BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

A SIMPLE GAME ENGINE

JEDNODUCHÝ HERNÍ ENGINE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MICHAL ŠTEFÁČEK**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ STARKA**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Computer Graphics and Multimedia (DCGM) Academic year 2021/2022

Bachelor's Thesis Specification

Student:	**Štefáček Michal**
Programme:	Information Technology
Title:	**Simple Game Engine**
Category:	Computer Graphics

Assignment:

1. Study the existing game engines and algorithms needed to create one.
2. Design a simple 2D game engine.
3. Implement this engine.
4. Make a simple game or a demo application using the engine.
5. Make a short demo video.

Recommended literature:

- By the supervisor's recommendation.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Starka Tomáš, Ing.**
Head of Department:	Černocký Jan, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	May 11, 2022
Approval date:	May 3, 2022

Abstract

The usage of game engine greatly accelerates the process of game development. Many free-to-use game engines have emerged in the last few decades, some of them being open-source. The main goal of this thesis is to understand the process of developing such software. Because of its immense scope, the project is primarily oriented towards the creation of 2D games.

Abstrakt

Použití herního enginu značně urychluje proces vývoje her. Za posledních několik desetiletí vzniklo množství zdarma použitelných herních enginů, z nichž některé mají volně dostupný zdrojový kód. Cílem této práce je pochopení procesu vývoje takovéhoto softwaru. Pro zúžení rozsahu se projekt primárně zaměřuje na vývoj 2D her.

Keywords

game engine, OpenGL, 2D rendering, Java, LWJGL, JNI

Klíčová slova

herní engine, OpenGL, 2D rendering, Java, LWJGL, JNI

Reference

ŠTEFÁČEK, Michal. *A simple game engine.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Starka

Rozšířený abstrakt

Vývoj her je pomalý a náročný proces. Použití herního enginu může dramaticky snižuje čas potřebný pro tvorbu her po technické stránce. Za posledních několik desetiletí se technologie herních enginů posunula od malých knihoven obalující API pro 3D vykreslování v ucelená řešení pro množství platforem. Tato práce se zaměřuje na tématiku vývoje herních enginů a v rámci výzkumu se návrh a implementaci podobného řešení pokouší.

Herní engine může být dle účelu rozdělen na množství podsystémů, kde každý systém zastává určitou funkci. Cílem enginů je zajistit spolupráci jednotlivých systémů a správu jejich životního cyklu.

Byl proveden průzkum dvou existujících řešení — Unity a Godot Engine. Oba enginy obsahují podporu 2D vykreslování světa a textu, systém přehrávání zvuku, správu zdrojových médií a integrované řízení herní smyčky. Tyto jednotlivé prvky byly vybrány jako základní systémy potřebné pro tvorbu jednoduchých 2D her. Unity i Godot Engine implementují vykreslování ve 2D pomocí ortografické projekce a rendering textu rasterizací jednotlivých znaků písma do textury — atlasu, odkud jsou následně využívány. U jiných systémů se oba enginy více rozcházejí. Unity například modeluje jednotlivé prvky scény pomocí systému entitních komponent (ECS), zatímco Godot reprezentuje prvky jako instance tříd — objekty. Největší nevýhodou Unity je platba licenčních poplatků za určitých podmínek, Godot Engine je naopak plně zdarma a má otevřený zdrojový kód.

Herní engine byl navržen jako sada na sobě závislých komponent. Pro zjednodušení řešení se vývoj zaměřuje primárně na platformu stolních počítačů, avšak s vhodným návrhem a použitím správných technologií by platforma neměla být omezena. Pro 2D vykreslování jsou využity standardní metody graficky akcelerovaného 3D renderingu s ortografickou projekcí, zamezujíc vzniku nežádané perspektivy. Navrhovaný engine úmyslně vynechává podporu systému prvků scény a herní smyčky a ponechává implementaci tohoto systému na vývojářích konkrétních her.

Engine byl implementován v jazyce Java s využitím knihovny LWJGL pro použití grafického rozhraní OpenGL. Byly implementovány dva systémy pro vykreslování textu — prvním z nich jsou rastrová písma, druhým písma s využitím technologie distančních polí načítána pomocí knihovny `stb_truetype`. Při implementaci byl kladen důraz na modularitu výsledného řešení, jednotlivé moduly jsou načítány za běhu aplikace pomocí jazykové introspekce centrálním správcem modulů. Byly vyvinuty různé typy grafických 2D prvků — spritů — statické, orientované a animované. Při vykreslování je možné aplikovat různé efekty pomocí GLSL shaderů. Zvukový podsystém byl implementován na rozhraní OpenAL, s podporou zvuků ve formátu Ogg Vorbis, načítaných knihovnou `stb_vorbis`. Výstupem sestavení projektu je sada Maven knihoven, dostupná z veřejného repozitáře balíčků.

V rámci vyhodnocení práce byl engine porovnán s dříve prozkoumanými řešeními. Bylo nalezeno několik nedostatků, jako je absence podpory systému kamer a rozdělení vykreslování GUI a 2D světa, nedostatečná abstrakce v některých systémech a vyšší náročnost implementace základních herních prvků, způsobená absencí systému scén. Na druhou stranu je engine schopen kvalitního a efektivního vykreslování textu s využitím rastrových fontů či fontů na bázi distančních polí. Některé součásti projektu je nutné modularizovat, například systém 2D vykreslování.

Následně byla na enginu reimplementována existující hra, původně napsaná v jazyce C s využitím platformy SDL2. Na základě provedených testů nedošlo k žádnému razantnímu poklesu výkonu v porovnání s nativní implementací hry. Na platformě Windows byl zaznamenán rozdíl ve výkonu renderingu, ale následná měření naznačují, že se jedná o rozdíl v efektivitě ovladačů rozhraní OpenGL napříč platformami.

Výstupem práce je využitelný herní engine pro tvorbu menších 2D her, několik ukázek daného enginu a video popisující aktuální stav řešení a obsahující návod pro vytvoření nového projektu založeném na tomto enginu. Toto řešení může být dále rozšiřováno jako cíl dalšího výzkumu.

A simple game engine

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by me under the supervision of Mr. Tomáš Starka. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Michal Štefáček
May 10, 2022

Acknowledgements

I would like to thank my supervisor, Mr. Tomáš Starka for assistance in creation of this work, as well as the authors of all sources listed. I would also like to thank Lucy for mental support and proofreading.

Contents

Chapter 1

Introduction

Game development is a very slow and complicated process. The usage of a game engine greatly reduces time needed to build such a project. Game engines have evolved over the years, from simple abstractions over rendering APIs with some utility features to massive multi-project platforms.

The development of game engines is complicated, even when broken down into smaller parts. Despite that, many small projects have emerged, some of them becoming successful. The goal of this thesis is to design and subsequently implement such game engine, while keeping it simple and using open standards and open-source software components. As creating a general purpose game engine is way out of scope of this project, only a small subset of features and platforms was selected.

In chapter 2, this thesis provides a look into existing solutions, as well as the foundation of systems necessary to create a viable 2D game engine. This includes breaking down such software into individual components and performing a deep analysis, as there is no need to reinvent existing industry standards.

Chapter 3 follows up with proposals for solutions of problems and questions formed during the research. An overall architecture of a game engine is created, with individual features abstracted as components and modules.

The implementations of solutions proposed are then described in chapter 4. Every subsystem is developed using a set of available open technologies, as well as an intermediate infrastructure allowing systems to communicate is created.

Ultimately, in chapter 5, the final product is evaluated against the existing solutions, followed by the process of testing the implemented features and comparison to their expected behaviour.

Chapter 2

Game engines

A *game engine* is a piece of software designed for development of a certain set of games on a certain set of platforms [4]. Some game engines may be built for a wide range of game genres and platforms, while the majority is designed for a specific use case.

Some games use a custom engine by choice or due to special requirements, namely Noita[1] (figure 2.1) for its cellular automata particle simulations, Don't Starve[2] with its 2.5D unique art style, or Factorio[3] for high-performance factory simulation and mod support. However, many 2D games use an existing solution to save on development time and costs.

Figure 2.1: A screenshot of the game Noita powered by the Falling Everything[4] engine. Image courtesy of Nolla Games.

[1] https://noitagame.com
[2] https://www.klei.com/games/dont-starve
[3] https://www.factorio.com/
[4] https://nollagames.com/fallingeverything/

2.1 Case studies

As many solutions exist in this field, a selection of them may be analyzed to scout for key features, standards, and conventions. In particular, game engines designed for 2D game development should be prioritized, especially the free ones, as they are more likely to be picked by small game developers.

The game engines mentioned in this section will be referenced throughout the remainder of this chapter.

2.1.1 Unity

Unity[5] is a proprietary general-purpose multi-platform real-time application development platform created by Unity Technologies, suitable for the development of both 2D and 3D games. Unity provides an integrated editor, as seen in figure 2.2. Online documentation is available[6].

A vast range of platforms and devices is supported with a large amount of tooling and support, making Unity the de facto industry standard for small and medium size game development.

Because Unity is primarily a 3D game engine, some features may be more complicated to implement compared to a dedicated 2D game engine. While Unity is free for personal use, a commercial license must be purchased under certain conditions[7].

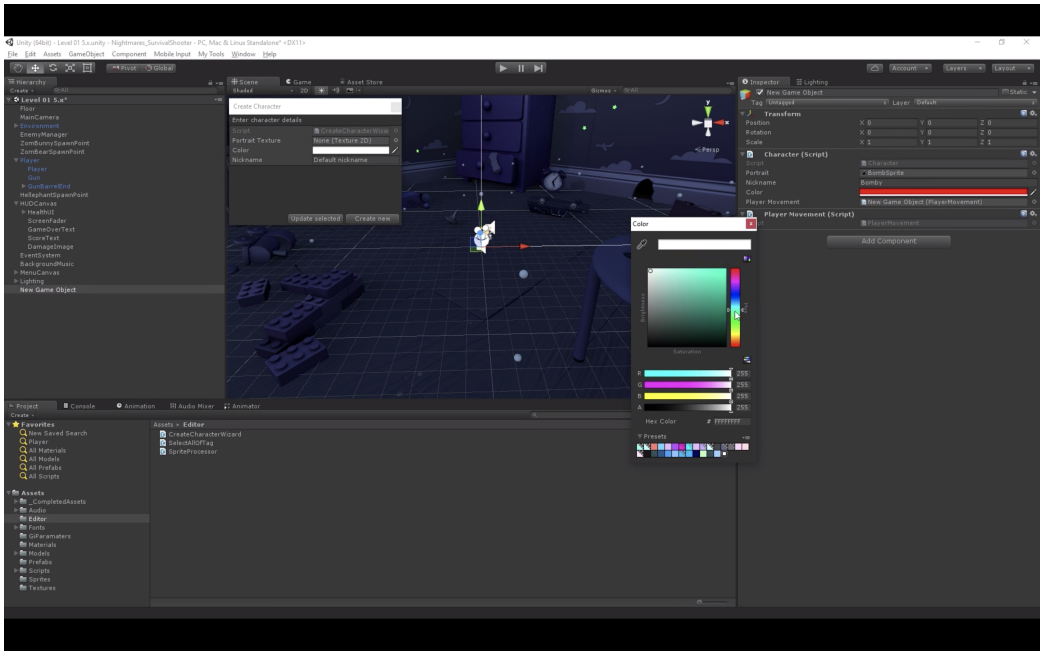


Figure 2.2: A screenshot of a project opened in the Unity Editor, image courtesy of Unity Technologies.

[5] Available from https://unity.com/

[6] Available at https://docs.unity3d.com/Manual/UnityManual.html, retrieved 2022-05-10.

[7] Based on https://store.unity.com/compare-plans, retrieved 2022-05-10.

Popular 2D games recently released and based on the Unity engine include Among Us[8], Cuphead[9], Enter the Gungeon[10], or Hollow Knight[11] (figure 2.3).

Figure 2.3: A screenshot of gameplay from the game Hollow Knight based on the Unity engine, image courtesy of Team Cherry.

2.1.2 Godot Engine

Godot Engine[12] is a multi-platform free and open-source game engine. Just like Unity, Godot comes with an included editor, screenshot of which may be seen in figure 2.4. Online documentation is available[13].

Since Godot Engine is a community-driven project, commercial support is not available. Combined with a slower development pace, this is a major disadvantage compared to commercial solutions. On the other hand, no licensing fees are attached.

[8] https://www.innersloth.com/games/among-us/
[9] http://cupheadgame.com/
[10] https://www.dodgeroll.com/gungeon/
[11] https://hollowknight.jp/
[12] Available from https://godotengine.org/
[13] https://docs.godotengine.org/en/stable/

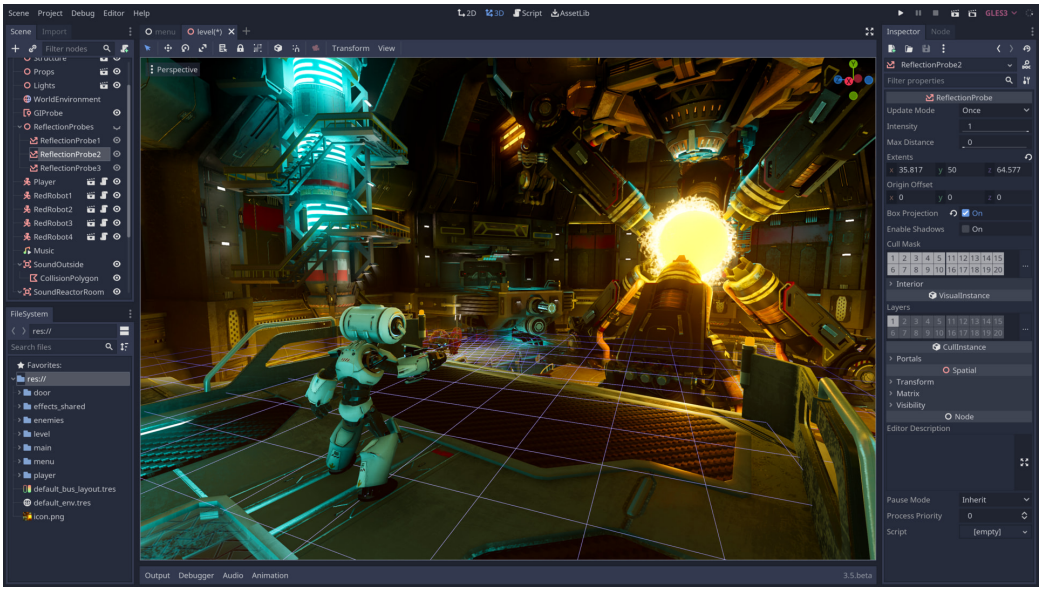


Figure 2.4: A screenshot of the Godot Engine editor, image courtesy of the Godot Engine project contributors.

2D games developed on the Godot Engine include Bottomless[14] (figure 2.5), Kingdoms of the Dump[15], or ROTA[16].

Figure 2.5: A screenshot of the game Bottomless built on the Godot Engine. Image courtesy of Raffaele Picca.

[14] https://picster.itch.io/bottomless
[15] https://kingdomsofthedump.com/
[16] https://harmonyhoney.itch.io/rota

2.2 Engine subsystems

The specific needs of games in the target genre must be taken into account when designing a game engine. These functions may be categorized into individual *subsystems* [4]. Building an infrastructure around them, these systems are then combined into a game engine.

Each system has a certain lifetime and must be initialized and shut down properly [4]. This is usually done by the engine automatically.

2.3 Resource management

A game engine should handle the lifetimes of individual *resources* on behalf of the game developer. A resource may be any kind of media used by a game [4] — such as images, audio clips, fonts, localization definitions, configuration files, or engine-specific structures.

After importing an asset into Unity, it is stored in a database, from which it may be referenced using a path.

Godot Engine uses Universal Resource Identifier (URI) based path identifiers for individual resources. Imported assets are converted to a custom format storing all necessary metadata.

2.4 Game loop and scene management

A game is essentially a *simulation* [4]. However, aside from running the simulation, game engines also have to present the state of the game to the player, as well as receive inputs and translate them into simulation events.

Modern game engines represent the state of the game as a *scene* [4], a graph (commonly a tree) of individual game objects — *entities*. Some game engines contain a graphical editor to manage scenes with a real-time preview, as is the case with both Unity and Godot Engine.

Unity and Godot Engine take different approaches for the representation of scene objects. Unity implements the *entity component system* (ECS) design pattern, Godot Engine represents objects as standard class instances with the use of inheritance. ECS is a compositional design pattern, where certain properties of game entities are decoupled into smaller components, separating system logic and data [2].

2.5 Rendering

In 2D games, the most commonly used unit of rendering is a *sprite* — a quadrilateral, usually a square or a rectangle, with an assigned bitmap texture [4].

Unity provides a highly abstract and cross-platform renderer, backed by either DirectX, Metal, OpenGL, or Vulkan; based on the target platform. Shaders are primarily written in the HLSL programming language.

Godot by default uses a renderer based on OpenGL ES, a cross-platform subset of the OpenGL rendering API[17].

In both Unity and Godot Engine, sprites are a first-class member of the scene hierarchy. All objects are rendered in 3D, leveraging orthographic projection to avoid perspective distortion.

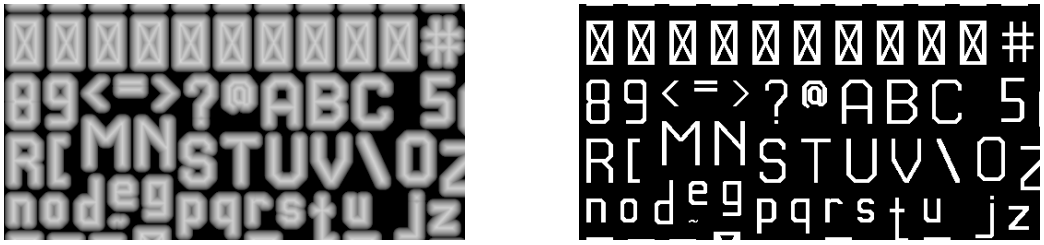
[17] https://www.khronos.org/opengles/

2.6 Font rendering

Text is a ubiquitous element in the space of user interfaces. Text can be drawn to the screen using several approaches. One of them is the rendering of *glyphs* from a bitmap texture — an *atlas*. A glyph is a graphical representation of a single character [10]. However, this poses several technical difficulties. Glyphs rendered using standard "textured rectangle" techniques, with the use of texture filtering or without, will only look high-fidelity at certain resolutions. While this effect may be desired in some cases (e.g. pixel-art games), many fonts are stored in a vector format.

As rasterization of glyph outlines in real-time or even rendering the glyphs as a mesh would be unreasonably costly, each glyph is rasterized only once and extensively cached [10]. However, the fact fonts are rasterized for a given size brings back the problem of scaling. Rasterizing glyphs for each font size used consumes high amounts of memory — especially for large sizes, where each glyph may be hundreds of pixels in each dimension.

One of the solutions for this issue is the usage of *distance field* bitmaps to describe the glyph outlines [3]. A distance field is a field of scalars, describing the minimum distance to a certain shape [1]. In *signed* distance fields, given a threshold "on-edge" value, points sampled inside the shape have a greater distance value than the threshold, while points outside the shape have a lower value; or vice versa. Leveraging the linear texture filtering and programmable shader features of modern GPUs, uniform sampling of distance fields can be achieved with little to no performance costs. Relatively low-resolution bitmaps may be used to render high-resolution text using much lower resolution glyph textures. A comparison of the input bitmap and the result is shown in figure 2.6.



(a) Before (b) After

Figure 2.6: A region of a texture atlas containing packed distance fields of glyphs, (a) before and (b) after applying a threshold filter.

There are several approaches for sampling distance field textures [3], two of the the most common ones are *thresholding* and *inverse linear interpolation*. A visual comparison of both techniques can be seen in figure 2.7. Defining the edge as a numerical range instead of a single threshold creates significantly smoother outlines for drawn text.

Given an input sampled grayscale value $y \in \langle 0, 1 \rangle$, the threshold filter outputs an alpha value a as described in formula 2.1a, parametrized by the threshold $T \in \langle 0, 1 \rangle$, offsetting the edge inside or outside. The inverse linear interpolation (formula 2.1b) filter defines two parameters instead, T_1 and T_2, where $0 < T_1 < T_2 < 1$.

$$a = \begin{cases} 1 & y \geq T \\ 0 & \text{else} \end{cases} \tag{2.1a}$$

$$a = \begin{cases} 0 & y < T_1 \\ \frac{y - T_1}{T_2 - T_1} & T_1 \leq y \leq T_2 \\ 1 & y > T_2 \end{cases} \tag{2.1b}$$

(a) Naive thresholding (b) Inverse linear interpolation

Figure 2.7: A close-up of two approaches of sampling distance fields.

The size of glyph atlases may further be brought down with the usage of various compression mechanisms, such as *rectangle packing* of glyphs in the resulting atlas, or storing multiple sets of glyphs in the atlas by utilizing each color channel as a separate layer. In this case, rectangle packing is the process of fitting rectangles into a larger rectangle without any overlap, maximizing the area utilized [14]. One such algorithm utilizes the *skyline bottom-left* heuristic. As this algorithm is greedy by nature, it may be used without rearranging already placed rectangles.

Unity supports fonts in the TrueType and OpenType formats. Imported font assets may be configured for a certain pixel size, and are internally converted to a texture. Individual characters are then rendered as textured quadrilaterals.

Godot Engine provides two font asset implementations. The first type, "BitmapFont", is an implementation of the BMFont[18] bitmap font definition format, essentially pre-generated texture atlases of glyphs and their metadata. The other type, "DynamicFont", just like Unity, relies on the rasterization of vector-based fonts for a given size and the rendering of textured quadrilaterals.

2.7 Audio

Sound effects and music are an integral part of basically any game. The game queries the engine to play a certain sound effect or track, while the game engine is expected to handle the streaming and playback lifetime. Advanced audio engines provide high-quality modifier effects, audio channel mixing, surround sound, and other features increasing immersivity [4][11].

Pulse-code modulation (PCM) is the standard method of digital audio encoding [4], usually with a given sample rate, typically storing the value of each sample in a scalar. However, audio files are usually not stored in raw PCM, as large amounts of memory

[18] Available at http://www.angelcode.com/products/bmfont/

and disk space are used. Compression greatly reduces the storage size and streaming bandwidth required. *Lossy compression* can be used with little perceptual quality loss, removing features undetectible by most humans [7].

A trade-off between playback performance and memory consumption has to be considered. Frequently used small audio files should reside in the system memory uncompressed to reduce the stress on the CPU.

Both Unity and Godot Engine support the import of audio in standard formats, such as PCM WAV, Ogg Vorbis, and MP3.

2.8 Summary

In this chapter, two existing well-known game engines have been explored. Based on observations and available literature, systems necessary for the development of 2D games have been picked and further researched. Some of the most important components include a resource manager, a game loop, a renderer, or an audio engine.

Chapter 3

Design

This chapter brings insight into individual design choices made during the creation of the game engine. The game engine of interest will be further referred to as "**PlutoEngine**".

3.1 Target platform

A game engine is not limited by the platform it runs on, as long as it can be compiled for said architecture and all required abstract systems have a proper underlying implementation. The number of required implementation may grow exponentially with the amount of systems and platforms supported, therefore only a subset of platforms and systems is selected.

For the above reasons, PlutoEngine is limited to being an exclusively desktop computer oriented game engine, primarily for 2D games. Combined with the right choice of underlying implementations, this reduces the number of combinations to *one*.

3.2 System abstraction

While game engines can be designed bottom-up and scaled up going forwards, this approach has several considerable trade-offs. Bottom-up design may initially be faster and achieve a working product in a shorter amount of time, however expanding platform support or adding new systems may require large amounts of refactoring due to implementation-specific integrations.

The opposite approach — top-down design — sacrifices early progress and development time for a more solid structure. The overall architecture is planned out first, breaking down entire systems into individual subsystems. Extra time has to be dedicated to the development of the infrastructure to make systems inter-operable.

PlutoEngine takes a hybrid approach, creating a minimal proof of concept first, and then slowly migrate to a modular architecture of abstract components. The four major systems discussed in this chapter may be seen in figure 3.1, these systems will be broken down into smaller components as the PlutoEngine project grows.

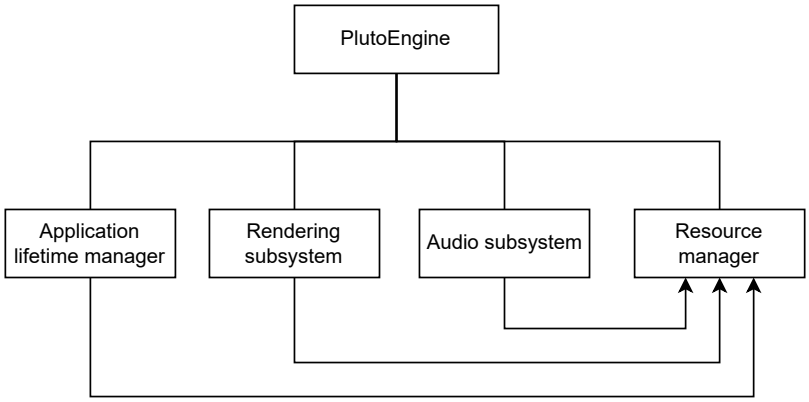


Figure 3.1: The four major systems of the initial iteration of PlutoEngine: application lifecycle manager, rendering subsystem, audio subsystem, and a resource manager.

3.3 Resource management

A platform-independent resource identifier format has been created, optionally becoming a Universal Resource Identifier (URI) by prepending the `pluto+asl` URI scheme. The engine can intercept these URIs, and normalize them by parsing their path part. Identifiers may be absolute or relative. Absolute identifiers comprise the owner module's name, resource container identifier, and the path of the resource. Relative identifiers only contain the path component.

Each resource identifier is internally converted to a path based on the native file system to maintain compatibility with the platform's standard library functions.

The resource system itself is not discussed here as its design heavily depends on the underlying platform and available APIs.

3.4 Application lifecycle and game loop

The lifecycle of the entire application, at the highest level of abstraction, can be consolidated into three main stages: initialization, the main loop, and the unload process [4]. Application lifecycle management is the highest-level system present in any game engine.

Before initializing all resources, all declared modules and their dependencies are initialized by the module manager. Each module may then request its own resources.

While the application is running, the game engine has to manage all resources without performance regressions. This includes rendering, audio streaming and input handling. During the game loop, individual systems are queried to perform routine tasks.
Repeating tasks the engine has to manage each update include :

1. Preparations to draw the next frame
2. Notifying the application of an update
3. Presenting the updated state of the game to the user
4. Clearing the previous states of active systems
5. Notifying all systems of an update

6. Checking for possible conditions of exiting the game loop

Once an exit is requested, the engine completes the in-progress update and begins unloading and freeing all active resources and modules. Once all resources are freed, the application may safely exit.

The entire lifecycle of an application is described by figure 3.2,

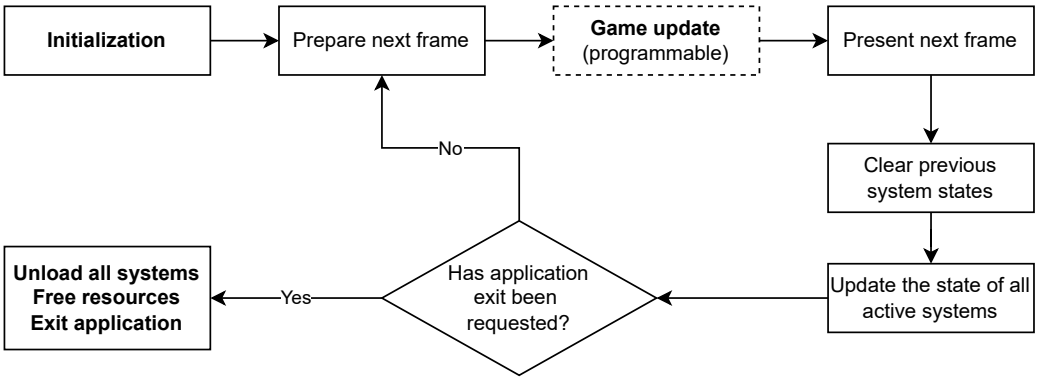


Figure 3.2: A simplified execution flowchart of an application. Primary system lifetime events are marked as bold.

The scene and game loop management subsystem has been omitted from the initial design of PlutoEngine, as it is not required for the development of games. Developers are expected to build a custom solution fit for their game. This trades off development time for higher flexibility. However, PlutoEngine provides basic tools for the implementation of a game loop, including a timer for the measurement of *frame deltas* [4].

3.5 2D renderer

There are various 2D rendering techniques. However, just like other major game engines, standard GPU-accelerated rasterization rendering techniques are used, applied in two dimensions. No depth-testing is used, sprites are assumed to be rendered back-to-front. As there is no integrated scene management, the ordering of sprite rendering is determined in an imperative manner by the game developer.

While 3D support is possible with this setup, PlutoEngine will focus exclusively on 2D rendering until all basic systems are in place.

3.6 Font renderer

A dedicated solution is required for the implementation of the font renderer, since a highly specialized solution may be necessary for this system to be efficient. All drawn glyphs are rectangles, and the atlas texture does not change between individual glyph draws, leaving room for major optimizations. For that reason, the font renderer is considered a separate system.

3.6.1 Bitmap font renderer

There are no special requirements for the bitmap font renderer, as bitmap fonts are usually used in smaller games or as placeholders. The process of rendering text using a bitmap font is very analogical to that of drawing sprites. A simple texture can be used as a backing atlas for glyphs.

3.6.2 Signed distance field font renderer

The inverse linear interpolation sampling technique has been opted for to reduce edge aliasing [3]. A small bias been introduced, slightly offsetting the thresholds for small sizes to make small text more legible.

3.7 Audio

As audio playback is a very platform-dependent and low-level function, a third-party solution providing an abstraction over the various system implementations will be used.

For the sake of terminology, let's call the data sources providing raw PCM data *streamers*. Two main streamer types were designed for the most common sound playback use cases: sound effects and music.

The first streamer type is *clip*. Audio clips are designed to hold short sound effects and allow random access and parallel read from an unlimited amount of sources. Due to their small size and random access requirements, they are stored as raw PCM in the system memory.

Tracks are longer audio streamers — possibly streamed over a network, decoded from their compressed format on-demand. They contain an integrated read pointer as only one track is expected to be streamed at a time. While random access is may be implemented, it comes at a very high overhead due to decoder seeking. Medium-length tracks can be stored in their compressed format in the system memory and decoded from memory.

A central audio engine component keeps track of active audio sources, copying data from their streamers in the game loop. Inactive sources or sources which have finished playing are automatically closed and all resources are freed.

3.8 Summary

An architecture for a 2D game engine has been designed as a set of components. Solutions of individual systems — selected in chapter 2 — to be implemented were proposed. This includes a component system, basic game loop management, a 2D renderer with support for sprite sheets, two font renderers, and an audio engine.

Chapter 4

Implementation

PlutoEngine is developed as a set of *Java* libraries written in the Java programming language with transitive dependencies as seen in figure 4.1. The *Java Virtual Machine* (further referred to as "the JVM") provides a solid foundation for developers to write multi-platform software without relying on the executing system's architecture. Each PlutoEngine library is a Jar (Java archive) file, which can then be added to the classpath of a project, loading it on the application's startup.

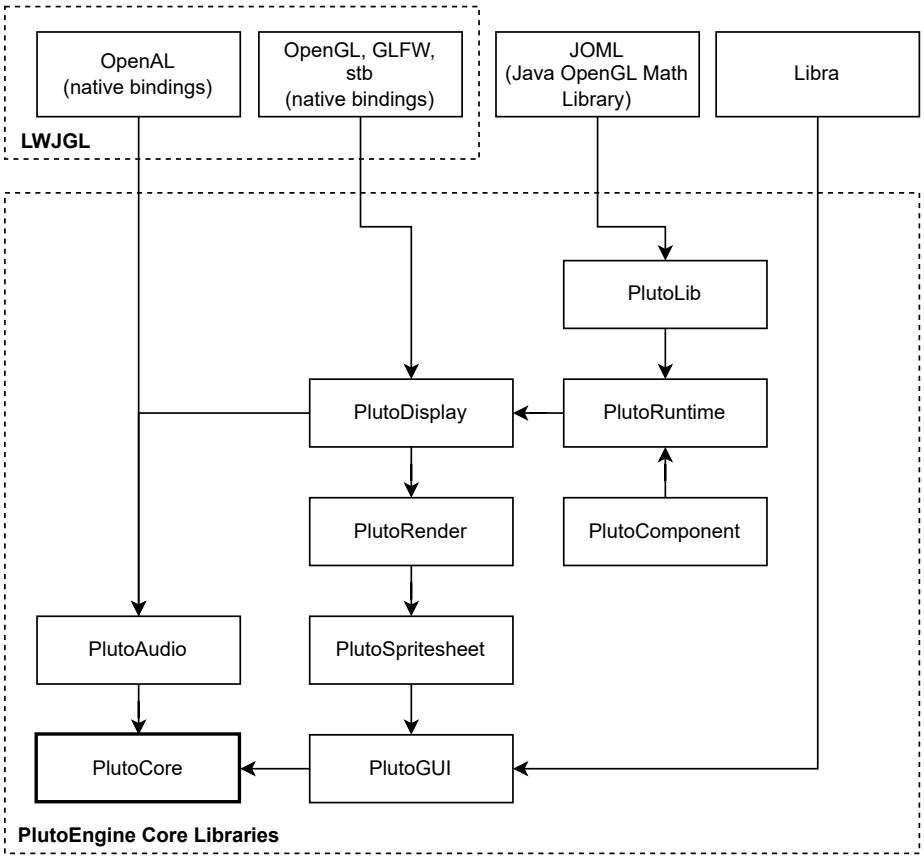


Figure 4.1: A simplified dependency graph of PlutoEngine. Note each dependency is transitive.

Directory	Description
buildSrc/	Build scripts and definitions
engine-core/	Core libraries needed for a minimal setup
engine-demo/	Demonstrative applications
engine-ext/	Libraries with optional features
libra/	GUI layout library submodule

Table 4.1: Repository structure layout

4.1 Build system and structure

Gradle has been chosen as the project's build system for its versatility, as each build script is written in either Groovy or Kotlin, languages that compile to Java.

An existing Git repository[1] on the source control platform GitHub has been used. Note that some parts of the engine were written before the work on this thesis has begun, notably some parts of the renderer and display subsystems and utility classes providing resource management and logging.

Directories for core libraries, demonstrative applications and extension features have been created, as seen in table 4.1. Each project is contained within a separate directory, managed by the central Gradle build script. See appendix A for a more in-depth look into the repository's structure and authorship.

4.2 Used technologies

The Java ecosystem contains many powerful libraries and projects, such as the IntelliJ Platform[2], licensed under permissive open source licenses. The license of *OpenJDK*[3], an open source implementation of the Java platform, allows it to be redistributed with finished products. The latest stable Java release, version 17, has been chosen for its language features and performance improvements in the OpenJDK implementation of the Java Development Kit (JDK) [4].

Many parts of the engine directly rely on the class introspection and reflection features of the Java language, especially the module loader and the component system.

One of the most popular programming interfaces designed for graphics rendering is *OpenGL* [12], allowing programmers to accelerate 3D graphics computation, leveraging dedicated hardware. OpenGL supports programmable shaders, written in the C-like GLSL [5] (OpenGL Shading Language) programming language.

OpenGL core profile in version 3.3 presents a decent compromise between modern rendering features and backwards compatibility with older graphics cards. GLSL shaders are written in version `330 core`.

[1] Available at https://github.com/493msi/plutoengine
[2] Available from https://github.com/JetBrains/intellij-community
[3] Available from https://openjdk.java.net/
[4] https://www.optaplanner.org/blog/2021/09/15/HowMuchFasterIsJava17.html, retrieved 2022-04-27.

4.3 Third-party libraries

Configuration and module definition files are loaded using the Jackson library set, which has a unified API to parse various data file formats like JSON, YAML or CSV[5].

By far the most important library — allowing PlutoEngine to use all natives libraries listed further — is *LWJGL*[6] (Lightweight Java Game Library), well-known for its use in the game Minecraft. LWJGL contains Java Native Interface bindings and wrappers for native libraries listed below.

The `stb`[7] public domain library set have been used. Audio files are stored in the Ogg Vorbis format and decoded using the `stb_vorbis` library, `stb_rect_pack` has been used as an implemention of the skyline bottom-left algorithm, and `stb_truetype` has been used for the parsing of the TrueType font format.

GLFW[8] has been used to create native windows and the OpenGL context, and handle window events, such as keyboard and mouse input.

Audio playback is provided by the OpenAL Soft[9] open source implementation of the OpenAL audio API.

While the Java standard library Abstract Window Toolkit and Graphics APIs contain a font loader and even a font renderer, the usage of the `stb_truetype` library has deemed sufficient and convenient for the generation of distance fields.

The majority of linear algebra computations on the CPU are handled by the *Java OpenGL Math Library*[10] (JOML).

4.4 Implementation details

The subsystems mentioned in the chapter 3 have been developed, notable implementations of which are listed in this section. Additional systems necessary for application lifetime management are included.

4.4.1 Subsystems

All individual systems can be retrieved from the global singleton instance of the engine as components. While making the global instance a thread-local multiton was experimented with, both GLFW and OpenGL are very sensitive about the threads they run on. Therefore, the feature of running an instance of the engine on a per-thread basis — while theoretically possible — was omitted for the time being. However, support for running multiple concurrent instances of the engine may prove useful further down the road in the implementation of a dedicated editor.

4.4.2 Components

The component manager is a data structure modeled around the provider software design pattern, while also resembling the entity component system (ECS) pattern. Components

[5] Available from https://github.com/FasterXML/jackson
[6] Available from https://www.lwjgl.org/
[7] Available from https://github.com/nothings/stb
[8] Available from https://github.com/glfw/glfw
[9] Available from https://openal-soft.org/
[10] Available from https://github.com/JOML-CI/JOML

(providers) can be retrieved by requesting their class or any of its superclasses, walking up to the abstract base component class in the class hierarchy tree.

Upon adding a component to the manager, it is registered as a provider and mounted, notifying the component of this event. Components can declare other components as dependencies during mounting.

Due to type erasure of type parameters in Java, components are created and kept track of using a token object with the same type parameter — a zero argument functor that creates an instance of said component. This type token can then be used to unmount all components created by that functor. The component manager also stores reverse references of components to the tokens that created them, so that removing components by supplying their instances is also possible.

The component system is fully reusable and may be used as a separate library, as well as in games running PlutoEngine.

4.4.3 Modules

The module loader is a component managing the lifetime of runtime-loaded modules in a stack-like manner. The module load and unload order is fully deterministic, modules always being unloaded in the reverse of the load order.

The component and module systems are orthogonal; in fact, the module loader itself is a component. While the module loader could have been implemented as a special case of the component manager, the provider behavior is not required. A separate solution thus reduces complexity.

It is critical that module dependencies represent a tree with no cycles as any dependency cycles introduced would have to be manually resolved by the authors of the modules. Forbidding dependency cycles therefore greatly reduces the complexity of the module system.

4.4.4 Resource management

The JVM uses a well-defined memory model, possibly greatly distinct from the underlying platform [9]. For raw data transfer with native function calls, Java's built-in array types are unfit, as Java uses exclusively big-endian for its integral types.

For the above reasons, the Java standard library provides the Buffer API, low-level containers for primitive data types [8]. These Buffers' endianness can be configured to use the native one. It is very important to distinguish off-heap (further referred to as "direct") Buffers from on-heap Buffers as per the JVM specification the garbage collector is free to relocate objects on the heap, and the heap is not required to be a continuous block of memory, while native functions require an exact pointer to a continuous sequence of values.

However, direct Buffer allocation using the Java standard library still presents significant overhead, as all values in a newly created Buffer are set to zero, and the lifetime of Buffers is managed by the JVM's garbage collector.

Due to the aforementioned issues, LWJGL provides an explicit memory management API[11], allowing for direct Buffer allocation via stack frames and native memory allocators, such as `malloc` from the C standard library, all integrated with the Java platform.

[11] As explained in https://blog.lwjgl.org/memory-management-in-lwjgl-3/, retrieved 2022-05-04.

4.4.5 Asset loading

Java's NIO API allows programmers to create custom file system providers[12]. In conjunction with SPI (Service Provider Interface), these providers can then be registered as URI handlers within the application. Whenever a new NIO `Path` is created from a URI, a `FileSystem` corresponding to the URI's scheme is resolved and the `Path` is constructed in an implementation-dependent manner.

Wrapping around other `FileSystem` implementations in an abstract way, the module system can seamlessly provide named resource containers with unified path accessors. Each module can declare its resource containers in a configuration file in the modules's root directory. These resource containers may be of any format, as long as a corresponding provider is installed, and the engine supports it. By default, support for Zip archives is included.

Therefore, all I/O methods in PlutoEngine accept NIO `Path`s, accepting both standard file system paths and PlutoEngine-specific resource paths.

4.4.6 Display

The display subsystem is primarily a wrapper around the GLFW native library, managing the lifetime of windows, OpenGL contexts, and user input — keyboard and mouse.

Double-buffering is used by default, meaning two screen buffers are always in flight, one being rendered to and one being presented[13]. Screen tearing may be prevented by enabling vertical synchronization.

4.4.7 Application lifetime management

Individual modules are loaded by mounting the module loader component to the global component manager and declaring an entry point. The depedency tree is walked recursively, loading each required module before loading the parent module itself.
The render loop consists mainly of the following routines:

1. Clear the main framebuffer
2. Run a single update of the game loop
3. Swap screen buffers
4. Ensure data buffers are streamed to all audio sources
5. Destroy unneeded objects
6. Poll for possible window and input events

In the final stage of the application's lifetime, modules and resources are unloaded in the reverse order of load. This ensures dependency trees are not broken. Once all modules are unloaded and all components are unmounted, the application can exit.

[12] Based on the guide available from https://docs.oracle.com/javase/8/docs/technotes/guides/io/fsp/filesystemprovider.html, retrieved 2022-04-28.

[13] Based on https://www.glfw.org/docs/latest/quick.html, retrieved 2022-04-29

4.4.8 2D renderer

Most features — such as textures, meshes (vertex array objects), shader programs — are thin wrappers around existing OpenGL concepts, however some parts take advantage of the Java programming language. Notably, the shader system exploits reflection to automatically link uniform locations based on their Java field names. Uniform fields may be annotated to automatically receive the projection matrix, should it change.

Image files are loaded using the ImageIO class from the Java standard library and subsequently flipped vertically, as the BufferedImage class assumes the Y coordinate increases going downwards, while OpenGL uses the bottom left corner as the origin point for UV coordinates. The color components are also swizzled using the `GL_TEXTURE_SWIZZLE_RGBA` texture parameter, since the BufferedImage is loaded in the 4-byte ABGR color component format and OpenGL expects the color components in the RGBA order.

The vertices of input meshes are transformed in the vertex shader to the corresponding clip-space coordinates [6]. The coordinate vector is first matrix-multiplied by the transformation matrix, the view matrix, and ultimately by the projection matrix. Therefore, given an input vector A, transformation matrix M, view matrix V, projection matrix P, the clip-space coordinates B of the vertex can be calculated as seen in formula 4.1.

$$B = P \times V \times M \times A \tag{4.1}$$

Because rendering in 2D usually does not require perspective projection, apart from use cases like parallax scrolling, the view frustum essentially takes shape of a rectangular prism, as seen in figure 4.2. This means the w component of the four-dimensional coordinate vector stays 1 and perspective divide has no effect [6]. The projection matrix does not change as long as the clip-space prism is not resized, which can be triggered by resizing the viewport.

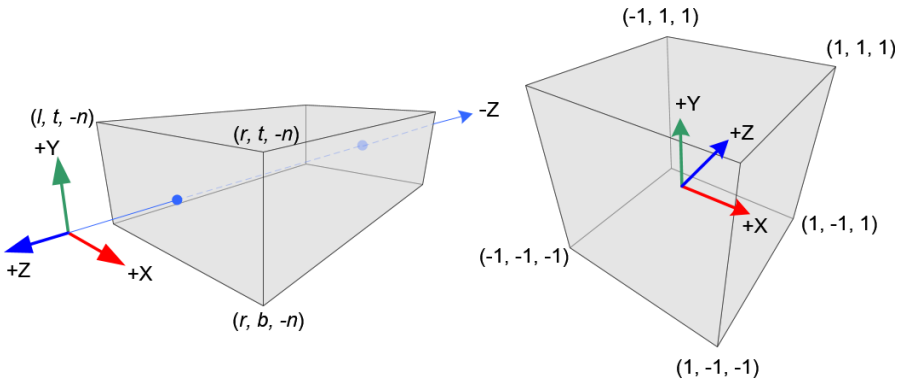


Figure 4.2: Orthographic projection volume and *normalized device coordinates* (NDC).[14]

For this use-case, the projection matrix can be defined as seen in formula 4.2 [6], where n is the distance of the near plane from the camera, f is the distance of the far place from the camera, l and r are the distances of left and right planes from the camera respectively, and t and b are the distances of the top and bottom planes from the camera.

[14] Figure sourced from https://www.songho.ca/opengl/gl_projectionmatrix.html, retrieved 2022-04-29.

$$P = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

PlutoEngine by default uses projection parameters to map the window coordinates directly to the view coordinates — setting the l and t parameters to 0, the b parameter to the window height, and the r parameter to the window width. This modifies the coordinate system to start in the upper left window corner, the positive x axis pointing to the right and flipping the y positive axis to point downwards.

When rendering without the use of a camera view, for example in user interfaces, the view matrix can safely be omitted entirely, as multiplying the model coordinates by an identity matrix would have no effect.

4.4.9 Sprite abstraction

Because sprites are extensively used in 2D games, special care must be taken for their implementation. As a subregion of a texture, sprites may be considered *views* of their parent texture. Based on that, sprites may substitute textures and UV coordinates when communicating with a renderer. Various sprite abstractions have been implemented — including animated sprites, oriented sprites, placeholder sprites, and sprites with managed lifetimes.

Framebuffers can be used to dynamically generate sprite sheets at application startup. By rendering individual sprites to an active framebuffer, the framebuffer's backing color attachment texture essentially becomes an atlas. While this solution is rather complicated, the atlas texture does not have to leave the graphics card's VRAM at any moment, as the process of blending the sprite into the sprite sheet is done on the GPU. The sprite sheet is automatically expanded when running out of space by creating a new texture and rendering the old texture into the larger one using framebuffer objects.

4.4.10 Font renderer

Metadata for the correct placement of individual glyphs is stored, such as its width, height, horizontal and vertical offset, and *kerning* offsets [4]. Some metadata is stored on a font-wide level, especially the font's name, ascent, descent, and line gap.

Originally developed as a part of the PlutoGUI module, a module named Libra has been separated from PlutoEngine to create an abstract API for building complex user interfaces. Currently, it serves as a base for the font renderer implementations contained within PlutoGUI.

A third-party library has been used for the process of parsing vector-defined fonts and distance field bitmap generation. For each character in a predefined set of characters, a distance field bitmap is generated, which is then blended onto an atlas texture. Rectangle packing is used to fit as many glyphs into a single atlas as possible.

As space may run out during the generation of a texture atlas and rectangle packing, a 2D texture array has been opted for. Once the rectangle packer cannot find a valid spot for a glyph rectangle, a new "page" is created, clearing the packing algorithm's state and writing the glyph image to the new texture. This approach is limited only by the maximum texture array size and hardware limitations.

Bitmap fonts are defined in a custom YAML-based configuration format, accompanied by an image file — the atlas containing all glyphs. TrueType fonts are loaded into a similar format using third-party libraries and dynamic texture atlas generation. This approach allows for combining a large part of the font rendering logic, emitting nearly identical draw commands for both approaches. A different GLSL fragment shader is required for the thresholding process of signed distance field rendering. The `smoothstep` GLSL function has been used for distance field sampling, ensuring smooth edges. This function is essentially equivalent to the inverse linear interpolation method, with an extra step of the *cubic Hermite interpolation* added. The cubic Hermite interpolation for any $x \in \langle 0, 1 \rangle$ may be defined as $3x^2 - 2x^3$ [13].

`RGBA` textures are used in the bitmap font renderer to allow developers to draw non-monochrome symbols aside from standard characters. Instead of using each color component as a separate page for the distance field font renderer to save video memory, a grayscale texture has been used for a similar effect.

As performing a large amount of draw calls — several OpenGL API calls per character — would significantly slow down the application, optimizations are necessary. Generated abstract draw calls from the font layout system are stored in an intermediate command buffer and optimized as seen in figure 4.3. These commands are then parsed and corresponding draw calls are emitted. Draw calls can be combined when rendering many glyphs using the same atlas. In particular, a mesh can be created on the CPU for the entire drawn string of text. This dramatically reduces the amount of issued GPU commands.

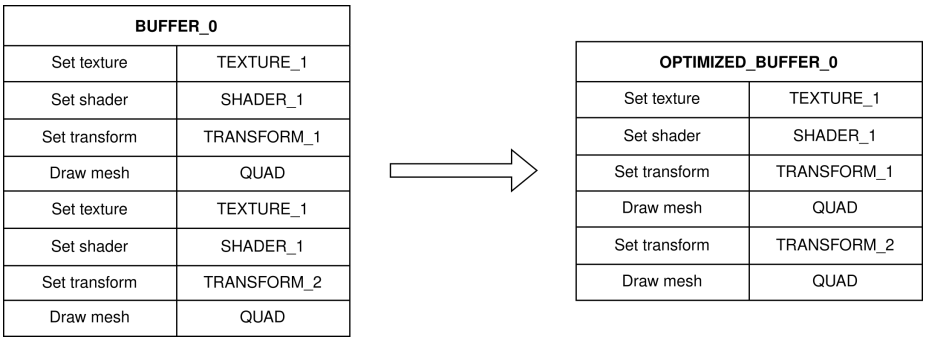


Figure 4.3: Command optimization process. Since the texture and shader did not change since the last draw call, these commands can be safely removed.

4.4.11 Audio subsystem

The Ogg Vorbis audio format has been selected for its industry support and having no licensing fees, as well as being an open standard. `stb_vorbis` is used to decode the Vorbis format to *interleaved* PCM. Interlaved formats store individual audio channel grouped by sample index in a pre-defined order [4]. While support for optional downmixing[15] to mono audio has been experimented with — using `stb_vorbis` — the downmixing algorithm used caused the sound to become audibly distorted.

The audio engine is implemented as two components: an abstraction around a native audio API, in this case OpenAL; and a manager, a subsystem that monitors the lifetime of individual audio sources and ensures data is always transferred to their copy buffers.

[15]The process of transforming audio with a certain number of channels into audio with a lower number of channels.

Just like with the cycle of transferring of images to the screen, the double-buffer strategy has been used to stream data to OpenAL sources. While one buffer is queued and streamed from, the second buffer may be written to in the meantime to minimize downtime.

Since the default orthographic projection used in PlutoEngine directly maps the view-space coordinates to the window coordinates, the movement of only 100 pixels on the screen translates into 100 units of world-space movement. Since OpenAL by default uses the reference distance of 1 unit[16], this causes the game sound effect volume to drop-off sharply. While this may be solved with the addition of a view matrix (camera), in order to allow games to use arbitrary coordinate scales, audio coordinate transform support was added. The audio engine has a configurable transformation matrix, set to identity by default, that may be modified to scale the audio coordinate system to desired levels. The OpenAL distance model may also be configured using the `AL_REFERENCE_DISTANCE` parameter on a per-source basis, however this solution was not implemented as its portability is not clear.

4.5 Engine distribution

PlutoEngine is automatically built using the continuous deployment solution GitHub Actions, a set of Jar file libraries (Maven artifacts) being the output. These artifacts are then published to a self-hosted Maven repository. The libraries can then be directly added to the classpath of a project or consumed using a build system, such as Maven or Gradle — see appendix B. The PlutoCore module contains a convenience entry point for applications to accelerate prototyping.

4.6 Summary

A game engine has been developed using the Java platform, implementing an OpenGL-based renderer and an OpenAL-based audio engine. The resulting engine provides a 2D renderer, two font renderer implementations, an audio engine, and a module-based resource system. Additional solutions have been implemented to manage the lifetime of individual subsystems. The project is publicly available and usable as a set of open-source libraries.

[16] As defined in the OpenAL 1.1 specification: https://www.openal.org/documentation/openal-1.1-specification.pdf, retrieved 2022-05-07

Chapter 5

Results and metrics

PlutoEngine was not developed in a vacuum. Therefore, it is important to evaluate the resulting implementation, as well as compare it to existing solutions. This section compares the implementation of PlutoEngine to the engines mentioned in chapter 2, and then evaluates the results of work done so far.

5.1 Comparison against existing solutions

One thing becomes almost instantly apparent: PlutoEngine lacks the distinction between GUI and world 2D rendering. This may be remedied by adding a new renderer with view matrix transformation support, combined with a camera subsystem.

The lack of an integrated scene management solution may be seen as both an advantage and a disadvantage. Leaving the game loop implementation at the discretion of the game developer enables them to build basically any kind of a 2D game at the cost of initial infrastructure development overhead. This design choice will have to be reevaluated as PlutoEngine matures, as it significantly raises the skill floor required to build a small game.

Many parts of PlutoEngine still contain no abstractions whatsoever, game developers have to rely on underlying graphics API implementations, especially lower-level framebuffer operations, such as adjusting the blend mode. Ideally, the underlying rendering API would not be exposed to game developers at all, turning each rendering implementation essentially module-private.

A lighting system is necessary to create more immersive games. While a lighting system may be created by the game developer, an integrated solution providing at least a basic implementation should make PlutoEngine more accessible to beginner developers.

The default font loader and renderer implementations in PlutoEngine allow for high resolution rendering of text at any size. Combined with various shader techniques and because texture coordinates are generated for each of the drawn characters (at the quadrilateral level), essentially any “paint” effect may be applied to the drawn text without the use of framebuffers or stencil buffers, as seen in figure 5.1. The fact a single-size atlas may be used for the rendering of text of any size is a great advantage over bitmap fonts in Unity and Godot Engine.

Audio support in PlutoEngine is currently very crude and only basic sound effect playback is included. Some features can be implemented on the behalf of the game developer, however some features — like music playback — should be included in the base package.

Figure 5.1: A screenshot of text drawn using the distance field-based font renderer, with a horizontal rainbow gradient applied. Note the shadow is not a shader effect — while technically possible — merely the same text drawn with a black paint offset on the vertical axis.

5.2 Third-party adoption

No third-party projects have been developed using PlutoEngine so far as the project is still in the rapid development stage of software development, being iterated upon constantly.

5.3 Demonstration and testing

A first-party basic space shooter game temporarily named "JSRClone", licensed under the open-source MIT License, has been developed to demonstrate the engine's capabilities. The game takes advantage of the sprite renderer, both the bitmap font renderer and TrueType font renderer, and the audio subsystem. As PlutoEngine does not provide a particle system yet, animations have been implemented as sprites managed by the game.

A short video demonstrating the features of the engine and gameplay of the game has been created[1].

Figure 5.2: A screenshot from the gameplay of JSRClone. Text rendered on the left side of the screen is drawn with the bitmap font renderer, while the frame-per-second counter leverages the distance field font renderer.

This game has been previously prototyped in C using the SDL2 library stack, however it is only used as a reference implementation in this case, being in no way related to PlutoEngine.

[1] See appendix C.

Special builds of both the original game "SRClone" and the PlutoEngine port JSRClone have been constructed to measure the *frame times* (time taken to process one frame) of each version of the game. It is important to note the implementations of both games are not identical and each implementation uses a different programming method. While SRClone is heavily data-oriented and procedural, JSRClone takes an object-oriented approach. Several shortcuts and compromises have been made while implementing the game logic, possibly impairing the performance in complex situations. That being said, this benchmark is a good indicator of any possible performance regressions, which could cause up to orders of magnitude lower performance.

JSRClone testing has been performed using the Eclipse Temurin[2] distribution of the OpenJDK implementation of the Java platform for both Windows and Linux in version `17.0.3+7`. Minimal Java runtime images have been created using the `jlink` command to remove all unnecessary features. The resulting size of a distribution for a single platform averages at around 60 MB.

The Windows 10 testing environment uses official proprietary drivers provided by AMD, while the Arch Linux environment uses the open-source Mesa[3] implementation. See appendix C for the full benchmark suite and software used.

All tests were carried out on a desktop computer with the AMD Ryzen 7 5700G CPU, 32 GB of system memory at 3200 MHz, and the AMD Radeon RX Vega 56 graphics card. Frame times in an internal pre-allocated array to avoid skewing the resulting data. The window resolution was always set to 1280×720 pixels. A single run of each of the tested implementations was performed.

The *reference benchmark scene* (figure 5.3) consists of 500 seconds of standard gameplay. Roughly $520-800$ sprites are rendered each frame. A static set of 512 sprites consists of the stars rendered in the background. The player consists of one sprite, a second one being dedicated to the ship's engine trail. Enemy ships approach the player in increasing but fluctuating numbers and shoot projectiles. A slight variation is present due to the random nature of enemy spawn rates. The same random seed could be used to fix this issue, however a different pseudo-random number generator implementation has been used for the SRClone implementation.

[2] Available at `https://adoptium.net/`
[3] `https://mesa3d.org/`

Figure 5.3: A screenshot from one of the more demanding periods of the benchmark build of the game JSRClone.

As seen in table 5.1 and figure 5.4, on Windows, SRClone averages approximately 2461 frames per second; JSRClone is closer to 1183 frames per second, while also fluctuating more in terms of frame times. This may be caused by the Java Virtual Machine (JVM) garbage collector (GC), differences in process scheduling, audio streaming, or insufficiently optimized paths of the game logic.

Implementation	**Frame time** (ms)		**FPS** $(1/s)$	
	Average	**Std Dev**	**Average**	**Std Dev**
JSRClone Windows	0.862	0.330	1183.01	92.823
SRClone Windows	0.407	0.021	2461.62	87.253
JSRClone Linux	0.406	0.052	2488.89	228.811
SRClone Linux	0.309	0.037	3261.32	257.367

Table 5.1: Average frame times and frames per second (FPS) and their standard deviations of the aforementioned standardized testing scene implementation runs.

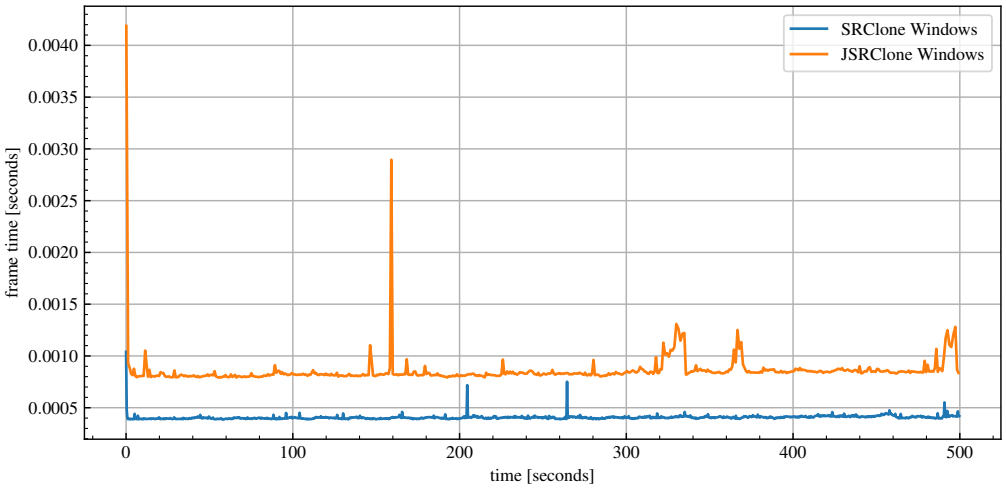


Figure 5.4: A comparison of frame times of Windows builds of both the Java version and C version. The Java build averages at around 0.85 milliseconds per frame, while the native build is closer to 0.4 milliseconds per frame.

JSRClone in the Linux testing environment (figure 5.5) achieves average frame rate of 2489 frames per second, while the native C version runs at approximately 3261 frames per second.
On Linux, the frame time gap closes significantly, suggesting one or multiple of the following:

- The SDL2 implementation of the game does not explicitly specify its rendering API and uses the first available accelerated renderer[4].
- The Windows AMD OpenGL driver implementation may have a significantly higher overhead compared to the open-source Mesa implementation.
- The Linux implementation of the OpenJDK used may be significantly faster.

To research this disparity, the Windows JSRClone build has been run using the Wine[5] compatibility layer in the Linux testing environment, experiencing much higher performance, approaching frames per second observed in the Linux build. For this reason, I have concluded that this performance difference is caused by the overhead of the OpenGL driver implementation of the Windows environment.

[4]The -1 renderer index has been supplied to the `SDL_CreateRenderer` initializer function (https://wiki.libsdl.org/SDL_CreateRenderer).
[5]Version 7.7; available at https://www.winehq.org/.

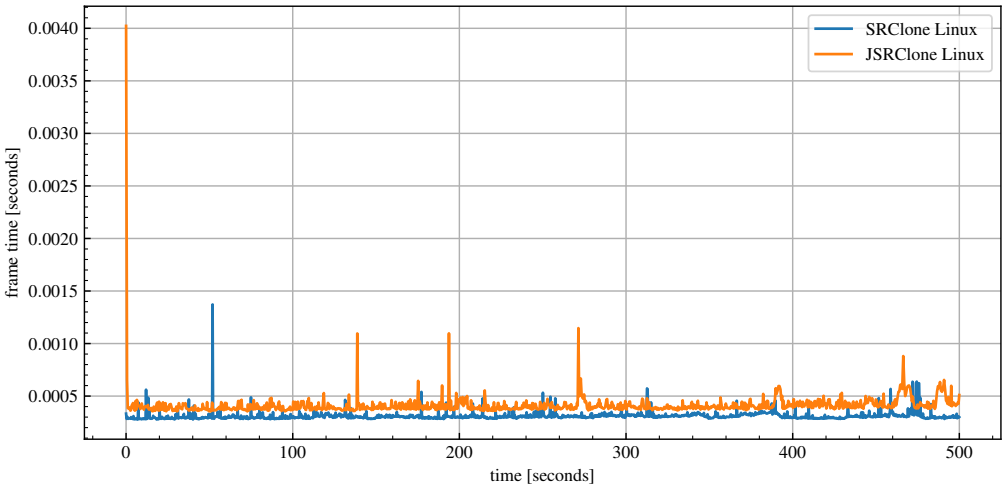


Figure 5.5: A comparison of frame times of Linux builds of both the Java version and C version. The Java build averages at around 0.4 milliseconds per frame while the native build is closer to 0.3 milliseconds per frame.

As seen in figures 5.6 and 5.7, during the first second of run time, the JVM is significantly slower as it warms up [15] and the just-in-time compiler optimizes hot parts of the code. However, high throughput is quickly achieved within the first second.

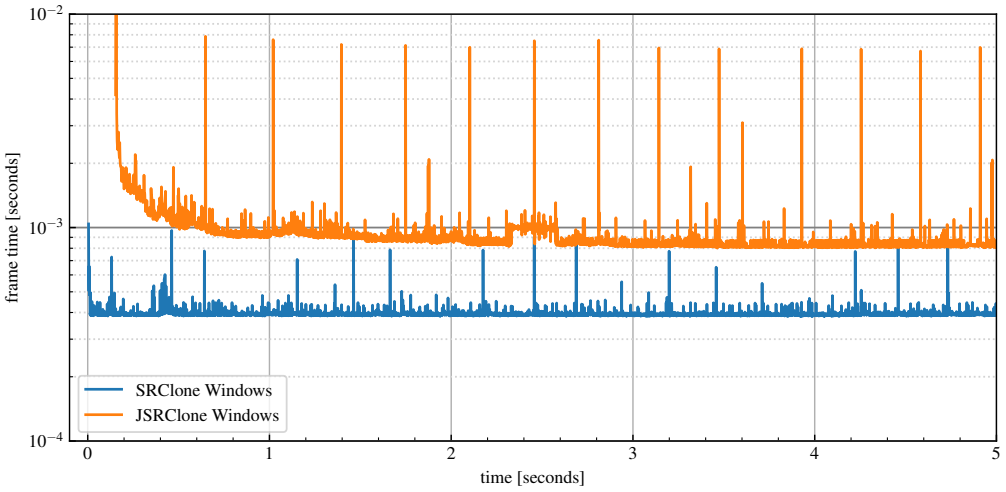


Figure 5.6: A close-up of startup frame times of Windows builds of both the Java version and C version.

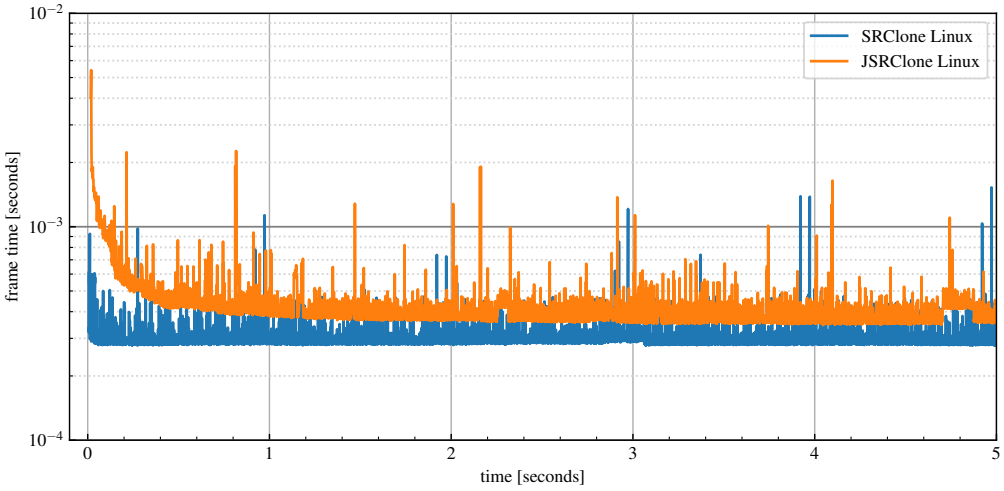


Figure 5.7: A close-up of startup frame times of Linux builds of both the Java version and C version.

5.4 Summary

PlutoEngine has been found to provide a solid and versatile base for 2D game development at the cost developer-friendly features, like a built-in editor or scene management. An example project — a simple 2D space shooter game — has been created under a permissive open-source license to promote PlutoEngine and demonstrate its features. This game has also served as a benchmark for performance testing. No significant slowdowns were observed compared to the native implementation of the same game.

Chapter 6

Conclusion

A modular 2D game engine has been designed based on the observations of existing solutions and available technologies. Such game engine named “PlutoEngine” has then been implemented in the Java programming langauge using OpenGL as its rendering API — some features are demonstrated in figure 6.1. The resulting product has then compared to previously observed game engines. PlutoEngine has been tested by creating a game (figure 6.2) demonstrating its features, providing comparable runtime performance to that of native applications with no significant slowdowns. Several design flaws have been identified, prompting for future work and research.

Figure 6.1: A demonstration of various features of the implemented engine — sprite rendering with recoloring and animations, module system, distance field font rendering, and bitmap font rendering.

In future the “engine as a library” model will be replaced with a full solution comprising an editor, an integrated development environment, and build tools. This concepts brings additional hurdles, being one of the major points of further research.

When using the module subsystem, developers have to explicitly manage the lifetime of all assets, slowing down development and introducing resource leak possibilities. A possibility of a low-level module governing the lifetime of assets may be investigated.

Figure 6.2: A 2D game developed using the implemented game engine.

PlutoEngine uses exclusively OpenGL as its graphics API and its usage is in no way abstracted, restricting this engine to platforms supporting OpenGL. While translation layers are an option, it is desirable to provide a universal abstract rendering frontend with interchangeable backends.

Currently, linear command buffers are used for the storage of abstract draw commands. However, for advanced rendering techniques, such as render targets, using acyclic graphs to model these advanced concepts may allow the optimizer to reorder draw commands allowing for more merges and therefore less graphics API calls. The concept of using abstract render commands for world rendering as well may be explored.

Bibliography

[1] Frisken, S. F., Perry, R. N., Rockwood, A. P. and Jones, T. R. Adaptively Sampled Distance Fields: A General Representation of Shape for Computer Graphics. In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. USA: ACM Press/Addison-Wesley Publishing Co., 2000, p. 249–254. SIGGRAPH '00. DOI: 10.1145/344779.344899. ISBN 1581132085. Available at: https://doi.org/10.1145/344779.344899.

[2] Garcia, F. E. and Almeida Neris, V. P. de. A Data-Driven Entity-Component Approach to Develop Universally Accessible Games. In: Stephanidis, C. and Antona, M., ed. *Universal Access in Human-Computer Interaction. Universal Access to Information and Knowledge*. Cham: Springer International Publishing, 2014, p. 537–548. ISBN 978-3-319-07440-5.

[3] Green, C. Improved Alpha-Tested Magnification for Vector Textures and Special Effects. In: *ACM SIGGRAPH 2007 Courses*. New York, NY, USA: Association for Computing Machinery, 2007, p. 9–18. SIGGRAPH '07. DOI: 10.1145/1281500.1281665. ISBN 9781450318235. Available at: https://doi.org/10.1145/1281500.1281665.

[4] Gregory, J. *Game Engine Architecture, Third Edition*. 3rd ed. A K Peters/CRC Press, 2018. ISBN 978-131-5267-845.

[5] Kessenich, J., Baldwin, D. and Rost, R. *The OpenGL® Shading Language, Version 4.60.7* [online]. 2019-07-10 [cit. 2022-04-29]. Available at: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf.

[6] Lengyel, E. *Mathematics for 3D game programming and computer graphics*. 3rd ed. Course Technology, 2012. ISBN 978-143-5458-864.

[7] Marzen, S. E. and DeDeo, S. The evolution of lossy compression. *Journal of The Royal Society Interface*. 2017, vol. 14, no. 130, p. 20170166. DOI: 10.1098/rsif.2017.0166. Available at: https://royalsocietypublishing.org/doi/abs/10.1098/rsif.2017.0166.

[8] Oracle Corporation. *Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification* [online]. 2021 [cit. 2022-04-29]. Available at: https://docs.oracle.com/en/java/javase/17/docs/api/index.html.

[9] Oracle Corporation. *The Structure of the Java Virtual Machine* [online]. 2021 [cit. 2022-04-29]. Available at: https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-2.html.

[10] Recker, J. L., Beretta, G. B. and Lin, I.-J. Font rendering on a GPU-based raster image processor. In: Eschbach, R., Marcu, G. G., Tominaga, S. and Rizzi, A., ed. *Color Imaging XV: Displaying, Processing, Hardcopy, and Applications*. SPIE, 2010, vol. 7528, p. 85 – 99. DOI: 10.1117/12.839486. Available at: https://doi.org/10.1117/12.839486.

[11] Schissler, C., Nicholls, A. and Mehra, R. Efficient HRTF-based Spatial Audio for Area and Volumetric Sources. *IEEE Transactions on Visualization and Computer Graphics*. 2016, vol. 22, no. 4, p. 1356–1366. DOI: 10.1109/TVCG.2016.2518134.

[12] Segal, M. and Akeley, K. *The OpenGL® Graphics System: A Specification* [online]. 2019-10-22 [cit. 2022-04-29]. Available at: https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf.

[13] Spitzbart, A. A Generalization of Hermite's Interpolation Formula. *The American Mathematical Monthly*. Mathematical Association of America. 1960, vol. 67, no. 1, p. 42–46. ISSN 00029890, 19300972. Available at: http://www.jstor.org/stable/2308924.

[14] Wei, L., Oon, W.-C., Zhu, W. and Lim, A. A skyline heuristic for the 2D rectangular packing and strip packing problems. *European Journal of Operational Research*. 2011, vol. 215, no. 2, p. 337–346. DOI: https://doi.org/10.1016/j.ejor.2011.06.022. ISSN 0377-2217. Available at: https://www.sciencedirect.com/science/article/pii/S0377221711005510.

[15] Westrelin, R. *How the JIT compiler boosts Java performance in OpenJDK* [online]. 2021-06-23 [cit. 2022-04-29]. Available at: https://developers.redhat.com/articles/2021/06/23/how-jit-compiler-boosts-java-performance-openjdk.

Appendices

List of Appendices

Appendix A

Repository structure

The engine codebase is separated into multiple sub-trees based on the purpose of contained projects. `engine-core`, `engine-demo` and `engine-ext` are metaprojects with the sole purpose of combining build logic of their subprojects.

All code in the repository, with the exception of autogenerated Gradle wrapper files[1] and parts of the `libra` submodule[2], is authored by me and I claim ownership of it.

The repository contains third-party media, such as audio tracks, licensed as specified in the `LICENSING_INFO.txt` copyright notice in the root directory.
See figure A.1 for a full overview of the repository tree.

[1] Files within the `gradle/wrapper/` directory, the `gradle` and `gradlew.bat` executables. Not distributed with finished products.

[2] Co-authorship of the Gradle build script, which is not related to the content of this thesis.

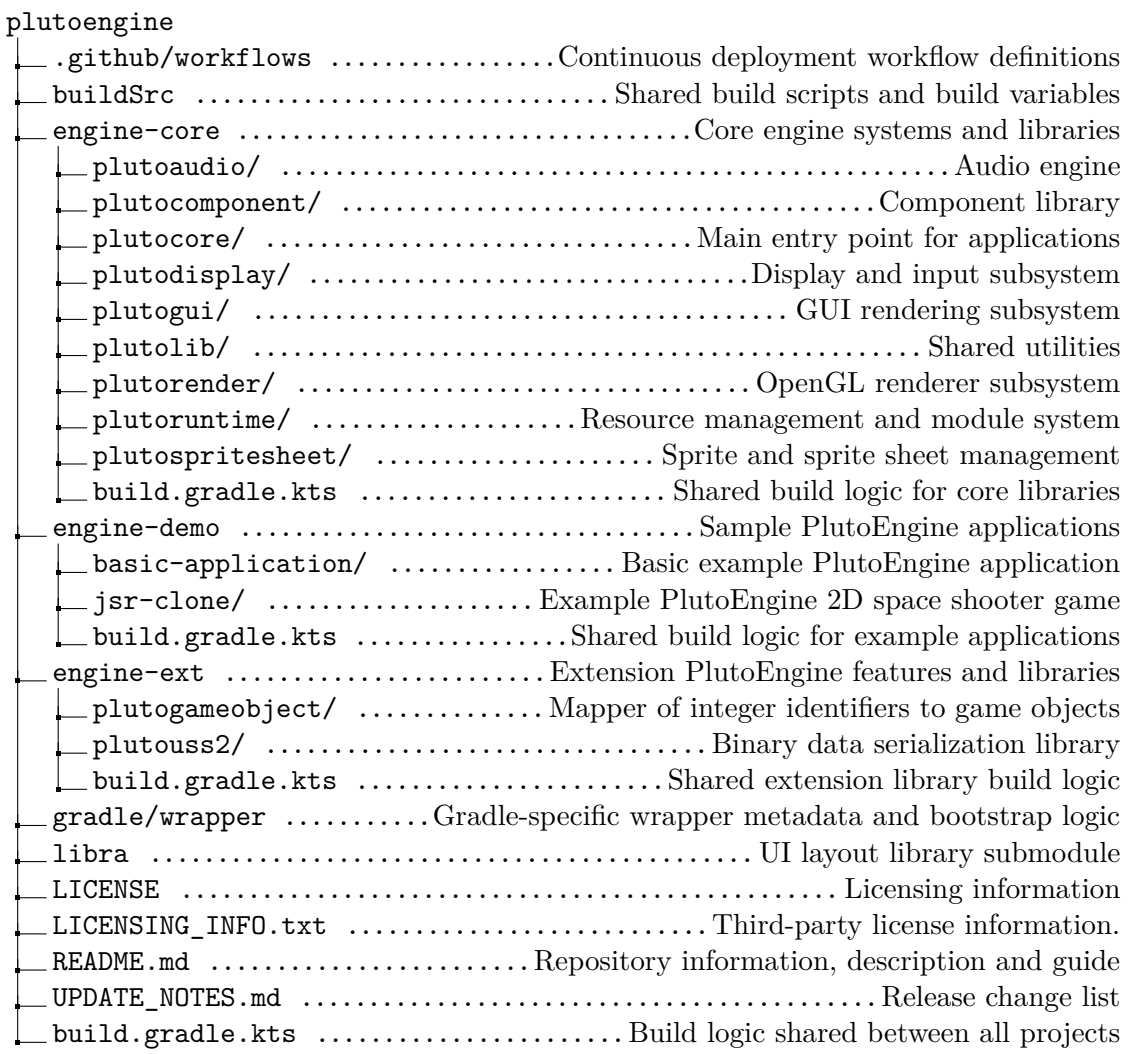


Figure A.1: Complete PlutoEngine repository directory tree with short explanations for each of the files and directories

Appendix B

Engine usage

The PlutoEngine build system uses a wrapper script (`gradlew`) to handle the installation of Gradle, having a Java Development Kit of version 17 or higher is necessary to compile and run the project. All further mentioned commands assume running `./gradlew` for Unix-based operating systems and `.\gradlew.bat` for Windows-based operating systems in place of `./gradlew`.

B.1 Building

The PlutoEngine project and all subprojects can be built running the following:

```
$ ./gradlew build
```

B.2 Running demos

Demonstrative applications are located in the `engine-demo` directory, individual applications may be run with `gradlew :plutoengine-sdk:engine-demo:demoName:run`, where `demoName` is a valid subproject of `:plutoengine-demos`. It is important to set the working directory to the root directory of said demo application to load module assets properly.
For example, the `jsr-clone` game may be run the following:

```
$ cd engine-demo/jsr-clone
$ ../../gradlew :plutoengine-demos:jsr-clone:run
```

B.3 Publishing libraries

Maven build artifacts produced by Gradle may be published to a Maven repository, however it is necessary to modify the target Maven repositories as specified in the `build.gradle.kts` script located in `engine-core/` and `engine-ext/`.
Once the project is configured, building and publishing may be triggered by running:

```
$ ./gradlew :plutoengine:publish -x test
$ ./gradlew :plutoengine-ext:publish -x test
```

engineUsage

B.4 Linking libraries

In order to make the process of adding PlutoEngine libraries and all their dependencies to a project easier for developers, PlutoEngine releases are hosted on a publicly accessible Maven repository.
For example, adding PlutoEngine as a dependency to a Kotlin-based Gradle build script is done as follows:

```
repositories {
    // ...

    maven {
        name = "Vega"
        url = uri("https://vega.botdiril.com/")
    }
}

dependencies {
    // ...

    implementation("org.plutoengine", "plutocore", "22.2.0.0-alpha.2")
}
```

Appendix C

Attached media

The media attached to this thesis contains the following data:

- `benchmark-suite.zip` — The full testing kit for performance testing.
  Available online from: https://get.assets.tefek.cz/benchmark-suite.zip
  SHA-256: e7d77caaf748a46404c86eb69462fdb07962d025997a3ec04915dde33591f96d
- `plutoengine.zip` — The final implementation of the game engine.
  Available online from: https://get.assets.tefek.cz/plutoengine.zip
  SHA-256: a0de7e0ca64db0005c42c29dd6c37020ced934f75955e7336d0635f506a3e2bd
- `plutoengine-docs/` — The Javadoc for this version of the game engine.
- `jsr-clone-final.zip` — The final implementation of the testing game.
  Available online from: https://get.assets.tefek.cz/jsr-clone-final.zip
  SHA-256: 71eada0fbce40706191ebf8e8adea81f3e5852c92f0e509e1261514fbf71dbe0
- `thesis-src/` — The LaTeX source files of this thesis and all media used.
- `xstefa25-bt.pdf` — This thesis in the PDF format.
- `demo.mp4` — A short video demonstrating the created game.