

Warsaw University of Technology

FACULTY OF ELECTRONICS AND
INFORMATION TECHNOLOGY



Bachelor's diploma thesis

in the field of study Computer Science
and specialization Computer Systems and Networks

Creating multiplatform applications with creation of match three game
engine as an example

thesis number in the Faculty thesis register 103B-ISA-IN/307585/1202985

Krzysztof Stefan Rudnicki

student record book number 307585

thesis supervisor
dr hab. inż. Tomasz Martyn

konsultacje
Tomasz Martyn

WARSAW 2023

Streszczenie pracy

Tworzenie aplikacji wieloplatformowych na przykładzie silnika gier z gatunku "match 3"

Rynek systemów operacyjnych na PC jest podzielony na trzech głównych graczy, tworzy to zapotrzebowanie na silniki do gier, które są wszechstronne i działają tak samo, niezależnie od platformy. Niniejsza praca inżynierska przedstawia projekt i implementację nowego silnika do gier opracowanego z myślą o grach typu "dopasuj 3", gatunku który jest zarówno popularny, jak i stosunkowo łatwy do wdrożenia, mimo to nadal nie ma wyspecjalizowanego silnika open-source pod gry "dopasuj 3". Silnik został zbudowany w oparciu o bibliotekę graficzną OpenGL i system zarządzania oknami glfw, zapewniając zaawansowane możliwości graficzne przy jednoczesnym zachowaniu kompatybilności między platformami. Silnikowi przyświecają dwa cele: prostota obsługi dla deweloperów i solidna wydajność. Architektura skupia się na wykorzystaniu dobrych praktyk, nowoczesnego kodu C++ i sprawieniu że czytanie kodu silnika będzie łatwe i zrozumiałe dla developerów. W rezultacie twórcy gier mogą skupić się na tworzeniu gier typu "dopasuj 3" bez konieczności nauki programowania, umożliwiając artystom bez zaplecza technicznego, którzy chcą stworzyć wyjątkowe doświadczenie. Niniejsza praca opisuje motywacje, tło, dokumentację i przypadki użycia na trzech głównych platformach: Linux, Windows i MacOS. Silnik demonstruje użyteczność, możliwość dostosowania do różnych platform i zapewnia narzędzia niezbędne do tworzenia prostych gier typu "dopasuj 3". Niniejsza praca stara się wypełnić lukę w wyspecjalizowanych silnikach gier wideo open-source. {Multiplatformowość, Gry, Silniki do gier, Gry "dopasuj 3" OpenGL, GLFW, Grafika, Design, Prostota, Linux, Windows, MacOS, C++, Dobre praktyki kodowania}

Thesis abstract

Creating multiplatform applications with creation of match three game engine as an example
With three major operating systems on PC, there exists a need for game engines that are versatile and platform-agnostic. This thesis introduces the design and implementation of a new game engine developed with match-three games in mind, which is a game genre that is both popular and relatively easy to implement, despite this there is still no specialized open-source match three game engine. The engine is built upon the foundations of the OpenGL graphics library and the glfw windowing system, ensuring advanced graphics capabilities while retaining platform compatibility. There are two goals for the engine: simplicity of use for developers, and robustness in performance. The architecture focuses on using good practices, modern C++ code and making the code developer friendly. As a result, game developers can focus on crafting match-three games without need to learn programming, empowering artists without technical background who want to create an unique experience. This work provides motivation, background, documentation and use-case across three major platforms: Linux, Windows, and MacOS. The engine demonstrates usability, adaptability to different platforms, and provides the tools necessary for creation of simple match-three games. This thesis is trying to fill the gap of specialized open-source video game engines. {Multiplatform, Gaming, Game engines, Match-three games, OpenGL, GLFW, Graphics, Platform-agnostic, Design, Robustness, Simplicity, Linux, Windows, MacOS, C++, Good coding practices}

Contents

1	Introduction	8
1.1	Background and Motivation	8
1.2	Scope of the Thesis	9
1.2.1	Game Engine Focus: Match Three Genre	9
1.2.2	Multiplatform Development	9
1.2.3	Tool and Library Integration	9
1.2.4	Theoretical and Practical Exploration	9
1.2.5	Limitations	9
1.3	Objectives of the Thesis	9
1.3.1	Design of a Cross-Platform Core Architecture	10
1.3.2	Effective Integration of Libraries	10
1.3.3	Development of Core Match Three Mechanics	10
1.3.4	Providing a Blueprint for Future Development	10
1.4	Match Three Games and Engines	10
1.4.1	Origins and Early Forerunners	10
1.4.2	Mainstream Adoption and Innovations	10
1.4.3	Technological Influences and Platform Diversification	11
1.4.4	Enduring Appeal and Contemporary Significance	11
1.5	Overview of Multiplatform Game Development	11
1.5.1	The Need for Multiplatform Development	12
1.5.2	Challenges in Multiplatform Development	12
1.5.3	Tools and Libraries for Cross-Platform Development	12
1.5.4	Strategies for Effective Multiplatform Development	12
1.5.5	Future Trajectory of Multiplatform Development	13
1.6	Existing Game Engines and Their Limitations	13
1.6.1	Unity Game Engine	13
1.6.2	Unreal Engine	13
1.6.3	Godot	14
1.6.4	RPG Maker and Ren'Py	14
1.6.5	Limitations	14
2	Basics of Game Engine Design	15
2.1	Core Components of Game Engines	15
2.1.1	Rendering Engine	15
2.1.2	Input Handling	15
2.1.3	Transformations	16
2.2	The Role of OpenGL in Game Development	16
2.2.1	Introduction to OpenGL	16
2.2.2	Rendering Mechanism	16
2.2.3	Shader Programming	16
2.2.4	Extensibility	16
2.2.5	Cross-Platform Development	17

2.2.6	Integration with Other Libraries and Tools	17
2.3	Multiplatform Considerations	17
2.3.1	Platform-Specific Hardware and Software	18
2.3.2	User Input Variations	18
2.3.3	Graphical Rendering Nuances	18
2.3.4	File System and Data Management	18
3	Toolchain and Libraries Overview	19
3.1	Introduction to GLFW	19
3.1.1	Historical Context	19
3.1.2	GLFW's Core Competencies	19
3.1.3	Integration with OpenGL	19
3.1.4	Cross-Platform Capabilities	20
3.1.5	Community and Support	20
3.2	Role of GLAD in OpenGL Loading	20
3.2.1	The Need for Loading OpenGL Extensions	20
3.2.2	GLAD's Functionality in Extension Loading	20
3.2.3	Integration with GLFW and OpenGL	22
3.3	Handling text and images with FreeType, ft2build and stb_image	22
3.3.1	Introduction to FreeType	22
3.3.2	Key Capabilities of FreeType	22
3.3.3	Understanding ft2build	22
3.3.4	Introduction to stb_image	23
3.3.5	Key Capabilities of stb_image	23
4	Design and Architecture of the Match Three Engine	24
4.1	Game Loop and State Management	24
4.1.1	The Anatomy of a Game Loop	24
4.1.2	Importance of State Management	24
4.1.3	Interlinking Game Loop and State Management	25
4.1.4	Challenges and Considerations	25
4.1.5	Role in the Multiplatform Match Three Game Engine	25
4.2	Asset Management and Rendering	25
4.2.1	What are Game Assets?	25
4.2.2	Rendering: Bringing Assets to Life	26
4.2.3	Cross-Platform Considerations in Asset Management	26
4.2.4	Role in the Multiplatform Match Three Game Engine	26
4.3	Event Handling and User Input	26
4.3.1	Understanding User Input and Events	27
4.3.2	Event Polling vs. Event Callbacks	27
4.3.3	Role in the Multiplatform Match Three Game Engine	27
5	Cross-Platform Development Challenges and Solutions	28
5.1	Operating System Variabilities	28
5.1.1	Core Architectural Differences	28
5.1.2	Graphical User Interface (GUI) Divergence	28
5.1.3	Driver Support	29
5.1.4	Middleware & Third-party Libraries	29
5.1.5	Implications for the Multiplatform Match Three Game Engine	29
5.2	Addressing Hardware and Driver Differences	29
5.2.1	A Landscape of Diverse Hardware	29
5.2.2	Drivers: The Link to Hardware	30

5.2.3	Challenges Posed to Game Engines	30
5.2.4	Abstraction Layers in Match Three Game Engine	30
6	Implementation Details	31
6.1	Requirements	31
6.1.1	Functional Requirements (FR)	31
6.1.2	Non-functional Requirements (NFR)	32
6.2	Core Engine Implementation	32
6.2.1	Foundational Principles	32
6.3	Integration of Libraries and Tools	33
6.3.1	Choice of graphic rendering API	33
6.3.2	Choice of OpenGL Library	34
6.3.3	GLAD and OpenGL Extension Handling	34
6.3.4	Text and Image Rendering with FreeType, stb_image, and ft2build	34
6.3.5	Core Modules	34
6.3.6	Main function	35
6.3.7	Constants file	36
6.4	Game class	37
6.4.1	Game initialization	38
6.4.2	Game update	40
6.4.3	Controls	40
6.4.4	Rendering	42
6.5	Game level class	44
6.5.1	Highlighting, selecting and swapping tiles	45
6.5.2	Matching logic	46
6.5.3	Processing matches	48
6.5.4	Game objects	48
6.5.5	Text Renderer	50
6.5.6	Resource Manager	53
6.5.7	Shader class	54
6.5.8	Dependency Management	55
7	Practical Use of Engine	57
7.1	Initial Setup and Configuration	57
7.1.1	Choosing assets	57
7.1.2	Loading assets	60
7.2	Game Concept and Design	61
7.2.1	Game Concept	61
7.2.2	Game Design	61
7.2.3	Core Mechanics and Gameplay	63
7.2.4	User Interface and User Experience	63
7.2.5	Advantages of a Custom-built Engine	64
8	Future Work and Enhancements	66
8.1	Potential Extensions to the Engine	66
8.1.1	Enhanced Graphics and Physics	66
8.1.2	Adaptive Difficulty	66
8.1.3	Real-time Multiplayer	66
8.1.4	More platforms	66

9 Conclusion	67
9.1 Summary of Achievements	67
9.1.1 Multiplatform Support	67
9.1.2 Library Integrations	67
9.1.3 Robust Game Mechanics	67
9.1.4 Practical usage	67
9.1.5 Contributions to the Field of Game Development	67
9.2 Reflection on the Development Process	67
9.2.1 Embracing the Multiplatform Challenge	68
9.2.2 Library Integrations	68
9.2.3 Time management	68
Bibliography	69

Chapter 1

Introduction

1.1 Background and Motivation

Among us, computer science students, video games were—and still are—one of the driving forces pushing us into the field of IT. Video games allow people with technical background to express their artistic side and create visually stunning projects. In particular "Match Three" games, characterized by their intuitive mechanics and satisfying gameplay, have attracted millions of players around the globe. They offer satisfying experience of aligning three or more similar game pieces, often resulting in gratifying chain reactions. This led to success of titles like "Candy Crush Saga" and "Bejeweled", staple titles for both mobile and desktop market.

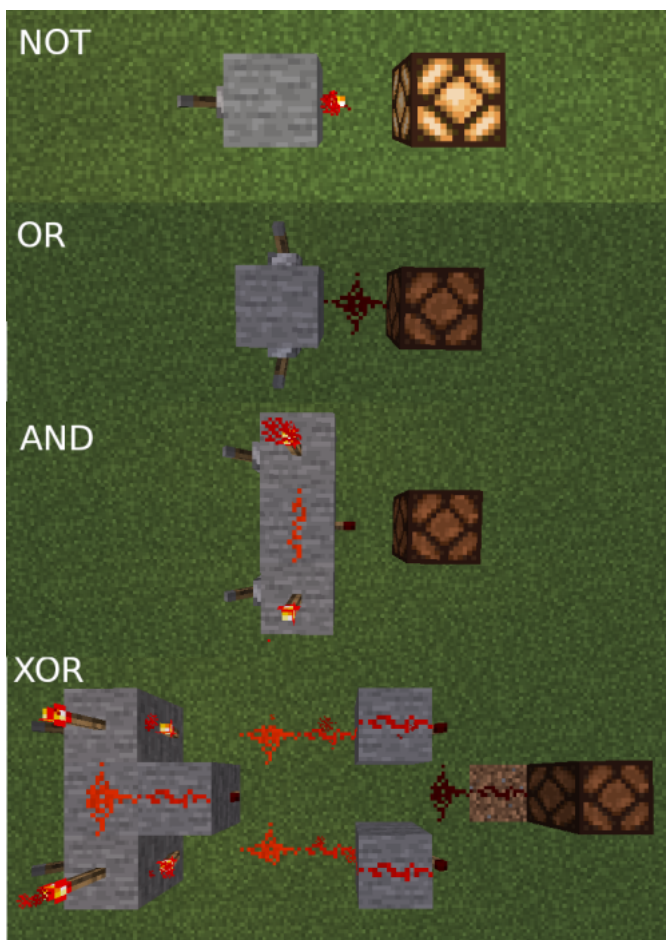


Figure 1: Minecraft redstone offers introduction to logical gates [O'Hanlon \(2023\)](#)

Match three games are from technological point of view relatively easy to make, despite that there is no dedicated open-source game engine designed just for them. This engineering thesis aims to fill this void, to give back to the world of game developers and open-source projects.

Another motivation is a chance to learn one of the most popular graphical API - OpenGL, It is a graphical API which proved useful again and again with video game engines, rendering engines or for desktop applications.

1.2 Scope of the Thesis

Game development is a broad spectrum, This section aims to define to what extent the game engine will be developed

1.2.1 Game Engine Focus: Match Three Genre

While game engines can be created for multiple game genres, this thesis focuses on Match Three genre. This decision comes from its ease of implementation, combined with relatively high popularity and applicability on all platforms. We will focus on core logic, rendering techniques, and controls focused on Match Three games.

1.2.2 Multiplatform Development

Another important part of the thesis is multiplatform game development. We will focus on three major desktop operating systems: Windows, Mac, and GNU/Linux. The challenges associated with creating an uniform gaming experience across these platforms, taking into consideration differences in architecture, functionality and graphics.

1.2.3 Tool and Library Integration

This thesis will make biggest use of OpenGL API together with libraries: GLFW, GLAD, FreeType, stb_image, and ft2build. In order used for platform-independed creation of windows, context and handling inputs, loader for OpenGL, library for rendering text, library for loading images and library for building free type sources. Later in the thesis we will dvelve deeper into those libraries

1.2.4 Theoretical and Practical Exploration

Main portion of this thesis is practical implementation, still we are going to explore theoretical concepts crucial for game engine development.

1.2.5 Limitations

Since the engine aims to be simple and robust, it will not contain every possible feature or optimization like in commercial-grade game engines. The primary intent is to create a solid structure for which further advancements can be made.

In conclusion, this thesis goal is to provide an understanding of multiplatform Match Three game engine development. By setting managable boundaries, it aims to dive deep into specific challenges, tools, and solutions, providing valuable insights for aspiring engine developers. Next sections will further describe the objectives, methodologies, and findings of this exploration into the world of game development.

1.3 Objectives of the Thesis

Clear objectives allow for accomplishments that can be measured. Therefore it is important to define goals that underpin this thesis. In this section, we describe the primary goals of this thesis.

1.3.1 Design of a Cross-Platform Core Architecture

The foremost objective is to design a core multiplatform engine architecture. This demands a good understanding of the operating systems differences, from their system calls to their graphics pipelines. The engine should be modular and scalable, in order to make sure that the core mechanics and logic can be implemented across Windows, Mac, and GNU/Linux platforms.

1.3.2 Effective Integration of Libraries

With numerous libraries at our disposal, an important goal is to combine them into the engine in a cohesive manner. We will make use of GLFW for window management, GLAD for OpenGL function pointers, FreeType and ft2build for text rendering, and stb_image for image handling. In addition to integrating them into an engine, these libraries must operate in such a way that they complement each other to improve the engine's overall performance and capability.

1.3.3 Development of Core Match Three Mechanics

Architecture and libraries used form the skeleton, the Match Three mechanics are the most important part of the engine. The aim is to create algorithms and techniques to take care of typical Match Three functionalities – from matching detection to gravity-induced piece drops and chain reactions.

1.3.4 Providing a Blueprint for Future Development

While the first objective is the engine's development, a broader goal is to expand upon created engine. By addressing challenges and sharing solutions, this thesis hopes to offer an example for engine developers. Whether they wish to enhance this engine or create new game engines. This thesis tries to improve their understanding of the topic

1.4 Match Three Games and Engines

The world of video games is vast and varied, Offering anything from massive multiplayer role playing games to small mobile puzzles. Match Three games offer players a blend of strategy, pattern recognition, and instant gratification. To better understand the significance of our task in creating a multiplatform Match Three game engine, it's important to trace how the genre evolved, understanding its roots, its development, and its appeal.

1.4.1 Origins and Early Forerunners

Match Three games owe their existence to the broader genre of tile-matching video games. The earliest entrants in this category were games like "Tetris" in the 1980s, where players manipulated falling blocks to create complete lines. "Tetris" kickstarted games focused on spatial reasoning and pattern recognition.

Expanding on this formula, 1980s and 1990s saw games like "Columns" and "Dr. Mario". These games introduced the mechanic of matching three or more identical items, but they were still using the falling-block paradigm. The true prototype of modern Match Three mechanics can be credited to "Shariki," a 1994 game where players swapped adjacent pieces to form chains of three or more identical items.

1.4.2 Mainstream Adoption and Innovations

In 2001 game titled "Bejeweled" came out. Simplifying and refining the mechanics of "Shariki", "Bejeweled" was the first mainstream success of Match Three genre. It provided intuitive gameplay, visually appealing graphics and set the benchmark for games that followed.



Figure 2: Bejeweled gameplay [Bejeweled gameplay \(2001\)](#)

The success of "Bejeweled" led to several innovations. Developers experimented with various twists on the core mechanics. Games introduced power-ups, barriers, objectives, and narrative elements. "Candy Crush Saga", released in 2012, integrated a level-based progression system, increasing the genre's complexity and depth.

1.4.3 Technological Influences and Platform Diversification

The evolution of Match Three games is interlocked with technological improvements in the gaming industry. Initially developed for PCs and consoles, the rise of mobile devices opened new opportunities for the genre. The touch interface of smartphones and tablets proved to be an ideal medium for the drag-and-swap mechanics of Match Three games.

At the same time, browser-based games, using technologies like Flash, allowed players to engage with Match Three puzzles without heavy downloads or installations. As technology progressed, the need for engines that worked on multiple platforms – mobile, browser, consoles and desktop – became apparent.

1.4.4 Enduring Appeal and Contemporary Significance

Despite multiple competing genres, Match Three games remain popular. Their success can be attributed to their 'easy to learn, hard to master' trait. The games offer immediate rewards – the satisfaction of creating a match, beauty of pieces disappearing, and the strategic depth of planning several moves ahead. Additionally, the modular nature of their design allows for easy adaptation, whether it's incorporating a narrative, integrating with popular cultures, or tailoring to specific demographics.

In conclusion, the history of Match Three games is a proof to their adaptability and universal charm. They established themselves as one of the most recognizable genres in mobile and desktop gaming; they continue to gather players. This perspective marks the significance of engines tailored to Match Three games specifically.

1.5 Overview of Multiplatform Game Development

Game developers try to reach wider audiences by ensuring that their creations are accessible across a variety of operating systems. Multiplatform game development aims to deliver the same experience to different user bases. This section describes challenges and methodologies connected with multiplatform game development.

1.5.1 The Need for Multiplatform Development

There are three main operating systems, Windows, Mac, and GNU/Linux which together are responsible for over 90% of desktop market share [Desktop Operating System Market Share Worldwide Chart \(2023\)](#), the potential to reach a bigger audience increases when a game is made available across all of these platforms.

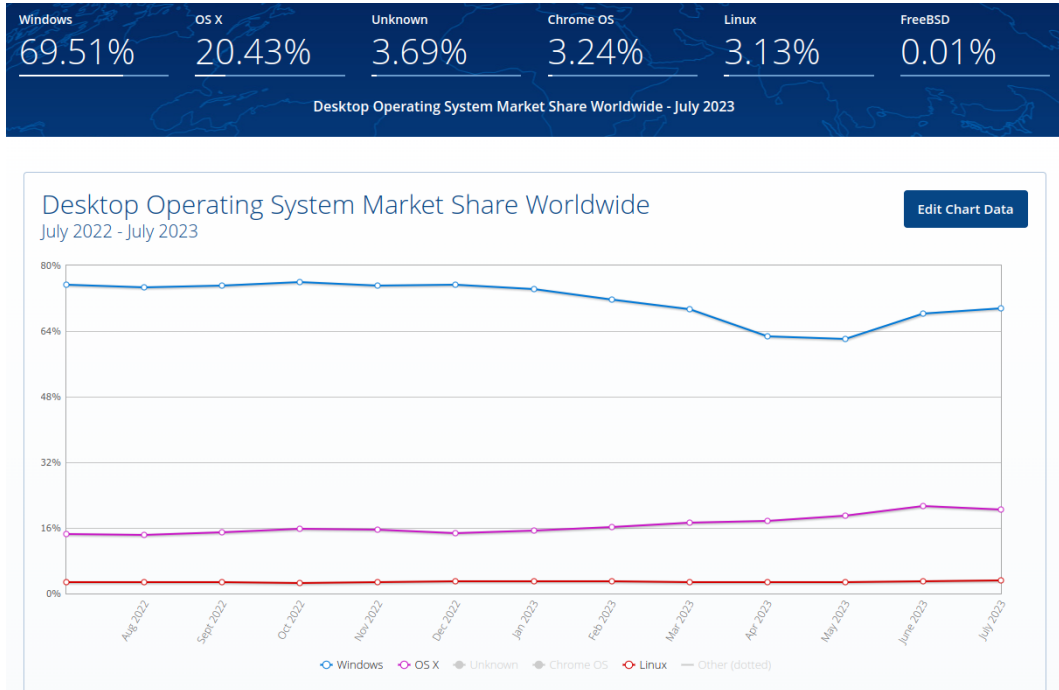


Figure 3: Desktop OS market share worldwide [Desktop Operating System Market Share Worldwide Chart \(2023\)](#)

1.5.2 Challenges in Multiplatform Development

Idea of developing once and deploying everywhere comes with its set of challenges:

- **Hardware Disparity:** Different platforms might have varying hardware specifications, which can influence the game's performance.
- **OS Specific Limitations:** Each operating system operates under its own rules, from system calls to user interface.
- **Toolchain Variability:** Development tools and libraries might have platform-specific versions or may not be available for all platforms.

1.5.3 Tools and Libraries for Cross-Platform Development

In order to face multiplatform development challenges, we use a set of tools and libraries. Libraries like OpenGL offer an unified graphics rendering solution, while GLFW enforces consistent window management and input handling. stb_image, and ft2build are other libraries that offer consistent behavior across platforms for handling images and font rendering.

1.5.4 Strategies for Effective Multiplatform Development

- **Modular Design:** By dividing the game engine into distinct modules, developers can isolate platform-specific code, ensuring that core logic remains unchanged by platform dependencies.
- **Continuous Integration and Testing:** Automated testing across platforms can identify inconsistencies, ensuring that the game offers an uniform experience.

- **Community Engagement:** Using the community that uses the game engine across multiple platforms offers developers insights into existing bugs and potential optimizations

1.5.5 Future Trajectory of Multiplatform Development

With the cloud gaming on the rise, resurrection of web-based games, and increasingly powerful mobile devices, multiplatform development's scope is constantly expanding. There will be newer platforms and more varied devices to handle. The goal will remain the same: ensuring that games offer a consistent, engaging, and seamless experience, independent of the platform.

Multiplatform game development requires technological knowledge and good design. By understanding its challenges and methodologies, developers are better equipped to face-on multiple platforms of modern gaming. This foundation will become crucial later into the development of our multiplatform Match Three game engine.

1.6 Existing Game Engines and Their Limitations

The gaming industry has seen rise of various game engines, each offering a unique set of tools and functionalities for different needs. While these engines have facilitated the creation of iconic games, they also come with certain limitations, especially concerning specific genres or platforms. This section will describe prominent game engines and analyze their shortcomings, providing a reference for our multiplatform Match Three game engine.

1.6.1 Unity Game Engine

Unity is one of the most popular game engines available today. It's known for its versatility, supporting a wide range of platforms, from PCs to consoles to mobile devices.

- **Strengths:** Plenty of community resources, a vast asset store, and an intuitive interface make Unity a popular choice among both beginners and professional developers.
- **Limitations:** Unity's all-purpose nature might introduce unnecessary load for simpler games, like Match Three titles, potentially impacting performance.

1.6.2 Unreal Engine

Epic Games' Unreal Engine stands out for its impressive graphical capabilities, often employed in AAA game titles.

Strengths:

- **Blueprints:** Blueprint visual scripting system allows for easier game development especially for non-technical users
- **Graphics:** Unreal Engine offers top-tier rendering capabilities, which makes it ideal for creating visually stunning games.

Limitations:

- **Learning Curve:** Its advanced features can be overwhelming for beginners.
- **Simpler Genres:** As with Unity, for a simple game like Match Three game, the engine might be too sophisticated, leading to longer load times and increased resource consumption compared to custom solution.

1.6.3 Godot

Godot is the biggest multipurpose game engine that is also free and open-source.

Strengths:

- No licensing: Godot is free to use
- Open: Godot source code can be accessed and modified by anyone
- Community: As with many open source projects community offers huge support for potential developers

Limitations:

- Limited professional presence: While unity and unreal dominate professional market, Godot is a less popular alternative
- Scalability: Godot game engine is not famous for building big game projects

1.6.4 RPG Maker and Ren'Py

RPG Maker and Ren'Py are examples of game engines focusing on one genre of games. RPG Maker focuses on top-down rpg games, while ren'py aims to ease the process of making visual novel games. Both of those engines were used with commercial and critical success, with creation of games like "To The Moon" and "Doki Doki Literature Club". Their limitations are both their strenght and weaknesses, allowing to speedup development process but blocking the developer for leaving the area of interest for those engines.

1.6.5 Limitations

While each game engine has its unique strengths and weaknesses, some common limitations come apparent when considering the development of a specialized game like a Match Three title, those engines either do not allow for match three development (RPG Maker or Ren'Py) or deliver match three titles with too much technological bloat (Unity/Unreal/Godot)

While existing game engines offer a lot of tools and capabilities, there exists a space for specialized engines made for specific genres. Recognizing the limitations of mainstream engines explains development of our multiplatform Match Three game engine, which tries to fix these gaps while providing an easy development.

Chapter 2

Basics of Game Engine Design

2.1 Core Components of Game Engines

Game engines serve as the backbone for game development, providing a selection of tools and features that simplify the process of creating a game from scratch. At the heart of every game engine lie its core components, which define its capabilities, flexibility, and performance. This section explains those elements that we are going to implement in our engine.

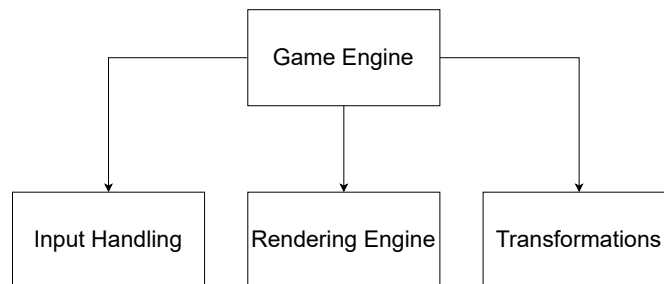


Figure 4: Components used in our game engine

2.1.1 Rendering Engine

The rendering engine is responsible for all visual aspects within a game. It takes the game's data – including textures and shaders – and converts them into pixels on the screen.

- **Role:** Ensures that all visual elements, from static backgrounds to animations, are displayed with clarity, smoothness, and consistency.
- **Key Features:** Supports various rendering techniques such as ray tracing, rasterization, and shadow mapping to enhance visual quality.

2.1.2 Input Handling

Games are interactive by nature, and this interaction is delivered by the engine's input handling system.

- **Role:** Processes user inputs from various sources, like a keyboard, mouse, gamepad, or touch screen. Translates these inputs into in-game actions or commands.
- **Key Features:** Detects multiple simultaneous key presses, supports touch gestures, and allows for input remapping.

2.1.3 Transformations

In order for object in game to move, game engine must handle its position on screen, this is a job of transformation system

- Role: Move objects based on user input, delete, add and modify existing coordinates
- Key Features: Matrices and vectors calculation

Each component of game engine plays a different role, in combination they ensure that the game runs smoothly, offers an enjoyable experience, and responds to user interactions. These core parts form the foundation upon which our multiplatform Match Three game engine will be built and updated.

2.2 The Role of OpenGL in Game Development

Game engines need graphics libraries, those libraries provide the necessary tools to bring visual elements to the screen. Among these libraries, OpenGL (Open Graphics Library) has established itself as one of the most popular choices for graphics rendering. This section explores the role of OpenGL in game development, focusing on library functionalities and how they can be used.

2.2.1 Introduction to OpenGL

OpenGL is a cross-language, cross-platform application programming interface (API) designed for rendering vector graphics. Initially developed by Silicon Graphics in the 1990s, it is now taken care by the non-profit technology organization, the Khronos Group. It is platform-agnostic which makes experience consistent across different platforms.

2.2.2 Rendering Mechanism

At its core, OpenGL is about rendering. It provides developers with the tools needed to draw complex 3D and 2D graphics.

- Role: OpenGL communicates directly with a system's GPU (Graphics Processing Unit) and translates the game's data into visual elements displayed on screen.
- Key Features: Supports techniques such as texture mapping, anti-aliasing, and shader programming.

2.2.3 Shader Programming

Shaders allow for creating stunning visuals programmatically, and OpenGL provides extensive support for shader programming.

- Role: Shaders allow developers to program the GPU directly, which gives massive control over how individual pixels or vertices are processed.
- Key Features: OpenGL's GLSL (OpenGL Shading Language) lets developers create custom shaders, enabling dynamic lighting, shadows, and other advanced visual effects.

2.2.4 Extensibility

One of OpenGL's strengths lies in its extensible design, allowing to include advancements in hardware and software.

- Role: As hardware evolves, new extensions can be added to OpenGL without changing the entire API. This makes OpenGL relevant, independent of technological improvements.
- Key Features: These extensions can be used to utilize latest graphics hardware capabilities, making games look and perform their best.

2.2.5 Cross-Platform Development

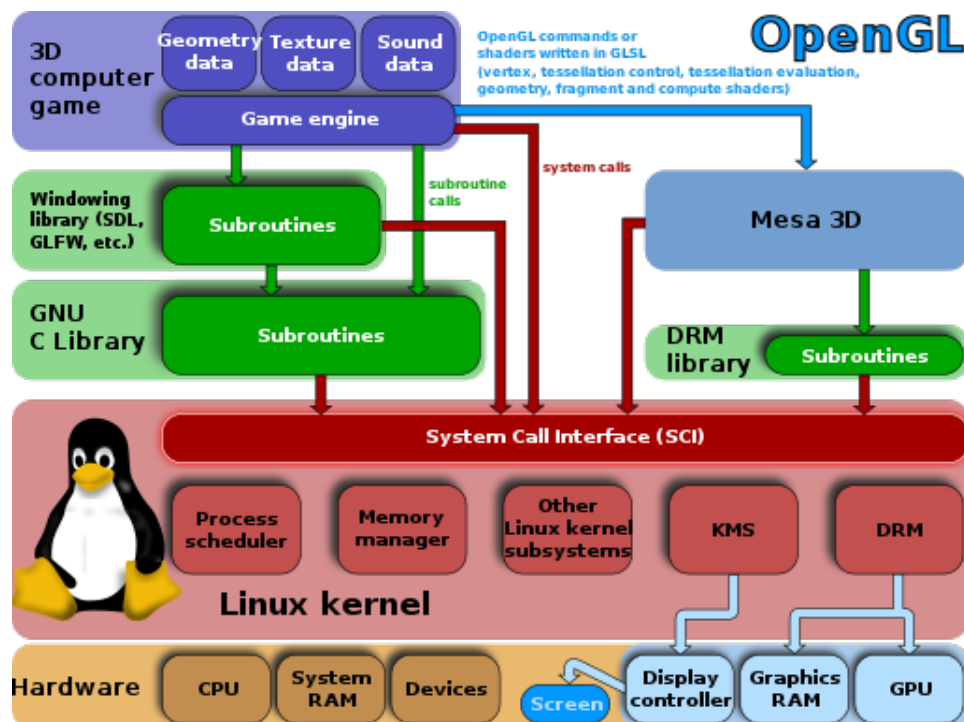


Figure 5: Connection between Linux and OpenGL-based games [Linux kernel and OpenGL video games \(2014\)](#)

Gamers are spread across Windows, Mac, GNU/Linux, and other platforms, OpenGL's platform-independent allows for creating games for all platforms.

- **Role:** OpenGL ensures that games look consistent across different platforms, easing the process of multiplatform game development.
- **Key Features:** It abstracts underlying platform-specific differences, allowing developers to focus on creating the game without need to adapt to platform-specific constraints.

2.2.6 Integration with Other Libraries and Tools

OpenGL is compatible with many libraries and development tools.

- **Role:** Enhances OpenGL capabilities by working with libraries like GLFW for window management, GLAD for handling extensions, and more.
- **Key Features:** Provides an unified development environment, where various tools and libraries work together under OpenGL, again simplifying the process of game development.

In summary, OpenGL combines software and hardware, enabling developers to create experiences with enough precision and speed. Understanding impact of OpenGL will make explaining our own game engine inner workings much easier.

2.3 Multiplatform Considerations

In order to reach as wide an audience as possible, developers try to develop a game for multiple platforms – mostly Windows, Mac, and GNU/Linux. Creating multiple platforms game engine is filled with challenges and considerations. This section explores difficulties of multiplatform game development, describing what developers must be conscious of when targeting multiple operating systems.

2.3.1 Platform-Specific Hardware and Software

Every platform has its unique hardware and software configurations, which influence a game's performance and appearance.

- Role: Ensuring the game runs smoothly across different hardware setups, from different GPU architectures to memory configurations.
- Key Features: Developers need tools to abstract these hardware-software differences, allowing the game engine to react dynamically based on the platform.

2.3.2 User Input Variations

Different platforms often have different input methods, from touchscreens to gamepads to keyboard-mouse setups.

- Role: Making sure that the game responds accurately to various input methods.
- Key Features: Input handling system that can detect and adapt to different input devices.

2.3.3 Graphical Rendering Nuances

There are platform-specific nuances in how graphics are presented.

- Role: Making engine work under different display resolutions, aspect ratios, and screen sizes.
- Key Features: Dynamic resolution scaling and responsive UI layouts to ensure the game looks similar across platforms.

2.3.4 File System and Data Management

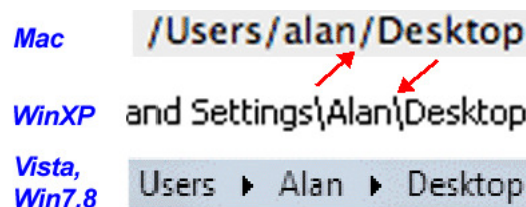


Figure 6: Separating folders with backslash and forward slash is one of many differences between Windows and Linux systems [PCMag encyclopedia backslash](#) (2013)

File systems vary between Windows, Mac, and GNU/Linux, influencing how game data is stored, retrieved, and updated.

- Role: Managing game saves, configurations, and other data across platforms.
- Key Features: Data management system that behaves the same for platform-specific directory structures and access permissions.

Chapter 3

Toolchain and Libraries Overview

3.1 Introduction to GLFW

In the arena of game development, libraries related to window management and input handling play a crucial role. GLFW offers an interface for developers to create interactive applications. This section offers an introduction to GLFW, explaining its origins, functionalities, and relevance in game development.

3.1.1 Historical Context

GLFW, which stands for Graphics Library Framework, was a response to the need for an open-source library that simplifies the challenges of window management and input handling, especially for OpenGL applications. Initially developed to provide a more straightforward alternative to existing solutions, GLFW has grown in popularity due to its simplicity, efficiency, and cross-platform capabilities.

3.1.2 GLFW's Core Competencies

GLFW serves as a bridge between the game's logic and the operating system, handling tasks that are crucial for interactive applications.

- **Window Management:** GLFW manages window creation, querying, and manipulation. It allows developers to focus on their game logic rather than handling platform-specific windowing.
- **Input Handling:** GLFW provides a system to capture and process user inputs. It forms for keyboard strokes, mouse movements, or even joystick inputs, GLFW makes sure that user interactions are detected and communicated to the application efficiently.
- **Context Management:** Especially relevant for OpenGL applications, GLFW helps create contexts, manage profiles, and handle extensions, for easier development experience.

3.1.3 Integration with OpenGL

While GLFW is platform-agnostic and can support multiple rendering systems, it is mostly used with OpenGL.

- **Role:** It functions as the companion to OpenGL, offering an environment where OpenGL can work more efficiently by addressing system-specific challenges that are outside OpenGL's specification.
- **Key Features:** Context creation for OpenGL.

3.1.4 Cross-Platform Capabilities

GLFW's is able to work across Windows, Mac, and GNU/Linux.

- Role: GLFW is a consistent interface for developers, regardless of the target platform. This ensures uniformity in window creation, input handling, and other functionalities across different operating systems.
- Key Features: An architecture that ensures platform-specific norms while offering a consistent API for developers.

3.1.5 Community and Support

Being open-source, GLFW benefits from a huge community that continually refines, optimizes, and extends the library.

- Role: Ensures GLFW remains up-to-date, addresses challenges and evolves with the needs of developers.
- Key Features: Tutorials and forums, assisting both new and trained developers in using GLFW

GLFW proves useful to many game developers, especially those using OpenGL. Its ability to handle window management, input processing, and other tasks ensures that developers can focus on crafting game mechanics and narratives.

3.2 Role of GLAD in OpenGL Loading

GLAD is a robust and flexible loader-generator designed for OpenGL. It ensures the dynamic loading of OpenGL functions based on the specific version and extensions a developer chooses. GLAD was developed to automate handling OpenGL extensions.

3.2.1 The Need for Loading OpenGL Extensions

OpenGL's is an extensible API. With changes in hardware, new functionalities are introduced through extensions without changing the core API. This creates new challenges on their own.

- Role: Extensions allow developers to use the latest graphics capabilities, which are not always part of the core OpenGL library.
- Key Features: Handling these extensions requires dynamic loading at runtime, so that the game can access and utilize these advanced functionalities.

3.2.2 GLAD's Functionality in Extension Loading

GLAD simplifies working with OpenGL extensions through its automated approach.

- Role: Provided with the OpenGL version and extensions, GLAD generates C/C++ code to load these extensions dynamically at runtime. This removes the need to manually address each extension, reducing errors and inefficiencies.
- Key Features: GLAD supports multiple languages and specifications. It is also up-to-date with the latest OpenGL specifications, making it compatible with newest graphics capabilities.

Glad

Multi-Language GL/GLES/EGL/GLX/WGL Loader-Generator based on the official specs.

Language
C/C++

Specification
OpenGL

API

glVersion 4.6

gles1None

gles2None

glsc2None

Profile
Compatibility

Extensions
Search
GL_3DFX_multisample
GL_3DFX_tbuffer
GL_3DFX_texture_compression_FXT1
GL_AMD_blend_minmax_factor
GL_AMD_conservative_depth
GL_AMD_debug_output
GL_AMD_depth_clamp_separate
GL_AMD_draw_buffers_blend

↔

Search

ADD LISTADD ALLREMOVE ALL

Options

☒ Generate a loader
☐ Omit KHR (due to recent changes to the specification, this may not work anymore)
☐ Local Files

GENERATE

Figure 7: Glad website allowing for different configurations of OpenGL [Glad configuration site](#) (2023)

3.2.3 Integration with GLFW and OpenGL

GLAD interacts with both OpenGL and GLFW, creating an unified development environment. Once GLFW creates the OpenGL context, GLAD is utilized to load the necessary functions. This collaboration ensures that GLFW handles the platform-specific instructions, while GLAD focuses on OpenGL's extensibility.

3.3 Handling text and images with FreeType, ft2build and stb_image

3.3.1 Introduction to FreeType

FreeType is an open-source software font engine, It offers ability to render text onto bitmaps and other font-related functionalities.

3.3.2 Key Capabilities of FreeType

- Font Parsing and Loading: Before rendering, fonts need to be parsed and loaded. FreeType supports many font formats, ensuring broad compatibility.
- Glyph Rendering: FreeType is good at converting individual glyphs into bitmaps, handling anti-aliasing and hinting to create clear, sharp text rendering.
- Font Metrics and Kerning: Rendering text isn't just about individual glyphs. The spacing between them (kerning) and their metrics are crucial for readability and aesthetics, and FreeType is able to deliver this functionality.

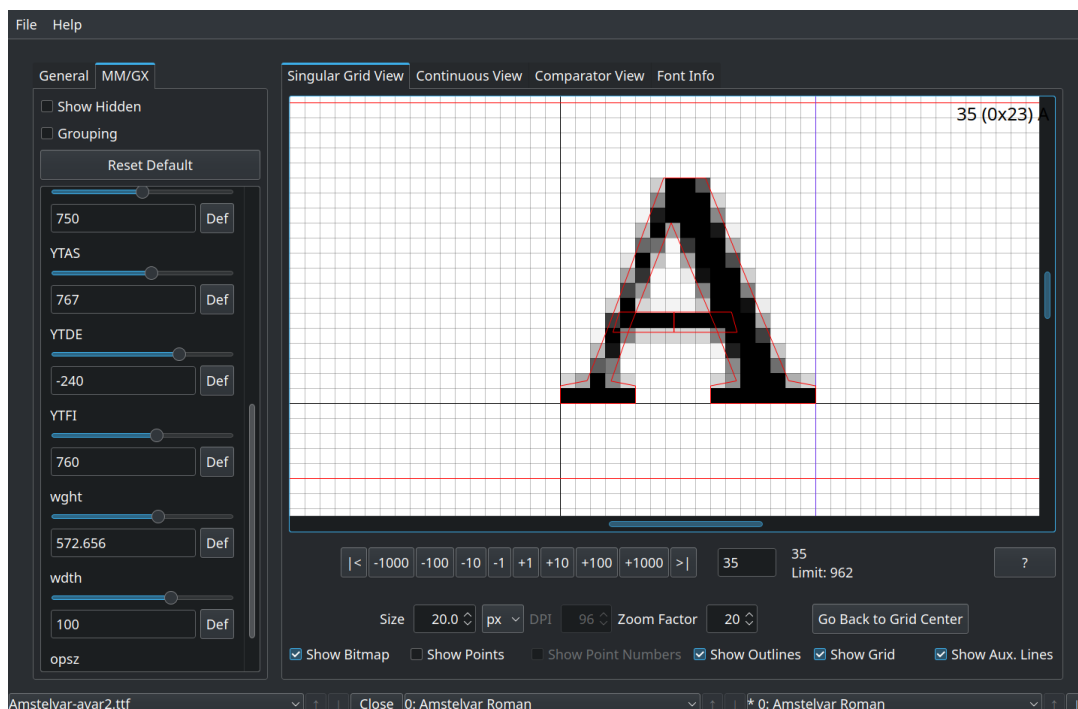


Figure 8: ftinspect showcasing capabilities of FreeType library [Free Type site](#) (2023)

3.3.3 Understanding ft2build

ft2build serves as an essential interface, allowing for the inclusion and deployment of FreeType headers in development projects. ft2build was created to ease the integration process of FreeType 2.

3.3.4 Introduction to stb_image

While FreeType takes charge of fonts, stb_image is a solution for image loading and decoding, popular for its simplicity and compactness. It is part of the stb collection of single-file libraries.

3.3.5 Key Capabilities of stb_image

- Image Loading: It is capable of handling popular formats like JPEG, PNG, BMP, and more, this way developers can incorporate multiple types of graphical assets into their game
- Image Decoding: stb_image decodes images into a format that can be directly used with OpenGL or other rendering systems.
- Simplicity and Efficiency: stb_image has minimalistic design. With just a single header file, developers can use its capabilities without employing complex dependencies or configurations.

Specialized tools like FreeType and stb_image elevate the visual experience of games with text rendering and images inclusion into game engine

Chapter 4

Design and Architecture of the Match Three Engine

4.1 Game Loop and State Management

In the heart of every video game lies the game loop. This cycle dictates the game's pacing, ensuring that events, updates, and rendering processes all appear in a correct order. Game loop functions together with state management, which ensures the game behaves consistently in response to player actions and internal events. This section will focus on both the game loop and state management and how they influence our multiplatform Match Three game engine.

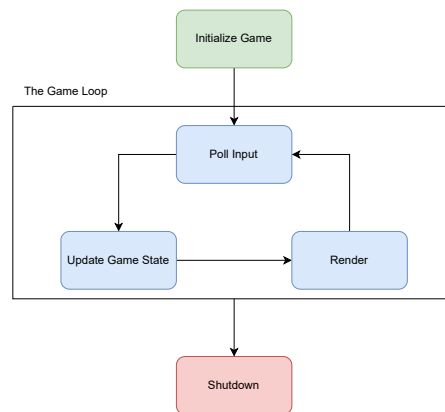


Figure 9: Game loop flowchart representation

4.1.1 The Anatomy of a Game Loop

The game loop is a repetitive process that continues for the lifespan of the game session. It tries to:

- Poll Input: Gather and process player inputs.
- Update Game State: Based on the inputs and game rules, modify game entities and overall state.
- Render: Draw the current state of the game onto the screen.
- Sleep (Optional): Introduce a short delay, if necessary, to control the loop's frequency and match the target framerate.

4.1.2 Importance of State Management

As games grow complex, they may contain numerous states, including:

- Menu Screens: Where players can start, configure settings, or view credits.

- **Gameplay Modes:** Different modes or levels of gameplay, each with its unique rules and environments.
- **Pause/Resume States:** Allowing players to stop the game and later resume it.
- **Endgame Scenarios:** Winning, losing, or drawing conditions.

State management ensures the game recognizes and responds appropriately to these states.

4.1.3 Interlinking Game Loop and State Management

Game Loop and State Management must work together:

- **State-Driven Rendering:** What the game displays during each loop iteration depends on its current state. For instance, the game may render a main menu, gameplay screen, or a 'game over' message based on its active state.
- **State Transitions:** Player inputs or internal events can trigger state changes. The game loop continually checks for such triggers and initiates the necessary transitions.

4.1.4 Challenges and Considerations

Managing a game loop and states is filled with challenges:

- **Performance:** The game loop must run efficiently to maintain smooth gameplay, especially crucial for real-time games where fast decisions matter.
- **State Persistence:** Some states may require data persistence, like saving a game, which requires storing and retrieving game data.
- **Synchronization:** Ensuring that state transitions and game loop iterations are in sync is essential to avoid unexpected behaviors.

4.1.5 Role in the Multiplatform Match Three Game Engine

For our game engine, the game loop and state management fulfill following functions:

- **Cross-Platform Consistency:** The game's behavior, pacing, and responses should be consistent across Windows, Mac, and GNU/Linux platforms.
- **Match-Three Logic:** The game engine will manage states specific to match-three mechanics, like checking for matches, handling cascades, and introducing new game pieces.

In summary, game loop maintains the rhythm of the game, while state management ensures logic in the game's behavior.

4.2 Asset Management and Rendering

Game development consists of art and logic, where visual assets behave based on game code. Ensuring these assets are well-organized, efficiently loaded, and rendered is vital for any gaming experience. This section explores asset management and rendering.

4.2.1 What are Game Assets?

In the context of game development, assets refer to:

- **Graphics:** These include textures, sprites, animations, and UI elements.
- **Data:** Configurations, level designs, scripts, and other data structures that define gameplay mechanics.



Figure 10: Exemplary assets for match three tiles [Match three assets \(2017\)](#)

4.2.2 Rendering: Bringing Assets to Life

Once assets are organized and loaded, rendering puts them onto the screen:

- **Displaying Graphics:** Textures and sprites are mapped onto game entities.
- **Shaders & Effects:** Advanced graphical effects, such as lighting, shadows, and particle effects, are layered on to assets to elevate visuals.

4.2.3 Cross-Platform Considerations in Asset Management

Given our engine's multiplatform nature, certain challenges arise:

- **Format Compatibility:** Not all asset formats are supported uniformly across platforms. Ensuring assets are in universally compatible formats and using libraries like `stb_image` that unify asset formats among different operating systems is crucial.
- **Optimization:** Different platforms may have varying memory capacities and processing power. Assets might need to be optimized, resized, or compressed to respond to those limitations.

4.2.4 Role in the Multiplatform Match Three Game Engine

Our game engine's being multiplatform complicates assets rendering:

- **Uniformity Across Platforms:** Visual experience should remain consistent across Windows, Mac, and GNU/Linux.
- **Match-Three Specifics:** Given the genre, the engine must effectively manage assets like gems, power-ups, grid structures, and cascade animations and then render them in a visually pleasing way.

4.3 Event Handling and User Input

Main trait of video game is interactivity. Game must have the ability to capture, interpret, and respond to user input, whether it is the click of a mouse, the press of a keyboard key, or the swipe on a touchscreen. Managing those interactions is called event handling. In this section we will describe event handling and user input, and their role in our game engine.

4.3.1 Understanding User Input and Events

Every action by the user generates an event. This event contains specific data, such as:

- Type of Input: Whether it's a key press, mouse movement, click, or touch.
- Coordinates: In case of mouse or touch, the precise location of the interaction.
- Duration: The length of time an input is maintained, like a long press.

4.3.2 Event Polling vs. Event Callbacks

There are two primary ways to handle these events:

- Event Polling: The game loop consistently checks (or "polls") for user inputs at fixed intervals, processing any detected events.
- Event Callbacks: The system is set up to notify (or "call back") the game engine immediately when an event occurs.

4.3.3 Role in the Multiplatform Match Three Game Engine

Our game engine should provide following features connected with user input:

- Intuitive Matching: Selecting and swaping game pieces should be easy and convenient, with the engine accurately capturing and responding to these interactions.
- Feedback Loop: Upon receiving an input, the game engine must also provide immediate feedback, such as highlighting a selected piece, initiating a combo move or playing animation.

Event handling and user input are ways for players to communicate with the game. It is crucial to make them intuitive and smooth.

Chapter 5

Cross-Platform Development Challenges and Solutions

5.1 Operating System Variabilities

Operating systems (OS) are a middle ground between human interaction and hardware functionality. With Windows, Mac, and GNU/Linux operating systems as the most popular desktop environments, a multiplatform game engine must work under each of them. This section describes differences between these operating systems and how our engine can handle them.

5.1.1 Core Architectural Differences

Some of the distinctive features of OS architecture are:

- **Kernel Design:** Windows uses a hybrid kernel, while Linux is a monolithic kernel, MacOS is based around Unix XNU kernel.
- **File Systems:** Windows primarily offers NTFS, MacOS stores files using APFS, and GNU/Linux has multiple options like ext4, Btrfs, and many, many more.

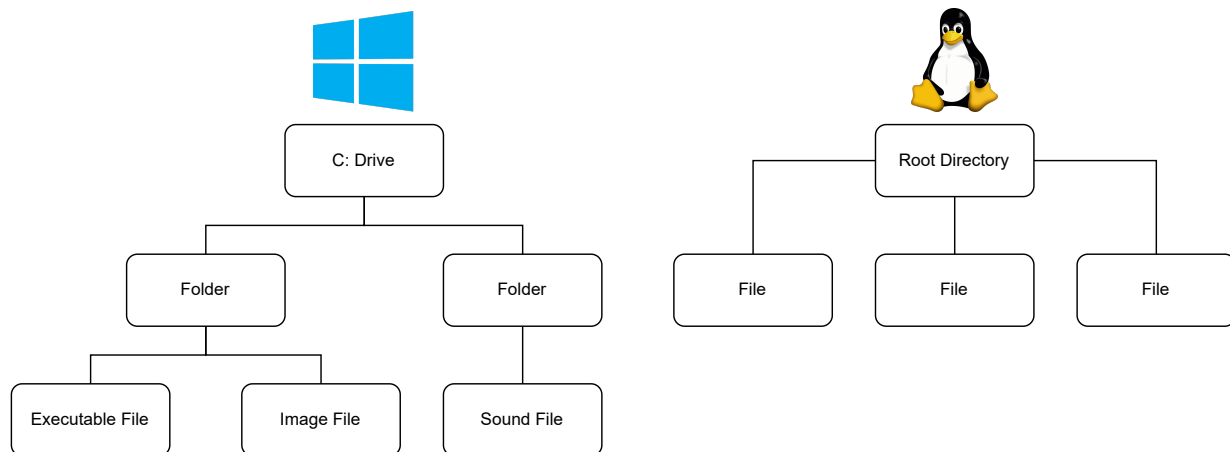


Figure 11: Windows file system consists of drives, which contain folders which contain subfolders and files, in Linux everything is a file

- **APIs & System Calls:** Each OS provides developers with its own set of APIs and system calls, changing how software communicates with the system.

5.1.2 Graphical User Interface (GUI) Divergence

Differences in GUI of each OS:

- **Window Management:** Windows uses a floating window system, MacOS has its unique mission control, and Linux distributions can use anything from floating, fixed to no graphical interface at all.

- **Input Methodologies:** Multi-touch gestures work different under MacOS than under Windows. Linux, depending on its distribution and desktop environment, may have a the same or different approach altogether.

5.1.3 Driver Support

Certain hardware, sometimes crucial to game engine like graphics card or keyboard, might be optimized for one OS over the others, or might not be supported entirely.

5.1.4 Middleware & Third-party Libraries

The accessibility, performance, and support of middleware and third-party libraries can differ, while some libraries are universally compatible, others might be OS-specific, another problem might be with the update cycle, the frequency of libraries updates can vary, impacting game engine compatibility and performance.

5.1.5 Implications for the Multiplatform Match Three Game Engine

Operating system variances impact directly on our game engine:

- **Performance Optimizations:** Depending on the OS, certain code paths might be more efficient. The engine must be adaptable.
- **Consistent Gameplay Experience:** Players on all platforms should have a similar gameplay experience.
- **Updates & Maintenance:** As OSs changes, the game engine must be ready for regular updates to ensure compatibility.

Multitude of operating systems poses challenges for game engine developers. By understanding these differences, developers can make sure that the game functions across different platforms.

5.2 Addressing Hardware and Driver Differences

Hardware and its associated drivers act as the final execution devices for any game engine. When creating a game engine, developer must keep in mind a variety of hardware configurations and their driver implementations. This section focuses on understanding these differences and how we can challenge them.

5.2.1 A Landscape of Diverse Hardware

Across all platforms, multiple hardware components exists:

- **Processors:** Mostly Intels and AMD's x86 architecture (32 and 64 bits), together with ever more popular ARM processors
- **Graphics Cards:** NVIDIA, AMD, and Intel dominate the market, each with their architectures and feature sets.
- **Memory Configurations:** RAM specifications, speeds, and type of storage devices (SSDs and HDDs) can influence performance.
- **Input Devices:** Various keyboards, mice, touchpads, touchscreens, and game controllers, each have unique specs and capabilities.

5.2.2 Drivers: The Link to Hardware

Drivers allow an operating system to interact with hardware:

- **Proprietary vs. Open-Source:** While NVIDIA provides proprietary drivers, AMD and Intel often offer open-source alternatives for Linux. There are also community driven, universal, open-source drivers.
- **Update Frequencies:** Hardware vendors release driver updates at different intervals, each update possibly affects game performance and compatibility.
- **Feature Support:** New hardware features may be enabled in driver updates.

5.2.3 Challenges Posed to Game Engines

- **Optimization Issues:** What's optimized for one graphics card might not be for another.
- **Feature Inconsistencies:** Some hardware may support specific graphical features, while others might not.
- **Input Latency Variations:** Different input devices and their drivers can lead to varied response times and inconsistent responses.

5.2.4 Abstraction Layers in Match Three Game Engine

To address these challenges, our engine adopts abstraction layers: by utilizing existing libraries and solutions (OpenGL, GLAD and GLFW), our game engine is ready for easier adjustments for specific hardware.

Chapter 6

Implementation Details

6.1 Requirements

6.1.1 Functional Requirements (FR)

1. Game Initialization and Setup (FR1):

- The engine shall initialize a game board with a predetermined square size (e.g., 8x8).
- The game board shall be populated randomly with different colored pieces.
- The initial game board should not have any matches (three or more of the same colored pieces in a row or column).

2. User Interactivity (FR2):

- Players shall be able to select a piece.
- Players shall be able to swap the selected piece with an adjacent piece (up, down, left, or right) to form a match.

3. Matching Logic (FR3):

- When three or more pieces of the same color align vertically or horizontally, they shall be recognized as a match.
- Matched pieces shall be removed from the board.

4. Board Update (FR4):

- After pieces are matched and removed, new pieces shall drop down to fill the vacant spaces.
- The board shall continuously check for matches after each update.

5. Scoring System (FR5):

- Players shall earn points for each match made.
- Bonus points shall be awarded for matches greater than three pieces.

6. Game Over Logic (FR6):

- The game shall end when no moves are left.

7. Pause and Resume (FR7):

- Players shall be able to pause and resume the game.

8. Game Difficulty Levels (FR8):

- Make the game more difficult or easy by giving player more or less turns to play

6.1.2 Non-functional Requirements (NFR)

1. Performance (NFR1):

- The game engine shall ensure smooth animations without any lag.
- Match detection should occur within milliseconds to ensure real-time response.

2. Usability (NFR2):

- The game interface shall be intuitive.

3. Portability (NFR3):

- The game engine should be platform-independent, allowing for easy deployment across different operating systems using C++ and OpenGL.

4. Maintainability (NFR4):

- The code shall be well-structured, modular, and commented, allowing for easy updates and maintenance.
- Utilize design patterns where applicable for cleaner and more understandable code.

5. Reliability (NFR5):

- The engine shall be robust enough to handle unexpected inputs without crashing.
- Memory leaks should be minimized, and efficient memory management practices should be followed.

6. Scalability (NFR6):

- The game engine shall support additional features or levels in the future without requiring a complete overhaul.

7. Documentation (NFR7):

- A comprehensive documentation detailing the architecture, algorithms, and functionalities should be provided. (For example in this thesis)

6.2 Core Engine Implementation

For our Match Three game engine, we aimed to create a simple core, capable of ensuring basic gameplay and small codebase allowing developers to extend or modify the engine's capabilities as needed. This section describes the implementation of the Match Three game engine.

6.2.1 Foundational Principles

- Platform Independence: Core engine functionalities independent of any specific platform, allowing for easy porting across Windows, MacOS, and GNU/Linux.
- Simplicity: There are no more than 1500 lines of code in the entire engine
- Good coding style: When writing code, popular coding styles of cpplint and clang-tidy were used
- Good coding principles: when writing code, clang-tidy was extensively used to highlight potential errors and eliminate bad practices

6.3 Integration of Libraries and Tools

In this section we will explain rationality for choosing libraries and tools for our engine and how they were implemented

6.3.1 Choice of graphic rendering API

There are 3 main APIs for graphical rendering

- DirectX
- OpenGL
- Vulkan

DirectX developed by Microsoft focuses on Windows operating systems and Microsoft line of consoles Xbox, it is deemed as being harder with more low level programming and requiring better understanding of how underlying mechanisms work but in turn offers functionalities and better performance. It does not have free license.

OpenGL is developed by Khronos Group and offers good compatibility, especially if using OpenGL ES subset which works on Windows, Linux, Mac OS, Android, iOS and all major consoles. It is widely recognized as easiest of APIs and most popular choice for writing first game engine. On the other hand it lacks some of more advanced features which have to be written manually. It uses open source license similar to BSD

Vulkan is also developed by Khronos Group and as such is deemed as a spiritual successor of OpenGL with focus on using modern C++ features and fixing issues created by OpenGL 30 years old development time. Out of these three it is recognized as the hardest one as it is both complicated and newest. Similarly as OpenGL it uses open source license, namely Apache License 2.0.

Table 1: Comparison of Graphics APIs

	Developer	Platform Focus	Complexity	License
DirectX	Microsoft	Windows, Xbox	Harder, requires understanding of underlying mechanisms	Non-free
OpenGL	Khronos Group	Windows, Linux, Mac OS, Android, iOS, major consoles (especially with OpenGL ES)	Easiest, lacks some advanced features	Open source (similar to BSD)
Vulkan	Khronos Group	Modern platforms, successor to OpenGL	Hardest, newest	Open source (Apache License 2.0)

Considering all of those characteristics I decided to go with OpenGL API, specifically OpenGL ES subset with its focus on compatibility as making a multiplatform application is one of the focuses of this thesis. I decided that since this will be my first attempt at game engine development I need something that is relatively easy and has a lot of resources online. I would most likely not use advanced features of Vulkan and DirectX and therefore finish my thesis before approaching problems where OpenGL does not deliver more complicated architecture. From my own private preferences I also prefer software with open source license.

6.3.2 Choice of OpenGL Library

There are 4 basic OpenGL libraries that I considered:

- freeGLUT
- SDL
- SFML
- GLFW

freeGLUT was created as opensource alternative to GLUT, is considered to be the worst out of all 4, written in archaic way, using C or very old C++, which in turn results in unexpected "buggy" behaviour, it is also not really popular with lack of online guides

SDL - Simple DirectMedia Layer has big userbase, it is not designed to be used as a standalone library and requires additional libraries to do networking or to create more complex applications.

SFML is the library with most features out of all 4, it supports networking, audio and has system features by default. It uses modern object oriented C++. Main problem with SFML is that it is not very popular API, therefore troubleshooting problems with SFML is quite hard and it has only few use guides online

GLFW is an library that is both the most popular and with fewest features by default. It forces users to use additional libraries for networking, sound, physic calculations and so on but in turn is also quite small and flexible. It has biggest community and a lot of guides, like one hosted at learnopengl.com or one created by programming youtuber Cherno

I decided to use GLFW library. I wanted something that is relatively easy to troubleshoot and has abundance of learning materials online.

6.3.3 GLAD and OpenGL Extension Handling

GLAD enabled the dynamic loading of OpenGL functions, this way engine can use the latest OpenGL features.

6.3.4 Text and Image Rendering with FreeType, stb_image, and ft2build

FreeType and stb_image were chosen for their simplicity and popularity

- Font Rendering: With FreeType and ft2build, the engine can render text, crucial for messages, scores, and menus.
- Image Loading: stb_image simplifies the task of loading and processing various image formats, from PNGs and JPGs to more complex formats, making asset integration much easier.

6.3.5 Core Modules

The engine's architecture is segmented into modules:

- Game object: (game_object.cpp) Defines base class for game objects used by the engine, this class is later used to define tiles
- Main Game class: (game.cpp) This class combines all of the other modules and runs the game loop
- Game Level: (game_level.cpp) Loads and saves game levels.
- Particle Generator: (particle_generator.cpp) Generates particle effects for tiles

- Resource Manager: (resource_manager.cpp) Handles assets and shaders files
- Shader: (shader.cpp) calculates shaders and applies their effects
- Sprite Renderer: (sprite_renderer.cpp) Renders sprites from images
- Text Renderer: (text_renderer.cpp) Renders text
- Texture Renderer: (texture.cpp) Renders textures

6.3.6 Main function

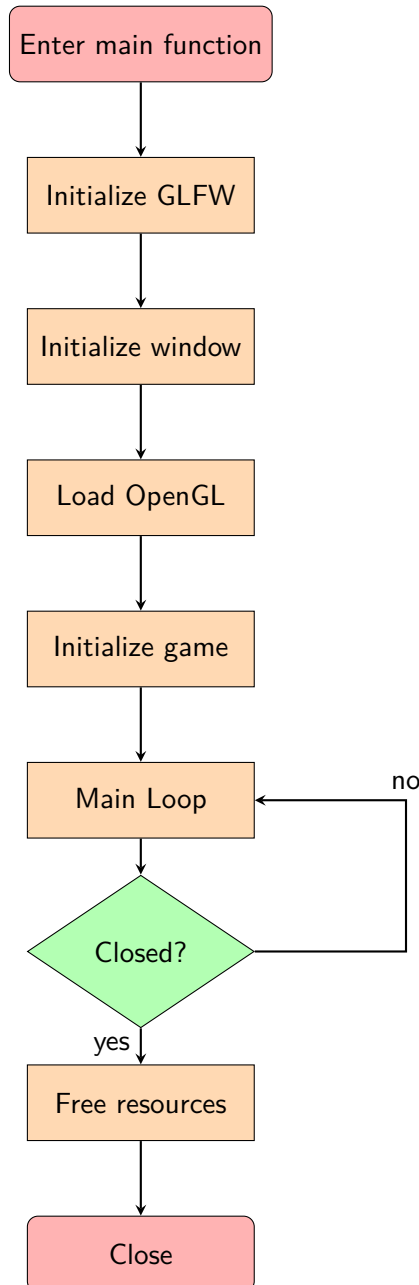


Figure 12: Main function logic

GLFW initialization Very first operation of our engine is to initiaze GLFW context, window and openGL, we set version of glfw to 3.3, resizable window and the profile depending on whether the engine got initialized on MacOS, or Linux/Windows, we set the windows width and height based on values from constants file

Main Loop

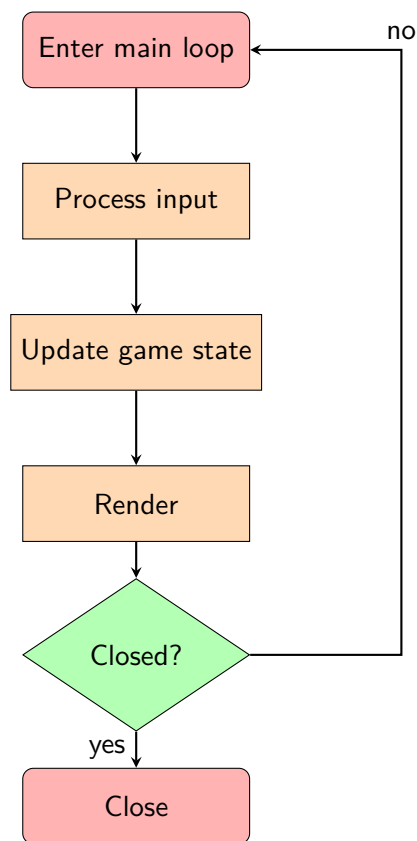


Figure 13: Main loop

Process inputs Reacts to user inputs and sets up game state based on them, for example which tile was chosen by user

Update game state Changes game level, removes blocks if applicable, drops new blocks and checks for win/lose conditions

Render Based on changes from previous steps draws actual game window.

6.3.7 Constants file

Game parameters, from graphics settings to gameplay variables, are stored in constants.cpp file, which can be modified, to apply the changes the engine must be compiled again.

Game Constants

constants namespace: Used for constants that are used over the project, for example colors, colors are hold in open gl vector of 3 values (for RGB colors)

```
1 namespace constants {
2     constexpr glm::vec3 LIGHT_BLUE = glm::vec3(0.2F, 0.6F, 1.0F);
3     ...
4 }
```


text namespace: Parameters concerning text display positions on screen.

```
1 struct Point {
2     float x;
3     float y;
4 };
5
6 namespace text {
7     constexpr Point LIVES_POSITION = {5.0F, 5.0F};
8     ...
9 }
```

textures namespace: Provides paths to texture files and their respective identifiers, in this examples gems are named as: "g" for gems, number of gem and then color of gem

```
1
2 struct TextureInfo {
3     const char* path;
4     const char* name;
5 };
6
7 namespace textures {
8     static constexpr TextureInfo textures[] = {
9         {"resources/textures/background.jpg", "background"},
10        {"resources/textures/g1black.png", "block_black"},
11        {"resources/textures/g2blue.png", "block_blue"},
12        {"resources/textures/g3green.png", "block_green"},
13        {"resources/textures/g4purple.png", "block_purple"}
14    };
15 }
```

6.4 Game class

Game class defined in game.h and game.cpp is the main and biggest class in the project, it brings together all other classes and functionalities of the engine.

Includes

```
1 #include "../constants.hpp"
2 #include "../filesystem.h"
3 #include "../game_level.h"
4 #include "../game_object.h"
5 #include "../resource_manager.h"
6 #include "../sprite_renderer.h"
7 #include "../text_renderer.h"
```

Constructor Constructor initializes game window size, empties clicked keys, sets game state to game menu, sets initial level map and retrieves max turns from constants file

```
1 Game::Game(ScreenDimensions screen)
2     : State(GAME_MENU),
3       Keys(),
4       KeysProcessed(),
5       Width(screen.width),
6       Height(screen.height),
7       Level(0),
8       MaxTurns(constants::MAX_TURNS),
9       Turns(0)
10 {}
```

6.4.1 Game initialization

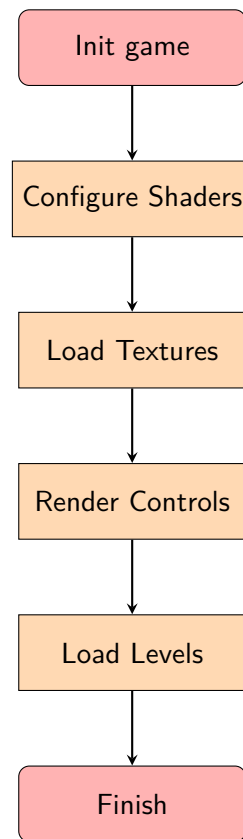


Figure 14: Initialization logic

We will describe each step below

Shaders Two shaders are used throughout the project, one for rendering sprites (gem tiles) and one for generating post processes (for example shaking the map when tiles get matched)
More on the shaders usage later when we will be discussing postProcessor and Shader classes

```
1 void Game::loadShaders() {  
2     ResourceManager::LoadShader("sprite.vs", "sprite.fs", "sprite");  
3     ResourceManager::LoadShader("post_processing.vs", "post_processing.fs",  
4         "postprocessing");  
5 }
```

Textures Textures names are first defined in the constants file, then the game object invokes resource manager to go through texture paths and load them

```
1 void Game::loadTextures() {  
2     const auto *begin = std::begin(textures::textures);  
3     const auto *end = std::end(textures::textures);  
4  
5     for (auto it = begin; it != end; ++it) {  
6         ResourceManager::LoadTexture(FileSystem::getPath(it->path).c_str(),  
7             it->name);  
8     }  
9 }
```

Setting controls After loading sprite renderers, postprocesses and text, we set instances of classes using loaded resources.

```
1 void Game::renderSpecificControls() {
2     // set render-specific controls
3     Renderer =
4         std::make_unique<SpriteRenderer>(ResourceManager::GetShader("sprite"));
5     Effects = std::make_unique<PostProcessor>(
6         ResourceManager::GetShader("postprocessing"), this->Width, this->Height);
7     Text =
8         std::make_unique<TextRenderer>(TextRenderer(this->Width, this->Height));
9     Text->Load(FileSystem::getPath("resources/fonts/OCRAEXT.TTF"),
10        text::OCRAEXT_FONT_SIZE);
11 }
```

Levels Levels are loaded automatically from the levels folder, game assumes that every file with extension ".lvl" is a level file and adds it to an array of levels

```
1 void Game::loadLevels() {
2     const std::string path = "../resources/levels/";
3     for (const auto& entry : std::filesystem::directory_iterator(path)) {
4         if (entry.is_regular_file() && entry.path().extension() == ".lvl") {
5             GameLevel level;
6             level.Load(entry.path().c_str(), this->Width, this->Height);
7             this->Levels.push_back(level);
8         }
9     }
10
11     this->Level = 0;
12 }
```

6.4.2 Game update

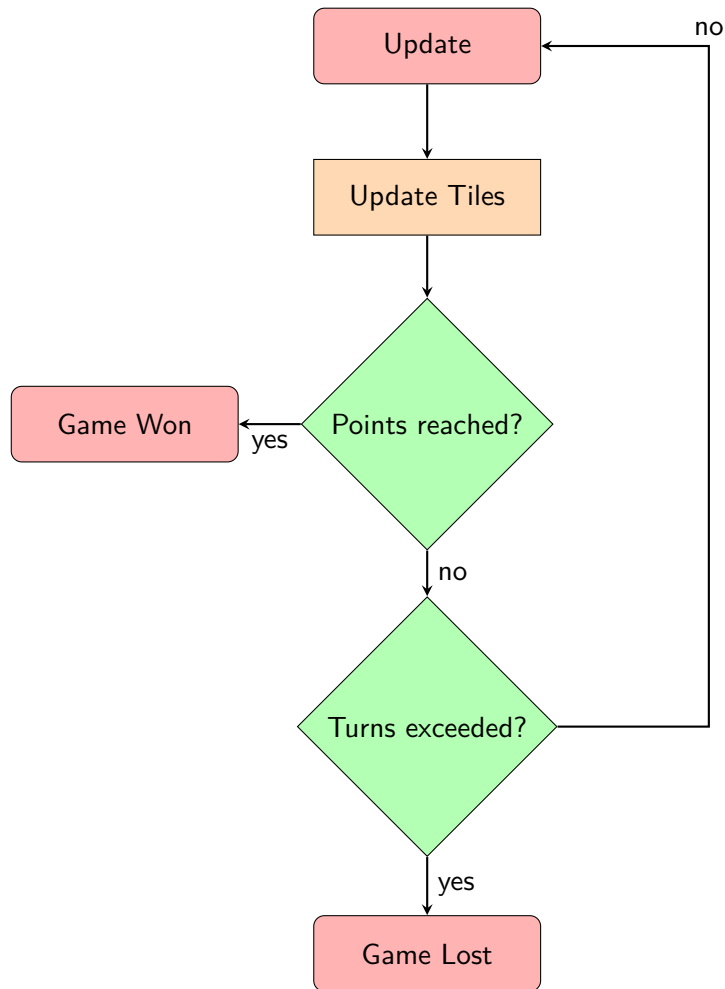


Figure 15: Game updates logic

Game is ended either when the player exceeds maximum move limit (game lost) or when points limit is reached (game won), otherwise the game continues, notice how first we check for win condition then for loss condition, this makes the gameplay a little bit more satisfying and fair.

6.4.3 Controls

Game allows to move through the grid to highlight a tile, then after clicking "enter" a tile is selected, then by using arrows or WSAD keys user can highlight the tile that will be swapped when again "enter" gets clicked

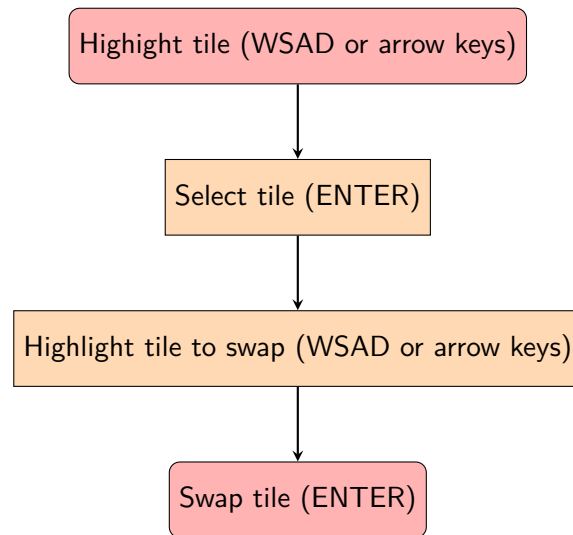


Figure 16: User controls

All of input processing is taken care by game class, and then relied to game level class.

```

1 void Game::handleActiveGame() {
2     if (this->State == GAME_ACTIVE) {
3         this->moveLeft();
4         this->moveRight();
5         this->moveUp();
6         this->moveDown();
7         this->selectPiece();
8         this->unselectPiece();
9     }
10 }

```



(a) View of highlighted tile



(b) View of selected tile

Figure 17: Tile selection views



(a) Left tile is set to be swapped when enter key will be hit



(b) Right tile is set to be swapped when enter key will be hit



(c) Upper tile is set to be swapped when enter key will be hit



(d) Down tile is set to be swapped when enter key will be hit

Figure 18: Tile swap controls

6.4.4 Rendering

As mentioned in main loop, the game runs a render method every frame.

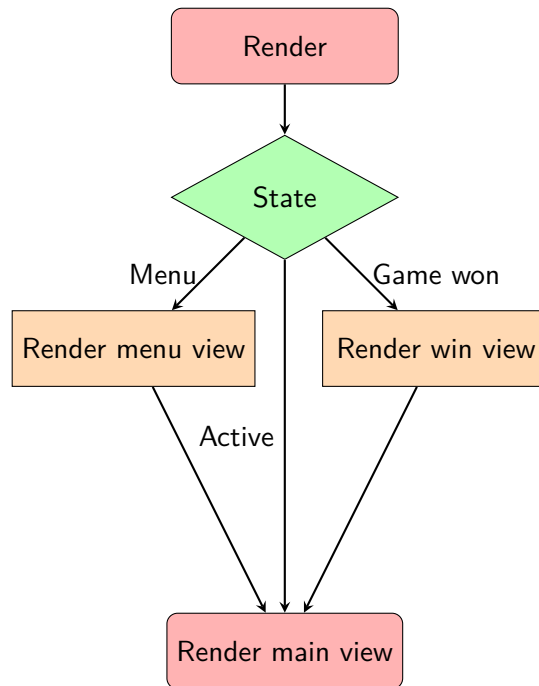


Figure 19: Rendering logic

Rendering text Rendering menu and win view consists of giving a text class correct coordinates and what text should be displayed

```

1 From constants.hpp file:
2 namespace text {
3     ...
4     constexpr float WON_X_POSITION = 320.0F;
5     constexpr float WON_Y_OFFSET = -20.0F;
6     constexpr float WON_ESCAPE_X_POSITION = 130.0F;
7     ...
8 }
9
10 From game.cpp file:
11 void Game::renderWin() const {
12     if (this->State == GAME_WIN) {
13         const float screenMiddleYCoordinate =
14             static_cast<float>(this->Height) / 2.0F;
15         Text->RenderText("You WON!", {text::WON_X_POSITION,
16             screenMiddleYCoordinate + text::WON_Y_OFFSET}, 1.0F,
17             glm::vec3(0.0F, 1.0F, 0.0F));
18         Text->RenderText("Press ENTER to retry or ESC to quit",
19             {text::WON_ESCAPE_X_POSITION, screenMiddleYCoordinate}, 1.0F,
20             glm::vec3(1.0F, 1.0F, 0.0F));
21     }
22 }
  
```

```

4 2 3 2 4 3 3 1 3 3
1 1 2 1 1 2 1 3 1 1
4 1 4 3 3 4 4 3 2 3
4 3 3 4 3 3 2 1 2 1
3 2 3 1 4 4 3 2 4 2
4 2 1 3 3 2 2 3 1 1
4 3 2 1 2 2 3 3 4 4
1 4 3 2 1 4 3 2 1 4
3 4 3 3 4 4 1 2 3 1
2 3 2 2 1 2 3 3 4 2

```

Figure 20: Exemplary level file

Rendering main view Rendering main view consists of rendering map and then applying postprocesses and text representing number of turns

```

1 void Game::renderMain() {
2     if (this->State == GAME_ACTIVE || this->State == GAME_MENU ||
3         this->State == GAME_WIN) {
4         // begin rendering to postprocessing framebuffer
5         Effects->BeginRender();
6         this->renderDraw();
7         // end rendering to postprocessing framebuffer
8         Effects->EndRender();
9         // render postprocessing quad
10        const auto time = static_cast<float>(glfwGetTime());
11        Effects->Render(time);
12        // render turns (don't include in postprocessing)
13        this->renderTurns();
14    }
15 }

```

6.5 Game level class

Game level class takes a 2D array of game objects instances (tiles), receives inputs from game class and reacts to them, it handles removing, swapping, adding new tiles on a map and checking for sequences of tiles. It also loads the level file and translates it to in game object

Reading file Game level class goes through each line of level file.

It loads each row to a vector of unsigned int as the level file contains rows of numbers representing tile type

Level initialization Initializing level takes the loaded vector of ids and creates game objects based on the window dimensions (so that the tiles fill the window evenly), textures are based on the tile id, we assume that all levels are square so the width and height of the window is always the same.

1

```

1 void GameLevel::handleTile(std::vector<std::vector<unsigned int>> tileData,
2                             unsigned int x_coordinate, unsigned int
3                             y_coordinate,
4                             float unit_width, float unit_height) {
5     const unsigned int tileNumber = tileData[y_coordinate][x_coordinate];
6     glm::vec3 color = this->assignColor(static_cast<int>(tileNumber));
7
8     // Directly use the constructor arguments for GameObject with emplace_back
9     this->Bricks.emplace_back(tileNumber,
10                               glm::vec2(unit_width * static_cast<float>(x_coordinate), unit_height *
11                                           static_cast<float>(y_coordinate)),
12                               glm::vec2(unit_width, unit_height),
13                               ResourceManager::GetTexture(idTextureMap[tileNumber]),
14                               color
15     );
16 }
17
18 void GameLevel::init(std::vector<std::vector<unsigned int>> tileData, unsigned int
19                     levelWidth, unsigned int levelHeight)
20 {
21     // calculate dimensions
22     unsigned int height = tileData.size();
23     unsigned int width = tileData[0].size(); // note we can index vector at [0]
24     since this function is only called if height > 0
25     float unit_width = static_cast<float>(levelWidth) / static_cast<float>(width);
26     // initialize level tiles based on tileData
27     for (unsigned int y_coordinate = 0; y_coordinate < height; ++y_coordinate)
28     {
29         for (unsigned int x_coordinate = 0; x_coordinate < width; ++x_coordinate)
30         {
31             this->handleTile(tileData, x_coordinate, y_coordinate, unit_width,
32                             unit_height);
33         }
34     }
35 }

```

2

6.5.1 Highlighting, selecting and swapping tiles

There are four steps in order for the tile to be swapped, logic of fulfilling this conditions is handled in game level class

Highlighting tile First the tile must be highlighted, game level makes sure that only one tile is highlighted by keeping the index of highlighted tile and changing it whenever user inputs a change

Selecting tile Then the tile must be selected, again game level has a special parameter just to keep track of which tile is selected

```

1 void GameLevel::selectPiece() {
2     this->Bricks.at(selectedTile).Selected = true;
3     ...
4 }
5
6 void GameLevel::unselectPiece() {
7     this->Bricks.at(selectedTile).Selected = false;
8     this->unswapAll();
9 }

```

Which tile will be swapped Then the user must choose which tile will be swapped, game level does not keep special parameter for that, it just automatically deselects all tiles in the selected tile neighborhood if user choose any other tile

```
1 void GameLevel::moveRight()
2 {
3     if(!this->Bricks.at(selectedTile).Selected) {
4         this->Bricks.at(selectedTile).Highlighted = false;
5         this->selectedTile++;
6         if(static_cast<size_t>(this->selectedTile) > this->Bricks.size()) {
7             this->selectedTile--;
8         }
9         this->Bricks.at(selectedTile).Highlighted = true;
10        return;
11    }
12    this->swapRight();
13 }
```

Confirming swap At the end user must confirm the swap, then the swap is checked if it creates the pattern of at least 3 tiles

```
1 void GameLevel::confirmSwap()
2 {
3     const bool canSwap = this->checkSwap();
4     if(this->swappableTile != -1 && canSwap) {
5         this->Bricks.at(selectedTile).Selected = false;
6         this->Bricks.at(selectedTile).Highlighted = false;
7         this->unswapAll();
8         const GameObject temp = this->Bricks[this->swappableTile];
9         this->Bricks[this->swappableTile].Position =
10        this->Bricks[this->selectedTile].Position;
11        this->Bricks[this->selectedTile].Position = temp.Position;
12        std::swap(this->Bricks[this->swappableTile],
13        this->Bricks[this->selectedTile]);
14        this->selectedTile = 0;
15        this -> swappableTile = -1;
16    }
17 }
18
19 void GameLevel::selectPiece() {
20     this->Bricks.at(selectedTile).Selected = true;
21     if(this->Bricks.at(selectedTile).Selected) {
22         this->confirmSwap();
23     }
24 }
```

6.5.2 Matching logic

When 3, 4 or 5 tiles get match in a row or column, matching function removes those tiles, fills the empty space with blocks from above and fills the final empty space with randomly generated tiles

Finding matches To simplify finding matches first game objects tiles is transformed into vector of ids, then this vector is converted into vectors of columns and rows and finally those vectors are checked for matches

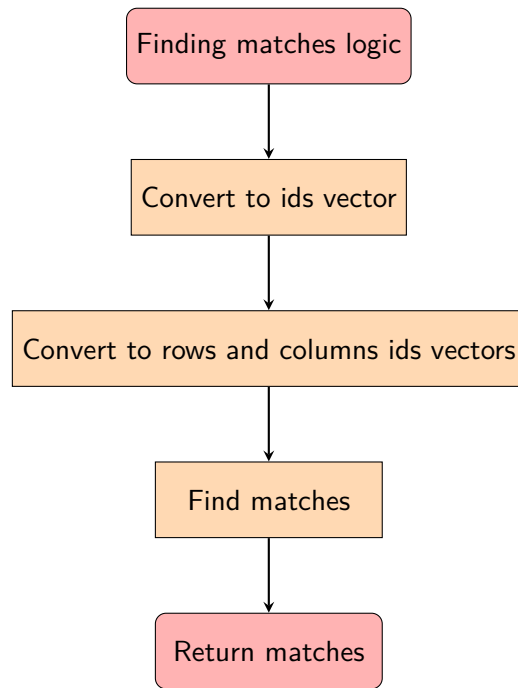


Figure 21: Finding matches logic

6.5.3 Processing matches

Removing blocks After finding matches, blocks where matches appear are set to be replaced by replacing their tile ids with -1

Replacing blocks After setting tiles to be replaced, algorithm goes through blocks with ids equal to -1, checks if there are any blocks above, if yes it replaces the block from above with block from below, if there is no block above, it generates one randomly

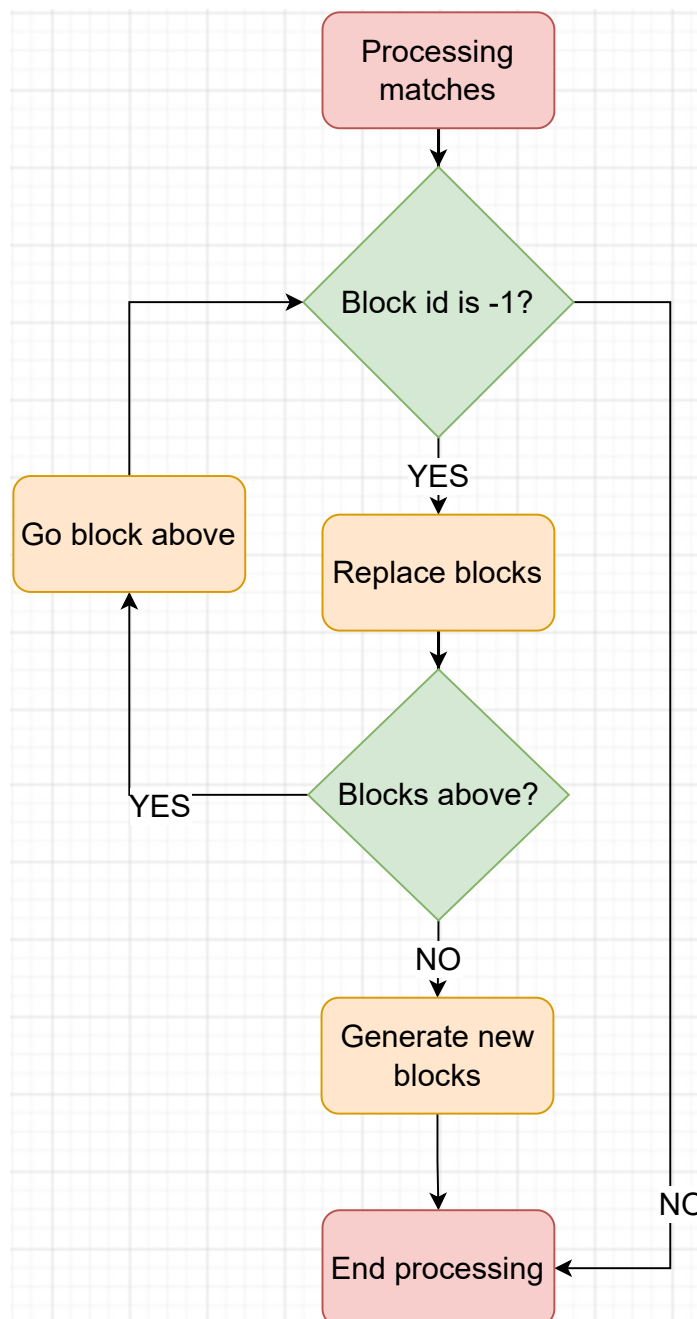


Figure 22: Logic for processing matches

6.5.4 Game objects

Game object class holds informations about tiles, its position, textures and statuses

Tile draw Based on a status of tile it can be drawn in 4 ways:

1. No status at all, it is drawn without any color overlayed on it
2. Highlighted, it is drawn with red overlay
3. Selected, it is drawn with green overlay
4. ToSwap, it is drawn with orange overlay

```
1
2 void GameObject::Draw(SpriteRenderer &renderer)
3 {
4     if(ToSwap) {
5         glm::vec3 color = hexToVec3("#f97316");
6         renderer.DrawSprite(this->Sprite, this->Position, this->Size,
7             this->Rotation, color);
8         return;
9     }
10    if(!Highlighted) {
11        glm::vec3 empty(1.0f, 1.0f, 1.0f);
12        renderer.DrawSprite(this->Sprite, this->Position, this->Size,
13            this->Rotation, empty);
14        return;
15    }
16    if(!Selected) {
17        glm::vec3 red(1.0f, 0.0f, 0.0f);
18        renderer.DrawSprite(this->Sprite, this->Position, this->Size,
19            this->Rotation, red);
20        return;
21    }
22    glm::vec3 green(0.0f, 1.0f, 0.0f);
23    renderer.DrawSprite(this->Sprite, this->Position, this->Size, this->Rotation,
24        green);
25 }
```

Engine uses two helpful methods to easily translate color hex to glm::vec3 color

```
1 glm::vec3 hexToVec3(const std::string& hex) {
2     if (hex.size() != 6 && hex.size() != 7) {
3         throw std::invalid_argument("Invalid hex string length.");
4     }
5     size_t offset = hex[0] == '#' ? 1 : 0;
6
7     int r = std::stoi(hex.substr(offset, 2), nullptr, 16);
8     int g = std::stoi(hex.substr(offset + 2, 2), nullptr, 16);
9     int b = std::stoi(hex.substr(offset + 4, 2), nullptr, 16);
10
11     return glm::vec3(r / 255.0f, g / 255.0f, b / 255.0f);
12 }
13
14 glm::vec3 vectorToVec3(const std::vector<int>& vec) {
15     if (vec.size() != 3) {
16         throw std::invalid_argument("Vector must have exactly 3 elements.");
17     }
18
19     return glm::vec3(vec[0] / 255.0f, vec[1] / 255.0f, vec[2] / 255.0f);
20 }
```



Figure 23: Game object states

6.5.5 Text Renderer

Text renderer class loads the library and single characters of font used to render text, applies text shaders, sets the text at the position requested by engine and removes text after it is not used. We also use a shader for text, we will describe the class and shader in this section.

initialization When text render class is initialized, it loads the correct shader and sets it up to work.

Loading font

When initializing text render objects, we provide a ttf file from resources/fonts folder. Text render object calls free type library to load a font.

Initializing loading First we clear any characters that were loaded previously, initialize freeType library and check if it was correctly loaded.

```

1 void TextRenderer::Load(const std::string &font, unsigned int fontSize)
2 {
3     // first clear the previously loaded Characters
4     this->Characters.clear();
5     // then initialize and load the FreeType library
6     FT_Library freeTypeLibrary;
7     // all functions return a value different than 0 whenever an error occurred
8     if (FT_Init_FreeType(&freeTypeLibrary)) {
9         std::cout << "ERROR::FREETYPE: Could not init FreeType Library" <<
10         std::endl;
11     }
12     ...

```

Actual loading and configuration We load the font using freeType, after checking if it was loaded correctly we set size of a single character, disable byte-alignment in order for the text to be rendered correctly, preload first 128 [ASCII] characters and since we already loaded everything we wanted we destroy FreeType library

```
1 void TextRenderrer::Load(const std::string &font, unsigned int fontSize)
2 {
3     ...
4     FT_Face face = this -> loadFontAsFace(freeTypeLibrary, font);
5     // set size to load glyphs as
6     FT_Set_Pixel_Sizes(face, 0, fontSize);
7     // disable byte-alignment restriction
8     glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
9     face = this -> preloadCharacters(face);
10    this -> destroyFreeType(face, freeTypeLibrary);
11 }
```

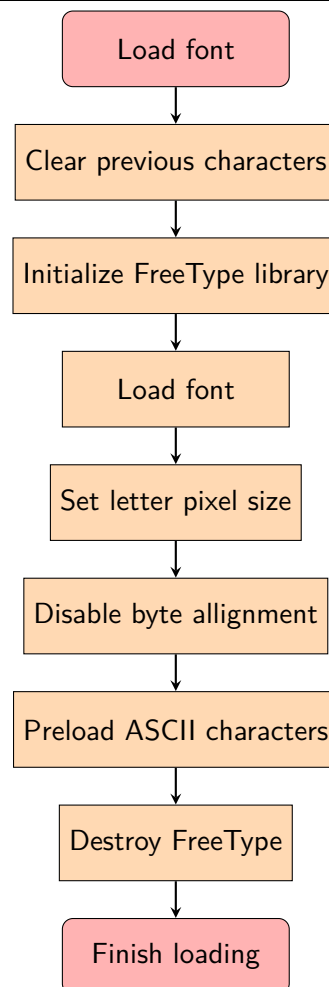


Figure 24: Loading font logic

Loading characters

In order to preload ASCII characters we use a method loadCharacter from TextRenderrer, it uses FreeType library to load character, then generates textures for a character (it needs to allign a texture to character), sets options for texture and adds the character object to characters array

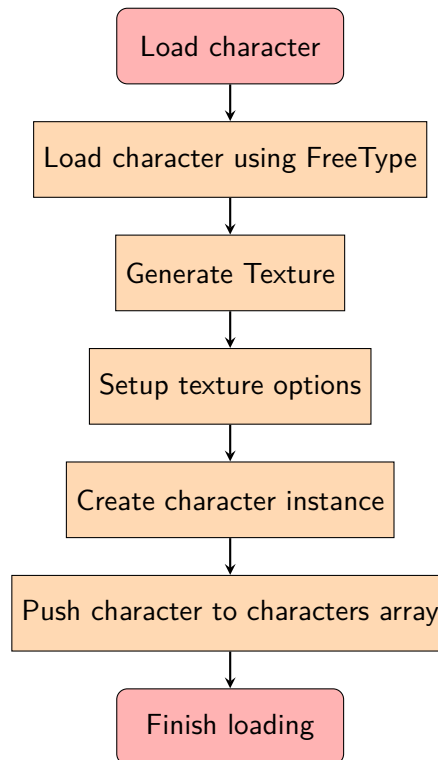


Figure 25: Loading single character logic

Texture options

- The first two lines set the wrapping mode of the texture in the S and T directions (usually the x and y axes) to `GL_CLAMP_TO_EDGE`. This means if the texture coordinates go beyond `[0,1]`, the texture will not repeat but instead clamp to the edge values.
- The next two lines set the minification and magnification filter to `GL_LINEAR`. This means when the texture is scaled down or up, it will use linear interpolation between the texture coordinates, resulting in a smoother texture appearance.

```
1 void TextRenderer::setTextureOptions() const {  
2     // set texture options  
3     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);  
4     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);  
5     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);  
6     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);  
7 }
```


Rendering text

We are given text as a C++ string and a position, method is supposed to automatically render text so that it fits in a position that was provided

We achieve that by iterating through each character of a string and putting it next to each other at a gap equal to a previous character width and height

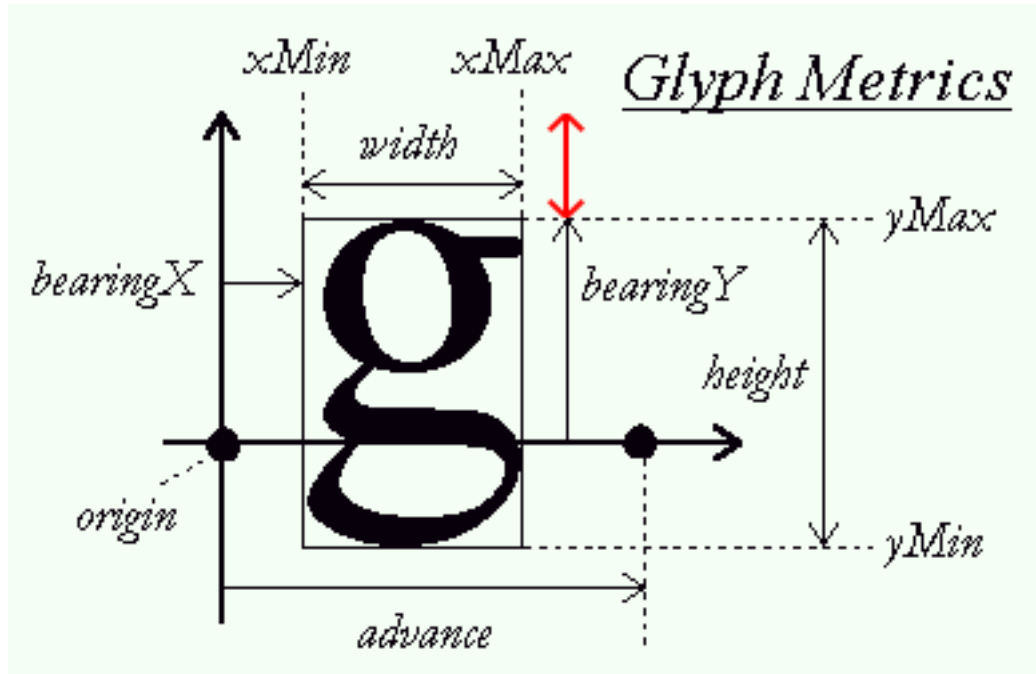


Figure 26: Glyph metrics from learnopengl

Calculating x and y positions of each character

x_{pos} and y_{pos} is the final x/y position of character

x_{start} and y_{start} is the position inputed to the method render text

$x_{bearing}$ and $y_{bearing}$ is a parameter shown in image above

s is scale

We calculate final bearing of y_{pos} by subtracting bearing of "H" character from the bearing of actual character, we do it this way because "y" bearing of "H" character is highest

$$x_{pos} = x_{start} + x_{bearing} * s$$

$$y_{pos} = y_{start} + (H_{bearing} - y_{bearing}) * s$$

6.5.6 Resource Manager

Resource manager class loads textures and shaders from files.

Loading textures After giving a name of texture to a method it is loaded (converted to raw byte data) using stb image library, we receive texture width, height and number of channels and later use it to generate texture using texture class. At the end we free image data

```
1 // RGB for Red Green Blue, RGBA for alpha channel
2 GLenum getTextureFormat(int nrChannels) {
3     switch (nrChannels) {
4         case 1: return GL_RED;
5         case 3: return GL_RGB;
6         case 4: return GL_RGBA;
7         default:
```

```

8         throw std::runtime_error("Unsupported number of channels in the
9         image.");
10    }
11 }
12 Texture2D ResourceManager::loadTextureFromFile(const char *file)
13 {
14     // create texture object
15     Texture2D texture;
16     // load image
17     int width;
18     int height;
19     int nrChannels;
20     unsigned char* data = stbi_load(file, &width, &height, &nrChannels, 0);
21     GLenum format = getTextureFormat(nrChannels);
22     texture.Internal_Format = format;
23     texture.Image_Format = format;
24     // now generate texture
25     texture.Generate(width, height, data);
26     // and finally free image data
27     stbi_image_free(data);
28     return texture;
29 }

```

Loading shaders

In order to load shaders we load up to three different source codes for vertex, fragment and geometry shaders and compile them in shader class.

Shader types

- **Vertex Shader:** This is the first stage in the graphics pipeline that processes each vertex and is responsible for transforming vertex positions from object space (local coordinates) to clip space. Additionally, it can manipulate vertex attributes such as colors, normals, and texture coordinates.
- **Fragment Shader:** Often referred to as a pixel shader, this stage operates on each fragment (potential pixel) generated by rasterization. It determines the final color of the pixels on the screen. Fragment shaders can apply textures, compute lighting, and handle other surface details.
- **Geometry Shader:** This is an optional shader stage that sits between the vertex and fragment shaders. It can generate new graphics primitives (like points, lines, and triangles) from input primitives. It offers more flexibility in processing than the vertex shader but can be computationally intensive.

6.5.7 Shader class

Shader object compiles shader from shader string code and sets its input parameters

Compiling shader

When compiling shader we provide shader source code, create shader from shader type (vertex, fragment or geometry), compile it, attach to program and link program so that it can be used by engine, at the end we remove the shader code as it is already in our engine in order to free resources.

```

1 unsigned int Shader::compileShader(const char *shaderSource, const std::string&
  type, const GLenum shaderType) {
2     unsigned int shaderPointer = 0;
3     if(shaderSource != nullptr) {
4         shaderPointer = glCreateShader(shaderType);
5         glShaderSource(shaderPointer, 1, &shaderSource, nullptr);
6         glCompileShader(shaderPointer);
7         checkCompileErrors(shaderPointer, type);
8     }
9     return shaderPointer;
10 }
11
12 void Shader::Compile(const char* vertexSource, const char* fragmentSource, const
  char* geometrySource)
13 {
14     unsigned int sVertex = this->compileShader(vertexSource, "VERTEX",
  GL_VERTEX_SHADER);
15     // shader program
16     this->ID = glCreateProgram();
17     glAttachShader(this->ID, sVertex);
18     ...
19     glLinkProgram(this->ID);
20     checkCompileErrors(this->ID, "PROGRAM");
21     // delete the shaders as they're linked into our program now and no longer
  necessary
22     glDeleteShader(sVertex);
23     ...
24 }

```

6.5.8 Dependency Management

There are several libraries: OpenGL, GLFW, stb_image and freetype being integrated into engine, they are simply included at the top of any files that need them.

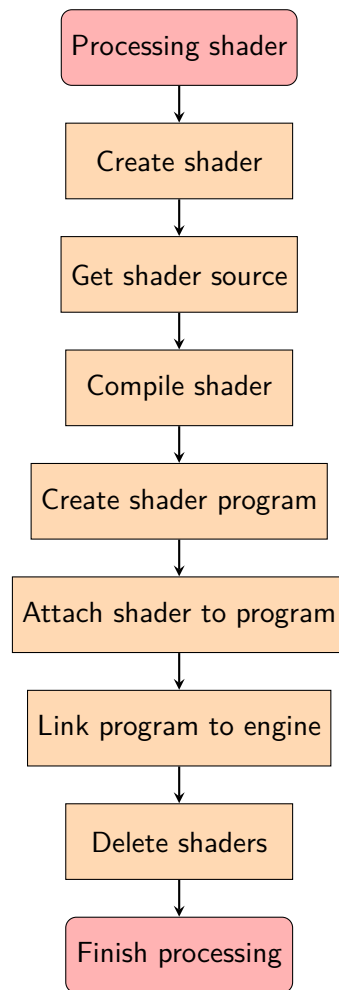


Figure 27: Processing shader

Chapter 7

Practical Use of Engine

In the previous chapters, we explained on the architecture, design, and construction of our match-three game engine. This chapter focuses on practical application of this engine by illustrating the creation of a functional match-three game from scratch.

7.1 Initial Setup and Configuration

Before we start creating the game, we need to set up the development environment to utilize our game engine's capabilities fully. Using a modern IDE (Integrated Development Environment) like Visual Studio or Code::Blocks can ease the process.

Importing the engine Initiate a new C++ project and import our game engine's header and source files. Ensure that the necessary libraries related to OpenGL are linked.

Configuration of OpenGL Validate that the OpenGL version compatible with our engine is installed. This often involves setting up GLEW (OpenGL Extension Wrangler Library) or similar middleware.

7.1.1 Choosing assets

Choosing assets is the most important part of our match three engine design.

We are going to use site <https://opengameart.org/> which offers free game assets with permissive license. We are going to search for assets with the most permissive CC0 license which is an equivalent to public license.

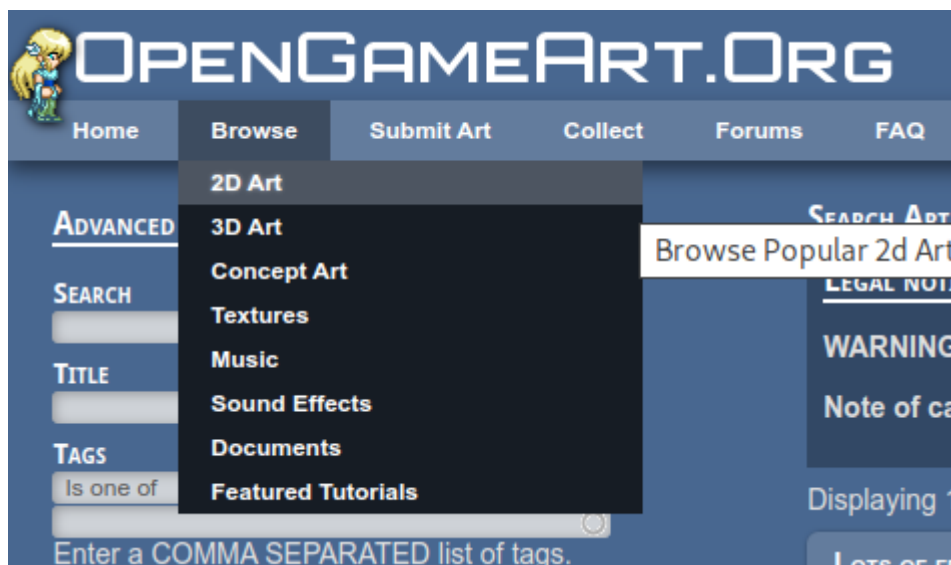


Figure 28: Searching for 2D art on OpenGameArt

OPENGAMEART.ORG

Home Browse Submit Art Collect Forums FAQ

ADVANCED SEARCH

SEARCH
match three

TITLE
[Empty text box]

TAGS
Is one of [Dropdown menu]
[Empty text box]
Enter a COMMA SEPARATED list of tags.
(example: "sword, weapon, item")

SUBMITTER
[Empty text box]

ART TYPE

<input type="checkbox"/> 2D Art	<input type="checkbox"/> 3D Art
<input type="checkbox"/> Concept Art	<input type="checkbox"/> Texture
<input type="checkbox"/> Music	<input type="checkbox"/> Sound Effect
<input type="checkbox"/> Document	

LICENSE(s)

<input type="checkbox"/> CC-BY 3.0	<input type="checkbox"/> OGA-BY 3.0
<input type="checkbox"/> OGA-BY 4.0	<input type="checkbox"/> CC-BY 4.0
<input type="checkbox"/> GPL 3.0	<input type="checkbox"/> GPL 2.0
<input checked="" type="checkbox"/> CC0	<input type="checkbox"/> CC-BY-SA 4.0
<input type="checkbox"/> CC-BY-SA 3.0	

SORT BY
Favorites [Dropdown menu]

ORDER
Desc [Dropdown menu]

ITEMS PER PAGE
24 [Dropdown menu]

COLLECT INTO...
Select a collection [Dropdown menu]

SEARCH

Figure 29: Searching for CC0 licensed assets

Filtering non tiles assets Our first step is to filter all assets that refer to graphics that cannot be used as tiles, for example UI elements, icons or buttons

Filtering tile assets

We end up with 4 assets to choose from, based on 2 parameters we filtered 3 of them

Using mechanics not available in engine First we filter asset pack that uses dynamic falling mechanic which is not present in our engine

Too much details Two more tile assets use too much details inside the tiles, this would require our game to react to those details to make sure that the gameplay experience is compatible with the tiles look.

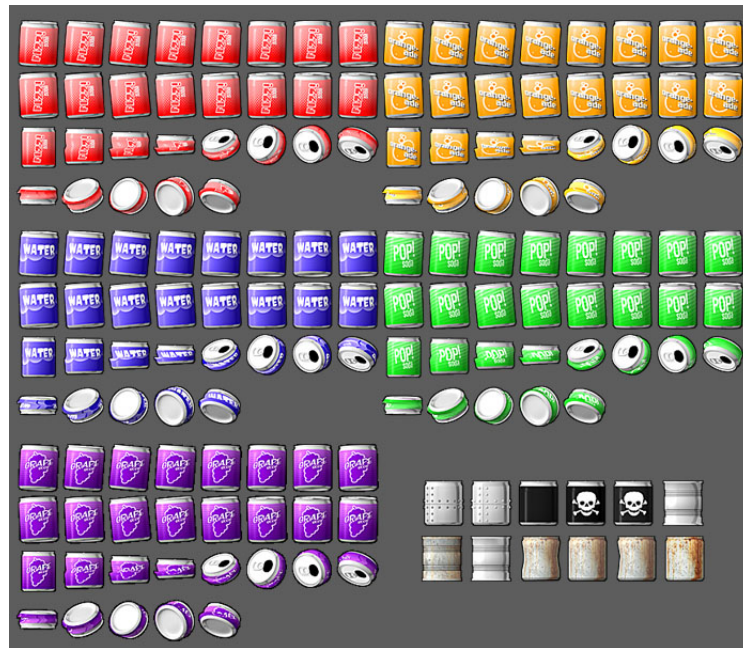


Figure 30: Cans have different sprite depending on if they are falling or not, we do not have such mechanic in our engine [Pelikan \(2012\)](#)



Figure 31: First assets with too much details inside the detail [ChiliGames \(2017\)](#)



Figure 32: Second assets with too much details inside the detail [Halcyon \(2021\)](#)

Chosen assets Finally we settled on these tiles, they look simple, work well with our shaders and engine gameplay and offer high quality non-pixelated files



Figure 33: Chosen assets [Match three assets](#) (2017)

7.1.2 Loading assets

In order to use our assets in the game engine we need to define their names in the engine code and put the files into correct folder

Setting files All non-code files used in our engine are stored in resources older

Textures in particular are stored in "textures" folder

That is where we unpack our tiles, files in the unzipped folder are named with a pattern: "Gem Type[number] [color].png"

In order to simplify importing assets name we created a short python script that renames all tiles to "g[number][color].png"

```
1 import os
2 import re
3
4 def rename_files_in_directory(directory_path):
5     # List all files in the directory
6     for filename in os.listdir(directory_path):
7         # Match filenames using regex
8         match = re.match(r"Gem Type(\d+) (\w+)\.png", filename)
9         if match:
10             number = match.group(1)
11             color = match.group(2).lower() # Convert to lowercase
12             new_filename = f"g{number}{color}.png"
13             # Rename files
14             os.rename(
15                 os.path.join(directory_path, filename),
16                 os.path.join(directory_path, new_filename)
17             )
18
19 # Call the function with the path to your directory
20 rename_files_in_directory("./")
```


Setting tiles names in constants file After choosing tiles we are going to use we modify constants.hpp file, namespace textures and add our tiles tot textures array

```
1 namespace textures {
2     static constexpr TextureInfo textures[] = {
3         {"resources/textures/background.jpg", "background"},
4         {"resources/textures/g1black.png", "block_black"},
5         {"resources/textures/g2blue.png", "block_blue"},
6         {"resources/textures/g3green.png", "block_green"},
7         {"resources/textures/g4purple.png", "block_purple"}
8     };
9 }
```

7.2 Game Concept and Design

7.2.1 Game Concept

A game concept encapsulates the core idea or essence of a game. It's the core vision that drives the entire game development process, giving direction to the gameplay and user experience. In match-three games, the primary concept is to align three or more similar game elements in a row or column to achieve points.

Our concept In order to benchmark our game engine we are going to focus on simplicity and rudimentary features of match-three games. We are going to implement a game that while offers only elementary features can still be called a match-three game and can be won or lost

7.2.2 Game Design

Game design describes how a game should be structured and played. It's the process of deciding game rules, creating levels, setting challenges, and determining how players interact with the game. While the game concept is the overarching idea, game design provides the detailed blueprint. In match-three games factors such as how new elements are introduced, the complexity of levels, the scoring system and win/fail condistions are all vital components of the design.

Our design

Scoring system Let's devise a scoring system for a match-three game with increasing difficulty levels. We will also differentiate the scores based on matching 3, 4, or 5 tiles.

First, let's determine the scores for each matching set:

1. Match 3 tiles = 10 points
2. Match 4 tiles = 20 points
3. Match 5 tiles = 40 points

When devising the point system for matching sets in the game, there are several considerations we kept in mind:

Progressive Increase We wanted the points to represent a clear progression, making matches of larger sets more valuable, incentivizing players to aim for them.

3-tile matches are the baseline, so they were assigned a modest 10 points.

4-tile matches are rarer and more difficult to create than 3-tile matches, so they're assigned 20 points, doubling the value from the 3-tile matches.

5-tile matches are even rarer, so they were given a point value that's a bit more than the sum of 3-tile and 4-tile matches, settling at 40 points.

Simple numbers Using round numbers like 10, 20, and 35 makes it easy for players to quickly understand and calculate their scores. Complexity can deter casual gamers, and match-three games are typically aimed at a wide audience, including those who may prefer simpler mechanics.

Winning Given these scores, let's decide the target scores for each difficulty level. We'll set these target scores based on an estimated number of matches a player might make in a given game, we will create Easy, Normal and Hard levels

- Easy Level:
 - Target: 400 points
 - This might involve roughly 40 matches 3 tiles, or a combination of fewer 4 and 5 tile matches. This offers the player plenty of room for errors and learning the game mechanics.
- Normal Level:
 - Target: 600 points
 - For this target, the player might need to make 60 matches of 3 tiles, which will not be possible since we set up the limit of matches to 40 or a combination of fewer 4 and 5 tile matches, which will force the player to use bigger matches. This requires the player to make more strategic matches and utilize power-ups or special moves (if any).
- Hard Level:
 - Target: 800 points
 - Achieving this score might involve 80 matches of 3 tiles, or a combination of fewer 4 and 5 tile matches. With our limit of matches set up to 40 will force the player to keep the average matching to at least 20 points per match. This level is challenging and will likely require the player to plan moves carefully and maximize opportunities to match 4 or 5 tiles for higher scores.

Losing The game is lost whenever player reaches 40 turns, this forces the player to act under limited moveset and ensures that the game offers a challenge

Number 40 was chosen based on the similar choice in professional match-three games, it is the same number used for matching 5 tiles, it offers gameplay that is not too short neither too long and it is a "nice" number, even, round and with many divisors

Setting up values in constants file In order to implement this design in our game engine we need to again modify constants.hpp file in following way:

```
1 struct DifficultySettings {
2     const std::string name;
3     const int points;
4     const int limit;
5     const std::vector<int> matchScores;
6
7     DifficultySettings(const std::string& n, int p, int l, const std::vector<int>&
8         scores)
9         : name(n), points(p), limit(l), matchScores(scores) {}
10 };
11 namespace difficulty {
12     std::vector<DifficultySettings> settings = {
13         DifficultySettings("easy", 400, 40, {10, 20, 40}),
14         DifficultySettings("normal", 600, 40, {10, 20, 40}),
15         DifficultySettings("hard", 80, 40, {10, 20, 40})
16     };
17 }
```

7.2.3 Core Mechanics and Gameplay

Game utilizes full functionality of our engine, player needs to match three or more gems, chaining matches or achieving higher combos grants the player higher score

Generating levels We used python script to first generate levels user can choose from in order to simplify our work so we do not need to create the levels on our own
Later in a course of our game the game engine handles generating new random blocks when old ones are replaced

7.2.4 User Interface and User Experience

The Heads-Up Display is minimalistic, showcasing only essential elements, score and moves left.

Choosing level and difficulty User can choose both level and difficulty on the menu screen using WSAD keys

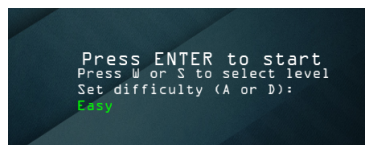


Figure 34: UI when choosing easy level

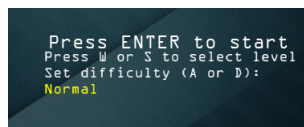


Figure 35: UI when choosing normal level

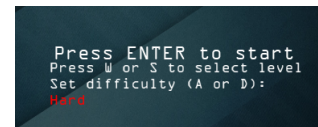


Figure 36: UI when choosing hard level

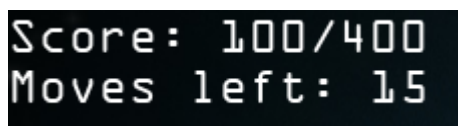
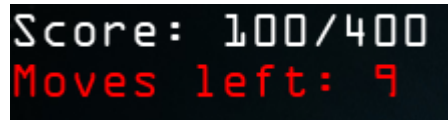


Figure 37: Score and moves left text



Score: 100/400
Moves left: 9

Figure 38: Score and moves left text, when there are less than 10 moves left

Losing and winning Whenever player loses (reaches max number of turns) or wins (reaches target points), game stops, clears the level and displays a message

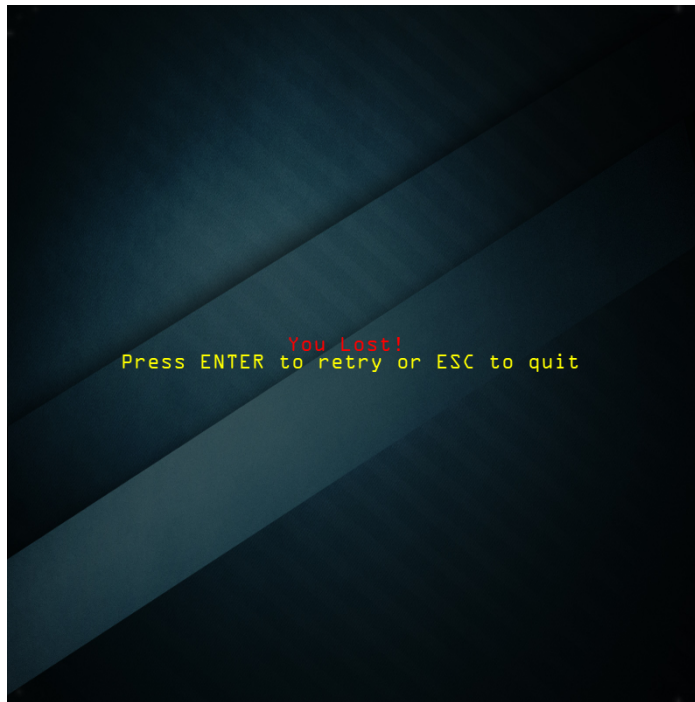


Figure 39: Game message when lost

7.2.5 Advantages of a Custom-built Engine

- **Optimized Performance:** Tailored specifically for a Match Three game, the engine could be highly optimized for this genre, reducing unnecessary overheads. Using OpenGL and plain C++ also added to engine speed
- **Direct Control:** Having complete control over engine we could introduce, modify, or remove features as needed.

We were able to create a game using our engine, which means that the goal of this entire thesis was reached.



Figure 40: Game message when won

Chapter 8

Future Work and Enhancements

8.1 Potential Extensions to the Engine

While we were able to produce a game using our engine, it still has a potential to grow more and introduce many new features.

8.1.1 Enhanced Graphics and Physics

Enhancing the visual and interactive experience:

- Ray Tracing: Using ray tracing can provide more realistic lighting, shadows, and reflections, increasing visual depth with little cost to an engine user.
- Advanced Particle Systems: Enhancing particle effects can create more immersive environments, from weather effects to more dynamic game elements.
- Physics Enhancements: More advanced physics simulations can improve feel to game interactions, from simple collisions to complex motion dynamics.

8.1.2 Adaptive Difficulty

Machine learning can analyze player performance and adjust game difficulty in real-time, ensuring a balanced challenge.

8.1.3 Real-time Multiplayer

Adding capabilities for online multiplayer modes can increase engagement, replayability and competitiveness.

8.1.4 More platforms

Engine could be adapted to work under mobile systems (Android and IOs) and consoles (Xbox, PlayStation and Switch)

As can be seen there are still a lot of features that can be added to our engine to make it even more useful

Chapter 9

Conclusion

9.1 Summary of Achievements

In this section we will look back at our project and describe our achievements.

9.1.1 Multiplatform Support

- **OS Versality:** We successfully developed a game engine that operates across Windows, Mac, and GNU/Linux platforms, addressing the title of this thesis.
- **Addressed Platform-specific Challenges:** Overcame technical challenges associated with different operating systems, ensuring a consistent user experience.

9.1.2 Library Integrations

We managed to integrate libraries such as GLFW, glad, freetype, stb_image, and ft2build, all within one game engine

- **GLFW/glad:** easing the process of using OpenGL
- **freetype/ft2build:** allowing for rendering text within game
- **stb_image:** for ease of importing image assets into game engine

9.1.3 Robust Game Mechanics

- **Match Three Logic:** Core match three logic was implemented, matching 3 or more tiles, chain reactions and limited moves
- **Event Management:** The game responds successfully to user interactions

9.1.4 Practical usage

We managed to create a fully functional match-three game, using only our game engine and royalty free assets.

9.1.5 Contributions to the Field of Game Development

Our engine allows aspiring game developers to create their own match three games, and engine developers to learn from it and expand it.

In conclusion, we managed to accomplish all of the goals we set for this thesis, our engine successfully produces a game for all three operating systems we were aiming for.

9.2 Reflection on the Development Process

Creating this game engine was a long process, it took almost a year to bring it to the state it is now in, during this time several reflections came about.

9.2.1 Embracing the Multiplatform Challenge

In order to overcome differences for different operating system, a lot of time was put into choosing correct graphical api and libraries that could work on Windows, GNU/Linux and MacOS. This proved crucial when later developing the engine as this issue did not come back and I could focus on creating the engine itself.

9.2.2 Library Integrations

Using multiple different libraries and integrating them with our game engine was a challenge, thanks to existing solutions like learnopengl repository which already overcame those difficulties and KDevelop easy approach to external libraries it was possible to implement all of libraries within our game engine.

9.2.3 Time management

Lastly, probably the biggest challenge was managing time for this thesis. Finishing it required a lot of determination, setting barriers and working every day to ensure that the thesis is finished on time. Thankfully past experiences helped in guiding through this task that could be compared to colossus.

Bibliography

Bejeweled gameplay (2001).

ChiliGames (2017), 'Vector match three blocks'. Accessed: 2023-09-08.

URL: <https://opengameart.org/content/vector-match-three-blocks>

Desktop Operating System Market Share Worldwide Chart (2023).

Free Type site (2023).

Glad configuration site (2023).

Halcyon, H. (2021), 'Color match puzzle rainbow panels'. Accessed: 2023-09-08.

URL: <https://opengameart.org/content/color-match-puzzle-rainbow-panels>

Linux kernel and OpenGL video games (2014).

Match three assets (2017).

O'Hanlon, M. (2023), 'Exploring logic gates in minecraft'. Accessed: 2023-08-20.

URL: <https://helloworld.raspberrypi.org/articles/bbcc-exploring-logic-gates-in-minecraft>

PCMag encyclopedia backslash (2013).

Pelikan, D. (2012), 'Soda pop can-themed game pieces'. Accessed: 2023-09-08.

URL: <https://opengameart.org/content/soda-pop-can-themed-game-pieces>