLAYMODE_ONCE.

```
    // Play sound effect
 al_play_sample(sfx, 1.0, 0.0, 1.0, ALLEGRO_
PLAYMODE_ONCE, nullptr);
```

4. **Load and Play Music**
   For background music, unlike the one-shot sound effect, we will use the looping function, as background music tends to play continuously. We specify this mode with ALLEGRO_PLAYMODE_LOOP.

```
ALLEGRO_AUDIO_STREAM *music = al_load_audio_
stream("music.ogg", 4, 2048);
if (!music) {
    fprintf(stderr, "Couldn't load music.ogg\n");
    return–1;
}
al_set_audio_stream_playmode(music,
ALLEGRO_PLAYMODE_LOOP);
al_attach_audio_stream_to_mixer(music,
al_get_default_mixer());
```

   Note also the use of two new functions here related to playback in Allegro: the audio stream mode and the audio stream mixer. Here, our music file is a larger continuous audio event (unlike a short sound sample), and so it is a "stream" of audio. We denote "music" as our "stream" and so set the play mode for that to LOOP. We then take that music stream and feed it to the default Allegro sound mixer. The Allegro mixer function lets us modify the playback characteristics if we wish; here, we don't make any changes to the

https://doi.org/10.15215/remix/9781998944224.01

audio and use the defaults. In a future chapter, we'll look at some of the uses of the mixer and its effect, but for now, just consider it a requirement for the output of the sound—the software speaker, as it were.

EXAMPLE CODE

Here's a complete example demonstrating how to set up and play sound and music in an Allegro game. It will start the music playback with a quick sound effect at the beginning, then continue to play. At the end of the music, after a brief pause, it will play again but with no sound effect and continue to do so until the example is ended:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_audio.h>
#include <allegro5/allegro_acodec.h>
#include <allegro5/allegro_native_dialog.h> // Optional:
error dialogs
#include <cstdio>

int main() {
    if (!al_init()) {
        fprintf(stderr, "Allegro failed to init\n");
        return-1;
    }

    al_install_keyboard();
    al_install_audio();
    al_init_acodec_addon();
    al_reserve_samples(8);

    ALLEGRO_DISPLAY *display = al_create_display(800,
600);
    if (!display) {
        fprintf(stderr, "Display creation failed\n");
        return-1;
    }

    ALLEGRO_SAMPLE *sfx = al_load_sample("sound.wav");
    if (!sfx) {
        fprintf(stderr, "Couldn't load sound.wav\n");
```

https://doi.org/10.15215/remix/9781998944224.01

```
        return-1;
    }

    ALLEGRO_AUDIO_STREAM *music = al_load_audio_
stream("music.ogg", 4, 2048);
    if (!music) {
        fprintf(stderr, "Couldn't load music.ogg\n");
        return-1;
    }
    al_set_audio_stream_playmode(music,
ALLEGRO_PLAYMODE_LOOP);
    al_attach_audio_stream_to_mixer(music,
al_get_default_mixer());

    // Play sound effect
    al_play_sample(sfx, 1.0, 0.0, 1.0, ALLEGRO_PLAYMODE_
ONCE, nullptr);

    // Set up event queue
    ALLEGRO_EVENT_QUEUE *queue = al_create_event_queue();
    al_register_event_source(queue,
al_get_keyboard_event_source());
    al_register_event_source(queue,
al_get_display_event_source(display));

    bool running = true;
    while (running) {
        ALLEGRO_EVENT ev;
        if (al_wait_for_event_timed(queue, &ev, 0.1)) {
            if (ev.type == ALLEGRO_EVENT_KEY_DOWN &&
ev.keyboard.keycode == ALLEGRO_KEY_ESCAPE) {
                running = false;
            }
            if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
                running = false;
            }
        }

        // Could redraw or update something here
    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    // Cleanup
    al_destroy_audio_stream(music);
    al_destroy_sample(sfx);
    al_destroy_display(display);
    al_destroy_event_queue(queue);
    al_uninstall_audio();
    return 0;
}
```

2.4.3. Streaming

Streaming in video games can refer to three distinct concepts: live streaming gameplay, game streaming services, and streaming media (the playback of audio or video) within a game. In this subsection, we describe all three uses of streaming in games, although only streaming media is directly related to game programming and development.

Live Streaming Gameplay

Live streaming gameplay involves broadcasting your gaming sessions to an audience in real time. This has become incredibly popular on platforms like Twitch, YouTube Gaming, and Facebook Gaming. Here's how you can get started:

1. **Equipment**
   Good Gaming PC: Ensure your PC can handle both gaming and streaming simultaneously.
   Webcam and Microphone: Use for better interaction with your audience.
   Capture Card: Use if you're streaming from a console.
2. **Software**
   OBS Studio: A free and open-source software for video recording and live streaming.
   Streamlabs OBS: A user-friendly version of OBS with additional features for streamers.
3. **Setting Up**
   Create an account on your chosen streaming platform.
   Configure your streaming software with your platform's stream key.
   Set up scenes and sources in OBS to include your game, webcam, and overlays.
4. **Engage with Your Audience**
   Interact with viewers through chat.
   Use alerts and notifications to acknowledge new followers, subscribers, and donations.

https://doi.org/10.15215/remix/9781998944224.01

Game Streaming Services

Game streaming services allow you to play games on various devices without needing powerful hardware. The game runs on a remote server or on your gaming PC, and the video is streamed to your device. Here are some popular services:

Steam Remote Play: Stream games from your PC to other devices like phones, tablets, and TVs using Steam Link.

NVIDIA GeForce NOW: Play your PC games from the cloud on various devices. It supports a wide range of games from different platforms.

Xbox Cloud Gaming (xCloud): Part of Xbox Game Pass Ultimate, allowing you to stream Xbox games to your PC, phone, or tablet.

PlayStation Now: Stream a library of PlayStation games to your PC or PlayStation console.

Getting Started with Game Streaming Services

Choose a Service: Select a service that supports the games you want to play and is available in your region.

Set Up Your Account: Create an account and subscribe to the service if necessary.

Install the App: Download and install the app on your device.

Connect a Controller: Many services support Bluetooth controllers for a better gaming experience.

Start Playing: Launch the app, log in, and start streaming your games.

Streaming in video games, whether live streaming your gameplay or using game streaming services, offers a flexible and engaging way to enjoy and share gaming experiences.

Media Streaming in Allegro Games

With Allegro, we have the ability to stream audio or video content within your game. Audio streaming capabilities are one of Allegro's add-on functions for video, though one will need to utilize an external library. Here's how you would handle both.

Streaming Audio

To stream audio in Allegro, you can use ALLEGRO_AUDIO_STREAM. This is useful for playing large audio files or continuous audio streams without loading

https://doi.org/10.15215/remix/9781998944224.01

the entire file into memory. We saw the use of this in the previous example code, but the steps here, once more, are:

1. **Initialize the Required Audio Add-Ons**

```
al_install_audio();
al_init_acodec_addon();
al_reserve_samples(8);
```

2. **Load and Play Audio Stream**

```
   ALLEGRO_AUDIO_STREAM *stream = al_load_audio_
stream("music.ogg", 4, 2048);
   if (!stream) {
       fprintf(stderr, "Failed to load audio
stream!\n");
       return–1;
   }
   al_set_audio_stream_playmode(stream,
ALLEGRO_PLAYMODE_LOOP);
   al_attach_audio_stream_to_mixer(stream,
al_get_default_mixer());
```

Controlling the Stream

When one is utilizing larger files, it might be useful to have some control over how those are played back. Allegro offers control over audio playback functions with al_set_audio_stream_playing, which is used to stop or start the current stream. A bool value of true starts or resumes a stream, while false stops or pauses it.

For example, this starts the stream:

```
bool playing = true;                    // start playing
immediately
al_set_audio_stream_playing(stream, true);
```

And then one might control the playback like this:

```
while (running) {
     ALLEGRO_EVENT ev;
     al_wait_for_event(queue, &ev);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
        running = false;
    } else if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
        if (ev.keyboard.keycode == ALLEGRO_KEY_ESCAPE) {
            running = false;
        } else if (ev.keyboard.keycode == ALLEGRO_KEY_
SPACE) {
            playing = !playing;
            al_set_audio_stream_playing(stream,
playing);
            draw_ui(display, font, playing);
        }
    } else if (ev.type == ALLEGRO_EVENT_DISPLAY_EXPOSE) {
        // Redraw if the window is exposed
        draw_ui(display, font, playing);
    }
 }
```

Streaming Video

For video streaming, Allegro does not have built-in support, but you can use external libraries like FFmpeg to decode video frames and display them using Allegro.

1. **Set Up FFmpeg**
   Install FFmpeg and include it in your project.
2. **Decode Video Frames**
   Use FFmpeg to decode video frames. This involves setting up FFmpeg to read the video file and decode each frame.
3. **Display Frames with Allegro**
   Convert the decoded frames to ALLEGRO_BITMAP and display them:

```
   ALLEGRO_BITMAP *frame_bitmap = al_create_
bitmap(width, height);
   while (av_read_frame(format_context, &packet) >=
0) {
       // Decode packet and convert to
ALLEGRO_BITMAP
       al_draw_bitmap(frame_bitmap, 0, 0, 0);
       al_flip_display();
   }
```

https://doi.org/10.15215/remix/9781998944224.01

Why bother streaming? In our previous example, when we called al_load_sample, we loaded sound.wav into memory in its entirety and kept it there. Most game sound effects aren't too long—probably a few seconds at most—so, provided you don't have too many, it's often more than reasonable to load all of them into memory when your program starts, and leave them there until it quits.

With this in mind, how do we handle longer pieces—specifically music? Your average 3:30 pop track, when encoded as a 320 Kbps MP3, weighs in at 8.4 MB. On today's computers, where even the cheapest laptop can bring 4GB RAM to the table, keeping this one track in memory wouldn't be too bad. Right?

Well, even if your game did only need to play this one piece of music, we haven't considered how much space it'll occupy once it's *decoded*. So what does this mean for our program?

- Most audio formats—MP3 included—are heavily compressed to take up less storage space.
- When playing back an MP3, it needs to be decoded—temporarily removing the compression.
- So for snappy playback, files read into ALLEGRO_SAMPLEs are thus stored decoded and uncompressed in memory. If they weren't, we'd need to decode them every time we wanted to play them, which would be slow.
- If you've ever converted an MP3 to WAV, you'll probably have noted the crazy increase in size; this is because WAVs are generally uncompressed, unlike MP3s.
- We can therefore assume that the size of the decoded, uncompressed audio data, as stored in memory, is also going to be big.

So when loading your 3:30 track into an ALLEGRO_SAMPLE, Allegro will actually be allocating something more like 37 MB of RAM. Scale this up to multiple tracks, and you're easily into the *hundreds* of megabytes; safe to say that things aren't looking so fresh now.

If only there was an easy way to seamlessly load a bit of the music at a time, decode it, play it, and then move on to the next bit. That way, we'd only have to store a small part of the track in memory at any given time! Great? Great. Streaming audio lets you do this.

EXAMPLE OF AUDIO STREAMING WITH PAUSE FUNCTION

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_audio.h>
#include <allegro5/allegro_acodec.h>
#include <allegro5/allegro_font.h>
```

https://doi.org/10.15215/remix/9781998944224.01

```
#include <stdio.h>
#include <stdbool.h>

static void draw_ui(ALLEGRO_DISPLAY *display, ALLEGRO_
FONT *font, bool playin
    // Clear screen
    al_clear_to_color(al_map_rgb(20, 20, 24));

    // Centered message
    const char *msg = playing ? "Playing (SPACE to
pause)" : "Paused (SPACE t
    ALLEGRO_COLOR col = playing ? al_map_rgb(30, 220,
120) : al_map_rgb(240,

    int w = al_get_display_width(display);
    int h = al_get_display_height(display);

    al_draw_text(font, col, w/2,
h/2-al_get_font_line_height(font)/2, ALLEG
    al_flip_display();
}

int main() {
    // Init core
    if (!al_init()) { fprintf(stderr, "Failed to
initialize Allegro!\n"); ret
    if (!al_install_audio()) { fprintf(stderr, "Failed to
initialize audio!\n
    if (!al_init_acodec_addon()) { fprintf(stderr,
"Failed to init acodec add
    if (!al_install_keyboard()) { fprintf(stderr, "Failed
to install keyboard
    al_init_font_addon();

    al_reserve_samples(16);

    ALLEGRO_DISPLAY *display = al_create_display(800,
600);
    if (!display) { fprintf(stderr, "Failed to create
display!\n"); return-1
```

https://doi.org/10.15215/remix/9781998944224.01

```
    // Built-in font (no TTF dependency)
    ALLEGRO_FONT *font = al_create_builtin_font();
    if (!font) { fprintf(stderr, "Failed to create
font!\n"); al_destroy_disp

    // Load audio stream
    ALLEGRO_AUDIO_STREAM *stream = al_load_audio_
stream("music.ogg", 4, 2048)
    if (!stream) {
        fprintf(stderr, "Failed to load audio
stream!\n");
        al_destroy_font(font);
        al_destroy_display(display);
        return–1;
    }

    al_attach_audio_stream_to_mixer(stream,
al_get_default_mixer());
    al_set_audio_stream_playmode(stream,
ALLEGRO_PLAYMODE_LOOP);

    // Event queue
    ALLEGRO_EVENT_QUEUE *queue = al_create_event_queue();
    al_register_event_source(queue,
al_get_keyboard_event_source());
    al_register_event_source(queue,
al_get_display_event_source(display));

    bool running = true;
    bool playing = true;                    // start
playing immediately
    al_set_audio_stream_playing(stream, true);
    draw_ui(display, font, playing);       // initial draw

    while (running) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(queue, &ev);

        if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            running = false;
```

https://doi.org/10.15215/remix/9781998944224.01

```
        } else if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
            if (ev.keyboard.keycode == ALLEGRO_KEY_
ESCAPE) {
                running = false;
            } else if (ev.keyboard.keycode == ALLEGRO_
KEY_SPACE) {
                playing = !playing;
                al_set_audio_stream_playing(stream,
playing);
                draw_ui(display, font, playing);
            }
        } else if (ev.type == ALLEGRO_EVENT_DISPLAY_
EXPOSE) {
            // Redraw if the window is exposed
            draw_ui(display, font, playing);
        }
    }

    // Cleanup
    al_destroy_audio_stream(stream);
    al_destroy_font(font);
    al_destroy_event_queue(queue);
    al_destroy_display(display);
    al_uninstall_audio();
    return 0;
}
```

2.5. Timers and Game Timing

Timers and timing are crucial elements in games, affecting gameplay, performance, and the overall player experience.

2.5.1. Roles of Timers and Timing in Games

Timers and timing mechanisms are used in several key aspects of game development.

https://doi.org/10.15215/remix/9781998944224.01

Importance of Timers and Timing

1. **Gameplay Mechanics**
   Timers are often used to create time-based challenges, such as countdowns for completing a task or time limits for levels. This adds urgency and excitement to the gameplay.
2. **Animation and Movement**
   Timing is essential for smooth animations and character movements. Consistent timing ensures that animations run at the correct speed and look natural.
3. **Game Loops**
   The game loop relies on precise timing to update game states and render frames consistently. This ensures a smooth and responsive gaming experience.

Types of Timers

1. **Real-Time Timers**
   These timers measure actual elapsed time and are used for tasks that need to happen at specific intervals, such as spawning enemies or updating the game state.
2. **Frame-Based Timers**
   These timers are based on the number of frames rendered. They are useful for animations and movements that need to be consistent regardless of the frame rate.
3. **Event Timers**
   These timers trigger specific events after a set period. They are often used for delayed actions, such as power-ups or timed events.

2.5.2. Implementing Timers in Allegro

Here's how you can implement timers in an Allegro game:

1. **Initialize Allegro and Create a Timer**

```
//——Timer & Events——
ALLEGRO_TIMER *timer = al_create_timer(1.0 / 60.0);
// start at 60 FPS
ALLEGRO_EVENT_QUEUE *queue =
al_create_event_queue();
al_register_event_source(queue,
al_get_display_event_source(display));
```

https://doi.org/10.15215/remix/9781998944224.01

```
al_register_event_source(queue,
al_get_keyboard_event_source());
al_register_event_source(queue,
al_get_timer_event_source(timer));
al_start_timer(timer);
```

2. **Control Game Loop with Timer Events**
   Use the timer to control the game loop and ensure consistent updates:

```
   while (true) {
       ALLEGRO_EVENT ev;
       al_wait_for_event(event_queue, &ev);

       if (ev.type == ALLEGRO_EVENT_TIMER) {
           // Update game state
           // Render frame
           al_flip_display();
       } else if (ev.type == ALLEGRO_EVENT_DISPLAY_
CLOSE) {
           break;
       }
   }
```

3. **Use Timers for Delayed Actions**
   You can also use timers to trigger events after a delay:

```
   double start_time = al_get_time();
   double delay = 5.0; // 5 seconds delay

   while (true) {
       double current_time = al_get_time();
       if (current_time–start_time >= delay) {
           // Trigger event
           start_time = current_time; // Reset timer
       }

       // Other game logic
   }
```

https://doi.org/10.15215/remix/9781998944224.01

2.5.3. Example Code

In this example, we demonstrate how to use a timer to control the game loop and update the game state at a consistent frame rate. The space bar will allow the user to change the frame rate limiter via the timer, with rates of 15 fps, 30 fps, 60 fps, and unlimited. The effect of the frame rate can be seen with the animation of the ball. Look at the edges of the ball as it moves and you'll see the effect:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_primitives.h>
#include <allegro5/allegro_font.h>
#include <stdio.h>
#include <math.h>
#include <stdbool.h>

static const int SCREEN_W = 800;
static const int SCREEN_H = 600;

typedef enum { FPS_15, FPS_30, FPS_60, FPS_UNLIMITED,
FPS_COUNT } FpsMode;

static double fps_value(FpsMode m) {
    switch (m) {
        case FPS_15: return 15.0;
        case FPS_30: return 30.0;
        case FPS_60: return 60.0;
        case FPS_UNLIMITED: return 0.0; // sentinel for
unlimited
        default: return 60.0;
    }
}

static const char* fps_label(FpsMode m) {
    switch (m) {
        case FPS_15: return "15 FPS";
        case FPS_30: return "30 FPS";
        case FPS_60: return "60 FPS";
        case FPS_UNLIMITED: return "Unlimited";
        default: return "?";
    }
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
int main(void) {
    //——Init Allegro——
    if (!al_init()) { fprintf(stderr, "Failed to init
Allegro\n"); return–1; }
    if (!al_install_keyboard()) { fprintf(stderr, "Failed
to install keyboard\n"); return–1; }
    if (!al_init_primitives_addon()) { fprintf(stderr,
"Failed to init primitives\n"); return–1; }
    al_init_font_addon(); // for builtin font

    ALLEGRO_DISPLAY *display = al_create_
display(SCREEN_W, SCREEN_H);
    if (!display) { fprintf(stderr, "Failed to create
display\n"); return–1; }

    ALLEGRO_FONT *font = al_create_builtin_font();
    if (!font) { fprintf(stderr, "Failed to create
builtin font\n"); al_destroy_display(display); return–1;
}

    //——Timer & Events——
    ALLEGRO_TIMER *timer = al_create_timer(1.0 / 60.0);
// start at 60 FPS
    ALLEGRO_EVENT_QUEUE *queue =
al_create_event_queue();
    al_register_event_source(queue,
al_get_display_event_source(display));
    al_register_event_source(queue,
al_get_keyboard_event_source());
    al_register_event_source(queue,
al_get_timer_event_source(timer));
    al_start_timer(timer);

    //——Ball state——
    float radius = 20.0f;
    float x = SCREEN_W * 0.5f, y = SCREEN_H * 0.5f;
    float vx = 220.0f, vy = 180.0f; // pixels per second

    //——Control state——
    FpsMode mode = FPS_60;
```

https://doi.org/10.15215/remix/9781998944224.01

```
    bool redraw = true;
    bool running = true;

    // Timekeeping for unlimited mode and smooth motion
    double last_time = al_get_time();

    while (running) {
        ALLEGRO_EVENT ev;
        if (mode == FPS_UNLIMITED) {
            // In unlimited mode, don't block waiting for
a timer
            if (al_get_next_event(queue, &ev)) {
                // process any pending event
            } else {
                // no events pending: mark to update/
render immediately
                redraw = true;
            }
        } else {
            al_wait_for_event(queue, &ev);
        }

        // Process events if we have one (in unlimited we
may or may not)
        if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            running = false;
        } else if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
            if (ev.keyboard.keycode == ALLEGRO_KEY_
ESCAPE) {
                running = false;
            } else if (ev.keyboard.keycode == ALLEGRO_
KEY_SPACE) {
                // Cycle FPS mode
                mode = (FpsMode)((mode + 1) % FPS_COUNT);
                double f = fps_value(mode);
                if (mode == FPS_UNLIMITED) {
                    al_stop_timer(timer);
                } else {
                    al_set_timer_speed(timer, 1.0 / f);
```

https://doi.org/10.15215/remix/9781998944224.01

```
                    if (!al_get_timer_started(timer))
al_start_timer(timer);
                }
                // reset time reference to avoid a big dt
jump
                last_time = al_get_time();
                redraw = true;
            }
        } else if (ev.type == ALLEGRO_EVENT_TIMER) {
            // Timer tick—> redraw/update
            if (ev.timer.source == timer) {
                redraw = true;
            }
        }

        if (redraw && (mode == FPS_UNLIMITED || al_is_
event_queue_empty(queue))) {
            //——Compute dt——
            double now = al_get_time();
            double dt;
            if (mode == FPS_UNLIMITED) {
                dt = now—last_time; // real elapsed time
            } else {
                // Fixed step from timer frequency
                dt = 1.0 / fps_value(mode);
            }
            last_time = now;

            //——Update ball physics——
            x += vx * (float)dt;
            y += vy * (float)dt;

            // Collide with edges and bounce
            if (x—radius < 0.0f) { x = radius; vx =
fabsf(vx); }
            if (x + radius > SCREEN_W) { x =
SCREEN_W—radius; vx =—fabsf(vx); }
            if (y—radius < 0.0f) { y = radius; vy =
fabsf(vy); }
```

https://doi.org/10.15215/remix/9781998944224.01

```
            if (y + radius > SCREEN_H) { y =
SCREEN_H–radius; vy =–fabsf(vy); }

            //––Render––
            al_clear_to_color(al_map_rgb(18, 18, 22));
            al_draw_filled_circle(x, y, radius, al_map_
rgb(80, 180, 255));
            al_draw_textf(font, al_map_rgb(230, 230,
230), 10, 10, 0,
                          "FPS: %s  |  SPACE to
toggle  |  ESC to quit", fps_label(mode));
            al_flip_display();

            redraw = false;
        }
    }

    //––Cleanup––
    al_destroy_event_queue(queue);
    al_destroy_timer(timer);
    al_destroy_font(font);
    al_destroy_display(display);
    return 0;
}
```

2.6. Using Files for Games

Using files in video games is essential for storing and managing various types of data, such as game assets, configurations, save files, and more.

2.6.1. File Types Commonly Used in Games

Games rely on several different types of files to support gameplay, customization, persistence, and debugging.

Types of Files in Games

1. **Asset Files**

   These include images, sounds, music, models, and other resources used in the game. They are often stored in formats like PNG, WAV, MP3, OBJ, and so on.

https://doi.org/10.15215/remix/9781998944224.01

2. **Configuration Files**
   These files store settings and preferences for the game. Common formats include INI, JSON, and XML.
3. **Save Files**
   Save files store the player's progress and game state. They can be in various formats, often custom to the game.
4. **Log Files**
   Log files record events and errors that occur during gameplay, useful for debugging and support.

2.6.2. Managing Files in Allegro

Here's how you can handle files in an Allegro game:

1. **Loading Asset Files**
   Use Allegro's built-in functions to load images, sounds, and other assets:

```
   ALLEGRO_BITMAP *image = al_load_bitmap("image.
png");
   if (!image) {
       fprintf(stderr, "Failed to load image!\n");
       return-1;
   }

   ALLEGRO_SAMPLE *sound = al_load_sample("sound.
wav");
   if (!sound) {
       fprintf(stderr, "Failed to load sound!\n");
       return-1;
   }
```

2. **Reading and Writing Configuration Files**
   Use standard C functions or libraries like libconfig to read and write configuration files:

```
   FILE *file = fopen("config.ini", "r");
   if (file) {
       char buffer[256];
       while (fgets(buffer, sizeof(buffer), file)) {
           // Process each line
```

https://doi.org/10.15215/remix/9781998944224.01

```
        }
        fclose(file);
    } else {
        fprintf(stderr, "Failed to open config
file!\n");
    }
```

3. **Handling Save Files**
   Save and load game state using file I/O operations:

```
    // Saving game state
    FILE *save_file = fopen("save.dat", "wb");
    if (save_file) {
        fwrite(&game_state, sizeof(GameState), ,
save_file);
        fclose(save_file);
    } else {
        fprintf(stderr, "Failed to save game
state!\n");
    }

    // Loading game state
    FILE *load_file = fopen("save.dat", "rb");
    if (load_file) {
        fread(&game_state, sizeof(GameState), ,
load_file);
        fclose(load_file);
    } else {
        fprintf(stderr, "Failed to load game
state!\n");
    }
```

4. **Logging Events**
   Write log messages to a file for debugging purposes:

```
    FILE *log_file = fopen("game.log", "a");
    if (log_file) {
        fprintf(log_file, "Game started\n");
        fclose(log_file);
    } else {
```

https://doi.org/10.15215/remix/9781998944224.01

```
        fprintf(stderr, "Failed to open log
file!\n");
    }
```

2.6.3. Code Example

The code example we'll examine in this section is intentionally designed to simulate a failure scenario. The goal is to demonstrate how a game initialization routine can log the status of various subsystems—such as display, controller, video, and sound—into a log file (game.log), even when required resources are missing.

Behavior

The program attempts to read from a configuration file (config.ini) located in the same directory as the executable. This file would typically contain user-defined settings for various subsystems. On each execution, the program logs the success or failure of initializing these components to game.log.

Since the required assets (e.g., sound files, graphics, and config.ini) are deliberately omitted, the program will launch and then exit immediately. After execution, you can inspect game.log to see which components failed to initialize.

How to Test

To observe changes in behavior:

1. Run the executable as is and examine game.log.
2. Add one or more of the missing files (e.g., config.ini) to the executable's directory.
3. Run the program again and compare the updated log entries.

This approach helps illustrate how initialization routines and logging mechanisms behave under failure conditions and how they can be used to diagnose missing or misconfigured resources.

Here's the complete example demonstrating how to load an image, read a configuration file, save game state, and log events in an Allegro game:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_audio.h>
#include <allegro5/allegro_acodec.h>
#include <stdio.h>
#include <time.h>
```

https://doi.org/10.15215/remix/9781998944224.01

```
typedef struct {
    int level;
    int score;
} GameState;

int main() {
    /*——open log first so errors get recorded even if
early failures
    FILE *log_file = fopen("game.log", "a");
    if (!log_file) {
        fprintf(stderr, "Failed to open log file!\n");
    } else {
        time_t t = time(NULL);
        struct tm *tmv = localtime(&t);
        char ts[64] = {0};
        if (tmv) strftime(ts, sizeof(ts), "%Y-%m-%d
%H:%M:%S", tmv);
        fprintf(log_file, "[%s] Game started\n", ts[0] ?
ts : "unknown-
        fflush(log_file);
    }

    al_init();
    al_init_image_addon();
    if (!al_install_audio()) {
        fprintf(stderr, "Failed to initialize audio!\n");
        if (log_file) fprintf(log_file, "Failed to
initialize audio!\n"
    }
    if (!al_init_acodec_addon()) {
        if (log_file) fprintf(log_file, "Failed to init
acodec addon!\n
    }
    if (!al_reserve_samples(16)) {
        if (log_file) fprintf(log_file, "Failed to
reserve samples!\n")
    }

    ALLEGRO_DISPLAY *display = al_create_display(800,
600);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    if (!display) {
        fprintf(stderr, "Failed to create display!\n");
        if (log_file) fprintf(log_file, "Failed to create
display!\n");
        /* continue so we can still read config and write
log */
    }

    ALLEGRO_BITMAP *image = al_load_bitmap("image.png");
    if (!image) {
        fprintf(stderr, "Failed to load image!\n");
        if (log_file) fprintf(log_file, "Failed to load
image: image.png\n");
        /* do not return; keep logging/cleanup consistent
*/
    }

    ALLEGRO_SAMPLE *sound = al_load_sample("sound.wav");
    if (!sound) {
        fprintf(stderr, "Failed to load sound!\n");
        if (log_file) fprintf(log_file, "Failed to load
sound: sound.wav\n");
    }

    FILE *config_file = fopen("config.ini", "r");
    if (config_file) {
        if (log_file) fprintf(log_file, "Opened config.
ini for reading\
        char buffer[256];
        while (fgets(buffer, sizeof(buffer), config_
file)) {
            /* Process each line */
        }
        fclose(config_file);
        if (log_file) fprintf(log_file, "Finished reading
config.ini\n"
    } else {
        fprintf(stderr, "Failed to open config file!\n");
        if (log_file) fprintf(log_file, "Failed to open
config.ini\n");
    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    GameState game_state = {0, 0};
    FILE *save_file = fopen("save.dat", "wb");
    if (save_file) {
        size_t wrote = fwrite(&game_state,
sizeof(GameState), 1, save_f
        fclose(save_file);
        if (log_file) {
        fprintf(log_file, "Attempted to save game state:
wrote %zu record(s)\n", wrote);
              if (wrote != 1) fprintf(log_file, "Warning:
partial write to save.dat\n");
        fprintf(log_file, "Warning: partial write to
save.dat\n");
        }
    } else {
            fprintf(stderr, "Failed to save game
state!\n");
                if (log_file) fprintf(log_file, "Failed
to open save.dat for writing\n");
    }

    /* final log + cleanup */
    if (log_file) {
        fprintf(log_file, "Shutting down\n");
        fclose(log_file);
    }

    if (image) al_destroy_bitmap(image);
    if (sound) al_destroy_sample(sound);
    if (display) al_destroy_display(display);
    al_uninstall_audio();
    return 0;
}
```

2.6.4. PhysicsFS

Using PhysicsFS in Allegro 5 games allows you to manage files and archives more efficiently. PhysicsFS provides abstract access to various archives (like ZIP files), making it easier to handle game assets.

Here's how you can integrate and use PhysicsFS with Allegro 5.

https://doi.org/10.15215/remix/9781998944224.01

Setting Up PhysicsFS

1. **Install PhysicsFS**
   Download and install PhysicsFS from the official website.
2. **Include PhysicsFS in Your Project**
   Make sure to include the PhysicsFS header and link to the PhysicsFS library in your project.

Integrating PhysicsFS with Allegro

1. **Initialize PhysicsFS**
   Initialize PhysicsFS and set up the search path:

```
if (PHYSFS_init(NULL) == 0) {
     fprintf(stderr, "Failed to initialize
PhysicsFS—%s\n",
             PHYSFS_getErrorByCode(PHYSFS_
getLastErrorCode()));
     return—1;
 }
 if (PHYSFS_mount("data.zip", NULL, 1) == 0) {
     fprintf(stderr, "Failed to mount archive—%s\n",
             PHYSFS_getErrorByCode(PHYSFS_
getLastErrorCode()));
     PHYSFS_deinit();
     return—1;
 }
```

2. **Set Allegro to Use PhysicsFS**
   Use Allegro's PhysicsFS add-on to set the file interface:

```
   al_set_physfs_file_interface();
```

3. **Load Files Using Allegro's File I/O API**
   Now you can load files from the archive using Allegro's standard file I/O functions:

```
   ALLEGRO_BITMAP *image = al_load_bitmap("image.
png");
if (!image) {
    fprintf(stderr, "Failed to load image!\n");
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_destroy_display(display);
    PHYSFS_deinit();
    return-1;
}
ALLEGRO_SAMPLE *sound = al_load_sample("sound.wav");
if (!sound) {
    fprintf(stderr, "Failed to load sound!\n");
    al_destroy_bitmap(image);
    al_destroy_display(display);
    PHYSFS_deinit();
    return-1;
}
```

EXAMPLE CODE

In this example, we'll use PhysicsFS with Allegro 5 to open a data file, data.zip, and extract an image to display (image.png) and a sound to play (sound.wav). The program will get the data files from data.zip, play a sound, and display the image for 15 seconds:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_audio.h>
#include <allegro5/allegro_acodec.h>
#include <allegro5/allegro_physfs.h>
#include <physfs.h>
#include <stdio.h>

int main() {
    al_init();
    al_init_image_addon();
    al_install_audio();
    al_init_acodec_addon();
    al_reserve_samples(8);  // specify number of
samples

    if (PHYSFS_init(NULL) == 0) {
        fprintf(stderr, "Failed to initialize
PhysicsFS-%s\n",
                PHYSFS_getErrorByCode(PHYSFS_
getLastErrorCode()));
```

https://doi.org/10.15215/remix/9781998944224.01

```
        return-1;
    }

    if (PHYSFS_mount("data.zip", NULL, 1) == 0) {
        fprintf(stderr, "Failed to mount
archive-%s\n",
                PHYSFS_getErrorByCode(PHYSFS_
getLastErrorCode()));
        PHYSFS_deinit();
        return-1;
    }

    al_set_physfs_file_interface();

    ALLEGRO_DISPLAY *display = al_create_display(800,
600);
    if (!display) {
        fprintf(stderr, "Failed to create display!\n");
        PHYSFS_deinit();
        return-1;
    }

    ALLEGRO_BITMAP *image = al_load_bitmap
("image.png");
    if (!image) {
        fprintf(stderr, "Failed to load image!\n");
        al_destroy_display(display);
        PHYSFS_deinit();
        return-1;
    }

    ALLEGRO_SAMPLE *sound = al_load_sample("sound.wav");
    if (!sound) {
        fprintf(stderr, "Failed to load sound!\n");
        al_destroy_bitmap(image);
        al_destroy_display(display);
        PHYSFS_deinit();
        return-1;
    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_clear_to_color(al_map_rgb(0, 0, 0));
    al_draw_bitmap(image, 0, 0, 0);
    al_flip_display();
    al_play_sample(sound, 1.0, 0.0, 1.0, ALLEGRO_
PLAYMODE_ONCE, NULL);
    al_rest(15.0);

    al_destroy_bitmap(image);
    al_destroy_sample(sound);
     al_destroy_display(display);
    PHYSFS_deinit();
    return 0;
}
```

2.6.5. Memfiles

Using memfiles in Allegro 5 allows you to treat a block of memory as a file, which can be very useful for handling data stored in memory rather than on disk. Here's how you can use memfiles in your Allegro 5 games.

Setting Up

1. **Include the memfile Header**
   Make sure to include the allegro_memfile.h header in your project:

   ```
   #include <allegro5/allegro.h>
   #include <allegro5/allegro_memfile.h>
   ```

2. **Link with the memfile Add-On**
   Ensure your project links against the allegro_memfile library.
3. **Create and Use memfiles**
   Create a Memfile: Use al_open_memfile to create a memfile from a block of memory:

   ```
   char data = "This is some data in memory.";
   ALLEGRO_FILE *memfile = al_open_memfile(data,
sizeof(data), "r");
   if (!memfile) {
       fprintf(stderr, "Failed to create
memfile!\n");
       return-1;
   }
   ```

https://doi.org/10.15215/remix/9781998944224.01

Read from the memfile: You can read from the memfile using Allegro's file I/O functions:

```
    char buffer[256];
    al_fread(memfile, buffer, sizeof(data));
    printf("Read from memfile—%s\n", buffer);
```

Write to the memfile: If the memfile is opened in write mode, you can write to it:

```
    ALLEGRO_FILE *writable_memfile = al_open_
memfile(data, sizeof(data), "w");
    if (!writable_memfile) {
        fprintf(stderr, "Failed to create writable
memfile!\n");
        return-1;
    }
    const char *new_data = "New data in memory.";
    al_fwrite(writable_memfile, new_data, strlen(new_
data) + );
```

Close the memfile: Always close the memfile when you're done:

```
    al_fclose(memfile);
    al_fclose(writable_memfile);
```

EXAMPLE CODE

Here's a complete example demonstrating how to create, read from, and write to a memfile in Allegro 5:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_memfile.h>
#include <stdio.h>
#include <string.h>

int main() {
    al_init();

    char data = "This is some data in memory.";
```

https://doi.org/10.15215/remix/9781998944224.01

```
    ALLEGRO_FILE *memfile = al_open_memfile(data,
sizeof(data), "r");
    if (!memfile) {
        fprintf(stderr, "Failed to create memfile!\n");
        return-1;
    }

    char buffer[256];
    al_fread(memfile, buffer, sizeof(data));
    printf("Read from memfile—%s\n", buffer);

    al_fclose(memfile);

    ALLEGRO_FILE *writable_memfile = al_open_
memfile(data, sizeof(data), "w");
    if (!writable_memfile) {
        fprintf(stderr, "Failed to create writable
memfile!\n");
        return-1;
    }

    const char *new_data = "New data in memory.";
    al_fwrite(writable_memfile, new_data, strlen(new_
data) + );

    al_fclose(writable_memfile);

    printf("Updated data—%s\n", data);

    return 0;
}
```

This example initializes Allegro, creates a memfile from a block of memory, reads from it, writes new data to it, and then prints the updated data.

https://doi.org/10.15215/remix/9781998944224.01

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Section 2.1: Basic Structure and Setup

1. **"Hello World!" Modifications**
   Modify the "Hello World!" program to display your name in a custom font, center-aligned, with a gradient background. Use al_draw_gradient() for the background.
2. **Dynamic Window Sizing**
   Create a program where the user can input window dimensions (via command-line arguments) and render text at the center of the dynamically sized window.
3. **Error Handling**
   Enhance the basic Allegro setup code to handle failures gracefully (e.g., missing fonts, display creation errors) without crashing. Print descriptive error messages.

Section 2.2: User Input Handling

4. **Keyboard-Controlled Sprite**
   Create a 2D scene where a square moves continuously using arrow keys (state-based input) and jumps when the space bar is pressed (event-based input).
5. **Mouse Drawing Tool**
   Build a program that lets users draw lines or shapes on the screen using mouse clicks and drags. Include a button (rendered on-screen) to clear the canvas.
6. **Joystick Calibration**
   Write a program that detects connected joysticks, prints their axes/buttons, and lets the user calibrate joystick sensitivity.
7. **Menu Navigation**
   Design a menu system (Start, Options, Exit) navigable via keyboard or mouse. Highlight selected options and play sound effects on interaction.

Section 2.3: Collision Detection

8. **Pong Clone**
   Implement a two-player Pong game with paddle-ball collision (AABB) and scoring. The ball speeds up after each hit.
9. **Platformer Physics**
   Create a character that jumps and moves on platforms. Use AABB collision to prevent falling through floors and to handle wall slides.

https://doi.org/10.15215/remix/9781998944224.01

10. **Circle-Based Shooter**
    Develop a top-down shooter where bullets (circles) collide with enemy targets. Use circle collision detection and visual feedback on hits.
11. **SAT Implementation**
    Implement the Separating Axis Theorem (SAT) for collision between two rotating rectangles, including visualization of collision normals—the direction along which the overlap is minimal—and penetration depth, which represents how far the rectangles intersect along that normal.

Section 2.4: Sound and Music

12. **Rhythm Game Prototype**
    Create a rhythm game where players press keys in sync with music. Play sounds for correct inputs and track accuracy.
13. **Ambient Soundscapes**
    Build a scene with dynamic ambient sounds (e.g., rain, wind) that adjust volume based on the player's position. Use audio streaming for background music.
14. **Voice-Acted Dialogue**
    Design a text-based RPG with voice-acted dialogue snippets triggered by player choices. Manage audio memory efficiently.

Section 2.5: Timers and Timing

15. **Frame Rate Limiter**
    Implement a game loop using ALLEGRO_TIMER to lock the frame rate at 60 fps. Measure and display the actual fps achieved.
16. **Animation Sequencer**
    Animate a sprite sheet (e.g., walking character), using timers to cycle frames smoothly. Allow pausing/resuming animations.

Section 2.6: File Management

18. **Save/Load System**
    Develop a game where player progress (e.g., level, score) is saved to a binary file. Implement a "Continue" option to load saved data.
19. **Configuration File Parser**
    Read game settings (e.g., resolution, volume) from an INI/JSON file and apply them at start-up. Allow runtime changes and saving preferences.
20. **PhysicsFS Asset Loader**
    Package game assets (images, sounds) into a ZIP archive. Use PhysicsFS to load resources dynamically during gameplay.

https://doi.org/10.15215/remix/9781998944224.01

21. **Memfile Level Editor**
    Design a tool that lets users create levels in memory (using memfiles) and export them to a file format for later loading.

Comprehensive Projects

22. **2D Arcade Game**
    Build a complete game with scoring, sound effects, collision, and a high-score system (think *Space Invaders*, *Breakout*).
23. **Multiplayer Quiz Game**
    Create a quiz game where two players answer questions using keyboards. (*Optional:* Use timers for question limits and network sockets for communication.)
24. **Procedural Terrain Generator**
    Generate random terrain using Perlin noise, save/load maps to files, and allow players to explore with mouse/keyboard controls.
25. **Roguelike Prototype**
    Develop a dungeon-crawling game with grid-based movement, enemy AI, and permadeath. Use AABB collision for combat.
26. **Music Visualizer**
    Analyze audio streams (e.g., beat detection) and render real-time visual effects synchronized to the music.
27. **Physics Sandbox**
    Simulate gravity and collisions between objects (circles, rectangles). Let users spawn objects with mouse clicks and adjust physics parameters.
28. **AI Racing Game**
    Implement a car-racing game with AI opponents. Use state-based input for steering and event-based sounds for collisions.
29. **Interactive Storybook**
    Combine mouse input, animations, and voice acting to create an interactive story with branching narratives. Save progress between sessions.
30. **Final Project: Portfolio Game**
    Integrate all concepts (input, collision, sound, timing, files) into a polished game of your design. Include a README explaining technical choices.

https://doi.org/10.15215/remix/9781998944224.01

3

Using Graphics in Games

In the beginning, video games made use of the technologies available to them; the methods of input and output framed the development of games and how they operated. From the earliest text-only games to today's realistic three-dimensional world renderings, programmers have advanced along with the latest hardware implementations. Changes tend to be incremental in nature; in adventure games, for example, we began with the use of textual descriptions for the scene/area, which then evolved into the use of simple character-based ASCII or extended ASCII graphics to illustrate scenes, or created top-down versions of worlds to explore (see figure 3.1).

Soon after simple primitive screen graphics hardware and colour displays arrived (typically a television in many cases), the era of blocky and simple graphic game styles began; this then was refined further into the familiar 8-bit blocky retro games. The 8-bit character evolved into the development and

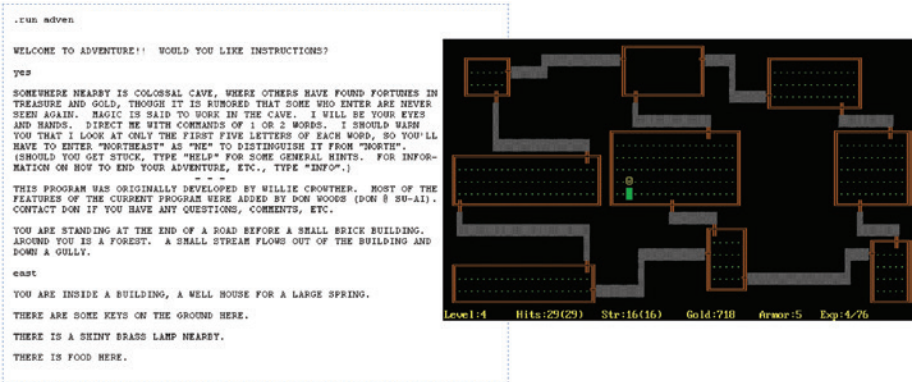

Figure 3.1: Text-based *The Colossal Cave Adventure* (*left*) and *Rogue*, using ASCII graphics (*right*). Images by William Crowther and Don Woods (*left*) and Artoftransformation (*right*), published under a CC BY-SA 3.0 license and modified for size and fitting by Kyle Flemmer.

https://doi.org/10.15215/remix/9781998944224.01

use of better-organized bitmap objects or "sprite graphics"—the main forte of Allegro's graphical capabilities.

3.1. Bitmap Fundamentals

Let's do a quick overview of bitmap graphics and how they relate to game development. The typical way a computer performs its display of graphics functions is by utilizing a section of memory often referred to as video memory. Graphics-hardware chips access this block memory to display information to the user. This information can be text based, graphical in nature, or a combination of both. A character displayed on a computer screen is essentially a bit matrix composed of two colours (think white and black, or on and off) displaying a letter. That character is then displayed by mapping that pattern onto a location in the screen memory. An example of an early bitmap for characters might look like figure 3.2.

This same process can also be utilized to display more complex information items ranging from game objects to backgrounds.

You will probably be familiar with terms related to screen memory capability: the amount of video RAM in a graphics card (external or on a motherboard) and the screen resolution. Video RAM (VRAM) is memory used to hold information to display on the computer's screen; it typically will be a combination of the visible screen area, perhaps some "buffer" or holding areas equal to the visible screen area, some working memory, and some additional storage. Video

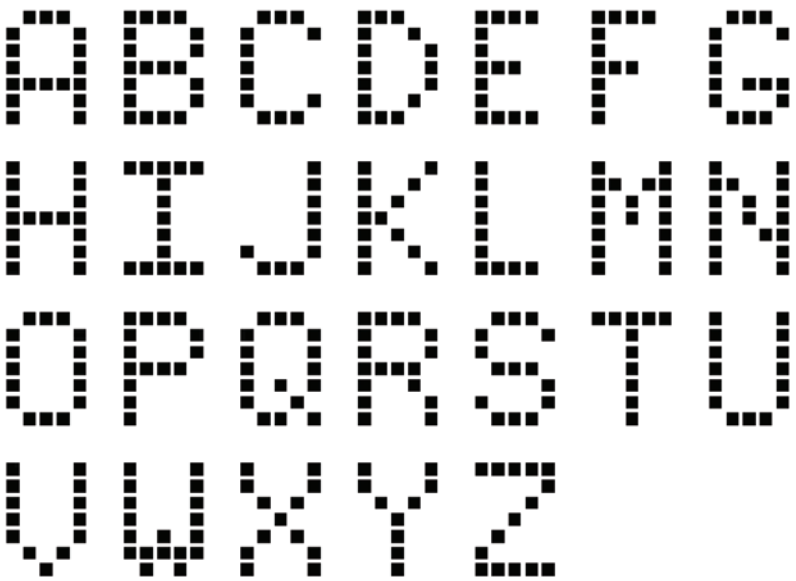

Figure 3.2: A 5 × 8 dot matrix alphabet. Illustrated by Kyle Flemmer.

https://doi.org/10.15215/remix/9781998944224.01

RAM is associated with the graphics-chip hardware and designed for fast-access, high-speed math (computer graphics use lots of that), and so we have graphics processing units (GPUs) to do the heavy lifting. All of this comes together to allow bytes of memory holding 1's and 0's to be displayed as a matrix of "bits" to create an image—hence "bitmaps."

The *screen resolution* refers to the dimensions of the screen in pixels (picture elements), which are basically display bits. A standard screen of 1920 × 1024 is 1920 bits wide and 1024 bits tall. Each one of those bits has a memory location and a colour depth (we'll talk about that in a bit), which can be referenced in most graphical programming languages by an x- and a y-coordinate. The top leftmost location is 0,0, and the bottom rightmost is the screen resolution value minus 1 (as we start at location 0)—so for our example, the screen coordinates 1919, 1023. While it may seem unintuitive to have a coordinate system that uses the top leftmost location as 0,0, this is a legacy association with old video display units (and our tendency to have a left-to-right frame of mind—for example, writing) for creating a screen image. Old computer monitor systems utilizing cathode ray tubes (CRTs) "painted" an image on-screen with an electron beam, starting from the top left, scanning left to right, and moving down line by line. This process then mapped well into "reading" and displaying computer memory, and so early computer graphics had a typical memory layout with locations in the left-to-right format.

Let's say we have a simple 4 × 4 display screen; our memory might look like table 4.

We could say our screen resolution here is 4 × 4: The top-left pixel coordinate is 0,0 and the bottom right is then 3,3 (as mentioned previously, X-1, Y-1). In our computer system this would be 16 bytes of VRAM—let's say a memory address location starting at $C000 in HEX and ending at $C00F (see table 5).

Note how the screen sequence of coordinates translates to the linear layout of the memory here so that the 3,3 display location lands in the $C00F memory address location. In its simplest description, game graphics is putting data onto

Table 4: Screen Coordinates

X →				
Y ↓	0,0	0,1	0,2	0,3
	1,0	1,1	1,2	1,3
	2,0	2,1	2,2	2,3
	3,0	3,1	3,2	3,3

https://doi.org/10.15215/remix/9781998944224.01

Table 5: Memory Address Locations

$C000	0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3
$C008	2,0	2,1	2,2	2,3	3,0	3,1	3,2	3,3

those VRAM memory locations, which correspond then to a change/display in pixels on the screen.

A single bit by itself would be rather dull, as it can represent only two states—on or off, such as a white area or a black dot. Bytes, however, introduce an additional element that adds variety to visual appearance: colour. In computer graphics, colour depth refers to the number of discrete levels or gradations available for representing colour information on a display. An 8-bit colour depth, for example, allows for 256 possible colour values; a 12-bit depth supports 4,096 values; a 16-bit depth supports 65,536 values; and higher depths allow for even greater colour precision.

In some cases, the value is simply a colour from a palette, so 1 of 256 choices, in other cases it might be the level of one of the primary colours (red, green, or blue—RGB), and the combination of those three values creates a colour. So an RGB value of 0,0,0 would be black (no level/brightness of RGB); to the other end of the spectrum (255,255,255) would be white (maximum RGB brightness).

Let's look back at our simple display example with a basic range of 256 colours—so a value of 64 is green, and let's use 128 for red. We will plot a border onto the screen, a green one with a red center—illustrating the VRAM memory of a bitmap screen object derived from table 4.

In VRAM memory, then we would have the values shown in table 6.

This mechanism of writing data (values) to a screen (display) through memory locations (RAM, VRAM) is the basic process for all screen graphics. When we write programs, we are using arrays of data, representing a graphic object—be it a bitmap screen image like a background, a defined area pattern like a square, or a circle like in our example. We refer to mobile bit array objects as "sprites," a bitmap that we typically move around on a larger object like the screen itself—the background, which could be a large static bitmap.

Table 6: Memory Address Example

$C000	64	64	64	64	64	128	128	64
$C008	64	128	128	64	64	64	64	64

https://doi.org/10.15215/remix/9781998944224.01

3.2. Bitmaps Creation and Use in Allegro 5

3.2.1. Bitmap Creation Tools

The game developer has a wide variety of tools and sources to acquire and create graphics utilized in game development. These range from self-developed graphics, to open-source and public domain works one could download, to purchasable artist-created graphical elements. Here, we'll talk briefly about some of the available options very generally, as they can vary greatly over time and depending on the manufacturer and operating systems used.

Images for use as bitmaps can be drawn, captured, or generated depending on the tools utilized. Most computer systems come with some form of simple graphical drawing program or image display/manipulation program. These programs may let you create stand-alone drawings, such as a Paint-type drawing package, or annotate/manipulate preexisting graphics like photographs.

After creation or modification, those items can then often be saved in differing graphics formats such as .jpg, .bmp, or .png. In the software used for working on images, there typically are functions that can transform the image via, for example, rotating, scaling, changing the number of colours (altering the bitmap colour depth—we'll talk about this later), or perhaps applying other effects. It is through the use of such tools that the game developer can often create the graphical elements required for their games, such as backgrounds and other in-game objects.

Game developers can also search online for more powerful, free open-source graphical tools to use in the development of their games. These can range from graphical editing programs such as GIMP to artificial intelligence image producers such as DALL-E. With these tools a game developer can create their own graphics for development use.

Intellectual Property and Acknowledging Its Use

An important point to keep in mind when creating and using graphics (or any game resource) is the intellectual property rights involved. If you are creating your own work drawings or photos, those tend to be straightforward, as you know the parties involved and can handle any issues regarding rights and use. When using online sources such as AI-created images or re-editing materials from a source other than your own, you need to consider the context of derived works and the rights and permissions that go along with those. As a game developer you will find a vast amount of available open-source and public domain materials for your work, and someone did you a favor by creating those; all they often ask in return is the proper acknowledgement of that work, a small price to pay for such generosity.

https://doi.org/10.15215/remix/9781998944224.01

3.2.2. Creating Bitmaps

As noted, one can often use the simple graphics tools that come with a system to create required bitmaps. Let's say I want to create a game about cats exploring space. I'll call it the Cat Astronaut Training Simulator, or CATS for short. The game will have a background of outer space, and for that, I'll use a star field photograph I've taken. The size of the images from my camera, resolution-wise, is 3008 × 2000, 24-bit colour, and the file size is 3 MB, so I'll want to shrink it down to, say, 1024 × 768 and perhaps reduce the colour depth to make it a smaller image.

3.2.3. Supported Bitmap Formats

One of the first questions you might ask at this point is, What does Allegro support for graphics in terms of formats? The choice of format will depend on the specific requirements of the application, such as the desired resolution, colour depth, and level of compression (to save on file space for your resources . . .). Here, the Allegro 5 development library supports a good variety of graphics formats, providing the developer with a wide selection.

Note that to use the bitmap graphics, we need to use another one of Allegro's many add-ons. Here, for the purpose of graphics, it's the image handling add-on, so in any program, we would include:

```
#include <allegro5/allegro_image.h>
```

And initialize that with:

```
bool al_init_image_addon(void)
```

Here is a brief overview of the supported graphics formats and their features:

BMP (Bitmap): BMP is a raster graphics format that is commonly used for images and animations. It supports a wide range of resolutions and colour depths, making it a versatile choice for game development.

PNG (Portable Network Graphics): PNG is a lossless image format that is widely used for web and mobile applications. It supports transparency, which allows for the creation of images with transparent backgrounds.

JPEG (Joint Photographic Experts Group): JPEG is a widely used lossy image format that is commonly used for photographs and other high-quality images. It supports high compression ratios, making it a good choice for applications where file size is a concern.

https://doi.org/10.15215/remix/9781998944224.01

GIF (Graphics Interchange Format): GIF is an animated image format that is commonly used for web and mobile applications. It supports animation, which allows for the creation of moving images.

TGA (True Colour Graphics): TGA is a lossless image format that is commonly used for professional applications. It supports high colour depths and is well suited for applications that require high-quality images.

The supported colour depths and resolutions for bitmaps are:

- 8-bit greyscale (1 colour)
- 16-bit greyscale (256 colours)
- 24-bit colour (16 million colours)
- 32-bit colour (4,294,967,296 colours)

The 32-bit colour depth is quite impressive but probably seldom used. (From the human perspective, could one even tell 4 billion colour differences from 16 million? And would the file size increase be worth it?)

Allegro 5 supports a wide range of screen resolutions as well. Here is a list of some of the commonly used screen resolutions that Allegro 5 supports:

- 1920 × 1080 (Full HD)
- 1600 × 900
- 1366 × 768
- 1280 × 720 (HD)
- 1024 × 768
- 800 × 600
- 640 × 480
- 512 × 384
- 400 × 300
- 320 × 240

Note that this is not an exhaustive list; Allegro 5 may support other screen resolutions as well.

3.2.4. Bitmap Information

In every operating system you might use, there will be a way to determine the information for an image, details one will need to know to utilize graphics in their game programs. If you've initialized your game to the wrong screen size or wrong colour depth, your graphics will not display properly. As such, it's prudent to find out that information before implementing a graphical object in your program to avoid errors, since Allegro will load your image fine and not

https://doi.org/10.15215/remix/9781998944224.01

generate an error, but when displayed, you won't see anything. This situation often leads one to believe that they have an error in their code, but it's actually a problem with the source image. Improper colour depth can also spoil the look of your game. For example, a 24-bit image rendered in 8-bit will look nothing like the original, losing both resolution and colour variation.

For this example, we will leave the colour depth at 24 but reduce the image size to 1024 × 768. I'm on a Windows 10 system, and so I can use the Photos program that comes with the system. The original image is in a .jpg format, 3008 × 2000, and 3 MB in size. To start with, I want to resize it and then convert it to a bitmap format (.bmp). In the Photos program, under the resize option, when I change the image size to 1024 × 768, it sets the height to 681 to maintain the aspect ratio. I'll live with that and see how it looks in the demo program, so I select the "save as" function to export the image as CatCosmos.bmp. The file size is now reduced somewhat to 2.75 MB (see figure 3.3).

3.2.5. Choices, Choices . . .

Let's talk a bit about file types here: the pros and cons of JPEGs and bitmaps. Allegro's library can handle both formats, but its strength lies in using bitmaps. Bitmaps and JPEGs are both image file formats, but they have some key differences that can affect their performance and usage in certain situations. Here are five potential advantages of using bitmaps over JPEGs; the last three points listed are key for our needs as game developers:

Figure 3.3: CatCosmos.bmp for use as a background image. Photograph by Walter Ridgewell.

https://doi.org/10.15215/remix/9781998944224.01

Higher Resolution: Bitmaps are typically used for images that require high resolution, such as logos, icons, and other graphical elements. Bitmaps can have a higher resolution than JPEGs, which means they can display more pixels per inch (PPI) and provide a clearer, sharper image.

Lossless Compression: Bitmaps use lossless compression, which means that the original image data is preserved during compression. This is important for images that need to be edited or manipulated in the future, as the original data can be restored without any loss of quality. JPEGs, however, use lossy compression, which means that some of the original data is discarded during compression, and the resulting image may not be as sharp or detailed as the original.

Transparency: Bitmaps can support transparency, which means that certain parts of the image can be made transparent or semitransparent. This is useful for creating images with overlays or other visual effects that require transparency. JPEGs do not support transparency.

Flexibility: Bitmaps can be used in a variety of contexts, including web design, graphic design, and game development. They can be scaled up or down to different sizes without losing quality, and they can be used in a variety of software applications. JPEGs are primarily used for web images and are not as versatile as bitmaps.

Smaller File Size: Bitmaps can be smaller in file size than JPEGs, especially in images that do not require high resolution or complex colour schemes. This is because bitmaps use a different compression algorithm than JPEGs, which allows them to use fewer bits per pixel and results in smaller file sizes. However, this comes at the cost of lower resolution and less detail in the image.

3.3. Utilizing Bitmaps in a Program

3.3.1. Loading Bitmaps

To use graphics in programs, we will call various routines in the Allegro 5 library. To load a bitmap in Allegro 5, we will use the al_load_bitmap routine. To load a 24-bit colour image like the one we created previously, CatCosmos.bmp, you would call

```
ALLEGRO_BITMAP *image = al_load_bitmap("CatCosmos.bmp");
```

Here, ALLEGRO_BITMAP is a *struct*, a custom container representing the bitmap image, and *image is our pointer to the routine call, along with our filename. A short program to display the image follows: BitmapDisplay.c.

https://doi.org/10.15215/remix/9781998944224.01

3.3.2. Complete Coding Example

```
// BitmapDisplay.c
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_native_dialog.h>
#include <allegro5/allegro_primitives.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>

int main(int argc, char** argv)
{
    // Initialize Allegro—checks if Allegro is properly
initialized.
    if (!al_init()) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to initialize Allegro!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        return—1;
    }

    // Initialize the display—creates a display window of
size 1024 × 768
    ALLEGRO_DISPLAY* display = al_create_display(1024,
768);
    if (!display) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to create display!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        return—1;
    }

    // Initialize the image addon to load bitmap images.
    if (!al_init_image_addon()) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to initialize image addon!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        al_destroy_display(display);
        return—1;
    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    // Loads the image file CatCosmos.bmp.
    ALLEGRO_BITMAP* image = al_load_bitmap("CatCosmos.
bmp");
    if (!image) {
        al_show_native_message_box(NULL,
"Error", "Error", "Failed to load image!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        al_destroy_display(display);
        return-1;
    }

    // Initialize the keyboard for input handling
    if (!al_install_keyboard()) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to install keyboard!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        al_destroy_bitmap(image);
        al_destroy_display(display);
        return-1;
    }

    // Create an event queue to handle our
program  events.
    ALLEGRO_EVENT_QUEUE* event_queue =
al_create_event_queue();
    if (!event_queue) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to create event queue!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        al_destroy_bitmap(image);
        al_destroy_display(display);
        return-1;
    }

    // Register the display and keyboard to the event
queue
    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_keyboard_event_source());
```

https://doi.org/10.15215/remix/9781998944224.01

```
    // Clear the display to white
    al_clear_to_color(al_map_rgb(255, 255, 255));

    // Draw the image on the display
    al_draw_bitmap(image, 0, 0, 0);

    // Flip the display to show the image
    al_flip_display();

    // Event loop—keeps the display open and listens for
the ESC key press or window close event to exit the program

    bool running = true;
    while (running) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(event_queue, &ev);

        if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            running = false;
        }
        else if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
            if (ev.keyboard.keycode == ALLEGRO_KEY_
ESCAPE) {
                running = false;
            }
        }
    }

    // Cleanup
    al_destroy_bitmap(image);
    al_destroy_display(display);
    al_destroy_event_queue(event_queue);

    return 0;
}
```

3.3.3. Manipulating Bitmap Images

While you can manipulate bitmap images in external graphic programs, as we have mentioned previously, you can also do some basic image transformations using the Allegro library on a loaded image in your program.

https://doi.org/10.15215/remix/9781998944224.01

There are two other transformations—scaling (or changing an image's size) and rotating (changing the orientation of the displayed image). Allegro provides a few routines representing variations and combinations of those, along with incorporating tinting, so we have:

- al_draw_scaled_bitmap
- al_draw_rotated_bitmap
- al_draw_tinted_bitmap
- al_draw_scaled_rotated_bitmap
- al_draw_tinted_scaled_bitmap
- al_draw_tinted_rotated_bitmap
- al_draw_tinted_scaled_rotated_bitmap

We'll just consider the base routine calls (scaled, rotated, and tinted) here, as the combined routines simply add parameters from those base calls.

3.4. Scaling and Rotating Bitmaps

To begin with, we can change the size of the displayed image with al_draw_scaled_bitmap:

```
void al_draw_scaled_bitmap(ALLEGRO_BITMAP *bitmap, float
sx, float sy, float sw, float sh, float dx, float dy,
float dw, float dh, int flags)
```

This draws a scaled version of the given bitmap, where:

sx: source x
sy: source y
sw: source width
sh: source height
dx: destination x
dy: destination y
dw: destination width
dh: destination height
flags: same as for al_draw_bitmap

The initial parameters for the routine describe characteristics of the unmodified bitmap, the origin x,y of bitmap source, and the width and height of the object. We then specify how we want to scale those parameters, providing a new destination (x,y) and its associated width and height change. These

https://doi.org/10.15215/remix/9781998944224.01

variables let us scale the image in a variety of ways, expanding or shrinking it as we wish (see figure 3.4).

Next, we can also change the orientation of that image with al_draw_rotated_bitmap:

```
void al_draw_rotated_bitmap(ALLEGRO_BITMAP *bitmap, float
cx, float cy, float dx, float dy, float angle, int flags)
```

where

cx: center
cy: center y
dx: destination x
dy: destination y
angle: angle by which to rotate
flags: same as for al_draw_bitmap

This draws a rotated version of the specified bitmap. Again, we see the use of some familiar variables with respect to destination x- and y-coordinates. As we are rotating an image about a point, we need the center x- and y-coordinates and an amount to rotate the bitmap by.

In Allegro, we specify an "angle" described using radians rather than degrees, with the rotation going clockwise (turning to the right . . .) around the axis (center-point coordinates).

The point defined by cx/cy is the rotation point inside the bitmap; the image will be drawn at dx/dy on the display, and the bitmap is rotated around this point 90 degrees clockwise. If the cx/cy point is in the center of the object, it's rotated on that point; if it's on an edge or corner, the object is rotated at the point.

Figure 3.5 shows two examples rotated 45 degrees.

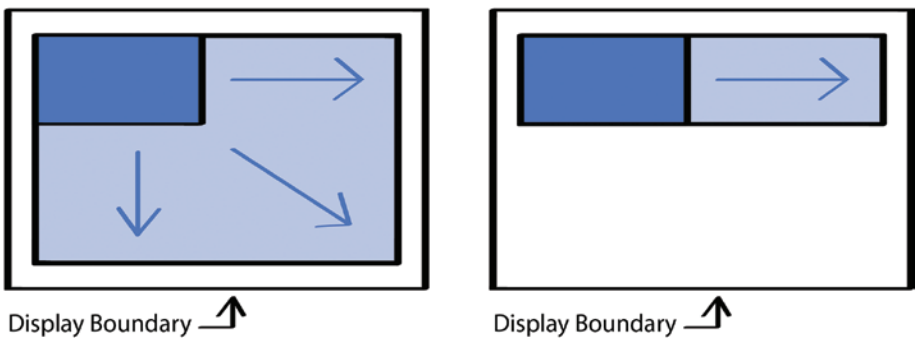

Figure 3.4: Scaling bitmaps. Illustration by Kyle Flemmer.

https://doi.org/10.15215/remix/9781998944224.01

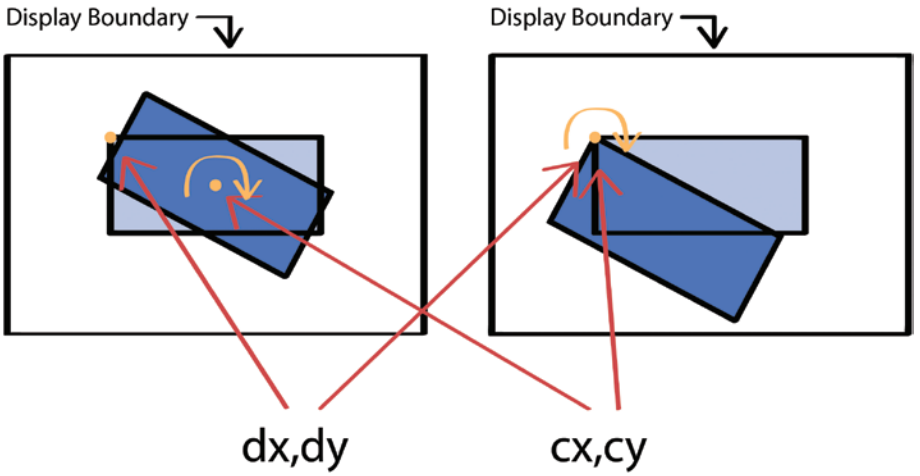

Figure 3.5: Bitmaps rotated by 45 degrees. Illustrated by Kyle Flemmer.

In these two examples, we see how the coordinates affect where the rotation occurs. In the first we have different coordinates for the destination x,y of the bitmap (where it's drawn from dx and dy) and the rotation point in that object (cx and cy, respectively). In the first example, the rotation point is at the center of the object. In the second example, the destination dx,dy and the rotation point cx,cy are the same. Now let's advance the rotation to 90 degrees.

As you can see in figure 3.6, the location of the center point for rotation makes a difference in the appearance of the image.

Let's take a diversion here from the example and into the use of radians for the angle, as opposed to degrees, which is what most people will be used to. Degrees and radians are both units for measuring angles. Degrees divide a circle into 360 parts, making them intuitive and easy to use in everyday

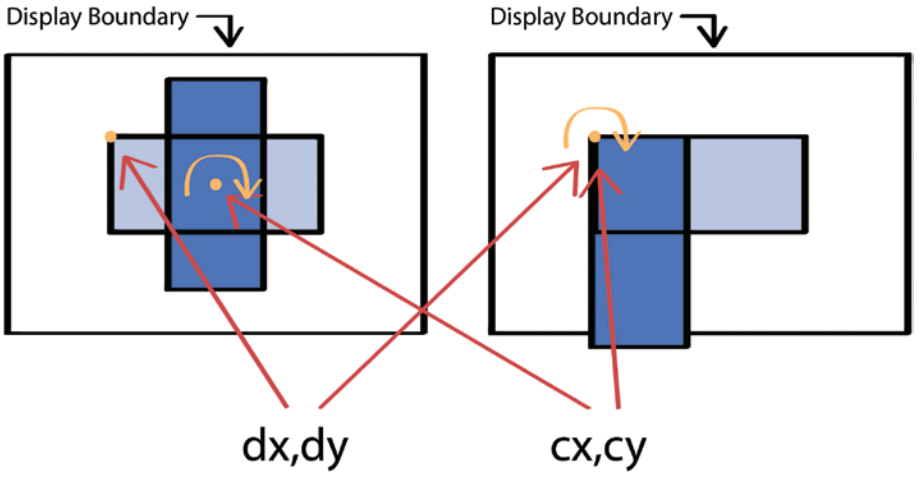

Figure 3.6: Bitmaps rotated by 90 degrees. Illustrated by Kyle Flemmer.

https://doi.org/10.15215/remix/9781998944224.01

contexts like navigation and geography. Radians, however, measure angles based on a circle's radius, with one radian being the angle subtended by an arc equal in length to the radius. So that means there are 2π radians in a full circle (360 degrees) and π radians in a half circle (180 degrees). Radians are preferred in computer graphics because they simplify mathematical formulas, improve computational efficiency, and provide consistency in calculations involving trigonometric functions and rotational transformations. This leads to faster and more accurate rendering of graphics. As a result, the base math library in C (and many other programming languages) requires the use of radians rather than degrees. This is a simple example of a conversion program:

```
#include <stdio.h>
#define PI 3.14159265358979323846

int main(void) {
    // Angle in degrees
    double degrees = 45.0;
    // Conversion: radians = degrees * (PI / 180)
    double radians = degrees * (PI / 180.0);
    // Output the result
    printf("%.2f degrees is equal to %.5f radians.\n",
degrees, radians);
    return 0;
}
```

So rotating an object by 90 degrees in a graphics program translates to

$\pi/2$ radians (π being the equivalent of 180 degrees)

Allegro has a struct ALLEGRO_PI, and so to rotate an object by 90 degrees, you would use a routine call such as:

```
al_draw_rotated_bitmap(bitmap, cx, cy, dx, dy, ALLEGRO_PI
/ 2, 0)
```

We'll see how this looks in the following demonstration program example BitmapDisplayRotate.c. Here, every press of the space bar rotates the background image 90 degrees clockwise.

https://doi.org/10.15215/remix/9781998944224.01

3.4.1. Complete Coding Example

```
// BitmapDisplayRotate.c
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_native_dialog.h>
#include <allegro5/allegro_primitives.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>

int main(int argc, char** argv)
{
    // Initialize Allegro.
    if (!al_init()) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to initialize Allegro!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        return-1;
    }

    // Create a display (window) of size 1024 × 768.
    ALLEGRO_DISPLAY* display = al_create_display(1024,
768);
    if (!display) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to create display!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        return-1;
    }

    // Initialize the image addon to load bitmap images.
    if (!al_init_image_addon()) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to initialize image addon!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        al_destroy_display(display);
        return-1;
    }

    // Load the bitmap image "CatCosmos.bmp".
    ALLEGRO_BITMAP* image = al_load_bitmap("CatCosmos.bmp");
```

https://doi.org/10.15215/remix/9781998944224.01

```
    if (!image) {
        al_show_native_message_box(NULL,
"Error", "Error", "Failed to load image!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        al_destroy_display(display);
        return-1;
    }

    // Initialize the keyboard for input handling.
    if (!al_install_keyboard()) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to install keyboard!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        al_destroy_bitmap(image);
        al_destroy_display(display);
        return-1;
    }

    // Create an event queue for handling events.
    ALLEGRO_EVENT_QUEUE* event_queue =
al_create_event_queue();
    if (!event_queue) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to create event queue!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        al_destroy_bitmap(image);
        al_destroy_display(display);
        return-1;
    }

    // Register the display and keyboard event sources.
    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_keyboard_event_source());

    // Show initial instructions.
    al_show_native_message_box(display,
        "Instructions",
        "Bitmap Rotation",
```

https://doi.org/10.15215/remix/9781998944224.01

```
        "Press SPACE to rotate the image 90° clockwise.\
nPress ESCAPE to exit.",
        NULL,
        0);

    // Variables for rotation and positioning.
    double angle = 0.0;
    int image_width = al_get_bitmap_width(image);
    int image_height = al_get_bitmap_height(image);
    int display_center_x = 1024 / 2;
    int display_center_y = 768 / 2;

    // Draw the initial image (no rotation) centered in
the display.
    al_clear_to_color(al_map_rgb(255, 255, 255));
    al_draw_rotated_bitmap(image,
        image_width / 2.0,   // Rotation center x (center
of image)
        image_height / 2.0,  // Rotation center y (center
of image)
        display_center_x,    // Destination x (center of
display)
        display_center_y,    // Destination y (center of
display)
        angle,               // Rotation angle (in
radians)
        0);
    al_flip_display();

    // Main event loop.
    bool running = true;
    while (running)
    {
        ALLEGRO_EVENT ev;
        al_wait_for_event(event_queue, &ev);

        if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            running = false;
        }
        else if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
            if (ev.keyboard.keycode == ALLEGRO_KEY_
ESCAPE) {
                running = false;
            }
            else if (ev.keyboard.keycode == ALLEGRO_KEY_
SPACE) {
                // Rotate the image 90° clockwise (90° =
PI/2 radians).
                angle += ALLEGRO_PI / 2;
                if (angle >= 2 * ALLEGRO_PI)
                    angle-= 2 * ALLEGRO_PI;

                // Redraw the image with the new
rotation.
                al_clear_to_color(al_map_rgb(255, 255,
255));
                al_draw_rotated_bitmap(image,
                    image_width / 2.0,
                    image_height / 2.0,
                    display_center_x,
                    display_center_y,
                    angle,
                    0);
                al_flip_display();
            }
        }
    }

    // Cleanup resources.
    al_destroy_bitmap(image);
    al_destroy_display(display);
    al_destroy_event_queue(event_queue);

    return 0;
}
```

3.4.2. Bitmap Colour Manipulation

On to our last base routine: al_draw_tinted_bitmap. Its purpose is to create the ability to "highlight" a specific colour in the bitmap (red, green, blue) by multiplying all colours in the bitmap with the highlight colour. The routine also

https://doi.org/10.15215/remix/9781998944224.01

allows for setting the opacity of the bitmap from solid to transparent. While changing the transparency of a bitmap image in your game may not seem useful immediately, it is a handy function for visual effects when utilized with sprites in games (an example could be of a game character that gets "zapped" by a laser and then disintegrates—fades away, a change in transparency). In relation to the background, perhaps as you change scenes, you might fade out from one and fade into the next. The routine's structure is:

```
void al_draw_tinted_bitmap(ALLEGRO_BITMAP *bitmap, tint,
float dx, float dy, int flags)
```

The "tint value" we're talking about comes from the ALLEGRO_COLOR that you create with the routine al_map_rgba_f(float r, float g, float b, float a). This routine takes four floating-point numbers (each between 0 and 1) representing the red, green, blue, and alpha (transparency) channels, respectively. Here is an example:

```
al_draw_tinted_bitmap(bitmap, al_map_rgba_f(1, 1, 1,
0.5), x, y, 0);
```

In this case, you're drawing the bitmap with full intensity on the red, green, and blue channels (that's what the 1's mean), but the alpha value is set to 0.5. This results in the bitmap being rendered at 50% transparency.

Another example is:

```
al_draw_tinted_bitmap(bitmap, al_map_rgba_f(1, 0, 0, 1),
x, y, 0);
```

Here, the tint is specified to use only the red component (1 for red, 0 for green, 0 for blue) while keeping full opacity (alpha is 1). This means the bitmap gets tinted red.

So essentially, by adjusting the values you pass to al_map_rgba_f, you control both the transparency and the colour tint applied to your bitmap when you draw it.

3.5. Displaying a Section of a Bitmap and the Concept of a Viewport

Allegro 5 will let the user display a selected area of a loaded bitmap, the size of which is determined by the programmer, and have that available for another graphics display. Here, the most recognizable use of this routine would be the on-screen minimap display.

https://doi.org/10.15215/remix/9781998944224.01

Here, the main loaded bitmap image comprises the whole map or game level, with sections of it outside the main display screen; this is your viewport. We then add to that viewport a minimap, which might show all of the existing map (scaled down), and then a highlighted section indicating which area is currently being seen on the main display.

What follows is an example program, DisplayMap.c, which loads a background image, displays a section of it on the main screen (the viewport) with a minimap showing a scaled-up version of the complete image.

3.5.1. Example Program DisplayMap.c

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_native_dialog.h>
#include <allegro5/allegro_primitives.h>
#include <allegro5/allegro_font.h>

int main(int argc, char** argv)
{
    if (!al_init()) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to initialize Allegro!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        return-1;
    }

    if (!al_init_image_addon()) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to initialize image addon!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        return-1;
    }

    if (!al_init_primitives_addon()) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to initialize primitives addon!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        return-1;
    }

    const int display_w = 1024;
    const int display_h = 768;
```

https://doi.org/10.15215/remix/9781998944224.01

```
    ALLEGRO_DISPLAY* display = al_create_
display(display_w, display_h);
    if (!display) {
        al_show_native_message_box(NULL, "Error",
"Error", "Failed to create display!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        return-1;
    }

    ALLEGRO_BITMAP* map = al_load_bitmap("CatCosmos.bmp");
    if (!map) {
        al_show_native_message_box(display,
"Error", "Error", "Failed to load map.bmp!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        al_destroy_display(display);
        return-1;
    }

    int map_w = al_get_bitmap_width(map);
    int map_h = al_get_bitmap_height(map);

    int mini_x = display_w-210;
    int mini_y = 10;
    int mini_w = 200;
    int mini_h = 150;

    int mini_view_x = 0;
    int mini_view_y = 0;
    int mini_view_w = mini_w / 8;
    int mini_view_h = mini_h / 8;
    const int mini_scroll_speed = 5;

    if (!al_install_keyboard()) {
        al_show_native_message_box(display, "Error",
"Error", "Failed to install keyboard!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        al_destroy_bitmap(map);
        al_destroy_display(display);
        return-1;
    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    ALLEGRO_EVENT_QUEUE* event_queue =
al_create_event_queue();
    if (!event_queue) {
        al_show_native_message_box(display, "Error",
"Error", "Failed to create event queue!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
        al_destroy_bitmap(map);
        al_destroy_display(display);
        return-1;
    }

    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_keyboard_event_source());

    ALLEGRO_TIMER* timer = al_create_timer(1.0 / 60);
    al_register_event_source(event_queue,
al_get_timer_event_source(timer));
    al_start_timer(timer);

    bool running = true;
    ALLEGRO_EVENT ev;

    while (running)
    {
        al_wait_for_event(event_queue, &ev);

        if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            running = false;
        }
        else if (ev.type == ALLEGRO_EVENT_KEY_DOWN)
        {
            if (ev.keyboard.keycode ==
ALLEGRO_KEY_ESCAPE)
            {
                running = false;
            }
            else if (ev.keyboard.keycode ==
ALLEGRO_KEY_LEFT)
```

https://doi.org/10.15215/remix/9781998944224.01

```
            {
                mini_view_x-= mini_scroll_speed;
                if (mini_view_x < 0) mini_view_x = 0;
            }
            else if (ev.keyboard.keycode ==
ALLEGRO_KEY_RIGHT)
            {
                mini_view_x += mini_scroll_speed;
                if (mini_view_x > mini_w-mini_view_w)
mini_view_x = mini_w-mini_view_w;
            }
            else if (ev.keyboard.keycode ==
ALLEGRO_KEY_UP)
            {
                mini_view_y-= mini_scroll_speed;
                if (mini_view_y < 0) mini_view_y = 0;
            }
            else if (ev.keyboard.keycode ==
ALLEGRO_KEY_DOWN)
            {
                mini_view_y += mini_scroll_speed;
                if (mini_view_y > mini_h-mini_view_h)
mini_view_y = mini_h-mini_view_h;
            }
        }
        else if (ev.type == ALLEGRO_EVENT_TIMER)
        {
            al_clear_to_color(al_map_rgb(0, 0, 0));

            al_draw_bitmap(map, 0, 0, 0);

            al_draw_scaled_bitmap(map,
                mini_view_x * (map_w / mini_w), mini_
view_y * (map_h / mini_h),
                mini_view_w * (map_w / mini_w), mini_
view_h * (map_h / mini_h),
                mini_x, mini_y, mini_w, mini_h, 0);

            al_draw_rectangle(mini_x, mini_y, mini_x +
mini_w, mini_y + mini_h, al_map_rgb(255, 255, 255), 2);
```

https://doi.org/10.15215/remix/9781998944224.01

```
            al_draw_rectangle((mini_view_x * (map_w /
mini_w)), (mini_view_y * (map_h / mini_h)), ((mini_view_x
+ mini_view_w) * (map_w / mini_w)), ((mini_view_y + mini_
view_h) * (map_h / mini_h)), al_map_rgb(255, 255, 255),
2);
            al_draw_textf(al_create_builtin_font(), al_
map_rgb(255, 255, 255), 10, 10, 0, "mini_x: %d, mini_y:
%d, mini_w: %d, mini_h: %d", mini_x, mini_y, mini_w,
mini_h);

            al_flip_display();
        }
    }

    al_destroy_timer(timer);
    al_destroy_event_queue(event_queue);
    al_destroy_bitmap(map);
    al_destroy_display(display);

    return 0;
}
```

3.5.2. Program Overview

There is a lot going on in this next example: We're loading a bitmap; magnifying a section of it, which is selected by a moving window; and displaying that in a window on the main bitmap display. As such, let's go through it in more detail.

Breaking It Down Step-by-Step

1. Initializing Allegro

Before doing anything graphical, Allegro must be initialized:

```
if (!al_init()) {
    al_show_native_message_box(NULL, "Error",
"Error", "Failed to initialize Allegro!", NULL,
ALLEGRO_MESSAGEBOX_ERROR);
    return–1;
}
```

This ensures that Allegro is ready to be used. If it fails, an error message pops up.

https://doi.org/10.15215/remix/9781998944224.01

Next, the necessary Allegro add-ons are initialized:

```
al_init_image_addon();
al_init_primitives_addon();
```

The image add-on lets us load and draw bitmap images.

The primitives add-on is needed for drawing shapes like rectangles.

2. Creating the Display

```
const int display_w = 1024;
const int display_h = 768;
ALLEGRO_DISPLAY* display = al_create_display(display_w,
display_h);
```

This creates a 1024 × 768 pixel window for the game. If it fails, the program exits.

3. Loading the Map

```
ALLEGRO_BITMAP* map = al_load_bitmap("CatCosmos.bmp");
```

The map image (CatCosmos.bmp) is loaded into memory.

If the file is missing or incorrectly formatted, the program stops.

4. Setting Up the Minimap

```
int mini_x = display_w–210;
int mini_y = 10;
int mini_w = 200;
int mini_h = 150;
```

This defines the minimap's position and size.

The minimap is placed in the top-right corner with a 10-pixel margin.

The viewport within the minimap:

```
int mini_view_x = 0, mini_view_y = 0;
int mini_view_w = mini_w / 2, mini_view_h = mini_h / 2;
```

The viewport inside the minimap starts at (0,0).

It's set to half the minimap's width and height, which creates a "zoomed-in" effect.

https://doi.org/10.15215/remix/9781998944224.01

5. Setting Up Keyboard Input

```
al_install_keyboard();
ALLEGRO_EVENT_QUEUE* event_queue =
al_create_event_queue();
al_register_event_source(event_queue,
al_get_keyboard_event_source());
```

The keyboard is installed so we can detect arrow key presses.

The event queue listens for keyboard inputs.

6. Scrolling the Minimap

The program listens for keyboard events:

```
if (ev.keyboard.keycode == ALLEGRO_KEY_LEFT)
{
    mini_view_x-= mini_scroll_speed;
    if (mini_view_x < 0) mini_view_x = 0;
}
```

Pressing the left arrow key moves the minimap viewport left.

The viewport is prevented from scrolling beyond the left boundary (0,0).

Similarly, the Right, Up, and Down keys move the viewport in different directions.

7. Rendering the Scene

Inside the game loop, we clear the screen:

```
al_clear_to_color(al_map_rgb(0, 0, 0));
```

This ensures we don't get unwanted artifacts from previous frames.

The full-sized map is drawn:

```
al_draw_bitmap(map, 0, 0, 0);
```

The minimap is drawn by scaling the full map down:

```
al_draw_scaled_bitmap(map,
    mini_view_x * (map_w / mini_w), mini_view_y * (map_h
/ mini_h),
```

https://doi.org/10.15215/remix/9781998944224.01

```
    mini_view_w * (map_w / mini_w), mini_view_h * (map_h
/ mini_h),
mini_x, mini_y, mini_w, mini_h, 0);
```

Instead of drawing the entire map, only a selected portion is drawn inside the minimap.

8. Highlighting the Minimap Viewport

```
al_draw_rectangle((mini_view_x * (map_w / mini_w)),
                  (mini_view_y * (map_h / mini_h)),
                  ((mini_view_x + mini_view_w) * (map_w /
mini_w)),
                  ((mini_view_y + mini_view_h) * (map_h /
mini_h)),
                  al_map_rgb(255, 255, 255), 2);
```

This rectangle highlights the portion of the minimap that is currently being viewed.

The example program displays the main graphic bitmap with a smaller movable viewport indicator: here, a white rectangle and a static viewport in the top-right corner showing an enlarged image of the bitmap in the rectangle area.

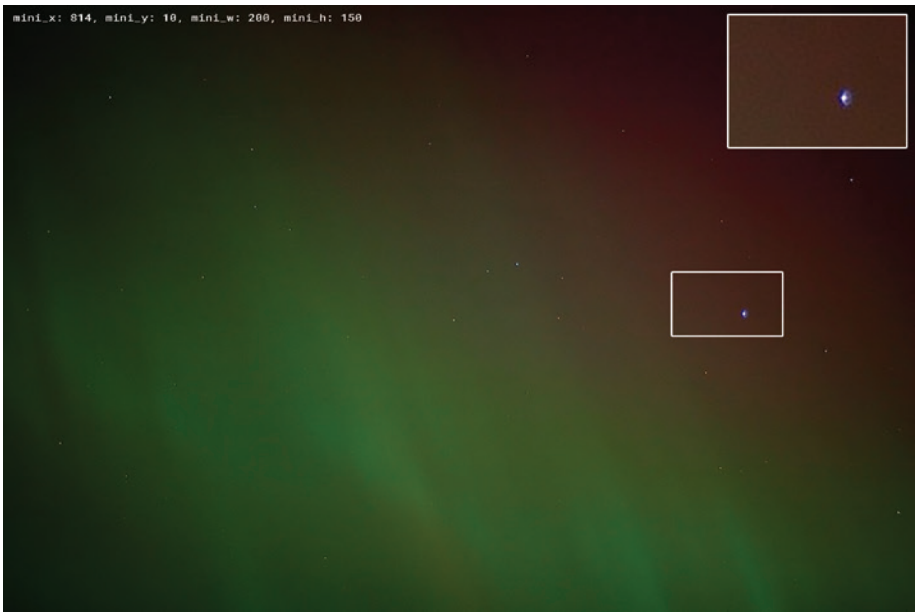

Figure 3.7: Display map with a minimap viewport. Image by Walter Ridgewell.

https://doi.org/10.15215/remix/9781998944224.01

3.6. Advanced Graphics Techniques in Allegro 5

3.6.1. Resolution Independence

Resolution independence refers to the ability of a software application to run on displays of different resolutions without losing quality or functionality. In Allegro 5, achieving resolution independence involves scaling graphics and user interface elements dynamically based on the screen's resolution. This ensures that the application looks consistent and sharp across various devices. The typical approach to this is to utilize routines to query the platform the device is running on for display information and then assign those values to appropriate variables in your program—for example, screen_width and screen_height. This looks something like:

```
int screen_width = al_get_display_width(ALLEGRO_DISPLAY
*display)
in screen_height = al_get_display_height(ALLEGRO_DISPLAY
*display)
```

After that, one would optimize routines for screen coordinates to work with the returned information. This can prove to be a complex process if you're wanting to work with a variety of screen sizes, keep aspect ratios, and so on.

In Allegro 5, one can use transformations to dynamically adapt one's code to the current system's screen resolution. By using a transformation that considers screen resolution, you can ensure that graphics scale properly and dynamically. This would be implemented utilizing the following library functions:

```
al_identity_transform(&transform);
al_scale_transform(&transform, scaleX, scaleY);
al_use_transform(&transform);
```

where scaleX and scaleY are the scaling factors for width and height (respectively) that take into account the commonly used screen_width and screen_height information:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>

int main() {
    if (!al_init()) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
        return-1;
    }
    al_init_image_addon();
    al_init_primitives_addon();
    ALLEGRO_DISPLAY *display = al_create_display(1024,
768);
    if (!display) {
        return-1;
    }
    // Query display dimensions
    int screen_width = al_get_display_width(display);
    int screen_height = al_get_display_height(display);
    // Setup transformation for resolution independence
    float scaleX = (float)screen_width / 1024.0;
    float scaleY = (float)screen_height / 768.0;
    ALLEGRO_TRANSFORM transform;
    al_identity_transform(&transform);
    al_scale_transform(&transform, scaleX, scaleY);
    al_use_transform(&transform);
    al_clear_to_color(al_map_rgb(0, 0, 0));
    al_flip_display();
    al_rest(2.0);
    al_destroy_display(display);
    return 0;
}
```

But we're a bit ahead of ourselves here, as we haven't explained transformations. In Allegro 5, these are routines used to modify the position, scale, rotation, and other properties of graphical objects in a 2D space. These effects can be applied to multiple bitmap objects at once, alleviating the need to otherwise perform those actions individually per bitmap object.

Transformations are used extensively in game development and graphics programming. They allow for dynamic object movement, animations, camera movement, and more. For instance, in a platformer game, transformations are used to move characters, enemies, and the camera that follows the player. As such, transformations in Allegro 5 are powerful tools that allow developers to manipulate graphics in 2D space. They are essential for creating dynamic, interactive, and visually appealing applications. By understanding and using translation, scaling, and rotation, and combining these transformations

https://doi.org/10.15215/remix/9781998944224.01

effectively, developers can greatly enhance the visual quality and interactivity of their Allegro 5 projects.

3.6.2. Types of Transformations

Looking at the list of transformation routines here, you'll note that the first three have equivalent routines to some of the earlier bitmap manipulation operations:

Translation (Moving Objects): This changes the position of an object in the 2D space. For example, moving an object from one location to another.

Scaling (Resizing Objects): Scaling either increases or decreases the size of an object. This can be uniform (keeping the aspect ratio the same) or nonuniform (changing the width and height by different amounts).

Rotation (Spinning Objects): This involves rotating an object around a point, typically its center. Rotation is usually measured in degrees or radians.

Shearing: Shearing is a less commonly used transformation that skews the shape of an object.

Let's consider scaling. Using transformations in Allegro 5 (such as al_rotate_transform), vs. relying on simpler drawing commands like al_draw_rotated_bitmap, comes down to flexibility, performance, and ease of applying multiple transformations at once.

Here's a breakdown of why you might prefer one over the other:

1. **Flexibility with Multiple Transformations**

   Using al_rotate_transform and the transformation matrix (ALLEGRO_TRANSFORM)

   If you need to apply multiple transformations together (e.g., rotation, scaling, and translation), a transformation matrix allows you to combine all transformations before drawing.

   Example: You can scale, rotate, and translate an object in a single transformation rather than doing them separately with different routines.

   Using al_draw_rotated_bitmap

   This routine is simpler but only rotates around a specific point when drawing.

   If you need additional transformations, like scaling or skewing, you cannot do that easily with al_draw_rotated_bitmap.

https://doi.org/10.15215/remix/9781998944224.01

2. **Performance Considerations**
   Using al_draw_rotated_bitmap
   This is optimized for performance when all you need is rotation.
   The routine is simpler and faster if you don't need complex transformations.
   It's great for simple sprite rotation, such as rotating a spaceship or a top-down character.
   Using al_rotate_transform
   If you apply multiple transformations, using a transformation matrix is often more efficient than applying multiple individual drawing operations.
   It reduces redundant calculations since transformations can be precomputed and reused.
3. **Transforming the Entire Scene vs. Individual Objects**
   Using al_use_transform with al_rotate_transform
   If you need to apply transformations to the entire coordinate system, using al_use_transform allows all subsequent drawing operations to inherit the transformation.
   Example: If you want to rotate everything in a scene (like a rotating camera view), a transformation matrix is the correct approach.
   Using al_draw_rotated_bitmap
   If you only need to rotate one object while keeping everything else unchanged, al_draw_rotated_bitmap is easier.
4. **Example Comparisons**
   Using al_draw_rotated_bitmap (Simpler, Object-Specific)

```
ALLEGRO_BITMAP *sprite = al_load_bitmap("sprite.png");
float x = 200, y = 200; // Position on-screen
float angle = ALLEGRO_PI / 4; // Rotate 45 degrees
float cx = al_get_bitmap_width(sprite) / 2.0;
float cy = al_get_bitmap_height(sprite) / 2.0;

al_draw_rotated_bitmap(sprite, cx, cy, x, y, angle,
0);
```

   This rotates only one bitmap at a time. It's good for rotating sprites but not for complex transformations.
   Using al_rotate_transform with al_use_transform (More Advanced, Affects All Drawing)

https://doi.org/10.15215/remix/9781998944224.01

```
ALLEGRO_TRANSFORM transform;
al_identity_transform(&transform);
al_translate_transform(&transform,-100,-100); // Move
pivot point
al_rotate_transform(&transform, ALLEGRO_PI / 4); //
Rotate 45 degrees
al_translate_transform(&transform, 100, 100); // Move
back
al_use_transform(&transform);

al_draw_bitmap(sprite, 200, 200, 0);
```

This approach affects all drawing commands after al_use_transform(). It's useful for scene-wide effects or handling multiple transformations at once.

As you can see, transforms are useful for more complex operations and can affect a whole screen view, which brings us back to why we might want to use them to achieve display resolution independence in our games.

3.7. How Transformations Help with Resolution Independence

Perhaps one of the most critical application transformations in games is that they allow you to remap coordinates so that your game logic can stay consistent across different resolutions. This is important for a developer if they choose to create a game for multiple platforms while minimizing coding work. Let's look at some of those possible uses.

Table 7: When to Use Which?

Situation	Use al_draw_rotated_bitmap	Use al_rotate_transform
Rotate a single sprite	✓	×
Rotate multiple objects together	×	✓
Rotate and scale at the same time	×	✓
Move a camera or worldview	×	✓
Simple, fast sprite rotation	✓	×

https://doi.org/10.15215/remix/9781998944224.01

3.7.1. Scaling the Entire Scene

By using al_scale_transform(), you can make the coordinate system of your game match the target resolution dynamically. This means that drawing at coordinates (100,100) on a small window will appear in the same relative position on a larger window.

The following code sample scales from a base resolution of 800 × 600 to fit any screen size:

```
// Base resolution
float base_width = 800;
float base_height = 600;

// Get actual screen size
float screen_width = al_get_display_width(display);
float screen_height = al_get_display_height(display);

// Compute scaling factors
float scale_x = screen_width / base_width;
float scale_y = screen_height / base_height;

ALLEGRO_TRANSFORM transform;
al_identity_transform(&transform);
al_scale_transform(&transform, scale_x, scale_y);
al_use_transform(&transform);
```

So here in this code segment, if the display size of the platform running the code is 1600 × 1200, the values supplied to the transform would be scale_x = 2 and scale_y = 2, meaning everything is doubled in size.

Conversely, if the screen is 400 × 300 (like on a small mobile device platform), the calculated values of scale_x = 0.5 and scale_y = 0.5 would be applied, meaning everything is shrunk to fit. As a result, all your drawing code remains the same regardless of screen size.

3.7.2. Centering the Game on Any Screen Size

If you scale up or down based on resolution, sometimes you also need to center your game. This can be done by translating the coordinate system.

EXAMPLE: CENTERING THE SCALED CONTENT

```
// Compute translation to center the game
float offset_x = (screen_width—(base_width * scale_x)) / 2;
```

https://doi.org/10.15215/remix/9781998944224.01

```
float offset_y = (screen_height–(base_height * scale_y))
/ 2;

al_translate_transform(&transform, offset_x, offset_y);
al_use_transform(&transform);
```

Here, the code ensures that even if the screen is wider or taller than your base resolution, your game remains centered.

3.7.3. Using Logical Coordinates

To abstract your code if you don't have a specific resolution in mind, sometimes the best approach, instead of working directly with pixels, is to define a virtual resolution (e.g., 800 × 600) and scale everything accordingly. By utilizing this process, you let the game engine/software scale the game screen dynamically to fit the current display screen. The advantages of this method include maintaining the aspect ratio of the screen, better performance optimization and smoother-looking graphics when utilizing antialiasing, and simply keeping game items like the user interface (UI) consistent proportionally to avoid distortions or difficulties reading text.

EXAMPLE

```
#define VIRTUAL_WIDTH 800
#define VIRTUAL_HEIGHT 600

float sx = screen_width / VIRTUAL_WIDTH;
float sy = screen_height / VIRTUAL_HEIGHT;

al_identity_transform(&transform);
al_scale_transform(&transform, sx, sy);
al_use_transform(&transform);
```

So here, all drawing routines use a consistent coordinate system based on VIRTUAL_WIDTH and VIRTUAL_HEIGHT. If you always draw and place objects using your base resolution (e.g., 800 × 600), then no matter what resolution the player uses, everything will be resized properly.

If you look at table 8, you'll see the benefits of using transformations vs. needing to hand tweak your code for multiple displays (not to mention the fact that a programmer might miss incorporating the multitude of resolutions out there these days).

https://doi.org/10.15215/remix/9781998944224.01

Table 8: Transformations vs. Manual Scaling

Approach	Pros	Cons
Manual Scaling (Resize assets, reposition elements manually)	Fine-tuned control over elements	Requires hard-coded adjustments for each resolution
Transformations (al_scale_transform)	Automatic scaling across all resolutions	Can lead to minor distortions if aspect ratio is not handled
Transformations + Letterboxing (Black bars to maintain aspect ratio)	Consistent aspect ratio	Some screen space is unused

Aspect Ratio Considerations

While this might not seem like a concern foremost in a game developer's mind, when scaling, aspect ratio mismatches can cause stretching. As a result, your nicely circular objects can become oblong, or your tall items suddenly become short and squat.

If you want to maintain the correct aspect ratio, you can apply a transform like so:

- Scale using the same factor for both axes (min(scale_x, scale_y)).
- Add black bars (letterboxing) to fill the remaining space:

```
float scale_factor = fmin(scale_x, scale_y); // Maintain
aspect ratio
float new_width = VIRTUAL_WIDTH * scale_factor;
float new_height = VIRTUAL_HEIGHT * scale_factor;

float offset_x = (screen_width–new_width) / 2;
float offset_y = (screen_height–new_height) / 2;

al_identity_transform(&transform);
al_translate_transform(&transform, offset_x, offset_y);
al_scale_transform(&transform, scale_factor,
scale_factor);
al_use_transform(&transform);
```

https://doi.org/10.15215/remix/9781998944224.01

3.7.4. Summing Up How Transformations Work in Allegro 5

Understanding transformations in Allegro 5 is all about working with an ALLEGRO_TRANSFORM object to change how graphics are drawn. The process begins with creating and initializing this object using al_identity_transform, setting it up with the identity matrix. From there, you can move objects around by translating them with al_translate_transform or resize them while keeping their aspect ratios intact using al_scale_transform. If you need to rotate objects, you can use al_rotate_transform to set a specific angle.

One of the cool things about Allegro 5 is that you can combine multiple transformations in sequence. Once you have your transformations set up just the way you want them, you make it the current transformation for drawing operations with al_use_transform.

A couple of examples might help to clarify things. You might use scaling and centering to maintain the aspect ratio or set up a basic transformation by creating and applying it in a straightforward manner. Overall, transformations are a key part of effectively positioning, resizing, and rotating graphics within Allegro 5.

3.7.5. Creating and Setting a Transformation

Before applying a transformation, you first create an ALLEGRO_TRANSFORM object and then apply various transformation operations to it. After setting up the transformation, you make it the current transformation for drawing operations.

Here's an example of how to create and set a basic transformation:

```
ALLEGRO_TRANSFORM transform;
al_identity_transform(&transform); // Initialize with the
identity matrix
al_rotate_transform(&transform, angle); // Rotate
al_translate_transform(&transform, x, y); // Move
al_scale_transform(&transform, scaleX, scaleY); // Scale
al_use_transform(&transform); // Set as the current
transformation
```

Transformations can be combined to produce more complex effects. For example, you might want to rotate an object around a point other than its center, scale it, and then move it. The order of transformations is crucial as it affects the final result.

https://doi.org/10.15215/remix/9781998944224.01

3.7.6. Order of Transformations

The typical order of transformations for achieving a common effect (like rotating an object around its center and then moving it) is:

1. Translate the object to the origin (0,0).
2. Perform the rotation.
3. Scale if necessary.
4. Translate the object to its intended position.

When we use a transformation matrix to modify drawing operations, we typically follow these steps.

Reset the Transformation Matrix

When working with transformations in Allegro 5, each transformation matrix builds on the previous one. If you don't reset the transformation matrix before applying new transformations, your transformations could accumulate in unexpected ways.

By calling al_identity_transform(), you ensure that you start from a clean slate before applying any new transformations.

Imagine a case where you don't use al_identity_transform() before applying transformations:

```
ALLEGRO_TRANSFORM transform;
al_rotate_transform(&transform, ALLEGRO_PI / 4);
al_use_transform(&transform);
```

If transform was already modified elsewhere in the code, this would continue rotating based on the existing transformations, leading to an unexpected effect.

Now, using al_identity_transform() ensures we reset it first:

```
ALLEGRO_TRANSFORM transform;
al_identity_transform(&transform); // Reset to a clean
state
al_rotate_transform(&transform, ALLEGRO_PI / 4);
al_use_transform(&transform);
```

This ensures that the only transformation applied is the new 45-degree rotation.

The routine al_identity_transform(ALLEGRO_TRANSFORM *trans) is used to reset a transformation matrix to the identity matrix. This means it sets

https://doi.org/10.15215/remix/9781998944224.01

up a transformation that does nothing—it doesn't rotate, scale, or translate anything.

Apply transformations (translate, rotate, scale, etc.) with:

```
al_translate_transform(&transform,–100,–100); (Move
origin)
al_rotate_transform(&transform, ALLEGRO_PI / 4); (Rotate
45°)
al_translate_transform(&transform, 100, 100); (Move back)
```

Use the transformation:

```
    al_use_transform(&transform);
```

This applies the transformations to all subsequent drawing operations.

3.8. Antialiasing and Texture Smoothing

3.8.1. Explanation

Here, we'll take a brief look at some Allegro routines that can be applied to improve the look of your graphics and to utilize the graphics processing unit (GPU) on your graphics cards for increased performance.

Antialiasing is a technique used to reduce the visual defects (like jagged edges) that occur when high-resolution images are displayed in a lower resolution. Texture smoothing refers to the process of blending the colours of an image's pixels to create a smoother transition between them.

3.8.2. Allegro 5 Implementation

In Allegro 5, you enable antialiasing and texture smoothing through the use of bitmap flags and settings in the graphics library.

```
al_set_new_bitmap_flags(ALLEGRO_MIN_LINEAR |
ALLEGRO_MAG_LINEAR);
```

These flags tell Allegro to use linear filtering for minification and magnification, which results in smoother textures.

In Allegro 5, both antialiasing and texture smoothing are important for creating more visually appealing and polished graphics in your applications.

https://doi.org/10.15215/remix/9781998944224.01

3.8.3. Antialiasing

Antialiasing is used to reduce the jagged edges that can appear on diagonal or curved lines in digital images, a phenomenon often referred to as "aliasing." This effect is especially noticeable in lower-resolution graphics or when scaling images up or down.

In Allegro 5, antialiasing can be applied in several ways, but one of the most common methods is through the use of multisampling. Multisampling works by taking multiple samples at different points within each pixel and then averaging these samples to determine the pixel's colour. This process smooths out the edges of lines and curves.

Enabling Multisample Antialiasing in Allegro 5

To enable multisample antialiasing in Allegro 5, you need to set the appropriate display option before creating your display. Here's an example:

```
al_set_new_display_option(ALLEGRO_SAMPLE_BUFFERS, 1,
ALLEGRO_SUGGEST);
al_set_new_display_option(ALLEGRO_SAMPLES, 4, ALLEGRO_
SUGGEST); // Set to 4 samples
ALLEGRO_DISPLAY* display = al_create_display(width, height);
```

In this example, ALLEGRO_SAMPLE_BUFFERS is set to 1 to enable multisampling, and ALLEGRO_SAMPLES is set to 4 to use four samples per pixel.

3.8.4. Texture Smoothing

Texture smoothing, also known as filtering, is used to improve the appearance of textures when they are scaled or transformed. Without texture smoothing, textures can appear pixelated or blocky, especially when magnified.

In Allegro 5, texture smoothing is typically achieved through linear filtering. Linear filtering works by interpolating the colours of neighboring pixels to calculate the colour of a pixel when a texture is scaled. This results in a smoother transition between pixels.

Enabling Texture Smoothing in Allegro 5

To enable texture smoothing in Allegro 5, you set bitmap flags before loading or creating your bitmaps. Here's an example:

```
al_set_new_bitmap_flags(ALLEGRO_MIN_LINEAR |
ALLEGRO_MAG_LINEAR);
ALLEGRO_BITMAP* bitmap = al_load_bitmap("image.png");
```

https://doi.org/10.15215/remix/9781998944224.01

In this code, ALLEGRO_MIN_LINEAR and ALLEGRO_MAG_LINEAR are set to enable linear filtering for both minification (making the texture smaller) and magnification (making the texture larger), respectively.

3.9. Shaders

Shaders are programs that run on a computer's GPU to control the rendering of graphics. They are used for a variety of effects in real-time applications, such as video games.

3.9.1. Allegro 5 Implementation

Allegro 5 supports GLSL (OpenGL Shading Language) shaders. You can load and use shaders to affect rendering in your application:

```
ALLEGRO_SHADER *shader =
al_create_shader(ALLEGRO_SHADER_GLSL);
al_attach_shader_source(shader, ALLEGRO_VERTEX_SHADER,
vertex_shader_source);
al_attach_shader_source(shader, ALLEGRO_PIXEL_SHADER,
pixel_shader_source);
al_build_shader(shader);
al_use_shader(shader);
```

3.9.2. Shaders and Their Use in Allegro 5

What Are Shaders?

Shaders are small programs that run on a computer's graphics processing unit (GPU) and are used to control how graphics are rendered. They offer a high degree of flexibility and power, allowing developers to create a variety of visual effects and control the rendering pipeline in fine detail.

In the context of 2D and 3D graphics, there are mainly two types of shaders:

1. **Vertex Shaders**
   These process each vertex of a shape. They can manipulate attributes like position, colour, and texture coordinate of each vertex.
2. **Fragment Shaders (or Pixel Shaders)**
   These handle how pixels (fragments) are coloured. They're used for effects like lighting, colour blending, and texture mapping.

https://doi.org/10.15215/remix/9781998944224.01

Use of Shaders in Allegro 5

Allegro 5 supports the use of shaders through its integration with OpenGL (and DirectX on Windows). This integration allows you to write custom shaders in GLSL (OpenGL Shading Language) or HLSL (High-Level Shading Language for DirectX) and apply them to your graphics in Allegro.

Shaders in Allegro 5 can be used for a variety of tasks such as:

Creating Special Effects: Like simulating water, fire, shadows, or other complex visual phenomena

Image Processing: Like colour transformations, blurring, or edge detection

Performance Optimization: By offloading complex calculations to the GPU

Custom Rendering Pipelines: Tailoring the rendering process to specific needs of your application

Basic Steps to Use Shaders in Allegro 5

1. **Write the Shader Code**
   You write your vertex and fragment shaders. These can be written as separate files or as strings within your C++ code.
2. **Load and Compile the Shader**
   In your application, load the shader source code, compile it, and link it to create a shader program.
3. **Use the Shader**
   Bind the shader program before drawing operations to apply its effects.
4. **Passing Data to Shaders**
   You can pass data (like variables or textures) to the shaders through "uniforms" and "attributes."

EXAMPLE

Here's an outline of the steps of using a shader in Allegro 5:

```
// Create a shader
ALLEGRO_SHADER *shader =
al_create_shader(ALLEGRO_SHADER_GLSL);

// Load shader source code
al_attach_shader_source(shader, ALLEGRO_VERTEX_SHADER,
vertex_shader_source);
```

https://doi.org/10.15215/remix/9781998944224.01

```
al_attach_shader_source(shader, ALLEGRO_PIXEL_SHADER,
fragment_shader_source);

// Compile and link the shader
if (!al_build_shader(shader)) {
    // Handle error
}

// Use the shader
al_use_shader(shader);

// Your rendering code here

// Reset to default shader
al_use_shader(NULL);

// Clean up
al_destroy_shader(shader);
```

3.10. Using Premultiplied Alpha in Allegro 5

Premultiplied alpha is a technique used in computer graphics to handle transparency more effectively. In traditional alpha blending, an image has RGB (red, green, blue) channels and an alpha channel, which defines the transparency of each pixel. In premultiplied alpha, the RGB values are already multiplied by the alpha value before blending occurs. This approach simplifies the blending equations and helps resolve common issues associated with regular alpha blending, such as handling semitransparent edges and overlapping transparent objects.

3.10.1. Advantages of Premultiplied Alpha

When working with transparency and blending in graphics programming, especially in game development using Allegro 5, the choice between straight alpha and premultiplied alpha can significantly affect both visual quality and performance. Premultiplied alpha offers several practical advantages:

Simpler Blending Equations: With RGB values already scaled by alpha, blending calculations become more straightforward, often leading to better performance and fewer rendering artifacts.

https://doi.org/10.15215/remix/9781998944224.01

Better Handling of Semitransparent Pixels: Premultiplied alpha reduces visual artifacts around the edges of semitransparent textures, such as halos or dark fringes.

Improved Overlapping Transparency: It ensures more predictable blending behavior when multiple transparent objects overlap, avoiding darkening or incorrect blending effects.

3.10.2. Using Premultiplied Alpha in Allegro 5

To use premultiplied alpha in Allegro 5, follow these steps:

1. **Prepare Images with Premultiplied Alpha**
   Ensure that the RGB values in your image are premultiplied by the alpha channel. This can be done in image editing software or programmatically.
2. **Set the Correct Blender**
   Use the following Allegro 5 function to set the blending mode:

```
   al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA,
ALLEGRO_INVERSE_ALPHA);
```

   ALLEGRO_ADD: Adds source and destination colours.
   ALLEGRO_ALPHA: Uses the source's alpha channel.
   ALLEGRO_INVERSE_ALPHA: Uses the inverse of the source's alpha.
3. **Draw Your Images**
   After setting the blender, draw your images as usual. Allegro will apply premultiplied alpha blending to all subsequent drawing operations.

EXAMPLE

```
// Initialize Allegro and create a display...

// Set the blender for premultiplied alpha
al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA,
ALLEGRO_INVERSE_ALPHA);

// Load a bitmap with premultiplied alpha
ALLEGRO_BITMAP* bitmap = al_load_bitmap("premultiplied_
alpha_image.png");

// Draw the bitmap
al_draw_bitmap(bitmap, x, y, 0);
```

https://doi.org/10.15215/remix/9781998944224.01

```
// Reset blender if necessary
al_set_blender(ALLEGRO_ADD, ALLEGRO_ONE,
ALLEGRO_INVERSE_ALPHA);

// Continue with other drawing operations...
```

This example assumes the bitmap has premultiplied alpha. The al_set_blender routine configures the correct blending mode. More details on Allegro 5 blending modes will be covered in the next chapter.

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

1. **Basic Bitmap Display**
   Write a program that loads and displays a bitmap centered on a 1024 × 768 screen. Handle errors if the image fails to load.
2. **Dynamic Resolution Handling**
   Modify the basic bitmap display to dynamically adjust to different screen resolutions (e.g., 800 × 600, 1920 × 1080).
3. **Rotating Sprite**
   Create a sprite that rotates 45 degrees clockwise each time the space bar is pressed.
4. **Minimap System**
   Implement a minimap that shows a scaled-down version of a larger game world. Allow arrow keys to pan the viewport.
5. **Bitmap vs. PNG Comparison**
   Write a report comparing BMP and PNG formats in terms of file size, transparency support, and compression.
6. **Resolution Independence with Transformations**
   Use transformations to ensure a game scene scales proportionally across resolutions (e.g., 640 × 480 to 1920 × 1080).
7. **Tinting Effects**
   Create a program that tints a bitmap red when the R key is pressed and resets to normal when released.
8. **Panning Large Bitmap**
   Develop a system to pan across a large bitmap (e.g., 4096 × 4096) using arrow keys, displaying only a 1024 × 768 section.

https://doi.org/10.15215/remix/9781998944224.01

9. **Colour Depth Experiment**
   Convert a 24-bit image to 8-bit using an external tool and discuss visual differences in a short report.
10. **Border Drawing with VRAM**
    Programmatically draw a coloured border around a bitmap by directly manipulating VRAM values.
11. **Shader Colour Inversion**
    Implement a GLSL shader to invert the colours of a loaded bitmap.
12. **Rotating Background**
    Use Allegro transformations to create a continuously rotating background image.
13. **Frame Animation**
    Load multiple bitmap frames and cycle through them to simulate animation (e.g., a spinning coin).
14. **Antialiasing Demonstration**
    Compare screenshots of a diagonal line rendered with and without antialiasing. Explain the differences.
15. **Texture Smoothing**
    Load a low-resolution bitmap, scale it up, and apply linear filtering to observe the smoothing effects.
16. **Combined Transformations**
    Rotate and scale a bitmap simultaneously using al_draw_scaled_rotated_bitmap.
17. **Fade-Out Effect**
    Simulate a fade-out by gradually reducing the alpha value of a tinted bitmap over time.
18. **Viewport System**
    Create a viewport that displays only a 512 × 384 section of a larger game world.
19. **Palette Impact Analysis**
    Load the same image with different colour palettes (8-bit vs. 24-bit) and document visual changes.
20. **Aspect Ratio Maintenance**
    Allow window resizing while maintaining the game's aspect ratio using letterboxing.
21. **Particle System**
    Simulate a particle effect (e.g., fire) using scaled and rotated bitmaps.
22. **Premultiplied Alpha Compositing**
    Load two transparent PNGs and composite them correctly using premultiplied alpha blending.

https://doi.org/10.15215/remix/9781998944224.01

23. **Tile-Based Level Editor**
    Build a tool that lets users place bitmap tiles on a grid and export the layout as a file.
24. **Transformation Performance Test**
    Compare the performance of rendering 100 sprites with individual rotations vs. batch transformations.
25. **Intellectual Property Report**
    Research and summarize legal considerations when using third-party graphics (e.g., AI-generated art).
26. **Dynamic Health Bar**
    Create a health bar that changes colour (green to red) based on a player's health value.
27. **Day-Night Cycle**
    Simulate a day-night cycle by dynamically tinting the screen with varying alpha and RGB values.
28. **Screenshot Utility**
    Add a feature to capture the screen and save it as a BMP file when the S key is pressed.
29. **Parallax Scrolling**
    Implement a parallax background with three layers scrolling at different speeds.
30. **Greyscale Shader**
    Write a fragment shader to convert a bitmap to greyscale and apply it in real time.

https://doi.org/10.15215/remix/9781998944224.01

4

Programming Sprites in Allegro 5

4.1. The Fundamentals

4.1.1. Using Static Sprites in Allegro 5

In Allegro 5, a sprite is simply another bitmap object that can be displayed on the screen. You may notice that this sounds similar to the description of bitmap given in the previous chapter which raises an important question: What distinguishes sprites from ordinary bitmaps? A sprite bitmap (which we will just refer to as a sprite) is typically a moving object and not considered part of the background of a game. So we have now two layers, if you will, of images being utilized: a bitmap for a game background and then the moving of stationary objects on top of that—sprites. Sprites tend then to be small graphic images, with sizes like 32 × 32 and 64 × 64, or 128 × 128 and 256 × 256, although they could be of any size. However, when it comes to the animation aspect of the sprite, these smaller sizes are of benefit (since an animate sprite is a matrix of single bitmaps amalgamated in a larger object—more on that later).

Sprites are commonly utilized for the player character, nonplayer characters (NPCs), in-game objects, and weapons fire (projectiles, beams, etc.) The sprite is then a self-contained object that will have characteristics allowing it to be unitized through various means: movement around the screen (or the denial of movement), detection of collisions with objects, and so on.

As we look at an example of a sprite program, here we see some of the logic in the simplistic style of Allegro, providing the ability to use the same routines, building upon them to do more complex actions, thus easing the programmer's work by using familiar processes. As such, you will see commonalities in the routines and setup used for bitmaps and what we classify as sprites, allowing us to use similar library routines but for differing purposes.

https://doi.org/10.15215/remix/9781998944224.01

In the example program, we load two bitmaps: one to use as the background and the other as the movable sprite. We'll use our CatCosmos.png as the background and a simple flying saucer sprite, SmallSaucer.png (see figure 4.1)

Now one of the issues we have to deal with for sprites is, What do we display and what do we hide? If we use the saucer image as is, we will see the full image: the red square and the saucer. Oh, but it's way too big for the size of our screen, as you can see in figure 4.2.

Figure 4.1: Saucer sprite image. Illustrated by Walter Ridgewell.

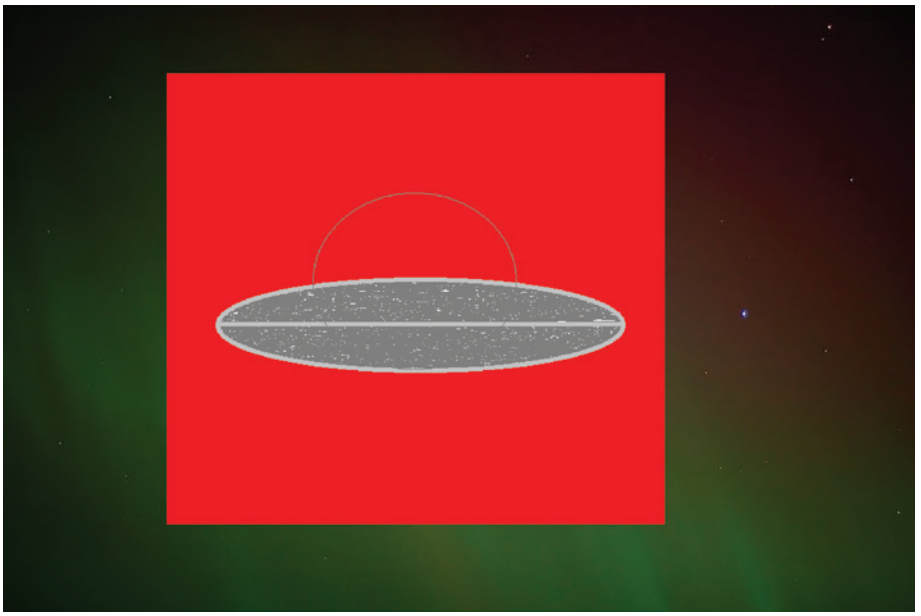

Figure 4.2: Saucer image displayed on a background image. Image by Walter Ridgewell.

https://doi.org/10.15215/remix/9781998944224.01

So we need to resize the saucer (we can do that in a graphics package like GIMP) and hide the red areas.

4.1.2. Alpha Blending and Its Purpose

In computer graphics, we use a function called alpha blending to make areas transparent or opaque. In addition to RGB (red, green, blue), some graphics come with a fourth value associated with their colours, giving them RGBA, where the A value is an alpha, a bit value used to indicate transparency. If the bitmap you decide to use doesn't have an alpha channel, Allegro has a routine to specify what colour the developer wishes to define as that alpha value, which will display as transparent or opaque. The process is to define the specified colour to establish a "mask" (the area that will be transparent, letting a background show through), defining it as an "alpha channel."

The routine to do this is:

```
void al_convert_mask_to_alpha(ALLEGRO_BITMAP *bitmap,
ALLEGRO_COLOR mask_color)
```

Colours in Allegro 5 are typically represented using the ALLEGRO_COLOR structure, which includes red, green, blue, and alpha (transparency) components. For the routine, one can define a transparent colour in advance:

```
ALLEGRO_COLOR transparentColor = al_map_rgba(255, 0, 0,
128); // Red color with 50% transparency
```

Here, you can use the ALLEGRO_COLOR al_map_rgb(unsigned char r, unsigned char g, unsigned char b) routine, which converts r, g, b (ranging from 0 to 255) into an ALLEGRO_COLOR, using 255 for alpha, and then call it like so:

```
al_convert_mask_to_alpha(sprite, transparentColor); //
Red color with 50% transparency
```

Alternately, one can embed the al_map_rgb routine right in the other routine:

```
al_convert_mask_to_alpha(sprite, al_map_rgb(255, 0, 0));
// Red (255, 0, 0) becomes fully transparent
```

We then need to call the blending routine in our code to enable the use of the alpha channel and have transparency:

https://doi.org/10.15215/remix/9781998944224.01

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA,
ALLEGRO_INVERSE_ALPHA);
```

Then Allegro will handle the rest when we draw the sprite.

Transparency is a powerful tool for creating effects like ghostly figures, smoke, shadows, or UI elements that need to overlay the game without completely obscuring the background. Translucent sprites are also useful for visual cues, such as indicating that a character is in stealth mode or partially invisible, or for effects like energy shields and force fields.

Our sprite's red background, though, isn't pure red. This becomes clear when one tries to run a test program using the bitmap and setting the transparency value of 255 for al_map_rgb. If one does that, you will still see the red square background. Computers are precise when it comes to numerical values, and so we need to find out what "shade" of red this is. As such it is necessary here to use a graphics program (like the free GIMP graphics package) and then a tool like the colour sampler (usually a small dropper type of icon), which will tell you the RGB values for that point, for example.

From the information provided, we see the RGB values are 237, 28, 36 for the red square area. We then would change those values in the code like so:

```
   al_convert_mask_to_alpha(sprite, al_map_rgb(237, 28,
36))
```

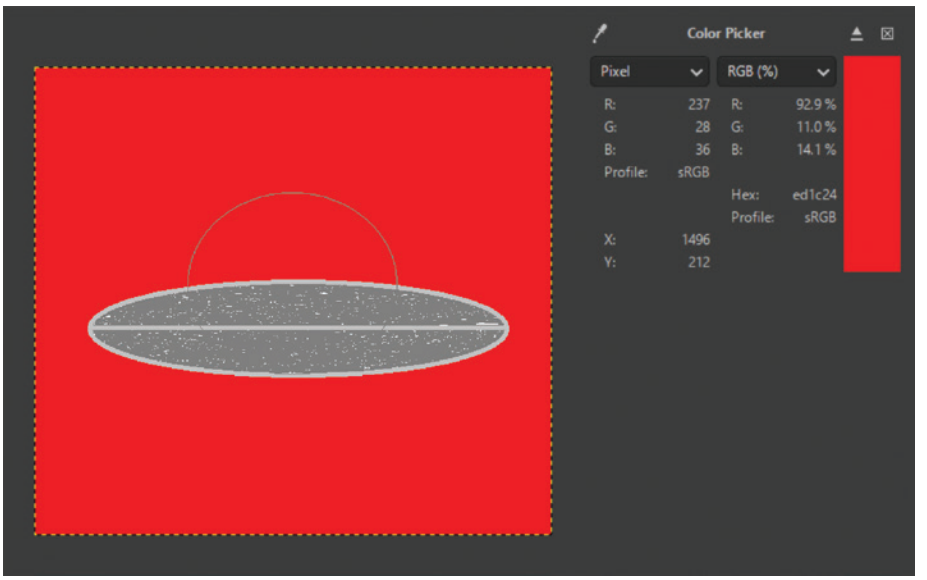

Figure 4.3: Saucer image open in a graphics editing program. Image by Walter Ridgewell.

https://doi.org/10.15215/remix/9781998944224.01

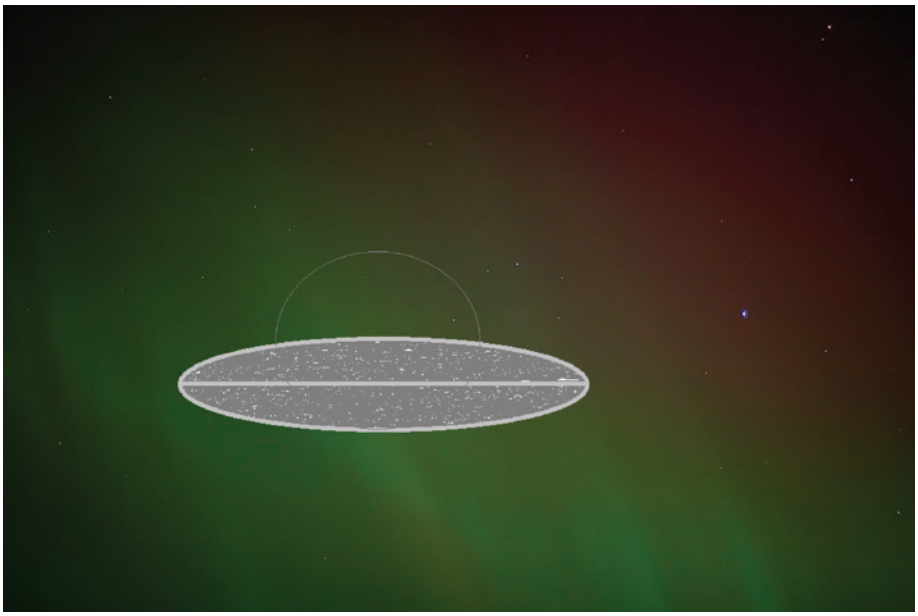

Figure 4.4: Saucer image with alpha transparency applied. Image by Walter Ridgewell.

This allows the red areas to become transparent, as seen in figure 4.4.

Let's look at the program code to do this. Here is a simple Allegro 5 program that displays a spaceship sprite and lets one move it across a background image using the keyboard keys.

4.1.3. Program Example—SpriteMove.c

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <stdio.h>

int main(int argc, char **argv) {
    ALLEGRO_DISPLAY *display = NULL;
    ALLEGRO_BITMAP *background = NULL;
    ALLEGRO_BITMAP *sprite = NULL;
    ALLEGRO_EVENT_QUEUE *event_queue = NULL;
    ALLEGRO_TIMER *timer = NULL;
    float sprite_x = 496; // Center of the screen
(1024/2–32/2)
    float sprite_y = 368; // Center of the screen
(768/2–32/2)
```

https://doi.org/10.15215/remix/9781998944224.01

```
    bool key[4] = { false, false, false, false }; //
Array to keep track of key states
    bool redraw = true;

    if (!al_init()) {
        fprintf(stderr, "Failed to initialize
Allegro!\n");
        return-1;
    }

    if (!al_install_keyboard()) {
        fprintf(stderr, "Failed to initialize the
keyboard!\n");
        return-1;
    }

    timer = al_create_timer(1.0 / 60.0);
    if (!timer) {
        fprintf(stderr, "Failed to create timer!\n");
        return-1;
    }

    display = al_create_display(1024, 768);
    if (!display) {
        fprintf(stderr, "Failed to create display!\n");
        al_destroy_timer(timer);
        return-1;
    }

    if (!al_init_image_addon()) {
        fprintf(stderr, "Failed to initialize image
addon!\n");
        al_destroy_display(display);
        al_destroy_timer(timer);
        return-1;
    }

    background = al_load_bitmap("CatCosmos.png");
    if (!background) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
        fprintf(stderr, "Failed to load background
image!\n");
        al_destroy_display(display);
        al_destroy_timer(timer);
        return-1;
    }

    sprite = al_load_bitmap("SmallSaucer.png");

    // Convert a specific color to alpha (e.g., pure red)
    al_convert_mask_to_alpha(sprite, al_map_rgb(237,
28, 36)); // Not quite Red (255, 0, 0) becomes fully
transparent

    if (!sprite) {
        fprintf(stderr, "Failed to load sprite
image!\n");
        al_destroy_bitmap(background);
        al_destroy_display(display);
        al_destroy_timer(timer);
        return-1;
    }

    event_queue = al_create_event_queue();
    if (!event_queue) {
        fprintf(stderr, "Failed to create
event_queue!\n");
        al_destroy_bitmap(sprite);
        al_destroy_bitmap(background);
        al_destroy_display(display);
        al_destroy_timer(timer);
        return-1;
    }

    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_timer_event_source(timer));
    al_register_event_source(event_queue,
al_get_keyboard_event_source());
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_start_timer(timer);

    while (1) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(event_queue, &ev);

        if (ev.type == ALLEGRO_EVENT_TIMER) {
            if (key[0] && sprite_y >= 4.0) {
                sprite_y-= 4.0;
            }

            if (key[1] && sprite_y <= 768-36.0) {
                sprite_y += 4.0;
            }

            if (key[2] && sprite_x >= 4.0) {
                sprite_x-= 4.0;
            }

            if (key[3] && sprite_x <= 1024-36.0) {
                sprite_x += 4.0;
            }

            redraw = true;
        } else if (ev.type == ALLEGRO_EVENT_DISPLAY_
CLOSE) {
            break;
        } else if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
            switch (ev.keyboard.keycode) {
                case ALLEGRO_KEY_UP:
                    key[0] = true;
                    break;
                case ALLEGRO_KEY_DOWN:
                    key[1] = true;
                    break;
                case ALLEGRO_KEY_LEFT:
                    key[2] = true;
                    break;
                case ALLEGRO_KEY_RIGHT:
                    key[3] = true;
```

https://doi.org/10.15215/remix/9781998944224.01

```
                    break;
            }
        } else if (ev.type == ALLEGRO_EVENT_KEY_UP) {
            switch (ev.keyboard.keycode) {
                case ALLEGRO_KEY_UP:
                    key[0] = false;
                    break;
                case ALLEGRO_KEY_DOWN:
                    key[1] = false;
                    break;
                case ALLEGRO_KEY_LEFT:
                    key[2] = false;
                    break;
                case ALLEGRO_KEY_RIGHT:
                    key[3] = false;
                    break;
            }
        }

        if (redraw && al_is_event_queue_empty(event_
queue)) {
            redraw = false;

            // Clear screen to white
            al_clear_to_color(al_map_rgb(255, 255, 255));

            al_draw_bitmap(background, 0, 0, 0);

            // Enable blending
            al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA,
ALLEGRO_INVERSE_ALPHA);

            al_draw_bitmap(sprite, sprite_x, sprite_y,
0);

            al_flip_display();
        }
    }

    al_destroy_bitmap(sprite);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_destroy_bitmap(background);
    al_destroy_timer(timer);
    al_destroy_display(display);
    al_destroy_event_queue(event_queue);

    return 0;
}
```

Now let's walk through the structure so we understand how it all fits together.

4.1.4. Program Overview

Step 1. Initialization

First, the program starts by including the necessary Allegro libraries for core functionality, images, and drawing primitives. These are the core components for our "game setup":

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <stdio.h>
```

Here, we also define variables like the display, sprite positions, and event handling objects (e.g., event_queue, timer).

Step 2. Preparing the Environment

Before proceeding, we need to ensure everything is set up correctly:

Initialize Allegro: We check if Allegro and its components (like the keyboard and image add-ons) are loaded successfully. If not, we quit early.

Create a Display: A window is created with dimensions 1024 × 768, which acts as the canvas for our game:

```
display = al_create_display(1024, 768);
```

Load Assets: The program loads the background and sprite images. If the assets don't load properly, we clean up and exit to avoid crashes:

https://doi.org/10.15215/remix/9781998944224.01

```
background = al_load_bitmap("CatCosmos.png");
sprite = al_load_bitmap("SmallSaucer.png");
```

Transparency Trick: By calling al_convert_mask_to_alpha(sprite, al_map_rgb(237, 28, 36)), it makes a specific colour in the sprite (near-red) transparent. This ensures the sprite blends well with the background.

Step 3. The Core Timer and Event Queue

The heart of our game logic is driven by the event queue and a timer. The timer runs at 60 frames per second (1.0 / 60.0), ensuring smooth updates:

```
timer = al_create_timer(1.0 / 60.0);
```

Events like key presses, timer ticks, and display closing are registered here. Think of the event queue as a to-do list the program processes continuously.

Step 4. The Game Loop

This is where the action happens! The program runs an infinite while loop to handle events. Here's the breakdown:

Check for Events: The al_wait_for_event() routine listens for events like timer updates, key presses, or the display closing.

Move the Sprite: If the timer fires (ALLEGRO_EVENT_TIMER), the sprite's position is adjusted based on which keys are pressed (key[0] to key[3] for up, down, left, right). The program ensures the sprite stays within screen boundaries using conditions like:

```
if (key[0] && sprite_y >= 4.0) {
    sprite_y-= 4.0;
}
```

Update Key States: When keys are pressed (ALLEGRO_EVENT_KEY_DOWN) or released (ALLEGRO_EVENT_KEY_UP), the corresponding flags in the key[] array are updated. This is like flipping switches for movement.

Redraw the Screen: If the redraw flag is set and no events are in the queue, the display is updated:

- The background is drawn first.
- The sprite is then drawn on top, with transparency applied.

https://doi.org/10.15215/remix/9781998944224.01

```
al_draw_bitmap(sprite, sprite_x, sprite_y, 0);
```

Step 5. Cleanup

After the user closes the window or exits, the program cleans up by destroying all created resources (e.g., the sprite, background, display, timer, and event queue).

Key Highlights

The game logic ties sprite movement directly to key states, ensuring smooth, responsive controls.

The al_set_blender() routine enables blending, making the sprite's transparency work correctly.

Safety checks are sprinkled throughout to handle failures gracefully (e.g., missing images).

In short, this program combines Allegro's features to create a basic game where a spaceship moves over a background. It's modular, with clear steps for initializing, handling events, updating positions, and rendering graphics. It's like building a foundation for a game you can expand on later—perhaps by adding collisions, sound, or even enemies!

4.2. Additional Sprite Handling Routines

We will now expand on the previous code to demonstrate the different sprite manipulation techniques and discuss why you might want to use these routines in a game. Additionally, we'll explain the importance of using al_get_bitmap_width and al_get_bitmap_height routines in some of the routines. We've seen how to draw a sprite; here in this section we'll explore additional methods we can use to manipulate those sprites—scaling, flipping, rotating, pivoting, and applying translucency.

We'll use the following base code snippet from our previous example program as our starting point:

```
if (redraw && al_is_event_queue_empty(event_queue)) {
            redraw = false;

            // Clear screen to white
            al_clear_to_color(al_map_rgb(255, 255, 255));
            al_draw_bitmap(background, 0, 0, 0);

            // Enable blending
```

https://doi.org/10.15215/remix/9781998944224.01

```
            al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA,
ALLEGRO_INVERSE_ALPHA);
            al_draw_bitmap(sprite, sprite_x, sprite_y, 0);
            al_flip_display();
        }
    }
```

To begin, we'll be looking at what changes to this code segment line can be implemented from our previous example, which draws a sprite at a specified position (sprite_x and sprite_y) on the screen via the following:

```
            al_draw_bitmap(sprite, sprite_x, sprite_y, 0);
```

4.2.1. Drawing Scaled Sprites

Scaling a sprite involves resizing it by specific factors along the x and y axes. Scaling is useful when you want to create different-sized enemies or objects without needing multiple versions of the same sprite. For instance, you might have a boss enemy that's a larger version of a regular enemy sprite. Scaling can also be used for visual effects, like zooming in on a specific game element or making an object appear closer or farther away. If your game was a target shooting game, you could start with a small-scale sprite making the target appear farther away and then gradually increase the scale, creating the appearance of the object getting closer.

To scale a sprite in Allegro 5, you use the al_draw_scaled_bitmap routine. This routine allows you to stretch or shrink the sprite's width and height to fit your desired dimensions.

```
al_draw_scaled_bitmap
void al_draw_scaled_bitmap(ALLEGRO_BITMAP *bitmap,float
sx, float sy, float sw, float sh, float dx, float dy,
float dw, float dh, int flags)
```

This draws a scaled version of the given bitmap to the target bitmap.

sx: source x
sy: source y
sw: source width
sh: source height
dx: destination x
dy: destination y

https://doi.org/10.15215/remix/9781998944224.01

dw: destination width
dh: destination height
flags: same as for al_draw_bitmap

In the following code line, the sprite is scaled to twice its original size in both dimensions, as indicated by these parameters in the routine call referencing the destination width and height (dw and dh, respectively):

```
  (al_get_bitmap_width(sprite) * 2, al_get_bitmap_
height(sprite) * 2).
```

So in our previous example, we replace:

```
            al_draw_bitmap(sprite, sprite_x, sprite_y,
0);
```

with:

```
al_draw_scaled_bitmap(sprite,  0, 0, al_get_bitmap_
width(sprite), al_get_bitmap_height(sprite), sprite_x,
sprite_y,  al_get_bitmap_width(sprite) * 2, al_get_
bitmap_height(sprite) * 2,  0);
```

You may also notice that we are utilizing two other Allegro routines here, al_get_bitmap_width() and al_get_bitmap_height(), in place of simply *source width* and *source height*—our previous sh and sw values, respectively.

Why do we want to use al_get_bitmap_width and al_get_bitmap_height? In some of these bitmap manipulation routines, we need to specify information related to the dimensions of the sprite in use. The routines al_get_bitmap_width(sprite) and al_get_bitmap_height(sprite) retrieve the original width and height of the sprite, respectively. These values are crucial because they ensure that you are scaling the sprite based on its actual dimensions. Without these routines, you would need to manually track the sprite's size, which can lead to errors and inconsistencies, especially if the sprite's dimensions change during development or if you are working with multiple sprites of varying sizes. As such, we simplify the processes here by getting that information from the sprite being used at that moment.

So going back to our code snippet, it would look like this for a scaled bitmap:

```
if (redraw && al_is_event_queue_empty(event_queue)) {
            redraw = false;
```

https://doi.org/10.15215/remix/9781998944224.01

```
            // Clear screen to white
            al_clear_to_color(al_map_rgb(255, 255, 255));
            al_draw_bitmap(background, 0, 0, 0);

            // Enable blending
            al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA,
ALLEGRO_INVERSE_ALPHA);

al_draw_scaled_bitmap(sprite,  0, 0, al_get_bitmap_
width(sprite), al_get_bitmap_height(sprite), sprite_x,
sprite_y,  al_get_bitmap_width(sprite) * 2, al_get_
bitmap_height(sprite) * 2,  0);

          al_flip_display();
        }
    }
```

4.2.2. Drawing a Flipped Sprite

Flipping a sprite involves mirroring it across the x- or y-axis. Allegro 5 allows you to flip a sprite horizontally, vertically, or both by using the al_draw_bitmap routine with specific flags. Flipping is particularly useful for character sprites that need to face different directions. For example, a character sprite facing right can be flipped to face left without needing a separate sprite. This is efficient for handling movement in platformers or side-scrolling games. Similarly, flipping can be used to mirror environmental elements or objects (think a reflection in a mirror or water here), adding enhancements to the game world environment without extra resources.

Here, we are utilizing the flags available for the al_draw_bitmap routine, namely:

ALLEGRO_FLIP_HORIZONTAL: Flip the bitmap about the y-axis
ALLEGRO_FLIP_VERTICAL: Flip the bitmap about the x-axis

```
al_draw_bitmap(sprite, sprite_x, sprite_y,
ALLEGRO_FLIP_HORIZONTAL);
```

That example flips the sprite horizontally. You can also flip it vertically using ALLEGRO_FLIP_VERTICAL. You can also flip both horizontally and vertically in one function call by combining the flags (ALLEGRO_FLIP_HORIZONTAL | ALLEGRO_FLIP_VERTICAL).

https://doi.org/10.15215/remix/9781998944224.01

4.2.3. Drawing Rotated Sprites

To rotate a sprite, Allegro 5 provides the al_draw_rotated_bitmap routine. This routine allows you to specify an angle of rotation in radians and the center point around which the sprite will rotate. Rotation is essential for many gameplay mechanics, such as aiming weapons, rotating objects, or handling vehicles like cars or spaceships. For example, in a top-down shooter, the player's ship might rotate to aim in the direction of the mouse or joystick. Rotation can also be used for environmental elements like rotating platforms or spinning objects:

```
al_draw_rotated_bitmap

void al_draw_rotated_bitmap(ALLEGRO_BITMAP *bitmap,
   float cx, float cy, float dx, float dy, float angle,
int flags)
```

This draws a rotated version of the given bitmap to the target bitmap. The bitmap is rotated clockwise by the specified angle, measured in radians. The point at cx/cy relative to the upper left corner of the bitmap will be drawn at dx/dy, and the bitmap is rotated around this point. If cx,cy is 0,0, the bitmap will rotate around its upper left corner:

cx: center x (relative to the bitmap)
cy: center y (relative to the bitmap)
dx: destination x
dy: destination y
angle: angle by which to rotate (radians)
flags: same as for al_draw_bitmap

Once again, we look to substitute our code line:

```
            al_draw_bitmap(sprite, sprite_x, sprite_y, 0);
```

with the new function, and so we might use something like the following:

```
al_draw_rotated_bitmap(sprite,  al_get_bitmap_
width(sprite) / 2, al_get_bitmap_height(sprite) /
2,sprite_x, sprite_y, ALLEGRO_PI / 4, 0);
```

https://doi.org/10.15215/remix/9781998944224.01

In this example, the sprite drawn on-screen is now rotated 45 degrees (π / 4 radians) around its center.

Here once more we call some other Allegro routines to get required information, making our coding a bit easier. We are using the al_get_bitmap_width(sprite) / 2 and al_get_bitmap_height(sprite) / 2 routines to make sure we find the center of rotation. These are the center x and center y (cx and cy values). This ensures that the sprite rotates around its center point, which is typically the desired behavior for most game objects. Without these routines, you would need to manually calculate the center based on known dimensions, which is prone to errors and less flexible if the sprite size changes.

4.2.4. Drawing Pivoted Sprites

Pivoting a sprite involves rotating it around a specified pivot point other than its center. Here we still use the al_draw_rotated_bitmap routine. We noted in an earlier chapter the differences between "rotate" and "pivot" as a function of how we define the rotational axis point in our sprite. Pivoted rotation is useful when you want to rotate objects around a specific point, such as a door rotating around its hinge or a windmill's blades rotating around their base. It adds realism to animations and interactions, making game mechanics more intuitive and visually appealing:

```
al_draw_rotated_bitmap(sprite, 0, 0, // Pivot point at
the top-left corner of the sprite
                       sprite_x, sprite_y, ALLEGRO_PI /
4, 0);
```

Here, the sprite rotates 45 degrees around its top-left corner (0, 0).

In other cases, you might use the previously studied get width and height routines to help you identify different pivot points. Determining the dimensions of the sprite provides information that allows you to calculate relative positions within the sprite.

For example, you might want to pivot around the sprite's bottom-right corner by implementing (al_get_bitmap_width(sprite), al_get_bitmap_height(sprite)) in the previous code:

```
al_draw_rotated_bitmap(sprite, (al_get_bitmap_
width(sprite), al_get_bitmap_height(sprite)),
sprite_x, sprite_y,  ALLEGRO_PI / 4, 0);
```

https://doi.org/10.15215/remix/9781998944224.01

Any of the three previously discussed routines can be substituted in the code example in place of the original al_draw_bitmap(sprite, sprite_x, sprite_y, 0) routine:

```
if (redraw && al_is_event_queue_empty(event_queue)) {
            redraw = false;

            // Clear screen to white
            al_clear_to_color(al_map_rgb(255, 255, 255));
            al_draw_bitmap(background, 0, 0, 0);

            // Enable blending
            al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA,
ALLEGRO_INVERSE_ALPHA);
            al_draw_rotated_bitmap(sprite,(al_get_bitmap_
width(sprite),al_get_bitmap_height(sprite)), sprite_x,
sprite_y,  ALLEGRO_PI / 4, 0);
al_flip_display();
        }
    }
```

4.2.5. Translucent Sprites

We previously covered the concept of transparency in the discussion of the alpha channel and hiding sprite backgrounds with transparency; now let's look at if we want to modify the sprite itself.

Transparency enables effects such as ghosts, smoke, shadows, and UI overlays and is commonly used for visual cues like stealth, partial invisibility, energy shields, and force fields. To draw a translucent sprite, you can modify its alpha (transparency) value before drawing it. This can be done through the routine al_draw_tinted_bitmap:

```
void al_draw_tinted_bitmap(ALLEGRO_BITMAP *bitmap,
ALLEGRO_COLOR tint,
   float dx, float dy, int flags)
```

Here, the value for ALLEGRO_COLOR tint will be a call to the al_map_rgba_f routine, which will allow us to vary the intensity of the red, green, and blue values, as illustrated in the Allegro documents:

```
al_draw_tinted_bitmap(bitmap, al_map_rgba_f(0.5, 0.5,
0.5, 0.5), x, y, 0);
```

https://doi.org/10.15215/remix/9781998944224.01

This will draw the bitmap at 50% transparency (the RGB values are premultiplied with the alpha component).

This alpha represents a value where 1 is solid and 0 is transparent. Values between these two numbers then represent the opaqueness or transparency level, so .5 is the 50%.

The concept of a premultiplied alpha is the RGB colour values with the alpha value applied (multiplied by) so A * R, A * G, and A * B, where A would be the same value, so for a half transparent value of 50%, we would have .5 * R, .5 * G and .5 * B or just the 0.5 values seen in the code previously.

Once more using our code snippet as the example, we will replace the basic draw bitmap line with this new routine, resulting in a semitransparent bitmap image:

```
        if (redraw && al_is_event_queue_empty(event_
queue)) {
            redraw = false;
            // Clear screen to white
            al_clear_to_color(al_map_rgb(255, 255, 255));
            al_draw_bitmap(background, 0, 0, 0);

            // Enable blending
            al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA,
ALLEGRO_INVERSE_ALPHA);
            al_draw_tinted_bitmap(bitmap, al_map_
rgba_f(0.5, 0.5, 0.5, 0.5), sprite_x, sprite_y, 0);
            al_flip_display();
        }
}
```

4.3. Allegro Blending Function and Sprites

Another routine line we haven't talked about much but that is very important for our earlier example code is why we set the blending mode before utilizing other routines implementing an alpha level:

```
 // Enable blending

al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA,
ALLEGRO_INVERSE_ALPHA);
```

https://doi.org/10.15215/remix/9781998944224.01

```
al_draw_tinted_bitmap(sprite, al_map_rgba_f(0.5, 0.5,
0.5, 0.5), sprite_x, sprite_y, 0);
```

Here, the context of a blending function is the process of combining the new colours we are utilizing on the screen with preexisting colours. In some circumstances, we need to blend the colour of the bitmap object we are adding to an existing bitmap (so a sprite onto a background image) to have it look good or to affect some other graphic process.

Previously we took the red background on our saucer image, defined it as a mask area, and made that portion of the graphic transparent by assigning it to the alpha channel. We then called the blending routine and assigned it parameters. When the two images were combined, the saucer was visible on top of the background image and the transparency of the alpha channel portion let the background show through.

Perhaps the best explanation of the processes involved here is in the Allegro 5 documentation on al_set_blender:

```
void al_set_blender(int op, int src, int dst)
```

which sets the function to use for blending for the current thread.

Blending means the source and destination colours are combined in drawing operations.

Assume the source colour (e.g., colour of a rectangle to draw, or pixel of a bitmap to draw) is given as its red/green/blue/alpha components (if the bitmap has no alpha, it always is assumed to be fully opaque, so 255 for 8-bit or 1.0 for floating point): *s = s.r, s.g, s.b, s.a*. And this colour is drawn to a destination, which already has a colour: *d = d.r, d.g, d.b, d.a*.

The conceptual formula used by Allegro to draw any pixel then depends on the parameters:

```
ALLEGRO_ADD
   r = d.r * df.r + s.r * sf.r
   g = d.g * df.g + s.g * sf.g
   b = d.b * df.b + s.b * sf.b
         a = d.a * df.a + s.a * sf.a

ALLEGRO_DEST_MINUS_SRC
   r = d.r * df.r - s.r * sf.r
   g = d.g * df.g - s.g * sf.g
   b = d.b * df.b - s.b * sf.b
          a = d.a * df.a - s.a * sf.a
```

https://doi.org/10.15215/remix/9781998944224.01

```
ALLEGRO_SRC_MINUS_DEST
   r = s.r * sf.r - d.r * df.r
   g = s.g * sf.g - d.g * df.g
   b = s.b * sf.b - d.b * df.b
          a = s.a * sf.a–d.a * df.a
```

Valid values for the factors sf and df passed to this function are as follows, where s is the source colour, d the destination colour, and cc the colour set with al_set_blend_color (white by default):

```
ALLEGRO_ZERO
   f = 0, 0, 0, 0
ALLEGRO_ONE
   f = 1, 1, 1, 1
ALLEGRO_ALPHA
   f = s.a, s.a, s.a, s.a
ALLEGRO_INVERSE_ALPHA
   f = 1 - s.a, 1 - s.a, 1 - s.a, 1 - s.a
ALLEGRO_SRC_COLOR (since: 5.0.10, 5.1.0)
   f = s.r, s.g, s.b, s.a
ALLEGRO_DEST_COLOR (since: 5.0.10, 5.1.8)
   f = d.r, d.g, d.b, d.a
ALLEGRO_INVERSE_SRC_COLOR (since: 5.0.10, 5.1.0)
   f = 1 - s.r, 1 - s.g, 1 - s.b, 1 - s.a
ALLEGRO_INVERSE_DEST_COLOR (since: 5.0.10, 5.1.8)
   f = 1 - d.r, 1 - d.g, 1 - d.b, 1 - d.a
ALLEGRO_CONST_COLOR (since: 5.1.12, not supported on
OpenGLES 1.0)
   f = cc.r, cc.g, cc.b, cc.a
ALLEGRO_INVERSE_CONST_COLOR (since: 5.1.12, not supported
on OpenGLES 1.0)
   f = 1 - cc.r, 1 - cc.g, 1 - cc.b, 1 - ccc.a
```

BLENDING EXAMPLES

So for example, to restore the default of using premultiplied alpha blending, you would use:

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_ONE,
ALLEGRO_INVERSE_ALPHA);
```

As formula, the process is

https://doi.org/10.15215/remix/9781998944224.01

$$
\begin{aligned}
r &= d.r * (1 - s.a) + s.r * 1 \\
g &= d.g * (1 - s.a) + s.g * 1 \\
b &= d.b * (1 - s.a) + s.b * 1 \\
a &= d.a * (1 - s.a) + s.a * 1
\end{aligned}
$$

I'm not going to continue illustrating the formulaic processes for the following routines for the sake of brevity.

If you are using non-pre-multiplied alpha, you could use this:

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA,
ALLEGRO_INVERSE_ALPHA);
```

Additive blending would be achieved with the following:

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_ONE, ALLEGRO_ONE);
```

Copying the source to the destination (including alpha) unmodified:

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_ONE, ALLEGRO_ZERO);
```

Multiplying source and destination components:

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_DEST_COLOR,
ALLEGRO_ZERO)
```

Tinting the source (like al_draw_tinted_bitmap):

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_CONST_COLOR,
ALLEGRO_ONE);
al_set_blend_color(al_map_rgb(0, 96, 255)); /* nice
Chrysler blue */
```

Averaging source and destination pixels:

```
al_set_blender(ALLEGRO_ADD, ALLEGRO_CONST_COLOR,
ALLEGRO_CONST_COLOR);
al_set_blend_color(al_map_rgba_f(0.5, 0.5, 0.5, 0.5));
```

https://doi.org/10.15215/remix/9781998944224.01

4.3.1. Program Example—SpriteMoveEffects.c

In the following example, we've taken our previous demonstration program and added the following effects:

1. Scaled the saucer size down by 50%
2. Flipped the saucer image horizontally
3. Rotated it 45 degrees clockwise
4. Made the transparency a variable function that oscillates

This numbered list of effects will be referenced in the comments of the code that follows:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <stdio.h>

int main(int argc, char** argv) {
    ALLEGRO_DISPLAY* display = NULL;
    ALLEGRO_BITMAP* background = NULL;
    ALLEGRO_BITMAP* sprite = NULL;
    ALLEGRO_EVENT_QUEUE* event_queue = NULL;
    ALLEGRO_TIMER* timer = NULL;

    // Same sprite coordinates for top-left of the sprite
    float sprite_x = 496;
    float sprite_y = 368;

    bool key[4] = { false, false, false, false };
    bool redraw = true;

    // (4) Variables for controlling translucent "fade
in/out" of the sprite
    float alpha = 1.0f;           // current alpha
    bool alpha_increasing = false; // direction of alpha
change

    if (!al_init()) {
        fprintf(stderr, "Failed to initialize Allegro!\n");
```

https://doi.org/10.15215/remix/9781998944224.01

```
        return-1;
    }

    if (!al_install_keyboard()) {
        fprintf(stderr, "Failed to initialize the
keyboard!\n");
        return-1;
    }

    timer = al_create_timer(1.0 / 60.0);
    if (!timer) {
        fprintf(stderr, "Failed to create timer!\n");
        return-1;
    }

    display = al_create_display(1024, 768);
    if (!display) {
        fprintf(stderr, "Failed to create display!\n");
        al_destroy_timer(timer);
        return-1;
    }

    if (!al_init_image_addon()) {
        fprintf(stderr, "Failed to initialize image
addon!\n");
        al_destroy_display(display);
        al_destroy_timer(timer);
        return-1;
    }

    background = al_load_bitmap("CatCosmos.png");
    if (!background) {
        fprintf(stderr, "Failed to load background
image!\n");
        al_destroy_display(display);
        al_destroy_timer(timer);
        return-1;
    }

    sprite = al_load_bitmap("SmallSaucer.png");
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_convert_mask_to_alpha(sprite, al_map_rgb(237, 28,
36)); // Make a specific color transparent

    if (!sprite) {
        fprintf(stderr, "Failed to load sprite
image!\n");
        al_destroy_bitmap(background);
        al_destroy_display(display);
        al_destroy_timer(timer);
        return-1;
    }

    event_queue = al_create_event_queue();
    if (!event_queue) {
        fprintf(stderr, "Failed to create
event_queue!\n");
        al_destroy_bitmap(sprite);
        al_destroy_bitmap(background);
        al_destroy_display(display);
        al_destroy_timer(timer);
        return-1;
    }

    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_timer_event_source(timer));
    al_register_event_source(event_queue,
al_get_keyboard_event_source());

    al_start_timer(timer);

    // Get sprite dimensions for proper pivoting in
rotation
    float sprite_w = (float)al_get_bitmap_width(sprite);
    float sprite_h = (float)al_get_bitmap_height(sprite);

    while (1) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(event_queue, &ev);
```

https://doi.org/10.15215/remix/9781998944224.01

```
        if (ev.type == ALLEGRO_EVENT_TIMER) {
            // Movement logic remains the same
            if (key[0] && sprite_y >= 4.0) {
                sprite_y -= 4.0;
            }
            if (key[1] && sprite_y <= 768 - 36.0) {
                sprite_y += 4.0;
            }
            if (key[2] && sprite_x >= 4.0) {
                sprite_x -= 4.0;
            }
            if (key[3] && sprite_x <= 1024 - 36.0) {
                sprite_x += 4.0;
            }

            // (4) Update alpha for a continuous fade in/
out effect
            if (alpha_increasing) {
                alpha += 0.01f;
                if (alpha >= 1.0f) {
                    alpha = 1.0f;
                    alpha_increasing = false;
                }
            }
            else {
                alpha -= 0.01f;
                if (alpha <= 0.0f) {
                    alpha = 0.0f;
                    alpha_increasing = true;
                }
            }

            redraw = true;
        }
        else if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE)
{
            break;
        }
        else if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
            switch (ev.keyboard.keycode) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
            case ALLEGRO_KEY_UP:
                key[0] = true;
                break;
            case ALLEGRO_KEY_DOWN:
                key[1] = true;
                break;
            case ALLEGRO_KEY_LEFT:
                key[2] = true;
                break;
            case ALLEGRO_KEY_RIGHT:
                key[3] = true;
                break;
            }
        }
        else if (ev.type == ALLEGRO_EVENT_KEY_UP) {
            switch (ev.keyboard.keycode) {
            case ALLEGRO_KEY_UP:
                key[0] = false;
                break;
            case ALLEGRO_KEY_DOWN:
                key[1] = false;
                break;
            case ALLEGRO_KEY_LEFT:
                key[2] = false;
                break;
            case ALLEGRO_KEY_RIGHT:
                key[3] = false;
                break;
            }
        }

        if (redraw && al_is_event_queue_empty(event_
queue)) {
            redraw = false;

            // Clear screen to white
            al_clear_to_color(al_map_rgb(255, 255, 255));
            al_draw_bitmap(background, 0, 0, 0);

            // Enable blending
```

https://doi.org/10.15215/remix/9781998944224.01

```
            al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA,
ALLEGRO_INVERSE_ALPHA);

            // (1) Scale by 50%
            // (2) Flip horizontally
(ALLEGRO_FLIP_HORIZONTAL)
            // (3) Rotate 45 degrees clockwise
            // (4) Tint with variable alpha for
translucency
            al_draw_tinted_scaled_rotated_bitmap(
                sprite,
                al_map_rgba_f(1.0f, 1.0f, 1.0f,
alpha),         // Tint color with current alpha
                sprite_w /
2,                                  // pivot x (center of
sprite)
                sprite_h /
2,                                  // pivot y (center of
sprite)
                sprite_x + sprite_w /
2,                      // draw x so sprite center ends
up at (sprite_x, sprite_y)
                sprite_y + sprite_h /
2,                      // draw y
                0.5f,
0.5f,                                       // (1) scale by
50% 0.5f horizontally & vertically
                ALLEGRO_PI /
4,                                  // (3) 45 degrees in
radians (clockwise)
                ALLEGRO_FLIP_
HORIZONTAL                          // (2) flip horizontally
            );

            al_flip_display();
        }
    }

    al_destroy_bitmap(sprite);
    al_destroy_bitmap(background);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_destroy_timer(timer);
    al_destroy_display(display);
    al_destroy_event_queue(event_queue);

    return 0;
}
```

A screencap of our semitransparent 45 degree rotated and flipped saucer is shown in figure 4.5.

As you can see, Allegro 5 offers a good selection of functions to modify the colour of screen graphics using various blending functions. These routines can be utilized in a variety of ways to create a more dynamic screen presentation of objects (changing fire intensity, glowing particles, or sparkles), new stylized looks in a scene (changing the palette to pastel shades or using watercolour effects), or immediate visual feedback (a player character glowing after drinking a potion or an indication its been hit by a nonplayer character [NPC] attack).

4.4. Advanced Sprite Use in Allegro 5

4.4.1. Using Dynamic Sprites

In the last chapter, we focused on static sprites, which means we worked with a single, unchanging image for each sprite. While we adjusted things like size,

Figure 4.5: Saucer image as semitransparent, rotated, and flipped. Image by Walter Ridgewell.

https://doi.org/10.15215/remix/9781998944224.01

orientation, and position, the actual image—what the sprite looked like—stayed the same. We learned how to move the sprite around the screen or change how it appeared, but the sprite's visual state itself didn't evolve.

Now, when we talk about more advanced graphics, we introduce the idea of making the sprite's image change as well. This is where dynamic sprites come into play. Instead of using a single image, we can swap between multiple images to give the impression that the sprite is moving or performing an action. This is what we call animation.

Animation is all about cycling through different images, or frames, to create the illusion of movement. For example, if you have a character walking, instead of just sliding the same image across the screen, we show different frames where the character's legs are in different positions. By displaying these frames in quick succession, it looks like the character is actually walking.

We can combine this idea of changing the sprite image with the other transformations we already learned—like adjusting its size or position—to create a fully animated and interactive graphic. In the next steps, we'll explore how to manage these image sequences and control the speed at which they change. This will allow us to create smooth animations, like making a character run, jump, or even perform more complex actions.

So what does animating a sprite mean? It means swapping between a series of images (or frames) over time to simulate movement. And how do we achieve that? We'll break it down step-by-step, starting with loading multiple frames for a sprite and figuring out how to time their display to create a seamless animation.

4.4.2. Animated Sprites—Drawing an Animated Sprite

The process of how animation occurs in computer systems is the same as it is in video or film or in retrospect static sprite movement. We have a single frame of image information; we display it, then we make some change to that information and display it, repeating the process as needed. For our static sprites, we displayed the same image repeatedly, for animated sprites we want then to display a changed image or better yet multiple changed images.

We will then have multiple sprite images (variations of a base/initial image) that we will play in sequence, giving the static sprite a dynamic, changing appearance. There are many familiar animation scenes in video games that utilize this effect: the player's character walking, jumping, or running as opposed to standing still; character combat or weapons firing; and vehicles moving or items being destroyed (e.g., in explosions). All these functions can be accomplished with animated sprites.

The basic routine goes like this: You start with an initial sprite image or a frame, borrowing from video/film terminology. Our character begins standing,

https://doi.org/10.15215/remix/9781998944224.01

then in the next image, he lifts a leg somewhat. In the following image, he extends it forward, then he lowers the leg, and finally the character moves up to that location. So in four images, you have a character take a step forward. Since computers like doing things in power of 2, the typical sequence of frames with some intermediate images will be some multiple, such as 4, 8, or 16. We can refer to this collection of images as a "sprite sheet."

In the context of a classical 8-bit sprite being displayed, we could look at it as 4 different sprite positions displayed while moving the character around the screen, as in figure 4.6, for example.

Figure 4.6: Sprite sheet for character animation. Illustrated by Jana Ochse for 2DPIXX, published under a CC BY-SA 3.0 license and modified for size by Kyle Flemmer.

https://doi.org/10.15215/remix/9781998944224.01

Now, we have a few choices in terms of the coding to do this. We start with 8 images that need to be displayed in sequence, so an easy process here would be loading and storing our bitmaps into an array, then calling up the indexed image as required. This could get complicated, though, depending on how many bitmaps we have related to the animation complexity and the number of game elements. It would be better to be able to somehow minimize the number of individual images required while keeping them associated with a character or object but still allowing us the animation ability that the multiple modified bitmaps provide. So what if we can combine those 8 images into a single larger image, a sprite sheet. Then the task coding-wise is to select a portion of that single bitmap representing one of the images in the sequence of 8. If one does this, then we only load the sprite sheet image once (as opposed to loading 8 images), and we use the array function to designate a location within the sprite sheet, like the top left and bottom right coordinates of the individual image we want. By making our sizes convenient, like 32 × 32, the math for selection can be very easy to implement.

In the previous example, we illustrated a forward walking motion; however, typically video game characters move in multiple directions. In this respect, your sprite sheet will need to contain multiple rows of sprites, with each row being utilized for a specific direction. Your images, then, might be arranged as four rows of image sequences—up, left, right, and down. If we use 4 images as the minimal number of changes (although there is nothing stopping you from using only 3 or 2 if you really want a minimalist graphics set or for learning), then that gives us a matrix of 16 images arranged in a 4 × 4 layout. Again, if you keep the individual images' sizes convenient multiples, the math is logical in its implementation. Recall the array from table 4 in the previous chapter, shown again here as table 9, and assume that each sprite image or cell is 32 × 32.

The x- and y-coordinates for the first row of images in terms of top-left corner and bottom right would be [(0,0), (32,32)]; the second cell, then, is [(32,0), (64,32)]; the third cell [(64,0), (96,32)]; and finally [(96,0), (128,32)]. For the second

Table 9: Screen Coordinates

X →				
Y ↓	0,0	0,1	0,2	0,3
	1,0	1,1	1,2	1,3
	2,0	2,1	2,2	2,3
	3,0	3,1	3,2	3,3

https://doi.org/10.15215/remix/9781998944224.01

row, then, the first cell is [(0,32), (32,64)], the second cell [(32,32), (64,64)], and so on—easy multiples of 32 to work with (see table 10).

4.4.3. Grabbing Sprite Frames from an Image—the Sprite Sheet

Sprite sheets are a combination of animation frames in a single larger bitmap; we then can load that single bitmap image and ideally select animation frame/areas from that. The process then becomes one of loading a main bitmap, the sprite sheet selecting fixed areas of that (the animation frames), and copying those into the arrays or matrixes for use. Allegro provides us with a function to copy areas of a larger bitmap image al_draw_bitmap_region, and through some clever foreplanning in terms of sizes of the images we create, the math to specify which images to select can become quite easy. If, for example, we have sprite images that are 32 × 32 and we have 4 frames, we combine them to create a main character bitmap that is 128 bits high and 128 bits wide (4 × 32 = 128).

The Allegro 5 function al_draw_bitmap_region is used to draw a specific rectangular region of a bitmap (image) onto the screen. This function is particularly useful when working with sprite sheets, where a single image contains multiple subimages (frames) that represent different parts of an animation or different states of a character:

```
void al_draw_bitmap_region(ALLEGRO_BITMAP *bitmap, float
sx, float sy,float sw, float sh, float dx, float dy,int
flags);
```

Parameters

bitmap: The source image (sprite sheet) from which the region will be drawn

sx (source x): The x-coordinate (top-left corner) of the region in the source bitmap

sy (source y): The y-coordinate (top-left corner) of the region in the source bitmap

Table 10: Sprite Sheet Coordinates

(0,0), (32,32)	(32,0), (64,32)	(64,0), (96,32)	(96,0), (128,32)
(0,32), (32,64)	(32,32), (64,64)	(64,32), (96,64)	(96,32), (128,64)
(0,64), (32,96)	(32,64), (64,96)	(64,64), (96,96)	(96,64), (128,96)
(0,96), (32,128)	(32,96), (64,128)	(64,96), (96,128)	(96,96), (128,128)

https://doi.org/10.15215/remix/9781998944224.01

sw (source width): The width of the region to draw
sh (source height): The height of the region to draw
dx (destination x): The x-coordinate on the screen where the top-left corner of the region will be drawn
dy (destination y): The y-coordinate on the screen where the top-left corner of the region will be drawn
flags: Determines how the bitmap is drawn. Common flags include:
0: Normal drawing (no transformations)
ALLEGRO_FLIP_HORIZONTAL: Flips the image horizontally
ALLEGRO_FLIP_VERTICAL: Flips the image vertically

Let's go back to our example, the sprite sheet that contains 16 frames in a 4 × 4 grid, where each frame is 32 × 32 pixels. Let's say you want to draw the third frame in the first row, the one labelled 0,2 in the diagram (frame index 2 in zero-based counting):

```
// Parameters for al_draw_bitmap_region
ALLEGRO_BITMAP *sprite_sheet = al_load_bitmap("sprite_
sheet.png");

int frame = 2;          // Third frame (zero-based)
int row = 0;            // First row (zero-based)
int cell_size = 32;     // Size of each frame (width and
height)

float sx = frame * cell_size;   // Source X
float sy = row * cell_size;     // Source Y
float sw = cell_size;           // Source Width
float sh = cell_size;           // Source Height

float dx = 100;                 // Destination X
float dy = 100;                 // Destination Y

// Draw the selected frame from the sprite sheet
al_draw_bitmap_region(sprite_sheet, sx, sy, sw, sh, dx,
dy, 0);
```

Here, you see we're calculating the sx and sy values based on the frame index and cell size, allowing us to specify that image. In the case of an animation sequence, we can then increment or decrement values to achieve that

https://doi.org/10.15215/remix/9781998944224.01

effect. If we want to go through the animation sequence in the first row, we increment the frame value from 0 to 3; if we want to use a different sequence, we change the row value.

In the simple demonstration program to follow, we'll use 4 movement animations to go with 4 directions from the keyboard. Pressing the Up, Down, Left, and Right keys changes the character's direction and updates its position, so we are selecting different sprite images and moving the sprite as well. This means we need 4 animation sets, and it will then fit nicely with our 4 × 4 sprite sheet.

The 4 × 4 sprite sheet layout assumes four rows for movement directions:

Row 0: Down
Row 1: Up
Row 2: Left
Row 3: Right

For the movement aspect, we then cycle through four frames (columns) for each direction. For our sprite sheet, we'll use an asset from OpenGameArt.org. One of the things a game designer will need if they aren't going to create their own graphic and sound assets are open-source graphics and sounds. Many sites can be found that provide these under the auspices of Creative Commons licensing agreements. In many cases, all the artists request is proper acknowledgement of their work as per the applicable Creative Commons licenses, and as a video game designer, you should always seek to attribute and acknowledge the work of others that you have used.

For this example, I'm using the sprite sheet RPG Hero (Walking Cycle) illustrated by Jana Ochse for 2DPIXX and published under a CC BY-SA 3.0 license (see figure 4.6). Many thanks to Jana for providing artwork for those of us lacking the graphical skill to create such necessary and integral assets in our quest to create new games for others to enjoy.

4.4.4. Complete Coding Example

```
// OneSpriteMove.c
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <iostream>

// Constants
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;
```

https://doi.org/10.15215/remix/9781998944224.01

```
const int SPRITE_WIDTH = 32;  // Width of each sprite cell
const int SPRITE_HEIGHT = 64; // Height of each sprite cell
const int FPS = 30;
const int NUM_FRAMES = 4;

// Directions
enum Direction { UP = 1, LEFT = 3, RIGHT = 2, DOWN = 0 };
// Values match sprite sheet layout

int main() {
    // Initialize Allegro
    if (!al_init()) {
        std::cerr << "Failed to initialize Allegro." <<
std::endl;
        return-1;
    }

    if (!al_init_image_addon()) {
        std::cerr << "Failed to initialize Allegro Image
Addon." << std::endl;
        return-1;
    }

    if (!al_install_keyboard()) {
        std::cerr << "Failed to initialize keyboard
input." << std::endl;
        return-1;
    }

    // Create display
    ALLEGRO_DISPLAY* display = al_create_display(SCREEN_
WIDTH, SCREEN_HEIGHT);
    if (!display) {
        std::cerr << "Failed to create display." <<
std::endl;
        return-1;
    }

    // Set up timer and event queue
    ALLEGRO_TIMER* timer = al_create_timer(1.0 / FPS);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    ALLEGRO_EVENT_QUEUE* event_queue =
al_create_event_queue();

    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_timer_event_source(timer));
    al_register_event_source(event_queue,
al_get_keyboard_event_source());

    // Load sprite sheet
    ALLEGRO_BITMAP* sprite_sheet = al_load_bitmap("rpg-
character.png");
    // Convert a specific color to alpha (As noted one
can use graphics program like GIMP and the color selector
to find values)
    al_convert_mask_to_alpha(sprite_sheet, al_map_
rgb(192, 0, 128)); // Here the pinkish background has
values of  (R:192 G:0, B:128) classic magic pink values

    if (!sprite_sheet) {
        std::cerr << "Failed to load sprite sheet." <<
std::endl;
        al_destroy_display(display);
        return-1;
    }

    // Variables for animation
    int sprite_x = SCREEN_WIDTH / 2-SPRITE_WIDTH / 2;
    int sprite_y = SCREEN_HEIGHT / 2-SPRITE_HEIGHT / 2;
    int frame = 0;
    Direction current_direction = DOWN;
    bool redraw = true;
    bool running = true;

    al_start_timer(timer);

    while (running) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(event_queue, &ev);
```

https://doi.org/10.15215/remix/9781998944224.01

```
        if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            running = false;
        }

        if (ev.type == ALLEGRO_EVENT_TIMER) {
            redraw = true;

            // Update animation frame
            //frame = (frame + 1) % NUM_FRAMES;
        }

        if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {

            frame = (frame + 1) % NUM_FRAMES;

            switch (ev.keyboard.keycode) {
            case ALLEGRO_KEY_UP:
                current_direction = UP;
                sprite_y-= SPRITE_HEIGHT;
                break;
            case ALLEGRO_KEY_DOWN:
                current_direction = DOWN;
                sprite_y += SPRITE_HEIGHT;
                break;
            case ALLEGRO_KEY_LEFT:
                current_direction = LEFT;
                sprite_x-= SPRITE_WIDTH;
                break;
            case ALLEGRO_KEY_RIGHT:
                current_direction = RIGHT;
                sprite_x += SPRITE_WIDTH;
                break;
            case ALLEGRO_KEY_ESCAPE:
                running = false;
                break;
            }
        }

        if (redraw && al_is_event_queue_empty(event_
queue)) {
            redraw = false;
```

https://doi.org/10.15215/remix/9781998944224.01

```
            // Clear screen
            al_clear_to_color(al_map_rgb(0, 0, 0));

            // Calculate source rectangle for current
frame and direction
            int sx = frame * SPRITE_WIDTH;
            int sy = current_direction *
SPRITE_HEIGHT;

            // Draw sprite
            al_draw_bitmap_region(sprite_sheet, sx, sy,
SPRITE_WIDTH, SPRITE_HEIGHT, sprite_x, sprite_y, 0);

            // Flip display
            al_flip_display();
        }
    }

    // Cleanup
    al_destroy_bitmap(sprite_sheet);
    al_destroy_timer(timer);
    al_destroy_event_queue(event_queue);
    al_destroy_display(display);

    return 0;
}
```

Our result here is the character sprite on-screen changing orientation with respect to the direction it's moving in and a small animation sequence showing its steps (see figure 4.7). All the graphics required to do this are in the single loaded sprite sheet.

4.4.5. Animated Sprites in Allegro 5

Animated sprites in games have both pros and cons.

Pros

1. **Greater Control**
   Using sprites gives you more control over every aspect of the animation, including frame rate, sequence, and transitions. You can dynamically change animations in response to user input or game events.

https://doi.org/10.15215/remix/9781998944224.01

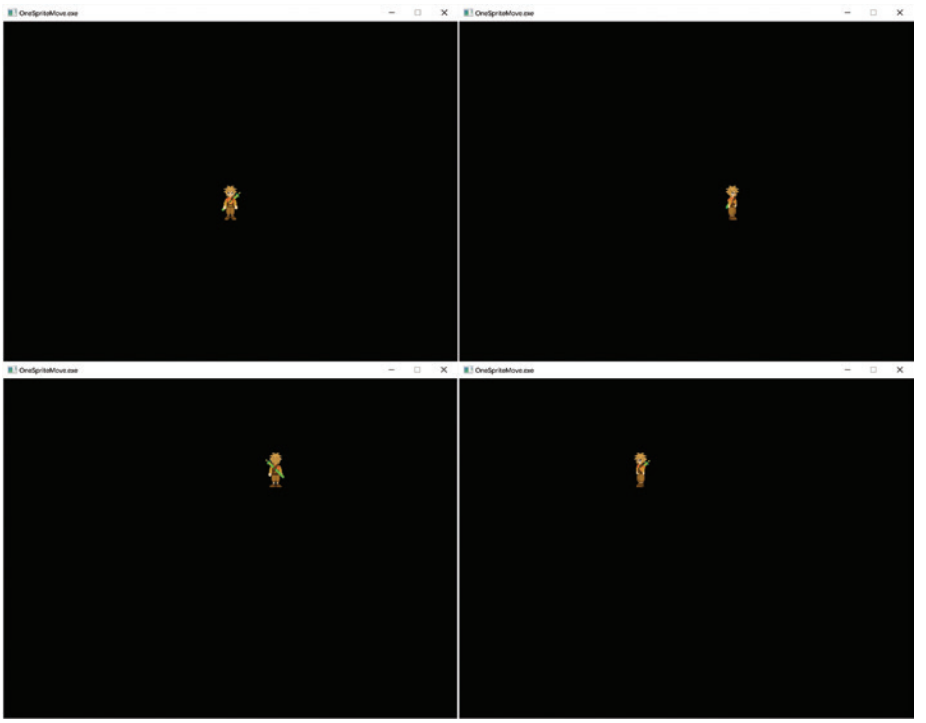

Figure 4.7: Changing character orientation on-screen. Image by Walter Ridgewell.

2. **Better Quality**
   Since you're not limited to the GIF format's colour palette, you can use full-colour images with higher quality.
3. **Performance Optimization**
   Sprites can be more efficient, especially when combined with sprite sheets. This reduces the memory footprint and can improve rendering performance.

Cons

1. **Complexity**
   Implementing sprite-based animations requires more coding effort. You need to handle the logic for frame switching, timing, and possibly interpolating between frames.
2. **Asset Management**
   You'll have to manage multiple image files or sprite sheets, which can increase the complexity of asset management.
3. **Development Time**
   More time is needed to develop and fine-tune sprite animations, especially if you require sophisticated or interactive animations.

https://doi.org/10.15215/remix/9781998944224.01

The choice between using animated GIFs and sprites in Allegro 5 largely depends on your project's specific needs. If simplicity and ease of use are your primary concerns, you are experienced with utilizing/implementing external libraries, and the limitations of GIFs are acceptable for your project, then using animated GIFs might be the way to go. However, if you need higher-quality animations, more control, and better performance in terms of using routines that already exist in Allegro 5, then implementing animations with sprites is usually the better choice.

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Basic Sprite Handling

1. **Theory:** Explain the purpose of al_convert_mask_to_alpha and provide an example of its usage.
2. **Code Modification:** Modify SpriteMove.c to allow diagonal movement using the WASD keys.
3. **Project:** Create a program where a sprite's transparency (alpha value) oscillates between 0% and 100% over time.

Sprite Transformations

4. **Theory:** Compare al_draw_scaled_bitmap and al_draw_rotated_bitmap. When would you use each?
5. **Code Modification:** Update SpriteMoveEffects.c to make the sprite rotate counterclockwise instead of clockwise.
6. **Project:** Implement a sprite that scales up when the mouse hovers over it and returns to normal when the mouse exits.
7. **Theory:** Explain how al_set_blender works and provide two blending mode examples not covered in the chapter.

Animation and Sprite Sheets

8. **Project:** Design a sprite sheet with four frames for a "jumping" animation. Write code to cycle through these frames when the space bar is pressed.
9. **Code Modification:** Modify OneSpriteMove.c to animate the character's walking cycle automatically while moving, rather than per key press.

https://doi.org/10.15215/remix/9781998944224.01

10. **Theory:** What are the advantages of using sprite sheets over individual frame files?

Advanced Sprite Effects

11. **Project:** Create a particle system where translucent sprites (e.g., sparks) emit from a central point and fade over time.
12. **Code Modification:** Add collision detection to SpriteMove.c to prevent the saucer from moving over a specific region of the background.
13. **Theory:** How would you simulate a "shadow" effect under a sprite using alpha blending?

Game Integration

14. **Project:** Develop a simple "collectathon" game where the player controls a sprite to collect items (static sprites) scattered on the screen.
15. **Code Modification:** Integrate sound effects into SpriteMoveEffects.c (e.g., play a sound when the saucer changes direction).

Blending and Visual Effects

16. **Project:** Use al_draw_tinted_bitmap to create a day-night cycle by gradually tinting the background and sprites.
17. **Theory:** Describe how to use ALLEGRO_FLIP_HORIZONTAL and ALLEGRO_FLIP_VERTICAL to mirror a sprite dynamically.

Performance and Optimization

18. **Project:** Benchmark the performance difference between using individual sprites and a sprite sheet for a 16-frame animation.
19. **Theory:** What steps would you take to optimize memory usage when loading multiple high-resolution sprites?

Practical Challenges

20. **Project:** Create a "bullet hell" demo where the player sprite dodges projectiles (rotating/scaling sprites) fired from the edges of the screen.
21. **Code Modification:** Implement smooth mouse-following movement for the saucer in SpriteMove.c.

Creative Applications

22. **Project:** Design an interactive UI with button sprites that change colour when hovered over and trigger actions (e.g., pause/resume).

https://doi.org/10.15215/remix/9781998944224.01

23. **Code Modification:** Add a "pivot point" to the saucer in SpriteMoveEffects.c so it rotates around its center when moving diagonally.

Exploration and Analysis

24. **Theory:** Research and explain how to implement sprite z-ordering to manage layers (e.g., background, player, foreground).
25. **Project:** Use trigonometric functions to animate a sprite moving in a circular or sine-wave pattern.

Final Project

26. **Capstone:** Build a mini 2D platformer with the following:
    - Animated player sprite (idle, running, jumping)
    - Parallax-scrolling background
    - Collectible items and enemies (dynamic sprites)
    - Collision detection and health bar (using scaling sprites)

Experimental Tasks

27. **Project:** Simulate a "water reflection" effect by drawing a flipped, translucent copy of the sprite below its original position.
28. **Code Modification:** Replace the static background in SpriteMove.c with a dynamically generated starfield using al_draw_pixel.

Documentation and Debugging

29. **Theory:** Explain how to use Allegro's debug tools to identify memory leaks in a sprite-heavy program.
30. **Project:** Write a technical report comparing the use of animated GIFs vs. sprite sheets in Allegro 5, including pros/cons and performance metrics.

https://doi.org/10.15215/remix/9781998944224.01

This page intentionally left blank

5

Programming Backgrounds for Video Games

Backgrounds in video games are essential for setting up the scene and enhancing the visual appeal of the game world.

Backgrounds play the following important roles in video games:

1. **Setting the Scene**
   Backgrounds help establish the game's environment, whether it's a bustling city, a serene forest, or an alien planet. They provide context and make the game world feel more immersive.
2. **Enhancing Atmosphere**
   The design and colour scheme of backgrounds can significantly influence the game's mood and atmosphere. For example, dark, gloomy backgrounds can create a sense of tension and mystery, while bright, colourful backgrounds can evoke a cheerful and adventurous feeling.
3. **Guiding the Player**
   Backgrounds can also serve functional purposes, such as guiding the player's attention to important areas or providing visual cues for navigation.

5.1. Types of Backgrounds

When programming games, there are four types of backgrounds one may choose to use:

https://doi.org/10.15215/remix/9781998944224.01

1. **Static Backgrounds**
   These are nonmoving images that provide a backdrop for the game. They are commonly used in 2D games and can be highly detailed to create a rich environment.
2. **Scrolling Backgrounds**
   Often used in side-scrolling games, these backgrounds move horizontally or vertically to give the illusion of depth and motion. Parallax scrolling, where multiple layers move at different speeds, enhances this effect.
3. **Dynamic Backgrounds**
   These backgrounds change in response to game events or player actions. For example, the background might shift from day to night or change weather conditions.
4. **3D Backgrounds**
   In 3D games, backgrounds are part of the 3D environment and can include complex structures and landscapes. These backgrounds are rendered in real time and can interact with the game's lighting and physics.

5.2. Creating and Using Static Backgrounds

Static backgrounds in video games are nonmoving images that serve as the backdrop for the game's action. They are commonly used in 2D games, especially in genres such as platformers, puzzle games, arcade shooters, and visual novels.

Static backgrounds remain popular because they are:

- **Simple and efficient**, requiring minimal processing power
- **Artistically expressive**, allowing detailed, handcrafted environments
- **Easy to integrate**, since they are drawn once per frame, before all other objects
- **Ideal for stable scenes**, such as menus, overworld maps, and indoor environments

Classic examples include *Super Mario Bros.*, *Mega Man*, *Sonic the Hedgehog*, *The Legend of Zelda*, and many visual novels where the background sets the mood but does not move.

5.2.1. Types of Static Backgrounds

Static backgrounds come in a variety of styles depending on the game's design, visual direction, and performance needs:

https://doi.org/10.15215/remix/9781998944224.01

Full-Screen Static Image: A single image that fills the entire screen
- Common in puzzle games, RPG towns, menus, and narrative scenes
- Easiest type to implement
- Often painted or rendered as a complete scene

Tiled Backgrounds: Built from smaller repeated tiles (e.g., 16 × 16, 32 × 32, or 64 × 64 pixel tiles)
- Saves memory compared to large images
- Provides modularity and easy level creation
- Common in platformers (*Mario*, *Metroid*, *Celeste*)

Layered Static Backgrounds (Non-Parallax): Multiple static layers (foreground, mid-ground, background), but none that scroll or move
- Used for atmospheric depth
- Allows artists to organize elements logically (e.g., walls, props, sky)

Static with Selective Animated Elements: A static background with a few animated components (e.g., flickering lights, moving water)
- Still considered "static backgrounds" because the world behind gameplay does not scroll
- Often implemented using sprites placed on top of the static image
- Good compromise between aesthetics and performance

Room or Scene Backgrounds (Adventure/Visual Novels): Highly detailed illustrations representing rooms, landscapes, or story locations
- The player character may not move relative to the background
- Widely used in visual novel engines and point-and-click adventures

UI-Integrated Backgrounds: Backgrounds forming part of the user interface or menu system
- Main menus, inventory screens, pause menus
- Sometimes blurred or darkened gameplay screens

5.2.2. Technical Considerations

Regardless of art style, implementing static backgrounds follows common technical patterns. In Allegro 5 with C++, the basic steps are as listed here. (For brevity, the header section is omitted from some code examples unless it is required.)

1. **Loading Background Assets**
   Static backgrounds are typically stored as:
   - PNG (most common, supports transparency)
   - JPG (smaller file size, used for detailed/nontransparent art)
   - BMP (large, rarely used today)

https://doi.org/10.15215/remix/9781998944224.01

Using Allegro 5:

```
ALLEGRO_BITMAP *background = al_load_
bitmap("background.png");
```

Best practices:

- Keep background resolution consistent with your game resolution
- Compress images appropriately
- Preload backgrounds during scene initialization to avoid in-game stutter

2. **Drawing the Background**

Static backgrounds should always be drawn before all movable objects.

Basic method:

```
al_draw_bitmap(background, 0, 0, 0);
```

For tiled backgrounds:

```
for (int y = 0; y < SCREEN_H; y += tile_h) {
            for (int x = 0; x < SCREEN_W; x +=
tile_w) {
                        al_draw_bitmap(tile, x, y,
0);
            }
}
```

3. **Scaling, Centering, or Fitting Backgrounds**

Depending on your display resolution, you may need to:

- Scale the background to fit
- Letterbox to maintain aspect ratio
- Crop and center

Example for scaling:

```
al_draw_scaled_bitmap(
background, 0, 0, original_w, original_h,
0, 0, SCREEN_W, SCREEN_H, 0);
```

4. **Managing Layers**

Static backgrounds may consist of multiple layers (foreground, middle ground).

https://doi.org/10.15215/remix/9781998944224.01

Use separate bitmaps:

```
al_draw_bitmap(bg_far, 0, 0, 0);
al_draw_bitmap(bg_mid, 0, 0, 0);
al_draw_bitmap(bg_near, 0, 0, 0);
```

Even without parallax movement, this can create depth and easily accommodate special effects.

5. **Optimizing for Performance**
   Because static backgrounds do not change, the following can be used for optimization:
   - Load them once, not every frame
   - Avoid redrawing complex tiles unless necessary
   - Use Allegro's bitmap flags for speed:

```
al_set_new_bitmap_flags(ALLEGRO_VIDEO_BITMAP);
```

For retro-style pixel art:

```
al_set_new_bitmap_flags(ALLEGRO_MIN_LINEAR |
ALLEGRO_MAG_NEAREST);
```

Static backgrounds are ideal for levels or scenes where the environment does not need to change or move. They are often used in side-scrolling games, adventure games, and certain RPGs.

Classic games like *Super Mario Bros.* and *The Legend of Zelda* use static backgrounds to create immersive worlds without the need for complex animations.

5.2.3. Steps to Take

The next section will show a complete example of implementing a static game background using Allegro in C++. Here are the steps for creating this code:

1. **Initialize Allegro**
   Make sure you have Allegro5 installed and properly set up in your project.
2. **Load the Background Image**
   Use Allegro functions to load your background image.
3. **Draw the Background**
   In your game loop, draw the background image before drawing other game elements.

https://doi.org/10.15215/remix/9781998944224.01

5.2.4. Code Template and Example

Here's a simple example template to illustrate the steps shown in the previous section:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>

int main() {
    // Initialize Allegro
    al_init();
    al_init_image_addon();

    // Create display
    ALLEGRO_DISPLAY *display = al_create_display(800,
600);

    // Load background image
    ALLEGRO_BITMAP *background = al_load_
bitmap("background.png");

    // Main game loop
    while (true) {
        // Clear the screen
        al_clear_to_color(al_map_rgb(0, 0, 0));

        // Draw the background
        al_draw_bitmap(background, 0, 0, 0);

        // Flip the display
        al_flip_display();

        // Add your game logic here

        // Break the loop if the display is closed
        ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();
        ALLEGRO_EVENT event;
        al_wait_for_event(event_queue, &event);
        if (event.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
            break;
        }
    }

    // Clean up
    al_destroy_bitmap(background);
    al_destroy_display(display);

    return 0;
}
```

The main elements from this template will be utilized in the full code example provided next:

1. **Initialization**
   Allegro and the image add-on are initialized.
2. **Loading the Background**
   The background image is loaded using al_load_bitmap.
3. **Drawing the Background**
   The background is drawn in the game loop using al_draw_bitmap.
4. **End**
   Check to see if the Esc key is pressed to exit the example.

When you test the code, make sure to replace background.png with the path to your actual background image file if you don't locate it in the same directory as your executable code (which is where it expects to find the image currently):

COMPLETE CODING EXAMPLE

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>

int main() {
    // Initialize Allegro and image addon
    al_init();
    al_init_image_addon();
    al_install_keyboard();  // Still required to use
keyboard input

    // Create display
```

https://doi.org/10.15215/remix/9781998944224.01

```
    ALLEGRO_DISPLAY *display = al_create_display(800,
600);
    if (!display) return–1;

    // Load background image
    ALLEGRO_BITMAP *background = al_load_
bitmap("background.png");
    if (!background) return –1;

    // Create event queue and register sources
    ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();
    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_keyboard_event_source());

    // Main loop
    bool running = true;
    while (running) {
        // Draw background
        al_clear_to_color(al_map_rgb(0, 0, 0));
        al_draw_bitmap(background, 0, 0, 0);
        al_flip_display();

        // Wait for and process input events
        ALLEGRO_EVENT event;
        if (al_wait_for_event_timed(event_queue, &event,
0.01)) {
            if (event.type == ALLEGRO_EVENT_DISPLAY_
CLOSE) {
                running = false;
            } else if (event.type == ALLEGRO_EVENT_KEY_
DOWN &&
                       event.keyboard.keycode == ALLEGRO_
KEY_ESCAPE) {
                running = false;
            }
        }
    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    // Cleanup
    al_destroy_event_queue(event_queue);
    al_destroy_bitmap(background);
    al_destroy_display(display);

    return 0;
```

5.2.5. Questions for Further Study

1. **Art Style and Composition**
   - How does the choice of static background art style affect gameplay, mood, or readability?
   - When should a developer prefer hand-drawn art over photorealistic backgrounds?
2. **Memory and Optimization**
   - What are the trade-offs between using a single large background image vs. many smaller tiles?
   - How does texture resolution affect GPU memory usage?
3. **Dynamic vs. Static Backgrounds**
   - When is a static background appropriate?
   - How does adding parallax or animation change performance and player experience?
4. **Resolution Independence**
   - How can developers design static backgrounds for multiple screen sizes and aspect ratios?
   - What strategies help reduce distortion (stretching, cropping)?
5. **Pipeline and Tools**
   - What tools are best for creating background art (Photoshop, Krita, Aseprite, Blender)?
   - How do level editors or map editors (Tiled, LDtk) integrate static backgrounds into workflow?
6. **Transitions and Scene Changes**
   - How should a game fade, slide, or transition between static backgrounds?
   - What narrative or pacing effects do transitions create?

https://doi.org/10.15215/remix/9781998944224.01

5.3. Creating and Using Scrolling Backgrounds

Scrolling backgrounds in video games create the illusion of movement and depth, enhancing the visual experience. Here are some key points about scrolling backgrounds.

5.3.1. Types of Background Scrolling

Horizontal Scrolling: Common in side-scrolling games like *Super Mario Bros.*, where the background moves horizontally as the player progresses.

Vertical Scrolling: Seen in games like *Space Invaders*, where the background moves vertically.

Parallax Scrolling: Uses multiple layers moving at different speeds to create a sense of depth. This technique is often used in platformers and adventure games.

5.3.2. Technical Considerations

Tile-Based Scrolling: The background is made up of smaller tiles that are drawn and moved as the player navigates the game world.

Seamless Textures: For continuous scrolling, seamless or tileable textures are used to ensure the background wraps around without visible seams.

Performance: Efficient scrolling techniques are crucial for maintaining smooth gameplay. This often involves optimizing the drawing process and managing memory effectively.

Tools and Libraries: Many game development frameworks and libraries—such as MonoGame, Unity, and Allegro—provide built-in support for implementing scrolling backgrounds.

Artistic Considerations: The design of scrolling backgrounds should complement the game's theme and enhance the player's immersion. Artists often create layered backgrounds to achieve a more dynamic and engaging visual effect.

5.3.3. Steps to Take

Creating a scrolling background in a game using Allegro 5 involves a few steps. Here's a basic guide to help you get started:

1. **Initialize Allegro**
   Ensure Allegro and its add-ons are properly initialized.

https://doi.org/10.15215/remix/9781998944224.01

2. **Load the Background Image**
   Load your background image using Allegro functions.
3. **Implement Scrolling Logic**
   Update the background's position to create the scrolling effect.
4. **Draw the Background**
   Draw the background at its updated position in the game loop.

5.3.4. Code Template and Example

Here's a simple example template to illustrate these steps:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>

int main() {
    // Initialize Allegro
    al_init();
    al_init_image_addon();

    // Create display
    ALLEGRO_DISPLAY *display = al_create_display(800,
600);

    // Load background image
    ALLEGRO_BITMAP *background = al_load_
bitmap("background.png");

    // Variables for scrolling
    float bg_x = 0;
    float scroll_speed = 2.0;

    // Main game loop
    while (true) {
        // Update background position
        bg_x -= scroll_speed;
        if (bg_x <= -al_get_bitmap_width(background)) {
            bg_x = 0;
        }

        // Clear the screen
        al_clear_to_color(al_map_rgb(0, 0, 0));
```

https://doi.org/10.15215/remix/9781998944224.01

```
        // Draw the background twice for seamless scrolling
        al_draw_bitmap(background, bg_x, 0, 0);
        al_draw_bitmap(background, bg_x + al_get_bitmap_
width(background), 0, 0);

        // Flip the display
        al_flip_display();

        // Add your game logic here

        // Break the loop if the display is closed
        ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();
        ALLEGRO_EVENT event;
        al_wait_for_event(event_queue, &event);
        if (event.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            break;
        }
    }

    // Clean up
    al_destroy_bitmap(background);
    al_destroy_display(display);

    return 0;
}
```

The main elements from this template will be utilized in the full code example provided next:

1. **Initialization**
   Allegro and the image add-on are initialized.
2. **Loading the Background**
   The background image is loaded using al_load_bitmap.
3. **Scrolling Logic**
   The background's x-coordinate (bg_x) is updated to create the scrolling effect. When the background moves completely off-screen, its position is reset.
4. **Drawing the Background**
   The background is drawn twice to ensure seamless scrolling.

https://doi.org/10.15215/remix/9781998944224.01

5. **End**
   Check for a key press of the Esc key to end the example.

This is a basic implementation. You can adjust the scroll_speed variable to control the scrolling speed or add more complex logic for different scrolling effects.

CODE EXAMPLE

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>

int main() {
    al_init();
    al_init_image_addon();
    al_install_keyboard();

    ALLEGRO_DISPLAY *display = al_create_display(800,
600);
    if (!display) return-1;

    ALLEGRO_BITMAP *background = al_load_
bitmap("background.png");
    if (!background) return-1;

    ALLEGRO_EVENT_QUEUE *queue =
al_create_event_queue();
    al_register_event_source(queue,
al_get_display_event_source(display))
    al_register_event_source(queue,
al_get_keyboard_event_source());

    ALLEGRO_TIMER *timer = al_create_timer(1.0 /
60.0);     // 60 fps log
    al_register_event_source(queue,
al_get_timer_event_source(timer));
    al_start_timer(timer);

    float bg_x = 0.0f;
    const float scroll_speed = 2.0f;
```

https://doi.org/10.15215/remix/9781998944224.01

```
    bool running = true;
    while (running) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(queue, &ev);

        /*——logic tick——*/
        if (ev.type == ALLEGRO_EVENT_TIMER) {
            bg_x—= scroll_speed;
            if (bg_x <=—al_get_bitmap_width(background))
                bg_x = 0;
        }

        if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE)
            running = false;

        if (ev.type == ALLEGRO_EVENT_KEY_DOWN &&
            ev.keyboard.keycode == ALLEGRO_KEY_ESCAPE)
            running = false;
        if (ev.type == ALLEGRO_EVENT_TIMER) {
            al_clear_to_color(al_map_rgb(0, 0, 0));

            // draw two copies for seamless wrap
            int bw = al_get_bitmap_width(background);
            al_draw_bitmap(background, bg_x, 0, 0);
            al_draw_bitmap(background, bg_x + bw, 0, 0);

            al_flip_display();
        }
    }

    al_destroy_timer(timer);
    al_destroy_event_queue(queue);
    al_destroy_bitmap(background);
    al_destroy_display(display);
    return 0;
}
```

5.3.5. Questions for Further Study

1. How do you add multiple layers for parallax scrolling?
2. Can you explain bitmap handling in Allegro 5?

https://doi.org/10.15215/remix/9781998944224.01

3. What are some common performance tips for scrolling backgrounds?

5.4. Creating and Using Dynamic Backgrounds

Dynamic backgrounds in video games add a layer of immersion and visual interest by changing or reacting to in-game events.

5.4.1. Types of Dynamic Backgrounds

Animated Backgrounds: These include moving elements like clouds, water, or other environmental effects.

Interactive Backgrounds: They may change based on player actions or game events, such as day-night cycles or weather changes.

Procedural Backgrounds: Generated algorithmically, these can create unique and varied environments each time the game is played.

5.4.2. Technical Considerations

Animation: Use sprite sheets or frame-based animations to create dynamic objects on backgrounds.

Shaders: Utilize shaders for complex visual effects like water reflections or dynamic lighting.

Event-Driven Changes: Implement background changes triggered by specific game events, such as entering a new area or completing a level.

Performance Considerations: Dynamic backgrounds can be resource intensive. Optimize performance by managing memory efficiently and minimizing the number of draw calls.

Tools and Libraries: Many game development frameworks and engines—such as Unity, Unreal Engine, and Allegro—provide built-in support for creating dynamic backgrounds.

Artistic Considerations: Dynamic backgrounds should enhance the game's atmosphere and complement the overall art style. They can be used to convey mood, indicate progress, or provide visual feedback to the player.

5.4.3. Steps to Take

Creating a dynamic background for games in C++ with Allegro 5 can add a lot of visual interest to your game. Here's a basic guide to get you started:

https://doi.org/10.15215/remix/9781998944224.01

1. **Initialize Allegro**
   Ensure Allegro and its add-ons are properly initialized.
2. **Load the Background Image**
   Load your background image using Allegro functions.
3. **Implement Dynamic Elements**
   Add elements that change over time, such as moving objects or changing colours.
4. **Draw the Background**
   In your game loop, draw the background and update the dynamic elements.
5. **Cleanup**

5.4.4. Code Template and Example

Here is a basic template:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>

int main() {
    // Initialize Allegro
    al_init();
    al_init_image_addon();
    al_init_primitives_addon();

    // Create display
    ALLEGRO_DISPLAY *display = al_create_display(800, 600);

    // Load background image
    ALLEGRO_BITMAP *background = al_load_
bitmap("background.png");

    // Variables for dynamic elements
    float star_x = 400, star_y = 300;
    float star_speed = 2.0;

    // Main game loop
    while (true) {
        // Update dynamic elements
        star_x += star_speed;
```

https://doi.org/10.15215/remix/9781998944224.01

```
        if (star_x > 800) {
            star_x = 0;
        }

        // Clear the screen
        al_clear_to_color(al_map_rgb(0, 0, 0));

        // Draw the background
        al_draw_bitmap(background, 0, 0, 0);

        // Draw dynamic elements (e.g., a moving star)
        al_draw_filled_circle(star_x, star_y, 5, al_map_
rgb(255, 255, 0));

        // Flip the display
        al_flip_display();

        // Add your game logic here

        // Break the loop if the display is closed
        ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();
        ALLEGRO_EVENT event;
        al_wait_for_event(event_queue, &event);
        if (event.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            break;
        }
    }

    // Clean up
    al_destroy_bitmap(background);
    al_destroy_display(display);

    return 0;
}
```

For movements here, we will use a random number generator to provide the animation for the circle-star object. We use the event timer to trigger the star's movement and adjust its position based on the randomly generated values to offset the x and y positions.

https://doi.org/10.15215/remix/9781998944224.01

CODE EXAMPLE

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <cstdlib>   // for rand()
#include <ctime>     // for seeding rand()

int main() {
    // Initialize Allegro and addons
    al_init();
    al_init_image_addon();
    al_init_primitives_addon();
    al_install_keyboard();
    srand(static_cast<unsigned int>(time(nullptr)));  //
seed RNG

    // Create display
    ALLEGRO_DISPLAY *display = al_create_display(800,
600);
    if (!display) return-1;

    // Load background image
    ALLEGRO_BITMAP *background = al_load_
bitmap("background.png");
    if (!background) return-1;

    // Initial star position
    float star_x = 400, star_y = 300;

    // Create event queue
    ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();
    al_register_event_source(event_queue,
al_get_display_event_source
    al_register_event_source(event_queue,
al_get_keyboard_event_sourc

    // Timer for movement updates
    ALLEGRO_TIMER *timer = al_create_timer(1.0 / 60.0);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_register_event_source(event_queue,
al_get_timer_event_source(t
    al_start_timer(timer);

    bool running = true;
    while (running) {
        ALLEGRO_EVENT event;
        al_wait_for_event(event_queue, &event);

        if (event.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            running = false;
        } else if (event.type == ALLEGRO_EVENT_KEY_DOWN &&
                   event.keyboard.keycode == ALLEGRO_KEY_
ESCAPE) {
            running = false;
            // Here the timer generates a 'move' event 60
times a sec
        } else if (event.type == ALLEGRO_EVENT_TIMER) {
            // Randomly move the star a little bit
            star_x += (rand() % 7–3);  // random move
between–3 an
            star_y += (rand() % 7–3);

            // Keep within bounds
            if (star_x < 0) star_x = 0;
            if (star_x > 800) star_x = 800;
            if (star_y < 0) star_y = 0;
            if (star_y > 600) star_y = 600;

            // Redraw everything
            al_clear_to_color(al_map_rgb(0, 0, 0));
            al_draw_bitmap(background, 0, 0, 0);
            al_draw_filled_circle(star_x, star_y, 5,
al_map_rgb(255,
            al_flip_display();
        }
    }

    // Cleanup
// Cleanup
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_destroy_bitmap(background);
    al_destroy_timer(timer);
    al_destroy_event_queue(event_queue);
    al_destroy_display(display);

    return 0;
}
```

5.4.5. Areas for Completion and Enhancement

Animated Sprites, Livening Up the Scene

In the previous example, we used an Allegro graphics primitive, a filled circle, as the dynamic star object moving around on the background. One can easily replace this with a sprite image (or images of other items) with some simple changes to the code, as seen here. We will utilize the image in figure 5.1 for our star sprite.

In this example, our star sprite is much calmer than our previous "nervous" star circles; it merely glides horizontally across our night sky.

CODE EXAMPLE

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>

int main() {
       // Initialize Allegro
    al_init();
    al_init_image_addon();
    al_init_primitives_addon();
    al_install_keyboard();
```

Figure 5.1: Star sprite image. Image by Walter Ridgewell.

https://doi.org/10.15215/remix/9781998944224.01

```
   // Create display
    ALLEGRO_DISPLAY *display = al_create_display(800,
600);
    if (!display) return-1;
    // Load background image and sprite
    ALLEGRO_BITMAP *background  = al_load_
bitmap("background.png");
    ALLEGRO_BITMAP *star_sprite = al_load_bitmap("star.
png");
    if (!background || !star_sprite) return-1;
    // Get star dimensions
    int star_w = al_get_bitmap_width(star_sprite);
    int star_h = al_get_bitmap_height(star_sprite);

    // Create event queue
    ALLEGRO_EVENT_QUEUE *queue = al_create_event_queue();
    al_register_event_source(queue,
al_get_display_event_source(display));
    al_register_event_source(queue,
al_get_keyboard_event_source());

    //Star initial state
    float star_x = 400.0f, star_y = 300.0f;
    const float speed = 2.0f;

    bool running = true;
    while (running) {
              // Update dynamic elements
        star_x += speed;
        if (star_x > 800) star_x =-star_w;  // wrap at
right edge

                // Clear the screen
        al_clear_to_color(al_map_rgb(0, 0, 0));
        // Draw the background
        al_draw_bitmap(background, 0, 0, 0);
        // Draw sprite instead of filled circle, centered
on star_x, star_y
        al_draw_bitmap(star_sprite, star_x-star_w / 2,
star_y-star_h / 2, 0);
```

https://doi.org/10.15215/remix/9781998944224.01

```
        // Flip the display
        al_flip_display();

               // Wait for an event (non-blocking for
display close)
        ALLEGRO_EVENT ev;
        while (al_get_next_event(queue, &ev)) {
            if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE)
                running = false;
            else if (ev.type == ALLEGRO_EVENT_KEY_DOWN &&
                     ev.keyboard.keycode ==
ALLEGRO_KEY_ESCAPE)
                running = false;
        }
        // Add a short delay for a basic frame rate
control
        al_rest(0.01);
    }

       // Clean up
    al_destroy_event_queue(queue);
    al_destroy_bitmap(star_sprite);
    al_destroy_bitmap(background);
    al_destroy_display(display);
    return 0;
}
```

Changing Weather Effects for the Scene

So now we have a sprite-based starry sky; let's go back and add the primitive graphic, the filled circle we had used as a star previously, and make it represent "snow" to create a wintery snow-falling prenight scene.

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <cstdlib>
#include <ctime>

struct Snowflake {
    float x, y, speed, drift;
};
```

https://doi.org/10.15215/remix/9781998944224.01

```
int main() {
    // Initialize Allegro
    al_init();
    al_init_image_addon();
    al_init_primitives_addon();
    al_install_keyboard();  //Enable keyboard input

    // Create display
    ALLEGRO_DISPLAY *display = al_create_display(800,
600);
    al_set_window_title(display, "Star Sprite with
Snowfall");

    // Load background image and sprite
    ALLEGRO_BITMAP *background = al_load_
bitmap("background.png");
    ALLEGRO_BITMAP *star_sprite = al_load_bitmap("star.
png");

    // Get star dimensions
    int star_width = al_get_bitmap_width(star_sprite);
    int star_height = al_get_bitmap_height(star_sprite);

    // Star motion state
    float star_x = 400, star_y = 300;
    float star_speed = 2.0f;

    // Create event queue and register sources
    ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();
    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue, al_get_
keyboard_event_source());  //Handle ESC

    // Initialize RNG
    std::srand(static_cast<unsigned
int>(std::time(nullptr)));

    // Create snowflakes
    const int NUM_SNOWFLAKES = 150;
```

https://doi.org/10.15215/remix/9781998944224.01

```
    Snowflake snow[NUM_SNOWFLAKES];
    for (int i = 0; i < NUM_SNOWFLAKES; ++i) {
        snow[i].x = std::rand() % 800;
        snow[i].y = std::rand() % 600;
        snow[i].speed = 1 + std::rand() % 3;// Fall speed
        snow[i].drift = ((std::rand() % 21)—10) /
40.0f;// Gentle horizontal drift
    }

    // Main game loop
    bool running = true;
    while (running) {
        // Update star position
        star_x += star_speed;
        if (star_x > 800) star_x =—star_width;

        // Update snowflakes
        for (int i = 0; i < NUM_SNOWFLAKES; ++i) {
            snow[i].y += snow[i].speed;
            snow[i].x += snow[i].drift;
            if (snow[i].y > 600) {
                snow[i].y = 0;
                snow[i].x = std::rand() % 800;
            }
            if (snow[i].x < 0) snow[i].x += 800;
            if (snow[i].x > 800) snow[i].x—= 800;
        }

        // Clear screen
        al_clear_to_color(al_map_rgb(0, 0, 0));
        al_draw_bitmap(background, 0, 0, 0);

                // Draw the gradient overlay (blue to
dusk)
        for (int y = 0; y < 600; ++y) {
            float t = 1.0f—(float)y / 599; //inverted the
gradient to simulate a sunset
            ALLEGRO_COLOR gradient_color = al_map_rgba_f(
                (1.0f—t) * 1.0f + t * 0.0f,
                (1.0f—t) * 0.2f + t * 0.2f,
```

https://doi.org/10.15215/remix/9781998944224.01

```
                (1.0f-t) * 0.2f + t * 1.0f,
                0.3f
            );
            al_draw_filled_rectangle(0, y, 800, y + 1,
gradient_color);
        }

        // Draw star and snowflakes
        al_draw_bitmap(star_sprite, star_x-star_width /
2, star_y-star_height / 2, 0);
        for (int i = 0; i < NUM_SNOWFLAKES; ++i) {
            al_draw_filled_circle(snow[i].x, snow[i].y,
2, al_map_rgba(255, 255, 255, 180));
        }

        // Flip display
        al_flip_display();

        // Handle events (ESC and close)
        ALLEGRO_EVENT event;
        while (al_get_next_event(event_queue, &event)) {
            if (event.type == ALLEGRO_EVENT_DISPLAY_
CLOSE) {
                running = false;
            } else if (event.type == ALLEGRO_EVENT_KEY_
DOWN &&
                       event.keyboard.keycode == ALLEGRO_
KEY_ESCAPE) {
                running = false;
            }
        }

        al_rest(0.01);
    }

    // Cleanup
    al_destroy_event_queue(event_queue);
    al_destroy_bitmap(background);
    al_destroy_bitmap(star_sprite);
    al_destroy_display(display);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    return 0;
}
```

5.4.6. Questions for Further Study

1. How can you implement animated sprites in the background?
2. Can you explain colour-changing effects?
3. What are some tips for optimizing dynamic backgrounds?

5.5. Creating and Using 3D Backgrounds

3D backgrounds in video games add depth and realism, enhancing the overall gaming experience. 3D backgrounds create a more immersive environment, making players feel like they are part of the game world. This is especially important in genres like first-person shooters, RPGs, and adventure games.

5.5.1. Types of 3D Backgrounds

Static 3D Models: These are prerendered 3D models that do not change during gameplay. They provide a detailed and realistic backdrop.

Dynamic 3D Environments: These backgrounds can change in real time based on player actions or game events. Examples include changing weather, day-night cycles, and destructible environments.

Parallax Scrolling: This technique involves multiple layers of 3D backgrounds moving at different speeds to create a sense of depth.

5.5.2. Technical Considerations

Performance Considerations: Rendering 3D backgrounds can be resource intensive. Optimizing performance involves techniques like level of detail (LOD), culling, and efficient use of shaders.

Tools and Engines: Modern game engines like Unity, Unreal Engine, and Godot provide robust tools for creating and managing 3D backgrounds. These engines offer features like real-time lighting, physics, and advanced rendering techniques.

Artistic Considerations: The design of 3D backgrounds should complement the game's art style and enhance the atmosphere. Artists often use a combination of textures, lighting, and environmental effects to achieve the desired look.

https://doi.org/10.15215/remix/9781998944224.01

5.5.3. Steps to Take

Creating a 3D background for games in C++ with Allegro 5 is a bit more complex than working with 2D graphics, as Allegro 5 is primarily a 2D graphics library. However, you can achieve 3D effects by integrating Allegro with OpenGL, which Allegro supports.

1. **Initialize Allegro and OpenGL**
   Ensure Allegro and OpenGL are properly initialized.
2. **Set Up OpenGL**
   Configure OpenGL settings for 3D rendering.
3. **Load and Render 3D Models**
   Use OpenGL functions to load and render 3D models.
4. **Draw the Background**
   In your game loop, draw the 3D background using OpenGL.

5.5.4. Code Template and Example

This is a basic implementation template. For more complex 3D backgrounds, you can load and render 3D models using libraries like assimp for model loading and more advanced OpenGL techniques:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_opengl.h>
#include <GL/gl.h>
#include <GL/glu.h>

void initOpenGL() {
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    gluPerspective(45.0, 800.0 / 600.0, 1.0, 1000.0);
    glMatrixMode(GL_MODELVIEW);
}

void draw3DBackground() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0,
0.0);

    // Draw a simple 3D plane (1 quad or a single square
in OpenGL terminology) as the background
```

https://doi.org/10.15215/remix/9781998944224.01

```
    glBegin(GL_QUADS);
    glColor3f(0.0, 1.0, 0.0); // Green color
    glVertex3f(-1.0,–1.0,–1.0);
    glVertex3f(1.0,–1.0,–1.0);
    glVertex3f(1.0, 1.0,–1.0);
    glVertex3f(-1.0, 1.0,–1.0);
    glEnd();
}

int main() {
    // Initialize Allegro
    al_init();
    al_set_new_display_flags(ALLEGRO_OPENGL);
    ALLEGRO_DISPLAY *display = al_create_display(800,
600);

    // Initialize OpenGL
    initOpenGL();

    // Main game loop
    while (true) {
        // Draw the 3D background
        draw3DBackground();

        // Flip the display
        al_flip_display();

        // Add your game logic here

        // Break the loop if the display is closed
        ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();
        ALLEGRO_EVENT event;
        al_wait_for_event(event_queue, &event);
        if (event.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            break;
        }
    }

    // Clean up
    al_destroy_display(display);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    return 0;
}
```

In this example, there are the following key steps:

1. **Initialization**
   Allegro and OpenGL are initialized.
2. **OpenGL Setup**
   OpenGL settings are configured for 3D rendering.
3. **Drawing the Background**
   A simple 3D cube is drawn as the background.

In the next example, we'll build on the template to create a simple multi-colour rotating cube.

COMPLETE CODING EXAMPLE

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_opengl.h>
#include <GL/gl.h>
#include <GL/glu.h>

static float rotation_angle = 0.0f;

void initOpenGL(int w, int h) {
    glViewport(0, 0, w, h);
    glClearColor(0.06f, 0.06f, 0.08f, 1.0f);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (double)w / (double)h, 0.1,
1000.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void drawCube(float angleDeg) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    glLoadIdentity();
    // Camera
    gluLookAt(0.0, 0.0, 5.0,
              0.0, 0.0, 0.0,
              0.0, 1.0, 0.0);

    // Rotate cube
    glRotatef(angleDeg, 0.0f, 1.0f, 0.0f);
    glRotatef(angleDeg * 0.6f, 1.0f, 0.0f, 0.0f);

    // A unit cube centered at origin, side length 2
    glBegin(GL_QUADS);
    // Front (z = +1)
    glColor3f(1, 0, 0); // red
    glVertex3f(-1,–1, 1);
    glVertex3f( 1,–1, 1);
    glVertex3f( 1, 1, 1);
    glVertex3f(-1, 1, 1);

    // Back (z =–1)
    glColor3f(0, 1, 0); // green
    glVertex3f( 1,–1,–1);
    glVertex3f(-1,–1,–1);
    glVertex3f(-1, 1,–1);
    glVertex3f( 1, 1,–1);

    // Left (x =–1)
    glColor3f(0, 0, 1); // blue
    glVertex3f(-1,–1,–1);
    glVertex3f(-1,–1, 1);
    glVertex3f(-1, 1, 1);
    glVertex3f(-1, 1,–1);

    // Right (x = +1)
    glColor3f(1, 1, 0); // yellow
    glVertex3f( 1,–1, 1);
    glVertex3f( 1,–1,–1);
    glVertex3f( 1, 1,–1);
    glVertex3f( 1, 1, 1);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    // Top (y = +1)
    glColor3f(0, 1, 1); // cyan
    glVertex3f(-1, 1, 1);
    glVertex3f( 1, 1, 1);
    glVertex3f( 1, 1,–1);
    glVertex3f(-1, 1,–1);

    // Bottom (y =–1)
    glColor3f(1, 0, 1); // magenta
    glVertex3f(-1,–1,–1);
    glVertex3f( 1,–1,–1);
    glVertex3f( 1,–1, 1);
    glVertex3f(-1,–1, 1);
    glEnd();
}

int main() {
    al_init();
    al_install_keyboard();

    // Ask Allegro for an OpenGL display with a depth
buffer
    al_set_new_display_flags(ALLEGRO_OPENGL);
    al_set_new_display_option(ALLEGRO_DEPTH_SIZE, 24,
ALLEGRO_REQUIRE);

    const int W = 800, H = 600;
    ALLEGRO_DISPLAY *display = al_create_display(W, H);
    if (!display) return–1;

    initOpenGL(W, H);

    ALLEGRO_EVENT_QUEUE *queue = al_create_event_queue();
    al_register_event_source(queue,
al_get_display_event_source(display));
    al_register_event_source(queue,
al_get_keyboard_event_source());

    bool running = true;
    double last_time = al_get_time();
```

https://doi.org/10.15215/remix/9781998944224.01

```
    while (running) {
        // Update rotation
        double now = al_get_time();
        double dt = now-last_time;
        last_time = now;

        rotation_angle += 60.0f * (float)dt;
        if (rotation_angle >= 360.0f) rotation_angle-=
360.0f;

        // Draw
        drawCube(rotation_angle);
        al_flip_display();

        // Process events (non-blocking)
        ALLEGRO_EVENT ev;
        while (al_get_next_event(queue, &ev)) {
            if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
                running = false;
            } else if (ev.type == ALLEGRO_EVENT_KEY_DOWN
&&
                       ev.keyboard.keycode == ALLEGRO_
KEY_ESCAPE) {
                running = false;
            }
        }
    }

    al_destroy_event_queue(queue);
    al_destroy_display(display);
    return 0;
}
```

5.5.5. Questions for Further Study

1. How can lighting models (such as ambient, diffuse, and specular lighting) be incorporated to improve the realism of 3D backgrounds?
2. What techniques can be used to optimize the performance of complex 3D backgrounds, especially on lower-end hardware?
3. How could texture mapping be applied to 3D background geometry to enhance visual detail?

https://doi.org/10.15215/remix/9781998944224.01

4. In what ways can camera movement and perspective changes contribute to immersion when using 3D backgrounds?
5. How might modern shader-based rendering (e.g., using GLSL) be integrated with Allegro and OpenGL to create more advanced visual effects?

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Static Backgrounds (Section 5.2)

1. **Basic Implementation:** Modify the provided static background code to load and display two different static backgrounds for separate game levels.
2. **Resolution Handling:** Write a program that adjusts the static background image to fit varying screen resolutions without stretching or distortion.
3. **Artistic Integration:** Design a static background for a puzzle game level and implement it in Allegro5. Include at least three interactive UI elements overlaid on the background.
4. **Performance Analysis:** Compare the memory usage of static backgrounds using PNG vs. JPEG formats. Document your findings.

Scrolling Backgrounds (Section 5.3)

5. **Horizontal Scrolling:** Create a side-scrolling game prototype where the background scrolls left to right based on player movement.
6. **Parallax Effect:** Implement a parallax-scrolling system with three layers (e.g., sky, mountains, and ground) moving at different speeds.
7. **Seamless Wrapping:** Modify the provided scrolling code to support vertical scrolling in addition to horizontal movement.
8. **Tile-Based Scrolling:** Design a tile-based background system for a top-down RPG map and implement infinite scrolling.
9. **Optimization Challenge:** Reduce the CPU usage of a scrolling background by implementing dirty-rectangle rendering.

Dynamic Backgrounds (Section 5.4)

10. **Day-Night Cycle:** Create a dynamic background that transitions smoothly between day, sunset, and night using colour interpolation.

https://doi.org/10.15215/remix/9781998944224.01

11. **Weather System:** Simulate rain by animating falling particles over a static background. Use Allegro's primitives or sprite sheets.
12. **Interactive Background:** Design a background where trees sway when the player character approaches them.
13. **Procedural Generation:** Generate a randomized starfield background using algorithms to place stars dynamically.
14. **Shader Effects:** Integrate a fragment shader to create a water reflection effect on a static lake background.

3D Backgrounds (Section 5.5)

15. **Basic 3D Scene:** Use Allegro5's OpenGL integration to render a simple 3D landscape with static models (e.g., hills, trees).
16. **Dynamic Lighting:** Implement a 3D background with a moving light source (e.g., a rotating sun or torch).
17. **Destructible Environment:** Create a 3D scene where parts of the background (e.g., walls) can be destroyed by player actions.
18. **Performance Optimization:** Apply level of detail (LOD) techniques to a 3D background with distant objects rendered at lower resolution.

Mixed Concepts

19. **Hybrid Backgrounds:** Combine a static foreground with a parallax-scrolling middle ground and a dynamic sky background.
20. **Game Jam Project:** Build a minigame where the background changes type (static to scrolling to dynamic) based on player progress.
21. **Debugging Task:** Fix a provided broken scrolling background code that suffers from flickering and memory leaks.

Theory and Analysis

22. **Case Study:** Analyze how *Hollow Knight* uses parallax scrolling to create depth. Write a report comparing it to the Allegro5 implementation.
23. **Colour Theory:** Design a mood chart linking background colour palettes to specific game atmospheres (e.g., horror, adventure).
24. **Performance Trade-Offs:** Debate the advantages of procedural generation vs. prerendered backgrounds in open-world games.

Advanced Projects

25. **Particle System:** Integrate a particle engine into a dynamic background to simulate fire, smoke, or magic effects.

https://doi.org/10.15215/remix/9781998944224.01

26. **Audio-Reactive Background:** Modify a dynamic background to pulse or change colours in sync with background music.
27. **Multiplayer Sync:** Implement a dynamic background (e.g., weather) that synchronizes across two networked players.
28. **Memory Management:** Develop a texture streaming system for large 3D backgrounds to load assets dynamically during gameplay.

Creative Challenges

29. **Art Pipeline:** Create a toolchain (e.g., Python script) to batch-convert and optimize background assets for Allegro5.
30. **Final Project:** Build a complete game level with all four background types (static, scrolling, dynamic, 3D) and a write-up explaining your design choices.

https://doi.org/10.15215/remix/9781998944224.01

This page intentionally left blank

6

Game Design and Development Fundamentals

We learned in the first chapter of this book about different kinds of video games, including action games, adventure games, arcade games, board games, puzzle games, role-playing games, sports games, strategy games, and utilities games, as well as game demos and emulators. This is in no way a complete list, and from time to time, new types of games will be invented and added to the list. For now, however, these genres at least can get you started thinking about what type of game you would like to design and develop as your first video game.

6.1. Conceive an Amazing Game

Creating an amazing video game involves a combination of creativity, technical skills, and strategic planning. Here are some key elements needed to conceive a standout game:

1. **Unique Concept and Vision**
   Original Idea: Start with a unique and engaging concept that sets your game apart from others. This could be a novel gameplay mechanic, an intriguing story, or a distinctive art style.
   Clear Vision: Have a clear vision of what you want to achieve with your game. This includes the overall feel, tone, and experience you want to deliver to players.

https://doi.org/10.15215/remix/9781998944224.01

2. **Strong Game Design**
   Gameplay Mechanics: Develop solid and fun gameplay mechanics that are easy to learn but challenging to master. Ensure the game is balanced and offers a rewarding experience.
   Story and Characters: Create compelling stories and memorable characters that players can connect with. Even in nonnarrative games, a strong theme or setting can enhance the experience.
   Level Design: Design levels that are engaging, varied, and progressively challenging. Good level design keeps players interested and motivated to continue playing.
3. **Technical Proficiency**
   Programming Skills: Strong programming skills are essential for implementing game mechanics, optimizing performance, and ensuring stability.
   Graphics and Animation: High-quality graphics and smooth animations can significantly enhance the visual appeal of your game. Use tools and techniques that suit your game's style.
   Sound Design: Sound effects and music play a crucial role in creating an immersive experience. Invest in good audio design to complement your visuals and gameplay.
6. **User Experience (UX)**
   Intuitive Controls: Ensure that the controls are intuitive and responsive. Players should be able to easily understand and interact with the game.
   User Interface (UI): Design a clean and user-friendly interface that provides necessary information without overwhelming the player.
   Accessibility: Consider accessibility options to make your game playable by a wider audience, including those with disabilities.
5. **Testing and Feedback**
   Playtesting: Regularly playtest your game to identify and fix bugs, balance issues, and other problems. Gather feedback from a diverse group of testers.
   Iterative Development: Be prepared to iterate on your design based on feedback. Continuous improvement is key to refining your game.
6. **Marketing and Community Engagement**
   Marketing Strategy: Develop a marketing strategy to build awareness and excitement for your game. Use social media, trailers, demos, and other promotional materials.
   Community Building: Engage with your community through forums, social media, and events. Building a loyal fanbase can help sustain your game's success.

https://doi.org/10.15215/remix/9781998944224.01

7. **Passion and Perseverance**
   Passion: A genuine passion for game development will drive you to overcome challenges and stay motivated throughout the development process.
   Perseverance: Game development can be a long and challenging journey. Perseverance and a willingness to learn from failures are essential for success.

Combining these elements can help you conceive and develop an amazing video game that stands out in the competitive gaming industry.

6.2. Transforming a Billion-Dollar Idea into a Detailed Game Design

Turning a compelling game concept into a fully detailed design requires a structured approach. This section outlines a practical workflow to help you move from idea to implementation-ready documentation:

1. **Refine the Core Concept**
   Core Idea: Articulate the central theme or mechanic that defines your game. What sets it apart? Is it the narrative, gameplay innovation, or visual style?
   Target Audience: Identify who your game is for. Consider player demographics, preferences, and platform usage to tailor the experience effectively.
2. **Develop a Game Design Document (GDD)**
   The game design document (GDD) serves as the master blueprint for your game. It evolves throughout development and ensures alignment across the team.
   Essential Sections of a GDD:
   Game Overview
   Title: The working or final name of your game
   Genre: Classification (e.g., platformer, RPG, puzzle)
   Platform: Target platforms (e.g., PC, console, mobile)
   Story and Setting: A concise summary of the narrative and world
   Core Gameplay: Description of primary mechanics and player interactions
   Gameplay Details
   Controls: Input schemes and control mappings
   Mechanics: Detailed systems (e.g., combat, puzzles, progression)

https://doi.org/10.15215/remix/9781998944224.01

Levels and Environments: Structure and flow of game stages
Characters: Profiles of key characters, including abilities and roles
Art and Visual Design
Visual Style: Artistic direction, references, and concept art
UI/UX: Interface layout and user experience considerations
Asset List: Inventory of required assets (e.g., models, textures, icons)
Audio Design
Sound Effects: Required audio cues for actions and feedback
Music: Style, themes, and any licensed or original compositions
Technical Specifications
Engine and Tools: Technologies used (e.g., Allegro 5, IDEs)
System Requirements: Minimum and recommended specs
Technical Challenges: Anticipated issues and mitigation strategies
Project Management
Milestones and Timeline: Development phases and deadlines
Team Roles: Responsibilities of each contributor
Budget: Estimated costs for development, marketing, and distribution

3. **Build and Iterate on a Prototype**
   Create a minimal playable version to test core mechanics. Use playtesting feedback to refine gameplay and identify design flaws early.
4. **Create Art and Assets**
   Concept Art: Create visual drafts for characters, environments, and UI.
   3D Models and Textures: Create production assets based on concept art.
   Animations: Create movement and interaction sequences for game elements.
5. **Design Levels**
   Layouts: Develop spatial designs for each level, including objectives and obstacles.
   Pacing and Flow: Ensure a balanced challenge and smooth progression to keep gameplay engaging.
6. **Implement Sound Design**
   Sound Effects: Enhance immersion and feedback.
   Music: Support mood and narrative through audio themes.
7. **Test and Refine**
   Playtesting: Gather user feedback to improve gameplay and balance.
   Quality Assurance: Test systematically to eliminate bugs and polish the experience.

https://doi.org/10.15215/remix/9781998944224.01

8. **Finalize Documentation and Prepare for Release**
   Update GDD: Reflect all changes made during development.
   Release Plan: Outline marketing, distribution, and postlaunch support.

By following this structured process, you can transform a promising idea into a comprehensive and actionable game design. Each phase builds upon the last, ensuring your game is well-conceived, technically sound, and ready for development.

6.3. Implement the Game

Implementing a game from a detailed design involves several stages, each requiring careful planning and execution. The following are steps one should follow:

6.3.1. Set Up Your Development Environment

Choose a Game Engine or Library: Select a game engine or library that suits your project's needs, such as Allegro 5—the one we are using in this book.

Install Necessary Tools: Install the game engine, integrated development environment (IDE), version control system (e.g., Git), and any other tools required for development. In section 1.5, you should have set up your IDE with VS Code, GCC, CMake, and Ninja.

6.3.2. Assemble Your Team

Assign Roles: Ensure each team member knows their responsibilities, such as programming, art, sound design, and project management.

Communication Tools: Set up communication tools (e.g., Slack, Trello) to facilitate collaboration and project tracking.

6.3.3. Create a Project Plan

Milestones and Deadlines: Break down the project into manageable milestones with clear deadlines.

Task Allocation: Assign tasks to team members based on their roles and expertise.

6.3.4. Prototype Core Mechanics

Build a Prototype: Develop a basic prototype to test core gameplay mechanics and ensure they work as intended.

Iterate and Refine: Use feedback from playtesting to refine the mechanics and address any issues.

https://doi.org/10.15215/remix/9781998944224.01

6.3.5. Develop Game Features

Programming: Implement the game's features according to the detailed design document. This includes gameplay mechanics, AI, physics, and user interface.

Art and Animation: Create and integrate art assets, including characters, environments, and animations.

Sound and Music: Develop or source sound effects and music, and integrate them into the game.

6.3.6. Level Design

Design Levels: Create levels based on the detailed design document, ensuring they are engaging and balanced.

Test Levels: Playtest levels to identify and fix any issues with pacing, difficulty, or flow.

6.3.7. User Interface (UI) and User Experience (UX)

Design UI Elements: Create and implement UI elements such as menus, HUDs, and buttons.

Ensure Usability: Test the UI to ensure it is intuitive and user-friendly.

6.3.8. Testing and Quality Assurance

Bug Fixing: Identify and fix bugs through rigorous testing.

Performance Optimization: Optimize the game to ensure it runs smoothly on target platforms.

Playtesting: Conduct extensive playtesting to gather feedback and make necessary adjustments.

6.3.9. Polish and Finalize

Polish: Add final touches to the game, such as visual effects, sound enhancements, and minor gameplay tweaks.

Final Testing: Perform a final round of testing to ensure the game is polished and free of major issues.

6.3.10. Prepare for Launch

Marketing and Promotion: Develop a marketing strategy to build awareness and excitement for your game.

Distribution: Prepare the game for distribution on various platforms (e.g., Steam, consoles, mobile).

https://doi.org/10.15215/remix/9781998944224.01

Launch: Release the game and monitor its performance, addressing any postlaunch issues that arise.

6.3.11. Postlaunch Support

Updates and Patches: Release updates and patches to fix any issues and add new content.

Community Engagement: Engage with your community to gather feedback and build a loyal player base.

By following these steps, you can effectively create a detailed design and implement it, ensuring a smooth development process and a high-quality final product.

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Design and Documentation

1. **Game Design Document (GDD):** Draft a GDD for a 2D action-adventure game, including genre, target audience, core mechanics, and art style.
2. **Concept Pitch:** Create a one-page pitch for a game targeting middle-school players, highlighting unique mechanics, themes, and player engagement goals.
3. **Genre Analysis:** Compare gameplay mechanics of strategy games vs. puzzle games, citing three examples each.
4. **Prototype Proposal:** Outline a prototype for a roguelike game, focusing on procedural generation and permadeath mechanics.
5. **Accessibility Plan:** Design accessibility features (e.g., colour-blind modes, remappable controls) for a hypothetical sports game.

Game Prototype Implementation

6. **Basic Game Loop:** Implement a windowed game loop in Allegro 5 that displays a moving sprite.
7. **Sprite Animation:** Animate a character sprite sheet (e.g., walking cycle) using Allegro's ALLEGRO_BITMAP and timers.
8. **Collision Detection:** Create a minigame where two sprites bounce off each other using bounding box collision.
9. **Tile Map Level:** Design a platformer level using a tile map and render it with Allegro's grid system.

https://doi.org/10.15215/remix/9781998944224.01

10. **Sound Integration:** Add background music and sound effects (e.g., jump, collect) to a simple game using ALLEGRO_SAMPLE.
11. **UI System:** Build a main menu with clickable buttons (Start, Options, Exit) using Allegro's primitives and event handling.
12. **HUD Implementation:** Display a health bar, score counter, and timer in a top-down shooter prototype.
13. **State Management:** Program a pause menu that toggles game states (playing/paused) using finite state machines.
14. **Particle Effects:** Simulate rain or snow using Allegro's particle system or custom sprites.
15. **Save/Load System:** Serialize player progress (e.g., score, level) to a text file and reload it on start-up.
16. **Power-Up System:** Implement collectible power-ups (e.g., speed boost, invincibility) in a platformer.
17. **Physics Simulation:** Code basic platformer physics (gravity, jump velocity) without using external libraries.
18. **Procedural Generation:** Generate randomized terrain for a dungeon crawler using Perlin noise or cellular automata.
19. **Multiplayer Prototype:** Build a local two-player game (e.g., Pong) using split-screen or shared keyboard input.

Testing and Optimization

20. **Playtest Report:** Conduct a playtest for a peer's game and document bugs, balance issues, and UX feedback.
21. **Performance Optimization:** Identify and fix lag in a resource-heavy game (e.g., reduce draw calls, optimize assets).
22. **Unit Testing:** Write test cases for critical functions (e.g., collision detection, score calculation).

Project Management and Collaboration

23. **Version Control Workflow:** Set up a Git repository for a team project, including branching strategies and commit guidelines.
24. **Task Breakdown:** Create a Trello board or Gantt chart for a 3-month game development timeline with milestones.

Postlaunch and Advanced Topics

25. **Marketing Mock-Up:** Design a social media campaign (e.g., Twitter posts, trailer storyboard) for a game launch.
26. **Modding Support:** Allow players to load custom levels from JSON/XML files in a puzzle game.

https://doi.org/10.15215/remix/9781998944224.01

27. **Porting Challenge:** Adapt a desktop game to mobile by adjusting controls and UI scaling (simulate touch input).
28. **Postmortem Analysis:** Reflect on a completed project, detailing successes, failures, and lessons learned.
29. **Community Engagement Plan:** Draft a Discord server structure with channels for feedback, updates, and bug reports.

https://doi.org/10.15215/remix/9781998944224.01

This page intentionally left blank

7

Advanced Topics in Practical Game Programming

Allegro 5 is a powerful library for game development in C++. So far, in the previous chapters, we have learned some fundamental knowledge and skills of game programming with Allegro 5 in C/C++. However, there are still some more advanced topics to explore and learn that can have added benefits for enhancing the graphics, sound, and performance aspects of your game.

7.1. Multithreading

Multithreading is essential in video games for several reasons, primarily related to performance and responsiveness.

7.1.1. Key Benefits of Multithreading

The following are some key benefits multithreading can offer to video games:

Improved Performance

This is achieved through:

Parallel Processing: Multithreading allows different parts of the game to run simultaneously on multiple CPU cores. This parallel processing can significantly improve performance, especially in complex games with many simultaneous tasks.

Efficient Resource Utilization: By distributing tasks across multiple threads, the game can make better use of the available CPU resources, reduce idle time, and increase overall efficiency.

https://doi.org/10.15215/remix/9781998944224.01

Responsiveness

This is reflected as:

Smooth Gameplay: Multithreading helps maintain a smooth and responsive gameplay experience by ensuring that critical tasks, like rendering and input handling, are not delayed by other processes.

Reduced Lag: By handling tasks like physics calculations, AI processing, and asset loading on separate threads, the game can reduce lag and provide a more seamless experience.

Scalability

Future-Proofing: As hardware evolves and more CPU cores become available, multithreaded games can scale to take advantage of these advancements, ensuring better performance on newer systems.

Complex Simulations: Multithreading enables more complex simulations and interactions within the game world, allowing for richer and more detailed environments.

Task Separation

Dedicated Threads for Specific Tasks: Different aspects of the game—such as rendering, physics, AI, and audio—can be handled on separate threads. This separation ensures that intensive tasks do not interfere with one another, leading to a more stable and efficient game.

7.1.2. Uses of Multithreading in Games

The main purpose of using multithreading is to speed things up, and it is often used in the following three areas:

Rendering: One thread can handle rendering graphics while another manages game logic.

Physics: Physics calculations can be offloaded to a separate thread to ensure they do not slow down the main game loop.

AI: AI processing can run on its own thread, allowing for more complex and responsive behaviors without impacting frame rates.

7.1.3. Tools and Techniques for Multithreading

Threading Libraries: Just as in many game engines and game libraries, Allegro 5 provides built-in support for multithreading.

https://doi.org/10.15215/remix/9781998944224.01

Synchronization: Proper synchronization techniques, like mutexes and semaphores, are crucial to avoid issues like race conditions and deadlocks.

Threads: These are independent paths of execution within a program. Use ALLEGRO_THREAD to create and manage threads.

Mutexes: These are used to prevent multiple threads from accessing shared resources simultaneously. Use ALLEGRO_MUTEX to create and manage mutexes.

Conditions: These synchronize threads by allowing them to wait for certain conditions to be met. Use ALLEGRO_COND for condition variables.

7.1.4. Essential Steps to Implement Multithreading

1. **Include Headers and Initialize Allegro**
   As always, ensure your required language library and Allegro add-ons are installed.
2. **Define Shared Data and Mutexes**
   Use ALLEGRO_MUTEX to protect shared resources (e.g., game state, player position).
3. **Separate Thread-Safe and Non-Thread-Safe Tasks**
   Main Thread: Handle rendering, input events, and display updates.
   Worker Threads: Offload tasks like physics, AI, or network communication.
4. **Create Threads with Allegro's Core API**
   Use al_create_thread() to spawn threads and define their entry functions.
5. **Synchronize Threads**
   Use al_lock_mutex() and al_unlock_mutex() to safely access shared data.
6. **Cleanup Resources**
   Join threads with al_join_thread() and destroy mutexes to avoid leaks.

Multithreading is a powerful tool in game development that, when used correctly, can greatly enhance the performance and experience of a game.

The following example demonstrates how to create and manage threads to perform tasks concurrently:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
```

https://doi.org/10.15215/remix/9781998944224.01

```
#include <stdio.h>
#include <stdbool.h>

// Shared mutex for drawing/state updates
ALLEGRO_MUTEX *mutex;
ALLEGRO_FONT *font;

typedef struct {
    int id;              // 1 or 2 (for display)
    int counter;         // per-thread counter
    bool running;        // true while thread loop is
active
} ThreadData;

// Worker thread function
void* worker_thread(ALLEGRO_THREAD *thread, void *arg) {
    ThreadData *td = (ThreadData *)arg;

    // mark running
    al_lock_mutex(mutex);
    td->running = true;
    al_unlock_mutex(mutex);

    while (!al_get_thread_should_stop(thread)) {
        // update this thread's own counter
        al_lock_mutex(mutex);
        td->counter++;
        al_unlock_mutex(mutex);

        // simulate some work at different paces so you
can see both change
        // Thread 1 = 1.0s, Thread 2 = 0.5s
        al_rest(td->id == 1 ? 1.0 : 0.5);
    }

    // mark stopping
    al_lock_mutex(mutex);
    td->running = false;
    al_unlock_mutex(mutex);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    return NULL;
}

int main() {
    // Allegro init
    if (!al_init()) { fprintf(stderr, "Failed to init
Allegro\n"); return–1; }
    al_init_font_addon();
    al_init_ttf_addon();
    al_install_keyboard();

    ALLEGRO_DISPLAY *display = al_create_display(800,
600);
    if (!display) { fprintf(stderr, "Failed to create
display\n"); return–1; }

    font = al_load_ttf_font("arial.ttf", 24, 0);
    if (!font) { fprintf(stderr, "Failed to load font
arial.ttf\n"); return–1; }

    mutex = al_create_mutex();

    // Prepare two thread data blocks
    ThreadData td[2] = {
        {.id = 1, .counter = 0, .running = false},
        {.id = 2, .counter = 0, .running = false}
    };

    // Create & start two threads
    ALLEGRO_THREAD *threads[2];
    threads[0] = al_create_thread(worker_thread, &td[0]);
    threads[1] = al_create_thread(worker_thread, &td[1]);
    al_start_thread(threads[0]);
    al_start_thread(threads[1]);

    // Event loop setup
    ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();
    ALLEGRO_TIMER *timer = al_create_timer(1.0 / 60);  //
60 FPS
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_register_event_source(event_queue,
al_get_keyboard_event_source());
    al_register_event_source(event_queue,
al_get_timer_event_source(timer));
    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_start_timer(timer);

    bool running = true;
    while (running) {
        ALLEGRO_EVENT event;
        al_wait_for_event(event_queue, &event);

        if (event.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            running = false;
        } else if (event.type == ALLEGRO_EVENT_KEY_DOWN
&&
                   event.keyboard.keycode == ALLEGRO_KEY_
ESCAPE) {
            running = false;
        }

        if (event.type == ALLEGRO_EVENT_TIMER ||
            event.type == ALLEGRO_EVENT_DISPLAY_EXPOSE) {
            al_clear_to_color(al_map_rgb(0, 0, 0));

            al_lock_mutex(mutex);
            // Labels show which thread the numbers
belong to
            al_draw_textf(font, al_map_rgb(255, 255,
255), 400, 220, ALLEGRO_ALIGN_CENTRE,
                          "Thread 1 Counter: %d", td[0].
counter);
            al_draw_textf(font, al_map_rgb(200, 200,
0),   400, 260, ALLEGRO_ALIGN_CENTRE,
                          "Thread 1 Status: %s", td[0].
running ? "Running..." : "Stopping");

            al_draw_textf(font, al_map_rgb(255, 255,
255), 400, 320, ALLEGRO_ALIGN_CENTRE,
```

https://doi.org/10.15215/remix/9781998944224.01

```
                            "Thread 2 Counter: %d", td[1].
counter);
            al_draw_textf(font, al_map_rgb(200, 200,
0),   400, 360, ALLEGRO_ALIGN_CENTRE,
                            "Thread 2 Status: %s", td[1].
running ? "Running..." : "Stopping");
            al_unlock_mutex(mutex);

            al_flip_display();
        }
    }

    // Ask both threads to stop and join them
    al_set_thread_should_stop(threads[0]);
    al_set_thread_should_stop(threads[1]);
    al_join_thread(threads[0], NULL);
    al_join_thread(threads[1], NULL);

    // Cleanup
    al_destroy_thread(threads[0]);
    al_destroy_thread(threads[1]);
    al_destroy_mutex(mutex);
    al_destroy_font(font);
    al_destroy_display(display);
    al_destroy_event_queue(event_queue);
    al_destroy_timer(timer);

    return 0;
}
```

This example initializes Allegro, creates two threads that run concurrently, and prints messages to the console. Each thread runs a loop that prints its identifier and a counter value, simulating work by sleeping for one second between iterations. The main program waits for both threads to finish before cleaning up and exiting.

7.2. Custom Shaders

Shaders are powerful tools for creating visual effects in games. We'll use an example to guide you through writing, compiling, and using vertex and

https://doi.org/10.15215/remix/9781998944224.01

fragment shaders with Allegro 5 in C/C++. We'll create a simple colour-shifting effect and a greyscale filter. We'll use OpenGL's functions for doing this process in conjunction with Allegro. First, a short explanation of what's happening in regard to graphics and rendering—the process of putting objects on a computer screen.

When you send a shape such as a triangle, square, or 3D model to the graphics processor, it moves through a series of stages. First, each vertex, or corner point, of the shape is processed. Then the shape is rasterized, which means it gets broken down into fragments, or potential pixels. Each fragment is shaded; given its colour, lighting, and texture; and finally the surviving fragments are output as the pixels you see on your screen. In OpenGL Shading Language (GLSL), we can write shaders that give us control over both the vertex stage and the fragment stage, making it possible to influence how objects are transformed and how their surfaces appear once displayed.

The vertex shader runs once per vertex and handles the geometry side of the pipeline. Its main job is to transform the vertex from model space into screen space and to pass along data like colour, normals, or texture coordinates for later use. After rasterization, the fragment shader takes over, running once for each fragment of the shape. It determines the final pixel colour, applies lighting or textures, and may even discard fragments if they fail certain tests. Shapes are defined by their vertices, but their shading and visual style come from this fragment stage, where interpolated values smooth across surfaces. Put together, the vertex and fragment shaders allow us to move objects into view and shade them so that they look flat, textured, or even fully realistic, depending on the effect we want.

The other concept in GLSL you'll see in the code that we need to cover is the use of vectors. In this context, a vector is simply a container that holds multiple related values together—rather than keeping red, green, blue, and alpha as four separate variables, we group them into a single four-component vector called a vec4. This makes it easier to pass colour information around, perform math on it, and keep it organized as a single unit. GLSL provides vec2, vec3, and vec4 types to represent pairs, triples, or quadruples of values, which map naturally to things like texture coordinates, 3D positions, or RGBA colours. We use a vec4 here because each fragment's colour is made up of four channels, and handling them as a single vector lets the shader process them efficiently while still letting us access the individual components when needed.

7.2.1. Setting Up Allegro 5

First, ensure you have Allegro 5 installed with OpenGL support. Initialize Allegro and its add-ons:

https://doi.org/10.15215/remix/9781998944224.01

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_opengl.h>

int main() {
    al_init();
    al_init_image_addon();
    al_install_keyboard();

    // Create a display with OpenGL context
    al_set_new_display_flags(ALLEGRO_OPENGL);
    ALLEGRO_DISPLAY *display = al_create_display(800,
600);
    ALLEGRO_BITMAP *texture = al_load_bitmap("example.
png");

    // Main loop and cleanup code go here
}
```

7.2.2. Writing Shaders

Shaders are written in GLSL. Two files need to be created:

```
// Vertex Shader (vertex.glsl)
version 330 core
layout(location = 0) in vec2 al_pos;       // Allegro's
position attribute
layout(location = 1) in vec2 al_texcoord;  // Allegro's
texture coordinate attribute
out vec2 frag_uv;

uniform mat4 al_projview_matrix;           // Allegro-
provided projection, the camera perspective
      //or 'view' of the scene if you will

void main() {
    frag_uv = al_texcoord;
    gl_Position = al_projview_matrix * vec4(al_pos, 0.0,
1.0);
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
This passes texture coordinates to the fragment shader.
// Fragment Shader (fragment.glsl)
#version 330 core
in vec2 frag_uv;
out vec4 color;

uniform sampler2D tex;
uniform float time; // for a pulsating effect

void main() {
    vec4 original = texture(tex, frag_uv);
    color = original * (0.5 + 0.5 * sin(time));
}
```

The fragment shader then applies a dynamic effect to those elements.

7.2.3. Compiling and Using Shaders

Allegro provides functions to load and compile shaders:

```
// Load shader sources from files
const char *vertex_src = al_load_shader_source("vertex.
glsl");
const char *fragment_src = al_load_shader_
source("fragment.glsl");

// Create and compile shader program
ALLEGRO_SHADER *shader =
al_create_shader(ALLEGRO_SHADER_GLSL);
al_attach_shader_source(shader, ALLEGRO_VERTEX_SHADER,
vertex_src);
al_attach_shader_source(shader, ALLEGRO_FRAGMENT_SHADER,
fragment_src);

// Check for errors
if (!al_build_shader(shader)) {
    fprintf(stderr, "Shader Error: %s\n",
al_get_shader_log(shader));
    return–1;
}
```

https://doi.org/10.15215/remix/9781998944224.01

7.2.4. Applying Shaders in Rendering in C

```
float time = 0.0;
bool running = true;
ALLEGRO_EVENT_QUEUE *queue = al_create_event_queue();
al_register_event_source(queue,
al_get_keyboard_event_source());

while (running) {
    // Update time for animation
    time += 0.01;

    // Handle input (exit on ESC)
    ALLEGRO_EVENT event;
    while (al_get_next_event(queue, &event)) {
        if (event.type == ALLEGRO_EVENT_KEY_DOWN &&
            event.keyboard.keycode == ALLEGRO_KEY_ESCAPE)
{
            running = false;
        }
    }

    // Use the shader
    al_use_shader(shader);
    al_set_shader_float("time", time); // Pass 'time'
uniform to shader

    // Draw the texture with the shader applied
    al_draw_bitmap(texture, 0, 0, 0);

    al_flip_display();
    al_clear_to_color(al_map_rgb(0, 0, 0));
}

// Cleanup
al_destroy_shader(shader);
al_destroy_bitmap(texture);
al_destroy_display(display);
```

https://doi.org/10.15215/remix/9781998944224.01

7.2.5. Examples of Other Shader Effects

Grayscale Filter

To create a greyscale filter, modify the fragment shader—fragment.glsl—to create averages of the red, green, and blue values (original.r, original.g, and original.b) and apply that to the image:

```
void main() {
    vec4 original = texture(tex, frag_uv);
    float avg = (original.r + original.g + original.b) /
3.0;
    color = vec4(avg, avg, avg, original.a);
}
```

Wave Distortion

Wave distortions are applied by the fragment shader code here:

```
void main() {
    vec2 uv = frag_uv + 0.05 * sin(time + frag_uv.y *
10.0);
    color = texture(tex, uv);
}
```

Next up is a complete example that will load a texture image on the OpenGL fragment (here a rectangle) and slowly fade the image in and out.

VERTEX.GLSL

```
#version 330 core
in vec2 al_pos;         // pixel coords from Allegro
in vec2 al_texcoord;    // UVs from Allegro (usually
0..w, 0..h)
out vec2 frag_uv;

uniform vec2 u_view;    // (display_width, display_height)

void main() {
    frag_uv = al_texcoord;

    // Convert pixels—> Normalized Device Coordinates
    // x: 0..W  ->—1..+1
    // y: 0..H  -> +1..-1  (note the flip)
```

https://doi.org/10.15215/remix/9781998944224.01

```
    vec2 ndc;
    ndc.x = (al_pos.x / u_view.x) * 2.0–1.0;
    ndc.y = 1.0–(al_pos.y / u_view.y) * 2.0;

    gl_Position = vec4(ndc, 0.0, 1.0);
}
```

FRAGMENT.GLSL

```
#version 330 core
out vec4 color;
uniform sampler2D tex;
uniform float time;
void main() {
    vec2 wh = vec2(textureSize(tex, 0));
    vec2 uv = gl_FragCoord.xy / wh;      // 0..1 across
the window/texture
    vec4 original = texture(tex, uv);
    color = original * (0.5 + 0.5 * sin(time));
}
```

SHADERS.CPP

The following is a complete example of custom shaders.

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_opengl.h>
#include <cstdio>
#include <fstream>
#include <string>

// Fallback define for some MinGW header sets
#ifndef GL_SHADING_LANGUAGE_VERSION
#define GL_SHADING_LANGUAGE_VERSION 0x8B8C
#endif

static bool file_exists(const char* path) {
    std::ifstream f(path, std::ios::binary);
    return f.good();
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
int main() {
    // Init Allegro
    if (!al_init()) return-1;
    if (!al_init_image_addon()) return-1;
    al_install_keyboard();

    // Display (OpenGL)
    al_set_new_display_flags(ALLEGRO_OPENGL);
    ALLEGRO_DISPLAY* display = al_create_display(800,
600);
    if (!display) return-1;

    //--Required files present? (only checks we keep)--
    if (!file_exists("vertex.glsl"))   {
std::fprintf(stderr, "Missing vertex.
glsl\n");   return-1; }
    if (!file_exists("fragment.glsl")) {
std::fprintf(stderr, "Missing fragment.glsl\n");
return-1; }

    ALLEGRO_BITMAP* texture = al_load_bitmap("example.
png");
    if (!texture) { std::fprintf(stderr, "Failed to load
example.png\n"); return-1; }
    al_convert_bitmap(texture); // ensure GPU/video
bitmap

    // Shader: attach & build
    ALLEGRO_SHADER* shader =
al_create_shader(ALLEGRO_SHADER_GLSL);
    if (!shader) return-1;

    if (!al_attach_shader_source_file(shader, ALLEGRO_
VERTEX_SHADER, "vertex.glsl")) {
        std::fprintf(stderr, "Vertex attach error: %s\n",
al_get_shader_log(shader));
        return-1;
    }
    if (!al_attach_shader_source_file(shader, ALLEGRO_
PIXEL_SHADER, "fragment.glsl")) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
        std::fprintf(stderr, "Fragment attach error:
%s\n", al_get_shader_log(shader));
        return-1;
    }
    if (!al_build_shader(shader)) {
        std::fprintf(stderr, "Shader build error: %s\n",
al_get_shader_log(shader));
        return-1;
    }

    // Events
    ALLEGRO_EVENT_QUEUE* queue = al_create_event_queue();
    if (!queue) return-1;
    al_register_event_source(queue,
al_get_keyboard_event_source());
    al_register_event_source(queue,
al_get_display_event_source(display));

    // Main loop
    bool running = true;
    double start_time = al_get_time();

    while (running) {
        ALLEGRO_EVENT ev;
        while (al_get_next_event(queue, &ev)) {
            if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE)
running = false;
            if (ev.type == ALLEGRO_EVENT_KEY_DOWN &&
                ev.keyboard.keycode == ALLEGRO_KEY_
ESCAPE) running = false;
        }

        float t = static_cast<float>(al_get_time()-start_
time);

        // Bind shader + uniforms
        al_use_shader(shader);

        // u_view = backbuffer size (pixels) for pixel-
>NDC conversion in vertex.glsl
```

https://doi.org/10.15215/remix/9781998944224.01

```
        ALLEGRO_BITMAP* back = al_get_backbuffer(display);
        float view[2] = {
            (float)al_get_bitmap_width(back),
            (float)al_get_bitmap_height(back)
        };
        al_set_shader_float_vector("u_view", 2, view, 1);

        // bind sampler and time
        al_set_shader_sampler("tex", texture, 1);
        al_set_shader_float("time", t);

        // Draw
        al_clear_to_color(al_map_rgb(0, 0, 0));
        al_draw_bitmap(texture, 0, 0, 0);

        // Unbind and present
        al_use_shader(NULL);
        al_flip_display();
    }

    // Cleanup
    al_destroy_shader(shader);
    al_destroy_bitmap(texture);
    al_destroy_event_queue(queue);
    al_destroy_display(display);
    return 0;
}
```

7.3. Networking for Multiplayer Games

Multiplayer functionality adds a dynamic and engaging layer to games, enabling players to interact in real time across local or global networks. Implementing networking in a game built with Allegro 5 and C/C++ requires understanding both technical foundations and practical design patterns.

7.3.1. Understanding Networking Models

Before diving into implementation, choose the appropriate networking model:

Peer-to-Peer (P2P): Each client communicates directly with others. Suitable for small-scale games but complex to manage synchronization and security.

https://doi.org/10.15215/remix/9781998944224.01

Client-Server: A central server manages game state and communication. This model is more scalable and secure, ideal for most multiplayer games.

7.3.2. Core Concepts

Sockets: This is the mechanism for network communication. C/C++ games typically use reliable TCP (Transmission Control Protocol) or fast but less reliable UDP (User Datagram Protocol) sockets.

Serialization: Game data (e.g., player positions, actions) must be serialized for transmission.

Latency Compensation: This accounts for delays to maintain smooth gameplay.

State Synchronization: This ensures all clients have a consistent view of the game world.

7.3.3. Tools and Libraries

Allegro 5 does not include built-in networking support, so external libraries are required:

Berkeley Sockets (POSIX): Native C socket programming

ENet: Lightweight, reliable UDP library for games

Boost.Asio: Asynchronous networking in C++

RakNet: High-level networking engine tailored for games

7.3.4. Basic Architecture

Server Responsibilities

- Accept incoming connections.
- Maintain authoritative game state.
- Broadcast updates to clients.

Client Responsibilities

- Send player input to the server.
- Receive game state updates.
- Render the game locally.

7.3.5. Implementation Steps

1. Initialize Networking

Server Example: TCP Socket Setup

```
#include <stdio.h>
#include <stdlib.h>
```

https://doi.org/10.15215/remix/9781998944224.01

```
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>

int main() {
int server_fd = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in address;
int addrlen = sizeof(address);

address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(12345);

bind(server_fd, (struct sockaddr *)&address,
sizeof(address));
listen(server_fd, 3);

printf("Server listening on port 12345...\n");

int client_fd = accept(server_fd, (struct sockaddr
*)&address, (socklen_t*)&addrlen);
printf("Client connected!\n");

close(client_fd);
close(server_fd);
return 0;
}
```

2. Define Protocols

Create message formats for player actions and game events. For example:

```
struct PlayerUpdate {
int id;
float x, y;
};
```

3. Handle Communication

Use send() and recv() to transmit serialized data.

https://doi.org/10.15215/remix/9781998944224.01

Client Example: Sending Player Position

```
#include <arpa/inet.h>

int sock = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(12345);
inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr);

connect(sock, (struct sockaddr *)&server_addr,
sizeof(server_addr));

PlayerUpdate update = {1, 100.0f, 200.0f};
send(sock, &update, sizeof(update), 0);
```

Use threads or nonblocking I/O to manage communication without freezing the game loop.

4. Synchronize Game State

Use timestamps or sequence numbers to manage updates. Apply interpolation or prediction to smooth movement.

5. Security Considerations

- Validate all incoming data.
- Use server-side authority to prevent cheating.
- Consider encryption for sensitive data.

7.3.6. Debugging and Testing

Simulate Latency: Introduce artificial delays to test responsiveness.
Log Traffic: Print or store message logs for analysis.
Stress Testing: Connect multiple clients to evaluate performance.

7.3.7. Integration with Allegro

While Allegro 5 handles graphics, input, and audio, networking must be integrated carefully:

- Use Allegro's event system to queue network events.
- Synchronize network updates with the game loop to avoid race conditions.
- Ensure thread safety when accessing shared resources.

https://doi.org/10.15215/remix/9781998944224.01

EXAMPLE: INTEGRATING NETWORK EVENTS WITH ALLEGRO LOOP

```
ALLEGRO_EVENT_QUEUE *queue = al_create_event_queue();
ALLEGRO_TIMER *timer = al_create_timer(1.0 / 60);
al_register_event_source(queue,
al_get_timer_event_source(timer));

al_start_timer(timer);

while (true) {
ALLEGRO_EVENT ev;
al_wait_for_event(queue, &ev);

if (ev.type == ALLEGRO_EVENT_TIMER) {
// Check for network updates
// Update game state
// Render frame
}
}
```

When developing multiplayer games, begin with a simple local multiplayer prototype using loopback networking (127.0.0.1) before expanding to online play.

By incorporating networking into your Allegro 5 game, you unlock the potential for competitive, cooperative, and community-driven gameplay. Though it introduces complexity, a well-designed networking layer can significantly enhance player engagement and replayability.

7.4. AI Programming

Artificial intelligence (AI) is crucial for creating engaging NPCs (nonplayer characters) in games. In this section, we will learn how to implement pathfinding, finite state machines (FSM), and basic decision-making for enemies in Allegro 5 using C/C++. We'll create a simple game where an enemy chases the player using the A* pathfinding algorithm.

7.4.1. Setting Up Allegro 5

First, initialize Allegro and load assets (e.g., sprites, maps).

https://doi.org/10.15215/remix/9781998944224.01

EXAMPLE CODE

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>

#define GRID_SIZE 32 // Tile size for pathfinding grid

int main() {
    al_init();
    al_init_image_addon();
    al_init_primitives_addon();
    al_install_keyboard();

    ALLEGRO_DISPLAY *display = al_create_display(800, 600);
    ALLEGRO_BITMAP *player = al_load_bitmap("player.png");
    ALLEGRO_BITMAP *enemy = al_load_bitmap("enemy.png");

    // Game loop code here...
}
```

7.4.2. Grid-Based Pathfinding with A*

Define the Grid

Create a 2D grid to represent walkable and blocked tiles:

```
#define MAP_WIDTH 25
#define MAP_HEIGHT 18

// 0 = walkable, 1 = blocked
int grid[MAP_HEIGHT][MAP_WIDTH] = {
    {0,0,0,1,0,0,...}, // Example map data
    //...(fill with your own map)
};
```

A Algorithm Implementation*

We can use the A* algorithm to find the shortest path between two points. Think of each walkable tile as a node. For every node, we store:

- Its grid position (x,y)
- The cost from start g
- Our heuristic guess to the goal h

https://doi.org/10.15215/remix/9781998944224.01

- The total score f = g + h
- A pointer to the parent so we can reconstruct the path

We explore nodes using an open set (a priority queue ordered by the smallest f), and a closed set (visited nodes). The heuristic is our "distance guess."

On grids where movement is limited to four directions (up, down, left, and right), the Manhattan distance, calculated as |x1−x2| + |y1−y2|, is a good heuristic choice. It is computationally fast, guarantees optimal paths when used with A*, and helps guide the search efficiently toward the goal.

The processing loop here is:

pop the best node from the open set → if it's the goal, rebuild the
    path via parents →
else relax its neighbors, which means:
    compute tentative new cost g' = g + move_cost, update
        neighbor's g,h,f,parent and
    if this is better, and push/update it in the open set.

This process is repeated until we either reach the goal or the open set empties.

Key Coding Steps

1. Define a Node struct to track grid positions and costs.
2. Use a priority queue to explore nodes.
3. Calculate heuristic (e.g., Manhattan distance).

SIMPLIFIED CODE

```
typedef struct Node {
    int x, y;
    int g, h, f; // g = cost from start, h = heuristic, f
= g + h
    struct Node *parent;
} Node;

// Priority queue and heuristic functions omitted for
brevity

ALLEGRO_PATH *find_path(int start_x, int start_y, int
end_x, int end_y) {
    // Implement A* logic here...
```

https://doi.org/10.15215/remix/9781998944224.01

```
    // Return path as a list of grid positions
}
```

7.4.3. Finite State Machine (FSM) for Enemy Behavior

Define states for the enemy AI (e.g., Idle, Chase, Patrol):

```
typedef enum {
    STATE_IDLE,
    STATE_CHASE,
    STATE_PATROL
} AIState;

typedef struct {
    int x, y; // Current grid position
    AIState state;
    ALLEGRO_PATH *path;
} Enemy;
```

State Transitions

Update the enemy's state based on player proximity:

```
void update_enemy(Enemy *enemy, int player_x, int
player_y) {
    float distance = sqrt(pow(enemy->x–player_x, 2) +
pow(enemy->y–player_y, 2));

    switch (enemy->state) {
        case STATE_IDLE:
            if (distance < 5) { // If player is close
                enemy->state = STATE_CHASE;
                enemy->path = find_path(enemy->x,
enemy->y, player_x, player_y);
            }
            break;
        case STATE_CHASE:
            if (distance > 10) { // If player is far
                enemy->state = STATE_IDLE;
                al_destroy_path(enemy->path);
            }
            break;
```

https://doi.org/10.15215/remix/9781998944224.01

```
        // Add PATROL logic...
    }
}
```

7.4.4. Integrating AI with Allegro's Game Loop

Update and Render

In the game loop, update the enemy's position and draw sprites:

```
Enemy enemy = {12, 5, STATE_IDLE, NULL}; // Starting
position
int player_x = 8, player_y = 10;

while (running) {
    // Handle input (move player with arrow keys)...

    // Update enemy AI
    update_enemy(&enemy, player_x, player_y);

    // Move enemy along path
    if (enemy.state == STATE_CHASE && enemy.path) {
        ALLEGRO_PATH_POINT point;
        al_get_path_point(enemy.path, 0, &point); // Get
next step
        enemy.x = (int)(point.x / GRID_SIZE);
        enemy.y = (int)(point.y / GRID_SIZE);
        al_remove_path_point(enemy.path, 0); // Advance
path
    }

    // Draw everything
    al_clear_to_color(al_map_rgb(0, 0, 0));
    al_draw_bitmap(player, player_x * GRID_SIZE, player_y
* GRID_SIZE, 0);
    al_draw_bitmap(enemy, enemy.x * GRID_SIZE, enemy.y *
GRID_SIZE, 0);
    al_flip_display();
}
```

In the following example, only the Idle and Chase states are implemented. The student can add and experiment with adding code for Patrol and changing

https://doi.org/10.15215/remix/9781998944224.01

the grid matrix to add more blocked (not walkable) tiles. Arrow keys move the player, and if moved in close proximity to the enemy object, it will immediately intercept.

GRIDAI EXAMPLE

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <queue>
#include <vector>
#include <string>
#include <cstring>
#include <algorithm> // for std::reverse

#define GRID_SIZE 32
#define MAP_WIDTH 25
#define MAP_HEIGHT 18
#define SCREEN_WIDTH (MAP_WIDTH * GRID_SIZE)
#define SCREEN_HEIGHT (MAP_HEIGHT * GRID_SIZE)

typedef struct {
    int x, y;
    int g, h;
    int f;
    int came_from_x, came_from_y;
} Node;

// Comparator to use Node directly in priority queue
struct NodeCompareMinF {
    bool operator()(const Node& a, const Node& b)
const {
        return a.f > b.f; // lower f has higher priority
    }
};

typedef enum {
    STATE_IDLE,
```

https://doi.org/10.15215/remix/9781998944224.01

```
    STATE_CHASE
} AIState;

typedef struct {
    float x, y;
    AIState state;
    std::vector<std::pair<int, int>> path;
} Enemy;

int grid[MAP_HEIGHT][MAP_WIDTH] = {
    {0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
};

int heuristic(int x1, int y1, int x2, int y2) {
    return abs(x1—x2) + abs(y1—y2);
}

bool in_bounds(int x, int y) {
    return x >= 0 && y >= 0 && x < MAP_WIDTH && y <
MAP_HEIGHT;
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
std::vector<std::pair<int, int>> find_path(int sx, int
sy, int ex, int ey) {
    std::vector<std::pair<int, int>> path;

    std::priority_queue<Node, std::vector<Node>,
NodeCompareMinF> open_set;
    bool closed[MAP_HEIGHT][MAP_WIDTH] = {false};
    Node came_from[MAP_HEIGHT][MAP_WIDTH];

    Node start = {sx, sy, 0, heuristic(sx, sy, ex, ey),
0,-1,-1};
    start.f = start.g + start.h;
    open_set.push(start);

    while (!open_set.empty()) {
        Node current = open_set.top();
        open_set.pop();

        if (current.x == ex && current.y == ey) {
            while (current.came_from_x !=-1) {
                path.push_back({current.x, current.y});
                current = came_from[current.came_from_y]
[current.came_from_x];
            }
            std::reverse(path.begin(), path.end());
            return path;
        }

        closed[current.y][current.x] = true;

        const int dx[4] = {-1, 1, 0, 0};
        const int dy[4] = {0, 0,-1, 1};

        for (int i = 0; i < 4; i++) {
            int nx = current.x + dx[i];
            int ny = current.y + dy[i];

            if (!in_bounds(nx, ny) || closed[ny][nx] ||
grid[ny][nx] != 0) continue;
```

https://doi.org/10.15215/remix/9781998944224.01

```
            Node neighbor = {nx, ny, current.g + 1,
heuristic(nx, ny, ex, ey), 0, current.x, current.y};
            neighbor.f = neighbor.g + neighbor.h;
            came_from[ny][nx] = current;
            open_set.push(neighbor);
        }
    }

    return path;
}

void update_enemy(Enemy &enemy, int player_x, int
player_y) {
    float distance = sqrt(pow(enemy.x–player_x, 2) +
pow(enemy.y–player_y, 2));

    switch (enemy.state) {
        case STATE_IDLE:
            if (distance < 5) {
                enemy.state = STATE_CHASE;
                enemy.path = find_path((int)enemy.x,
(int)enemy.y, player_x, player_y);
            }
            break;
        case STATE_CHASE:
            if (distance > 10) {
                enemy.state = STATE_IDLE;
                enemy.path.clear();
            }
            break;
    }
}

int main() {
    al_init();
    al_install_keyboard();
    al_init_image_addon();
    al_init_primitives_addon();

    ALLEGRO_DISPLAY *display = al_create_display(SCREEN_
WIDTH, SCREEN_HEIGHT);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    ALLEGRO_EVENT_QUEUE *queue = al_create_event_queue();
    ALLEGRO_TIMER *timer = al_create_timer(1.0 / 60);
    al_register_event_source(queue,
al_get_timer_event_source(timer));
    al_register_event_source(queue,
al_get_keyboard_event_source());

    int player_x = 2, player_y = 2;
    Enemy enemy = {10, 10, STATE_IDLE};

    bool redraw = true, running = true;
    al_start_timer(timer);

    while (running) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(queue, &ev);

        if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
            switch (ev.keyboard.keycode) {
                case ALLEGRO_KEY_ESCAPE:
                    running = false;
                    break;
                case ALLEGRO_KEY_UP:
                    if (in_bounds(player_x, player_y-1)
&& grid[player_y-1][player_x] == 0) player_y-;
                    break;
                case ALLEGRO_KEY_DOWN:
                    if (in_bounds(player_x, player_y + 1)
&& grid[player_y + 1][player_x] == 0) player_y++;
                    break;
                case ALLEGRO_KEY_LEFT:
                    if (in_bounds(player_x-1, player_y)
&& grid[player_y][player_x-1] == 0) player_x-;
                    break;
                case ALLEGRO_KEY_RIGHT:
                    if (in_bounds(player_x + 1, player_y)
&& grid[player_y][player_x + 1] == 0) player_x++;
                    break;
            }
        } else if (ev.type == ALLEGRO_EVENT_TIMER) {
            update_enemy(enemy, player_x, player_y);
```

https://doi.org/10.15215/remix/9781998944224.01

```
            if (enemy.state == STATE_CHASE && !enemy.
path.empty()) {
                enemy.x = enemy.path.front().first;
                enemy.y = enemy.path.front().second;
                enemy.path.erase(enemy.path.begin());
            }

            redraw = true;
        }

        if (redraw && al_is_event_queue_empty(queue)) {
            redraw = false;

            al_clear_to_color(al_map_rgb(0, 0, 0));
            for (int y = 0; y < MAP_HEIGHT; y++) {
                for (int x = 0; x < MAP_WIDTH; x++) {
                    ALLEGRO_COLOR color = grid[y][x] == 1
? al_map_rgb(100, 100, 100) : al_map_rgb(50, 50, 50);
                    al_draw_filled_rectangle(x * GRID_
SIZE, y * GRID_SIZE, (x + 1) * GRID_SIZE, (y + 1) *
GRID_SIZE, color);
                }
            }

            al_draw_filled_circle(player_x * GRID_SIZE
+ GRID_SIZE / 2, player_y * GRID_SIZE + GRID_SIZE / 2,
GRID_SIZE / 2–2, al_map_rgb(0, 255, 0));
            al_draw_filled_circle(enemy.x * GRID_SIZE +
GRID_SIZE / 2, enemy.y * GRID_SIZE + GRID_SIZE / 2, GRID_
SIZE / 2–2, al_map_rgb(255, 0, 0));
            al_flip_display();
        }
    }

    al_destroy_display(display);
    al_destroy_event_queue(queue);
    al_destroy_timer(timer);
    return 0;
}
```

https://doi.org/10.15215/remix/9781998944224.01

7.4.5. Advanced AI Techniques

Steering Behaviors—Changing What the AI Does Based on Choices

Here is a coding example that demonstrates basic movement behavior using seek as a potential AI decision. The enemy moves toward a target position by normalizing the direction vector and advancing at a fixed speed:

```
void seek(Enemy *enemy, int target_x, int target_y) {
    float dx = target_x–enemy->x;
    float dy = target_y–enemy->y;
    float distance = sqrt(dx * dx + dy * dy);

    if (distance > 0) {
        enemy->x += (dx / distance) * speed;
        enemy->y += (dy / distance) * speed;
    }
}
```

Behavior Trees

For more complex AI, we can use behavior trees to manage decision hierarchies, emulating a decision process. Here, we give the AI the choice to chase the user (seek), but if the user moves out of range (flees), the AI returns to a patrol pattern:

```
// Example: Patrol–> Check for Player–> Chase
bool should_chase(Enemy *enemy) {
    return (distance_to_player < 5);
}
```

```
void update_ai(Enemy *enemy) {
    if (should_chase(enemy)) {
        chase_player(enemy);
    } else {
        patrol(enemy);
    }
}
```

The following implementation introduces a patrol behavior for the enemy and incorporates state transitions that allow the enemy to chase the player and return to patrolling if the player escapes:

https://doi.org/10.15215/remix/9781998944224.01

```
// AI_smooth_seek.cpp
// Patrol (square, 5 tiles per edge)—> Check player
distance—> Chase

#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <queue>
#include <vector>
#include <string>
#include <cstring>
#include <algorithm> // reverse

#define GRID_SIZE 32
#define MAP_WIDTH 25
#define MAP_HEIGHT 18
#define SCREEN_WIDTH (MAP_WIDTH * GRID_SIZE)
#define SCREEN_HEIGHT (MAP_HEIGHT * GRID_SIZE)

// ------------------------------------------------------------
// AI state & enemy
// ------------------------------------------------------------
typedef enum {
    STATE_PATROL,
    STATE_CHASE
} AIState;

typedef struct {
    float x, y;                         // grid coords
(integers stored in float)
    AIState state;
    std::vector<std::pair<int, int> > path; // used
during chase

    // Patrol bookkeeping
    int patrol_dir;    // 0=right, 1=down, 2=left, 3=up
```

https://doi.org/10.15215/remix/9781998944224.01

```
    int patrol_steps;  // steps taken along current edge
(0..5)
} Enemy;

// ----------------------------------------------------------
// Grid/map helpers
// ----------------------------------------------------------
int grid[MAP_HEIGHT][MAP_WIDTH] = {
    {0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
};

inline bool in_bounds(int x, int y) {
    return x >= 0 && y >= 0 && x < MAP_WIDTH && y <
MAP_HEIGHT;
}
inline bool is_free(int x, int y) {
    return in_bounds(x, y) && grid[y][x] == 0;
}

// Manhattan heuristic
inline int heuristic(int x1, int y1, int x2, int y2) {
    return abs(x1-x2) + abs(y1-y2);
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
// ------------------------------------------------------------
// Safe A* (fully initialized parents, guarded
reconstruction)
// ------------------------------------------------------------
std::vector<std::pair<int,int> > find_path(int sx, int
sy, int ex, int ey) {
    std::vector<std::pair<int,int> > path;

    if (!in_bounds(sx, sy) || !in_bounds(ex, ey)) return
path;
    if (!is_free(sx, sy) || !is_free(ex, ey)) return path;

    struct Parent { short px, py; };
    const int INF = 1e9;
    const int MAX_EXPANSIONS = MAP_WIDTH * MAP_HEIGHT *
8; // safety cap

    struct QN { int x, y, g, f; };
    struct Cmp { bool operator()(const QN& a, const QN&
b) const { return a.f > b.f; } };

    int g_cost[MAP_HEIGHT][MAP_WIDTH];
    bool closed[MAP_HEIGHT][MAP_WIDTH];
    Parent parent[MAP_HEIGHT][MAP_WIDTH];

    for (int y = 0; y < MAP_HEIGHT; ++y) {
        for (int x = 0; x < MAP_WIDTH; ++x) {
            g_cost[y][x] = INF;
            closed[y][x] = false;
            parent[y][x] = {-1,-1 };
        }
    }

    std::priority_queue<QN, std::vector<QN>, Cmp> open;
    g_cost[sy][sx] = 0;
    open.push({ sx, sy, 0, heuristic(sx, sy, ex, ey) });

    const int dx[4] = { 1,-1, 0, 0 };
    const int dy[4] = { 0, 0, 1,-1 };
    int expansions = 0;
```

https://doi.org/10.15215/remix/9781998944224.01

```
    while (!open.empty()) {
        QN cur = open.top(); open.pop();
        if (closed[cur.y][cur.x]) continue;
        closed[cur.y][cur.x] = true;

        if (++expansions > MAX_EXPANSIONS) {
            // safety bail-out → no path
            return std::vector<std::pair<int,int> >();
        }

        if (cur.x == ex && cur.y == ey) {
            // Reconstruct
            int cx = ex, cy = ey;
            std::vector<std::pair<int,int> > rev;
            while (!(cx == sx && cy == sy)) {
                rev.push_back(std::make_pair(cx, cy));
                Parent p = parent[cy][cx];
                if (p.px < 0 || p.py < 0) { // broken
parent chain → give up safely
                    rev.clear();
                    break;
                }
                cx = p.px; cy = p.py;
            }
            std::reverse(rev.begin(), rev.end());
            return rev; // may be empty if sx==ex &&
sy==ey or chain broken
        }

        for (int i = 0; i < 4; ++i) {
            int nx = cur.x + dx[i];
            int ny = cur.y + dy[i];
            if (!is_free(nx, ny) || closed[ny][nx])
continue;

            int tentative_g = cur.g + 1;
            if (tentative_g < g_cost[ny][nx]) {
                g_cost[ny][nx] = tentative_g;
                parent[ny][nx] = { (short)cur.x, (short)
cur.y };
```

https://doi.org/10.15215/remix/9781998944224.01

```
                int f = tentative_g + heuristic(nx, ny,
ex, ey);
                open.push({ nx, ny, tentative_g, f });
            }
        }
    }
    return path; // empty → no path
}

// ──────────────────────────────────────────────────────
// Behavior tree-style AI helpers
// ──────────────────────────────────────────────────────
static inline float dist_to_player(const Enemy* e, int
px, int py) {
    float dx = float(e->x)−float(px);
    float dy = float(e->y)−float(py);
    return sqrtf(dx*dx + dy*dy);
}

bool should_chase(Enemy* enemy, int player_x, int
player_y) {
    return dist_to_player(enemy, player_x, player_y) <
5.0f;
}

void chase_player(Enemy* enemy, int player_x, int
player_y) {
    enemy->path = find_path((int)enemy->x, (int)enemy->y,
player_x, player_y);
}

// Patrol: square march, 5 tiles per edge; 90° right
turns
void patrol(Enemy* enemy) {
    if (enemy->patrol_steps >= 5) {
        enemy->patrol_steps = 0;
        enemy->patrol_dir = (enemy->patrol_dir + 1) % 4;
// right turn
    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    int nx = (int)enemy->x;
    int ny = (int)enemy->y;

    switch (enemy->patrol_dir) {
        case 0: nx++; break; // right
        case 1: ny++; break; // down
        case 2: nx—; break; // left
        case 3: ny—; break; // up
    }

    if (is_free(nx, ny)) {
        enemy->x = nx;
        enemy->y = ny;
        enemy->patrol_steps++;
    } else {
        // If blocked or out of bounds, force a turn next
tick
        enemy->patrol_steps = 5;
    }
}

void update_ai(Enemy* enemy, int player_x, int player_y)
{
    if (should_chase(enemy, player_x, player_y)) {
        enemy->state = STATE_CHASE;
        // Only (re)plan when we have no path; keeps
things light
        if (enemy->path.empty()) {
            chase_player(enemy, player_x, player_y);
        }
    } else {
        if (enemy->state != STATE_PATROL) {
            enemy->state = STATE_PATROL;
            enemy->patrol_dir = 0;
            enemy->patrol_steps = 0;
            enemy->path.clear();
        }
        patrol(enemy);
    }
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
// ------------------------------------------------------------
// Demo loop
// ------------------------------------------------------------
int main() {
    al_init();
    al_install_keyboard();
    al_init_image_addon();
    al_init_primitives_addon();

    ALLEGRO_DISPLAY *display = al_create_display(SCREEN_
WIDTH, SCREEN_HEIGHT);
    ALLEGRO_EVENT_QUEUE *queue = al_create_event_queue();
    ALLEGRO_TIMER *timer = al_create_timer(1.0 / 8.0); //
slow to visualize
    al_register_event_source(queue,
al_get_timer_event_source(timer));
    al_register_event_source(queue,
al_get_keyboard_event_source());
    al_register_event_source(queue,
al_get_display_event_source(display));

    int player_x = 2, player_y = 2;

    Enemy enemy;
    enemy.x = 10; enemy.y = 10;
    enemy.state = STATE_PATROL;
    enemy.patrol_dir = 0;
    enemy.patrol_steps = 0;
    enemy.path.clear();

    bool redraw = true, running = true;
    al_start_timer(timer);

    while (running) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(queue, &ev);

        if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            running = false;
        } else if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
            switch (ev.keyboard.keycode) {
                case ALLEGRO_KEY_ESCAPE: running = false;
break;
                case ALLEGRO_KEY_UP:    if (is_
free(player_x, player_y-1))—player_y; break;
                case ALLEGRO_KEY_DOWN:  if (is_
free(player_x, player_y+1)) ++player_y; break;
                case ALLEGRO_KEY_LEFT:  if (is_
free(player_x-1, player_y))—player_x; break;
                case ALLEGRO_KEY_RIGHT: if (is_
free(player_x+1, player_y)) ++player_x; break;
            }
        } else if (ev.type == ALLEGRO_EVENT_TIMER) {
            // Update AI (BT root)
            update_ai(&enemy, player_x, player_y);

            // Step along path while chasing
            if (enemy.state == STATE_CHASE) {
                // if already at target, drop back to
patrol next update
                if ((int)enemy.x == player_x && (int)
enemy.y == player_y) {
                    enemy.state = STATE_PATROL;
                    enemy.patrol_steps = 0;
                    enemy.path.clear();
                } else {
                    if (enemy.path.empty()) {
                        // If no path available (e.g.,
blocked), try to replan
                        enemy.path = find_path((int)
enemy.x, (int)enemy.y, player_x, player_y);
                    }
                    if (!enemy.path.empty()) {
                        enemy.x = enemy.path.front().
first;
                        enemy.y = enemy.path.front().
second;
                        enemy.path.erase(enemy.path.
begin());
                    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
                }
            }

            redraw = true;
        }

        if (redraw && al_is_event_queue_empty(queue)) {
            redraw = false;

            al_clear_to_color(al_map_rgb(0, 0, 0));
            for (int y = 0; y < MAP_HEIGHT; y++) {
                for (int x = 0; x < MAP_WIDTH; x++) {
                    ALLEGRO_COLOR color = grid[y][x] == 1
? al_map_rgb(100, 100, 100)
                                                        :
al_map_rgb(50, 50, 50);
                    al_draw_filled_rectangle(x * GRID_
SIZE, y * GRID_SIZE,
                                             (x + 1) *
GRID_SIZE, (y + 1) * GRID_SIZE, color);
                }
            }

            // Player (green) and Enemy (red)
            al_draw_filled_circle(player_x * GRID_SIZE +
GRID_SIZE / 2,
                                  player_y * GRID_SIZE +
GRID_SIZE / 2,
                                  GRID_SIZE / 2–2, al_
map_rgb(0, 255, 0));

            al_draw_filled_circle((int)enemy.x * GRID_
SIZE + GRID_SIZE / 2,
                                  (int)enemy.y * GRID_
SIZE + GRID_SIZE / 2,
                                  GRID_SIZE / 2–2, al_
map_rgb(255, 0, 0));

            // Optional: visualize chase path
            /*
```

https://doi.org/10.15215/remix/9781998944224.01

```
            for (size_t i = 0; i < enemy.path.size();
++i) {
                int px = enemy.path[i].first;
                int py = enemy.path[i].second;
                al_draw_filled_circle(px * GRID_SIZE +
GRID_SIZE/2,
                                      py * GRID_SIZE +
GRID_SIZE/2,
                                      GRID_SIZE/6, al_
map_rgb(255, 128, 0));
            }
            */

            al_flip_display();
        }
    }

    al_destroy_display(display);
    al_destroy_event_queue(queue);
    al_destroy_timer(timer);
    return 0;
}
```

7.5. Optimization Techniques

Optimization ensures your game runs smoothly and efficiently across various hardware configurations. This section covers some practical techniques to enhance performance (and in some cases, just good resource management options) in Allegro 5, focusing on rendering, memory management, collision detection, and code efficiency.

7.5.1. Efficient Rendering Techniques

Use Texture Atlases

Consider combining multiple sprites into a single texture to reduce draw calls. We've seen this concept earlier in the sprite sheets used for animation, but a similar concept can be used for other tasks. You might, for example, put all your sprite images for terrain objects you will use in a game—rock types, ground textures, plants, and so on—in one large image composed of the individual sprite elements. When you need to access an item, you can then utilize a

https://doi.org/10.15215/remix/9781998944224.01

section of the larger image containing the item you want. So here, you load a single resource and utilize parts of it rather than needing multiple loads.

EXAMPLE

```
ALLEGRO_BITMAP *atlas = al_load_bitmap("texture_atlas.
png");

// Draw a specific region/item from the atlas
al_draw_bitmap_region(
    atlas,
    x, y,        // Source coordinates
    width,       // Region size
    height,
    dest_x,      // Destination coordinates
    dest_y,
    0
);
```

Batch Drawing

Group similar draw calls to minimize state changes. Here, I'm talking about drawing as many things as possible that share the same GPU state—same texture (atlas), same shader, same blend mode, same transforms—in one go. Every time we switch textures, bind a new shader, tweak blending, or swap target bitmaps, the driver has to reconfigure the pipeline, and that costs time. So instead of drawing one tiny piece at a time and constantly flipping states, we sort our work (e.g., by texture/atlas) and feed the GPU batches of vertices at once.

In Allegro, that process could be, for example, building an array of ALLEGRO_VERTEX that all reference the same atlas, then issuing a single al_draw_prim() to cover the whole set.

Why do this? Because batching keeps the GPU busy doing the fast part (rasterizing triangles) while avoiding the slow part (state churn on the CPU/driver side).

EXAMPLE

```
ALLEGRO_VERTEX vertices[4]; // Define vertices for a quad
//...populate vertices...

// Draw all vertices in one call
al_draw_prim(vertices, NULL, atlas, 0, 4,
ALLEGRO_PRIM_TRIANGLE_FAN);
```

https://doi.org/10.15215/remix/9781998944224.01

So here, we define ALLEGRO_VERTEX vertices[4] for a quad (two triangles) and render in one call with the shared atlas. Here, ALLEGRO_PRIM_TRIANGLE_FAN is just a primitive type constant that tells Allegro how to interpret the list of vertices you've passed in. It matches the OpenGL concept of a "triangle fan."

What a Triangle Fan Is

- Imagine you've got a central vertex, and every new vertex you add forms a triangle that "fans out" from that center.
- With 4 vertices, you can form a quad (two triangles).
- With more, you can create convex polygons very efficiently.

For example,

- Vertices: [v0, v1, v2, v3]
- Triangles drawn:
  - (v0, v1, v2)
  - (v0, v2, v3)

So you only need to specify each new vertex once instead of repeating corners like you would in a triangle list.

Now scale that thinking up: Accumulate many quads that all use the same atlas, then call al_draw_prim once for the entire batch. The same idea applies if you're drawing bitmaps: you can use a texture atlas and al_hold_bitmap_drawing(true) to reduce binds. The net result is fewer state switches, fewer driver round-trips, and measurably better frame times, especially when you've got lots of small sprites.

Enable Hardware Rendering

By default, Allegro will give you a hardware-accelerated display, but sometimes it's worth being explicit with ALLEGRO_OPENGL or ALLEGRO_DIRECT3D. The main reason is control: If you know you're going to be writing specifically for one of those two—perhaps depending on your development platform of choice or because you're creating your own shaders, sampling textures per fragment, or using GLSL features—you don't want Allegro to quietly hand you a Direct3D context on Windows when you want to use OpenGL. By forcing OpenGL (or conversely, Direct3D), you're guaranteeing that your vertex and fragment shaders will compile and run consistently, and that your fragment operations (like custom lighting, greyscale conversion, or texture effects) behave exactly the way you designed them.

Another case is debugging; it helps to know that for sure you're working in OpenGL space and can reference vec4, frag_uv, and the rest without worrying

https://doi.org/10.15215/remix/9781998944224.01

about backend differences. In short, the flag isn't needed for performance, but it's useful whenever you care about shader compatibility or you want your fragment processing code to behave the same way across platforms.

Specify OpenGL with the flag:

```
al_set_new_display_flags(ALLEGRO_OPENGL); // Enable OpenGL
ALLEGRO_DISPLAY *display = al_create_display(800, 600);
```

7.5.2. Memory Management

While systems these days come with plenty of memory, it's *always* good coding practice to manage the use and freeing of that resource properly. Memory leaks are a bane of developers to this day, causing games to eventually slow or outright crash, so a little preventive work here can save time and trouble later.

Destroy Unused Resources

Free bitmaps, fonts, and sounds when no longer needed:

```
ALLEGRO_BITMAP *sprite = al_load_bitmap("sprite.png");
//...use sprite...
al_destroy_bitmap(sprite); // Prevent memory leaks
```

Reuse Objects

Cache frequently used assets (e.g., fonts, particle effects):

```
ALLEGRO_FONT *font = al_load_font("arial.ttf", 24, 0);
// Reuse 'font' across multiple menus instead of
reloading.
```

7.5.3. Collision Detection Optimization

Spatial Partitioning with Grids

Here, we can speed up collision checks by binning objects into a coarse grid. Instead of checking every object against every other one (which can, depending on the number of objects, get cumbersome), each object goes into a cell based on its position, and we only test it against objects in its own cell and its eight neighbors. That keeps comparisons local and less intensive CPU-wise.

Divide the game world into cells to limit collision checks:

```
#define CELL_SIZE 64
int grid[SCREEN_WIDTH / CELL_SIZE][SCREEN_HEIGHT /
CELL_SIZE];
```

https://doi.org/10.15215/remix/9781998944224.01

```
// Update grid with object positions
void update_grid(GameObject *obj) {
    int cell_x = obj->x / CELL_SIZE;
    int cell_y = obj->y / CELL_SIZE;
    grid[cell_x][cell_y].add(obj);
}

// Check collisions only within nearby cells
void check_collisions(GameObject *obj) {
    int cell_x = obj->x / CELL_SIZE;
    int cell_y = obj->y / CELL_SIZE;
    for (int i = cell_x-1; i <= cell_x + 1; i++) {
        for (int j = cell_y-1; j <= cell_y + 1; j++) {
            // Check objects in neighboring cells
        }
    }
}
```

Use Bounding Boxes

Here, we have a concept we looked at earlier: The idea of checking every pixel location of a sprite for a collision (overlap) vs. having a shaped bounding box that would be applicable for the size and shape of sprites in use.

Replace pixel-perfect checks with simplified bounding boxes:

```
bool collision_check(GameObject *a, GameObject *b) {
    return (a->x < b->x + b->width &&
            a->x + a->width > b->x &&
            a->y < b->y + b->height &&
            a->y + a->height > b->y);
}
```

7.5.4. Code-Level Optimizations

Here, we have some common coding efficiencies.

Avoid Expensive Operations in Loops

Precompute values outside loops:

```
// Bad: sqrt() called every iteration
for (int i = 0; i < 1000; i++) {
    distance = sqrt(dx * dx + dy * dy);
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
// Good: Precompute squared distance
float dist_sq = dx * dx + dy * dy;
if (dist_sq < threshold_sq) {...}
```

Use Efficient Data Structures

Replace linked lists with arrays or spatial grids for faster iteration:

```
GameObject *objects[MAX_OBJECTS]; // Faster traversal
than linked lists
```

7.5.5. Multithreading

Offload Nonrendering Tasks

Use Allegro's threading API for tasks like pathfinding or AI:

```
void* ai_thread(ALLEGRO_THREAD *thread) {
    while (!al_get_thread_should_stop(thread)) {
        update_enemy_ai(); // Run AI logic in parallel
    }
    return NULL;
}

// Start thread
ALLEGRO_THREAD *thread = al_create_thread(ai_thread,
NULL);
al_start_thread(thread);
```

7.5.6. Profiling and Debugging

Every once in a while a developer might want to check on the performance of their code; a typical metric might be the frames per second (fps) your game runs at or the number of dropped frames when not reaching your desired fps. Here, the developers can use small routines in their game for that purpose.

Measure Frame Time

Track frame duration to identify slowdowns:

```
double prev_time = al_get_time();
while (running) {
    double current_time = al_get_time();
    double delta_time = current_time–prev_time;
    prev_time = current_time;
```

https://doi.org/10.15215/remix/9781998944224.01

```
    if (delta_time > 0.017) { // >60 FPS?
        printf("Frame drop: %.4f ms\n", delta_time *
1000);
    }
}
```

In the area of debugging code, this can be as simple as printing out to the console screen variable values, including lines to indicate "I'm in this part of the program code," "Now executing," or other status-indicating displays.

Use Profiling Tools

Use tools like Valgrind (Linux) or Visual Studio Profiler (Windows) to detect memory leaks or hotspots.

7.5.7. Asset Optimization

Asset optimization reduces file size and runtime cost by using only the assets necessary for the target platform and experience. Choose image formats, resolutions, and compression levels that match your chosen screen sizes and visual fidelity. Use scaled or tiled textures for large backgrounds, sprite atlases to cut draw calls, and vector art for UI where appropriate.

For audio, pick sample rates and bitrates that preserve perceived quality while minimizing size. Test perceptual differences—players rarely notice modest bitrate reductions, so prefer smaller formats when the difference is negligible (for example, lower bitrates for background music or short SFX). Compress and trim audio assets, and stream long music tracks instead of loading them entirely into memory.

Build automated steps into your pipeline to:

- Resize and compress images for each target resolution.
- Pack sprites and icons into atlases.
- Normalize audio levels, compress files, and stream long music tracks instead of loading them into memory.
- Strip unused data from assets and export only required variants.

Measure memory use, load times, and visual/audio fidelity on real devices, then iterate until you hit the best balance of performance, storage, and player experience.

Compress Textures

Use compressed formats like .png with al_load_bitmap_flags:

https://doi.org/10.15215/remix/9781998944224.01

```
ALLEGRO_BITMAP *sprite = al_load_bitmap_flags(
    "sprite.png",
    ALLEGRO_NO_PREMULTIPLIED_ALPHA // Reduce blending cost
);
```

Downsample Audio

Convert audio to lower bitrates if high quality isn't critical:

```
ALLEGRO_SAMPLE *sound = al_load_sample("sound.wav");
al_reserve_samples(10); // Preload samples to avoid
runtime delays
```

7.5.8. Testing and Scaling

One of the features of the Allegro library is it can run on multiple platforms, and so the possibility exists for it to use unconventional screen sizes. As such, you might want to explore the way your game looks at various resolutions; your 4K HD masterpiece won't look the same at 1024 or 720. Or perhaps you'll want to provide an option to players to lower the quality of your game graphics for better performance, a standard feature in all games these days.

Test on low-end hardware and adjust settings dynamically:

```
void adjust_graphics_quality() {
    if (system_slow) {
        al_set_new_bitmap_flags(ALLEGRO_MIN_LINEAR); //
Lower filtering quality
        disable_shaders();
    }
}
```

7.6. Custom GUI Systems

In this section, we will learn how to build custom graphical user interfaces (GUIs) for games using Allegro 5. We will create reusable components like buttons, sliders, and menus and handle user interactions such as clicks and hover effects. By the end, you'll have a functional main menu and options screen.

7.6.1. Setting Up the Project

Initialize Allegro and Create a Window

Start by setting up Allegro and creating a display window:

https://doi.org/10.15215/remix/9781998944224.01

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <allegro5/allegro_primitives.h>

#define SCREEN_WIDTH 800
#define SCREEN_HEIGHT 600

int main() {
    al_init();
    al_init_font_addon();
    al_init_ttf_addon();
    al_init_primitives_addon();
    al_install_keyboard();
    al_install_mouse();

    ALLEGRO_DISPLAY *display = al_create_display(SCREEN_
WIDTH, SCREEN_HEIGHT);
    ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();
    ALLEGRO_FONT *font = al_load_ttf_font("arial.ttf",
24, 0);

    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_mouse_event_source());

    bool running = true;
    // Game loop here...

    al_destroy_display(display);
    return 0;
}
```

https://doi.org/10.15215/remix/9781998944224.01

7.6.2. Building a Button Component

Define a Button Struct

Create a reusable Button struct to track position, size, text, and state:

```
typedef struct {
    int x, y;          // Position
    int width, height; // Size
    char label[50];    // Text label
    bool hovered;      // Hover state
    bool clicked;      // Click state
} Button;
```

Draw and Update the Button

Add functions to draw the button and check for interactions:

```
// Draw the button with hover/click effects
void draw_button(Button *button, ALLEGRO_FONT *font) {
    ALLEGRO_COLOR bg_color = button->hovered ? al_map_
rgb(100, 100, 255) : al_map_rgb(50, 50, 200);
    if (button->clicked) bg_color = al_map_rgb(200, 50, 50);

    al_draw_filled_rectangle(
        button->x, button->y,
        button->x + button->width,
        button->y + button->height,
        bg_color
    );
    al_draw_text(
        font, al_map_rgb(255, 255, 255),
        button->x + button->width/2,
        button->y + button->height/2–12,
        ALLEGRO_ALIGN_CENTRE,
        button->label
    );
}

// Check if the mouse is over the button
void update_button(Button *button, int mouse_x, int
mouse_y, bool mouse_clicked) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
    button->hovered = (
        mouse_x >= button->x &&
        mouse_x <= button->x + button->width &&
        mouse_y >= button->y &&
        mouse_y <= button->y + button->height
    );
    button->clicked = button->hovered && mouse_clicked;
}
```

7.6.3. Creating a Main Menu

Initialize Buttons

Create buttons for "Start Game" and "Options":

```
Button start_button = {300, 200, 200, 50, "Start Game",
false, false};
Button options_button = {300, 300, 200, 50, "Options",
false, false};
```

Handle Mouse Input

In the game loop, track mouse events and update buttons:

```
ALLEGRO_EVENT event;
bool mouse_clicked = false;
int mouse_x = 0, mouse_y = 0;

while (running) {
    al_wait_for_event(event_queue, &event);

    // Handle exit
    if (event.type == ALLEGRO_EVENT_DISPLAY_CLOSE)
        running = false;

    // Track mouse position and clicks
    if (event.type == ALLEGRO_EVENT_MOUSE_AXES) {
        mouse_x = event.mouse.x;
        mouse_y = event.mouse.y;
    }
    if (event.type == ALLEGRO_EVENT_MOUSE_BUTTON_DOWN)
        mouse_clicked = true;
```

https://doi.org/10.15215/remix/9781998944224.01

```
    else
        mouse_clicked = false;

    // Update buttons
    update_button(&start_button, mouse_x, mouse_y,
mouse_clicked);
    update_button(&options_button, mouse_x, mouse_y,
mouse_clicked);

    // Draw
    al_clear_to_color(al_map_rgb(0, 0, 0));
    draw_button(&start_button, font);
    draw_button(&options_button, font);
    al_flip_display();
}
```

7.6.4. Adding a Slider Component

Define a Slider Struct

Create a slider for volume control:

```
typedef struct {
    int x, y;          // Position
    int width;         // Length
    float value;       // 0.0 to 1.0
    bool dragging;     // Interaction state
} Slider;
```

Draw and Update the Slider

Implement slider logic:

```
void draw_slider(Slider *slider) {
    // Draw track
    al_draw_filled_rectangle(
        slider->x, slider->y–5,
        slider->x + slider->width, slider->y + 5,
        al_map_rgb(100, 100, 100)
    );
    // Draw thumb
    int thumb_x = slider->x + (slider->width *
slider->value);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_draw_filled_circle(thumb_x, slider->y, 10, al_map_
rgb(200, 200, 200));
}

void update_slider(Slider *slider, int mouse_x, int
mouse_y, bool mouse_down) {
    if (mouse_down) {
        // Check if mouse is near the thumb
        int thumb_x = slider->x + (slider->width *
slider->value);
        if (abs(mouse_x–thumb_x) < 15 &&
abs(mouse_y–slider->y) < 15)
            slider->dragging = true;
    } else {
        slider->dragging = false;
    }

    if (slider->dragging) {
        slider->value = (mouse_x–slider->x) / (float)
slider->width;
        slider->value = (slider->value < 0) ? 0 :
(slider->value > 1) ? 1 : slider->value;
    }
}
```

7.6.5. Building an Options Screen

Toggle Between Menu States

Add a state variable to switch between screens:

```
enum GameState { MENU, OPTIONS };
enum GameState current_state = MENU;
```

Add Sliders to Options Screen

Initialize a volume slider and handle its logic:

```
Slider volume_slider = {300, 200, 200, 0.5, false};

// In the game loop:
if (current_state == OPTIONS) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
    update_slider(&volume_slider, mouse_x, mouse_y,
mouse_clicked);
    draw_slider(&volume_slider);
}

// Transition to options screen when the "Options" button
is clicked
if (options_button.clicked)
    current_state = OPTIONS;
```

7.6.6. Key Techniques Used

1. **Modular Components**
   Use structs and functions to create reusable UI elements.
2. **Input Handling**
   Track mouse position and clicks to update UI states.
3. **State Management**
   Use enums to switch between different screens (e.g., menu, options).
4. **Visual Feedback**
   Change colours or positions to indicate hover/click states.

The following example demonstrates these concepts but is missing one element, the ability to go "back" to the main screen if you select options. That we'll leave for you, the student, to implement.

GUI EXAMPLE

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <allegro5/allegro_primitives.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#define SCREEN_WIDTH 800
#define SCREEN_HEIGHT 600

typedef struct {
    int x, y;
    int width, height;
```

https://doi.org/10.15215/remix/9781998944224.01

```
    char label[50];
    bool hovered;
    bool clicked;
} Button;

typedef struct {
    int x, y;
    int width;
    float value;
    bool dragging;
} Slider;

void draw_button(Button *button, ALLEGRO_FONT *font) {
    ALLEGRO_COLOR bg_color = button->hovered ? al_map_
rgb(100, 100, 255) : al_map_rgb(50, 50, 200);
    if (button->clicked) bg_color = al_map_rgb(200, 50,
50);

    al_draw_filled_rectangle(button->x, button->y,
button->x + button->width, button->y + button->height,
bg_color);
    al_draw_text(font, al_map_rgb(255, 255, 255),
button->x + button->width/2, button->y + button-
>height/2-12, ALLEGRO_ALIGN_CENTRE, button->label);
}

void update_button(Button *button, int mouse_x, int
mouse_y, bool mouse_clicked) {
    button->hovered = (mouse_x >= button->x && mouse_x
<= button->x + button->width && mouse_y >= button->y &&
mouse_y <= button->y + button->height);
    button->clicked = button->hovered && mouse_clicked;
}

void draw_slider(Slider *slider) {
    al_draw_filled_rectangle(slider->x, slider->y-5,
slider->x + slider->width, slider->y + 5, al_map_rgb(100,
100, 100));
    int thumb_x = slider->x + (slider->width *
slider->value);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_draw_filled_circle(thumb_x, slider->y, 10, al_map_
rgb(200, 200, 200));
}

void update_slider(Slider *slider, int mouse_x, int
mouse_y, bool mouse_down) {
    int thumb_x = slider->x + (slider->width *
slider->value);
    if (mouse_down) {
        if (abs(mouse_x-thumb_x) < 15 &&
abs(mouse_y-slider->y) < 15)
            slider->dragging = true;
    } else {
        slider->dragging = false;
    }

    if (slider->dragging) {
        slider->value = (mouse_x-slider->x) / (float)
slider->width;
        if (slider->value < 0) slider->value = 0;
        if (slider->value > 1) slider->value = 1;
    }
}

int main() {
    al_init();
    al_init_font_addon();
    al_init_ttf_addon();
    al_init_primitives_addon();
    al_install_keyboard();
    al_install_mouse();

    ALLEGRO_DISPLAY *display = al_create_display(SCREEN_
WIDTH, SCREEN_HEIGHT);
    ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();
    ALLEGRO_FONT *font = al_load_ttf_font("arial.ttf",
24, 0);
    ALLEGRO_TIMER *timer = al_create_timer(1.0 / 60);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_mouse_event_source());
    al_register_event_source(event_queue,
al_get_timer_event_source(timer));

    Button start_button = {300, 200, 200, 50, "Start
Game", false, false};
    Button options_button = {300, 300, 200, 50,
"Options", false, false};
    Slider volume_slider = {300, 200, 200, 0.5, false};

    enum GameState { MENU, OPTIONS };
    enum GameState current_state = MENU;

    int mouse_x = 0, mouse_y = 0;
    bool mouse_clicked = false;
    bool running = true;
    bool redraw = true;

    al_start_timer(timer);

    while (running) {
        ALLEGRO_EVENT event;
        al_wait_for_event(event_queue, &event);

        if (event.type == ALLEGRO_EVENT_DISPLAY_CLOSE)
            running = false;

        if (event.type == ALLEGRO_EVENT_MOUSE_AXES) {
            mouse_x = event.mouse.x;
            mouse_y = event.mouse.y;
        }

        if (event.type == ALLEGRO_EVENT_MOUSE_BUTTON_
DOWN) {
            mouse_clicked = true;
        } else if (event.type == ALLEGRO_EVENT_MOUSE_
BUTTON_UP) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
            mouse_clicked = false;
        }

        if (event.type == ALLEGRO_EVENT_TIMER) {
            if (current_state == MENU) {
                update_button(&start_button, mouse_x,
mouse_y, mouse_clicked);
                update_button(&options_button, mouse_x,
mouse_y, mouse_clicked);
                if (options_button.clicked)
                    current_state = OPTIONS;
            } else if (current_state == OPTIONS) {
                update_slider(&volume_slider, mouse_x,
mouse_y, mouse_clicked);
            }
            redraw = true;
        }

        if (redraw && al_is_event_queue_empty(event_
queue)) {
            redraw = false;
            al_clear_to_color(al_map_rgb(0, 0, 0));

            if (current_state == MENU) {
                draw_button(&start_button, font);
                draw_button(&options_button, font);
            } else if (current_state == OPTIONS) {
                draw_slider(&volume_slider);
                al_draw_textf(font, al_map_rgb(255, 255,
255), 300, 250, 0, "Volume: %.0f%%", volume_slider.value
* 100);
            }

            al_flip_display();
        }
    }

    al_destroy_font(font);
    al_destroy_display(display);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_destroy_event_queue(event_queue);
    al_destroy_timer(timer);
    return 0;
}
```

7.7. Advanced Audio

Allegro 5 provides robust audio capabilities for games, including sample mixing, spatial effects, and dynamic audio control. This section covers advanced techniques like 3D sound positioning, audio effects, streaming, and real-time adjustments to elevate your game's audio experience.

7.7.1. Initialization and Basic Setup

Install Audio Add-Ons

Initialize Allegro's audio system and codecs to load formats like WAV, OGG, and FLAC:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_audio.h>
#include <allegro5/allegro_acodec.h>

int main() {
    al_init();
    al_install_audio(); // Initialize audio subsystem
    al_init_acodec_addon(); // Enable codecs
    al_reserve_samples(16); // Reserve 16 sample
instances for mixing

    // Load a sound effect
    ALLEGRO_SAMPLE *laser_sound = al_load_sample("laser.
wav");
    if (!laser_sound) {
        fprintf(stderr, "Failed to load sound!\n");
        return-1;
    }

    // Play the sound
    al_play_sample(laser_sound, 1.0, 0.0, 1.0, ALLEGRO_
PLAYMODE_ONCE, NULL);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    // Cleanup
    al_destroy_sample(laser_sound);
    al_uninstall_audio();
    return 0;
}
```

7.7.2. Audio Mixing and Channels

Play Multiple Sounds Simultaneously

Use reserved sample instances to layer sounds (e.g., explosions and background music):

```
ALLEGRO_SAMPLE *explosion = al_load_sample("explosion.
wav");
ALLEGRO_SAMPLE *music = al_load_sample("music.ogg");

// Play music on loop
al_play_sample(music, 0.5, 0.0, 1.0, ALLEGRO_PLAYMODE_
LOOP, NULL);

// Play explosion once
al_play_sample(explosion, 1.0, 0.0, 1.0, ALLEGRO_
PLAYMODE_ONCE, NULL);
```

Adjust Volume and Panning Dynamically

Modify volume and pan during playback using al_set_sample_instance_gain and al_set_sample_instance_pan:

```
ALLEGRO_SAMPLE_INSTANCE *music_instance =
al_create_sample_instance(music);
al_attach_sample_instance_to_mixer(music_instance,
al_get_default_mixer());

// Set volume to 50% and pan slightly to the left
al_set_sample_instance_gain(music_instance, 0.5);
al_set_sample_instance_pan(music_instance,–0.3);
al_play_sample_instance(music_instance);
```

7.7.3. Spatial Audio (3D Sound)

Calculate Positional Audio

Adjust pan and volume based on the listener and sound source positions:

https://doi.org/10.15215/remix/9781998944224.01

```
typedef struct {
    float x, y; // Sound source position
} SoundSource;

void update_spatial_audio(SoundSource *source, float
listener_x, float listener_y) {
    float dx = source->x–listener_x;
    float dy = source->y–listener_y;
    float distance = sqrt(dx * dx + dy * dy);
    // Pan (-1.0 = left, 1.0 = right)
    float pan = (dx / 1000.0); // Adjust divisor for
sensitivity
    // Volume attenuation (max distance = 500 pixels)
    float volume = 1.0–(distance / 500.0);
    volume = (volume < 0) ? 0 : volume;

    al_set_sample_instance_pan(sound_instance, pan);
    al_set_sample_instance_gain(sound_instance, volume);
}
```

EXAMPLE USAGE

```
SoundSource monster = {400, 300};
float player_x = 200, player_y = 200;

// Update audio every frame
update_spatial_audio(&monster, player_x, player_y);
```

7.7.4. Audio Effects and Filters

Apply Low-Pass Filter (Simulate Underwater Effect)

Use a custom mixer and adjust frequency to mimic muffled sounds:

```
ALLEGRO_MIXER *lowpass_mixer = al_create_mixer(44100,
ALLEGRO_AUDIO_DEPTH_FLOAT32, ALLEGRO_CHANNEL_CONF_2);
ALLEGRO_MIXER *default_mixer = al_get_default_mixer();

// Attach the low-pass mixer to the default mixer
al_attach_mixer_to_mixer(lowpass_mixer, default_mixer);

// Lower the frequency cutoff (simulate underwater)
```

https://doi.org/10.15215/remix/9781998944224.01

```
al_set_mixer_frequency(lowpass_mixer, 2000); // Normal is
44100 Hz

// Play sounds through the low-pass mixer
al_attach_sample_instance_to_mixer(sound_instance,
lowpass_mixer);
```

7.7.5. Streaming Audio for Music

Load and Play Background Music

Stream large audio files without loading them entirely into memory:

```
ALLEGRO_AUDIO_STREAM *music_stream = al_load_audio_
stream("music.ogg", 4, 2048);
if (!music_stream) {
    fprintf(stderr, "Failed to load music stream!\n");
    return-1;
}

al_set_audio_stream_playmode(music_stream,
ALLEGRO_PLAYMODE_LOOP);
al_attach_audio_stream_to_mixer(music_stream,
al_get_default_mixer());
```

Fade-Out Music

Gradually reduce volume over time:

```
float volume = 1.0;
while (volume > 0) {
    volume-= 0.01;
    al_set_audio_stream_gain(music_stream, volume);
    al_rest(0.1);
}
al_stop_audio_stream(music_stream);
```

7.7.6. Real-Time Audio Control

Pause/Resume Audio

```
al_set_audio_stream_playing(music_stream, false); //
Pause
al_set_audio_stream_playing(music_stream, true);  //
Resume
```

https://doi.org/10.15215/remix/9781998944224.01

Dynamic Pitch Shifting

Alter playback speed for effects like slow motion:

```
al_set_sample_instance_speed(sound_instance, 0.5); //
Play at half speed
```

7.7.7. Best Practices in Using Advanced Audio

Advanced audio has both pros and cons. Please remember to follow the best practices:

Limit Active Samples: Use al_reserve_samples() to avoid audio dropouts.
Preload Frequently Used Sounds: Reduce latency during gameplay.
Use Spatial Audio Sparingly: Use only where needed (calculations can be CPU-intensive).
Test on Multiple Devices: Ensure compatibility with different sound cards.

This example program demonstrates the items covered previously:

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_audio.h>
#include <allegro5/allegro_acodec.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <math.h>
#include <stdio.h>

#define SCREEN_WIDTH 800
#define SCREEN_HEIGHT 600

typedef struct { float x, y; } SoundSource;

void update_spatial_audio(ALLEGRO_SAMPLE_INSTANCE
*instance,
                          SoundSource *source,
                          float listener_x, float
listener_y)
{
```

https://doi.org/10.15215/remix/9781998944224.01

```
    float dx = source->x-listener_x;
    float dy = source->y-listener_y;
    float distance = sqrtf(dx*dx + dy*dy);

    float pan    = dx / 1000.0f;              // crude
left/right balance
    float volume = 1.0f-(distance / 500.0f);
    if (volume < 0) volume = 0;

    al_set_sample_instance_pan (instance, pan);
    al_set_sample_instance_gain(instance, volume);
}

int main()
{
    al_init();
    al_install_keyboard();
    al_install_audio();
    al_init_acodec_addon();
    al_init_font_addon();
    al_init_ttf_addon();
    al_reserve_samples(16);

    ALLEGRO_DISPLAY *display = al_create_display(SCREEN_
WIDTH, SCREEN_HEIGHT);
    ALLEGRO_FONT    *font    = al_load_ttf_font("arial.
ttf", 20, 0);
    ALLEGRO_EVENT_QUEUE *queue =
al_create_event_queue();
    ALLEGRO_TIMER  *timer   = al_create_timer(1.0 / 60);

    al_register_event_source(queue,
al_get_keyboard_event_source());
    al_register_event_source(queue,
al_get_display_event_source(display));
    al_register_event_source(queue,
al_get_timer_event_source(timer));

    ALLEGRO_SAMPLE *laser      = al_load_sample("laser.
wav");
```

https://doi.org/10.15215/remix/9781998944224.01

```
    ALLEGRO_SAMPLE *explosion  = al_load_
sample("explosion.wav");
    ALLEGRO_SAMPLE *music      = al_load_sample("music.
ogg");

    ALLEGRO_SAMPLE_INSTANCE *music_instance =
al_create_sample_instance(music);
    al_attach_sample_instance_to_mixer(music_instance,
al_get_default_mixer());

    ALLEGRO_AUDIO_STREAM *music_stream =
        al_load_audio_stream("music.ogg", 4, 2048);
    if (music_stream) {
        al_set_audio_stream_playmode(music_stream,
ALLEGRO_PLAYMODE_LOOP);
        al_attach_audio_stream_to_mixer(music_stream,
al_get_default_mixer());
    }

    SoundSource monster = {400, 300};
    float player_x = 200, player_y = 200;

    bool running = true, fade_out = false;
    float volume = 1.0f;
    bool redraw = true;
    al_start_timer(timer);

    while (running) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(queue, &ev);

        if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE)
            running = false;

        if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
            switch (ev.keyboard.keycode) {
    case ALLEGRO_KEY_0:  // Stop/Reset everything started
by 1-9
    fade_out = false;                // cancel pending
fade
```

https://doi.org/10.15215/remix/9781998944224.01

```
    volume   = 1.0f;                  // reset our fade
volume
       al_stop_samples();               // stop all one-
shot/looped samples
    if (music_stream) {
         al_set_audio_stream_gain(music_stream, 1.0f);
al_set_audio_stream_playing(music_stream, false);  //
stop/pause stream
               }
if (music_instance) {
    al_set_sample_instance_speed(music_instance, 1.0f);
al_stop_sample_instance(music_instance);           //
stop the instance
                        }
                    break;
                case ALLEGRO_KEY_1:
                    al_play_sample(laser, 1.0, 0.0, 1.0,
                                   ALLEGRO_PLAYMODE_ONCE,
NULL);
                    break;
                case ALLEGRO_KEY_2:
                    al_play_sample(music, 0.5, 0.0, 1.0,
                                   ALLEGRO_PLAYMODE_LOOP,
NULL);
                    al_play_sample(explosion, 1.0, 0.0,
1.0,
                                   ALLEGRO_PLAYMODE_ONCE,
NULL);
                    break;
                case ALLEGRO_KEY_3:
                    al_set_sample_instance_gain(music_
instance, 0.5);
                    al_set_sample_instance_pan
(music_instance,–0.3);
                    al_play_sample_instance(music_
instance);
                    break;
                case ALLEGRO_KEY_4:
                    update_spatial_audio(music_instance,
&monster,
```

https://doi.org/10.15215/remix/9781998944224.01

```
                                          player_x,
player_y);
                    break;
                case ALLEGRO_KEY_5:
                    if (music_instance) {
                        ALLEGRO_MIXER *lowpass =
                            al_create_mixer(44100,
                                            ALLEGRO_
AUDIO_DEPTH_FLOAT32,
                                            ALLEGRO_
CHANNEL_CONF_2);
                        al_attach_mixer_to_mixer(lowpass,
                                                 al_get_
default_mixer());
                        al_set_mixer_frequency(lowpass,
2000);
                        al_attach_sample_instance_to_
mixer(music_instance,
                                                       lowpass);
                        al_play_sample_instance(music_
instance);
                    }
                    break;
                case ALLEGRO_KEY_6:
                    if (music_stream)
                        al_set_audio_stream_
playing(music_stream, true);
                    break;
                case ALLEGRO_KEY_7:
                    fade_out = true;
                    break;
                case ALLEGRO_KEY_8:
                    if (music_stream)
                        al_set_audio_stream_
playing(music_stream, false); // Pause
                    break;
                case ALLEGRO_KEY_9:
                    al_set_sample_instance_speed(music_
instance, 0.5); // slow
                    break;
```

https://doi.org/10.15215/remix/9781998944224.01

```
                case ALLEGRO_KEY_ESCAPE:
                    running = false;
                    break;
            }
        }

        /*——Fade-out handler————————————————————*/
        if (fade_out && music_stream && volume >
0.0f) {
            volume—= 0.01f;
            al_set_audio_stream_gain(music_stream,
volume);
            if (volume <= 0.0f) {
                fade_out = false;
                al_set_audio_stream_playing(music_stream,
false);  // ← FIX
            }
        }

        if (ev.type == ALLEGRO_EVENT_TIMER)
            redraw = true;

        if (redraw && al_is_event_queue_empty(queue)) {
            redraw = false;
            al_clear_to_color(al_map_rgb(0, 0, 0));

            const char *lines[] = {
                "Press keys 1-9 to play audio examples:",
                "1. Play Sound Effect (Laser)",
                "2. Mix Sound Effects (Explosion +
Looping Music)",
                "3. Volume/Pan Example",
                "4. Spatial Audio Based on Position",
                "5. Simulate Underwater with Low-pass",
                "6. Stream Audio (Music)",
                "7. Fade Out Music",
                "8. Pause Music Stream",
                "9. Play Slow Motion Effect",
                "0. Stop / Reset (cancel fade, stop all
audio)",
```

https://doi.org/10.15215/remix/9781998944224.01

```
                "ESC to Quit"
            };
           al_clear_to_color(al_map_rgb(0,0,0));

            int nlines = (int)(sizeof(lines) /
sizeof(lines[0]));
            for (int i = 0; i < nlines; ++i) {
            al_draw_text(font, al_map_rgb(255,255,255),
20, 20 + i*30, 0, lines[i]);
            }

            al_flip_display();
        }
    }

    /*--Cleanup----------------------------*/
    al_destroy_sample(laser);
    al_destroy_sample(explosion);
    al_destroy_sample(music);
    al_destroy_sample_instance(music_instance);
    if (music_stream)
al_destroy_audio_stream(music_stream);
    al_destroy_font(font);
    al_destroy_timer(timer);
    al_destroy_event_queue(queue);
    al_destroy_display(display);
    al_uninstall_audio();
    return 0;
}
```

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Multithreading

1. **Implement a Dual-Threaded Game Loop:** Create a program where one thread handles rendering and input, while a second thread updates game logic (e.g., moving a sprite). Use mutexes to synchronize shared variables like sprite position.

https://doi.org/10.15215/remix/9781998944224.01

2. **Race Condition Demonstration:** Write a program where two threads increment a shared counter *without* mutexes. Explain why the final value is inconsistent. Fix it using ALLEGRO_MUTEX.
3. **Thread Pool for Physics:** Simulate 100 bouncing balls. Use a thread pool to parallelize collision detection and position updates. Compare performance with a single-threaded approach.
4. **Deadlock Scenario:** Create a deadlock using two mutexes and threads. Document how to resolve it.

Custom Shaders

5. **Dynamic Colour Shader:** Modify the fragment shader to pulse a sprite's colour based on a sine wave controlled by a time uniform.
6. **Greyscale Transition:** Implement a shader that gradually converts a texture to greyscale when the player presses a key.
7. **Screen Distortion Effect:** Create a fragment shader that warps the screen using a noise function or sine wave.
8. **Health-Based Shader:** Change a character's appearance (e.g., adding a red tint) based on health value passed from C++ to the shader.

Networking

9. **Multiplayer Pong:** Build a TCP-based Pong game where the server manages ball physics and two clients control paddles.
10. **Chat System with UDP:** Create a UDP-based chat program where messages are broadcasted to all clients. Compare reliability with TCP.
11. **Lag Compensation:** Simulate network latency in a multiplayer game and implement interpolation to smooth player movement.
12. **Lobby System:** Extend the server to track player usernames and readiness status before starting a game.

AI Programming

13. **A Pathfinding Visualization:** Render a grid-based map with obstacles. Let the player click start/end points, and display the A path.
14. **Finite State Machine for NPCs:** Create an enemy that patrols, chases the player if within range, and returns to patrol when out of range.
15. **Flocking Behavior:** Implement Boid algorithms (separation, alignment, cohesion) for a group of creatures.
16. **Behavior Tree for Boss AI:** Design a boss that cycles through attack patterns (e.g., charge, shoot, retreat) using a decision tree.

https://doi.org/10.15215/remix/9781998944224.01

Optimization

17. **Texture Atlas Generator:** Write a tool to combine multiple sprites into a single texture atlas and update rendering code to use it.
18. **Collision Detection Grid:** Implement spatial partitioning to limit collision checks to nearby objects. Benchmark performance gains.
19. **Memory Leak Detector:** Use Valgrind or Allegro's debug tools to identify and fix leaks in a provided buggy code sample.
20. **Dynamic Resolution Scaling:** Adjust rendering resolution dynamically based on frame time to maintain 60 fps.

GUI Systems

21. **Pause Menu:** Create a pause menu with buttons to resume, adjust volume, or quit. Use custom button components.
22. **Inventory System:** Design a drag-and-drop inventory grid with item tool tips.
23. **Settings Screen:** Add sliders for volume, resolution, and key bindings. Save settings to a file.
24. **Dialogue Boxes:** Implement modal pop-ups for confirmation (e.g., "Are you sure you want to quit?").

Advanced Audio

25. **3D Audio Demo:** Simulate a sound source moving around the player. Adjust pan and volume based on relative position.
26. **Dynamic Music Mixer:** Layer background music with intensity-based tracks (e.g., calm vs. combat music).
27. **Echo Effect:** Apply a delay filter to audio samples using Allegro's mixer API.
28. **Voice Chat System:** Stream microphone input between clients over UDP.

Comprehensive Projects

29. **Tower Defense Game**
    - Use multithreading for pathfinding and wave updates.
    - Implement GUI for tower placement and upgrades.
    - Add shaders for projectile effects.
    - Include networked co-op mode.
30. **RPG Engine**
    - Create a state machine for quest-driven NPCs.
    - Optimize rendering with batch drawing.
    - Add spatial audio for footsteps and ambient sounds.
    - Design a skill tree using custom GUI components.

https://doi.org/10.15215/remix/9781998944224.01

This page intentionally left blank

8

Game Development Projects

In previous chapters, we have acquired the fundamental knowledge and skills for game programming. In this chapter, we will design and implement some real games with Allegro in C or C++.

8.1. Action Game: Space Shooter

Among action games, space shooter games are a popular subgenre that involves controlling a spaceship and battling against waves of enemies, often in outer space settings. These games typically feature fast-paced gameplay, challenging levels, and a variety of power-ups and upgrades. Designing and implementing a space shooter game with Allegro in C++ is an exciting project.

8.1.1. Game Design

Game Concept

Genre: A 2D space shooter game where the player controls a spaceship, shoots enemies, and avoids obstacles.

Objective: The player controls a spaceship, shoots at incoming enemies, and tries to survive as long as possible.

Core Components

Player Ship: The player-controlled spaceship

Attributes: Position (x, y), speed, sprite

Movement: Controlled via keyboard input (left, right, up, down)

Shooting: Fires bullets when the player presses a key

https://doi.org/10.15215/remix/9781998944224.01

Enemies: Enemy ships that move toward the player and can be destroyed
Attributes: Position (x, y), speed, sprite
Movement: Move downward toward the player
Spawning: New enemies spawn at the top of the screen at regular intervals

Bullets: Projectiles fired by the player and enemies
Attributes: Position (x, y), speed, sprite
Movement: Move upward (player bullets) or downward (enemy bullets)

Game State: Tracks the current state of the game (playing, game over, etc.)
Attributes: Current state (playing, game over), score, lives

Key Techniques and Features to Have

A 2D space shooter combines responsive controls, real-time entity management, and fast feedback into a tightly coupled gameplay loop. The following features define a clear and functional baseline for an engaging space shooter implemented with Allegro.

1. **Player Spaceship Control and Movement**
   The player ship should support smooth four-directional movement using keyboard input and remain clamped within screen boundaries. Input handling must be responsive to allow precise dodging and positioning.
2. **Player Shooting Mechanics**
   The player should be able to fire bullets upward using a cooldown-limited shooting system. Bullets must spawn relative to the ship's sprite and travel at a fixed speed, with active bullets stored in a dynamic container.
3. **Enemy Ship Behavior and Spawning**
   Enemy ships should spawn from the top of the screen, move downward, and vary in speed or pattern to increase difficulty. Spawning may occur at regular intervals or in structured waves.
4. **Collision Detection**
   Collisions between player bullets and enemies, as well as between enemies and the player, should be detected using AABB checks. Entities involved in collisions must be removed or deactivated immediately.

https://doi.org/10.15215/remix/9781998944224.01

5. **Game State, Scoring, and Survival Rules**
   The game should track core states such as playing and game over. Destroying enemies should increase the score, and optional lives or health systems may extend gameplay longevity.
6. **Enemy Variations and Difficulty Scaling**
   Difficulty should increase over time by introducing faster enemies, higher health values, or new movement and attack patterns. Scaling should remain gradual to preserve playability.
7. **Power-Ups and Collectibles (Optional)**
   Power-ups such as rapid fire, shields, spread shots, or speed boosts may be added. These should spawn occasionally and apply temporary effects to the player.
8. **Visual and Gameplay Feedback**
   Clear sprites for the player, enemies, and bullets should reinforce readability. Visual or audio feedback—such as flashes, explosions, or sounds—should confirm hits and successful actions.
9. **Main Game Loop and Allegro Integration**
   An Allegro timer (e.g., 60 fps) should drive all updates, including movement, spawning, collision checks, and rendering. Game logic and rendering should be separated into distinct update and render phases.
10. **Asset Management and Cleanup**
    All visual assets should be loaded during Initialization and released cleanly on shutdown. Allegro resources such as bitmaps, timers, event queues, and displays must be properly destroyed to ensure stability.

8.1.2. Implementation

Set Up Your Development Environment

Before writing any code, it's important to ensure that your tools and libraries are properly installed and configured. The following checklist summarizes the required software and setup steps needed to build and run the project.

Create Project Structure

A clean and organized folder layout helps keep the project maintainable as it grows. The structure here shows how the game's source files and assets should be arranged:

https://doi.org/10.15215/remix/9781998944224.01

```
SpaceShooter/
├── CMakeLists.txt
├── src/
│   ├── main.cpp
│   ├── spaceship.cpp
│   ├── spaceship.h
│   ├── enemy.cpp
│   ├── enemy.h
│   ├── bullet.cpp
│   ├── bullet.h
│   ├── game.cpp
│   └── game.h
└── assets/
    ├── spaceship.png
    ├── enemy.png
    ├── bullet.png
    └── background.png
```

Write CMakeLists.txt for Ninja Build

To compile the project efficiently, you'll use CMake with the Ninja generator. The following configuration file sets up the build process and links the Allegro libraries required by the game:

```
cmake_minimum_required(VERSION 3.10)
project(SpaceShooter)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)
add_executable(SpaceShooter src/main.cpp src/spaceship.
cpp src/enemy.cpp src/bullet.cpp src/game.cpp)
find_package(PkgConfig REQUIRED)
pkg_check_modules(ALLEGRO5 REQUIRED allegro-5
allegro_main-5 allegro_image-5 allegro_font-5
allegro_ttf-5 allegro_primitives-5)
include_directories(${ALLEGRO5_INCLUDE_DIRS})
target_link_libraries(SpaceShooter ${ALLEGRO5_LIBRARIES})
```

Key Operations in the Game

A space shooter game runs through a continuous cycle of updates and rendering steps that keep gameplay responsive and engaging. These operations define how the player interacts with the game world, how enemies behave, and how the

https://doi.org/10.15215/remix/9781998944224.01

system maintains real-time action. The following list summarizes the essential operations executed during each frame of the game loop:

1. **Process Player Input**
   - Read keyboard state to determine movement and shooting actions.
   - Apply movement constraints to keep the ship within screen boundaries.
2. **Update Player, Enemy, and Bullet States**
   - Move the player ship based on input.
   - Advance bullets upward or downward depending on their type.
   - Update enemy positions as they drift or spawn from the top of the screen.
3. **Spawn New Enemies**
   - Introduce new enemy ships at timed intervals or in waves.
   - Randomize spawn positions or speeds to increase difficulty.
4. **Perform Collision Detection**
   - Check for collisions between bullets and enemies.
   - Detect collisions between enemies and the player ship.
   - Remove or deactivate any entities involved in a collision.
5. **Manage Game State and Scoring**
   - Track whether the game is running, paused, or over.
   - Update the score when enemies are destroyed.
   - Handle player defeat conditions.
6. **Clean Up Inactive Entities**
   - Remove bullets that leave the screen.
   - Remove enemies that are destroyed or move off-screen.
7. **Render the Frame**
   - Draw the background, player ship, enemies, bullets, and UI elements.
   - Flip the display to present the updated frame to the player.

Structure of the Game Program

A 2D space shooter is composed of several interconnected systems that manage player control, enemy behavior, real-time action, and visual rendering. Organizing these systems clearly helps ensure responsive gameplay, predictable behavior, and ease of extension as new features such as scoring, power-ups, or additional enemy types are added. The following outline describes the typical structure of a space shooter game program:

https://doi.org/10.15215/remix/9781998944224.01

1. **Initialization**
   - Initialize Allegro subsystems (display, keyboard input, image, primitives, fonts).
   - Create the main display window, event queue, and timer.
   - Install input devices and prepare the game clock (e.g., 60 fps).
2. **Resource Loading**
   - Load sprite assets for the player ship, enemies, bullets, and background.
   - Load fonts and any sound assets used for feedback.
   - Validate that all required resources are available before entering gameplay.
3. **Entity Creation**
   - Create the player ship with its initial position and movement parameters.
   - Initialize enemy containers and enemy spawn timers.
   - Prepare data structures for bullets and other dynamic entities.
4. **Input Handling**
   - Capture keyboard events or keyboard state each frame.
   - Map input to player movement and shooting actions.
   - Handle global inputs such as pause, restart, or window-close events.
5. **Update Functions**
   - Update the player's position based on input and apply screen boundaries.
   - Advance bullets and enemy positions each frame.
   - Spawn new enemies according to timers or wave rules.
   - Detect and resolve collisions between bullets, enemies, and the player.
   - Update score, lives, and game state as required.
6. **Rendering Functions**
   - Clear the screen and draw the background.
   - Render the player ship, enemies, and active bullets.
   - Draw UI elements such as score, lives, or game-over messages.
   - Flip the display to present the rendered frame.
7. **Game Loop**
   - Wait for events from the event queue.
   - On timer events, update the game state and render the scene.
   - Maintain a consistent update rate to ensure smooth gameplay.
8. **Shutdown**
   - Destroy timers, event queues, fonts, and the display.

https://doi.org/10.15215/remix/9781998944224.01

- Release loaded bitmaps and other resources.
- Exit the program cleanly.

Coding the Game

The implementation is divided into several source files, each responsible for a specific part of the game's functionality. The following sections walk through the purpose and contents of each file.

1. MAIN.CPP

```
#include "game.h"
#include <allegro5/allegro_native_dialog.h>

int main() {
    Game game;
    if (!game.init()) {
        al_show_native_message_box(nullptr, "Init
failed", "Init returned false",
                                   "Pausing 5 seconds so
you can read prior errors.", nullptr, 0);
        al_rest(5.0);
        return 1;
    }

    al_show_native_message_box(nullptr, "Info",
"Startup",
                               "Game initialized
successfully!", nullptr, 0);

    game.run();

    al_show_native_message_box(nullptr, "Info",
"Shutdown",
                               "Game exiting. Pausing
2 seconds.", nullptr, 0);
    al_rest(2.0);

    game.shutdown();
    return 0;
}
```

https://doi.org/10.15215/remix/9781998944224.01

2. GAME.H

```
#ifndef GAME_H
#define GAME_H

#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <vector>
#include <string>
#include "spaceship.h"
#include "enemy.h"
#include "bullet.h"

class Game {
public:
    Game();
    bool init();
    void run();
    void shutdown();
private:
    void update();
    void render();
    bool check_collision(ALLEGRO_BITMAP* bmp1, float ax,
float ay,
                         ALLEGRO_BITMAP* bmp2, float bx,
float by)

    // core Allegro
    ALLEGRO_DISPLAY* display;
    ALLEGRO_EVENT_QUEUE* event_queue;
    ALLEGRO_TIMER* timer;
    bool running;
    ALLEGRO_BITMAP* background;

    // gameplay
    Spaceship player;
    std::vector<Enemy> enemies;
    std::vector<Bullet> bullets;
```

https://doi.org/10.15215/remix/9781998944224.01

```
    // input/game timing
    double last_shot_time;
    const double SHOT_COOLDOWN = 0.20; // seconds

    // asset paths
    std::string bullet_path;
};

#endif
```

3. GAME.CPP

```
#include "game.h"

#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <allegro5/allegro_native_dialog.h>

#include <algorithm>
#include <string>
#include <cstdio>

// Build a full path like "<exe_dir>/<subdir>/<filename>"
static std::string exe_path_join(const char* subdir,
const char* filename) {
    ALLEGRO_PATH* p =
al_get_standard_path(ALLEGRO_EXENAME_PATH);
    if (subdir && *subdir) {
        al_append_path_component(p, subdir);
    }
    al_set_path_filename(p, filename);
    std::string out = al_path_cstr(p,
ALLEGRO_NATIVE_PATH_SEP);
    al_destroy_path(p);
    return out;
}

Game::Game()
    : display(nullptr),
```

https://doi.org/10.15215/remix/9781998944224.01

```
      event_queue(nullptr),
      timer(nullptr),
      running(false),
      background(nullptr),
      player(400.0f, 520.0f),
      last_shot_time(0.0) {}

bool Game::init() {
    std::fprintf(stderr, "[INIT] Allegro starting\n");
    if (!al_init()) {
        al_show_native_message_box(nullptr, "Error",
"Allegro Init", "Failed to initialize Allegro!", nullptr,
0);
        return false;
    }
    if (!al_install_keyboard()) {
        al_show_native_message_box(nullptr, "Error",
"Keyboard", "Failed to install keyboard!", nullptr, 0);
        return false;
    }
    if (!al_init_image_addon()) {
        al_show_native_message_box(nullptr, "Error",
"Image Addon", "Failed to initialize image addon!",
nullptr, 0);
        return false;
    }
    if (!al_init_primitives_addon()) {
        al_show_native_message_box(nullptr, "Error",
"Primitives Addon", "Failed to initialize primitives
addon!", nullptr, 0);
        return false;
    }

    display = al_create_display(800, 600);
    if (!display) {
        al_show_native_message_box(nullptr, "Error",
"Display Creation", "Failed to create display!", nullptr,
0);
        return false;
    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    //——Asset loads relative to the EXE directory——
    const std::string bgPath    = exe_path_join("assets",
"background.png");
    const std::string shipPath  = exe_path_join("assets",
"ship.png");
    const std::string enemyPath = exe_path_join("assets",
"enemy.png");
    bullet_path                 = exe_path_join("assets",
"bullet.png");

    background = al_load_bitmap(bgPath.c_str());
    if (!background) {
        al_show_native_message_box(display, "Asset
error", "Load failed",
                                   ("Failed to load " +
bgPath).c_str(), nullptr, 0);
        return false;
    }

    if (!player.load(shipPath.c_str())) {
        al_show_native_message_box(display, "Asset
error", "Load failed",
                                   ("Failed to load " +
shipPath).c_str(), nullptr, 0);
        return false;
    }

    // Starter enemies across the top
    enemies.clear();
    for (int i = 0; i < 5; ++i) {
        float x = 60.0f + i * 140.0f;
        float y = 20.0f;
        enemies.emplace_back(x, y, enemyPath.c_str());
    }

    // Event queue + timer
    event_queue = al_create_event_queue();
    if (!event_queue) {
        al_show_native_message_box(nullptr, "Error", "Event
Queue", "Failed to create event queue!", nullptr, 0);
```

https://doi.org/10.15215/remix/9781998944224.01

```
        return false;
    }

    timer = al_create_timer(1.0 / 60.0);
    if (!timer) {
        al_show_native_message_box(nullptr, "Error",
"Timer", "Failed to create timer!", nullptr, 0);
        return false;
    }

    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_timer_event_source(timer));
    al_register_event_source(event_queue,
al_get_keyboard_event_source());

    return true;
}

void Game::run() {
    std::fprintf(stderr, "[RUN] start\n");
    running = true;
    al_start_timer(timer);

    // Initial clear so we don't see junk before the
first flip
    al_clear_to_color(al_map_rgb(0, 0, 0));
    al_flip_display();

    while (running) {
        ALLEGRO_EVENT ev;
        if (al_wait_for_event_timed(event_queue, &ev,
0.25)) {
            switch (ev.type) {
                case ALLEGRO_EVENT_TIMER:
                    std::fprintf(stderr, "[EV] TIMER\n");
                    update();
                    render();
                    break;
```

https://doi.org/10.15215/remix/9781998944224.01

```
                case ALLEGRO_EVENT_DISPLAY_CLOSE:
                    std::fprintf(stderr, "[EV]
DISPLAY_CLOSE\n");
                    running = false;
                    break;

                case ALLEGRO_EVENT_KEY_DOWN:
                    std::fprintf(stderr, "[EV] KEY_DOWN:
%d\n", ev.keyboard.keycode);
                    if (ev.keyboard.keycode == ALLEGRO_
KEY_ESCAPE) {
                        running = false;
                    }
                    break;

                default:
                    std::fprintf(stderr, "[EV] other:
%u\n", ev.type);
                    break;
            }
        } else {
            // No event within timeout—render anyway to
keep window responsive
            render();
        }
    }
    std::fprintf(stderr, "[RUN] end\n");
}

//———methods———

void Game::update() {
    player.update();

    ALLEGRO_KEYBOARD_STATE ks;
    al_get_keyboard_state(&ks);
    double now = al_get_time();
    if (al_key_down(&ks, ALLEGRO_KEY_SPACE) &&
(now—last_shot_time) >= SHOT_COOLDOWN) {
        player.shoot(bullets, bullet_path.c_str());
```

https://doi.org/10.15215/remix/9781998944224.01

```
        last_shot_time = now;
    }

    for (auto& b : bullets) b.update();
    bullets.erase(std::remove_if(bullets.begin(),
bullets.end(),
                                 [](const Bullet& b){
return !b.is_active(); }),
                  bullets.end());

    for (auto& e : enemies) e.update();

    for (auto& e : enemies) {
        if (!e.is_active() || !e.get_bitmap()) continue;
        for (auto& b : bullets) {
            if (!b.is_active() || !b.get_bitmap())
continue;
            if (check_collision(b.get_bitmap(),
b.get_x(), b.get_y(),
                                e.get_bitmap(),
e.get_x(), e.get_y())) {
                b.set_active(false);
                e.set_active(false);
                break;
            }
        }
    }

    if (player.get_bitmap()) {
        for (auto& e : enemies) {
            if (!e.is_active() || !e.get_bitmap())
continue;
            if (check_collision(player.get_bitmap(),
player.get_x(), player.get_y(),
                                e.get_bitmap(),
e.get_x(), e.get_y())) {
                std::fprintf(stderr, "[GAME] Player
collided with enemy—ending\n");
                running = false;
                break;
```

https://doi.org/10.15215/remix/9781998944224.01

```
            }
        }
    }
}

void Game::render() {
    if (background) al_draw_bitmap(background, 0, 0, 0);
    player.render();
    for (auto& e : enemies) e.render();
    for (auto& b : bullets) b.render();
    al_flip_display();
}

bool Game::check_collision(ALLEGRO_BITMAP* a, float ax,
float ay,
                           ALLEGRO_BITMAP* b, float bx,
float by) {
    if (!a || !b) return false;

    const float aw =
static_cast<float>(al_get_bitmap_width(a));
    const float ah =
static_cast<float>(al_get_bitmap_height(a));
    const float bw =
static_cast<float>(al_get_bitmap_width(b));
    const float bh =
static_cast<float>(al_get_bitmap_height(b));

    const float a_right  = ax + aw;
    const float a_bottom = ay + ah;
    const float b_right  = bx + bw;
    const float b_bottom = by + bh;

    return (ax < b_right) && (a_right > bx) && (ay < b_
bottom) && (a_bottom > by);
}

void Game::shutdown() {
    if (timer) { al_stop_timer(timer); al_destroy_
timer(timer); timer = nullptr; }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    if (event_queue) { al_destroy_event_queue(event_
queue); event_queue = nullptr; }

    if (background) { al_destroy_bitmap(background);
background = nullptr; }
    bullets.clear();
    enemies.clear();

    if (display) { al_destroy_display(display); display =
nullptr; }

    // Optional: shutdown addons (safe to omit; Allegro
cleans up on exit)
    // al_shutdown_primitives_addon();
    // al_shutdown_image_addon();
}
```

4. SPACESHIP.H

```
#ifndef SPACESHIP_H
#define SPACESHIP_H

#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include "bullet.h"
#include <vector>

class Spaceship {
public:
    Spaceship(float x, float y);
    ~Spaceship();

    // Non-copyable (owning raw pointer)
    Spaceship(const Spaceship&) = delete;
    Spaceship& operator=(const Spaceship&) = delete;

    // Movable
    Spaceship(Spaceship&& other) noexcept;
    Spaceship& operator=(Spaceship&& other) noexcept;

    bool load(const char* image_path);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    void update();
    void render();
    void shoot(std::vector<Bullet>& bullets, const char*
bullet_image_path);

    float get_x() const { return x; }
    float get_y() const { return y; }
    ALLEGRO_BITMAP* get_bitmap() const { return image; }

private:
    float x, y;
    ALLEGRO_BITMAP* image;
};

#endif
```

5. SPACESHIP.CPP

```
#include "spaceship.h"
#include <allegro5/allegro.h>

Spaceship::Spaceship(float x, float y) : x(x), y(y),
image(nullptr) {}
Spaceship::~Spaceship() { if (image) { al_destroy_
bitmap(image); image = nullptr; } }

// Move ctor
Spaceship::Spaceship(Spaceship&& other) noexcept
    : x(other.x), y(other.y), image(other.image)
{
    other.image = nullptr;
}

// Move assign
Spaceship& Spaceship::operator=(Spaceship&& other)
noexcept {
    if (this != &other) {
        if (image) al_destroy_bitmap(image);
        x = other.x; y = other.y; image = other.image;
        other.image = nullptr;
    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    return *this;
}

bool Spaceship::load(const char* image_path) {
    image = al_load_bitmap(image_path);
    return image != nullptr;
}

void Spaceship::update() {
    ALLEGRO_KEYBOARD_STATE state;
    al_get_keyboard_state(&state);
    if (al_key_down(&state, ALLEGRO_KEY_LEFT)) x-= 5;
    if (al_key_down(&state, ALLEGRO_KEY_RIGHT)) x += 5;
    if (al_key_down(&state, ALLEGRO_KEY_UP)) y-= 5;
    if (al_key_down(&state, ALLEGRO_KEY_DOWN)) y += 5;

    if (x < 0) x = 0;
    if (y < 0) y = 0;
    if (image) {
        int w = al_get_bitmap_width(image);
        int h = al_get_bitmap_height(image);
        if (x > 800-w) x = 800-w;
        if (y > 600-h) y = 600-h;
    }
}

void Spaceship::render() {
    if (image) al_draw_bitmap(image, x, y, 0);
}

void Spaceship::shoot(std::vector<Bullet>& bullets, const
char* bullet_image_path) {
    if (!image) return;
    float bx = x + al_get_bitmap_width(image) /
2.0f-2.0f;
    float by = y-10.0f;
    if (bullet_image_path && *bullet_image_path) {
        bullets.emplace_back(bx, by, 0.0f,-8.0f,
bullet_image_path);
    } else {
```

https://doi.org/10.15215/remix/9781998944224.01

```
        bullets.emplace_back(bx, by, 0.0f,–8.0f);
    }
}
```

6. ENEMY.H

```
#ifndef ENEMY_H
#define ENEMY_H

#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>

class Enemy {
public:
    Enemy(float x, float y, const char* image_path);
    void update();
    void render();
    void set_active(bool active) { this->active =
active; }
    bool is_active() const { return active; }
    float get_x() const { return x; }
    float get_y() const { return y; }
    ALLEGRO_BITMAP* get_bitmap() const { return
image; }
private:
    float x, y;
    bool active;
    ALLEGRO_BITMAP* image;
};

#endif
```

7. ENEMY.CPP

```
#include "enemy.h"
#include <cstdlib>

Enemy::Enemy(float x, float y, const char* image_path)
    : x(x), y(y), active(true), image(nullptr)
{
    image = al_load_bitmap(image_path);
    if (!image) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
        active = false;
    }
}

Enemy::~Enemy() {
    if (image) { al_destroy_bitmap(image); image =
nullptr; }
}

// Move ctor
Enemy::Enemy(Enemy&& other) noexcept
    : x(other.x), y(other.y), active(other.active),
image(other.image)
{
    other.image = nullptr;
    other.active = false;
}

// Move assign
Enemy& Enemy::operator=(Enemy&& other) noexcept {
    if (this != &other) {
        if (image) al_destroy_bitmap(image);
        x = other.x; y = other.y; active = other.active;
image = other.image;
        other.image = nullptr;
        other.active = false;
    }
    return *this;
}

void Enemy::update() {
    if (!active || !image) return;
    // simple downward drift
    y += 2.0f;
    if (y > 600) {
        int h = al_get_bitmap_height(image);
        y =-h;
        int w = al_get_bitmap_width(image);
        x = static_cast<float>(rand() % (800-w));
    }
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
void Enemy::render() {
    if (active && image) {
        al_draw_bitmap(image, x, y, 0);
    }
}
```

8. BULLET.H

```
#ifndef BULLET_H
#define BULLET_H

#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>

class Bullet {
public:
    Bullet(float x, float y, float dx, float dy);
    Bullet(float x, float y, float dx, float dy, const
char* ima
    ~Bullet();

    // Non-copyable (owning raw pointer)
    Bullet(const Bullet&) = delete;
    Bullet& operator=(const Bullet&) = delete;

    // Movable
    Bullet(Bullet&& other) noexcept;
    Bullet& operator=(Bullet&& other) noexcept;

    void update();
    void render();
    bool is_active() const { return active; }
    void set_active(bool a) { active = a; }
    float get_x() const { return x; }
    float get_y() const { return y; }
    ALLEGRO_BITMAP* get_bitmap() const { return image; }

private:
    void load_image_from(const char* image_path);

    float x, y;
    float dx, dy;
```

https://doi.org/10.15215/remix/9781998944224.01

```
    bool active;
    ALLEGRO_BITMAP* image;
};

#endif
```

9. BULLET.CPP

```
#include "bullet.h"

Bullet::Bullet(float x, float y, float dx, float dy)
    : x(x), y(y), dx(dx), dy(dy), active(true),
image(nullptr)
{
    load_image_from("assets/bullet.png");
}

Bullet::Bullet(float x, float y, float dx, float dy,
const char* image_path)
    : x(x), y(y), dx(dx), dy(dy), active(true),
image(nullptr)
{
    load_image_from(image_path);
}

void Bullet::load_image_from(const char* image_path) {
    image = al_load_bitmap(image_path);
    if (!image) active = false;
}

Bullet::~Bullet() {
    if (image) { al_destroy_bitmap(image); image =
nullptr; }
}

// Move ctor
Bullet::Bullet(Bullet&& other) noexcept
    : x(other.x), y(other.y), dx(other.dx), dy(other.dy),
      active(other.active), image(other.image)
{
    other.image = nullptr;
```

https://doi.org/10.15215/remix/9781998944224.01

```
    other.active = false;
}

// Move assign
Bullet& Bullet::operator=(Bullet&& other) noexcept {
    if (this != &other) {
        if (image) al_destroy_bitmap(image);
        x = other.x; y = other.y; dx = other.dx; dy =
other.dy;
        active = other.active; image = other.image;
        other.image = nullptr; other.active = false;
    }
    return *this;
}

void Bullet::update() {
    if (!active || !image) { active = false; return; }
    y += dy; x += dx;
    int w = al_get_bitmap_width(image);
    int h = al_get_bitmap_height(image);
    if (y <–h || y > 600 || x <–w || x > 800) active =
false;
}

void Bullet::render() {
    if (active && image) al_draw_bitmap(image, x, y, 0);
}
```

Build the Project

After building the executable, you can launch the game directly from your development environment. The following steps explain how to run the program and verify that everything is working correctly:

- Open the Command Palette (⇧⌘P or Ctrl + Shift + P) and run CMake: Configure.
- Select the appropriate kit (e.g., GCC) and specify the generator as Ninja.
- Run CMake to build.

https://doi.org/10.15215/remix/9781998944224.01

Run Your Game

Use the CMake: Run command or configure a launch configuration in the launch.json file. This is a basic framework to get you started.

TASKS FOR COMPLETION AND IMPROVEMENT

With the core game implemented, there are many opportunities to expand its features and polish the experience. The following tasks suggest meaningful enhancements you can add to deepen gameplay and improve presentation.

1. **Scoring and Health System:** Display hit counts or health bars clearly on the screen and refine how damage is calculated.
2. **Improved AI:** Enhance enemy tank behavior with smarter movement, targeting, and obstacle avoidance.
3. **Weapons and Shooting Mechanics:** Introduce multiple weapon types or adjust firing rate, speed, and damage for variety.
4. **Battlefield and Levels:** Create different maps or obstacle layouts and increase difficulty through more complex environments.
5. **Sound Effects:** Add sound effects for firing, impacts, and victory events to improve immersion.

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Exercises

1. **Action Game Characteristics**
   Identify the defining characteristics of action games demonstrated by the space shooter. Why are responsiveness and fast feedback more critical here than narrative depth or complex AI?
2. **Entity Lifecycle Management**
   Explain the lifecycle of a bullet from creation to removal. Why is it important to deactivate or erase bullets once they leave the screen or collide with an enemy?
3. **Collision Detection Basics**
   Describe how axis-aligned bounding box (AABB) collision checks work. Why are they well suited to a 2D space shooter with rectangular sprites?

https://doi.org/10.15215/remix/9781998944224.01

4. **Game State Awareness**
   Explain how the game distinguishes between different states such as "playing" and "game over." Why is explicit game-state tracking important even in simple action games?

Homework Questions

1. **Input Handling Strategies**
   Compare polling the keyboard state with event-based input handling. What are the advantages of using continuous keyboard state checks for movement in an action game?
2. **Difficulty and Player Skill**
   Discuss how enemy spawn frequency, speed, and quantity influence difficulty. How can designers balance challenge so the game remains fun rather than frustrating?
3. **Performance Considerations**
   Why should inactive enemies and bullets be removed from their containers regularly? What problems might arise if these objects accumulate over time?
4. **Visual Feedback and Player Perception**
   Explain how immediate visual feedback (such as explosions or flashing sprites) improves player perception of responsiveness and fairness during gameplay.

Programming Projects

1. **Scoring System (Core Project)**
   Add a visible scoring system that increases the player's score whenever an enemy is destroyed. Display the score prominently on the screen during gameplay.
2. **Lives and Health System**
   Extend the game so the player has multiple lives or a health bar. Deduct lives or health when colliding with enemies and trigger a game-over condition when it reaches zero.
3. **Power-Ups and Collectibles**
   Implement power-ups that occasionally drop from destroyed enemies, such as rapid fire, temporary shields, or spread shots. Ensure power-ups activate and expire correctly.
4. **Multiple Enemy Types**
   Introduce additional enemy types with different movement speeds, behaviors, or hit points. Use simple variation rules to increase gameplay depth.

https://doi.org/10.15215/remix/9781998944224.01

5. **Level Progression (Advanced)**
   Create multiple levels or stages. Each level should feature increased difficulty through faster enemies, denser waves, or new enemy behaviors.
6. **Sound Effects and Audio Feedback (Advanced)**
   Add sound effects for shooting, enemy destruction, and player damage using Allegro's audio add-ons. Balance volume levels to avoid overwhelming the player.

Capstone Project

Design and implement a fully polished space shooter game featuring multiple levels, score tracking, lives, power-ups, sound effects, and clear game-over and restart mechanics. Include a short design reflection describing how input handling, collision logic, and difficulty progression work together to create a compelling action-game experience.

8.2. Platform Game: Tank War

Tank war games are a popular genre that focuses on tank warfare, offering players the chance to control powerful armored vehicles in various combat scenarios.

Designing and implementing a tank war game with Allegro in C++ can be a fun and rewarding project. Here's a step-by-step guide to help you get started.

8.2.1. Game Design

Game Concept

Genre: A simple 2D tank war game where players control tanks, navigate a battlefield, and try to destroy each other.

Objective: Players control tanks with smooth movement, rotation, and shooting mechanics. The game should implement real-time combat using projectiles, collision detection, and health tracking. Design a bounded battlefield with obstacles that influence movement, cover, and tactics and provide clear win and loss conditions through a hit-based scoring system. Reinforce core game programming concepts, including the game loop, event-driven input, timing, collision handling, and basic AI behaviors.

Core Components

Player Tank: The primary tank controlled by the player

Attributes: Position (x, y), rotation angle, movement speed, sprite, health (or hit count)

https://doi.org/10.15215/remix/9781998944224.01

Movement: Controlled via keyboard input; the tank rotates left or right and moves forward or backward based on its facing direction

Shooting: Fires projectiles in the direction the tank is facing, subject to firing rate and projectile speed limits

Enemy Tank: An opposing tank controlled by scripted logic or simple AI behavior

Attributes: Position (x, y), rotation angle, movement speed, sprite, health (or hit count)

Movement: Moves based on AI rules such as patrolling, chasing the player, or repositioning for tactical advantage

Shooting: Fires projectiles toward the player or along its facing direction, using the same projectile system as the player tank

Bullets (Projectiles): Projectiles fired by both player and enemy tanks

Attributes: Position (x, y), velocity (dx, dy), speed

Movement: Travel in straight lines in the firing direction until they hit a tank, collide with an obstacle, or leave the playfield

Collision: Deactivate on impact with tanks or environmental obstacles

Battlefield and Obstacles: The bounded play area where tank combat takes place

Attributes: Screen bounds, obstacle positions, obstacle sizes

Function: Restricts tank movement to the playfield and provides cover or tactical chokepoints that influence combat and positioning

Collision System: Manages interactions between tanks, bullets, and obstacles

Detection Method: Axis-aligned bounding box (AABB) collision checks

Function: Prevents tanks from passing through obstacles, registers bullet hits, and enforces valid movement

Game State and Scoring: Tracks the overall progress and outcome of a round

Attributes: Current round state (playing, round won, paused), hit counters or health values, win condition thresholds

Win Condition: A round ends when a tank reaches a defined hit limit or health depletion, triggering a win or loss state

Key Techniques and Features to Have

The following features describe the core mechanics and engineering practices required to implement the tank war game. Each item corresponds directly to systems implemented in the Game, Tank, and Bullet classes.

https://doi.org/10.15215/remix/9781998944224.01

1. **Game Loop and Timing (60 fps)**
   The game should use an event-driven loop driven by an ALLEGRO_TIMER at 60 fps. All gameplay logic should execute on timer events to ensure deterministic updates and stable rendering.
2. **Input Handling and Control Mapping**
   Keyboard input should be mapped cleanly to tank movement, rotation, and firing, with separate control schemes for each tank. Input events should set intent flags that are applied during the update step, and global inputs (quit, close) should halt the loop cleanly.
3. **Rotation, Movement, and Orientation**
   Tank orientation should be represented using radians and normalized to prevent drift. Movement should project forward and backward motion from the tank's facing angle, accounting for sprite-specific forward offsets.
4. **World Constraints and Clamping**
   Tanks must be constrained to the playfield bounds. Invalid movement through obstacles should be prevented by reverting overlapping moves detected during updates.
5. **Obstacles: Placement, Blocking, and Rendering**
   Obstacles should be generated procedurally without overlapping tank spawn areas or each other. They must be stored centrally for collision checks and rendered clearly using filled shapes with outlines.
6. **Projectiles (Bullets)**
   Bullets should spawn just outside the firing tank's bounding box, travel at constant speed, and be removed when off-screen or when colliding with obstacles. Visual representations may be simplified without affecting collision logic.
7. **Collision Detection and Responses**
   Collision checks should use simple point-in-rectangle or AABB tests. Bullet–tank and bullet–obstacle collisions should immediately remove the bullet, while tank–obstacle collisions prevent movement. More advanced collision models can be deferred.
8. **Health, Scoring, and Round Flow**
   Each tank should track hits taken, with scoring debounced per update to prevent multiple counts. A round ends when a tank reaches the hit limit, displays a winner message, pauses briefly, and then resets the battlefield.
9. **UI and Player Feedback**
   The UI should display hit counters and a winner banner using readable fonts and colour coding. Feedback should be immediate and easy to interpret during active play and round transitions.

https://doi.org/10.15215/remix/9781998944224.01

10. **Resources and Asset Pipeline**
    Tank textures and other assets should be loaded once and destroyed properly to avoid leaks. Bitmap settings should favor smooth scaling, and build tools should automatically place assets alongside the executable.
11. **Randomness and Reproducibility**
    Random elements, such as obstacle placement, should be seeded for variation, with optional fixed seeds available for debugging or reproducible testing.
12. **Code Organization and Extensibility**
    Responsibilities should be clearly divided between Game, Tank, and Bullet classes. The design should include hooks for future extensions such as AI steering, external map loading, new weapons, and audio feedback.
13. **Performance and Stability**
    Per-frame operations should scale linearly with active bullets and obstacles. Allocations in hot paths should be avoided, and all Allegro resources should be validated and released cleanly on shutdown.

8.2.2. Implementation

Before implementing this and other games in this chapter, ensure you have Visual Studio Code, GCC, CMake, Ninja, and Allegro 5 installed and configured as described in chapter 1 and mentioned in 8.1.2.

Key Operations in the Game

A tank war game relies on a continuous sequence of operations that update the battlefield, process player actions, and maintain the flow of combat. These operations occur every frame and ensure that tanks move realistically, bullets behave predictably, and the game responds immediately to player input. The following list summarizes the essential runtime operations that drive the tank war gameplay experience:

1. **Process Player Input**
   - Read Up/Down key events to determine tank movement, rotation, and firing.
   - Update intent flags (e.g., forward, backward) and apply them during the update step.
2. **Update Tank Movement and Orientation**
   - Move tanks forward or backward based on their current facing angle.
   - Apply rotation increments to adjust tank direction.

https://doi.org/10.15215/remix/9781998944224.01

- Normalize angles to avoid drift and maintain consistent orientation.

3. **Apply World Constraints and Obstacle Collisions**
   - Clamp tank positions to remain within the 800 × 600 playfield.
   - Prevent tanks from passing through obstacles by reverting invalid moves.
4. **Update Bullets**
   - Advance bullets along their velocity vectors.
   - Remove bullets that leave the screen or collide with obstacles.
5. **Detect Collisions Between Bullets and Tanks**
   - Check whether any bullet intersects a tank's axis-aligned bounding box.
   - Mark hits and remove bullets that successfully strike a tank.
6. **Update Scoring and Round Flow**
   - Increment hit counters for each tank when struck.
   - Trigger win conditions when a tank accumulates five hits.
   - Display a winner banner and pause the game before resetting the round.
7. **Render the Frame**
   - Draw obstacles, tanks, bullets, and UI elements.
   - Display hit counters and winner messages when appropriate.
   - Flip the display to present the updated frame.

Structure of the Game Program

The tank war game is organized into several interconnected modules that work together to implement gameplay, rendering, and event handling. Each part of the program has a clear responsibility, making the code easier to maintain and extend. The following outline summarizes the major structural components of the game.

1. **Initialization**
   - Initialize Allegro subsystems (keyboard, image, primitives, font, TTF).
   - Create the display, event queue, and timer.
   - Load tank sprites and prepare the battlefield.
2. **Resource Loading**
   - Load tank textures, bullet graphics (if any), and fonts.
   - Set bitmap flags for smooth scaling and rendering.
   - Generate obstacles procedurally to populate the map.
3. **Entity Creation**
   - Instantiate Tank objects for both players.

https://doi.org/10.15215/remix/9781998944224.01

- Initialize bullet containers and obstacle lists.
- Set initial positions and angles for both tanks.

4. **Input Handling**
   - Capture keyboard events to set movement and rotation flags.
   - Trigger bullet firing when Ctrl keys are pressed.
   - Handle Esc key press and display-close events to exit the game.
5. **Update Functions**
   - Move tanks and apply collision checks.
   - Update bullet positions and remove invalid bullets.
   - Detect hits and update scoring logic.
   - Manage round transitions and pause states.
6. **Rendering Functions**
   - Draw obstacles, tanks, bullets, and UI text.
   - Display winner banners and hit counters.
   - Clear and flip the display each frame.
7. **Game Loop**
   - Wait for events from the event queue.
   - On timer events, call update() and render().
   - Maintain a consistent 60 fps update rate.
8. **Shutdown**
   - Destroy fonts, timers, event queues, and the display.
   - Release any loaded resources.
   - Cleanly exit the program.

Coding the Game

As shown in the program structure, code files for the game project can be organized in a tree, and the main.cpp is the root of the tree. The following is the outline of the main.cpp file for this game.

MAIN.CPP

```
#include "game.h"

int main() {
    Game game;
    if (game.init()
        game.run();
    }
    game.shutdown()
    return 0;
}
```

https://doi.org/10.15215/remix/9781998944224.01

GAME.H

```
#ifndef GAME_H
#define GAME_H

#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <vector>
#include <string>
#include <allegro5/allegro_font.h>

#include "tank.h"
#include "bullet.h"
#include "common.h"

struct Obstacle { float x, y, w, h; };

class Game {
public:
    Game();
    bool init();
    void run();
    void shutdown();

private:
    void generateObstacles();
    bool tankOverlapsAnyObstacle(const Tank& t) const;
    bool bulletHitsAnyObstacle(float bx, float by)
const;
    void renderObstacles() const;
    bool rectsOverlap(float ax,float ay,float aw,float
ah, float bx,float by,float bw,flo

private:
    void update();
    void render();
    void fireBulletFrom(const Tank& t);
    bool check_collision(const Tank& tank, const Bullet&
bullet);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    void clampTankToScreen(Tank& t);
    void resetRound();

    ALLEGRO_DISPLAY* display;
    ALLEGRO_EVENT_QUEUE* event_queue;
    ALLEGRO_TIMER* timer;
    bool running;

    Tank tank1; // right side (WASD + Right Ctrl), PNG
faces LEFT (offset PI)
    Tank tank2; // left  side (Arrows + Left Ctrl), PNG
faces RIGHT (offset 0)

    bool t1_forward=false, t1_backward=false;
    bool t2_forward=false, t2_backward=false;

    float moveSpeed = 3.0f;
    const float rotStep = PI / 4.0f;
    const float bulletSpeed = 8.0f;

    int score1 = 0; // hits on tank1
    int score2 = 0; // hits on tank2

    std::vector<Bullet> bullets;
    std::vector<Obstacle> obstacles;

    //——Round win & pause UI——
    static constexpr int WIN_SCORE = 5; // first tank to
TAKE 5 hits loses
    ALLEGRO_FONT* uiFont = nullptr;
    std::string   winText;
    int           winTextFrames = 0;   // frames to show
banner (300 == 5s @60FPS)
    bool          roundPaused   = false;

};

#endif // GAME_H
```

https://doi.org/10.15215/remix/9781998944224.01

GAME.CPP

```
#include<cmath>
#include "game.h"
#include "tank.h"
#include "bullet.h"
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <cstdio>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>

static constexpr int SCREEN_W = 800;
static constexpr int SCREEN_H = 600;

Game::Game()
    : display(nullptr), event_queue(nullptr),
timer(nullptr), running(true) {}

bool Game::init() {
    if (!al_init()) return false;
    if (!al_install_keyboard()) return false;
    if (!al_init_image_addon()) return false;
    if (!al_init_primitives_addon()) return false;
    if (!al_init_font_addon()) return false;
    al_init_ttf_addon();

    if (!uiFont) uiFont = al_create_builtin_font();

    al_set_new_bitmap_flags(ALLEGRO_VIDEO_BITMAP |
ALLEGRO_MIN_LINEAR | ALLEGRO_MAG_LINEAR);
    display = al_create_display(SCREEN_W, SCREEN_H);
    if (!display) return false;
    al_set_blender(ALLEGRO_ADD, ALLEGRO_ALPHA,
ALLEGRO_INVERSE_ALPHA);

    event_queue = al_create_event_queue();
    timer = al_create_timer(1.0 / 60.0);
    if (!event_queue || !timer) return false;
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_register_event_source(event_queue,
al_get_keyboard_event_source());
    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_timer_event_source(timer));

    // Tank2: left side, PNG faces RIGHT at 0°–> offset
0.0
    tank2 = Tank(600, 400, "assets/tank2.png", 0.0f);
    // Tank1: right side, PNG faces LEFT at 0°–> offset
PI
    tank1 = Tank(100, 100, "assets/tank1.png", PI);
    generateObstacles();
al_start_timer(timer);
    return true;
}

void Game::shutdown() {
    if (uiFont) { al_destroy_font(uiFont); uiFont =
nullptr; }
    if (timer) { al_destroy_timer(timer); timer =
nullptr; }
    if (event_queue) { al_destroy_event_queue(event_
queue); event_queue = nullptr; }
    if (display) { al_destroy_display(display); display =
nullptr; }
}

void Game::resetRound() {
    bullets.clear();
    tank2.setPosition(100, 100);
    tank2.setAngle(0);
    tank1.setPosition(600, 400);
    tank1.setAngle(0);
    generateObstacles();
}

void Game::run() {
    while (running) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
        ALLEGRO_EVENT ev;
        al_wait_for_event(event_queue, &ev);

        if (ev.type == ALLEGRO_EVENT_TIMER) {
            update();
            render();
        } else if (ev.type == ALLEGRO_EVENT_DISPLAY_
CLOSE) {
            running = false;
        } else if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
            switch (ev.keyboard.keycode) {
                // Tank1 (right side)—WASD + Right Ctrl
                case ALLEGRO_KEY_W:     t1_forward  =
true; break;
                case ALLEGRO_KEY_S:     t1_backward =
true; break;
                case ALLEGRO_KEY_A:     tank1.rotateBy(-
rotStep); break;
                case ALLEGRO_KEY_D:     tank1.
rotateBy(+rotStep); break;
                case ALLEGRO_KEY_LCTRL:
fireBulletFrom(tank1); break;

                // Tank2 (left side)—Arrows + Left Ctrl
                case ALLEGRO_KEY_UP:    t2_forward  =
true; break;
                case ALLEGRO_KEY_DOWN:  t2_backward =
true; break;
                case ALLEGRO_KEY_LEFT:  tank2.rotateBy(-
rotStep); break;
                case ALLEGRO_KEY_RIGHT: tank2.
rotateBy(+rotStep); break;
                case ALLEGRO_KEY_RCTRL:
fireBulletFrom(tank2); break;

                case ALLEGRO_KEY_ESCAPE: running = false;
break;
            }
        } else if (ev.type == ALLEGRO_EVENT_KEY_UP) {
            switch (ev.keyboard.keycode) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
                // Tank1
                case ALLEGRO_KEY_W: t1_forward  = false;
break;
                case ALLEGRO_KEY_S: t1_backward = false;
break;

                // Tank2
                case ALLEGRO_KEY_UP:   t2_forward  =
false; break;
                case ALLEGRO_KEY_DOWN: t2_backward =
false; break;
            }
        }
    }
}

void Game::update() {
    if (!roundPaused) {
        // Movement (blocked by obstacles)
        if (t1_forward)   { tank1.
moveForward(+moveSpeed); if
(tankOverlapsAnyObstacle(tank1)) { tank1.
        if (t1_backward)  { tank1.moveForward(-
moveSpeed); if (tankOverlapsAnyObstacle(tank1)) { tank1.
        if (t2_forward)   { tank2.
moveForward(+moveSpeed); if
(tankOverlapsAnyObstacle(tank2)) { tank2.
        if (t2_backward)  { tank2.moveForward(-
moveSpeed); if (tankOverlapsAnyObstacle(tank2)) { tank2.

        clampTankToScreen(tank1);
        clampTankToScreen(tank2);

        // Bullets
        for (auto& b : bullets) b.update();
        bullets.erase(std::remove_if(bullets.begin(),
bullets.end(),
            [this](const Bullet& b){
                if (b.getX() <–10 || b.getX() > SCREEN_W
+ 10 || b.getY() <–10 || b.getY() > SCREEN_H
```

https://doi.org/10.15215/remix/9781998944224.01

```
                    return true;
                if (bulletHitsAnyObstacle(b.getX(),
b.getY()))
                    return true;
                return false;
            }),
            bullets.end());

        // Tank hit detection—accumulate hits on each tank
        /*
        bool hit1 = false, hit2 = false;
        for (auto& b : bullets) {
            if (check_collision(tank1, b)) hit1 = true;
            if (check_collision(tank2, b)) hit2 = true;
        }
        if (hit1) score1++;   // score1 == hits on tank1
        if (hit2) score2++;   // score2 == hits on tank2
*/
bool hit1 = false, hit2 = false;
std::vector<size_t> toErase;
toErase.reserve(bullets.size());

for (size_t i = 0; i < bullets.size(); ++i) {
    const Bullet& b = bullets[i];
    bool consumed = false;

    if (check_collision(tank1, b)) {  // bullet overlaps
tank1
        hit1 = true;
        consumed = true;
    }
    if (check_collision(tank2, b)) {  // bullet overlaps
tank2
        hit2 = true;
        consumed = true;
    }
    if (consumed) {
        toErase.push_back(i);
    }
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
// remove bullets that hit any tank (back-to-front to
keep indices valid)
for (size_t k = 0; k < toErase.size(); ++k) {
    bullets.erase(bullets.begin() + (toErase[k]-k));
}

// Debounced scoring: add at most one hit per tank this
update
if (hit1) ++score1;  // score1 = hits on tank1
if (hit2) ++score2;  // score2 = hits on tank2

        // Round end: first tank to take 5 hits loses
        if (score1 >= WIN_SCORE || score2 >= WIN_SCORE) {
            if (score1 >= WIN_SCORE) {
                winText = "Tank 2 wins the round!";
            } else {
                winText = "Tank 1 wins the round!";
            }
            winTextFrames = 180; // 5 seconds at 60 FPS
            roundPaused = true;
        }
    } else {
        // Pause countdown
        if (winTextFrames > 0) {
           -winTextFrames;
        } else {
            // Reset battlefield and hit counters for
next round
            resetRound();
            score1 = 0;
            score2 = 0;
            roundPaused = false;
        }
    }
}

void Game::render() {
    al_clear_to_color(al_map_rgb(20, 30, 40));
    renderObstacles();
    tank1.render();
```

https://doi.org/10.15215/remix/9781998944224.01

```
    tank2.render();
    for (auto& b : bullets) b.render();

    // Draw winner banner if active
    if (winTextFrames > 0 && uiFont) {
        ALLEGRO_COLOR red = al_map_rgb(255, 0, 0);
        float tw = (float)al_get_text_width(uiFont,
winText.c_str());
        float x  = (SCREEN_W-tw) * 0.5f;
        float y  = 8.0f;
        al_draw_text(uiFont, red, x, y, 0,
winText.c_str());
    }

    ALLEGRO_FONT* f = al_create_builtin_font();
    if (f) {
        al_draw_textf(f, al_map_rgb(255,255,255), 10, 10,
0, "Hits on Tank1: %d", score1);
        al_draw_textf(f, al_map_rgb(255,255,255), 10, 30,
0, "Hits on Tank2: %d", score2);
        al_destroy_font(f);
    }
    al_flip_display();
}

void Game::fireBulletFrom(const Tank& t) {
    if (roundPaused) return;
    const float cx = t.getX() + t.getWidth()  * 0.5f;
    const float cy = t.getY() + t.getHeight() * 0.5f;
/*
    const float theta = t.getAngle() + SPRITE_FORWARD_
OFFSET + t.getForwardOffset();
    const float muzzleOffset = std::max(t.getWidth(),
t.getHeight()) * 0.5f + 4.0f;
*/
const float theta = t.getAngle() + SPRITE_FORWARD_OFFSET
+ t.getForwardOffset();
// Use half-diagonal so the spawn is guaranteed outside
the AABB for any angle
const float halfDiag = 0.5f * std::sqrt(
```

https://doi.org/10.15215/remix/9781998944224.01

```
    static_cast<float>(t.getWidth()) * t.getWidth() +
    static_cast<float>(t.getHeight()) * t.getHeight()
);
const float muzzleOffset = halfDiag + 6.0f; // small
safety margin

    const float sx = cx + std::cos(theta) * muzzleOffset;
    const float sy = cy + std::sin(theta) * muzzleOffset;

    const float dx = std::cos(theta) * bulletSpeed;
    const float dy = std::sin(theta) * bulletSpeed;

    bullets.emplace_back(sx, sy, dx, dy);
}

bool Game::check_collision(const Tank& tank, const
Bullet& bullet) {
    const float bx = bullet.getX();
    const float by = bullet.getY();
    const float tx = tank.getX();
    const float ty = tank.getY();
    const float tw = tank.getWidth();
    const float th = tank.getHeight();
    return (bx >= tx && bx <= tx + tw && by >= ty && by
<= ty + th);
}

void Game::clampTankToScreen(Tank& t) {
    float nx = t.getX();
    float ny = t.getY();
    if (nx < 0) nx = 0;
    if (ny < 0) ny = 0;
    if (nx > SCREEN_W-t.getWidth())  nx = SCREEN_W-t.
getWidth();
    if (ny > SCREEN_H-t.getHeight()) ny = SCREEN_H-t.
getHeight();
    t.setPosition(nx, ny);
}
bool Game::rectsOverlap(float ax,float ay,float aw,float
ah, float bx,float by,float bw,float bh) const
```

https://doi.org/10.15215/remix/9781998944224.01

```
    return (ax < bx + bw) && (ax + aw > bx) && (ay < by +
bh) && (ay + ah > by);
}

bool Game::tankOverlapsAnyObstacle(const Tank& t) const {
    for (const auto& ob : obstacles) {
        if (rectsOverlap(t.getX(), t.getY(),
t.getWidth(), t.getHeight(), ob.x, ob.y, ob.w, ob.h)) {
            return true;
        }
    }
    return false;
}

bool Game::bulletHitsAnyObstacle(float bx, float by)
const {
    for (const auto& ob : obstacles) {
        if (bx >= ob.x && bx <= ob.x + ob.w && by >= ob.y
&& by <= ob.y + ob.h) {
            return true;
        }
    }
    return false;
}

void Game::renderObstacles() const {
    for (const auto& ob : obstacles) {
        al_draw_filled_rectangle(ob.x, ob.y, ob.x + ob.w,
ob.y + ob.h, al_map_rgb(110,110,110));
        al_draw_rectangle(ob.x, ob.y, ob.x + ob.w, ob.y +
ob.h, al_map_rgb(40,40,40), 1.0f);
    }
}

void Game::generateObstacles() {
    obstacles.clear();
    std::srand((unsigned)std::time(nullptr));
    const int count = 10;
    const float w = tank1.getWidth()  > 0 ? tank1.
getWidth()  : 64.0f;
```

https://doi.org/10.15215/remix/9781998944224.01

```
    const float h = tank1.getHeight() > 0 ? tank1.
getHeight() : 64.0f;

    auto overlapsTankStarts = [&](float x, float y) {
        const float pad = 12.0f;
        if (rectsOverlap(x, y, w, h, tank1.getX()-pad,
tank1.getY()-pad, tank1.getWidth()+2*pad, tank1.
        if (rectsOverlap(x, y, w, h, tank2.getX()-pad,
tank2.getY()-pad, tank2.getWidth()+2*pad, tank2.
        return false;
    };

    for (int i=0; i<count; ++i) {
        int guard = 0;
        while (true) {
            ++guard;
            float x = (float)(std::rand() % (SCREEN_W-
(int)w-20) + 10);
            float y = (float)(std::rand() % (SCREEN_H-
(int)h-20) + 10);
            if (overlapsTankStarts(x, y)) { if (guard <
2000) continue; }
            bool ok = true;
            for (const auto& ob : obstacles) {
                if (rectsOverlap(x, y, w, h, ob.x, ob.y,
ob.w, ob.h)) { ok = false; break; }
            }
            if (!ok) { if (guard < 2000) continue; }
            obstacles.push_back(Obstacle{x,y,w,h});
            break;
        }
    }
}
```

TANK.H

```
#ifndef TANK_H
#define TANK_H

#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
```

https://doi.org/10.15215/remix/9781998944224.01

```
class Tank {
public:
    Tank();
    Tank(float x, float y, const char* image_path, float
forwardOffsetRadians = 0.0f);
    ~Tank();

    Tank(const Tank&) = delete;
    Tank& operator=(const Tank&) = delete;
    Tank(Tank&& other) noexcept;
    Tank& operator=(Tank&& other) noexcept;

    void setPosition(float nx, float ny) { x = nx; y = ny; }
    void moveBy(float dx, float dy) { x += dx; y += dy; }
    void rotateBy(float radians);
    void setAngle(float radians);

    void update() {} // placeholder
    void render();

    void moveForward(float amount);

    float getX() const { return x; }
    float getY() const { return y; }
    int   getWidth()  const { return width; }
    int   getHeight() const { return height; }
    float getAngle()  const { return angle; }
    float getForwardOffset() const { return forwardOffset; }

private:
    void normalizeAngle();

    float x = 0.0f, y = 0.0f;
    ALLEGRO_BITMAP* image = nullptr;
    int width = 0, height = 0;
    float angle = 0.0f;
    float forwardOffset = 0.0f;
};

#endif // TANK_H
```

https://doi.org/10.15215/remix/9781998944224.01

TANK.CPP

```
#include "tank.h"
#include "common.h"
#include <cstdio>
#include <allegro5/allegro_primitives.h>

Tank::Tank() : x(0.0f), y(0.0f), image(nullptr),
width(0), height(0), angle(0.0f), forwardOffset(0.0f) {}

Tank::Tank(float x, float y, const char* image_path,
float forwardOffsetRadians)
    : x(x), y(y), image(nullptr), width(0), height(0),
angle(0.0f), forwardOffset(forwardOffsetRadians) {
    image = al_load_bitmap(image_path);
    if (!image) {
        std::fprintf(stderr, "[Tank] Failed to load:
%s\n", image_path);
    } else {
        width = al_get_bitmap_width(image);
        height = al_get_bitmap_height(image);
    }
}

Tank::~Tank() {
    if (image) {
        al_destroy_bitmap(image);
        image = nullptr;
    }
}

Tank::Tank(Tank&& other) noexcept {
    x = other.x; y = other.y;
    image = other.image; other.image = nullptr;
    width = other.width; height = other.height;
    angle = other.angle;
    forwardOffset = other.forwardOffset;
}

Tank& Tank::operator=(Tank&& other) noexcept {
```

https://doi.org/10.15215/remix/9781998944224.01

```
    if (this != &other) {
        if (image) al_destroy_bitmap(image);
        x = other.x; y = other.y;
        image = other.image; other.image = nullptr;
        width = other.width; height = other.height;
        angle = other.angle;
        forwardOffset = other.forwardOffset;
    }
    return *this;
}

void Tank::rotateBy(float radians) {
    angle += radians;
    normalizeAngle();
}

void Tank::setAngle(float radians) {
    angle = radians;
    normalizeAngle();
}

void Tank::normalizeAngle() {
    if (angle >= PI) {
        angle = std::fmod(angle + PI, 2.0f * PI)–PI;
    } else if (angle <–PI) {
        angle = std::fmod(angle–PI, 2.0f * PI) + PI;
    }
}

void Tank::render() {
    if (!image) {
        al_draw_filled_rectangle(x, y, x + 40, y + 40,
al_map_rgb(255, 0, 0));
        return;
    }
    const float cx = width * 0.5f;
    const float cy = height * 0.5f;
    al_draw_rotated_bitmap(image, cx, cy, x + cx, y + cy,
angle, 0);
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
void Tank::moveForward(float amount) {
    const float theta = angle + SPRITE_FORWARD_OFFSET +
forwardOffset;
    x += std::cos(theta) * amount;
    y += std::sin(theta) * amount;
}
```

BULLET.H

```
#ifndef BULLET_H
#define BULLET_H

#include <allegro5/allegro.h>
#include <allegro5/allegro_primitives.h>

class Bullet {
public:
    Bullet(float x, float y, float dx, float dy);
    void update();
    void render();
    float getX() const { return x; }
    float getY() const { return y; }
private:
    float x, y, dx, dy;
};

#endif // BULLET_H
```

BULLET.CPP

```
#include "bullet.h"

Bullet::Bullet(float x, float y, float dx, float dy)
    : x(x), y(y), dx(dx), dy(dy) {}

void Bullet::update() {
    x += dx;
    y += dy;
}

void Bullet::render() {
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_draw_filled_circle(x, y, 2, al_map_rgb(255, 255,
255));
}
```

COMMON.H

```
#ifndef COMMON_H
#define COMMON_H

#include <cmath>

static constexpr float PI = 3.14159265358979323846f;
static constexpr float SPRITE_FORWARD_OFFSET = 0.0f;

#endif // COMMON_H
```

TASKS FOR COMPLETION AND IMPROVEMENT

1. **Features:** Add more features.
2. **AI:** Improve the AI.
3. **Graphics:** Enhance the graphics.

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Exercises

1. **Tank Movement and Orientation**
   Explain how representing tank orientation using radians enables smooth rotation and directional movement. Why is it important to normalize angles over time?
2. **Event-Driven Game Loop**
   Describe how the Allegro timer drives the game loop at 60 frames per second. Why should game logic updates be executed only in response to timer events?
3. **Collision Handling Strategy**
   Review the collision logic used for tanks, bullets, and obstacles. Why are axis-aligned bounding boxes (AABB) an appropriate choice for this game?

https://doi.org/10.15215/remix/9781998944224.01

4. **Round-Based Game Flow**
   Explain how hit counters, win conditions, pause states, and round resets work together to define the flow of a single round of gameplay.

Homework Questions

1. **Control Scheme Design**
   Discuss the advantages and challenges of having two players share a keyboard. How do control mappings affect player comfort and fairness?
2. **Projectile Management**
   Why is it important to spawn bullets outside the tank's bounding box? Describe what could go wrong if bullets begin inside their source tank.
3. **Obstacle-Driven Gameplay**
   Explain how obstacles influence player tactics and movement decisions. How do obstacles increase strategic depth in an otherwise simple combat arena?
4. **Simple AI Limitations**
   The enemy tank AI uses basic decision logic. What types of player behaviors might this AI fail to respond to effectively, and why?

Programming Projects

1. **Enhanced Enemy AI (Core Project)**
   Improve enemy tank behavior by implementing simple steering logic, such as seeking the player, maintaining distance, or avoiding obstacles dynamically.
2. **Multiple Weapon Types**
   Extend the shooting system to support additional weapon types, such as spread shots, slow but powerful shells, or ricocheting bullets. Balance their speed and damage appropriately.
3. **Map Loading from Data Files**
   Replace procedural obstacle generation with map loading from an external file format (e.g., JSON or CSV). Allow different battlefield layouts for variety.
4. **Health Bars and Visual Feedback**
   Add visible health bars or damage indicators to tanks. Provide visual feedback when a tank is hit, such as a brief flash or screen shake.
5. **Sound Effects Integration (Advanced)**
   Add sound effects for firing bullets, tank explosions, and round victories using Allegro's audio subsystem.

https://doi.org/10.15215/remix/9781998944224.01

6. **Single-Player Mode vs. AI (Advanced)**
   Modify the game to allow a single-player mode where the second tank is fully AI-controlled. Adjust difficulty by tuning AI speed, firing rate, or accuracy.

Capstone Project

Design and implement a polished game featuring multiple maps, improved AI behaviors, enhanced graphics, and audio feedback. Include configurable match rules (such as time-based rounds or score limits) and provide a menu system for mode selection. Submit a short design overview explaining how movement, combat, AI, and level design combine to create engaging gameplay.

8.3. Adventure Game

Adventure games are a beloved genre that focuses on storytelling, exploration, and puzzle-solving.

8.3.1. Game Design

Game Concept

Genre: Adventure game.

Objective: The player explores a narrative-driven world, interacts with characters and environments, solves puzzles, and progresses through a central story revealed through exploration, object interaction, and dialogue.

Core Components

Player Character: The primary character controlled by the player

Attributes: Position (x, y), movement rules, inventory capacity, interaction range

Movement: Controlled via keyboard or mouse input; movement may be grid-based, screen-to-screen, or free-form depending on world design

Interaction: Can examine objects, collect and use items, talk to NPCs, activate environmental triggers, and initiate puzzle actions

Nonplayer Characters (NPCs): Characters inhabiting the world that the player can interact with

Attributes: Position (x, y), dialogue state, quest flags, interaction availability

Function: Provide information, story context, quests, items, or puzzle hints

https://doi.org/10.15215/remix/9781998944224.01

Interaction: Engaged through dialogue menus or context-sensitive actions; dialogue options may change based on game state or player choices

World and Locations: The explorable environments that make up the game world

Attributes: Room or region ID, layout geometry, connected exits, contained objects, and NPCs

Structure: Organized into rooms, zones, or regions such as towns, caves, corridors, forests, or abstract puzzle spaces

Navigation: Movement between locations occurs through doors, passages, portals, or scripted transitions

Objects and Items: Interactive elements that support exploration and puzzle-solving

Attributes: Item ID, name, description, usage rules, combinability

Interaction: Items may be collected, examined, combined with other items, or used on world elements

Purpose: Enable puzzle resolution, gate progress, or advance narrative and quests

Interaction System: Defines how the player engages with the world

Input: Context-sensitive keyboard or mouse actions

Supported Actions: Talking to NPCs, using or inspecting items, opening containers, pressing switches, and reading notes or signs

Feedback: UI prompts, dialogue boxes, or visual/audio cues indicate successful or failed interactions

Puzzle Structures: Logical challenges embedded in the game world

Types: Item-based puzzles, sequence puzzles, environmental navigation puzzles, pattern or code puzzles, and dialogue-based puzzles

Function: Block or enable access to areas, items, or story content

Design Principle: Puzzles are integrated into narrative context and world logic to ensure fairness and clarity

Quest and Story Progression System: Tracks narrative and gameplay progress

Attributes: Active quest list, quest stages, story flags, optional objectives

Function: Controls availability of locations, puzzles, dialogue options, and endings

Progression: Advances through player actions such as puzzle completion, dialogue choices, or item usage

UI and Menu Systems: Interfaces that communicate information and support interaction

Components: Dialogue boxes, inventory menu, quest log, pause menu, settings screen

https://doi.org/10.15215/remix/9781998944224.01

Function: Present narrative text, objectives, item lists, and player choices clearly and unobtrusively

Art and Audio Assets: Visual and audio elements that reinforce atmosphere and storytelling

Visual Assets: Room backgrounds, character sprites, object icons, portraits

Audio Assets: Sound effects for interactions, ambient sounds, optional background music

Purpose: Enhance immersion, mood, and narrative tone

Key Techniques and Features to Have

Adventure games emphasize storytelling, exploration, and puzzle-solving over reflex-driven action. A well-designed adventure game depends on clear narrative structure, intuitive interaction, and consistent world logic. The following features form a strong foundational baseline:

1. **Narrative Structure and Branching Storylines**
   The game should present a clear story arc with main objectives and optional side content. Branching dialogue or player choices should influence progression, with optional multiple endings to enhance replay value.
2. **World Navigation and Exploration Systems**
   Player movement should follow a consistent navigation model (grid-based rooms, screen transitions, or scrolling maps). Visual cues, landmarks, and environmental detail should guide exploration and help players orient themselves.
3. **Interaction Mechanics (Objects, NPCs, Environment)**
   The game should support interaction with objects, NPCs, and the environment through proximity-based or input-driven actions. Typical interactions include talking, examining, using items, and triggering scripted events.
4. **Puzzle and Challenge Design**
   Puzzles should be integrated naturally into the world and narrative, such as item-based, sequence, environmental, or logic puzzles. Progression should be logical and reward observation and reasoning rather than guesswork.
5. **Game State Tracking and Persistence**
   The game should track key progress variables, including location, quest states, collected items, and completed events. Save and load functionality should preserve this state for long-term play.

https://doi.org/10.15215/remix/9781998944224.01

6. **Inventory and Item Management**
   An inventory system should allow players to collect, inspect, combine, and use items. Item usage should directly support puzzle resolution and narrative advancement.
7. **Room and Level Construction**
   Rooms should be clearly structured with walls, doors, and interactive elements, and may be grouped into themed areas or zones. Optional collision boundaries should ensure logical movement within spaces.
8. **User Interface and Feedback Systems**
   The UI should clearly present dialogue, inventory contents, quest information, and interaction prompts. Visual or audio feedback should confirm successful actions or indicate failed attempts.
9. **Event and Script Management**
   A flexible event system should handle story triggers, room transitions, puzzle activations, NPC behavior, and cutscenes. This can begin with simple conditional logic and expand into more scalable scripting.
10. **Art and Atmosphere**
    Consistent visual themes, lighting, and audio should reinforce mood and narrative tone. Ambient effects and sound design can enhance immersion and emotional impact.

8.3.2. Implementation

Key Operations in the Game

Adventure games rely on a continuous cycle of narrative progression, player interaction, and world updates. Unlike action-oriented genres, the core operations here focus on exploration, dialogue, puzzle resolution, and state management. These operations run throughout the game loop and ensure that the world responds consistently to player actions. The following list summarizes the essential operations that drive an adventure game during runtime.

1. **Process Player Input**
   - Read keyboard or mouse input to move the player, interact with objects, or navigate menus.
   - Trigger context-sensitive actions such as talking, examining, or using items.
2. **Update Player and NPC States**
   - Move the player character through rooms or environments.
   - Update NPC behaviors, dialogue availability, or scripted events.
3. **Handle Interactions and Events**
   - Detect when the player is near an interactive object or NPC.

https://doi.org/10.15215/remix/9781998944224.01

- Trigger dialogue, item pickups, puzzle steps, or environmental changes.
- Activate story events based on quest flags or conditions.

4. **Manage Inventory and Puzzle Logic**
   - Add or remove items from the inventory.
   - Check item combinations or usage attempts.
   - Update puzzle states when the player completes required steps.
5. **Track Game State and Story Progression**
   - Maintain variables for quests, visited locations, solved puzzles, and NPC interactions.
   - Unlock new dialogue options, rooms, or events based on progress.
6. **Render the World and UI**
   - Draw rooms, characters, objects, and environmental details.
   - Display dialogue boxes, inventory menus, quest logs, and prompts.
   - Provide visual or audio feedback for successful or failed actions.
7. **Room Transitions and Navigation**
   - Detect when the player enters a doorway or transition zone.
   - Load the next room's layout, objects, and NPCs.
   - Update the player's position accordingly.

Structure of the Game Program

An adventure game is composed of several interconnected systems that manage world layout, player interaction, narrative flow, and rendering. Organizing these systems clearly helps ensure that the game remains scalable as more rooms, puzzles, and story elements are added. The following outline describes the typical structure of an adventure game program.

1. **Initialization**
   - Initialize Allegro subsystems (display, keyboard, image, font, primitives).
   - Create the main window, event queue, and timer.
   - Load fonts, images, and any initial room or character data.
2. **Resource Loading**
   - Load room definitions, object data, and NPC information.
   - Prepare textures, background art, and UI assets.
   - *Optional:* Load dialogue scripts or quest data from external files.
3. **Entity Creation**
   - Create the player character with initial position and attributes.
   - Generate rooms, walls, doors, and interactive objects.
   - Instantiate NPCs and assign their dialogue or behaviors.

https://doi.org/10.15215/remix/9781998944224.01

4. **Input Handling**
   - Capture keyboard or mouse events.
   - Map input to movement, interaction, menu navigation, or dialogue choices.
   - Handle Esc key press or window-close events to exit the game.
5. **Update Functions**
   - Move the player character and apply room boundaries.
   - Check for interactions with objects, NPCs, or puzzle elements.
   - Update quest states, puzzle progress, and triggered events.
   - Manage transitions between rooms or scenes.
6. **Rendering Functions**
   - Draw rooms, walls, doors, and environmental details.
   - Render the player character and NPCs.
   - Display UI elements such as dialogue boxes, inventory menus, and prompts.
   - Flip the display to present the updated frame.
7. **Game Loop**
   - Wait for events from the event queue.
   - On timer events, update the game state and render the scene.
   - Maintain a consistent frame rate (e.g., 60 fps).
8. **Shutdown**
   - Destroy fonts, timers, event queues, and the display.
   - Release any loaded textures or resources.
   - Cleanly exit the program.

Coding the Game

PLAYER_MOVEMENT.CPP

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <allegro5/allegro_primitives.h>
#include "player_movement.h"
#include "room.h"
#include <iostream>

const float FPS = 60;
const int SCREEN_W = 800;
const int SCREEN_H = 600;
const int PLAYER_SIZE = 32;
```

https://doi.org/10.15215/remix/9781998944224.01

```
void run_game_loop(ALLEGRO_DISPLAY* display, ALLEGRO_
FONT* font) {
    al_init_primitives_addon();
    al_install_keyboard();

    ALLEGRO_TIMER *timer = al_create_timer(1.0 / FPS);
    ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();

    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_timer_event_source(timer));
    al_register_event_source(event_queue,
al_get_keyboard_event_source());

    bool running = true;
    bool redraw = true;
    float player_x = SCREEN_W / 2.0f–PLAYER_SIZE / 2.0f;
    float player_y = SCREEN_H / 2.0f–PLAYER_SIZE / 2.0f;
    float player_speed = 4.0f;

std::vector<Room> rooms = generateRooms(
    3,                // number of rooms
    SCREEN_W, SCREEN_H,
    110, 170,         // min/max room size
    6,                // wall thickness
    46                // door width
);

    al_start_timer(timer);

    while (running) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(event_queue, &ev);

        if (ev.type == ALLEGRO_EVENT_TIMER) {
            redraw = true;
        } else if (ev.type == ALLEGRO_EVENT_DISPLAY_
CLOSE) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
            running = false;
        } else if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
            switch (ev.keyboard.keycode) {
                case ALLEGRO_KEY_ESCAPE:
                    running = false;
                    break;
                case ALLEGRO_KEY_UP:
                    player_y -= player_speed;
                    break;
                case ALLEGRO_KEY_DOWN:
                    player_y += player_speed;
                    break;
                case ALLEGRO_KEY_LEFT:
                    player_x -= player_speed;
                    break;
                case ALLEGRO_KEY_RIGHT:
                    player_x += player_speed;
                    break;
            }
        }

        if (redraw && al_is_event_queue_empty(event_
queue)) {
            redraw = false;

            al_clear_to_color(al_map_rgb(0, 0, 0));
            renderRooms(rooms, al_map_rgb(120,120,120),
al_map_rgb(30,30,30));
            al_draw_filled_rectangle(player_x, player_y,
player_x + PLAYER_SIZE, playe
            // Center and draw a white ☺ inside the
square using the existing font
            const char* smile = u8"\u263A";  // ☺
(Unicode U+263A)
            int tw = al_get_text_width(font, smile);
            int th = al_get_font_line_height(font);
            float tx = player_x + (PLAYER_SIZE - tw) * 0.5f;
            float ty = player_y + (PLAYER_SIZE - th) * 0.5f;
            al_draw_text(font, al_map_rgb(255,255,255),
tx, ty, 0, smile);
```

https://doi.org/10.15215/remix/9781998944224.01

```
            al_draw_text(font, al_map_rgb(255, 255, 255),
SCREEN_W / 2, 20, ALLEGRO_AL
            al_flip_display();
        }
    }

    al_destroy_timer(timer);
    al_destroy_event_queue(event_queue);
}
```

PLAYER_MOVEMENT.H

```
#ifndef PLAYER_MOVEMENT_H
#define PLAYER_MOVEMENT_H

#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>

void run_game_loop(ALLEGRO_DISPLAY* display, ALLEGRO_
FONT* font);

#endif
```

ROOM.CPP

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_primitives.h>
#include "room.h"
#include <cstdlib>
#include <ctime>
#include <algorithm>

// Axis-aligned rectangle overlap check
static bool rectsOverlap(float ax,float ay,float aw,float
ah,
                         float bx,float by,float bw,float
bh) {
    return (ax < bx + bw) && (ax + aw > bx) && (ay < by +
bh) && (ay + ah > by);
}

std::vector<Room> generateRooms(
```

https://doi.org/10.15215/remix/9781998944224.01

```
    int count,
    int screenW, int screenH,
    int minSize, int maxSize,
    int wallThickness, int doorWidth
) {
    std::srand((unsigned)std::time(nullptr));
    std::vector<Room> rooms;
    rooms.reserve(count);

    for (int i=0; i<count; ++i) {
        int guard = 0;
        while (true) {
            ++guard;

            float size = (float)(minSize + (std::rand() %
std::max(1, maxSize-minSize+1)));
            // Keep some padding from the screen edges
            float x = (float)(10 + std::rand() %
std::max(1, screenW-(int)size-20));
            float y = (float)(10 + std::rand() %
std::max(1, screenH-(int)size-20));

            // Avoid overlapping prior rooms
            bool ok = true;
            for (const auto& r : rooms) {
                if (rectsOverlap(x, y, size, size, r.x,
r.y, r.size, r.size)) { ok = false; break; }
            }
            if (!ok) {
                if (guard < 1500) continue;
            }

            int   side = std::rand() % 4;      // which
wall gets the door
            float th   = (float)wallThickness;
            float span = size-2.0f*th;       // usable
interior span where a door can be centered
            float dW   = (float)doorWidth;
            if (dW > span-8.0f) dW = std::max(12.0f,
span-8.0f);
```

https://doi.org/10.15215/remix/9781998944224.01

```
            // door center along the wall's interior span
            float doorCenter = th + dW*0.5f + (float)
(std::rand() % std::max(1, (int)(span–dW)));

            rooms.push_back(Room{ x, y, size, th, side,
dW, doorCenter });
            break;
        }
    }
    return rooms;
}

static void drawWallWithGap(float x1, float y1, float x2,
float y2,   // wall outer span
                            float gx1, float gy1, float
gx2, float gy2,// gap rect (door)
                            ALLEGRO_COLOR wallColor,
ALLEGRO_COLOR edgeColor) {
    // Horizontal wall (y1==y2): draw two filled strips
split around the gap
    if (y1 == y2) {
        float th = gy2–gy1;
        if (gx1 > x1) al_draw_filled_rectangle(x1, y1,
gx1, y1 + th, wallColor);
        if (gx2 < x2) al_draw_filled_rectangle(gx2, y1,
x2,  y1 + th, wallColor);

        if (gx1 > x1) al_draw_rectangle(x1, y1, gx1, y1 +
th, edgeColor, 1.0f);
        if (gx2 < x2) al_draw_rectangle(gx2, y1, x2,  y1
+ th, edgeColor, 1.0f);
    } else { // Vertical wall (x1==x2)
        float tw = gx2–gx1;
        if (gy1 > y1) al_draw_filled_rectangle(x1, y1, x1
+ tw, gy1, wallColor);
        if (gy2 < y2) al_draw_filled_rectangle(x1, gy2,
x1 + tw, y2,  wallColor);

        if (gy1 > y1) al_draw_rectangle(x1, y1, x1 + tw,
gy1, edgeColor, 1.0f);
```

https://doi.org/10.15215/remix/9781998944224.01

```
        if (gy2 < y2) al_draw_rectangle(x1, gy2, x1 + tw,
y2,  edgeColor, 1.0f);
    }
}

void renderRooms(const std::vector<Room>& rooms,
                 ALLEGRO_COLOR wallColor,
                 ALLEGRO_COLOR edgeColor) {
    for (const auto& r : rooms) {
        const float x  = r.x;
        const float y  = r.y;
        const float s  = r.size;
        const float th = r.wall;

        // Door gap geometry per wall
        switch (r.doorSide) {
            case 0: { // top (horizontal)
                float gx1 = x + r.doorCenter-r.doorW *
0.5f;
                float gx2 = x + r.doorCenter + r.doorW *
0.5f;
                float gy1 = y, gy2 = y + th;
                drawWallWithGap(x, y, x + s, y, gx1, gy1,
gx2, gy2, wallColor, edgeColor);
            } break;
            case 1: { // right (vertical)
                float gy1 = y + r.doorCenter-r.doorW *
0.5f;
                float gy2 = y + r.doorCenter + r.doorW *
0.5f;
                float gx1 = x + s-th, gx2 = x + s;
                drawWallWithGap(x + s-th, y, x + s-th, y
+ s, gx1, gy1, gx2, gy2, wallColor, edgeColor);
            } break;
            case 2: { // bottom (horizontal)
                float gx1 = x + r.doorCenter-r.doorW *
0.5f;
                float gx2 = x + r.doorCenter + r.doorW *
0.5f;
                float gy1 = y + s-th, gy2 = y + s;
```

https://doi.org/10.15215/remix/9781998944224.01

```
                drawWallWithGap(x, y + s-th, x + s, y +
s-th, gx1, gy1, gx2, gy2, wallColor, edgeColor);
            } break;
            case 3: { // left (vertical)
                float gy1 = y + r.doorCenter-r.doorW *
0.5f;
                float gy2 = y + r.doorCenter + r.doorW *
0.5f;
                float gx1 = x, gx2 = x + th;
                drawWallWithGap(x, y, x, y + s, gx1, gy1,
gx2, gy2, wallColor, edgeColor);
            } break;
        }

        // Other three solid walls (no gap)
        if (r.doorSide != 0) { // top
            al_draw_filled_rectangle(x, y, x + s, y + th,
wallColor);
            al_draw_rectangle(x, y, x + s, y + th,
edgeColor, 1.0f);
        }
        if (r.doorSide != 1) { // right
            al_draw_filled_rectangle(x + s-th, y, x + s,
y + s, wallColor);
            al_draw_rectangle(x + s-th, y, x + s, y + s,
edgeColor, 1.0f);
        }
        if (r.doorSide != 2) { // bottom
            al_draw_filled_rectangle(x, y + s-th, x + s,
y + s, wallColor);
            al_draw_rectangle(x, y + s-th, x + s, y + s,
edgeColor, 1.0f);
        }
        if (r.doorSide != 3) { // left
            al_draw_filled_rectangle(x, y, x + th, y + s,
wallColor);
            al_draw_rectangle(x, y, x + th, y + s,
edgeColor, 1.0f);
        }
    }
}
```

https://doi.org/10.15215/remix/9781998944224.01

ROOM.H

```
#ifndef ROOM_H
#define ROOM_H

#include <vector>
#include <allegro5/allegro.h>

// A square room with a single doorway in one wall.
struct Room {
    float x;        // top-left of outer square
    float y;
    float size;     // outer square size (width ==
height)
    float wall;     // wall thickness
    int   doorSide; // 0=top, 1=right, 2=bottom, 3=left
    float doorW;    // door width
    float doorCenter; // along the selected wall, local
coordinate
};

// Generate non-overlapping rooms (best effort) inside
screen bounds
std::vector<Room> generateRooms(
    int count,
    int screenW, int screenH,
    int minSize = 100, int maxSize = 180,
    int wallThickness = 6,
    int doorWidth = 40
);

// Draw room borders with a gap for the door
void renderRooms(const std::vector<Room>& rooms, ALLEGRO_
COLOR wallColor, ALLEGRO_COLOR edgeColor);

#endif // ROOM_H
```

https://doi.org/10.15215/remix/9781998944224.01

TASKS FOR COMPLETION AND IMPROVEMENT

1. **Level Design:** Design the layout of each level, including key locations, obstacles, and objectives. Add collisions to make room walls solid
2. **Puzzles:** Create puzzles that challenge the player and fit seamlessly into the game's world.
3. **UI Elements:** Add code for holding the Shift key down as a function to make the player move continuously (fast movement).
4. **Menus and HUD:** Create intuitive menus and HUD elements that provide necessary information to the player.
5. **Usability Testing:** Test the UI to ensure it is user-friendly and accessible.
6. **Bug Fixing:** Identify and fix bugs through rigorous testing.
7. **Balancing:** Ensure the game is balanced and provides a fair challenge.
8. **Polish:** Add final touches to the game, such as visual effects, sound enhancements, and minor gameplay tweaks.
9. **Final Testing:** Perform a final round of testing to ensure the game is polished and free of major issues.

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Exercises

1. **Adventure Game System Mapping**
   Identify the major systems described in this chapter (player movement, rooms, NPCs, interaction, puzzles, inventory, quests, UI). Briefly explain how at least three of these systems interact to support exploration-driven gameplay.
2. **Exploration vs. Action**
   Compare adventure games with action-oriented games. Why do adventure games emphasize slower pacing, environmental detail, and player investigation rather than rapid reflexes?
3. **Room-Based World Design**
   Explain how dividing the world into rooms or locations simplifies navigation and puzzle design. What advantages does this provide over a single large continuous map for an adventure game?
4. **Puzzle Integration**
   Choose one type of puzzle described in the chapter (item-based, environmental, dialogue-based, or sequence puzzle). Explain how it can be integrated naturally into the game world and narrative.

https://doi.org/10.15215/remix/9781998944224.01

Homework Questions

1. **Narrative Structure and Player Agency**
   Discuss how branching dialogue and player choice can influence storytelling in adventure games. How can designers balance player freedom with a coherent narrative?
2. **Inventory-Based Puzzles**
   Why are inventory puzzles a common feature in adventure games? What design principles ensure these puzzles feel logical rather than frustrating?
3. **Game-State Tracking**
   Adventure games rely heavily on tracking progress (visited rooms, solved puzzles, items collected). What problems might occur if game-state variables are poorly designed or inconsistently updated?
4. **Feedback and Fairness**
   Explain why clear feedback is essential in adventure games when a player attempts an incorrect action. How does feedback help prevent players from becoming stuck?

Programming Projects

1. **Room Collision and Boundaries (Core Project)**
   Extend the room system by adding collision detection so the player cannot walk through walls except at door locations. Ensure movement feels smooth and predictable.
2. **Object Interaction System**
   Implement interactable objects within rooms, such as doors, switches, notes, or keys. Allow the player to examine or use these objects with a dedicated interaction key.
3. **Inventory and Item Usage**
   Add an inventory system that allows players to collect items, view them in a menu, and use them to solve puzzles or unlock new areas.
4. **Puzzle Implementation**
   Create at least one complete puzzle that requires the player to explore rooms, gather information or items, and perform actions in a specific order to progress.
5. **NPC Dialogue and Story Hooks (Advanced)**
   Add NPCs that provide dialogue, hints, or story context. Implement branching dialogue options that change based on quest progress or prior interactions.

https://doi.org/10.15215/remix/9781998944224.01

6. **Quest and Objective Tracking (Advanced)**
   Introduce a simple quest system that tracks objectives and updates as the player completes tasks or solves puzzles. Display the current objective in a quest log or HUD.
7. **Save and Load System (Advanced)**
   Implement a save/load feature that preserves the player's location, inventory contents, puzzle states, and story progress across sessions.

Capstone Project

Design and implement a small but complete adventure game scenario containing multiple rooms, interactive objects, at least one NPC, and a short puzzle-driven quest chain. The project should demonstrate coherent integration of exploration, interaction, puzzle-solving, and narrative feedback. Submit a brief design summary explaining how the story, puzzles, and environment work together to guide the player.

8.4. Arcade Game

Arcade games are a genre that emphasizes fast-paced gameplay, simple controls, and increasing difficulty. They originated in the coin-operated arcade machines of the seventies and eighties but have since evolved to include a wide variety of games available on multiple platforms. Here are some key aspects of arcade games.

8.4.1. Game Design

Game Concept

Genre: 2D arcade space shooter.

Objective: The player controls a spaceship to survive for as long as possible by destroying incoming enemies, avoiding collisions, preserving a limited number of lives, and achieving the highest possible score before all lives are lost.

Gameplay Mechanics:

- Move the spaceship left and right.
- Shoot bullets to destroy enemies.
- Avoid colliding with enemies.

Core Components

Player Ship: The spaceship controlled by the player

Attributes: Position (x, y), movement speed, sprite, remaining lives

https://doi.org/10.15215/remix/9781998944224.01

Movement: Controlled via keyboard input; the ship moves left and right along the bottom of the screen and is constrained within screen boundaries

Shooting: Fires bullets upward when the player presses the designated fire key

Enemies: Hostile entities that threaten the player

Attributes: Position (x, y), movement speed, size or sprite

Movement: Spawn near the top of the screen and move downward toward the player at a constant or gradually increasing speed

Behavior: Must be destroyed by player bullets or avoided to prevent collisions

Bullets: Projectiles fired by the player's spaceship

Attributes: Position (x, y), speed, size

Movement: Travel upward from the player's ship toward enemies

Collision: Deactivate when they strike an enemy or leave the screen

Scoring System: Tracks the player's progress and performance

Attributes: Current score

Function: Increases the score when an enemy is successfully destroyed

Lives System: Represents the player's remaining chances to stay in the game

Attributes: Remaining lives (e.g., three at the start)

Rules: A collision between the player ship and an enemy results in the loss of one life; the game ends when all lives are lost

Key Techniques and Features to Have

Arcade games depend on tight controls, fast feedback, and simple but addictive mechanics. The following features form a solid baseline for an arcade-style space shooter.

1. **Responsive Player Controls**
   Player movement should be smooth, immediate, and low latency. The ship must be constrained within screen boundaries to prevent off-screen movement, ensuring precise and reliable control.
2. **Shooting and Projectile Management**
   The player should fire bullets using a dedicated key. Bullets must spawn relative to the player's position, update every frame, and be removed when they leave the screen to maintain performance.
3. **Enemy Generation and Behavior**
   Enemies should spawn near the top of the screen and move downward toward the player. Speed and behavior variations (such as

https://doi.org/10.15215/remix/9781998944224.01

faster movement or simple patterns) can be introduced to increase challenge. Off-screen or destroyed enemies must be cleaned up promptly.

4. **Collision Detection**
   Collision checks should detect bullet–enemy and player–enemy interactions using simple AABB logic. Colliding objects should be deactivated immediately to keep gameplay fast and unambiguous.
5. **Scoring and Life System**
   The game should increment the score when enemies are destroyed and track player lives. Collisions with enemies reduce lives, and the game ends when lives reach zero, optionally displaying a game-over message.
6. **Visual Feedback and Simple Effects**
   The player, enemies, and bullets should be rendered with clear shapes or sprites. Simple visual effects—such as flashes, colour changes, or small explosion cues—can enhance readability and feedback without increasing complexity.
7. **Increasing Difficulty Over Time**
   Difficulty should rise gradually through increased enemy spawn rates, higher speeds, or tougher enemy variants, sustaining tension and encouraging repeat play.
8. **Game Loop Structure and Allegro Integration**
   An Allegro timer (e.g., 60 fps) should drive consistent updates. Game logic and rendering must be separated, and input and window events should be handled through Allegro's event queue.
9. **User Interface Elements**
   The interface should display the score and remaining lives clearly. When the game ends, a centered game-over message should be shown for clear feedback.
10. **Simplicity, Speed, and Replayability**
    The game should focus on a small set of core actions—moving, shooting, and dodging. Clear visuals, smooth motion, and short play sessions reinforce replayability, a hallmark of arcade games

8.4.2. Implementation

Just as for any game development project with Allegro 5 in C++, we will assume you have all the software tools and the Allegro 5 library installed on the computer platform you have chosen. If not, please go back to chapter 1 to complete the installation.

https://doi.org/10.15215/remix/9781998944224.01

Key Operations in the Game

Arcade games depend on a fast, tightly controlled sequence of operations that repeat every frame to maintain smooth gameplay and immediate responsiveness. These operations ensure that the player's actions, enemy behavior, collisions, and scoring all update in real time. The following list summarizes the essential runtime operations that drive the arcade-style space shooter:

1. **Process Player Input**
   - Read keyboard input for left/right movement and shooting.
   - Clamp the player's position to screen boundaries to prevent off-screen movement.
2. **Update Player, Bullet, and Enemy States**
   - Move the player based on input.
   - Advance bullets upward and deactivate them when they leave the screen.
   - Move enemies downward and remove them when they exit the screen.
3. **Spawn New Enemies**
   - Periodically generate enemies at random horizontal positions.
   - Adjust spawn frequency or speed to increase difficulty over time.
4. **Perform Collision Detection**
   - Check bullet–enemy collisions and award points for successful hits.
   - Check player–enemy collisions to determine life loss or trigger game-over conditions.
   - Deactivate bullets and enemies immediately upon collision.
5. **Manage Scoring and Lives**
   - Increase score when enemies are destroyed.
   - Track remaining lives and reduce them when the player is hit.
   - Activate a game-over state when lives reach zero.
6. **Clean Up Inactive Entities**
   - Remove bullets and enemies that are no longer active.
   - Prevent memory growth by regularly pruning inactive objects.
7. **Render the Frame**
   - Draw the player, bullets, enemies, score, and lives.
   - Display flashing effects, hit indicators, or "GAME OVER" banners when appropriate.
   - Flip the display to present the updated frame to the player.

https://doi.org/10.15215/remix/9781998944224.01

Structure of the Game Program

1. **Initialization**
   - Initialize Allegro and its add-ons for images, primitives, fonts, and keyboard input.
   - Create a display window, font, timer, and event queue.
2. **Game Loop**
   - Handle events for updating game state, such as player movement, bullet firing, and enemy spawning.
   - Update the positions of bullets and enemies, and check for collisions.
   - Draw the player, bullets, enemies, and score/lives on the screen.
3. **Player Movement**
   - Move the player left and right using the arrow keys.
   - Fire bullets using the space bar.
4. **Collision Detection**
   - Check for collisions between bullets and enemies, and update the score/lives accordingly.
5. **Rendering**
   - Clear the screen and draw the game elements (player, bullets, enemies, score, and lives).

Coding the Game

Here's the basic structure of the game:

```
// simple_shooter.cpp  (Lives, player collision, GAME
OVER)
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <vector>
#include <cstdlib>
#include <ctime>   // for std::srand / std::time

const float FPS = 60;
const int SCREEN_W = 800;
const int SCREEN_H = 600;
const int PLAYER_SIZE = 32;
const int BULLET_SIZE = 8;
```

https://doi.org/10.15215/remix/9781998944224.01

```
const int ENEMY_SIZE = 32;

struct Bullet {
    float x, y;
    bool active;
};

struct Enemy {
    float x, y;
    bool active;
};

int main() {
    al_init();
    al_init_image_addon();
    al_init_primitives_addon();
    al_init_font_addon();
    al_init_ttf_addon();
    al_install_keyboard();

    std::srand((unsigned)std::time(nullptr));

    ALLEGRO_DISPLAY *display = al_create_
display(SCREEN_W, SCREEN_H);
    ALLEGRO_FONT *font = al_load_ttf_font("arial.ttf",
24, 0);
    if (!font) font = al_create_builtin_font();

    ALLEGRO_TIMER *timer = al_create_timer(1.0 / FPS);
    ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();

    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_timer_event_source(timer));
    al_register_event_source(event_queue,
al_get_keyboard_event_source());
```

https://doi.org/10.15215/remix/9781998944224.01

```
    bool running = true;
    bool redraw = true;
    float player_x = SCREEN_W / 2.0f–PLAYER_SIZE / 2.0f;
    float player_y = SCREEN_H–PLAYER_SIZE–10;
    float player_speed = 5.0f;

    int score = 0;

    // Flashing faces on enemies
    int  flashCounter = 0;
    bool flashPhase = false;   // false => 🙂 (white),
true => 😃 (black)

    // NEW: lives + game-over state
    int  lives = 3;                 // start with 3 lives
    bool gameOver = false;
    int  gameOverFrames = (int)(5 * FPS);  // ~5s pause
before exit when game over

    std::vector<Bullet> bullets;
    std::vector<Enemy> enemies;

    al_start_timer(timer);

    while (running) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(event_queue, &ev);

        if (ev.type == ALLEGRO_EVENT_TIMER) {
            if (!gameOver) {
                // Update bullets
                for (auto &bullet : bullets) {
                    if (bullet.active) {
                        bullet.y–= 8;
                        if (bullet.y < 0) {
                            bullet.active = false;
                        }
                    }
                }
```

https://doi.org/10.15215/remix/9781998944224.01

```
                // Update enemies
                for (auto &enemy : enemies) {
                    if (enemy.active) {
                        enemy.y += 2;
                        if (enemy.y > SCREEN_H) {
                            enemy.active = false;
                        }
                    }
                }

                // Flash phase toggle ~4 times/sec (every
15 frames at 60 FPS)
                if (++flashCounter >= 15) {
                    flashCounter = 0;
                    flashPhase = !flashPhase;
                }

                // Bullet-enemy collisions
                for (auto &bullet : bullets) {
                    if (!bullet.active) continue;
                    for (auto &enemy : enemies) {
                        if (!enemy.active) continue;
                        if (bullet.x < enemy.x + ENEMY_
SIZE &&
                            bullet.x + BULLET_SIZE >
enemy.x &&
                            bullet.y < enemy.y + ENEMY_
SIZE &&
                            bullet.y + BULLET_SIZE >
enemy.y) {
                            bullet.active = false;
                            enemy.active = false;
                            score += 10;
                        }
                    }
                }

                // Spawn new enemies
                if (std::rand() % 50 == 0) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
                    enemies.push_back({static_
cast<float>(std::rand() % (SCREEN_W–ENEMY_SIZE)), 0,
true});
                }

                // NEW: Player-enemy collision (use
triangle's AABB)
                bool playerHit = false;
                for (auto &enemy : enemies) {
                    if (!enemy.active) continue;
                    bool overlap =
                        player_x < enemy.x + ENEMY_SIZE &&
                        player_x + PLAYER_SIZE > enemy.x
&&
                        player_y < enemy.y + ENEMY_SIZE &&
                        player_y + PLAYER_SIZE > enemy.y;
                    if (overlap) {
                        enemy.active = false;   //
consume the enemy
                        playerHit = true;
                    }
                }
                if (playerHit) {
                   –lives;
                    if (lives <= 0) {
                        gameOver = true;
                        gameOverFrames = (int)(5 * FPS);
                    }
                }
            } else {
                // NEW: simple 5-second pause then exit
                if (–gameOverFrames <= 0) {
                    running = false;
                }
            }

            redraw = true;
        } else if (ev.type == ALLEGRO_EVENT_DISPLAY_
CLOSE) {
            running = false;
```

https://doi.org/10.15215/remix/9781998944224.01

```
        } else if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {
            switch (ev.keyboard.keycode) {
                case ALLEGRO_KEY_LEFT:
                    player_x -= player_speed;
                    if (player_x < 0) player_x = 0;
                    break;
                case ALLEGRO_KEY_RIGHT:
                    player_x += player_speed;
                    if (player_x > SCREEN_W - PLAYER_SIZE)
player_x = SCREEN_W - PLAYER_SIZE;
                    break;
                case ALLEGRO_KEY_SPACE:
                    bullets.push_back({player_x + PLAYER_
SIZE / 2 - BULLET_SIZE / 2, player_y, true});
                    break;
                case ALLEGRO_KEY_ESCAPE:
                    running = false;
                    break;
            }
        }

        if (redraw && al_is_event_queue_empty(event_
queue)) {
            redraw = false;

            al_clear_to_color(al_map_rgb(0, 0, 0));

            // Draw player (upright triangle)
            al_draw_filled_triangle(
                player_x + PLAYER_SIZE * 0.5f,
player_y,                  // top
                player_x,                          player_y
+ PLAYER_SIZE, // bottom-left
                player_x + PLAYER_
SIZE,           player_y + PLAYER_SIZE, // bottom-right
                al_map_rgb(0, 255, 0)
            );

            // Draw bullets
            for (const auto &bullet : bullets) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
                if (bullet.active) {
                    al_draw_filled_rectangle(
                        bullet.x, bullet.y,
                        bullet.x + BULLET_SIZE, bullet.y
+ BULLET_SIZE,
                        al_map_rgb(255, 255, 255)
                    );
                }
            }

            // Draw enemies with flashing 🙂 / 😃 overlay
            for (const auto &enemy : enemies) {
                if (!enemy.active) continue;

                // enemy block
                al_draw_filled_rectangle(
                    enemy.x, enemy.y,
                    enemy.x + ENEMY_SIZE, enemy.y +
ENEMY_SIZE,
                    al_map_rgb(255, 0, 0)
                );

                // flashing face
                const char* whiteFace = u8"\u263A"; // 🙂
                const char* blackFace = u8"\u263B"; // 😃
                const char* face = flashPhase ? blackFace
: whiteFace;
                ALLEGRO_COLOR faceColor = flashPhase ?
al_map_rgb(0, 0, 0) : al_map_rgb(255, 255, 255);

                int tw = al_get_text_width(font, face);
                int th = al_get_font_line_height(font);
                float tx = enemy.x + (ENEMY_SIZE-tw) *
0.5f;
                float ty = enemy.y + (ENEMY_SIZE-th) *
0.5f;
                al_draw_text(font, faceColor, tx, ty, 0,
face);
            }
```

https://doi.org/10.15215/remix/9781998944224.01

```
            // Score (top-left)
            al_draw_textf(font, al_map_rgb(255, 255,
255), 10, 10, 0, "Score: %d", score);

            // NEW: Lives (top-right, opposite the score)
            al_draw_textf(font, al_map_rgb(255, 255,
255),
                          SCREEN_W-10, 10,
ALLEGRO_ALIGN_RIGHT,
                          "Lives: %d", lives);

            // NEW: GAME OVER banner
            if (gameOver) {
                const char* msg = "GAME OVER";
                int tw = al_get_text_width(font, msg);
                int th = al_get_font_line_height(font);
                float x = (SCREEN_W-tw) * 0.5f;
                float y = (SCREEN_H-th) * 0.5f;

                // optional dim panel behind text (works
best with alpha-enabled target)
                al_draw_filled_rectangle(x-20, y-10, x +
tw + 20, y + th + 10, al_map_rgba(0, 0, 0, 180));
                al_draw_text(font, al_map_rgb(255, 64,
64), x, y, 0, msg);
            }

            al_flip_display();
        }
    }

    al_destroy_font(font);
    al_destroy_timer(timer);
    al_destroy_event_queue(event_queue);
    al_destroy_display(display);
    return 0;
}
```

https://doi.org/10.15215/remix/9781998944224.01

TASKS FOR COMPLETION AND IMPROVEMENT

1. **Level Design and Difficulty Progression**
   Enemy Waves: Introduce structured waves of enemies with increasing complexity.
   Speed Scaling: Gradually increase enemy speed or spawn frequency over time.
   Obstacle Patterns: Add falling obstacles or hazards that require precise dodging.
2. **User Interface (UI) and User Experience (UX) Enhancements**
   HUD Improvements: Add visual indicators for score, lives, level, and power-ups.
   Menus: Create a start menu, pause menu, and game-over screen with restart options.
   Accessibility: Ensure clear feedback for hits, missed shots, and player damage.
3. **Testing and Quality Assurance**
   Collision Testing: Ensure bullet–enemy and player–enemy collisions are precise and fair.
   Spawn Testing: Test edge cases where too many enemies spawn simultaneously.
   Performance: Monitor frame performance under heavy activity (many bullets, enemies).
4. **Content Expansion and Gameplay Variety**
   Power-Ups: Add temporary boosts such as faster shooting, shields, bigger bullets.
   Enemy Types: Introduce different enemy behaviors (zigzag, homing, large slow enemies).
   Boss Fights: Add a special boss enemy every few waves for challenge and pacing.
5. **Visual and Audio Polish**
   Particle Effects: Add small explosion primitives when enemies are destroyed.
   Background Animation: Implement simple scrolling stars or parallax backgrounds.
   Sound Effects: Add sounds for firing, explosions, collecting power-ups, and losing lives.
6. **Finalization and Release Preparation**
   Bug Fixing: Ensure no soft-locks, infinite bullets, or overlapping UI issues occur.

https://doi.org/10.15215/remix/9781998944224.01

Balancing: Adjust enemy hit/health points (HP), speed, and spawn rates for an enjoyable difficulty curve.

Final Testing: Run full game sessions to ensure smooth pacing and stable performance.

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Exercises

1. **Arcade Design Principles**
   Explain why arcade games prioritize simple controls and immediate visual feedback. How do these principles contribute to short, replayable play sessions?
2. **Game Loop Structure**
   Describe the role of the game loop in an arcade shooter. Why is it important to separate update logic (movement, collisions) from rendering logic?
3. **Collision Detection Strategy**
   Review the bullet–enemy and player–enemy collision checks used in the game. Why are axis-aligned bounding box (AABB) checks sufficient for this type of arcade game?
4. **Lives and Game-Over Logic**
   Explain how the life system controls game difficulty and session length. What happens to gameplay tension as the number of remaining lives decreases?

Homework Questions

1. **Difficulty Progression**
   Arcade games often become more difficult over time. Discuss at least three ways difficulty can be scaled in a space shooter without changing the basic controls.
2. **Performance Considerations**
   Why is it important to remove inactive bullets and enemies from memory each frame? What problems might arise if inactive objects are never cleaned up?
3. **Visual Readability**
   Arcade games typically use high-contrast colours and simple shapes. Why is visual clarity especially important when many enemies or projectiles are on screen?

https://doi.org/10.15215/remix/9781998944224.01

4. **Replay Value**
   What features encourage players to replay arcade games repeatedly? Consider scoring systems, randomness, and escalating challenge in your answer.

Programming Projects

1. **Enemy Waves and Patterns (Core Project)**
   Replace random enemy spawning with structured waves. Each wave should increase in difficulty by introducing faster enemies, denser formations, or new movement patterns.
2. **Difficulty Scaling System**
   Implement gradual difficulty progression by increasing enemy speed, spawn frequency, or bullet speed as the player's score increases.
3. **Power-Ups and Bonuses**
   Add power-ups that temporarily enhance the player, such as faster firing, shields, wider bullets, or bonus points. Ensure power-ups expire after a fixed duration.
4. **Additional Enemy Types**
   Introduce new enemy behaviors, such as zigzag movement and homing shots, or large, slow enemies that require multiple hits to destroy.
5. **Score Multipliers and Combos (Advanced)**
   Implement a combo or multiplier system that rewards players for destroying enemies consecutively without missing or taking damage.
6. **Pause and Menu System (Advanced)**
   Add a start menu, pause menu, and game-over menu. Allow the player to restart or exit cleanly without closing the program.

Capstone Project (Optional)

Design and implement a polished arcade space shooter that includes multiple enemy waves, difficulty scaling, power-ups, scoring depth, and visual and audio effects. The game should be fully self-contained, support repeated play sessions, and clearly demonstrate classic arcade design principles. Include a short write-up explaining how your design choices enhance challenge, clarity, and replayability.

8.5. Board Game

Board games are a wonderful way to bring people together, promote social interaction, and create lasting memories. Whether you're playing with

https://doi.org/10.15215/remix/9781998944224.01

family, friends, or new acquaintances, the social benefits of board games are undeniable.

In this project, we'll create a basic tic-tac-toe game, where two players take turns placing their marks (X or O) on a 3 × 3 grid.

8.5.1. Game Design

Game Concept

Genre: Board game.

Objective: Two players take turns placing their marks (X or O) on a 3 × 3 grid. The first player to align three marks horizontally, vertically, or diagonally wins the game. If all cells are filled without a winning alignment, the game ends in a draw.

Gameplay Mechanics:

- Players click on a cell to place their mark.
- The game checks for a win or draw after each move.

Core Components

Game Board (Grid): The play area where the game takes place

Attributes: Grid size (3 × 3), cell dimensions, cell states (empty, X, O)

Function: Defines valid positions for placing marks and serves as the basis for win and draw detection logic

Player Marks: The symbols placed by players during the game

Types: X and O

Attributes: Mark type, grid position (row, column)

Rules: Marks can only be placed in empty grid cells and cannot be changed once placed

Turn Management System: Controls which player may act at any given time

Attributes: Current player (Player X or Player O)

Function: Alternates turns after each valid move and prevents multiple actions in the same turn

Feedback: Optionally displays a visual or textual indicator showing whose turn it is

Win and Draw Detection System: Determines the outcome of the game

Checks: Three matching marks in any row, column, or diagonal

Draw Condition: All grid cells are filled and no winning alignment exists

Function: Ends the game when a win or draw condition is met and prevents further input

https://doi.org/10.15215/remix/9781998944224.01

Key Techniques and Features to Have

Although Tic-Tac-Toe is a simple board game, a clean implementation in Allegro 5 still requires clear state management, precise input handling, and readable rendering. The following features define a solid and reliable baseline.

1. **Grid Representation and Board-State Management**
   The game board should be represented by a 3 × 3 data structure storing cell states (empty, X, or O). Helper functions should reset the board, query cell values, and enforce valid updates so marked cells cannot be overwritten.
2. **Player Input Handling (Mouse Interaction)**
   Mouse clicks should be detected through Allegro's event system and translated from screen coordinates to grid positions. Moves must be validated, and turns should alternate automatically between Player X and Player O.
3. **Turn Management and Visual Feedback**
   The current player should be tracked explicitly and updated after each valid move. The interface should clearly indicate whose turn it is, with optional hover highlighting to improve usability.
4. **Win and Draw Detection**
   After each move, the game should check for three-in-a-row in rows, columns, or diagonals. A draw should be detected when the board is full with no winner, and further input should be disabled when the game ends.
5. **Rendering the Board and Marks**
   The grid should be drawn using Allegro primitives, with X and O marks rendered clearly using consistent padding, contrast, and line thickness to ensure readability at all resolutions.
6. **End-of-Game Display**
   Clear messages such as "Player X Wins," "Player O Wins," or "It's a Draw" should be displayed and centered on the screen. Optional visual emphasis, such as dimming the background or highlighting the winning line, may be added.
7. **Game Reset or Replay (Optional)**
   The game may provide a simple reset mechanism, such as a key press or replay button, allowing players to start a new round without restarting the application.

https://doi.org/10.15215/remix/9781998944224.01

8. **Basic UI and User Experience Considerations**
   All text should use readable fonts with adequate spacing. Optional enhancements such as hover effects or subtle animations can modernize the experience without adding complexity.

8.5.2. Implementation

Just as for any game development project with Allegro 5 in C++, we will assume you have all the software tools and the Allegro 5 library installed on the computer platform you have chosen. If not, please go back to chapter 1 to complete the installation.

Key Operations in the Game

Implementing a tic-tac-toe game in Allegro 5 requires several core operations that ensure smooth gameplay, accurate input handling, and correct game state evaluation. These operations form the functional backbone of the program:

1. **Mouse Input and Cell Selection**
   - Detect mouse clicks using Allegro's mouse event system.
   - Convert screen coordinates (x, y) into grid indices (row, col) using the formula:
     - row = y / CELL_SIZE
     - col = x / CELL_SIZE
   - Validate that the selected cell is within bounds and currently empty before placing a mark.
2. **Mark Placement**
   - Place the current player's mark (X or O) into the selected grid cell.
   - Prevent overwriting by checking grid[row][col] == NONE before assignment.
   - Switch the turn to the other player immediately after a valid placement.
3. **Board Rendering**
   - Draw the 3 × 3 grid using al_draw_line() to create dividing lines.
   - Draw all X and O marks based on the grid array:
     - X marks: drawn with two diagonal lines.
     - O marks: drawn with a circle centered in the cell.
   - Ensure consistency using padding, stroke width, and centered alignment.
4. **Win Detection**
   - After every move, check the grid for a winning pattern:
     - Any of the three rows

https://doi.org/10.15215/remix/9781998944224.01

    - Any of the three columns
    - Either diagonal
  - Return the winning player when a match of three identical marks is detected.
5. **Draw Detection**
  - If the board is completely filled and no winner has been found, declare a draw.
  - This operation ensures the game ends correctly even without a winning line.
6. **Turn Management**
  - Alternate between PLAYER_X and PLAYER_O after each valid move.
  - Maintain a variable (e.g., currentPlayer) for easy swapping.
  - *Optional:* Display the active player on the screen to guide users.
7. **Game State Rendering**
  - Update the display every loop by drawing:
    - The grid
    - Current marks
    - Text messages such as "Player X Wins!" or "It's a Draw!"
  - Use Allegro's font features for clean, centered text output.
8. **End-of-Game Handling**
  - Stop accepting clicks once a winner or draw has been declared.
  - Show the final result clearly on-screen.
  - *Optional:* Offer restart behavior via a button, key press, or automatic reset.

Structure of the Game Program

Before you start coding, keep in mind the following general structure of the game program:

1. **Initialization**
  - Initialize Allegro and its add-ons for primitives, fonts, and mouse input.
  - Create a display window, font, and event queue.
2. **Game Loop**
  - Handle events for mouse clicks to place marks on the grid.
  - Update the game state, including checking for a winner or draw.
  - Draw the grid, marks, and game status (winner or draw).
3. **Drawing Functions**
  - draw_grid(): Draws the 3 × 3 grid.
  - draw_marks(): Draws the X and O marks on the grid.

https://doi.org/10.15215/remix/9781998944224.01

4. **Game Logic**
   - check_winner(): Checks if there is a winner.
   - is_draw(): Checks if the game is a draw.
5. **Shutdown**
   - Clean release of Allegro resources.

For gameplay, the first player selects a square, then the second player proceeds to do the same. Play proceeds until one player manages to complete three in a row for their respective character (X or O) or the game ends in a draw.

Coding the Game

CODE—BOARD_GAME.CPP

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_primitives.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <iostream>

const int SCREEN_W = 600;
const int SCREEN_H = 600;
const int GRID_SIZE = 3;
const int CELL_SIZE = SCREEN_W / GRID_SIZE;
const int PADDING = 12; // visual padding for X/O
rendering

enum Player { NONE, PLAYER_X, PLAYER_O };

Player grid[GRID_SIZE][GRID_SIZE];
Player currentPlayer = PLAYER_X;

// Draw the 3 × 3 grid (skip the outer border lines)
void draw_grid() {
    for (int i = 1; i < GRID_SIZE; ++i) {
        al_draw_line(i * CELL_SIZE, 0, i * CELL_SIZE,
SCREEN_H, al_map_rgb(255, 255, 255), 2);
        al_draw_line(0, i * CELL_SIZE, SCREEN_W, i *
CELL_SIZE, al_map_rgb(255, 255, 255), 2);
    }
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
// Draw X and O marks that are already placed
void draw_marks() {
    for (int row = 0; row < GRID_SIZE; ++row) {
        for (int col = 0; col < GRID_SIZE; ++col) {
            if (grid[row][col] == PLAYER_X) {
                // Two diagonals to form an X
                al_draw_line(col * CELL_SIZE +
PADDING,          row * CELL_SIZE + PADDING,
                             (col + 1) *
CELL_SIZE-PADDING,  (row + 1) * CELL_SIZE-PADDING,
                             al_map_rgb(255, 0, 0), 4);
                al_draw_line((col + 1) *
CELL_SIZE-PADDING,   row * CELL_SIZE + PADDING,
                             col * CELL_SIZE +
PADDING,          (row + 1) * CELL_SIZE-PADDING,
                             al_map_rgb(255, 0, 0), 4);
            } else if (grid[row][col] == PLAYER_O) {
                // Circle centered in the cell
                al_draw_circle(col * CELL_SIZE + CELL_
SIZE / 2.0f,
                               row * CELL_SIZE + CELL_
SIZE / 2.0f,
                               CELL_SIZE / 2.0f-PADDING,
                               al_map_rgb(0, 0, 255), 4);
            }
        }
    }
}

// Return the winner, if any; otherwise NONE
Player check_winner() {
    // Rows and columns
    for (int i = 0; i < GRID_SIZE; ++i) {
        // Row i
        if (grid[i][0] != NONE &&
            grid[i][0] == grid[i][1] &&
            grid[i][1] == grid[i][2]) {
            return grid[i][0];
        }
        // Column i
```

https://doi.org/10.15215/remix/9781998944224.01

```
        if (grid[0][i] != NONE &&
            grid[0][i] == grid[1][i] &&
            grid[1][i] == grid[2][i]) {
            return grid[0][i];
        }
    }
    // Diagonals
    if (grid[0][0] != NONE &&
        grid[0][0] == grid[1][1] &&
        grid[1][1] == grid[2][2]) {
        return grid[0][0];
    }
    if (grid[0][2] != NONE &&
        grid[0][2] == grid[1][1] &&
        grid[1][1] == grid[2][0]) {
        return grid[0][2];
    }
    return NONE;
}

bool is_draw() {
    for (int row = 0; row < GRID_SIZE; ++row) {
        for (int col = 0; col < GRID_SIZE; ++col) {
            if (grid[row][col] == NONE) return false;
        }
    }
    return true;
}

int main() {
    if (!al_init()) {
        std::cerr << "Failed to init Allegro\\n";
        return-1;
    }
    al_init_primitives_addon();
    al_init_font_addon();
    al_init_ttf_addon();
    al_install_mouse();

    // Explicitly clear board to NONE
```

https://doi.org/10.15215/remix/9781998944224.01

```
    for (int r = 0; r < GRID_SIZE; ++r) {
        for (int c = 0; c < GRID_SIZE; ++c) grid[r][c] =
NONE;
    }

    ALLEGRO_DISPLAY *display = al_create_
display(SCREEN_W, SCREEN_H);
    if (!display) {
        std::cerr << "Failed to create display\\n";
        return-1;
    }

    ALLEGRO_FONT *font = al_load_ttf_font("arial.ttf",
28, 0);
    if (!font) {
        // Fall back to built-in font so the program
still runs even if TTF load fails
        font = al_create_builtin_font();
    }

    ALLEGRO_EVENT_QUEUE *event_queue =
al_create_event_queue();
    al_register_event_source(event_queue,
al_get_display_event_source(display));
    al_register_event_source(event_queue,
al_get_mouse_event_source());

    bool running = true;
    Player winner = NONE;

    while (running) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(event_queue, &ev);
        if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            running = false;
        } else if (ev.type == ALLEGRO_EVENT_MOUSE_BUTTON_
DOWN) {
            int col = ev.mouse.x / CELL_SIZE;
            int row = ev.mouse.y / CELL_SIZE;
```

https://doi.org/10.15215/remix/9781998944224.01

```
            if (row >= 0 && row < GRID_SIZE && col >= 0
&& col < GRID_SIZE) {
                if (grid[row][col] == NONE && winner ==
NONE) {
                    grid[row][col] = currentPlayer;
                    currentPlayer = (currentPlayer ==
PLAYER_X) ? PLAYER_O : PLAYER_X; // toggle player
                    winner = check_winner();
                }
            }
        }

        al_clear_to_color(al_map_rgb(0, 0, 0));
        draw_grid();
        draw_marks();

        if (winner != NONE) {
            const char* msg = (winner == PLAYER_X) ?
"Player X Wins!" : "Player O Wins!";
            al_draw_text(font, al_map_rgb(255, 255, 255),
SCREEN_W / 2.0f, SCREEN_H / 2.0f, ALLEGRO_ALIGN_CENTER,
msg);
        } else if (is_draw()) {
            al_draw_text(font, al_map_rgb(255, 255, 255),
SCREEN_W / 2.0f, SCREEN_H / 2.0f, ALLEGRO_ALIGN_CENTER,
"It's a Draw!");
        }

        al_flip_display();
    }

    if (font) al_destroy_font(font);
    if (event_queue) al_destroy_event_queue(event_queue);
    if (display) al_destroy_display(display);
    return 0;
}
```

https://doi.org/10.15215/remix/9781998944224.01

TASKS FOR COMPLETION AND IMPROVEMENT

1. **Enhanced Game Logic and Features**
   Win Highlighting: Draw a line or highlight the three winning cells.
   Undo/Redo: Allow players to revert moves for casual play or teaching purposes.
   Replayability: Add a "Play Again" button or automatic board reset.
2. **User Interface (UI) and User Experience (UX) Improvements**
   Mark Previews: Show a faint preview (ghost mark) when hovering over a cell.
   Turn Display: Improve turn indication with stylized icons or animated text.
   Themes and Skins: Add selectable board themes (modern, classic, neon, emoji-based).
3. **Game Modes and Rule Variations**
   AI Opponent: Implement multiple AI difficulty levels (random, defensive, minimax).
   Custom Board Sizes: Allow players to pick 3 × 3, 4 × 4, or 5 × 5 grids.
   Timed Mode: Give each player a limited time per move.
4. **Testing and Quality Assurance**
   Input Validation: Test for out-of-bounds clicks or rapid click inputs.
   Draw Detection: Verify draw logic across all grid sizes if scaling is implemented.
   Device Compatibility: Ensure proper rendering on different screen resolutions.
5. **Polish and Presentation**
   Animations: Animate X/O placement for a smoother visual feel.
   Sound Effects: Add subtle click sounds, win chimes, and draw notifications.
   Game-Over Screen: Add smoother transition effects when declaring the winner.
6. **Final Touches and Optional Extensions**
   Tournament Mode: Track wins across multiple rounds or best-of-three series.
   Online Play: Add turn-based networking for remote two-player matches.
   Statistics: Track player wins, losses, draws, and longest streaks.

https://doi.org/10.15215/remix/9781998944224.01

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Exercises

1. **Board-State Representation**
   Explain how the 3 × 3 grid array represents the game board state. Why is it important to include an explicit NONE state in addition to PLAYER_X and PLAYER_O?
2. **Mouse-Based Input Mapping**
   Describe how mouse click coordinates are converted into grid indices. What assumptions does this conversion make about cell size and screen layout?
3. **Turn Management Logic**
   Explain how the program ensures that Player X and Player O alternate turns correctly. What could go wrong if turn switching were not centralized in one place?
4. **Win Detection Strategy**
   Review the check_winner() function. Why is it sufficient to check rows, columns, and diagonals explicitly for a 3 × 3 Tic-Tac-Toe grid?

Homework Questions

1. **State-Driven Game Logic**
   Discuss how Tic-Tac-Toe demonstrates the idea of a finite game state (playing → won → draw). How does freezing input after the game ends simplify game logic?
2. **User Feedback and Clarity**
   Why is immediate visual feedback (drawing marks instantly, showing win/draw messages) important for board games? What kinds of confusion could arise without it?
3. **Scalability Considerations**
   How would the logic for win detection need to change if the board size were increased from 3 × 3 to 4 × 4 or 5 × 5?
4. **Human vs. AI Play**
   What challenges arise when replacing one human player with an AI opponent? Why might Tic-Tac-Toe be a good introductory problem for implementing game AI?

https://doi.org/10.15215/remix/9781998944224.01

Programming Projects

1. **Win Highlighting (Core Project)**
   Extend the game to visually highlight the three winning cells or draw a line through the winning combination once a player has won.
2. **Replay and Reset Functionality**
   Add a button or key binding (such as R) that resets the board and starts a new round without restarting the program.
3. **Move Undo and Redo**
   Implement undo and redo functionality to allow players to step backward and forward through their move history. This feature is especially useful for teaching and experimentation.
4. **AI Opponent**
   Replace one of the human players with an AI. Begin with a random-move AI, then implement more advanced strategies such as defensive play or the minimax algorithm.
5. **Custom Board Sizes (Advanced)**
   Extend the game to support different board sizes (for example, 4 × 4 or 5 × 5). Update win and draw detection logic accordingly.
6. **Timed Mode (Advanced)**
   Add an optional timer that limits how long each player has to make a move. If time expires, the turn is forfeited or the game ends.

Capstone Project

Design a polished Tic-Tac-Toe application that includes visual win highlighting, sound effects, selectable themes, multiple game modes (human vs. human and human vs. AI), and persistent statistics tracking wins, losses, draws, and streaks. Include a short design reflection explaining how each added feature improves clarity, replayability, or user experience.

8.6. Emulator Game

Emulators are software programs that replicate the hardware and instruction set of older computing or gaming systems, allowing programs originally written for those systems to run unmodified on modern hardware. In this project, we design and implement a CHIP-8 emulator using Allegro 5 in C++.

CHIP-8 is a simple interpreted virtual machine developed in the 1970s and commonly used to teach emulation concepts. Despite its simplicity, it demonstrates all the core ideas behind real emulators,

https://doi.org/10.15215/remix/9781998944224.01

including instruction decoding, memory management, input handling, timing, and graphics output. Many classic games—such as *Pong*, *Breakout*, and *Space Invaders*—were written for CHIP-8 and can be executed directly by the emulator.

8.6.1. Emulator Design

Emulator Concept

Genre: Retro system emulator / virtual machine.

Objective: Implement a functional CHIP-8 emulator capable of loading and running original CHIP-8 game ROMs, faithfully reproducing their behavior, graphics, and input on a modern system.

Emulated System Characteristics:

- *Memory Size:* 4 KB
- *Registers:* 16 general-purpose registers (V0–VF)
- *Program Control:* Program counter and stack
- *Display:* Monochrome 64 × 32 pixel framebuffer
- *Input:* Hexadecimal keypad with 16 keys
- *Timers:* Delay timer and sound timer, updated at 60 Hz

Core Components

CPU (Interpreter): The central processing unit of the emulator.

Attributes: Program counter (PC), index register (I), general-purpose registers (V0–VF), stack, stack pointer

Function: Fetches, decodes, and executes CHIP-8 instructions (opcodes) according to the original specification

Behavior: Each opcode manipulates registers, memory, timers, graphics output, or program control flow

Memory System: Represents the CHIP-8 addressable memory space

Attributes: 4 KB memory array, reserved interpreter region, program memory region starting at address 0x200

Function: Stores program instructions, font sprites, and runtime data used by the emulator

Use: ROM files are loaded into memory and executed directly without modification

Display System: Handles graphical output for CHIP-8 programs

Attributes: 64 × 32 monochrome framebuffer

Rendering Model: Pixels are drawn using XOR logic, allowing sprite erasure and collision detection

https://doi.org/10.15215/remix/9781998944224.01

Collision Flag: A dedicated register flag is set when a sprite draw operation overwrites an existing pixel

Input System: Maps modern keyboard input to CHIP-8 controls

Attributes: 16-key hexadecimal keypad state

Function: Translates keyboard events into CHIP-8 key presses so that original games receive input as intended

Special Behavior: Supports blocking key-wait instructions used by some CHIP-8 programs

Timer System: Maintains consistent timing behavior across all programs

Attributes: Delay timer, sound timer

Update Rate: Both timers decrement at 60 Hz

Function: Controls instruction timing, animations, and sound behavior in accordance with the CHIP-8 specification

Key Techniques and Features to Have

Implementing a CHIP-8 emulator requires faithful reproduction of the original virtual machine's behavior. Although simpler than full console emulators, CHIP-8 demonstrates all core emulator design concepts:

1. **Instruction Fetch-Decode-Execute Cycle**
   The emulator should repeatedly fetch 16-bit opcodes from memory using the program counter, decode them via bit masks and shifts, execute the corresponding instruction, and update the program counter correctly.
2. **Timing and Synchronization**
   CPU execution should run at a controlled rate, while delay and sound timers decrement at 60 Hz using an Allegro timer. CPU execution should be decoupled from rendering to ensure stable timing.
3. **Rendering**
   The 64 × 32 monochrome framebuffer should be scaled to a modern window size and drawn using Allegro primitives or bitmaps. The display should be redrawn only when a draw instruction modifies the framebuffer.
4. **Input Handling**
   Keyboard input should be mapped to the CHIP-8 hexadecimal keypad. The emulator must support blocking key-wait instructions required by some CHIP-8 programs.

https://doi.org/10.15215/remix/9781998944224.01

5. **ROM Loading**
   External CHIP-8 ROM files should be loaded into memory at the correct start address (0x200), and execution should begin from that location without modification.

8.6.2. Implementation

Key Operations in the Emulator

1. **Initialization**
   - Initialize Allegro, keyboard input, display, timer, and event queue.
   - Allocate memory and registers.
   - Load built-in font sprites into memory.
2. **ROM Loading**
   - Read a CHIP-8 ROM from file.
   - Copy it into memory starting at address 0x200.
3. **Main Emulation Loop**
   - Fetch and execute one or more opcodes per tick.
   - Update timers at a fixed rate.
   - Handle input and redraw the display when required.
4. **Opcode Execution**
   - Implement arithmetic, logic, branching, graphics, and input instructions.
   - Maintain correctness with respect to the CHIP-8 specification.
5. **Rendering**
   - Convert the 64 × 32 framebuffer into scaled pixels on the screen.
   - Clear and redraw efficiently.

Structure of the Emulator Program

1. **Initialization**
   - Allegro setup and system Initialization
   - Emulator state allocation
2. **Emulation Loop**
   - Opcode execution
   - Timer updates
   - Event handling
3. **Graphics Rendering**
   - Draw scaled monochrome pixels
   - Refresh display only when necessary
4. **Shutdown**
   - Clean release of Allegro resources

https://doi.org/10.15215/remix/9781998944224.01

Coding the Emulator

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_primitives.h>
#include <iostream>
#include <fstream>
#include <cstring>

// ====================
// CHIP-8 CONSTANTS
// ====================
static const int MEM_SIZE   = 4096;
static const int VIDEO_W    = 64;
static const int VIDEO_H    = 32;
static const int SCALE      = 10;
static const int START_ADDR = 0x200;

// ====================
// CHIP-8 STATE
// ====================
uint8_t  memory[MEM_SIZE];
uint8_t  V[16];              // Registers V0-VF
uint16_t I;                  // Index register
uint16_t pc;                 // Program counter
uint16_t stack[16];
uint8_t  sp;

uint8_t  delay_timer;
uint8_t  sound_timer;

uint8_t  framebuffer[VIDEO_W * VIDEO_H];
bool     keypad[16];

// ====================
// STUDENT TODO FLAGS
// ====================
bool debugMode = false;
int  cycles_per_frame = 8;   // STUDENT TODO: make
configurable
```

https://doi.org/10.15215/remix/9781998944224.01

```
// ====================
// INITIALIZATION
// ====================
void reset() {
    memset(memory, 0, sizeof(memory));
    memset(V, 0, sizeof(V));
    memset(stack, 0, sizeof(stack));
    memset(framebuffer, 0, sizeof(framebuffer));
    memset(keypad, 0, sizeof(keypad));

    pc = START_ADDR;
    I  = 0;
    sp = 0;
    delay_timer = 0;
    sound_timer = 0;

    // TODO (Task 2): Load CHIP-8 fontset into memory
}

// ====================
// ROM LOADING
// ====================
bool loadROM(const char* filename) {
    std::ifstream file(filename, std::ios::binary);
    if (!file) return false;
    file.read((char*)&memory[START_ADDR], MEM_SIZE
- START_ADDR);
    return true;
}

// ====================
// OPCODE EXECUTION
// ====================
void executeCycle() {
    uint16_t opcode = memory[pc] << 8 | memory[pc + 1];
    pc += 2;

    // TODO (Task 2):
    // Decode and execute CHIP-8 opcodes.
```

https://doi.org/10.15215/remix/9781998944224.01

```
    // Implement at least:
    // - CLS (00E0)
    // - JP addr (1NNN)
    // - LD Vx, byte (6XNN)
    // - ADD Vx, byte (7XNN)
    // - DRW Vx, Vy, nibble (DXYN)

    if (debugMode) {
        std::cout << "PC=" << std::hex << pc
                  << " OP=" << opcode << std::dec <<
"\n";
    }
}

// ====================
// DEBUG DRAWING
// ====================
void drawDebugOverlay() {
    // TODO (Task 4):
    // Display register values, PC, I, timers.
    // Optional: display memory or stack contents.
}

// ====================
// MAIN PROGRAM
// ====================
int main(int argc, char** argv) {
    if (argc < 2) {
        std::cerr << "Usage: chip8 <rom>\n";
        return 1;
    }

    // Allegro setup
    al_init();
    al_init_primitives_addon();
    al_install_keyboard();

    ALLEGRO_DISPLAY* display =
        al_create_display(VIDEO_W * SCALE, VIDEO_H *
SCALE);
```

https://doi.org/10.15215/remix/9781998944224.01

```
    ALLEGRO_TIMER* timer = al_create_timer(1.0 / 60.0);
    ALLEGRO_EVENT_QUEUE* queue = al_create_event_queue();

    al_register_event_source(queue,
al_get_display_event_source(display));
    al_register_event_source(queue,
al_get_timer_event_source(timer));
    al_register_event_source(queue,
al_get_keyboard_event_source());

    reset();
    if (!loadROM(argv[1])) {
        std::cerr << "Failed to load ROM\n";
        return 1;
    }

    al_start_timer(timer);

    bool running = true;

    while (running) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(queue, &ev);

        if (ev.type == ALLEGRO_EVENT_TIMER) {

            // ====================
            // CPU EXECUTION
            // ====================
            for (int i = 0; i < cycles_per_frame; ++i) {
                executeCycle();
            }

            // ====================
            // TIMERS
            // ====================
            if (delay_timer > 0) delay_timer--;
            if (sound_timer > 0) sound_timer--;

            // TODO (Task 1):
```

https://doi.org/10.15215/remix/9781998944224.01

```
            // Play sound while sound_timer > 0

            // ====================
            // RENDERING
            // ====================
            al_clear_to_color(al_map_rgb(0, 0, 0));

            for (int y = 0; y < VIDEO_H; ++y) {
                for (int x = 0; x < VIDEO_W; ++x) {
                    if (framebuffer[y * VIDEO_W + x]) {
                        al_draw_filled_rectangle(
                            x * SCALE, y * SCALE,
                            (x + 1) * SCALE, (y + 1) *
SCALE,
                            al_map_rgb(255, 255, 255)
                        );
                    }
                }
            }

            if (debugMode) {
                drawDebugOverlay();
            }

            al_flip_display();
        }
        else if (ev.type == ALLEGRO_EVENT_KEY_DOWN) {

            // ====================
            // INPUT
            // ====================
            if (ev.keyboard.keycode ==
ALLEGRO_KEY_ESCAPE)
                running = false;

            // TODO (Task 3):
            // Increase/decrease cycles_per_frame

            // TODO (Task 4):
            // Toggle debug mode (e.g., F1)
```

https://doi.org/10.15215/remix/9781998944224.01

```
        }
        else if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            running = false;
        }
    }

    al_destroy_event_queue(queue);
    al_destroy_timer(timer);
    al_destroy_display(display);
    return 0;
}
```

TASKS FOR COMPLETION AND IMPROVEMENT

1. **Sound Support:** Add audible feedback when the sound timer is active.
2. **Instruction Accuracy:** Verify emulator behavior against known CHIP-8 test ROMs.
3. **Configurable Speed:** Allow the user to adjust CPU execution speed.
4. **Debug Mode:** Add register and memory inspection tools for learning and testing.

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Exercises

1. **Emulation vs. Game Recreation**
   Explain the difference between writing an emulator and re-implementing a game. Why does running original CHIP-8 ROMs require accurate replication of the original system's behavior?
2. **Instruction Fetch-Decode-Execute Cycle**
   Describe each stage of the fetch-decode-execute cycle used in the CHIP-8 emulator. Why is it important to increment or modify the program counter carefully during opcode execution?
3. **Timers and Synchronization**
   Explain the difference between the CPU execution rate and the 60 Hz delay and sound timers in CHIP-8. What problems can occur if these timers are updated incorrectly?

https://doi.org/10.15215/remix/9781998944224.01

4. **Framebuffer and XOR Drawing**
   Explain why CHIP-8 uses XOR logic when drawing sprites to the framebuffer. How does this approach simplify collision detection?

Homework Questions

1. **Opcode Accuracy and Undefined Behavior**
   Why is opcode accuracy critical for emulator correctness? What types of bugs might appear if even a single opcode is implemented incorrectly?
2. **Input-Mapping Challenges**
   Discuss the challenges involved in mapping a modern keyboard to the CHIP-8 hexadecimal keypad. How can poor input mapping affect game playability?
3. **Rendering Efficiency**
   Why is it beneficial to redraw the display only when the display flag is set, rather than every CPU cycle?
4. **Debugging Emulated Systems**
   Explain how a debug mode that displays registers, memory, and the program counter can help diagnose emulator errors more effectively than logging alone.

Programming Projects

1. **Sound Timer Support (Core Project)**
   Implement sound output that plays whenever the sound timer is greater than zero. Use Allegro's audio system to generate a short beep or tone, and ensure it stops precisely when the timer reaches zero.
2. **Opcode Set Completion**
   Extend the emulator by implementing all remaining CHIP-8 opcodes. Verify correctness by running multiple test ROMs and comparing behavior with known-good emulator results.
3. **Configurable CPU Speed**
   Add keyboard controls that allow the user to increase or decrease the number of CPU cycles executed per frame at runtime. Display the current speed on screen.
4. **Debug Overlay Implementation**
   Implement a toggleable debug overlay that displays register values (V0–VF), the index register, program counter, stack pointer, and timers. Optionally highlight memory regions accessed by the current opcode.

https://doi.org/10.15215/remix/9781998944224.01

5. **ROM Selector Interface (Advanced)**
   Create a simple menu system that allows the user to select and load CHIP-8 ROMs without restarting the program.
6. **Super-CHIP Extension (Advanced)**
   Extend the emulator to support Super-CHIP instructions, including higher-resolution graphics and extended opcodes.

Capstone Project

Develop a polished CHIP-8 emulator application that supports accurate opcode execution, sound, configurable speed, debugging tools, and multiple ROM loading. Include documentation describing how the emulator implements the fetch-decode-execute cycle, timer synchronization, and input mapping. Demonstrate correctness by running at least three classic CHIP-8 games successfully.

8.7. Puzzle Game

Puzzle games are a diverse and engaging genre that challenge players' problem-solving skills, logic, and creativity. They come in various forms and can be enjoyed by players of all ages.

For this project, we will be designing and implementing a simple puzzle game using Allegro 5 in C++: a basic sliding-puzzle game where the player needs to arrange tiles in the correct order.

8.7.1. Game Design

Game Concept

Genre: Puzzle game (sliding puzzle).

Objective: The player arranges shuffled tiles to form a complete image or sequence.

Gameplay Mechanics:

- The player uses the keyboard arrow keys to indicate which adjacent tile to move into the empty black space. For example, right arrow for the tile on the right side of the empty space.
- The empty space swaps positions accordingly.
- Gameplay is complete when the numbered tiles are in order numerically.

Core Components

Puzzle Grid: The structured layout that holds all tiles

Attributes: Grid dimensions (e.g., 4 × 4), cell size, grid coordinates

Function: Defines valid tile locations and constrains movement rules

https://doi.org/10.15215/remix/9781998944224.01

Tiles: Individual movable puzzle pieces
Attributes: Current grid position, correct grid position, identifier (number or image fragment), sprite
Movement: Tiles may only move into the adjacent empty space
Empty Space: The unoccupied grid cell that enables movement
Attributes: Grid position (row, column)
Function: Allows tiles to slide; moves when a tile enters its position
Shuffling System: Initial randomization logic
Function: Rearranges tiles into a solvable starting configuration
Rule: Only valid sliding moves are used to preserve solvability
Puzzle Completion Detection: Determines when the puzzle is solved
Condition: All tiles and the empty space are in their correct positions
Effect: Ends interaction and displays a completion message

Key Techniques and Features to Have

Sliding-tile puzzles depend on clear grid logic, constrained movement, and precise rendering. The following features form a clean and reliable baseline for implementing a sliding puzzle using Allegro 5:

1. **Grid Representation and Tile Management**
   The puzzle board should be represented as a 2D grid where each cell contains either a tile or the empty space. Each tile should track its current position, correct position, and identifier and be drawn based on its grid coordinates.
2. **Shuffling with Valid Puzzle States**
   The initial board state should be randomized while remaining solvable. A common approach is to shuffle the puzzle by performing many legal moves of the empty space rather than random permutations.
3. **Movement Logic and Sliding Mechanics**
   Only tiles adjacent to the empty space should be allowed to move. A valid move swaps the tile and empty space positions and updates the grid consistently. Movement should feel immediate and responsive.
4. **Rendering and Visual Feedback**
   Tiles should be drawn at positions derived from grid coordinates and tile size. The empty cell should be visually distinct, and tiles should clearly display either numbers or image fragments for easy recognition.

https://doi.org/10.15215/remix/9781998944224.01

5. **Win Condition and Completion Detection**
   After each move, the game should check whether all tiles (and the empty space) are in their correct positions. Upon completion, the game should display a clear success message and prevent further moves unless reset.
6. **Input Handling Using Allegro**
   Keyboard or mouse input should trigger tile movement and reset actions. Input should be ignored once the puzzle is solved, except for restart commands.
7. **Game Loop Structure and Smooth Updates**
   An Allegro timer (e.g., 60 fps) should drive consistent updates and rendering. Each frame should clear the screen, draw tiles and UI elements, and flip the display cleanly.
8. **UI Elements and Player Guidance**
   The interface should clearly present tile values or image fragments and may optionally include a move counter, timer, or brief instructions. Fonts and messages should remain readable and unobtrusive.

8.7.2. Implementation

Key Operations in the Game

1. **Initialization**
   - Initialize the Allegro library and its components (image add-on, keyboard input).
   - Create the game window with specified dimensions.
2. **Tile Creation**
   - Each tile has properties like its current position (x, y), its correct position (correctX, correctY), and its image.
   - Generate tiles and assign them their correct positions. Create a bitmap for each tile and fill it with a random colour.
3. **Shuffling**
   - Shuffle the tiles randomly to create the initial puzzle state. *Optional:* Ensure the puzzle is solvable.
4. **Rendering**
   - Draw each tile at its current position on the screen. This is done in the game loop to continuously update the display.
5. **Input Handling**
   - Capture user input to move the tiles. For example, use arrow keys to move the empty space and slide adjacent tiles into it.
6. **Game Logic**
   - Implement logic to move tiles into the empty space when the player presses a key.

https://doi.org/10.15215/remix/9781998944224.01

- Check if the tiles are in the correct order to determine if the player has solved the puzzle.

7. **Game Loop**
   - The game runs in a loop where it continuously checks for user input, updates the game state, and renders the tiles.

Structure of the Game Program

1. **Initialization**
   - Initializes Allegro and its image add-on and installs the keyboard.
2. **Tile Creation**
   - Creates a grid of tiles, each assigned a random colour.
3. **Shuffling**
   - Shuffles the tiles to create the puzzle.
4. **Drawing**
   - Draws the tiles on the screen.
5. **Main Loop of the Game**
6. **Shutdown**
   - Clean release of Allegro resources

Coding the Game

puzzle_game.cpp

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_image.h>
#include <allegro5/allegro_primitives.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>
#include <cstdlib>
#include <string>

//——Config——
const int GRID_SIZE   = 4;
const int TILE_SIZE   = 150;                // 4 × 4 *
150 = 600px square
const int SCREEN_W    = GRID_SIZE * TILE_SIZE;
const int SCREEN_H    = GRID_SIZE * TILE_SIZE;
```

https://doi.org/10.15215/remix/9781998944224.01

```
// A simple colored tile with a number label (1..15).
Empty has no Tile.
struct Tile {
    int row, col;                // current grid location
    int correctRow, correctCol;  // goal location
    int label;                   // 1..15 (goal order
index); not used for empty
    ALLEGRO_BITMAP* image;       // visual for the tile
};

std::vector<Tile> tiles;     // all tiles except the
empty space
int emptyRow = GRID_SIZE-1;
int emptyCol = GRID_SIZE-1;
bool solved = false;

ALLEGRO_FONT* g_font_numbers = nullptr;
ALLEGRO_FONT* g_font_ui = nullptr;

static inline bool in_bounds(int r, int c) {
    return r >= 0 && r < GRID_SIZE && c >= 0 && c <
GRID_SIZE;
}

// Find the index of the tile currently occupying (r,c);
returns-1 if it's empty.
int find_tile_at(int r, int c) {
    for (size_t i = 0; i < tiles.size(); ++i) {
        if (tiles[i].row == r && tiles[i].col == c)
return static_cast<int>(i);
    }
    return-1; // empty cell
}

// Initialization
bool init() {
    if (!al_init()) { std::cerr << "al_init failed\n";
return false; }
    if (!al_init_image_addon()) { std::cerr << "image
addon init failed\n"; return false; }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    if (!al_install_keyboard()) { std::cerr << "keyboard
install failed\n"; return false; }
    if (!al_init_primitives_addon()) { std::cerr <<
"primitives addon init failed\n"; return false; }
    al_init_font_addon();
    al_init_ttf_addon();
    std::srand(static_
cast<unsigned>(std::time(nullptr)));
    return true;
}

void draw_number_centered(ALLEGRO_BITMAP* bmp, ALLEGRO_
FONT* font, int n, ALLEGRO_COLOR color) {
    ALLEGRO_BITMAP* back = al_get_target_bitmap();
    al_set_target_bitmap(bmp);
    std::string s = std::to_string(n);
    int w = al_get_text_width(font, s.c_str());
    int h = al_get_font_line_height(font);
    int x = TILE_SIZE/2-w/2;
    int y = TILE_SIZE/2-h/2;
    al_draw_text(font, color, x, y, 0, s.c_str());
    al_set_target_bitmap(back);
}

// Tile Creation
void create_tiles() {
    tiles.clear();
    emptyRow = GRID_SIZE-1;
    emptyCol = GRID_SIZE-1;
    solved = false;

    // create 15 tiles in goal order, positioned at their
correct locations
    int label = 1;
    for (int r = 0; r < GRID_SIZE; ++r) {
        for (int c = 0; c < GRID_SIZE; ++c) {
            if (r == emptyRow && c == emptyCol) continue;
// skip empty 16th cell
            Tile t;
            t.row = r; t.col = c;
```

https://doi.org/10.15215/remix/9781998944224.01

```
            t.correctRow = r; t.correctCol = c;
            t.label = label++;
            t.image = al_create_bitmap(TILE_SIZE,
TILE_SIZE);
            if (!t.image) { std::cerr << "Failed to
create tile bitmap\n"; continue; }

            ALLEGRO_BITMAP* back =
al_get_target_bitmap();
            al_set_target_bitmap(t.image);
            // pleasant pseudo-random color
(deterministic per label for consistency):
            std::srand(t.label * 2654435761u); // mix a
bit
            ALLEGRO_COLOR fill = al_map_rgb(70
+ std::rand()%160, 70 + std::rand()%160, 70 +
std::rand()%160);
            std::srand(static_
cast<unsigned>(std::time(nullptr))); // restore
randomness for shuffle later

            al_clear_to_color(fill);
            // border
            al_draw_filled_rectangle(0, 0, TILE_SIZE, 6,
al_map_rgb(0,0,0));
            al_draw_filled_rectangle(0, TILE_SIZE-6,
TILE_SIZE, TILE_SIZE, al_map_rgb(0,0,0));
            al_draw_filled_rectangle(0, 0, 6, TILE_SIZE,
al_map_rgb(0,0,0));
            al_draw_filled_rectangle(TILE_SIZE-6, 0,
TILE_SIZE, TILE_SIZE, al_map_rgb(0,0,0));
            // number
            draw_number_centered(t.image, g_font_numbers,
t.label, al_map_rgb(255,255,255));
            al_set_target_bitmap(back);

            tiles.push_back(t);
        }
    }
}
```

https://doi.org/10.15215/remix/9781998944224.01

```
// Perform one legal move by sliding a neighbor into the
empty
bool do_random_move() {
    int candidates[4][2] = {
        { emptyRow-1, emptyCol }, // tile above empty
moves down
        { emptyRow+1, emptyCol }, // tile below empty
moves up
        { emptyRow, emptyCol-1 }, // left of empty moves
right
        { emptyRow, emptyCol+1 }  // right of empty moves
left
    };
    std::vector<std::pair<int,int>> opts;
    for (auto& cand : candidates) {
        if (in_bounds(cand[0], cand[1])) opts.emplace_
back(cand[0], cand[1]);
    }
    if (opts.empty()) return false;
    auto pick = opts[std::rand() % opts.size()];
    int idx = find_tile_at(pick.first, pick.second);
    if (idx >= 0) {
        tiles[idx].row = emptyRow;
        tiles[idx].col = emptyCol;
        emptyRow = pick.first;
        emptyCol = pick.second;
        return true;
    }
    return false;
}

// Shuffling
void shuffle_tiles(int moves = 400) {
    for (int i = 0; i < moves; ++i) {
        (void)do_random_move();
    }
}

// Check if every tile is back in its correct row/col and
empty at bottom-right
```

https://doi.org/10.15215/remix/9781998944224.01

```
bool is_solved() {
    if (!(emptyRow == GRID_SIZE-1 && emptyCol ==
GRID_SIZE-1)) return false;
    for (const auto& t : tiles) {
        if (t.row != t.correctRow || t.col !=
t.correctCol) return false;
    }
    return true;
}

// Drawing
void draw_tiles() {
    al_clear_to_color(al_map_rgb(30, 30, 30));
    for (const auto& t : tiles) {
        int x = t.col * TILE_SIZE;
        int y = t.row * TILE_SIZE;
        al_draw_bitmap(t.image, x, y, 0);
    }
    // empty cell outline
    int ex = emptyCol * TILE_SIZE;
    int ey = emptyRow * TILE_SIZE;
    al_draw_rectangle(ex+4, ey+4, ex + TILE_SIZE-4, ey +
TILE_SIZE-4, al_map_rgb(255, 255, 255), 2);

    if (solved) {
        const char* msg = "Solved!";
        int w = al_get_text_width(g_font_ui, msg);
        int h = al_get_font_line_height(g_font_ui);
        int x = SCREEN_W/2-w/2;
        int y = SCREEN_H/2-h/2;
        al_draw_filled_rectangle(x-16, y-10, x+w+16,
y+h+10, al_map_rgba(0,0,0,200));
        al_draw_text(g_font_ui, al_map_rgb(255, 255,
255), SCREEN_W/2, SCREEN_H/2, ALLEGRO_ALIGN_CENTER,
msg);
    }
}

// Input handling: slide a tile into the empty cell
void handle_keypress(int keycode) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
    if (solved) return; // ignore input after solved
(optional)
    int srcR = emptyRow, srcC = emptyCol;
    if (keycode == ALLEGRO_KEY_UP)    srcR = emptyRow +
1; // tile below empty moves up
    if (keycode == ALLEGRO_KEY_DOWN)  srcR = emptyRow-1;
// tile above empty moves down
    if (keycode == ALLEGRO_KEY_LEFT)  srcC = emptyCol +
1; // tile right of empty moves left
    if (keycode == ALLEGRO_KEY_RIGHT) srcC = emptyCol-1;
// tile left of empty moves right

    if (!in_bounds(srcR, srcC)) return;
    int idx = find_tile_at(srcR, srcC);
    if (idx < 0) return;
    tiles[idx].row = emptyRow;
    tiles[idx].col = emptyCol;
    emptyRow = srcR;
    emptyCol = srcC;

    solved = is_solved();
}

// Main loop
int main() {
    if (!init()) return-1;

    ALLEGRO_DISPLAY* display = al_create_
display(SCREEN_W, SCREEN_H);
    if (!display) { std::cerr << "display creation
failed\n"; return-1; }

    // Load fonts (numbers/UI). If TTF load fails, fall
back to builtin.
    g_font_numbers = al_load_ttf_font("arial.ttf", 64,
0);
    if (!g_font_numbers) g_font_numbers =
al_create_builtin_font();
    g_font_ui = al_load_ttf_font("arial.ttf", 64, 0);
    if (!g_font_ui) g_font_ui = al_create_builtin_font();
```

https://doi.org/10.15215/remix/9781998944224.01

```
    ALLEGRO_EVENT_QUEUE* queue = al_create_event_queue();
    ALLEGRO_TIMER* timer = al_create_timer(1.0 / 60.0);
    if (!queue || !timer) {
        std::cerr << "event queue or timer creation
failed\n";
        if (timer) al_destroy_timer(timer);
        if (queue) al_destroy_event_queue(queue);
        if (g_font_numbers)
al_destroy_font(g_font_numbers);
        if (g_font_ui) al_destroy_font(g_font_ui);
        al_destroy_display(display);
        return -1;
    }

    al_register_event_source(queue,
al_get_display_event_source(display));
    al_register_event_source(queue,
al_get_keyboard_event_source());
    al_register_event_source(queue,
al_get_timer_event_source(timer));

    create_tiles();
    shuffle_tiles();
    al_start_timer(timer);

    bool running = true;
    while (running) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(queue, &ev);
        switch (ev.type) {
            case ALLEGRO_EVENT_DISPLAY_CLOSE:
                running = false;
                break;
            case ALLEGRO_EVENT_KEY_DOWN:
                if (ev.keyboard.keycode == ALLEGRO_KEY_R)
{
                    // quick reset/shuffle
                    create_tiles();
                    shuffle_tiles();
                } else {
```

https://doi.org/10.15215/remix/9781998944224.01

```
                    handle_keypress(ev.keyboard.keycode);
                }
                break;
            case ALLEGRO_EVENT_TIMER:
                draw_tiles();
                al_flip_display();
                break;
        }
    }

    // Finishing up
    for (auto& t : tiles) {
        if (t.image) al_destroy_bitmap(t.image);
    }
    if (g_font_numbers) al_destroy_font(g_font_numbers);
    if (g_font_ui) al_destroy_font(g_font_ui);
    al_destroy_timer(timer);
    al_destroy_event_queue(queue);
    al_destroy_display(display);
    return 0;
}
```

TASKS FOR COMPLETION AND IMPROVEMENT

1. **Scoring System:** Add visual feedback for scoring, such as displaying the score on the screen in regard to the number of moves it took to solve the puzzle.
2. **Sound Effects:** Add sound effects when moving a tile.
3. **Difficulty Levels:** Introduce different difficulty levels by varying the number of tiles to move—for example, change the matrix size to 3 × 3, 4 × 5, and so on.

https://doi.org/10.15215/remix/9781998944224.01

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Exercises

1. **Grid Representation and Tile Location**
   Explain how the two-dimensional grid representation simplifies both the movement logic and rendering of puzzle tiles. Why is it useful to separate a tile's *current position* from its *correct position*?
2. **Movement Constraints**
   Describe why only tiles adjacent to the empty space are allowed to move. How does enforcing this rule guarantee valid puzzle interactions?
3. **Solvability Considerations**
   Explain why random permutations of tiles may result in unsolvable puzzles. How does shuffling the puzzle by performing many legal moves avoid this problem?
4. **Win Condition Detection**
   Review the is_solved() function. Why is it important to check both the tile positions and the location of the empty space?

Homework Questions

1. **User Experience in Puzzle Games**
   Discuss how clear visual feedback (such as tile borders, numbered labels, and empty-space outlines) affects the player's ability to understand and solve the puzzle.
2. **Keyboard vs. Mouse Control**
   Compare keyboard-based tile movement with mouse-based interaction (clicking tiles to slide them). What are the advantages and disadvantages of each input method?
3. **Difficulty and Cognitive Load**
   How does increasing the grid size affect the puzzle's difficulty? At what point might a larger grid become frustrating rather than challenging?
4. **Puzzle Games and Replayability**
   Why are simple puzzle games often still highly replayable? Discuss the role of randomness, difficulty scaling, and performance tracking (time or moves).

https://doi.org/10.15215/remix/9781998944224.01

Programming Projects

1. **Move Counter and Scoring System (Core Project)**
   Add a move counter that increments each time a tile is slid. Display the number of moves on the screen and show the final move count when the puzzle is solved.
2. **Sound Effects Integration**
   Add a sound effect that plays whenever a tile moves. Optionally include a different sound for invalid moves or when the puzzle is completed.
3. **Difficulty Levels**
   Extend the game to support different grid sizes, such as 3 × 3, 4 × 4, or 5 × 5. Allow the player to select the difficulty level before starting or restarting the puzzle.
4. **Timer-Based Challenge Mode (Advanced)**
   Add a timer that tracks how long the player takes to solve the puzzle. Display the elapsed time and use it as an alternative scoring metric alongside the move counter.
5. **Image-Based Puzzle (Advanced)**
   Replace the coloured tiles with fragments of an image. Load a single image, divide it into tile-sized sections, and render each fragment on the corresponding tile.
6. **Animated Tile Sliding (Optional)**
   Add smooth animations so tiles slide into the empty space rather than snapping instantly. Interpolate tile positions over a short duration to improve visual polish.

Capstone Project (Optional)

Design a polished sliding-puzzle game that includes adjustable difficulty, scoring by moves and time, sound effects, and graphical enhancements. Include a menu screen for selecting puzzle size and restarting the game. Submit a short reflection describing how changes in grid size, feedback, and presentation influence the player's problem-solving experience.

8.8. Role-Playing Game

Role-playing games (RPGs) are a genre where players assume the roles of characters in a fictional setting. Players take control of these characters and guide them through various quests, battles, and storylines. Creating a role-playing game (RPG) with Allegro 5 in C++ is an exciting project! RPGs are complex in terms of the number of elements that need to be considered; there are often multiple

https://doi.org/10.15215/remix/9781998944224.01

enemy types, weapons, locations, quests, treasures, and so forth. As such, here, we'll cover only the outline of a possible basic design and some implementation steps for a simple RPG—a basic framework that you can expand upon.

8.8.1. Game Design

Game Concept

Genre: Role-playing game (RPG).

Objective: The player controls a character who explores the game world, interacts with NPCs (nonplayer characters), and battles enemies to complete quests and progress through the story.

Core Components

Player Character: The main controllable protagonist

Attributes: Position (x, y), HP, attack, defense, level, experience points (XP)

Movement: Controlled via keyboard input (e.g., WASD or arrow keys)

Enemies: Hostile entities that challenge the player

Attributes: Position, HP, combat stats, sprite

Behavior: Move and attack using simple AI patterns

Nonplayer Characters (NPCs): Interactive characters within the world

Attributes: Dialogue state, quest flags, position

Function: Provide information, quests, items, or services

Quests and Story System: Tracks narrative progress

Attributes: Quest states (inactive, active, completed), quest stages

Function: Unlocks content and guides player progression

Inventory System: Manages collected items

Attributes: Item slots, item IDs, quantities

Function: Supports consumables, equipment, and key items

Combat System: Handles battles with enemies

Type: Turn-based combat

Function: Resolves attacks, abilities, damage, and rewards

Key Techniques and Features to Have

Designing a role-playing game (RPG) requires coordinating exploration, interaction, combat, progression, and presentation into a cohesive system. The following features provide a clear, beginner-friendly foundation for an extensible RPG built with Allegro 5:

1. **World Navigation and Collision**
   The game world should use tile-based movement with clearly defined walkable and blocked areas. Collision layers should distinguish static

https://doi.org/10.15215/remix/9781998944224.01

terrain from interactive elements such as doors, NPCs, and triggers. The camera should follow the player and clamp to map boundaries.

2. **Player Controller and Stats**
   Player movement should be driven by keyboard input, with a dedicated interaction key. Core statistics such as HP, ATK, DEF, level, and XP should be tracked and updated throughout play, with hooks for future status effects.
3. **NPCs, Dialogue, and Choices**
   NPC interaction should trigger dialogue based on proximity or facing direction. Dialogue systems should support branching choices and simple conditions, and dialogue outcomes should connect to quests, items, or stat changes.
4. **Quests and World Triggers**
   A quest log should track quest states and progression stages. World triggers—such as entering areas, activating switches, or defeating enemies—should advance quests and control access to content based on progress or inventory.
5. **Inventory, Items, and Equipment**
   The inventory should support consumables, equipment, and key items. Consumables apply direct effects, while equipment optionally modifies player stats. Item usage should tie directly into puzzle-solving and progression.
6. **Combat System (Turn-Based Baseline)**
   Combat should use a clear turn order and a simple damage model (e.g., ATK − DEF). The system should support basic attacks, limited abilities, and defined win/loss outcomes, awarding XP, currency, and loot on victory.
7. **Progression and Balance**
   Player progression should follow a tunable XP curve with stat increases on level-up. Enemies should use drop tables for rewards, and difficulty should scale gradually through enemy stats and encounter design.
8. **Rendering and UI**
   Rendering should follow a consistent draw order: world, entities, overlays, then UI. The HUD should display key player information such as HP, level, XP, currency, and active objectives. Dialogue UI should be readable and choice-aware.
9. **Audio and Feedback**
   Sound effects and background music should reinforce actions, locations, and combat. Key events—attacks, healing, level-ups—should provide clear audio or visual feedback, using subtle effects for emphasis.

https://doi.org/10.15215/remix/9781998944224.01

10. **Data-Driven and Modular Structure**
    Maps, items, enemies, dialogue, and quests should be externalized into data files for easy iteration. The codebase should be modular, separating systems such as Map, Combat, Dialogue, and Inventory, with basic save/load support for persistent play.

8.8.2. Implementation

Key Operations in the Game

The following operations constitute the core runtime flow of a simple Allegro-based RPG. They map directly to the systems listed above and can be implemented incrementally.

1. **Input Collection and Intent Mapping**
   - Read Allegro keyboard events → set intent flags (move up/down/left/right, interact, open inventory, confirm/cancel).
   - Debounce menu inputs and prevent repeats inside a single frame where needed.
2. **Player Movement and Collision Resolution**
   - Compute candidate position from input; test against tile collision mask.
   - Accept move if walkable; otherwise, slide or cancel.
   - Update camera to follow the player and clamp to map edges.
3. **Interaction Detection**
   - On interact key press, raycast/check the front tile for NPC, sign, door, or chest.
   - If found, open the correct UI: dialogue box, treasure, or door transition.
4. **Dialogue Execution**
   - Load the dialogue node, render text and choices, and accept selection input.
   - Apply effects (start/advance quest, grant item, play SFX) when a node completes.
5. **Quest State Transitions**
   - When a trigger fires (e.g., entering a room, defeating an enemy, acquiring an item), update the quest log.
   - Refresh the on-screen objective text and enable/disable relevant world gates.
6. **Inventory Operations**
   - Open inventory UI; navigate slots, use items, and update stats (e.g., heal HP).

https://doi.org/10.15215/remix/9781998944224.01

- Apply item rules (consumable vs. equipment vs. key) and enforce stack counts.

7. **Encounter Handling (Overworld → Battle)**
   - Detect battle start (random encounter tile, scripted boss trigger, or touching an enemy).
   - Push a combat state and pause overworld updates until combat ends.
8. **Combat Turn Cycle**
   - Establish turn order; in the player's turn, accept action selection (attack, skill, item).
   - Compute damage/heal with the current formula and update HP/MP.
   - Check victory/defeat; on victory, award XP/loot and return to overworld.
9. **Progression Updates**
   - Add XP; if threshold reached, level up, raise stats, and show a level-up banner/SFX.
   - *Optional:* Unlock new abilities at specific levels.
10. **Map Transitions and Scene Management**
    - On door/warp tiles, save the player's exit point, switch the current map, and place the player at the entry point.
    - Reload map layers, entities, and background music as needed.
11. **Rendering Pipeline per Frame**
    - Clear → draw tile map (visible region) → draw entities → draw overlays (e.g., highlights) → draw HUD/Dialog/menus → flip display.
    - Respect draw order for clarity (UI on top).
12. **Audio Triggers**
    - Play loops for BGM; trigger one-shots for actions (confirm, attack, item use, level up).
    - Fade or switch tracks on map/scene changes.
13. **Saving and Loading**
    - Serialize player stats, position, inventory, quest flags, and map identifier to a save file.
    - On load, restore state and assets, then resume at the saved location.
14. **Performance and Housekeeping**
    - Avoid per-frame allocations in hot paths; reuse containers/buffers.
    - Destroy Allegro resources (bitmaps, fonts, audio) on shutdown; guard against nulls.

https://doi.org/10.15215/remix/9781998944224.01

Structure of the Game Program

1. **Initialization**
   - Initializes Allegro and its add-ons (image, font, audio, primitives, etc.) and installs input devices.
2. **Resource Loading**
   - Loads sprites, tilesets, fonts, audio, and other assets.
3. **Entity Creation**
   - Creates the player, NPCs, and enemies with attributes and positions.
4. **Map/Level System**
   - Loads tile-based maps, manages environment layout, collision layers, and triggers for interactions (doors, transitions, scripted events).
5. **Drawing**
   - Renders the player, NPCs, enemies, environment, and UI elements on the screen.
6. **Input Handling**
   - Captures keyboard (and optional mouse/controller) input for movement, menus, combat, and interactions.
7. **Game Loop**
   - Runs the main cycle (input → update → draw) to drive the game.
8. **Dialogue System**
   - Displays NPC conversations with branching text, choices, and conditional outcomes (e.g., starting quests, giving items).
9. **Quest Tracking**
   - Maintains active and completed quests, quest stages, and conditions for advancement. Connects to dialogue and world triggers.
10. **Inventory System**
    - Allows managing items, equipment, and consumables. Includes UI for viewing and using items.
11. **Combat System**
    - Handles battle flow (turn-based or real time), damage formulas, health management, and special abilities.
12. **Progression System**
    - Tracks experience points (XP), leveling up, and stat growth for the player and possibly NPC allies.

https://doi.org/10.15215/remix/9781998944224.01

Coding the Game

```
// ================ 1) Initialization ================
struct App {
    ALLEGRO_DISPLAY* display = nullptr;
    ALLEGRO_EVENT_QUEUE* queue = nullptr;
    ALLEGRO_TIMER* timer = nullptr;
    bool running = true;

    bool init(int w=1280, int h=720, int fps=60) {
        if (!al_init()) return false;
        al_install_keyboard();
        al_init_image_addon();
        al_init_font_addon();
        al_init_ttf_addon();
        al_install_audio();
        al_init_primitives_addon();
        display = al_create_display(w, h);
        timer = al_create_timer(1.0 / fps);
        queue = al_create_event_queue();
        al_register_event_source(queue,
al_get_display_event_source(display));
        al_register_event_source(queue,
al_get_timer_event_source(timer));
        al_register_event_source(queue,
al_get_keyboard_event_source());
        al_start_timer(timer);
        return display && timer && queue;
    }

    void shutdown() {
        if (queue) al_destroy_event_queue(queue);
        if (timer) al_destroy_timer(timer);
        if (display) al_destroy_display(display);
        al_uninstall_audio();
    }
};

// ================ 2) Resource Loading ================
struct Assets {
```

https://doi.org/10.15215/remix/9781998944224.01

```
    std::unordered_map<std::string, ALLEGRO_BITMAP*>
bitmaps;
    std::unordered_map<std::string, ALLEGRO_FONT*>
fonts;
    // SFX/BGM maps omitted for brevity.

    ALLEGRO_BITMAP* bmp(const std::string& id) { return
bitmaps[id]; }
    ALLEGRO_FONT* font(const std::string& id) { return
fonts[id]; }

    bool load() {
        bitmaps["player"] = al_load_bitmap("assets/
player.png");
        bitmaps["tiles"]  = al_load_bitmap("assets/tiles.
png");
        fonts["ui"]       = al_load_ttf_font("assets/
ui.ttf", 18, 0);
        // Check for nulls in real code.
        return true;
    }
    void unload() {
        for (auto& [k,v] : bitmaps) if (v)
al_destroy_bitmap(v);
        for (auto& [k,v] : fonts) if (v)
al_destroy_font(v);
    }
};

// ================ 3) Entity Creation ================
struct Stats { int hp=10, mp=0, atk=2, def=1, lvl=1,
xp=0; };
struct Entity {
    float x=0, y=0;
    ALLEGRO_BITMAP* sprite=nullptr;
    Stats stats;
    bool solid=true;
};
```

https://doi.org/10.15215/remix/9781998944224.01

```
struct WorldEntities {
    Entity player;
    std::vector<Entity> enemies;
    std::vector<Entity> npcs;
    void create(Assets& A) {
        player.sprite = A.bmp("player");
        player.x = 100; player.y = 100;
        enemies.push_back(Entity{300, 180, A.bmp("enemy_
slime"), Stats{6,0,1,0,1,0}, true});
        // Add NPCs similarly . . .
    }
};

// ================ 4) Map / Level System ===============
struct TileMap {
    int w=0, h=0, tile=32;
    std::vector<int> tiles;           // layer 0 visual
    std::vector<uint8_t> solidMask;   // 1 = blocked
    ALLEGRO_BITMAP* tileset=nullptr;

    bool load(const std::string& path, Assets& A) {
        // Minimal: pretend a single small room.
        w=40; h=25; tile=32;
        tiles.assign(w*h, 1);          // tile id 1
        solidMask.assign(w*h, 0);      // all walkable
for now
        tileset = A.bmp("tiles");
        return true;
    }

    bool blocked(float px, float py) const {
        int cx = int(px) / tile, cy = int(py) / tile;
        if (cx<0||cy<0||cx>=w||cy>=h) return true;
        return solidMask[cy*w+cx] != 0;
    }

    void draw(float camx, float camy) const {
        // Ultra-minimal: draw visible region (no culling
shown).
        for (int y=0; y<h; ++y)
```

https://doi.org/10.15215/remix/9781998944224.01

```
            for (int x=0; x<w; ++x) {
                int id = tiles[y*w+x];
                // Assume id maps to tileset cell (0,0).
Replace with atlas logic later.
                al_draw_bitmap_region(tileset, 0,0,
tile,tile, x*tile-camx, y*tile-camy, 0);
            }
    }
};

// ==================== 5) Drawing ====================
struct Renderer {
    void drawWorld(const TileMap& map, const
WorldEntities& E, float camx, float camy) {
        map.draw(camx, camy);
        if (E.player.sprite)
            al_draw_bitmap(E.player.sprite,
E.player.x-camx, E.player.y-camy, 0);
        for (auto& e : E.enemies)
            if (e.sprite) al_draw_bitmap(e.sprite,
e.x-camx, e.y-camy, 0);
        // NPCs, particles, etc.
    }
    void drawUI(Assets& A, const std::string& msg) {
        al_draw_text(A.font("ui"), al_map_
rgb(255,255,255), 16, 16, 0, msg.c_str());
    }
};

// ================= 6) Input Handling =================
struct Input {
    bool up=false, down=false, left=false, right=false,
interact=false, inventory=false, quit=false;
    void handle(const ALLEGRO_EVENT& ev) {
        if (ev.type == ALLEGRO_EVENT_KEY_DOWN || ev.type
== ALLEGRO_EVENT_KEY_UP) {
            bool downEv = (ev.type ==
ALLEGRO_EVENT_KEY_DOWN);
            switch (ev.keyboard.keycode) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
                case ALLEGRO_KEY_W: case ALLEGRO_KEY_
UP:    up = downEv; break;
                case ALLEGRO_KEY_S: case ALLEGRO_KEY_
DOWN:  down = downEv; break;
                case ALLEGRO_KEY_A: case ALLEGRO_KEY_
LEFT:  left = downEv; break;
                case ALLEGRO_KEY_D: case ALLEGRO_KEY_
RIGHT: right = downEv; break;
                case ALLEGRO_
KEY_E:                          interact = downEv; break;
                case ALLEGRO_
KEY_I:                          inventory = downEv;
break;
                case ALLEGRO_KEY_
ESCAPE:                     quit = downEv; break;
            }
        }
    }
};

// ============== 7) Game Loop (State-lite) =============
struct Game {
    App app;
    Assets assets;
    Renderer renderer;
    Input input;
    TileMap map;
    WorldEntities ents;
    float camx=0, camy=0;

    bool boot() {
        if (!app.init()) return false;
        assets.load();
        assets.bitmaps["enemy_slime"] = al_load_
bitmap("assets/enemy_slime.png");
        map.load("assets/maps/start.json", assets);
        ents.create(assets);
        return true;
    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    void update(double dt) {
        // Simple movement with collision
        float speed = 120.0f;
        float dx = (input.right—input.left) * speed * dt;
        float dy = (input.down—input.up  ) * speed * dt;
        float nx = ents.player.x + dx, ny = ents.player.y
+ dy;
        if (!map.blocked(nx, ents.player.y)) ents.
player.x = nx;
        if (!map.blocked(ents.player.x, ny)) ents.
player.y = ny;
        // Camera follows
        camx = ents.player.x—640; camy =
ents.player.y—360;
    }

    void draw() {
        al_clear_to_color(al_map_rgb(0,0,0));
        renderer.drawWorld(map, ents, camx, camy);
        renderer.drawUI(assets, "HP: " + std::to_
string(ents.player.stats.hp));
        al_flip_display();
    }

    void run() {
        while (app.running) {
            ALLEGRO_EVENT ev; al_wait_for_event(app.
queue, &ev);
            if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE)
app.running = false;
            if (ev.type == ALLEGRO_EVENT_TIMER) {
update(1.0/60.0); draw(); }
            if (ev.type == ALLEGRO_EVENT_KEY_DOWN ||
ev.type == ALLEGRO_EVENT_KEY_UP) {
                input.handle(ev);
                if (input.quit) app.running = false;
            }
        }
    }
};
```

https://doi.org/10.15215/remix/9781998944224.01

```
// ================== 8) Dialogue System ================
struct DialogueChoice { std::string text; std::string
nextId; };
struct DialogueNode {
    std::string id;
    std::string text;
    std::vector<DialogueChoice> choices; // empty =>
press to continue
    // Minimal: conditions/effects omitted; add callbacks
later.
};

struct DialogueDB {
    std::unordered_map<std::string, DialogueNode> nodes;
    const DialogueNode* get(const std::string& id) const
{
        auto it = nodes.find(id); return (it==nodes.
end())? nullptr : &it->second;
    }
};

struct DialogueRunner {
    const DialogueDB* db=nullptr;
    const DialogueNode* cur=nullptr;
    bool active=false;

    void start(const DialogueDB* d, const std::string&
id) { db=d; cur=d->get(id); active = (cur!=nullptr); }
    void choose(size_t idx) {
        if (!active || !cur) return;
        if (idx < cur->choices.size()) {
            cur = db->get(cur->choices[idx].nextId);
            active = (cur != nullptr);
        } else { active = false; }
    }
};

// ================== 9) Quest Tracking =================
enum class QuestState { Inactive, Active, Completed,
Failed };
```

https://doi.org/10.15215/remix/9781998944224.01

```
struct Quest {
    std::string id;
    int stage=0;            // 0..N
    QuestState state=QuestState::Inactive;
};
struct QuestLog {
    std::unordered_map<std::string, Quest> quests;
    void start(const std::string& id) { quests[id] =
Quest{id,0,QuestState::Active}; }
    void advance(const std::string& id) { auto&
q=quests[id]; if (q.state==QuestState::Active) ++q.stage;
}
    void complete(const std::string& id) { quests[id].
state = QuestState::Completed; }
    bool isActive(const std::string& id) const {
        auto it=quests.find(id); return it!=quests.end()
&& it->second.state==QuestState::Active;
    }
};

// ================ 10) Inventory System ===============
enum class ItemType { Consumable, Equipment, Key };
struct ItemDef {
    int id; std::string name; ItemType type; int power=0;
// power = heal amount or atk bonus, etc.
};
struct ItemDB {
    std::unordered_map<int, ItemDef> defs;
    const ItemDef& get(int id) const { return defs.
at(id); }
};

struct Inventory {
    struct Slot { int itemId=-1; int count=0; };
    std::vector<Slot> slots = std::vector<Slot>(24);
    bool add(int itemId, int n=1) {
        for (auto& s: slots) if (s.itemId==itemId) {
s.count+=n; return true; }
        for (auto& s: slots) if (s.itemId==-1) {
s.itemId=itemId; s.count=n; return true; }
```

https://doi.org/10.15215/remix/9781998944224.01

```
        return false;
    }
    bool use(int idx, Entity& target, const ItemDB& DB) {
        if (idx<0 || idx>= (int)slots.size()) return false;
        auto& s = slots[idx]; if (s.itemId<0 ||
s.count<=0) return false;
        const ItemDef& def = DB.get(s.itemId);
        if (def.type == ItemType::Consumable) {
            target.stats.hp = std::min(target.stats.hp +
def.power, 999);
            if (—s.count==0) s.itemId=-1;
            return true;
        }
        // Equipment/Key handling omitted in minimal
sketch.
        return false;
    }
};

// ======== 11) Combat System (Turn-based minimal) ======
struct Combatant { Entity* ref=nullptr; bool enemy=false;
};
struct Combat {
    std::vector<Combatant> order;
    size_t turnIndex=0;
    bool active=false;

    void start(Entity& player, std::vector<Entity>&
enemies) {
        order.clear();
        order.push_back({&player,false});
        for (auto& e: enemies) order.
push_back({&e,true});
        turnIndex = 0; active = true;
    }
    void playerAttack(int targetIdx) {
        if (!active) return;
        auto& you = *order[0].ref;
        auto& tgt = *order[targetIdx].ref;
```

https://doi.org/10.15215/remix/9781998944224.01

```
        int dmg = std::max(1, you.stats.atk–tgt.stats.
def);
        tgt.stats.hp–= dmg;
        if (tgt.stats.hp <= 0) {/* mark defeated */}
        endTurn();
    }
    void enemyTurnAI() {
        auto& foe = *order[turnIndex].ref;
        auto& you = *order[0].ref;
        int dmg = std::max(1, foe.stats.atk–you.stats.
def);
        you.stats.hp–= dmg;
        endTurn();
    }
    void endTurn() {
        turnIndex = (turnIndex + 1) % order.size();
        // End combat on victory/defeat checks (omitted
here).
    }
};

// ================ 12) Progression System =============
struct Progression {
    // Simple linear XP curve for demo.
    int xpToNext(int lvl) const { return 10 + lvl*10; }
    bool grantXP(Entity& e, int xp) {
        e.stats.xp += xp;
        bool leveled=false;
        while (e.stats.xp >= xpToNext(e.stats.lvl)) {
            e.stats.xp–= xpToNext(e.stats.lvl);
            ++e.stats.lvl; leveled=true;
            e.stats.hp += 3; e.stats.atk += 1; e.stats.
def += 1;
        }
        return leveled;
    }
};

// ================== Wiring it together ===============
int main() {
```

https://doi.org/10.15215/remix/9781998944224.01

```
    Game game;
    if (!game.boot()) return 1;

    // Build minimal data for Dialogue/Quest/Items and
wire events as needed.
    DialogueDB ddb;
    ddb.nodes["hello"] = DialogueNode{
        "hello", "Hello, traveler!", { { "Who are you?",
"about" }, { "Goodbye.", "" } }
    };
    ddb.nodes["about"] = DialogueNode{
        "about", "I'm the mayor. Please clear the slimes
in the well.",
        { { "Accept quest", "" } }
    };
    DialogueRunner dlg;
    QuestLog quests;
    ItemDB itemdb; itemdb.defs[1] = ItemDef{1,"Potion",
ItemType::Consumable, 10};
    Inventory inv; inv.add(1, 2);

    // Example "interact" flow (pseudo):
    // if (player presses E near mayor) dlg.start(&ddb,
"hello");
    // if (player chooses "Accept quest") quests.
start("clear_slimes");

    game.run();
    game.app.shutdown();
    return 0;
}
```

https://doi.org/10.15215/remix/9781998944224.01

TASKS FOR COMPLETION AND IMPROVEMENT

1. **Combat Mechanics:** Although a basic turn-based combat scaffold exists in the Combat struct, it currently lacks full resolution logic. Students should complete the combat system by defining win/loss conditions, handling defeated enemies, and integrating combat flow into the main game loop.
2. **NPC Interactions:** NPC entities are present, and a dialogue system exists, but interactions are not yet fully wired. Students should implement logic that allows the player to interact with nearby NPCs and initiate dialogue sequences.
3. **Inventory Management:** The Inventory and ItemDB systems support adding and using items, but there is no UI or interaction logic yet. Students should expand inventory management to allow browsing items, selecting them, and applying their effects during gameplay.
4. **More Complex Game Logic:** The current game loop handles movement, rendering, and simple state updates, but more advanced logic (such as state switching and conditional behavior) is needed for a full RPG experience.
5. **Quest System:** The QuestLog system is defined but not yet fully integrated. Students should connect NPC dialogue options to quest creation, advancement, and completion.
6. **Leveling System:** A simple progression system already exists in Progression, but it is not fully linked to combat outcomes. Students should ensure that defeating enemies grants experience and triggers level-ups when thresholds are reached.
7. **Advanced AI:** Enemy behavior is currently static and predictable. Students should extend enemy logic to make encounters more engaging and tactical.
8. **Sound and Music:** The engine already initializes Allegro's audio system, but no sounds are played. Students should add audio feedback to enhance immersion

https://doi.org/10.15215/remix/9781998944224.01

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Exercises

1. **RPG System Identification**
   List the major gameplay systems described in this section (movement, dialogue, quests, inventory, combat, progression, and UI). Briefly explain how at least three of these systems interact with one another during normal gameplay.
2. **Tile-Based World Navigation**
   Explain how tile-based maps simplify collision detection and movement logic. Why is it useful to separate visual tile layers from collision or trigger layers?
3. **Dialogue and Choice Flow**
   Examine the dialogue system structure. How do dialogue nodes and choices enable branching conversations, and why is this approach preferable to hard-coded dialogue strings?
4. **Turn-Based Combat Reasoning**
   Describe the basic turn order used in the combat scaffold. How does turn-based combat simplify decision-making and state management compared to real-time combat?

Homework Questions

1. **Balancing Complexity in RPG Design**
   RPGs often include many interconnected systems. What risks arise when too many features are introduced too early in development, and how can a staged or modular approach mitigate these risks?
2. **Quest Design and Player Motivation**
   Discuss how quests help guide player behavior and provide narrative structure. What types of quest objectives are best suited for beginner-level RPG implementations?
3. **Inventory Management Trade-offs**
   Compare simple inventory systems (fixed slot lists) with more complex systems (weight limits, equipment slots). What trade-offs do these designs present for both developers and players?
4. **Progression and Player Engagement**
   Explain how experience points and leveling systems contribute to long-term player engagement. What problems might arise if progression is too fast or too slow?

https://doi.org/10.15215/remix/9781998944224.01

Programming Projects

1. **Combat Mechanics Completion (Core Project)**
   Complete the turn-based combat system by implementing victory and defeat conditions, removing defeated enemies, and returning control to the overworld after combat ends.
2. **NPC Interaction and Dialogue Integration**
   Fully integrate NPC interactions so that the player can initiate dialogue through proximity or facing direction. Connect dialogue outcomes to quest progression, item rewards, or other gameplay effects.
3. **Inventory UI and Item Usage**
   Design and implement a user interface for the inventory system. Allow the player to browse items, select consumables, and apply their effects during gameplay.
4. **Quest System Implementation**
   Extend the quest system so that quests can be started through dialogue, advanced through world triggers or combat events, and completed to grant rewards such as experience, items, or currency.
5. **Leveling and Progression Enhancements**
   Link combat outcomes to the progression system so that defeating enemies grants experience points. Display visual or audio feedback when the player levels up and unlocks stat improvements or abilities.
6. **Enemy AI Improvements (Advanced)**
   Improve enemy behavior by introducing patrol routes, pursuit logic, or simple decision-making (for example, retreating at low health or prioritizing certain player actions).
7. **Audio and Feedback Integration (Advanced)**
   Add sound effects for movement, combat actions, item usage, and level-ups, along with background music for different map areas or combat encounters.

Capstone Project (Optional)

Design and implement a small but complete RPG scenario featuring at least one town, one dungeon, multiple NPCs, and a short quest line with combat and rewards. The game should demonstrate coherent integration of movement, dialogue, quests, combat, inventory, and progression systems. Submit a brief design document explaining how these systems interact and how the player is guided through the experience.

https://doi.org/10.15215/remix/9781998944224.01

8.9. Sports Game

Sports games are a popular genre that simulates the practice of sports. They can range from realistic simulations to more arcade-style games. As with the previous RPG games, other than a simple simulator, sports games have multiple elements that need to be addressed, along with the associated programming tasks that go with those. For example, the simulation of a team spot would require multiple NPCs on both your and the opposing team. The actions of those players need to be considered in terms of responses to gameplay. How does that affect the players' actions, and to what depth and extent might the logic be implemented?

If one considers games like basketball, hockey, or soccer, we have some fairly common elements that can be applied to the game itself: We have a player who has an object—a basketball, a puck, or a soccer ball—and that object moves with the player in an attempt for that player to get that item into the opposing team's goal. Conversely, you have the opposing team players, which are trying to stop your player from getting to that primary objective, so they are going to actively want to head for that player character on the screen, and then they're going to take that object (ball or puck) and try to get it into the first player's goal.

In the simplest implementation, we would make that a one-on-one game. For a game like basketball, we would have one NPC trying to block or take the ball away from the user who's trying to shoot the ball into the hoop. For the game mechanic of shooting the ball into the hoop, we might implement a key press where the amount of time you hold it down is the force with which the ball is thrown toward the hoop. We can set it up control-wise so you don't have to aim at the hoop—that could be automatic—but gameplay-wise, the idea is that as the player, depending on the distance you are from the hoop, you have to determine whether you have enough power to make it to the hoop.

Perhaps we add some other deflection penalty based on the angle to the hoop for increased difficulty. All this time, as well, we have an NPC that's trying to take your ball or block it (if the NPC moves into the path of your ball during flight, a random percentage determines whether they block it). In terms of your throw, are you going to overshoot or undershoot? Then add to that the constraint of gravity.

In a game played on a flat plane, like hockey or soccer, you might use the same approach in regard to using key-press (or mouse-click) duration to affect power. However, in these games, it might be more apt to have the player need to control the angle of the shot to direct his ball or puck toward the goal in question.

https://doi.org/10.15215/remix/9781998944224.01

Recall what we've covered in previous chapters in terms of AI for NPCs. Here, we might do something where the NPC goes toward the player with the ball (seeking) with the goal of trying to take the ball away from the player (where a collision transfers an object). The user player is trying to evade (triggering a flee) function to get away. Here, we need to consider the speed of the player and the NPC in terms of running/skating. If both the user player and the NPC are moving at the same speed, we've got a situation where each will never catch up to the other if they evade. So we might implement a mechanic whereby if you are running in a straight line, your speed gradually increases, but if you change direction, you go back to that base speed. We'd give that same factor to the NPC character as well. Then we'd have a situation where when you're running after an object, you can overtake the object, and a direction change becomes strategic in getting away from a pursuer. As mentioned before, with a collision detection, your player transfers the ball/puck object on contact, so it's a game of figuring out angles and velocities to keep your ball/puck and get into position to make your shot into the goal.

In game scenarios like these, you start to see where one can apply elements from other chapters:

- Animated sprites for characters that change perspective on movement or position
- AI for character movements and actions/reaction
- Background graphics for that nice background screen of a basketball court, soccer field, or an ice rink
- Sound effects when you shoot, when you get goals, or when you miss the shot
- Multithreading for AI

Taking it to the next level, you could have multiple NPCs on-screen: your team and the opponent. This will add multiple layers of logic, where your player NPCs would defend against the opposing team's NPCs and so on. Such is the complexity of elements that you would need to put into even the simplest sports game.

8.9.1. Game Design

Game Concept

Genre: Sports game (basketball shooting).

Objective: The player controls a character that shoots a basketball into a hoop. The goal is to score as many points as possible within a time limit and avoid the opposing team's NPCs.

https://doi.org/10.15215/remix/9781998944224.01

Core Components

Player Character: The athlete controlled by the player
Attributes: Position (x, y), movement speed, sprite
Movement: Controlled via keyboard input
Basketball: The object used to score points
Attributes: Position, velocity, possession state
Physics: Affected by gravity and collisions
Opposing NPC: Defensive character
Attributes: Position, speed, sprite
Behavior: Seeks the player, attempts steals, or blocks shots
Hoop and Scoring Zone: The scoring target
Attributes: Position, collision bounds
Function: Registers a score when the ball passes through correctly
Score and Timer System: Tracks performance
Attributes: Current score, remaining time
Function: Ends the game when time expires

Key Techniques and Features to Have

A sports-style minigame combines responsive controls, lightweight physics, scoring logic, and AI pressure into a compact, fast-paced experience. Although smaller in scope than a full sports title, these systems must integrate cleanly to feel fluid and competitive. The following features define a solid, extensible baseline:

1. **Responsive Player Control and Speed Ramp**
   Player movement should be immediate and low-latency. An optional speed-ramp mechanic can increase maximum speed during sustained straight-line movement and reset on sharp turns, encouraging positioning and intentional movement.
2. **Possession and Ball Carry Logic**
   The ball should exist in clear possession states: player-owned, NPC-owned, or free. When carried, it follows a fixed offset from the owner. Possession transitions occur on shots, steals, blocks, or drops, keeping state management simple and consistent.
3. **Hold-to-Shoot Power (and Optional Angle Assist)**
   Shots should use a press-and-hold mechanic where charge time maps to initial velocity. Optional aim assistance (horizontal alignment or arc adjustment) can be added and scaled to tune difficulty.
4. **Ball Physics and World Bounds**
   Ball movement should use simplified projectile physics with gravity, mild drag, and damped bounces. The ball must remain within court boundaries and come to rest naturally after low-energy impacts.

https://doi.org/10.15215/remix/9781998944224.01

5. **Rim/Hoop Collision and Scoring Window**
   Rim collision and scoring detection should be separated. The rim reflects the ball, while a dedicated scoring window registers successful shots only when the ball passes through in the correct direction.
6. **Defensive NPC AI (Seek/Mark/Intercept)**
   The NPC should follow a priority system: pressure the player when they have the ball, chase a loose ball, or defend the hoop lane. Shot interception attempts should use distance- and timing-based rules with tunable success rates.
7. **Steal and Block Mechanics**
   Steals can occur during close-range ball carrying, transferring possession on success. Blocks or catches may occur while the ball is airborne, governed by probability and short cooldowns to prevent repeated attempts.
8. **HUD: Score, Timer, and Power Bar**
   The HUD should display score, remaining time, and a shot-charge meter. Optional indicators such as streaks or accuracy can support challenge or practice modes while remaining unobtrusive.
9. **Game States and Flow**
   The game should transition cleanly between play, pause, and game-over states. Pausing freezes physics and AI, and game over should offer a quick restart to maintain pacing.
10. **Feedback and Polish**
    Sound effects and subtle visuals should reinforce shots, blocks, and scores. Optional effects include brief screen nudges, particles, or a debug overlay for hitboxes and scoring zones.
11. **Difficulty and Tuning Hooks**
    Difficulty should be adjustable through exposed parameters such as NPC speed, block chance, hoop window size, gravity, bounce damping, and round length, enabling rapid balancing.
12. **Extensibility**
    The design should support easy extension, including power-ups, enhanced shot meters, or multiplayer modes. Modular systems and clean state logic minimize refactoring as features expand.

8.9.2. Implementation

Key Operations in the Game

1. **Input Sampling and Intent Flags**
   - Read Allegro keyboard events each frame; set flags for move left/right/up/down, shoot hold/release, pause, debug.

https://doi.org/10.15215/remix/9781998944224.01

2. **Player Movement Update**
   - Convert intent → desired velocity; apply accel/decel.
   - Apply speed ramp when sustaining direction; reset on sharp turns.
   - Integrate position and clamp to court.
3. **Possession Handling**
   - If ball is carried, pin ball to owner offset.
   - If free on ground, check pickup radius for player or NPC to claim.
   - On shoot release, switch to Free + Airborne.
4. **Shot Charge → Release**
   - While the space bar is held, accumulate charge up to a cap.
   - On release, compute initial (vx, vy) from charge (and optional distance/angle assist), set airborne, play "shot" SFX, reset charge.
5. **Ball Physics Integration**
   - Per tick: gravity → velocity; velocity → position; apply drag.
   - Handle floor/wall collisions with damped reflections; stop when energy is low.
6. **Rim/Hoop Collision and Scoring Check**
   - If ball intersects rim, reflect and play "block/clang" SFX.
   - If ball passes through score window with forward velocity, increment score, resolve postscore state (e.g., hand to NPC or reset).
7. **Steal/Block Attempts**
   - Steal: While player/NPC carries ball, if opponent enters contact radius, roll chance to transfer possession.
   - Block/Catch: While airborne, if NPC near trajectory, roll chance to deflect or catch (ball becomes NPC owned).
8. **NPC AI Update**
   - Determine priority target (player, free ball, hoop lane).
   - Pursue target with accel/turning limits; apply the same speed-ramp rules used for the player.
   - If NPC owns ball and is near its shooting spot, trigger NPC shot release logic.
9. **Timer and Game State**
   - Decrement countdown; when ≤ 0, set gameOver.
   - Handle pause/resume toggles; mute SFX and freeze motion while paused.
10. **HUD and Power Meter Rendering**
    - Draw score and time remaining; render power bar proportional to charge while holding.
    - *Optional:* Accuracy streaks or debug text.
11. **Audio/FX Triggers**

https://doi.org/10.15215/remix/9781998944224.01

- Fire SFX for shot, score, block/clang.
- *Optional:* Small particle/flash at rim contact and score.

12. **Debug Overlay (Optional)**
    - Toggle to draw hitboxes, score window, rim rect, AI target, and current physics values.
13. **Housekeeping and Performance**
    - Avoid per-frame allocations in hot paths; reuse containers.
    - Validate/destroy Allegro resources on shutdown; guard null pointers.

Structure of the Game Program

1. **Initialization**
2. **Resource Loading**
3. **Court/Level and Camera** (bounds, hoop position/zone)
4. **Entities** (Player, NPC, Ball, Hoop; possession)
5. **Input** (movement + hold-to-shoot power)
6. **Physics** (ball gravity/drag, wall/floor bounces)
7. **Collision and Scoring** (ball ↔ rim/hoop, ball ↔ world, player/NPC ↔ ball for steals/blocks)
8. **NPC AI** (seek/mark, intercept ball, speed-ramp mechanic)
9. **Shooting Mechanic** (charge → release → initial velocity + optional angle penalty)
10. **Game Loop / State** (Play, Pause, gameOver)
11. **Score and Timer** (HUD)
12. **Audio and FX** (sfx for shot/score/block; simple particle flash)
13. **Debug Overlay** (toggle: hitboxes, AI targets, power bar)

Coding the Game

```
// ================ 1) Initialization ================
struct App {
    ALLEGRO_DISPLAY* disp=nullptr; ALLEGRO_EVENT_QUEUE*
q=nullptr; ALLEGRO_TIMER* t=nullptr;
    bool init(int W=1280,int H=720,int FPS=60){
        al_init(); al_install_keyboard();
al_install_audio();
        al_init_image_addon(); al_init_font_addon(); al_
init_ttf_addon(); al_init_primitives_addon();
        disp=al_create_display(W,H); t=al_create_
timer(1.0/FPS); q=al_create_event_queue();
        al_register_event_source(q,
al_get_display_event_source(disp));
```

https://doi.org/10.15215/remix/9781998944224.01

```
        al_register_event_source(q,
al_get_timer_event_source(t));
        al_register_event_source(q,
al_get_keyboard_event_source());
        al_start_timer(t); return disp&&q&&t;
    }
    void shutdown(){ if(q)al_destroy_event_queue(q);
if(t)al_destroy_timer(t); if(disp)al_destroy_
display(disp); }
};

// ================ 2) Resource Loading ================
struct Assets {
    std::unordered_map<std::string, ALLEGRO_BITMAP*> bmp;
    std::unordered_map<std::string, ALLEGRO_FONT*> font;
    std::unordered_map<std::string, ALLEGRO_SAMPLE*> sfx;
    bool load(){
        bmp["court"]=al_load_bitmap("assets/court.png");
        bmp["player"]=al_load_bitmap("assets/player.png");
        bmp["npc"]=al_load_bitmap("assets/npc.png");
        bmp["ball"]=al_load_bitmap("assets/ball.png");
        font["ui"]=al_load_ttf_font("assets/ui.ttf", 20,
0);
        sfx["shot"]=al_load_sample("assets/shot.ogg");
        sfx["score"]=al_load_sample("assets/score.ogg");
        sfx["block"]=al_load_sample("assets/block.ogg");
        return true; // check nulls in real code
    }
    ~Assets(){ for(auto&p:bmp) if(p.second) al_destroy_
bitmap(p.second);
               for(auto&p:font) if(p.second) al_destroy_
font(p.second);
               for(auto&p:sfx) if(p.second) al_destroy_
sample(p.second); }
};

// ============== 3) Court/Level & Camera ==============
struct Court {
    int W=1280, H=720; // world size
    // Hoop modeled as a "score window" rectangle (simple):
```

https://doi.org/10.15215/remix/9781998944224.01

```
    ALLEGRO_RECT scoreZone{1180, 220, 1220, 280}; //
right side hoop window
    ALLEGRO_RECT rimRect {1160, 260, 1180, 275};  //
simple rim for collisions
    bool inScoreZone(float x,float y) const {
        return x>=scoreZone.l && x<=scoreZone.r &&
y>=scoreZone.t && y<=scoreZone.b;
    }
    bool inWorld(float x,float y) const { return x>=0 &&
y>=0 && x<=W && y<=H; }
};

// ==================== helpers ====================
struct Vec { float x=0,y=0; };
inline Vec operator+(Vec a,Vec b){return {a.x+b.x,a.
y+b.y};}
inline Vec operator-(Vec a,Vec b){return
{a.x-b.x,a.y-b.y};}
inline Vec operator*(Vec a,float s){return
{a.x*s,a.y*s};}
inline float dot(Vec a,Vec b){return a.x*b.x+a.y*b.y;}
inline float len(Vec a){return std::sqrt(dot(a,a));}
inline Vec norm(Vec a){float L=len(a); return L>0?
a*(1.0f/L) : Vec{0,0};}

// ==================== 4) Entities ================
enum class Possession { Player, NPC, Free };

struct Ball {
    Vec pos{200, 500}, vel{0,0};
    float radius=12.0f; bool airborne=false;
    Possession owner = Possession::Player;
};

struct Actor {
    Vec pos{100,500}, vel{0,0};
    float baseSpeed=200; float burstGain=0; // speed-ramp
mechanic
    bool moving=false;
    ALLEGRO_BITMAP* sprite=nullptr;
};
```

https://doi.org/10.15215/remix/9781998944224.01

```
struct World {
    Court court;
    Actor player, npc;
    Ball ball;
    int score=0;
    float timeLeft=60.0f;
    bool gameOver=false;
};

// ==================== 5) Input ====================
struct Input {
    bool left=false,right=false,up=false,down=false;
    bool shootHold=false, shootJustReleased=false,
debug=false, quit=false;
    void onEvent(const ALLEGRO_EVENT& ev){
        if(ev.type==ALLEGRO_EVENT_KEY_DOWN ||
ev.type==ALLEGRO_EVENT_KEY_UP){
            bool d = ev.type==ALLEGRO_EVENT_KEY_DOWN;
            switch(ev.keyboard.keycode){
                case ALLEGRO_KEY_A: case ALLEGRO_KEY_
LEFT:  left=d; break;
                case ALLEGRO_KEY_D: case ALLEGRO_KEY_
RIGHT: right=d; break;
                case ALLEGRO_KEY_W: case ALLEGRO_KEY_
UP:    up=d; break;
                case ALLEGRO_KEY_S: case ALLEGRO_KEY_
DOWN:  down=d; break;
                case ALLEGRO_KEY_SPACE: shootHold=d;
shootJustReleased=!d; break;
                case ALLEGRO_KEY_F1: if(d) debug=!debug;
break;
                case ALLEGRO_KEY_ESCAPE: if(d) quit=true;
break;
            }
        }
    }
};

// ================ 6) Physics (ball) ==================
struct Physics {
```

https://doi.org/10.15215/remix/9781998944224.01

```
    float gravity=900.0f; float drag=0.995f;
    void stepBall(Ball& b, Court& c, float dt){
        if(!b.airborne) return;
        b.vel.y += gravity*dt;
        b.pos = b.pos + b.vel*dt;
        b.vel = b.vel*drag;

        // floor bounce
        if (b.pos.y + b.radius > c.H) {
            b.pos.y = c.H-b.radius;
            b.vel.y *=-0.55f; // damped bounce
            if (std::fabs(b.vel.y) < 60) {
b.airborne=false; b.vel={0,0}; }
        }
        // walls
        if (b.pos.x-b.radius < 0)  { b.pos.x = b.radius;
b.vel.x *=-0.6f; }
        if (b.pos.x + b.radius > c.W){ b.pos.x = c.W-b.
radius; b.vel.x *=-0.6f; }
    }
};

// ================ 7) Collision & Scoring ==============
bool intersectsCircleRect(Vec p,float r, ALLEGRO_RECT R){
    float cx = std::clamp(p.x, R.l, R.r);
    float cy = std::clamp(p.y, R.t, R.b);
    float dx = p.x-cx, dy=p.y-cy;
    return dx*dx + dy*dy <= r*r;
}

struct Collisions {
    // Rim knockback and score detection
    void handle(World& w, Assets& A){
        // score: ball passes through scoreZone while
moving left->right
        if (w.ball.airborne && w.court.inScoreZone(w.
ball.pos.x, w.ball.pos.y) && w.ball.vel.x>0){
            ++w.score; w.ball.airborne=false; w.ball.
owner=Possession::NPC; // after score, hand to NPC?
```

https://doi.org/10.15215/remix/9781998944224.01

```
            al_play_sample(A.sfx["score"], 1,0,1,
ALLEGRO_PLAYMODE_ONCE, nullptr);
        }
        // rim collision → reflect
        if (intersectsCircleRect(w.ball.pos, w.ball.
radius, w.court.rimRect)){
            w.ball.vel.x *=-0.6f; w.ball.vel.y *=-0.6f;
            al_play_sample(A.sfx["block"], 0.6,0,1,
ALLEGRO_PLAYMODE_ONCE, nullptr);
        }
        // steal/block: if NPC intersects ball in flight
→ chance to catch
        float dist = len(w.npc.pos-w.ball.pos);
        if (w.ball.airborne && dist < 32.0f) {
            if ((rand()%100)<30) { // 30% block/catch
                w.ball.airborne=false; w.ball.
owner=Possession::NPC; w.ball.vel={0,0};
                al_play_sample(A.sfx["block"], 1,0,1,
ALLEGRO_PLAYMODE_ONCE, nullptr);
            }
        }
        // pickup when free on ground
        if (!w.ball.airborne && w.ball.
owner==Possession::Free){
            if (len(w.player.pos-w.ball.pos)<28) w.ball.
owner=Possession::Player;
            if (len(w.npc.pos  -w.ball.pos)<28) w.ball.
owner=Possession::NPC;
        }
        // carry with owner
        if (!w.ball.airborne){
            if (w.ball.owner==Possession::Player) w.ball.
pos = w.player.pos + Vec{10,-20};
            if (w.ball.owner==Possession::NPC)    w.ball.
pos = w.npc.pos    + Vec{-10,-20};
        }
    }
};

// ========= 8) NPC AI (seek/mark + speed ramp) =========
```

https://doi.org/10.15215/remix/9781998944224.01

```
struct AI {
    float accel=800.0f, decel=1200.0f, burstMax=180.0f,
turnPenalty=160.0f;

    void update(World& w, float dt){
        // target = player if player has ball, else loose
ball, else hoop area to defend
        Vec target = (w.ball.owner==Possession::Player) ?
w.player.pos
                   : (w.ball.owner==Possession::Free)   ?
w.ball.pos
                   : Vec{w.court.scoreZone.l, (w.court.
scoreZone.t+w.court.scoreZone.b)/2};

        // pursue
        Vec dir = norm(target-w.npc.pos);
        Vec desired = dir * (w.npc.baseSpeed + w.npc.
burstGain);
        // turn penalty: reduce burst if direction
changes sharply
        if (dot(norm(w.npc.vel), dir) < 0.6f) w.npc.
burstGain = 0; // sharp turn resets run-up

        // accelerate towards desired
        Vec dv = desired-w.npc.vel;
        Vec step = norm(dv) * std::min(len(dv), accel*dt);
        w.npc.vel = w.npc.vel + step;

        // straight-line ramp up
        if (len(step)>0.0f && dot(norm(w.npc.vel), dir) >
0.95f)
            w.npc.burstGain = std::min(burstMax, w.npc.
burstGain + 60.0f*dt);

        // integrate & clamp to court
        w.npc.pos = w.npc.pos + w.npc.vel*dt;
        w.npc.pos.x = std::clamp(w.npc.pos.x, 20.0f,
(float)w.court.W-20.0f);
        w.npc.pos.y = std::clamp(w.npc.pos.y, 20.0f,
(float)w.court.H-20.0f);
```

https://doi.org/10.15215/remix/9781998944224.01

```
        // NPC simple shot if it owns ball and near left
hoop (mirror if needed)
        if (w.ball.owner==Possession::NPC && w.npc.pos.x
> w.court.W*0.7f) {
            // quick lob toward hoop
            w.ball.owner=Possession::Free; w.ball.
airborne=true;
            w.ball.vel = { 320.0f,-520.0f }; // tuned by
trial
        }
    }
};

// ========== 9) Shooting Mechanic (player) ==========
struct Shooter {
    float charge=0, chargeMax=1.5f; // seconds held
    void tick(Input& in, float dt){ if(in.shootHold)
charge = std::min(charge+dt, chargeMax); }
    void releaseIfAny(Input& in, World& w, Assets& A){
        if (!in.shootJustReleased) return;
        if (w.ball.owner!=Possession::Player) {
in.shootJustReleased=false; charge=0; return; }
        float p = charge / chargeMax; // 0..1
        // Auto-aim assist on X, gravity handled by
physics; add small angle penalty with distance
        float baseVX = 500.0f * (0.6f + 0.8f*p);
        float baseVY =-700.0f * (0.5f + 0.7f*p);
        // Optional difficulty: off-axis penalty
increases vertical error with distance:
        float dx = (w.court.scoreZone.l-w.player.pos.x);
        float distFactor = std::clamp(dx / 900.0f, 0.f,
1.f);
        baseVY-= 120.0f * distFactor; // need more arc
from far

        w.ball.owner=Possession::Free;
        w.ball.airborne=true;
        w.ball.vel = { baseVX, baseVY };
        al_play_sample(A.sfx["shot"], 0.9,0,1, ALLEGRO_
PLAYMODE_ONCE, nullptr);
```

https://doi.org/10.15215/remix/9781998944224.01

```
        charge=0; in.shootJustReleased=false;
    }
};

// =============== 10) Game Loop / State ===============
// ================= 11) Score & Timer =================
// ================== 12) Audio & FX ===================
// ===================== 13) Debug =====================
struct Game {
    App app; Assets assets; World world; Input input;
    Physics physics; Collisions coll; AI ai; Shooter
shooter;
    bool debug=false;

    bool boot(){
        if(!app.init()) return false;
        assets.load();
        world.player.sprite = assets.bmp["player"];
        world.npc.sprite    = assets.bmp["npc"];
        return true;
    }

    void handleEvent(const ALLEGRO_EVENT& ev){
        if(ev.type==ALLEGRO_EVENT_DISPLAY_CLOSE) input.
quit=true;
        if(ev.type==ALLEGRO_EVENT_KEY_DOWN ||
ev.type==ALLEGRO_EVENT_KEY_UP) input.onEvent(ev);
    }

    void update(float dt){
        if (input.quit) { world.gameOver=true; return; }
        if (world.gameOver) return;

        // Player move + speed-ramp like NPC
        Vec dir{ (float)input.right-(float)input.left,
(float)input.down-(float)input.up };
        if (len(dir)>0){ dir=norm(dir); world.player.
moving=true; }
        else { world.player.moving=false; }
```

https://doi.org/10.15215/remix/9781998944224.01

```
        // ramp
        if (world.player.moving && dot(norm(world.player.
vel), dir) > 0.95f)
            world.player.burstGain = std::min(180.0f,
world.player.burstGain + 80.0f*dt);
        else if (!world.player.moving)
world.player.burstGain = std::max(0.0f,
world.player.burstGain-200.0f*dt);

        Vec desired = dir * (world.player.baseSpeed +
world.player.burstGain);
        Vec dv = desired-world.player.vel;
        world.player.vel = world.player.vel + norm(dv) *
std::min(len(dv), 1200.0f*dt);
        world.player.pos = world.player.pos + world.
player.vel*dt;
        world.player.pos.x = std::clamp(world.player.
pos.x, 20.0f, (float)world.court.W-20.0f);
        world.player.pos.y = std::clamp(world.player.
pos.y, 20.0f, (float)world.court.H-20.0f);

        // AI + ball physics + collisions
        ai.update(world, dt);
        shooter.tick(input, dt);
        physics.stepBall(world.ball, world.court, dt);
        coll.handle(world, assets);
        shooter.releaseIfAny(input, world, assets);

        // Timer
        world.timeLeft-= dt;
        if (world.timeLeft <= 0) world.gameOver = true;
    }

    void draw(){
        al_clear_to_color(al_map_rgb(20,20,30));
        if (assets.bmp["court"])
            al_draw_bitmap(assets.bmp["court"], 0, 0, 0);

        // Draw actors
```

https://doi.org/10.15215/remix/9781998944224.01

```
        if (world.player.sprite) al_draw_
bitmap(world.player.sprite, world.player.pos.x-16,
world.player.pos.y-32, 0);
        if (world.npc .sprite)  al_draw_bitmap(world.npc
.sprite, world.npc .pos.x-16,  world.npc .pos.y-32,  0);
        if (assets.bmp["ball"]) al_draw_bitmap(assets.
bmp["ball"], world.ball.pos.x-12, world.ball.pos.y-12, 0);

        // HUD
        std::string hud = "Score: " + std::to_
string(world.score) + "   Time: " + std::to_string((int)
std::ceil(world.timeLeft));
        al_draw_text(assets.font["ui"], al_map_
rgb(255,255,255), 16, 16, 0, hud.c_str());

        // Power bar (shoot charge)
        float pw = 200.0f * (std::min(shooter.charge,
shooter.chargeMax)/shooter.chargeMax);
        al_draw_filled_rectangle(16, 48, 16+pw, 64,
al_map_rgb(80,220,120));

        // Debug
        if (input.debug){
            // score window & rim
            al_draw_rectangle(world.court.scoreZone.l,
world.court.scoreZone.t,
                              world.court.scoreZone.r,
world.court.scoreZone.b, al_map_rgb(255,255,0), 2);
            al_draw_rectangle(world.court.rimRect.l,
world.court.rimRect.t,
                              world.court.rimRect.r,
world.court.rimRect.b, al_map_rgb(255,0,0), 2);
        }

        if (world.gameOver){
            al_draw_text(assets.font["ui"], al_map_
rgb(255,200,0), 640, 320, ALLEGRO_ALIGN_CENTER, "GAME
OVER");
        }
```

https://doi.org/10.15215/remix/9781998944224.01

```
        al_flip_display();
    }

    void run(){
        while(true){
            ALLEGRO_EVENT ev; al_wait_for_event(app.q,
&ev);
            if (ev.type==ALLEGRO_EVENT_TIMER){
update(1.0f/60.0f); draw(); }
            else handleEvent(ev);
            if (input.quit) break;
        }
    }
};

int main(){
    Game g; if(!g.boot()) return 1;
    g.run(); g.app.shutdown(); return 0;
}
```

TASKS FOR COMPLETION AND IMPROVEMENT

1. **Shooting Mechanics:** Implement the logic for shooting the basketball and checking if it goes into the hoop.
2. **Scoring System:** Add a scoring system to keep track of successful shots.
3. **Animations:** Add animations for shooting and scoring.
4. **Sound Effects:** Add sound effects for shooting and scoring.

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Exercises

1. **Possession State Analysis**
   Identify the different possession states of the ball (player-owned, NPC-owned, free). Explain how transitions between these states occur during gameplay and why it is important to explicitly manage possession.

https://doi.org/10.15215/remix/9781998944224.01

2. **Hold-to-Shoot Mechanics**
   Explain how charging shot power using key-press duration affects gameplay difficulty. What advantages does this mechanic have over a single-press shooting model?
3. **Ball Physics Understanding**
   Describe the role of gravity, velocity, and damping in the basketball's movement. How do these factors influence whether a shot overshoots, undershoots, or scores?
4. **NPC Blocking Logic**
   Analyze the NPC's blocking and interception behavior. Why is it beneficial to use probability and cooldowns rather than deterministic blocking?

Homework Questions

1. **Sports Game Design Trade-Offs**
   Compare realism and fun in sports games. Why might an arcade-style basketball game intentionally simplify physics or automate aiming?
2. **Speed-Ramp Mechanics**
   Discuss how gradually increasing speed during sustained movement changes player strategy. How does this mechanic affect evasion, pursuit, and positioning?
3. **Scoring Reliability**
   Explain why separating rim collision detection from scoring window detection produces more reliable scoring behavior than treating the hoop as a single collision object.
4. **AI Pressure and Player Experience**
   How does continuous NPC pressure (stealing, blocking, intercepting) influence the pace and challenge of the game? What risks exist if NPC behavior is too aggressive or too passive?

Programming Projects

1. **Shooting Mechanics Completion (Core Project)**
   Fully implement the shooting logic, including charging, release, trajectory calculation, and successful scoring detection when the ball passes through the hoop's scoring window.
2. **Scoring System Enhancement**
   Extend the scoring system to track points per shot, streak bonuses, or accuracy percentages. Display this information in the HUD and update it dynamically.

https://doi.org/10.15215/remix/9781998944224.01

3. **Animations and Visual Feedback**
   Add animations for player movement, shooting, and successful baskets. Include simple visual effects such as flashes, particles, or camera nudges to emphasize key events.
4. **Sound Effects Integration**
   Incorporate sound effects for shooting, rim impacts, blocked shots, and scoring. Balance audio feedback to enhance immersion without overwhelming the player.
5. **Improved NPC Behavior (Advanced)**
   Enhance NPC AI by introducing varied defensive tactics, such as anticipating shots based on player distance or prioritizing blocking over stealing in certain situations.
6. **Difficulty Modes (Advanced)**
   Implement multiple difficulty settings by adjusting NPC speed, block probability, shot accuracy penalties, or game timer length.

Capstone Project

Design and implement a complete one-on-one basketball challenge with polished presentation. The game should include responsive controls, believable physics, competitive NPC behavior, clear scoring and timing rules, and audio-visual feedback. Submit a short design report explaining your choices for difficulty balancing, AI logic, and user interface clarity.

8.10. Strategy Game

Strategy games are a genre that emphasizes skillful thinking and planning to achieve victory. They come in various forms, each offering unique gameplay experiences, from terraforming a strange new world, to running a farm or factory, to creating armies to do battle. There is something for everyone in this genre.

Like the RPG game, there are multiple design elements that make the game complex. The player has multiple units and assets to control and manage; likewise for the NPC. The information needs to be presented to the player effectively, and the interface for the game needs to be intuitive (mostly) and smooth. Usually, this would be mouse driven, with menus and clickable icons or text elements. AI plays a role in not only the NPC playing against the human player but also the player's units following commands or taking actions unattended.

Let's consider a basic real-time strategy (RTS) combat game, where the player can control units to gather resources and build structures. In the simplest form, we will have a map with resources scattered around; these are required to build our combat units. The players each start off at random locations, and the

https://doi.org/10.15215/remix/9781998944224.01

game's goal is to gather resources, build your combat units, and use those to take control of the whole map.

8.10.1. Game Design

Game Concept

Genre: Real-time strategy (RTS).

Objective: The player manages units and resources, constructs structures, and defeats an opposing force through strategic planning and real-time decision-making.

Core Components

Units: Controllable characters

Attributes: Position, health, attack power, state

Actions: Move, gather resources, attack, build

Resources: Materials used for production

Attributes: Type (wood, stone), amount, location

Function: Enable unit and structure creation

Structures: Buildings placed on the map

Attributes: Position, health, type

Function: Produce units or support the economy

Economy System: Manages resource flow

Function: Handles gathering, carrying, and depositing resources

Enemy AI: Controls opposing forces

Behavior: Gathers resources, builds units, attacks player assets

UI and Command System: Player interaction layer

Function: Selection, issuing commands, displaying information

Key Techniques and Features to Have

A real-time strategy (RTS) game integrates unit control, economic simulation, pathfinding, artificial intelligence, and a readable user interface into a continuous gameplay loop. Even in simplified form, these systems must work together smoothly to remain responsive, understandable, and extensible. The following features define a robust baseline RTS implementation using Allegro:

1. **Mouse-Driven Selection and Commands**
   Players should select units with single clicks or box selection, using modifier keys (e.g., Shift) to add or remove units. Right-click commands should be context-sensitive, automatically resolving to move, gather, attack, or build actions. Clear visual feedback (selection rings or marquees) is essential.

https://doi.org/10.15215/remix/9781998944224.01

2. **World–Screen Coordinate Mapping**
   All interactions should rely on consistent conversions between screen space, camera space, and world or grid space. Coordinate mapping should be centralized and documented, especially when using a tile grid, to support selection, placement, and pathfinding reliably.
3. **Pathfinding and Movement**
   Unit movement can begin with simple steering and obstacle avoidance and later evolve into grid-based A* pathfinding with passability masks. Basic separation or formation logic should reduce unit clumping and improve group movement behavior.
4. **Finite-State Units**
   Units should use finite state machines to govern behavior such as idle, moving, gathering, attacking, building, or retreating. State transitions should depend on ranges, cooldowns, and pursuit limits to keep behavior predictable and controllable.
5. **Core Economy Loop**
   The economy should follow a clear gather-carry-deposit pipeline. Resource nodes should deplete over time, encouraging expansion. Multiple resource types and tunable costs enable balanced unit production, structure building, and upgrades.
6. **Building and Placement System**
   Structures should be placed using a ghost preview that indicates valid or invalid locations. Placement checks include collision and terrain passability. Structures should progress through construction stages, and production buildings should support rally points.
7. **Combat Model**
   Combat should follow a consistent damage model incorporating range, armor or defense, and attack type (projectile or hitscan). Units should prioritize targets logically and handle death through cleanup, optional effects, or refunds.
8. **Opponent AI**
   Enemy AI should follow a simple strategic loop: gathering resources, expanding infrastructure, training units, and attacking the player. AI difficulty can scale through resource rates, reaction timing, and unit production speed.
9. **Fog of War (Recommended)**
   Fog of war should track visibility per tile, distinguishing explored areas from currently visible ones. Unseen areas should be hidden or dimmed, with changes reflected on the minimap to enhance strategic planning.

https://doi.org/10.15215/remix/9781998944224.01

10. **UI/HUD**
    The interface should clearly display resources, population limits, selected unit information, and build options. Tool tips should explain costs and stats. An optional minimap can support navigation, alerts, and situational awareness.
11. **Game States and Save/Load**
    The game should transition cleanly between play, pause, victory, and defeat states. Pausing must freeze simulation without losing state. Optional save/load functionality may snapshot world state, resources, and production queues.
12. **Performance and Data-Driven Design**
    To maintain performance, avoid per-frame allocations and reuse frequently created objects. Units, structures, and costs should be defined in external data files (e.g., JSON or CSV) to allow tuning and extension without recompilation.

8.10.2. Implementation

Key Operations in the Game

1. **Event Intake and Intent Mapping**
   - Poll Allegro events (mouse, keyboard).
   - Update intent flags (selecting, dragging, issuing right-click orders, build hotkeys, camera pan/zoom, pause).
2. **Screen ↔ World Transform**
   - Convert cursor from screen → world using camera offset and zoom.
   - Use world coords for hit-tests (units, resources, structures) and placement ghosts.
3. **Selection Lifecycle**
   - On LMB down: Start marquee.
   - On LMB up: Compute marquee AABB in world space; mark units inside as selected (respect team).
   - Shift-click to add/remove single units.
4. **Command Resolution (RMB)**
   - Determine context at clicked world point (terrain, resource node, enemy, friendly building).
   - Issue corresponding order to all selected units: Move, Gather, Attack, Build/Repair, or Return.
   - Stagger goals slightly (formation spread) to reduce overlap.
5. **Per-Tick Unit FSM Update**
   - For each unit, process state (Idle/Move/Gather/Return/Attack/Build).
   - Update cooldowns, acquisition, and retreat checks.

https://doi.org/10.15215/remix/9781998944224.01

- Transition when goals are reached or conditions change (node empty, target lost).

6. **Path/Movement Step**
   - If path exists, advance along waypoints; else request path if stuck or goal changed.
   - Collision/passability checks against map mask; nudge or repath on obstruction.
7. **Economy Tick**
   - Gather when in range of node (decrease node amount, increase carried).
   - Return to depot; on arrival, deposit to player stockpile; resume gathering if node remains.
8. **Build System Tick**
   - If placing: update ghost position and validity; on confirm, spawn construction entity with progress.
   - On completion, transform to finished structure, play SFX, set rally point (if applicable).
9. **Combat Tick**
   - Target acquisition (in range and visible).
   - Attack when off cooldown (apply damage or spawn projectile).
   - Handle death: Remove entity, award bounty/XP (if used), trigger alerts.
10. **Opponent AI Step**
    - Simple planner: if resources ≥ threshold → build; if army size ≥ threshold → harass nearest player asset.
    - Assign worker routines and defense reactions.
11. **Camera and Minimap**
    - Update camera (edge-pan, WASD pan, mouse-wheel zoom).
    - Draw or update minimap buffer; map clicks to camera moves.
12. **Fog of War (If Enabled)**
    - Recompute visible tiles from unit vision each tick or every N frames.
    - Mask terrain/units in render pass; update minimap overlay.
13. **Render Pipeline (per Frame)**
    - Clear; draw terrain; draw resources/structures/units (with selection rings and team colours); draw projectiles/effects; then UI/HUD/minimap; flip.
    - Draw placement ghost and build progress bars.
14. **Audio and Alerts**
    - Trigger SFX on gather tick, build complete, attack, unit trained, under attack.
    - *Optional:* On-screen pings and minimap flashes.

https://doi.org/10.15215/remix/9781998944224.01

15. **Win/Lose and Persistence**
    - Check victory conditions (e.g., enemy HQ destroyed) or defeat (no HQ + no builders).
    - Save/load hooks for snapshotting world state.
16. **Housekeeping and Performance**
    - Cull off-screen effects; reuse vectors/buffers; guard all Allegro resources.
    - Frame budget awareness: Consider updating costly systems (pathfinding/FoW) on intervals.

Structure of the Game Program

1. **Initialization**
2. **Resource Loading** (sprites, fonts, SFX)
3. **Map/Terrain and Camera** (grid, passability, spawn points)
4. **Entities** (Units, Resources, Structures; teams/factions)
5. **Selection and Commands** (mouse box-select, right-click orders)
6. **Pathfinding and Movement** (grid A* later; stub steering now)
7. **Economy** (gather, carry, deposit, resource counts)
8. **Build System** (place ghost, validate, spawn structure)
9. **Combat** (targeting, attack cooldown, damage/death)
10. **Enemy AI** (simple "expand + harass")
11. **Game Loop / States** (Play, Pause, gameOver)
12. **UI / HUD** (resources, selected panel; optional minimap)
13. **Save/Load** (*Optional:* stubbed)

Coding the Game

```
// ================= 1) Initialization =================
struct App {
    ALLEGRO_DISPLAY* disp=nullptr; ALLEGRO_EVENT_QUEUE*
q=nullptr; ALLEGRO_TIMER* timer=nullptr;
    bool init(int W=1280,int H=720,int FPS=60){
        al_init(); al_install_keyboard(); al_install_
mouse(); al_install_audio();
        al_init_image_addon(); al_init_font_addon(); al_
init_ttf_addon(); al_init_primitives_addon();
        disp=al_create_display(W,H); timer=al_create_
timer(1.0/FPS); q=al_create_event_queue();
        al_register_event_source(q,
al_get_display_event_source(disp));
        al_register_event_source(q,
al_get_timer_event_source(timer));
```

https://doi.org/10.15215/remix/9781998944224.01

```
        al_register_event_source(q,
al_get_keyboard_event_source());
        al_register_event_source(q,
al_get_mouse_event_source());
        al_start_timer(timer); return disp&&q&&timer;
    }
    void shutdown(){ if(q)al_destroy_event_queue(q);
if(timer)al_destroy_timer(timer); if(disp)al_destroy_
display(disp); }
};

// ================ Common helpers =====================
struct Vec { float x=0,y=0; };
inline Vec operator+(Vec a,Vec b){return {a.x+b.x,a.
y+b.y};}
inline Vec operator-(Vec a,Vec b){return
{a.x-b.x,a.y-b.y};}
inline Vec operator*(Vec a,float s){return
{a.x*s,a.y*s};}
inline float dot(Vec a,Vec b){return a.x*b.x+a.y*b.y;}
inline float len(Vec a){return std::sqrt(dot(a,a));}
inline Vec norm(Vec a){float L=len(a); return L>0?
a*(1.0f/L):Vec{0,0};}

// ================ 2) Resource Loading ================
struct Assets {
    std::unordered_map<std::string, ALLEGRO_BITMAP*> bmp;
    std::unordered_map<std::string, ALLEGRO_FONT*>   font;
    std::unordered_map<std::string, ALLEGRO_SAMPLE*> sfx;
    bool load(){
        bmp["tiles"]=al_load_bitmap("assets/tiles.png");
        bmp["unit"]=al_load_bitmap("assets/unit.png");
        bmp["enemy"]=al_load_bitmap("assets/enemy.png");
        bmp["tree"]=al_load_bitmap("assets/tree.png");
        bmp["mine"]=al_load_bitmap("assets/mine.png");
        bmp["hq"]  =al_load_bitmap("assets/hq.png");
        bmp["barracks"]=al_load_bitmap("assets/barracks.
png");
        font["ui"]=al_load_ttf_font("assets/ui.ttf", 18,
0);
```

https://doi.org/10.15215/remix/9781998944224.01

```
        sfx["gather"]=al_load_sample("assets/gather.ogg");
        sfx["build"]=al_load_sample("assets/build.ogg");
        sfx["attack"]=al_load_sample("assets/attack.ogg");
        return true; // check nulls in real code
    }
    ~Assets(){ for(auto&p:bmp) if(p.second) al_destroy_
bitmap(p.second);
               for(auto&p:font) if(p.second) al_destroy_
font(p.second);
               for(auto&p:sfx)  if(p.second) al_destroy_
sample(p.second); }
};

// ================= 3) Map/Terrain & Camera ===========
struct Map {
    int W=128, H=72, tile=10;              // world grid
size; tile px size (camera scales to display)
    std::vector<uint8_t> pass;             // 0 free, 1
blocked
    bool in(int gx,int gy)const{ return
gx>=0&&gy>=0&&gx<W&&gy<H; }
    bool blockedPx(int px,int py) const {
        int gx=px/tile, gy=py/tile; if(!in(gx,gy)) return
true; return pass[gy*W+gx]!=0;
    }
    void generate(){
        pass.assign(W*H,0);
        // scatter some blocked tiles to simulate rocks/
trees (very minimal)
        for(int i=0;i<200;i++){ int gx=rand()%W,
gy=rand()%H; pass[gy*W+gx]=1; }
    }
    void draw(const Assets& A, float camx, float camy){
        // Simple flat color grid; replace with tileset
draw if desired
        for(int y=0;y<H;y++) for(int x=0;x<W;x++){
            ALLEGRO_COLOR c = pass[y*W+x]?
al_map_rgb(60,85,60):al_map_rgb(35,120,35);
            al_draw_filled_rectangle(x*tile-camx,y*tile-
camy,(x+1)*tile-camx,(y+1)*tile-camy,c);
```

https://doi.org/10.15215/remix/9781998944224.01

```
        }
    }
};
struct Camera { float x=0,y=0; void follow(Vec p,int
sw,int sh,int tile){ x=p.x-sw/2; y=p.y-sh/2; if(x<0)
x=0;if(y<0)y=0; }

// ================= 4) Entities ========================
enum class Team { Player, Enemy, Neutral };
enum class UnitState { Idle, Move, Gather, Return,
Attack, Build };
enum class ResourceType { Wood, Stone };

struct ResourceNode {
    Vec pos; int amount=300; ResourceType
type=ResourceType::Wood;
    ALLEGRO_BITMAP* sprite=nullptr;
};

struct Structure {
    Vec pos; Team team; int hp=500; std::string kind; //
"HQ","Barracks","Depot"
    ALLEGRO_BITMAP* sprite=nullptr;
};

struct Unit {
    Vec pos, vel; Team team=Team::Player; int hp=100,
atk=8; float range=22;
    UnitState st=UnitState::Idle; Vec goal;
int carry=0; int carryMax=50; ResourceType
carryType=ResourceType::Wood;
    Structure* homeDepot=nullptr; ResourceNode*
targetNode=nullptr; Unit* targetEnemy=nullptr;
    ALLEGRO_BITMAP* sprite=nullptr; bool selected=false;
};

struct World {
    Map map; Camera cam;
    std::vector<Unit> units;
    std::vector<ResourceNode> nodes;
```

https://doi.org/10.15215/remix/9781998944224.01

```
    std::vector<Structure> structures;
    int wood=100, stone=0;      // player resources
    int e_wood=100;             // enemy resources (toy)
    bool gameOver=false;
};

// ============== 5) Selection & Commands ===============
struct Input {
    bool quit=false; bool lmb=false, rmb=false; bool
dragging=false;
    Vec mouse{0,0}, dragStart{0,0};
    void handle(const ALLEGRO_EVENT& ev){
        if(ev.type==ALLEGRO_EVENT_DISPLAY_CLOSE)
quit=true;
        if(ev.type==ALLEGRO_EVENT_MOUSE_AXES){ mouse={
(float)ev.mouse.x,(float)ev.mouse.y }; }
        if(ev.type==ALLEGRO_EVENT_MOUSE_BUTTON_DOWN){
            if(ev.mouse.button==1){ lmb=true;
dragging=true; dragStart=mouse; }
            if(ev.mouse.button==2){ rmb=true; }
        }
        if(ev.type==ALLEGRO_EVENT_MOUSE_BUTTON_UP){
            if(ev.mouse.button==1){ lmb=false;
dragging=false; }
            if(ev.mouse.button==2){ rmb=false; }
        }
    }
};

struct Selection {
    void boxSelect(World& w, Vec a, Vec b, float
camx,float camy){
        float l=std::min(a.x,b.x)+camx,
r=std::max(a.x,b.x)+camx;
        float t=std::min(a.y,b.y)+camy,
d=std::max(a.y,b.y)+camy;
        for(auto& u:w.units) if(u.team==Team::Player){
            float ux=u.pos.x, uy=u.pos.y;
            u.selected = (ux>=l && ux<=r && uy>=t &&
uy<=d);
```

https://doi.org/10.15215/remix/9781998944224.01

```
        }
    }
    void clear(World& w){ for(auto& u:w.units)
u.selected=false; }
};

// ============== 6) Pathfinding & Movement ============
// Minimal "greedy steering" towards goal with obstacle
nudge.
// Swap with A* grid later.
struct Mover {
    float speed=90.0f;
    void step(Unit& u, const Map& m, float dt){
        Vec d = u.goal-u.pos; if(len(d)<4){ u.vel={0,0};
return; }
        Vec dir = norm(d);
        // Simple obstacle nudge
        Vec next = u.pos + dir*speed*dt;
        if (m.blockedPx((int)next.x,(int)u.pos.y)) next.
x=u.pos.x;
        if (m.blockedPx((int)u.pos.x,(int)next.y)) next.
y=u.pos.y;
        u.pos = next;
    }
};

// ============== 7) Economy (gather/deposit) ==========
struct Economy {
    void update(Unit& u, World& w, float dt){
        switch(u.st){
        case UnitState::Gather:
            if (!u.targetNode || u.targetNode->amount<=0)
{ u.st=UnitState::Idle; break; }
            if (len(u.pos-u.targetNode->pos) > 18) {
u.goal = u.targetNode->pos; } // move closer
            else {
                // gather tick
                int take = std::min(5,
u.targetNode->amount);
```

https://doi.org/10.15215/remix/9781998944224.01

```
                u.targetNode->amount-= take; u.carry +=
take; u.carryType = u.targetNode->type;
                if (u.carry >= u.carryMax)
{ u.st=UnitState::Return; u.goal = u.homeDepot?
u.homeDepot->pos : u.pos; }
            }
            break;
        case UnitState::Return:
            if (len(u.pos-u.goal) > 18) break;
            // deposit
            if (u.team==Team::Player){
                if (u.carryType==ResourceType::Wood)
w.wood += u.carry; else w.stone += u.carry;
            }
            u.carry=0; u.st=UnitState::Gather; // resume
node
            break;
        default: break;
        }
    }
};

// ============== 8) Build System ======================
struct Build {
    bool placing=false; std::string kind="Barracks"; Vec
ghostPx;
    int costWood=150;
    bool canPlace(const World& w, Vec px){
        // crude footprint check (block tiles?)
        return !w.map.blockedPx((int)px.x, (int)px.y);
    }
    void confirm(World& w, const Assets& A){
        if(!placing) return;
        if (kind=="Barracks" && w.wood>=costWood &&
canPlace(w, ghostPx)){
            w.wood-= costWood;
            Structure s; s.kind="Barracks";
s.pos=ghostPx; s.team=Team::Player; s.sprite=A.bmp.
at("barracks");
```

https://doi.org/10.15215/remix/9781998944224.01

```
            w.structures.push_back(s);
        }
        placing=false;
    }
};

// ============== 9) Combat ============================
struct Combat {
    float attackCD=0.8f;
    void tick(Unit& u, World& w, float dt){
        static std::unordered_map<Unit*, float> cd;
        cd[&u] = std::max(0.0f, cd[&u]-dt);
        if (u.st==UnitState::Attack && u.targetEnemy){
            if (len(u.pos—u.targetEnemy->pos) > u.range)
{ u.goal = u.targetEnemy->pos; return; }
            if (cd[&u]<=0.0f){
                u.targetEnemy->hp—= u.atk;
cd[&u]=attackCD;
                if (u.targetEnemy->hp<=0) u.targetEnemy-
>hp=0; // (remove later)
            }
        }
    }
};

// ============== 10) Enemy AI =========================
struct EnemyAI {
    void update(World& w, float dt){
        // Very minimal: enemy gathers from nearest node
until enough wood, then spawns a unit at enemy HQ to
harass.
        // Find enemy HQ
        Structure* ehq=nullptr;
        for(auto& s:w.structures) if(s.team==Team::Enemy
&& s.kind=="HQ"){ ehq=&s; break; }
        if(!ehq) return;

        // Periodically spawn a small raider if resources
allow
        static float acc=0; acc+=dt;
```

https://doi.org/10.15215/remix/9781998944224.01

```
        if (acc>5.0f && w.e_wood>=50){ acc=0; w.e_wood-=50;
            Unit e; e.team=Team::Enemy; e.pos=ehq->pos;
e.sprite=nullptr; e.hp=80; e.atk=6;
            // set it to attack nearest player structure
            Structure* tgt=nullptr; float best=1e9;
            for(auto& s:w.structures) if(s.
team==Team::Player){
                float d = len(s.pos-e.pos); if(d<best)
{best=d; tgt=&s;}
            }
            if (tgt){ e.st=UnitState::Attack; /* treat
structure as a proxy: set goal near pos; in real code,
allow unit->structure attack */ }
            w.units.push_back(e);
        }
    }
};

// ============== 11) Game Loop / State ===============
struct Game {
    App app; Assets A; World W; Input input; Selection
sel; Mover mover; Economy eco; Combat combat; Build
build; EnemyAI enemyAI;
    bool boot(){
        if(!app.init()) return false;
        A.load(); W.map.generate();
        // Seed player HQ, a worker, a resource node, and
enemy HQ
        W.structures.push_back( Structure{ Vec{100,100},
Team::Player, 600, "HQ", A.bmp["hq"] } );
        W.units.push_back( Unit{ Vec{140,140}, {},
Team::Player, 100, 8, 22, UnitState::Idle, {}, 0, 60,
ResourceType::Wood, &W.structures.back(), nullptr,
nullptr, A.bmp["unit"] } );
        W.nodes.push_back( ResourceNode{ Vec{320,200},
600, ResourceType::Wood, A.bmp["tree"] } );
        W.structures.push_back( Structure{ Vec{1000,500},
Team::Enemy, 600, "HQ", A.bmp["hq"] } );
        return true;
    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    // Right-click command resolution: move / gather /
attack / build place
    void issueRightClick(Vec worldPx){
        // If clicking a resource → order gather for
selected workers
        ResourceNode* rn=nullptr; for(auto& n:W.nodes)
if(len(n.pos-worldPx)<20) { rn=&n; break; }
        Unit* enemy=nullptr; for(auto& u:W.units) if(u.
team==Team::Enemy && len(u.pos-worldPx)<20) { enemy=&u;
break; }

        for(auto& u:W.units) if(u.selected &&
u.team==Team::Player){
            if (rn){ u.st=UnitState::Gather;
u.targetNode=rn; u.goal=rn->pos; }
            else if (enemy){ u.st=UnitState::Attack;
u.targetEnemy=enemy; u.goal=enemy->pos; }
            else { u.st=UnitState::Move; u.goal=worldPx;
}
        }
    }

    void update(float dt){
        if (input.quit) { W.gameOver=true; return; }
        enemyAI.update(W, dt);

        // Selection drag
        if (!input.dragging && input.lmb){ /* no-op */ }
        if (!input.lmb && input.dragging==false) { /*
release happened */ }

        // Move units & run state machines
        for(auto& u:W.units){
            switch(u.st){
                case UnitState::Move:    mover.step(u,
W.map, dt); break;
                case UnitState::Gather:  eco.update(u, W,
dt); mover.step(u, W.map, dt); break;
                case UnitState::Return:  eco.update(u, W,
dt); mover.step(u, W.map, dt); break;
```

https://doi.org/10.15215/remix/9781998944224.01

```
                case UnitState::Attack:  combat.tick(u,
W, dt); mover.step(u, W.map, dt); break;
                default: break;
            }
        }
        // Cleanup dead units (very basic)
        W.units.erase(std::remove_if(W.units.begin(),
W.units.end(), [](const Unit& u){ return u.hp<=0; }),
W.units.end());
    }

    void draw(){
        al_clear_to_color(al_map_rgb(25,35,25));
        W.map.draw(A, W.cam.x, W.cam.y);
        // Draw nodes
        for(auto& n:W.nodes) al_draw_filled_
circle(n.pos.x-W.cam.x, n.pos.y-W.cam.y, 6,
al_map_rgb(60,160,60));
        // Draw structures
        for(auto& s:W.structures) al_draw_filled_
rectangle(s.pos.x-12-W.cam.x, s.pos.y-12-W.cam.y, s.pos.
x+12-W.cam.x, s.pos.y+12-W.cam.y, s.team==Team::Player?
al_map_rgb(70,120,255):al_map_rgb(220,70,70));
        // Draw units
        for(auto& u:W.units){
            ALLEGRO_COLOR c = (u.team==Team::Player)?
al_map_rgb(120,200,255):al_map_rgb(255,120,120);
            al_draw_filled_circle(u.pos.x-W.cam.x,
u.pos.y-W.cam.y, 6, c);
            if(u.selected) al_draw_circle(u.pos.x-W.
cam.x, u.pos.y-W.cam.y, 9, al_map_rgb(255,255,0), 2);
        }
        // HUD
        std::string hud = "Wood: "+std::to_string(W.
wood)+"  Stone: "+std::to_string(W.stone);
        al_draw_text(A.font["ui"], al_map_
rgb(255,255,255), 10, 10, 0, hud.c_str());
        al_flip_display();
    }
```

https://doi.org/10.15215/remix/9781998944224.01

```
    // Basic event pump: selection & commands
    void run(){
        bool selecting=false; Vec selStart;
        while(true){
            ALLEGRO_EVENT ev; al_wait_for_event(app.q,
&ev);
            if (ev.type==ALLEGRO_EVENT_TIMER){
update(1.0f/60.0f); draw(); continue; }
            input.handle(ev);
            if (input.quit) break;

            if (ev.type==ALLEGRO_EVENT_MOUSE_BUTTON_DOWN
&& ev.mouse.button==1){
                selecting=true; selStart={ (float)
ev.mouse.x,(float)ev.mouse.y };
            }
            if (ev.type==ALLEGRO_EVENT_MOUSE_BUTTON_UP &&
ev.mouse.button==1){
                selecting=false;
                sel.boxSelect(W, selStart, input.mouse,
W.cam.x, W.cam.y);
            }
            if (ev.type==ALLEGRO_EVENT_MOUSE_BUTTON_UP &&
ev.mouse.button==2){
                Vec worldPx = input.mouse + Vec{W.cam.x,
W.cam.y};
                issueRightClick(worldPx);
            }
            // keyboard quick build (B to start placing;
Enter to confirm)
            if (ev.type==ALLEGRO_EVENT_KEY_DOWN){
                if (ev.keyboard.keycode==ALLEGRO_KEY_B){
build.placing=true; build.kind="Barracks"; }
                if (ev.keyboard.keycode==ALLEGRO_KEY_
ENTER){ build.confirm(W, A); }
            }
            if (build.placing && ev.type==ALLEGRO_EVENT_
MOUSE_AXES){
                build.ghostPx = input.mouse + Vec{W.
cam.x, W.cam.y};
```

https://doi.org/10.15215/remix/9781998944224.01

```
            }
        }
    }
};

int main(){ Game g; if(!g.boot()) return 1; g.run();
g.app.shutdown(); return 0; }
```

TASKS FOR COMPLETION AND IMPROVEMENT

1. **Resource Gathering:** Implement logic for units to gather resources and bring them back to structures.
2. **Building Structures:** Allow the player to build new structures using gathered resources.
3. **Combat Mechanics:** Implement combat between player units and enemy units.
4. **AI Behavior:** Improve enemy AI to make the game more challenging.
5. **User Interface:** Add a UI to display resources, unit health, and other game information

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Exercises

1. **Understanding Selection and Commands**
   Explain how box selection differs from single-click selection in an RTS. Why is it important to support both, and how does modifier-key selection (such as Shift-click) improve usability?
2. **Coordinate Systems in RTS Games**
   Describe the difference between screen coordinates and world coordinates. Why is accurate conversion between these coordinate spaces essential for unit selection, movement commands, and structure placement?
3. **Finite-State Unit Behavior**
   Review the UnitState enumeration. For each state listed (Idle, Move, Gather, Return, Attack, Build), briefly describe what conditions cause a unit to enter or leave that state.

https://doi.org/10.15215/remix/9781998944224.01

4. **Economy Flow Analysis**
   Trace the full lifecycle of a resource-gathering unit from locating a resource node to depositing resources at a structure. Identify which parts of the code manage each step of this process.

Homework Questions

1. **RTS Complexity vs. Turn-Based Games**
   Compare the challenges of implementing an RTS with those of a turn-based strategy game. Which systems become more complex in real time, and why?
2. **Pathfinding Trade-Offs**
   Discuss the limitations of the current "greedy steering" movement approach. Under what conditions does this approach fail, and how does grid-based A* pathfinding address those shortcomings?
3. **User Interface and Cognitive Load**
   RTS games present a large amount of information to players. What UI elements are most critical for preventing player confusion, and how should information priority be determined?
4. **AI Design Considerations**
   Why is it often preferable for enemy AI in strategy games to be predictable but efficient rather than optimal? How does this affect player enjoyment and perceived difficulty?

Programming Projects

1. **Resource Gathering System (Core Project)**
   Complete and refine the resource-gathering logic so that units correctly locate resources, gather them over time, return them to deposits, and resume gathering if resources remain. Add visual feedback for carrying resources.
2. **Structure Construction System**
   Extend the building and placement system to support multiple structure types. Implement construction progress indicators and prevent overlapping or invalid placement more robustly.
3. **Combat Mechanics Expansion**
   Enhance combat by adding armor or damage mitigation, ranged versus melee attacks, and visual attack indicators. Ensure combat resolves consistently for both player and enemy units.
4. **Improved Enemy AI**
   Expand the enemy AI beyond basic harassment. Add decision-making logic that allows the AI to choose between economic expansion, defense, or aggression based on game state.

https://doi.org/10.15215/remix/9781998944224.01

5. **User Interface and HUD Enhancements**
   Build a richer HUD that displays selected unit statistics, health bars, resource counts, and active production queues. Optionally add a minimap with click-to-pan functionality.
6. **Fog of War Implementation (Advanced)**
   Implement a fog-of-war system that tracks explored versus visible areas. Mask terrain and units outside current vision and update visibility dynamically as units move.
7. **Save and Load System (Advanced)**
   Create a save/load feature that serializes the game state, including map layout, resources, units, structures, and AI state. Restore the game accurately from saved data.

8.11. Utilities Game

Utilities games are a unique category of software that blends gaming elements with practical tools to enhance productivity, creativity, or system performance. These games often provide a fun and interactive way to accomplish tasks or improve your workflow.

For this project, we will create a basic focus timer game that helps users manage their work sessions and breaks. The game will have a timer that counts down, and the user can start, pause, and reset the timer.

8.11.1. Game Design

Game Concept

Genre: Utilities game (focus timer).

Objective: The user sets a timer for work sessions and breaks to improve productivity.

Core Components

Timer: The core functionality of the application

Attributes: Total time, remaining time, state (stopped, running, paused)

Function: Counts down accurately based on elapsed time

Control Buttons: User interaction elements

Types: Start, Pause, Reset, Skip

Function: Control timer state transitions

Display: Visual output of timer state

Attributes: Position, font, colour

Function: Shows remaining time in MM:SS format

Session Tracking System: Light gamification element

Attributes: Completed sessions, total focus time

https://doi.org/10.15215/remix/9781998944224.01

Function: Encourages consistent use

Persistence System: Stores user data

Content: Durations, preferences, session statistics

Purpose: Maintains continuity across application sessions

Key Techniques and Features to Have

Designing a focus timer as a utility-style game combines accurate timekeeping, light gamification, and a clean, reliable user interface. While mechanically simple, correctness and robustness are essential for user trust. The following features define a solid, extensible baseline for a focus timer implemented with Allegro:

1. **Accurate Timekeeping and Drift Control**
   The timer should compute remaining time using absolute elapsed time (via al_get_time()), not frame counts, to prevent drift. Pause and resume behavior should be handled by tracking cumulative paused duration so the countdown always reflects real elapsed time.
2. **Clear State Machine**
   Timer behavior should be governed by an explicit state machine with states such as stopped, running, paused, and completed. Optional Pomodoro cycling (work and break phases) can be layered on top. Clear state transitions simplify logic and reduce errors.
3. **Configurable Durations and Presets**
   Users should be able to customize work and break durations and save common presets: for example, 25/5 (25 minutes of work with a 5-minute break) or 50/10. Quick-set buttons for common intervals enable fast session starts with minimal interaction.
4. **Responsive UI Components**
   The interface should offer responsive controls for starting, pausing, resetting, and skipping sessions, with visual feedback for hover and press states. Keyboard shortcuts should mirror common actions. The display should include a clear MM:SS readout and a visible progress indicator.
5. **Nonintrusive Notifications**
   Session completion should be indicated through subtle audio or visual cues that do not disrupt focus. Users should control volume and muting. Optional desktop notifications may be used when the app is not in focus.
6. **Session Tracking and Light Gamification**
   The timer may track completed sessions, total focus time, and usage streaks. Optional achievements or badges can motivate continued use without distracting from the tool's primary purpose.

https://doi.org/10.15215/remix/9781998944224.01

7. **Persistence of Settings and Statistics**
   User preferences (durations, theme, audio) and usage statistics should persist across runs. Storing this data in a small external configuration file ensures continuity and reliability.
8. **Themes and Accessibility**
   The application should support light and dark themes and colour-blind-friendly palettes. Accessibility features such as large text, keyboard-only navigation, and visible focus indicators broaden usability.
9. **Robust Handling of Edge Cases**
   The timer should remain accurate if the window loses focus, the system sleeps, or rendering pauses. Remaining time must never underflow, and transitions into the completed state should be clear and deterministic.

8.11.2. Implementation

Key Operations in the Game

1. **Event Intake (Mouse/Keyboard) and Intent Mapping**
   - Poll Allegro events each frame.
   - Map input to actions: Start/Pause, Reset, Skip, Preset Selection, Theme Toggle, Mute.
   - Support keyboard shortcuts (e.g., space bar = toggle run/pause, R = reset, M = mute).
2. **Timebase Update (Drift-Resistant Countdown)**
   - Maintain sessionStartTime, pausedAccumulated, and lastTick.
   - Per timer tick: elapsed = al_get_time() − sessionStartTime − pausedAccumulated; remaining = totalTime − elapsed;
   - When pausing, capture pauseStart; when resuming, add to pausedAccumulated.
3. **State Transitions**
   - Stopped → Running: Initialize timestamps. (*Optional:* Play start cue.)
   - Running → Paused: Freeze countdown by recording pauseStart.
   - Paused → Running: Adjust pausedAccumulated.
   - Running → Completed: When remaining ≤ 0, clamp to 0, set Completed, trigger SFX/flash. (*Optional:* Advance to the next phase [break/work] if Pomodoro mode is enabled.)
4. **Button Hit-Testing and UI Feedback**
   - On mouse down, test cursor against button rectangles; set pressed state.

https://doi.org/10.15215/remix/9781998944224.01

- On mouse up, if still inside the same button, fire the action; update hover/pressed visuals.

5. **Time Formatting and Progress Visualization**
   - Convert seconds to MM:SS (zero-padded).
   - Compute progress ratio p = (totalTime − remaining)/totalTime for a bar or ring.
   - Animate subtle easing on the progress indicator for polish.
6. **Rendering Pass (per Frame)**
   - Clear background (theme colour).
   - Draw time read-out, progress bar/ring, and buttons with current states.
   - Draw status text (Running/Paused/Break/Completed). (*Optional:* Draw session counters.)
7. **Audio/Notification Triggers**
   - On start, pause, resume, complete, and play appropriate SFX if enabled.
   - *Optional:* On complete, display an overlay or desktop notification.
8. **Persistence (Load/Save)**
   - On start-up, load preferences (durations, theme, audio), and stats (sessions, total minutes).
   - On completion/reset/settings change, save back to disk.
9. **Edge-Case Handling**
   - If the app regains focus or the system clock jumps, recompute remaining from absolute time.
   - Ensure timer cannot underflow; clamp and transition once.
10. ***Optional:* Pomodoro Cycle Manager**
    - Maintain a small state machine: Work → Break → Work → . . . → Long Break after N cycles.
    - Auto-start the next phase (configurable) or wait for a Start click.

Structure of the Game Program

1. **Initialization**
   - Initializes Allegro and its add-ons and installs keyboard and mouse input.
2. **Definition of Timer**
   - A structure to keep track of the total time, remaining time, and running state.
3. **Definition of Buttons**
   - Structures to define the start, pause, and reset buttons.

https://doi.org/10.15215/remix/9781998944224.01

4. **Definition of Drawing Functions**
   - Functions to draw the timer and buttons on the screen.
5. **Definition of Input Handling**
   - Captures mouse input to start, pause, and reset the timer.
6. **Game Loop**
   - Runs the main game loop to update and render the game.
7. **Shutdown**
   - Clean release of Allegro resourcesFinishing

Coding the Game

FOCUS_TIMER.CPP

```
#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <allegro5/allegro_primitives.h>
#include <iostream>
#include <cstdio>

const int SCREEN_WIDTH  = 800;
const int SCREEN_HEIGHT = 600;

struct Timer {
    float totalTime;
    float remainingTime;
    bool running;
};

struct Button {
    float x, y, width, height;
    const char* label;
};

Timer timer = {1500.0f, 1500.0f, false}; // 25 minutes
Button startButton = {100, 500, 100, 50, "Start"};
Button pauseButton = {250, 500, 100, 50, "Pause"};
Button resetButton = {400, 500, 100, 50, "Reset"};

bool initAllegro() {
    if (!al_init()) return false;
```

https://doi.org/10.15215/remix/9781998944224.01

```
    al_init_font_addon();
    al_init_ttf_addon();
    if (!al_install_keyboard()) return false;
    if (!al_install_mouse()) return false;
    if (!al_init_primitives_addon()) return false; //
needed for draw_filled_rectangle
    return true;
}

bool isButtonClicked(const Button& button, int mx, int
my) {
    return mx >= button.x && mx <= button.x + button.
width &&
           my >= button.y && my <= button.y + button.
height;
}

void drawButton(const Button& button, ALLEGRO_FONT* font)
{
    // button body + border
    al_draw_filled_rectangle(button.x, button.y,
                             button.x + button.width,
button.y + button.height,
                             al_map_rgb(200, 200, 200));
    al_draw_rectangle(button.x, button.y,
                      button.x + button.width, button.y +
button.height,
                      al_map_rgb(0, 0, 0), 2.0f);

    // center label
    int th = al_get_font_line_height(font);
    float cx = button.x + button.width  / 2.0f;
    float cy = button.y + button.height / 2.0f–th / 2.0f;
    al_draw_text(font, al_map_rgb(0, 0, 0), cx, cy,
ALLEGRO_ALIGN_CENTER, button.label);
}

void updateTimer(Timer& t, float dt) {
    if (t.running && t.remainingTime > 0.0f) {
        t.remainingTime–= dt;
```

https://doi.org/10.15215/remix/9781998944224.01

```
        if (t.remainingTime <= 0.0f) {
            t.remainingTime = 0.0f;
            t.running = false;
            // (Optional) play a sound or flash the
screen here.
        }
    }
}

void drawTimer(const Timer& t, ALLEGRO_FONT* font) {
    int total = (t.remainingTime > 0.0f) ? static_
cast<int>(t.remainingTime) : 0;
    int minutes = total / 60;
    int seconds = total % 60;
    char timeStr[6];
    std::snprintf(timeStr, sizeof(timeStr), "%02d:%02d",
minutes, seconds);
    al_draw_text(font, al_map_rgb(0, 0,
0), SCREEN_WIDTH / 2.0f, SCREEN_HEIGHT /
2.0f–al_get_font_line_height(font)/2.0f, ALLEGRO_ALIGN_
CENTER, timeStr);
}

int main() {
    if (!initAllegro()) {
        std::cerr << "Failed to initialize Allegro.\n";
        return–1;
    }

    ALLEGRO_DISPLAY* display = al_create_display(SCREEN_
WIDTH, SCREEN_HEIGHT);
    if (!display) { std::cerr << "Display creation
failed.\n"; return–1; }

    ALLEGRO_EVENT_QUEUE* queue = al_create_event_queue();
    ALLEGRO_TIMER* ticker = al_create_timer(1.0 / 60.0);
    if (!queue || !ticker) {
        std::cerr << "Queue/timer creation failed.\n";
        if (ticker) al_destroy_timer(ticker);
        if (queue) al_destroy_event_queue(queue);
```

https://doi.org/10.15215/remix/9781998944224.01

```
        al_destroy_display(display);
        return-1;
    }

    ALLEGRO_FONT* font = al_load_ttf_font("arial.ttf",
32, 0);
    if (!font) { // graceful fallback
        font = al_create_builtin_font();
        if (!font) { std::cerr << "Font creation
failed.\n"; return-1; }
    }

    al_register_event_source(queue,
al_get_display_event_source(display));
    al_register_event_source(queue,
al_get_timer_event_source(ticker));
    al_register_event_source(queue,
al_get_mouse_event_source());

    al_start_timer(ticker);

    bool running = true;
    while (running) {
        ALLEGRO_EVENT ev;
        al_wait_for_event(queue, &ev);

        if (ev.type == ALLEGRO_EVENT_DISPLAY_CLOSE) {
            running = false;
        } else if (ev.type == ALLEGRO_EVENT_TIMER) {
            updateTimer(timer, 1.0f / 60.0f);
            al_clear_to_color(al_map_rgb(255, 255, 255));
            drawTimer(timer, font);
            drawButton(startButton, font);
            drawButton(pauseButton, font);
            drawButton(resetButton, font);
            al_flip_display();
        } else if (ev.type == ALLEGRO_EVENT_MOUSE_BUTTON_
DOWN) {
            if (isButtonClicked(startButton, ev.mouse.x,
ev.mouse.y)) {
```

https://doi.org/10.15215/remix/9781998944224.01

```
                timer.running = true;
            } else if (isButtonClicked(pauseButton,
ev.mouse.x, ev.mouse.y)) {
                timer.running = false;
            } else if (isButtonClicked(resetButton,
ev.mouse.x, ev.mouse.y)) {
                timer.remainingTime = timer.totalTime;
                timer.running = false;
            }
        }
    }

    if (font) al_destroy_font(font);
    if (ticker) al_destroy_timer(ticker);
    if (queue) al_destroy_event_queue(queue);
    if (display) al_destroy_display(display);
    return 0;
}
```

TASKS FOR COMPLETION AND IMPROVEMENT

1. **Customizable Timer:** Allow the user to set custom work and break durations.
2. **Sound Alerts:** Add sound effects to notify the user when the timer ends.
3. **Visual Themes:** Add different visual themes to make the game more appealing, load a background image, or modify the font and sizes used for the buttons and time display.
4. **Statistics:** Track and display statistics such as total work time and number of sessions completed.

EXERCISES, HOMEWORK QUESTIONS, AND PROJECTS

Exercises

1. **Understanding Timekeeping Accuracy**
   Explain why computing the remaining time using al_get_time() is more reliable than decrementing a counter each frame. Describe a

https://doi.org/10.15215/remix/9781998944224.01

scenario in which frame-based updates could cause noticeable timer drift.

2. **Timer State Transitions**
   Draw a state diagram showing the transitions between the stopped, running, paused, and completed states of the timer. Indicate which user inputs or conditions cause each transition.
3. **UI Interaction Analysis**
   Review the button-handling code. Describe how hit-testing works and explain why separating input detection (isButtonClicked) from rendering (drawButton) is beneficial.

Homework Questions

1. **Design Robustness**
   Consider what happens if the application window loses focus or the system temporarily freezes. How would you modify the current implementation to ensure the timer remains accurate when the application resumes?
2. **State Machines in Utility Games**
   Compare the timer's state machine to those used in traditional games (such as menus or combat systems). What similarities and differences do you observe?
3. **Persistence and User Trust**
   Why is saving user preferences and statistics important in a utilities game? What expectations might users have if this data were lost between sessions?

Programming Projects

1. **Customizable Timer Durations (Core Project)**
   Extend the timer so users can define custom work and break durations. Add UI elements (buttons or keyboard shortcuts) to select common presets (e.g., 15, 25, 45 minutes) and apply them at runtime.
2. **Sound Alerts and Notifications**
   Integrate Allegro's audio subsystem to play a sound when the timer reaches zero. Add a mute toggle and allow users to control alert volume. Optionally, implement a brief visual flash effect when a session is completed.
3. **Session Statistics and History Tracking**
   Track additional statistics such as total accumulated work time, number of completed sessions, and daily streaks. Display these statistics in a simple summary panel or overlay.

https://doi.org/10.15215/remix/9781998944224.01

4. **Persistent Configuration File (Advanced)**
   Store user preferences and statistics in an external configuration file (such as JSON or INI format). Load this file at start-up and save changes when the application exits or settings are modified.
5. **Accessibility Enhancements (Optional)**
   Add accessibility features such as a large-text mode, keyboard-only navigation, and high-contrast colour options. Explain how each modification improves usability.

https://doi.org/10.15215/remix/9781998944224.01

This page intentionally left blank

References and Resources

Official Resources

Allegro 5 Documentation

Allegro 5 Official Wiki (https://liballeg.org/a5docs/trunk/): Comprehensive documentation for functions, examples, and concepts.

Allegro 5 GitHub Repository (https://github.com/liballeg/allegro5): Source code and issue tracking.

Allegro.cc Forums (https://www.allegro.cc/forums/): Active community for troubleshooting and discussions.

Allegro 5 Keyboard API (https://liballeg.org/a5docs/trunk/keyboard.html). Allegro 5 documentation page for Keyboard API.

Books

Beginner-Friendly

Allegro 5 Game Programming by Jonathan S. Harbour: Step-by-step guide to building games with Allegro 5.

Beginning C++ Game Programming by John Horton: While not Allegro-specific, it teaches C++ concepts applicable to game devs.

Advanced

Game Programming Patterns by Robert Nystrom: Explores design patterns applicable to Allegro-based projects.

Tutorials and Courses

Free Tutorials

Official Allegro Tutorials (https://liballeg.org/a5docs/trunk/getting_started.html): Basic setup and example projects.

https://doi.org/10.15215/remix/9781998944224.01

Classic tutorials for Allegro (https://liballeg.org/docs.html): Useful tutorials for learning Allegro. You can ask an AI assistant about additional tutorials for Allegro 4/5.

YouTube Channels

Allegro 5 Tutorial Series by Sonar Systems (https://www.youtube.com/playlist?list=PLRtjMdoYXLf4od_bOKN3WjAPr7snPXzoe).

Making Games with Allegro 5 by VoidRealms (https://www.youtube.com/playlist?list=PLvv0ScY6vfd9wBflF0f6ynlDQuaeKYzyc).

Sample Projects and Code Examples

GitHub Repositories

Allegro Examples (https://github.com/liballeg/allegro5/tree/master/examples): Official example code (sprites, audio, input).

Simple 2D Game Template (https://github.com/j3soon/Allegro5Template): Basic framework for starting projects.

Allegro Pong Clone (https://github.com/Jshbrck/Pong): Classic Pong implementation.

Forums and Communities

Discussion Platforms

Allegro.cc Forums (https://www.allegro.cc/forums/): The primary hub for Allegro developers.

Reddit: r/gamedev (https://www.reddit.com/r/gamedev/): General game dev discussions.

Stack Overflow: Allegro Tag (https://stackoverflow.com/questions/tagged/allegro5). Site for asking and getting advice on programming questions.

Complementary Libraries

Box2D (https://box2d.org/): Physics engine integration.

OpenAL (https://www.openal.org/): Advanced audio (Allegro's audio module is sufficient for basics).

OpenGL (https://www.opengl.org/): Advanced graphics (Allegro's audio module is sufficient for basics).

https://doi.org/10.15215/remix/9781998944224.01