



The Cooper Engine

TU856
BSc in Computer Science

Aidan Dowling

C20394933

Bryan Duggan

School of Computer Science
Technological University, Dublin

11/04/2024

Abstract

I decided to create a game engine for my final year project, with a specific focus on leveraging the Rust programming language. I chose this task as I had no experience with the process of engine development, and as a learning exercise. As the landscape of video game technology continues to evolve, the demand for robust, performant, and secure game engines is paramount. Rust, known for its emphasis on memory safety, zero-cost abstractions, and concurrency without data races, emerges as a promising language for this purpose. The research conducted in this thesis spans multiple dimensions, from the conceptualization of a Rust-based game engine to the intricate details of prototype development.

The initial chapters provide a contextualization of Rust in the context of game development, highlighting its strengths and advantages over traditional languages. The thesis then delves into the pivotal role of prototyping in the iterative development process, emphasizing the need for experimentation, testing, and refinement. The stages of prototype development are meticulously examined, from the conceptualization phase to the documentation of a functional Rust game engine.

The prototype's core responsibilities, including scene graph management, rendering pipeline control, shader management, and basic update loop structures, are thoroughly investigated. This exploration encompasses a broad spectrum of topics, such as material and texture handling, lighting, and shadowing and camera management, offering insights into the complexities of rendering engines in game development.

In conclusion, this paper contributes to the growing body of knowledge in game engine development by providing a detailed examination of the prototype development process for a Rust-based game engine. The findings presented herein lay a foundation for future advancements in game engine technology, fostering innovation and pushing the boundaries of what is achievable in the realm of interactive digital entertainment.

Declaration

I hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:

Aidan Dowling
Aidan Dowling

Date

Acknowledgements

A huge thank you to Bryan Duggan for his guidance and insight in the development of this project. Without his help I would not have gotten anywhere near as far as I have.

A thank you to my family for providing support during the project's development.

Table of Contents

1. Introduction	1
1.1. Project Background	1
1.2. Project Description	1
1.3. Project Aims and Objectives	1
1.4. Project Scope	2
1.5. Thesis Roadmap	2
2. Literature Review	4
2.1. Introduction	4
2.2. Existing Solutions	4
Software Examined:	4
2.3. Technologies Researched	10
Java	11
Go	11
Zig	12
C++	13
Rust	14
Relevancy of these metrics	15
Graphics Libraries	16
2.4. Other Research	18
wgpu	18
specs	18
Legion	18
Doom	19
2.5. Existing Final Year Projects	19
SOL Engine	19
Deb Engine	20
2.6. Conclusions	20
3. System Design	21
3.1. Introduction	21
3.2. Software Methodology	23
3.3. Overview of System	25
1. Cooper Engine:	25
2. Lynch :	26
3. Frost:	33

4. Collaboration:.....	39
5. Extensibility and Customization:.....	40
3.4. Conclusions	40
4. Testing and Evaluation Plan	40
4.1. Introduction.....	40
4.2. Plan for Evaluation.....	41
4.3. Conclusions.....	41
5. Prototype Development	42
5.1. Introduction.....	42
5.2. Prototype Development	42
5.2.1 Window creation	42
5.2.2 Vulkan integration	44
5.2.3 Graph Structure, Creation and Building a render pass.	45
5.2.4 Main program and event loop	49
5.2.5 Event Loop and user functions.....	50
6. Software Development	54
Cooper.....	54
Lynch	56
Vulkan Device.....	56
Vulkan Context.....	58
Render Pass.....	58
Render Graph.....	59
Descriptor Set.....	59
Pipeline Descriptions.....	60
Buffers.....	61
Images	63
Textures	64
The View Uniform	66
GPU Vertex.....	67
GPU Materials	68
GPU Mesh	69
Renderer Internals	69
The Actual Renderer	71
Render Passes	71
The Camera	79

Frost	80
Entities	80
Components.....	81
Component Stores	82
Archetypes	82
ComponentPack (the glue which holds archetypes, entities, and components together)	83
Searching the World for Entities with Components	84
Macro Utilization.....	85
Systems	87
Rigid Body	89
Orientated Bounding Box (OBB)	89
Collision System	91
Physics System	92
Integration Function	93
The UI	94
7. Testing and Evaluation	95
5.1. Introduction	95
System Testing	96
Benchmarking	98
Lynch	98
Frost	100
Evaluation	106
The Application Layer.....	106
Lynch (Renderer).....	106
Frost (State Management).....	107
8. Issues and Future Work	108
8.1. Introduction.....	108
8.2. Issues and Risks	108
8.3. Plans and Future Work	108
Conclusion.....	108
Bibliography	113

1. Introduction

1.1. Project Background

The thought of game design and further game engine design has always been a driving factor behind learning how to program. The optimizations and unique programming style and flow of game engines poses numerous challenges not typically seen anywhere else within the world of programming. The effects these well-made technologies can have on people's individual experiences can shape their lives and imbue memories which can stick with people their entire lives. To create a system like that is to create a system which can provide enjoyment and memories to those who experience it. This is, in my mind, quite a unique aspect of computer science which is not thoroughly explored enough within my degree and without my experiences programming.

Throughout my years in college, I spent a lot of time within projects analysing and then optimizing code to a gratuitous extent. I created a very basic game engine built upon the Processing engine which could do some basic physics calculations, could handle collisions, and audio playback without a significant effect on performance.

According to the latest data, there are approximately [3.09 billion active video gamers](#) worldwide. Most of these people are playing games on mobile devices (which is not of particular interest to me). One of the largest forms of consumed media in the world means there are many aspects of interest to display in this field.

1.2. Project Description

My project will provide a code-only platform for game designers to utilize Rust and Vulkan to create performant cross-operating system multithreaded games, with features to access physics, and rendering systems and a method to implement new features using the Rust crate system. The focus on this is to

1.3. Project Aims and Objectives

The overall aim of this project is to create a technical demonstration of systems within the engine which show efficient and diverse methods of creating a game.

The engine will be entirely written in Rust, will be scripted in Rust and will use the Vulkan API.

The code base will use Rust best practices, following the standards set within the "idiomatic-rust" standards, and minimizing the disabling of compiler warnings. (1)

The game engine will exist and be editable through code. No features / implementations will be hidden from the developer.

Only libraries which are used and needed by the developer should be imported, such as the renderer, or resource management.

The renderer should support large amounts of entities being rendered, as well as a list of optional rendering strategies which can be configured by the engine user.

Objects will be manageable through an ECS (Entity Component System) structure.

The engine will support Physics and Collisions.

Entities in the game should be capable of interacting with each other, and the physics system should be as determinable as possible.

The state of entities and their components will be serializable to JSON and deserializable into the ECS.

3D models should be importable, with their shaders and textures being customizable.

1.4. Project Scope

The project covers the process of creating the rendering engine, the entity component system, and the connection of these processes. Aspects such as animation, a graphical editor, audio processing, networking, and artificial intelligence will not be included in this project's scope. While in the future I plan for these features to be possible, and hope that the design of the engine makes the addition of these components not only possible, but intuitive and simple.

1.5. Thesis Roadmap

This section will explain what information is contained in each chapter of the interim report:

1. Introduction
 - a. This chapter explains what the project encompasses and its overall goals. It will assess the scope of the project and what the final application should include.
2. Literature Review
 - a. This chapter encompasses research information that is relevant to producing this project that is understanding of other solutions and ideas that have already been produced. I also have systems for implementing these features.
3. System Design

- a. This section will examine the design and implementation of different areas of the project. It will include diagrams to explain the connectivity of the app and how components will interact with one another.
- 4. Testing and Evaluation
 - a. This section is where I will conduct my user and unit testing for the application. It will examine and evaluate different testing methods and analyse the most optimal testing environments for the project.
- 5. Prototype Development
 - a. This chapter will provide an explanation of where the current state of the project is at. It will include coding snippets and screenshots of the current working version of the project and explain in depth how the complexity of the project is being achieved.
- 6. Issues and Future Work
 - a. This section will give an overview of the risks that have been identified for the project. It will also outline the work that is still to be completed to achieve the end goal of the project.

2. Literature Review

2.1. Introduction

In this chapter I cover the different aspects of game engine development and my research into those fields. Deciding on language choices, project structures and game state management requires looking at many different parts of literature and design, taking and leaving parts which are useful for the project at hand.

2.2. Existing Solutions

Software Examined:

Unity



Figure 1, Unity Logo

Unity is a prominent game development platform, widely used due to its user-friendly interface and flexibility. A key feature is the Unity Editor, which simplifies the game development process with its drag-and-drop functionality and straightforward scripting capabilities. This aspect of Unity, emphasizing ease of use, is essential for developers who are looking to streamline their workflow.



Figure 2, Unity Editor

In terms of graphics, Unity supports both 2D and 3D, catering to a diverse range of game genres. Its rendering engine can produce high-quality visuals, which is a significant factor in its broad appeal. Another major advantage of Unity is its cross-platform compatibility, allowing for the deployment of games across various platforms, including PC, consoles, mobile devices, and VR/AR systems. (2)

The platform's asset store offers an array of resources like models, textures, and scripts. This not only enhances the development process but also provides a rich library for developers to draw from, saving time and effort in game creation.

Unity's extensive community contributes to its popularity, offering a robust support network and a wealth of shared knowledge. This community-driven environment, along with comprehensive documentation, makes Unity accessible to a wide range of developers, from beginners to experts.

In the context of developing a new game engine, studying Unity is highly relevant. It serves as an exemplary model in the industry, demonstrating successful integration of user-friendly design and versatile functionality. Examining Unity's architecture and the user experience it provides can offer valuable insights for anyone looking to create a new game engine. This analysis can help identify what makes a game engine successful in terms of usability, performance, and flexibility.

Understanding Unity's approach to these areas can guide the development of a new engine that addresses current market demands while potentially introducing new features or improvements.

Where it succeeds:

1. **Cross-Platform Development:** Unity excels in supporting multiple platforms, making it suitable for developing games for PC, consoles, mobile devices, AR/VR, and the web.
2. **Ease of Use:** Unity's user-friendly interface and the availability of a vast asset store facilitate rapid development, particularly for indie developers or those with limited experience.
3. **Community and Documentation:** Unity has a large and active community, providing extensive documentation, tutorials, and support. The Asset Store also offers a wide range of pre-built assets.
4. **Versatility:** Unity is flexible, accommodating both 2D and 3D game development. Its component-based architecture simplifies object behaviour customization.
5. **Scripting Languages:** Developers can use C# or UnityScript for scripting, providing options for both experienced programmers and those new to coding.

Features:

- 2D and 3D rendering system.
- Comprehensive Physics System

- ECS with event driven functions.

ECS (Entity-Component-System):

- Unity traditionally follows an Object-Oriented Programming (OOP) approach rather than a strict ECS architecture. However, it introduced the Unity ECS framework, DOTS (Data-Oriented Technology Stack), which allows developers to utilize ECS principles for performance optimization.
- ECS in Unity focuses on transforming game object data in a way that is more cache-friendly, improving performance for large-scale simulations.
- Unity's ECS can be beneficial for handling complex game object lifecycles and achieving high performance.

Game Object Lifecycles:

- Unity's GameObjects have predefined lifecycles with methods like Start, Update, and OnDestroy, providing control over initialization, continuous updates, and cleanup.
- Unity's ECS provides a more data-driven approach, separating the concerns of data and behaviour for efficient processing.
- Developers using Unity need to manage the lifecycles of MonoBehaviour components attached to GameObjects for more traditional object-oriented workflows.

Many of Unity's features were important to me, and its massive impact on the game industry at large could not be forgotten. It's out of the box renderer and physics / entity system is incredibly diverse and provides many features which make games made with unity diverse and distinctive.

Godot



Figure 3

Godot, an open-source game engine, has garnered significant attention and popularity in the game development community, primarily for its flexibility and user-friendly design. Unlike many proprietary game engines, Godot is freely available and offers a comprehensive suite of tools that cater to both 2D and 3D game development. One of its standout features is the scene system, which simplifies the organization and management of game elements. Godot's scripting language, GDScript, is tailored for game development, offering an easy learning curve for beginners while still being powerful enough for advanced users. The engine's design philosophy emphasizes ease of use and accessibility, making it an attractive option for indie developers and educators.(3)



Figure 4

Areas Where Godot Succeeds:

1. Cross-Platform: Godot supports multiple platforms, including Windows, macOS, Linux, HTML5, and various mobile platforms.
2. Integrated Development Environment (IDE): Comes with a feature-rich IDE that simplifies the development process.
3. Flexible Scene System: Its unique scene system allows for efficient organization and reuse of game components.
4. GDScript: A Python-like scripting language designed specifically for Godot, making it approachable for beginners.
5. Customizable Workflow: The engine's interface and functionality can be customized to suit different development needs.
6. Active Community: A vibrant community that provides support, tutorials, and plugins.

Key Features of Godot:

- **Node-based Architecture:** Everything in Godot is a node, which can be combined in various ways to create more complex systems.
- **Animation Tools:** Offers a robust set of animation tools, allowing for intricate animation sequences and effects.
- **Live Editing:** The live editing feature lets developers test and edit games in real-time on their target device.
- **Visual Shader Editor:** For those who prefer a more visual approach, Godot includes a visual editor for creating shaders.
- **Multi-Window Support:** Allows developers to organize their workflow across multiple monitors.
- **Signal System:** An event system that helps in creating complex game logic without tightly coupling code components.

Godot has made waves recently due to how lightweight the implementation is, the fact that the project itself is open source and the scene graph implementation of resource management.

Source Engine



Figure 5

The source engine is one of the main inspirations to me to begin programming in the first place. Much of my childhood, adolescence and adulthood has been spent playing games on this engine, on my low-spec home laptop. The speed and accessibility of the engine allowed me to experience this game on much lower end hardware. Designed by Valve, the engine has notoriously been the forefront of game development. The source engine is an amalgamation of very simple and basic design systems colliding and combining in a unique way, which gives source games a distinctive “weight” in their movement, physics, lighting, and audio systems that I have experienced with no other engine.

Compared to other more traditional engines, Source does not have an editor, instead using the level editor Hammer for most of the workload. The Source SDK can be used to extend the engine.

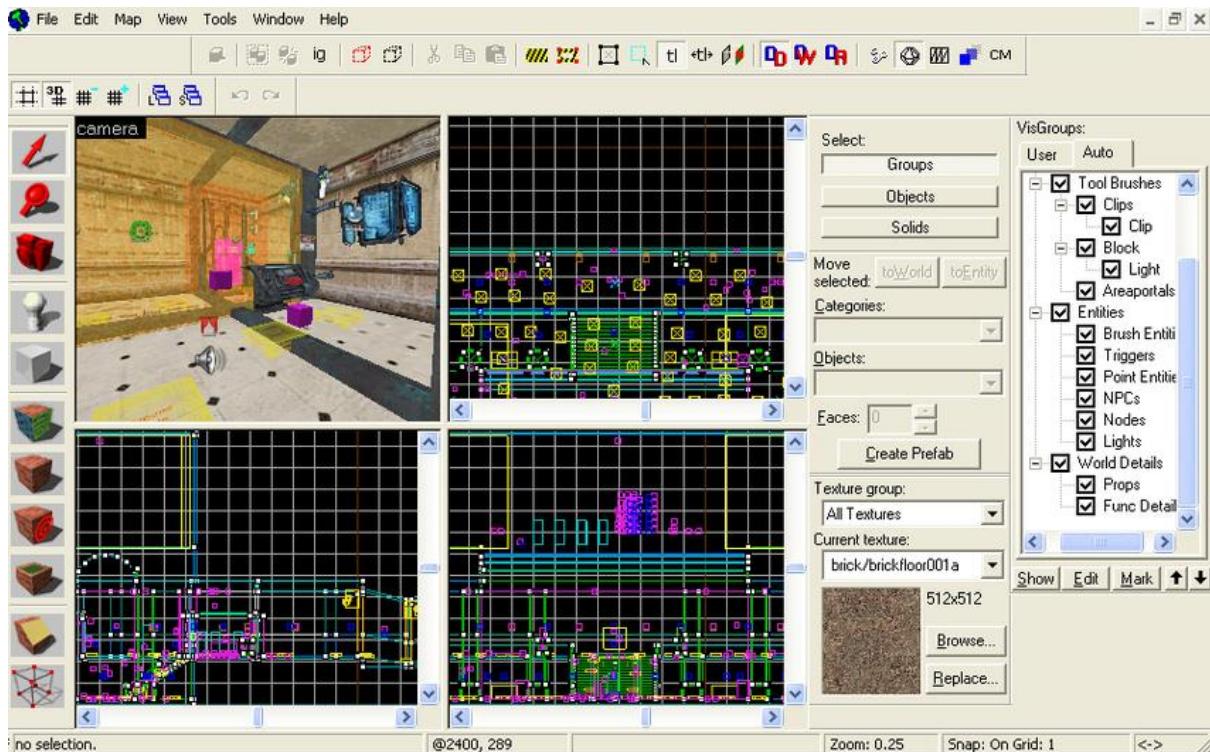


Figure 6

Where it succeeds:

1. Optimization: The Source Engine is known for its efficiency and optimization, allowing for smooth performance even on less powerful hardware.
2. Modularity: Designed with modularity in mind, making it a popular choice for user-generated content and modifications. Garry's Mod is a famous example of an incredibly extensive game with sustained long-term support.
3. Physics Engine: Source includes a capable physics engine, offering realistic interactions and simulations.
4. Longevity: Source Engine has a proven track record of being used in critically acclaimed games over an extended period.
5. Customizable: Source is highly customizable, allowing developers to tailor the engine to their specific needs.

Features:

- Hammer Editor: Source's level design tool for creating complex environments.
- Entity System: Supports entity-component architecture for flexible game object composition.
- Scripting: Utilizes the Source SDK for scripting and modification.

- Networking: Implements a robust networking system for multiplayer games.
- Sound System: Equipped with a sophisticated sound system for immersive audio experiences.

ECS:

- Source Engine is known for its unique entity-component architecture, predating the popularization of ECS as a term.
- It features entities that are composed of components, allowing for a modular and flexible design.
- Source Engine's system shares similarities with ECS but predates the formalization of ECS in contemporary game development.

Game Object Lifecycles:

- Entities in Source Engine have lifecycles, and various callbacks are available for initialization, continuous updates, and cleanup.
- The Hammer Editor in Source Engine allows for defining entity relationships and behaviours.
- Source Engine's entity-component system provides a versatile approach to handling game object lifecycles, allowing for customization within the constraints of the system.

2.3 Technologies Researched

Programming Languages

Programming languages were assessed based on these requirements:

Performance

Memory Safety

Library Support

Low-Level Memory Access

Community and Industry Adoption

Figure 7

Performance is an important aspect of Game Engine design. Game engines need not be limited by a languages speed and should operate as efficiently as possible on the hardware it runs on. Typically compiled and non-virtualized languages offer these benefits.(4)

There were many separate languages analysed for this project. To start I researched languages which I deemed were un-fit for game engine creation. A few examples were interpreted languages such as Python. The slower nature of iterators and overall, of control flow of the language, leads to significant slowdowns in regularly running tasks. While modules can be written in C, this almost defeats one of Python's greatest uses (it's simplicity).

In general, interpreted languages were ignored, as it would be an error to create an engine, which will inevitably run others code, in a language with built in performance overheads.

The next set of languages to look at are those with garbage collectors (systems which enforce memory safety through identifying and freeing up memory that is no longer in use), and runtime environments (a virtual space that provides the necessary resources and infrastructure for executing code correctly on any platform).

2.3.1 Java

I looked at Java, a very popular systems language. Some of its greatest benefits are the provided memory safety (the JVM or Java Runtime Environment) through reference counting, and the JIT compiler, which optimizes frequently used aspects of code in real-time, providing great optimizations in long running operations (which ostensibly a game is). (5)

Java (and java-based languages) became untenable when analysing its general performance. The memory safety of the garbage collector becomes negated by the extra processing times required for the garbage collector to run, releasing memories during calculations can slow the code execution. Java also does not support low-level memory management, and pointer access, which inhibit the programmer's ability to improve their code. Also, Java lacks overall library support. The language is not open source, which has caused legal trouble in the past and makes the future of the language unpredictable.(6) The lack of language features such as operator overriding makes game engine design a more complex than needed endeavour.(7)

2.3.2 Go



Another language investigated was Go; a modern language developed by Google. (8) In terms of performance, Go's garbage collection and runtime features may introduce some overhead, potentially impacting the consistent frame rates crucial for game rendering(9). However, its simplicity and efficiency can make it well-suited for certain game development tasks, such as backend logic or tools. Memory safety is a notable strength of Go, reducing the likelihood of memory-related bugs, but this comes at the cost of limited low-level memory access, hindering fine-grained control over resources—important for tasks like real-time rendering, or repetitive memory reads in game engines. The language lacks the extensive library support for game development that is available in more established languages like C++ or C#. As for community and industry adoption, Go is not as prevalent in the game development space, with many game engines and tools favouring other languages. While Go may find its place in specific aspects of game development, it currently falls short in meeting the diverse and performance-intensive requirements of building comprehensive game engines. (10,11)

2.3.3 Zig



Figure 8

Zig is an emerging programming language that offers a unique blend of low-level control, safety features, and simplicity. Regarding performance, Zig allows for fine-grained control over memory management and offers predictable performance, making it suitable for performance-critical tasks in game development. Its focus on minimizing abstractions contributes to a level of control like that of C or C++, enabling developers to optimize code for efficiency.(12) In terms of memory safety, Zig employs a no-runtime garbage collection model, promoting manual memory management for precise control over allocations and deallocations. Library support for game development in Zig is still growing, with a smaller ecosystem compared to more established languages, but the language's design makes it interoperable with C libraries, expanding available resources. Low-level memory access is a notable strength, allowing developers to work closely with hardware, an essential feature for tasks like real-time rendering. While Zig's community and industry adoption are currently smaller than mainstream languages, its momentum is increasing, and it has the potential to become a viable

choice for game development, especially for those who value control and safety in a low-level language. (13)

2.3.4 C++

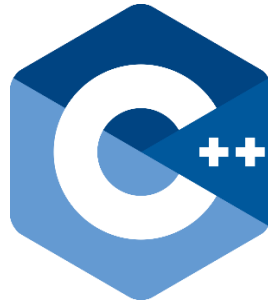


Figure 9

C++ is a highly favoured language in the game development industry, renowned for its robust performance, low-level control, and extensive library support. In terms of performance, C++ excels due to its ability to directly manipulate memory and efficient resource management, making it a top choice for developing high-performance game engines. Its memory safety features are versatile, allowing developers to choose between manual memory management and higher-level abstractions, providing flexibility based on project requirements. The language boasts an extensive collection of libraries, frameworks, and engines tailored for game development, such as Unreal Engine and Unity (for certain components). With its support for pointers and low-level memory access, C++ facilitates fine-tuned optimization, crucial for real-time rendering and resource-intensive tasks.(14) The widespread adoption of C++ in the game development community has resulted in a wealth of resources, documentation, and a large talent pool, making it a standard and reliable choice for creating comprehensive and efficient game engines. (15)

While C++ is widely used in game development and offers significant advantages, it's not without its challenges. The language's complexity and learning curve can be daunting for newcomers, potentially slowing down development and introducing a higher likelihood of errors, especially for those less experienced with its intricacies. C++ codebases can be more prone to issues related to memory management, potentially leading to memory leaks and undefined behaviour if not handled meticulously. The language lacks some modern features found in newer languages, which might impact developer productivity and code maintainability. Additionally, the diverse nature of C++ development practices and the variety of available libraries may lead to compatibility issues, making it challenging to achieve uniform standards across different projects. Despite its strengths, these drawbacks indicate there may be alternative languages with simpler syntax and more modern features which may be a preferable choice.

2.3.5 Rust



Figure 10

Rust stands out as an exceptional choice for game engine development, offering a powerful combination of performance, memory safety, extensive library support, low-level memory access, and a growing community. In terms of performance, Rust delivers efficiency akin to languages like C++ while providing a safer programming environment. Its ownership system and borrow checker ensure memory safety without the need for garbage collection, allowing developers to achieve high-performance results without sacrificing safety. The language's borrow checker also promotes concurrency and parallelism, crucial for modern game engines. Rust's library ecosystem, exemplified by projects like Amethyst, showcases a growing commitment to game development. With seamless interoperability with C and strong support for foreign function interfaces (FFI), Rust enables integration with existing C and C++ libraries, enhancing its library support even further. Rust's emphasis on zero-cost abstractions and low-level memory control provides game developers with the tools needed for resource optimization and real-time rendering. (16) The Rust community is vibrant, and the language is gaining rapid adoption in the industry, signalling a promising future for game development. The combination of performance, safety, library support, low-level access, and a thriving community positions Rust as a compelling choice; on top of Linux and Microsoft recent adoption of Rust, make it my ultimate preference for creating this engine. (17,18)

	Python	Java	Go	C++	C#	Haskell	Zig	Rust
Performance	N	N	N	Y	75 (not bare metal)	Y	Y	Y
Memory Safety	Y	Y	Y	N	Y	Y	Y	Y
Library Support	Y	N	N	Y	Y	N	N	Y

Low-Level Memory Access	N	N	N	Y	N (technically yes with unsafe flag)	Y	Y	Y
Community and Industry Adoption	N	N (excluding Minecraft)	N	Y	Y	N	N	Y
Relative Simplicity	Y	Y	Y	N	Y	N	N	N
Learning Resources	Y	Y	Y	Y	Y	Y	Y	Y

Figure 11

2.3.6 Relevancy of these metrics

The languages in Figure 11 were evaluated in their ability to match the criteria below, and their importance in relation to this decision.

1. Performance

Performance is incredibly important within the context of game engines. Game engines need to execute to produce upwards of 60 frames in a second, calculating both data on the GPU and CPU. Inefficiencies from a language used within the engine internals will cause exponential loss of *potential* performance from those using the engine.

A language which emphasizes stack allocation is important over languages which favour the heap. The selected language should emphasize stack usage. A language which utilizes stack memory will perform better and be less prone to null pointer exceptions (stack memory is almost always valid memory).(19)

2. Memory Safety

Game engines are glorified memory management systems. They create and sort game items, in both RAM and VRAM (graphics card memory) and need to accurately contain and control this data. The memory safety of a language is important to ensure the correct cleanup of both locations of data. Memory safety in a game engine is crucial for preventing crashes, avoiding security vulnerabilities, and ensuring stable, reliable performance across diverse gaming experiences. (20)

3. Library Support

Robust library support in game engines is essential for enhancing functionality and ensuring compatibility and efficiency across various platforms and technologies. Incorporating well-established libraries allows me to leverage pre-existing solutions for complex tasks such as physics simulation, graphics API interaction and window management, significantly reducing development time and resource overhead.

4. Low-Level Memory Access

Low-level memory access in game engines is fundamental for optimizing performance, enabling precise resource management, and facilitating direct hardware interactions, crucial aspects for demanding real-time applications like video games.(21)

5. Community and Industry Adoption

Community and industry adoption of a game engine plays a pivotal role in its development, providing a broad base of user feedback, fostering a rich ecosystem of shared resources, and ensuring long-term sustainability and relevance in the gaming market. Similar to library support, the use of the language or specialty it provides in the field its used in makes code examples, communities, books and actual programs using these features easier to come across with modern up to date information.(22)

6. Relative Simplicity

The relative simplicity of a programming language in game engine development is vital for accessibility, reducing the learning curve, and streamlining the process of game design and implementation, especially for developers with diverse skill levels.

2.3.7 Graphics Libraries

OpenGL

OpenGL, short for Open Graphics Library, is a widely used cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. It finds extensive use in various fields such as video game development, CAD (Computer-Aided Design), and scientific visualization, owing to its high efficiency and hardware-accelerated rendering capabilities. (23) The major benefits of OpenGL include its compatibility with a wide range of hardware and operating systems, an extensive library of functions for complex graphics rendering, and a strong community support. However, OpenGL also faces issues like its steep learning curve for beginners, less frequent updates compared to newer APIs, and potential performance limitations on certain platforms.(24,25) Despite these challenges, OpenGL remains a cornerstone in the realm of graphics programming, offering a robust

foundation for a multitude of graphics-intensive applications. OpenGL is an open standard which is implemented by the graphics drivers which use it.(23)

DirectX 12

DirectX, developed by Microsoft, is a suite of multimedia APIs used primarily for video game programming on Windows and Xbox. Its most notable component, Direct3D, manages 3D graphics rendering. DirectX is renowned for its high performance and ability to manage complex graphics, which is integral for modern gaming. While it continues to evolve and support the latest technology, DirectX's use is primarily limited to Microsoft platforms, and its complexity requires a substantial understanding of the API for effective utilization.

DirectX is not cross platform, or open source. This means that updates leave the user to the whims of Microsoft, and makes the task of adding any other platforms requiring a huge workload of supporting both DirectX and then some other (probably cross platform) API.(26)

Vulkan

Vulkan is an innovative graphics API that offers several significant advantages over its predecessors, particularly OpenGL. It provides lower overhead, more direct control over the GPU, and lower CPU usage. Vulkan is designed to be a unified API for both desktop and mobile graphics devices, which contrasts with the previous split between OpenGL and OpenGL ES. This unified approach makes Vulkan available on a wide range of operating systems, including Android, Linux, BSD Unix, QNX, Haiku, Nintendo Switch, Raspberry Pi, Stadia, Fuchsia, Tizen, and Windows. Additionally, there's third-party support for macOS, iOS, and tvOS via MoltenVK, which wraps over Apple's Metal API.(27)

One of Vulkan's major benefits is its support for a wide variety of platforms, allowing developers to save time and money by learning and supporting just one API. Game engine developers widely adopt it, with major engines like Unity, Unreal Engine 4, and CryEngine already featuring full Vulkan support. Vulkan's design includes batching to reduce CPU load, acceleration of effects with on-chip memory prioritization, and finer-grained control over memory allocation.(28)

Comparisons

	OpenGL	DirectX 12	Vulkan
Cutting Edge	N	Y	Y
0 Cost Abstractions	N	Y (12 not 11). (29)	Y
Open Source	Y	N	Y
Multi Core utilization	N	Y (12 not 11). (29)	Y
Cross Platform	Y (not to mobile)	N	Y

For the reasons listed above, and a general preference to Vulkan (specifically related to its open-source nature and innovative features, I have decided to choose it for the project. The support available for it is exceedingly high, its validation layer provides best practices tips, and its documentation is verbose to the point of fault.

2.4 Other Research

wgpu

wgpu is a cross-platform, safe, pure-rust graphics API. It runs natively on Vulkan, Metal, D3D12, and OpenGL; and on top of WebGL2 and WebGPU on wasm. (30) It is used in many successful projects and has some great examples of 3d rendering concepts with Vulkan and Rust.

specs

Specs is an Entity-Component System written in Rust. Unlike most other ECS libraries out there, it provides; easy parallelism, high flexibility, contains 5 different storages for components, which can be extended by the user, its types are mostly not coupled, so you can easily write some part yourself and still use Specs, systems may read from and write to components and resources, can depend on each other and you can use barriers to force several stages in system execution. (5)

Legion

Legion is a high-performance Entity Component System (ECS) library for Rust, designed to facilitate efficient and data-oriented game development and simulation systems. (31) In Legion, the emphasis is placed on maximizing performance by leveraging Rust's type safety and zero-cost abstractions, alongside the core principles of ECS and data-oriented design. While exploring Legion, I have learned about its powerful features, such as dynamic and static systems, which offer flexibility and performance optimization. (32) The library's design allows for highly parallelized execution of systems, taking full advantage of modern multi-core processors. Legion's architecture also emphasizes the use of archetypes for entity storage, which enhances cache locality and minimizes unnecessary data processing. Another key learning point could be its flexible query system, enabling complex data retrieval patterns while maintaining performance. Understanding Legion's approach to component storage, system execution, and entity management has provided valuable insights into effective ECS architecture and the practical application of data-oriented design principles.

Doom

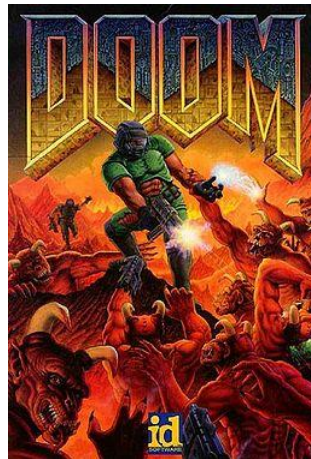


Figure 13

Doom is a game created in 1993 by John Carmack and John Romero. Analysing the source code of classic games like "Doom" provides invaluable insights for modern engine development. Despite being developed in the early 90s, Doom's engine showcases pioneering techniques in real-time 3D graphics, spatial data structures (like BSP trees), and efficient resource management, which remain relevant in today's game development landscape. By studying its architecture, I can gain a historical perspective on how fundamental problems in engine design were solved with limited hardware resources. This can help particularly in optimizing for performance and understanding the evolution of game engine components. Doom's source code, which is known for its readability and clever algorithms, serves as a great educational resource, offering many tips in writing clean, efficient, and maintainable code, which are crucial skills in modern engine development.(33)

2.5 Existing Final Year Projects

SOL Engine

The SOL game engine, created by Simon O'Neill is a similar concept in ways. It implements many of the features I wish to.

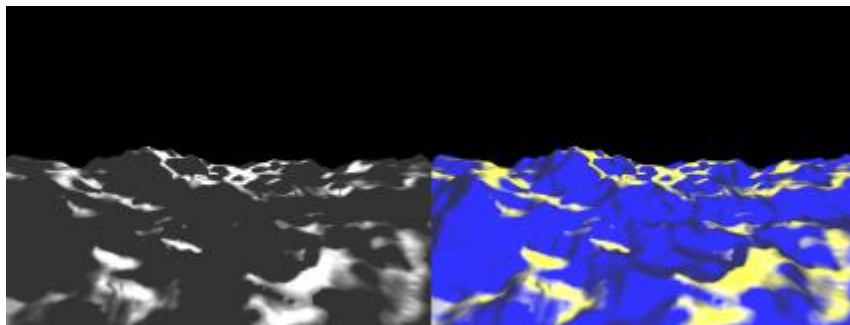


Figure 14

Where this project is complex: The project implements a fledged rendering system, physics system and audio system. The library, written by Simon, is in C++ and utilizes OpenGL for its rendering. The project has appealing examples which highlights the engines features.

Where this project lacks: The project has a dependency on older technologies which are not as forward looking and open the project to increased risk of memory leaks and falling out of compatibility with modern technology and features.

Deb Engine

The DEB game engine, created by Michal Debski is a Vulkan based game engine created by Michal Debski.

Where this project is complex: This project implements a featured editor which can be used to interact with components within the engine, as well as advanced shader implementations.

Where the project lacks: There is no implementation of a physics system, and the scope for the project is quite broad.

2.6. Conclusions

This chapter focuses on what the system is attempting to accomplish, and the technologies needed to do this. Additional software will be utilized throughout for profiling, debugging, and logging, as well as different areas will be taken for influence if a certain feature's requirements are blocked in development. Seeing how other people wrote and accomplished their feats in game engines will be critical to my completion and progression of this project.

3. System Design

3.1. Introduction

Developing a comprehensive system design plan is a crucial step in creating a game engine that meets performance, functionality, and scalability requirements. This plan provides a blueprint for the overall architecture, components, and interactions within the engine. This introduction outlines key considerations and methodologies essential for crafting an effective system design plan for your game engine.

System Design Objectives:

The primary objectives of the system design plan are to establish a coherent and scalable architecture for the game engine, define the relationships and interactions among various components, and outline mechanisms for extensibility and maintainability. The design should align with the intended use cases, account for potential future enhancements, and provide a clear foundation for the development team.

This is beneficial for maintainability and scalability of the engine and its components.

Design Methodologies:

Modular Architecture:

Embrace a modular design approach to promote component independence and reusability. Define clear interfaces and boundaries between modules. This allows for easy integration and future expansion, as well as integration with other existing libraries.

Component-Based Design:

Adopt a composition over inheritance architecture, where each struct provides reusable and interchangeable components. This approach facilitates a flexible and extensible framework, allowing developers to construct complex entities by combining different components, instead of through the incompatible and more difficult to understand object orientated patterns.(34)

Data-Orientated Design:

Implement a data-orientated design approach to separate game logic from data. Define game content, behaviours, and configurations in external data files, promoting flexibility and easing content creation and modification. (35)

Cache Utilization: Data-Orientated Design optimizes data layouts in memory to make effective use of the CPU cache, reducing cache misses and memory access times. This optimization is achieved by

organizing data to enhance spatial and temporal locality, which is crucial for high-performance computing tasks.

Vectorization and Parallelization: By organizing data in a way that is conducive to SIMD (Single Instruction, Multiple Data)(36) operations and multi-threading, Data-Orientated Design allows for more efficient use of modern CPU architectures, enabling better parallelization and faster processing.

Concurrency and Parallelism:

Plan for concurrency and parallelism in the design to leverage multi-core processors effectively. Identify opportunities for parallel execution of tasks, especially in performance-critical systems like physics simulations and rendering.

Design Documentation:

Develop design documentation that includes:

- Architectural Overview
 - A high-level description of the game engine's structure, including major components and their interactions.
- Component Specifications
 - Specifications for components, outlining their responsibilities, dependencies, and interfaces.
- Data Models
 - Define the data models for entities, components, and other relevant game data structures.
- Algorithms and Logic Flow
 - Document algorithms and logic flows for critical systems, such as rendering and physics.

Extensibility and Future Considerations:

Anticipate future needs and changes by designing the engine to be extensible. Incorporate hooks, event systems, and well-defined extension points that allow for easy integration of new features or modifications without extensive re or reverse engineering .

Iterative Development:

Encourage iterative development by employing version control systems, using kanban boards to mark progress, and ensure that the design plan is accessible clear and concise.

3.2. Software Methodology

Choosing an appropriate software methodology is paramount in guiding the systematic development of a game engine. Game engine design involves intricate systems and performance considerations, making the selection of the right methodology crucial for ensuring a robust and efficient development process. This introduction explores key considerations and methodologies integral to establishing an effective software methodology tailored specifically to game engine design.

Methodology Objectives:

The primary objectives of the chosen software methodology for game engine design are to address the complexity and performance requirements inherent in developing game engines, facilitate collaboration among multidisciplinary teams, and allow for iterative refinement as the engine evolves.

Iterative Prototyping:

- Flexibility and Adaptability
 - Game engine development can be dynamic, with evolving requirements and the need for continuous refinement. Agile methodologies, such as Scrum or Kanban, offer the flexibility to adapt to changing needs, fostering responsiveness and efficiency.
- User Stories for Features
 - Utilize user stories to define features from an end-user perspective. This helps prioritize development efforts based on the most critical functionalities and ensures that the engine aligns with the needs of game developers who will use it.

Kanban boards were used to indicate aspects of the project (determined by my requirements), and to reinforce iterative prototyping. This will allow development and testing features through progressive prototype creation and allow easier ability to track progress. This will grant greater insight into the state of the project, and when aspects will be ready, and to update the GANTT chart in case of unexpected issues.

The Kanban board will be used in cooperation with two-week sprints to complete features, with the weekly meetings happening with Bryan on Thursdays to engage in sprint reviews.

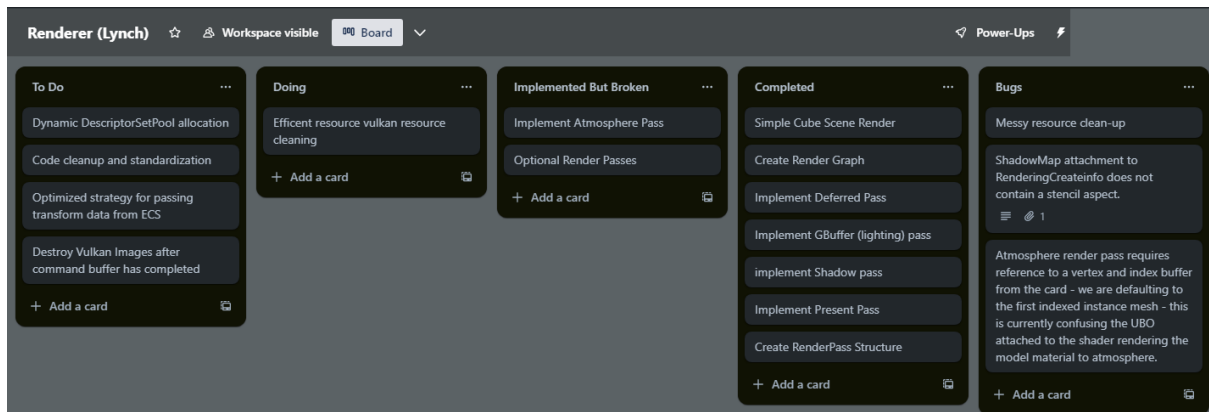


Figure 15, Kanban of Renderer

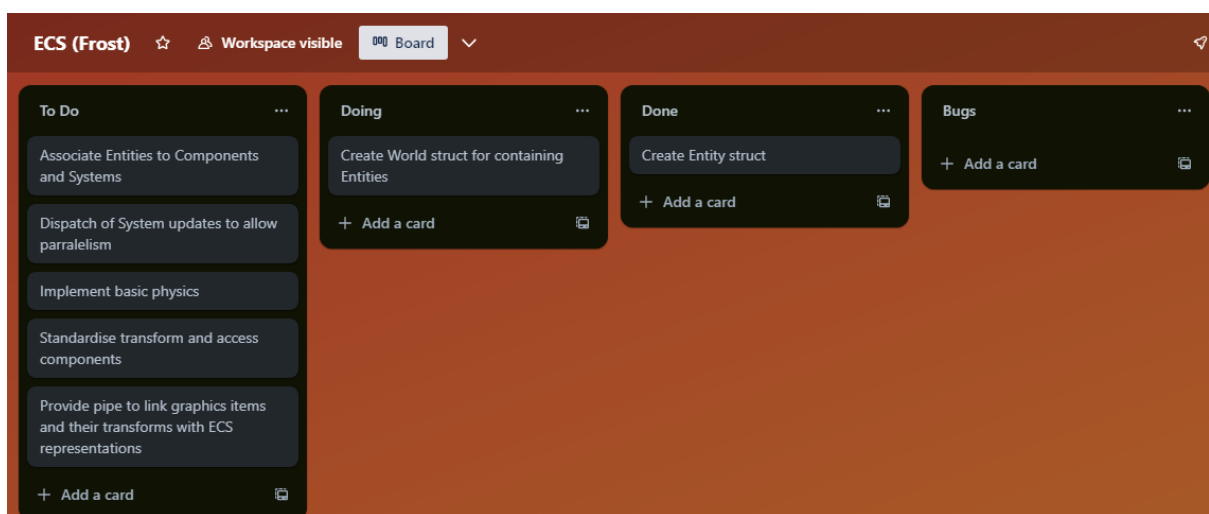


Figure 16, Kanban of ECS

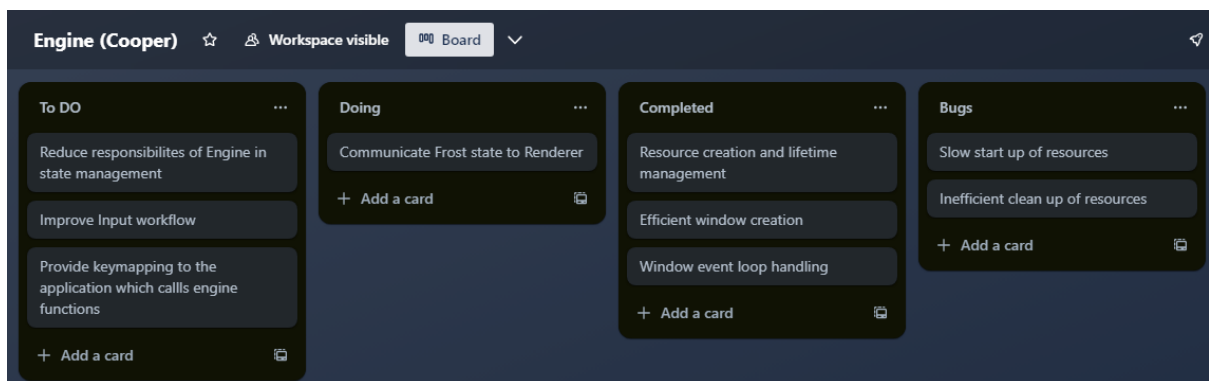


Figure 17, Kanban of Engine

Figure 15, Kanban of Renderer - Figure 17, Kanban of Engine shows the Kanban board used in collaboration with these sprints, showing which bugs were added as they are discovered and as the engine is tested, as well as the movement and progression of features as they are required.

3.3 Overview of System

The project is separated into three separate crates, which can be imported by the user to create an instance of the engine, the application, rendering and state crates. Dividing the project into crates allows for a modular design, where each crate encapsulates a specific functionality or feature. This modular approach makes the codebase easier to understand, maintain, and extend. I can work on individual crates independently, focusing on specific tasks without having to deal with the entire codebase, or effecting other compile targets. Crates provide a level of encapsulation, allowing me to hide implementation details and expose only the necessary interfaces or APIs to other parts of the engine. This also facilitates easier testing and maintenance. Each crate can have its own set of unit tests and documentation, making it simpler to verify correctness and ensure quality. Additionally, maintaining separate crates promotes code isolation, making it easier to identify and fix bugs without affecting other parts of the engine.



Figure 18, Cargo Crates

1. Cooper Engine:

Responsibilities:

Cooper is the central component of the game engine, responsible for managing the overall engine state, coordinating interactions between different subsystems, and facilitating the flow of control during gameplay.

Key Features:

- **Scene Management:** Cooper manages game scenes, including loading, unloading, and transitioning between different game environments.
- **Core Logic:** Implements core game logic, such as game loops, time management, and event handling.
- **Extensibility:** Designed to be extensible, allowing for the integration of additional modules and features.
- **Scriptable update loop:** OnStart, Update, FixedUpdate and Close

2. Lynch :

Responsibilities:

Lynch focuses on rendering and graphics-related tasks, ensuring the visual representation of the game world and objects.

Key Features:

- **Rendering Pipeline Control:**
 - Implement a rendering pipeline that processes geometric and visual data through various stages to produce the final image.
 - Control stages such as vertex shading, geometry shading, fragment shading, and post-processing.
- **Material and Texture Handling:**
 - Apply materials to objects, defining how they react to light and other environmental factors.
 - Manage texture mapping, including the application of textures to surfaces for realistic visual effects.
- **Lighting and Shadowing:**
 - Implement the standard lighting model.
 - Implement lighting algorithms to simulate the interaction of light with objects in the scene.
 - Generate shadows to add depth and realism to the rendered images, utilizing techniques like shadow mapping.
 - Plan to add point lights in the future potentially.
- **Camera Management:**
 - Render the virtual camera to define the perspective and view of the player.
 - Implement features such as camera movement, rotation, and field of view adjustments.
- **GPU Resource Management:**
 - Efficiently manage graphics resources on the GPU, including buffers, textures, and shader programs.
 - Minimize memory usage and optimize resource allocation for rendering tasks.

Vulkan Extension Features

Vulkan extensions and features are additional functionalities provided by the Vulkan graphics API beyond its core specifications. These extensions and features offer developers access to advanced rendering techniques, optimizations, and platform-specific capabilities. They are designed to enhance Vulkan's versatility and performance across a wide range of hardware and software environments. (37) Extensions planned to be used are, dynamic rendering and advanced memory management, although multi-view rendering and more dynamic features may be implemented in the future.

Command Buffer

A command buffer quite simply records command. This includes commands to begin and end the render pass, bind pipelines, bind vertex, and index buffers, and draw calls. Once the command buffer is recorded, it's submitted to a queue to be executed by the GPU. (38)

Pipeline Configuration

A pipeline is a fundamental Vulkan concept that encapsulates the complete set of configurations and states that control the rendering and compute operations. It defines how the GPU processes data, from the input stage all the way through to the output. This pipeline is highly configurable and designed to be specified upfront and used without changes, which allows for significant optimizations by the driver. There are typically two types of pipelines, a graphics and compute type pipeline. A graphics pipeline starts by collecting raw vertex data from buffers and organizes them into primitive types (points / lines /triangles) based on commands passed to the buffer. It also performs tasks such as transforming and lighting vertices. A fragment shader can then rasterize this shape with textures, before being combined in the colour blending stage. Finally a depth and stencil check is ran which will discard unnecessary fragments.(39)

A compute shader allows for general-purpose computing tasks, like physics simulations or image processing, utilizing the GPU's parallel processing capabilities.

Creating pipelines is a computationally expensive task which ideally should only be executed once during the lifetime of the engine / game, and only one pipeline needs to exist for N amount of inflight frames. A hash map storing the pipeline name will be used to receive pipelines quickly and cheaply from the renderer.

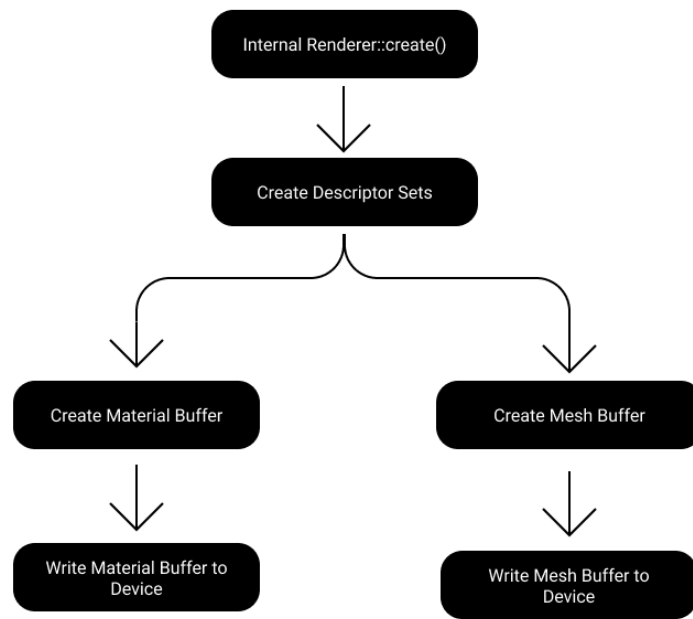


Figure 19, diagram of the Renderer initialization.

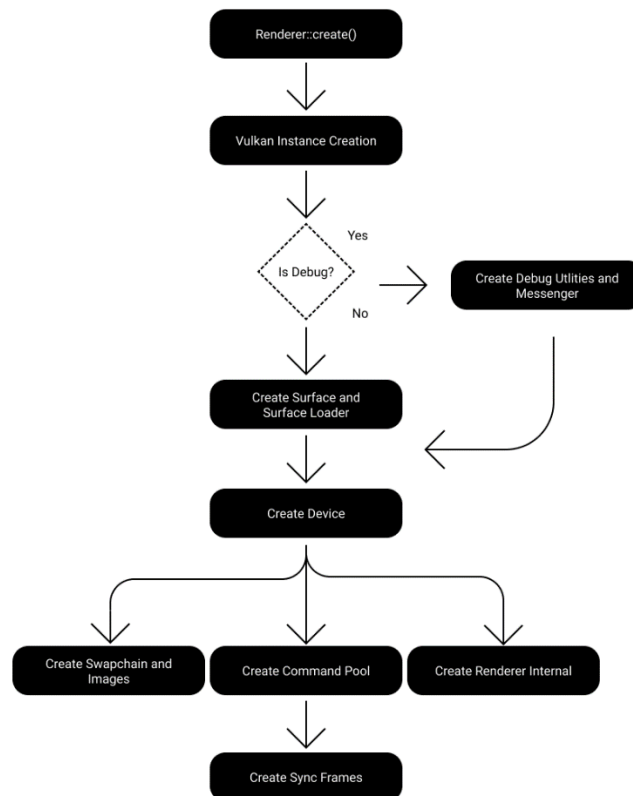


Figure 20, diagram of the Renderer creation.

Transferring data to and from a shader

To transfer data like vertices, textures, or arbitrary data to the shaders, we use `VkBuffer`'s and `VkImage`'s. These are GPU memory resources that shaders can read from or write to. Data is uploaded to these resources using staging buffers. Once the data is in GPU memory, you use descriptor sets to tell the shaders where to find this data. Descriptor sets reference buffers and images and are bound to the command buffer before drawing. We will use a standard format which provides a standard to communicate data to the shader with.(40)

Render Passes

A render pass in Vulkan involves a series of steps where the GPU executes commands to render graphics or perform computations. Vulkan is explicit and requires the developer to provide detailed information about how resources are used throughout the rendering process. The project will utilize Vulkan's dynamic rendering in Vulkan, introduced with the `VK_KHR_dynamic_rendering` extension, which enhances flexibility and reduces the complexity of code, particularly for game engines which require dynamic changes during rendering. (40) First, we will call define a `VkRenderInfo` object. This involves specifying the attachments (colour, depth, stencil buffers etc), their usage, and how they will be transitioned throughout the rendering process. This `VkRenderPass` is then passed to the command buffer through the begin rendering command. Although sub passes are generally done within a render pass, that process will not be used in this project as I deemed it out of scope. Within this render pass, we need to refer to any draw calls which need to need to be executed (the rendering part of the process). We the end the render pass, through the end rendering command, and begin our next step. It is important that the last pass (typically the "present pass") transitions the necessary images from their shader layout to presentation layout.

Render Graph

A render graph is a high-level abstraction used to represent the dependencies and execution order of various rendering operations within a frame. It organizes the rendering process into a directed acyclic graph where nodes represent rendering tasks or passes. (41) This structure allows for automatic management of resource lifetimes, efficient execution order determination. By clearly defining how different rendering passes interact and what data they produce or consume, a render graph enhances the clarity, maintainability, and performance of the rendering pipeline, enabling more complex and efficient rendering strategies in real-time applications.

I have decided to use a deferred pass for lighting, which allows for the efficient rendering of complex scenes, as it decouples the geometry processing from the lighting calculations, significantly reducing

the computational load. This approach also offers enhanced flexibility in applying post-processing effects and optimizing various lighting and shading techniques, making it ideal for a game engine environment. (42)

Passes To Be Implemented

Shadow Pass

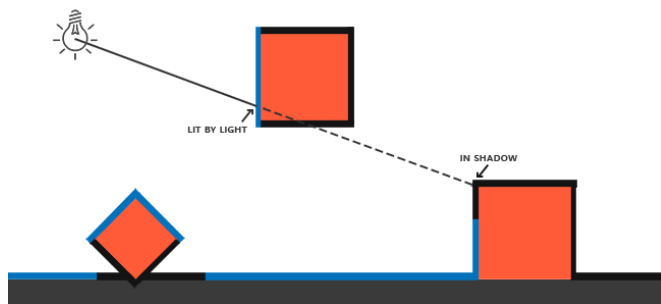


Figure 21, shadow map generation.

A shadow pass in graphics rendering is a process where the scene is rendered from the perspective of the light source to create a depth map, known as a shadow map. This shadow map is then used in subsequent rendering passes to determine which parts of the scene are in shadow and which are illuminated. The view of a

shadow pass is from the light's direction (in this project, scope is set so a global sun will be used).

The main rendering pass relies on the shadow map to determine lighting for each fragment.

Therefore, the shadow map must be fully generated before the main rendering can accurately calculate lighting. (43) Figure 21, shadow map generation., taken from a LearnOpenGL resource. (44)

G-Buffer Pass (Geometry Buffer)

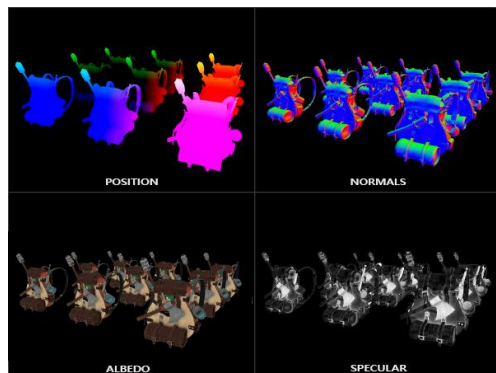


Figure 22, Four typical G-Buffer passes.

The Geometry Buffer pass, or G-Buffer, central to deferred shading in graphics rendering, involves rendering scene geometry into a series of textures. This process captures various surface attributes such as position, normal, colour, specular intensity, and depth at each pixel. Unlike traditional forward rendering, where each light's effect is computed directly on every surface, the G-buffer pass decouples geometry rendering from lighting. In a deferred shading pipeline, the scene is first

rendered to populate the G-buffer without applying lighting. Each attribute is stored in a separate texture, allowing for detailed surface information to be retained. Subsequently, lighting calculations are performed in a separate pass, utilizing the G-buffer's data to compute how light interacts with surfaces. (45) Figure 22, Four typical G-Buffer passes. four typical G-Buffer textures from LearnOpenGL. (45)

Deferred Pass (/ lighting pass)

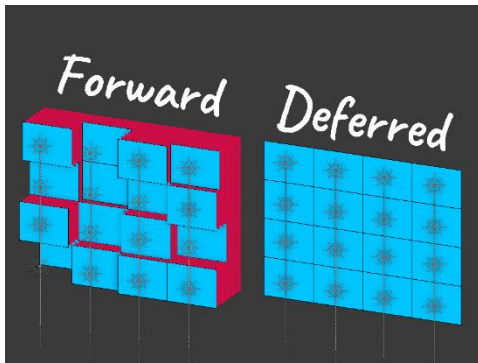


Figure 23, Forward Vs Deferred

A deferred pass, or lighting pass, is where lighting calculations are applied using the aforementioned G-buffer. In this stage, the scene's lighting is computed by iterating over the lights and applying their effects based on the information stored in the G-buffer, which includes data like normal, depth, and albedo from the scene geometry. Unlike in forward rendering, where each light source is calculated in conjunction with the geometry, the deferred pass separates these concerns, allowing for the

scene to be illuminated independently of the geometry pass. This methodology significantly enhances performance, especially with complex scenes featuring numerous light sources, as it avoids the repeated calculation of lighting for each pixel and light combination. (42) During the deferred pass, the renderer reads the G-buffer textures, applying lighting equations to determine the final colour of each pixel. This process will handle lighting models and effects, such as specular highlights and shadows, to create a realistic portrayal of how light interacts with the environment. By consolidating the intensive lighting calculations into this deferred pass, the rendering pipeline achieves greater scalability and flexibility, enabling more dynamic and detailed lighting scenarios in real-time applications. (45) Deferred rendering was chosen over forward rendering to reduce the requirements of rendering tasks. As Figure 23, Forward Vs Deferred shows, deferred rendering uses less passes by deferring fragment shading until the last step, where lighting is calculated per pixel, instead of per fragment. (46)

Present Pass

The present pass in graphics rendering is the final step in the rendering pipeline where the rendered image is sent to the display or the swap chain for display on the screen. This pass signifies the completion of a frame's rendering process, transitioning the rendered image from an off-screen buffer to the screen. The present pass involves submitting a present request to the swap chain, which handles the synchronization necessary to display the image at the right time, according to the display's refresh rate. This process ensures that the rendered content appears on the screen smoothly and without tearing, providing the visual output of the engines rendering pipeline. (47)

Extra Passes

The design should be robust enough to enable extra passes, such as an atmospheric pass, which would provide some skybox to the scene, and some post processing effects, such as FXAA, or screen

space ambient occlusion. Extra passes are not included in the scope of the project but may be added down the line. Some of these passes could be combined with existing passes.

FXAA



Figure 24, Nvidia FXAA demo

FXAA, or Fast Approximate Anti-Aliasing, is a screen-space anti-aliasing algorithm designed to smooth jagged edges in rendered images efficiently. It analyses rendered image's pixels and detects edges, then blending them to reduce the appearance of jagged lines. Unlike more computationally intensive anti-aliasing methods, FXAA is executed as a single post-processing pass, making it faster and less resource intensive than TAA or MSAA.

It can sometimes result in a slight blurring of the image, which is a trade-off against its performance benefits. (48) Figure 24, Nvidia FXAA demo taken from choi303's implementation. (49)

SSAO



Figure 25, SSAO in the Source Engine

SSAO, or Screen Space Ambient Occlusion, is a shading technique used to enhance depth perception in 3D scenes by simulating the way light is occluded by nearby objects. It approximates the shadowing effect that occurs in crevices and where objects meet, adding realism to the scene. Executed as a post-processing effect, SSAO analyses the depth and normal information

of the rendered image to calculate the occlusion amount, darkening areas where ambient light would be blocked. While not as accurate as global illumination methods, SSAO offers a good balance between visual quality and computational efficiency, making it a better choice for game rendering.

(50) Figure 25, SSAO in the Source Engine. (51)

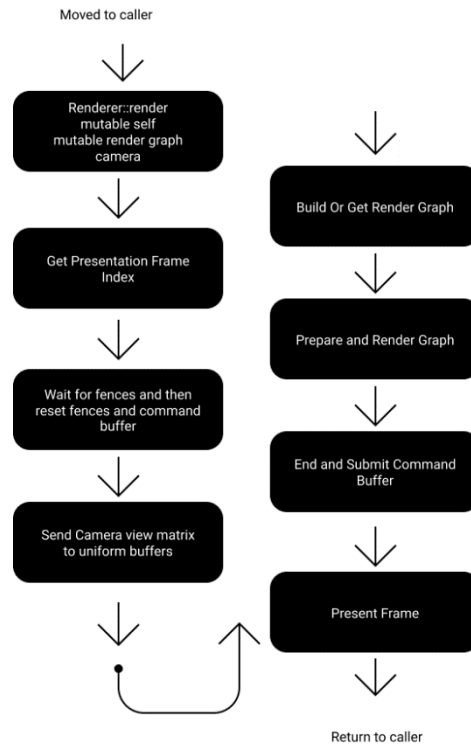


Figure 26, diagram of the Renderer Main loop.

3. Frost:

Responsibilities:

The state management library in this engine will deal exclusively with the ECS (Entity-Component System), physics and collisions, maintaining a separation of concerns, which will improve clarity and reusability. (52) The library must efficiently manage the lifecycle of entities, components, and systems within the ECS framework. This includes creating, updating, and destroying entities and components as the game world evolves. It should ensure that systems have timely and consistent access to the components they need to process, reflecting changes in the game state accurately and efficiently.

Entity-Component-System (ECS):

“Entity-Component–System (ECS) is an architectural pattern. This pattern is widely used in game application development. ECS follows the composition over the inheritance principle, which offers better flexibility and helps you to identify entities where all objects in a game’s scene are considered an entity.” (53)

Figure 27, Quote from Guru 99

This project will implement an ECS architecture for efficient data processing and modular game object design, allowing developers to create dynamic and complex relationships between components of their game.

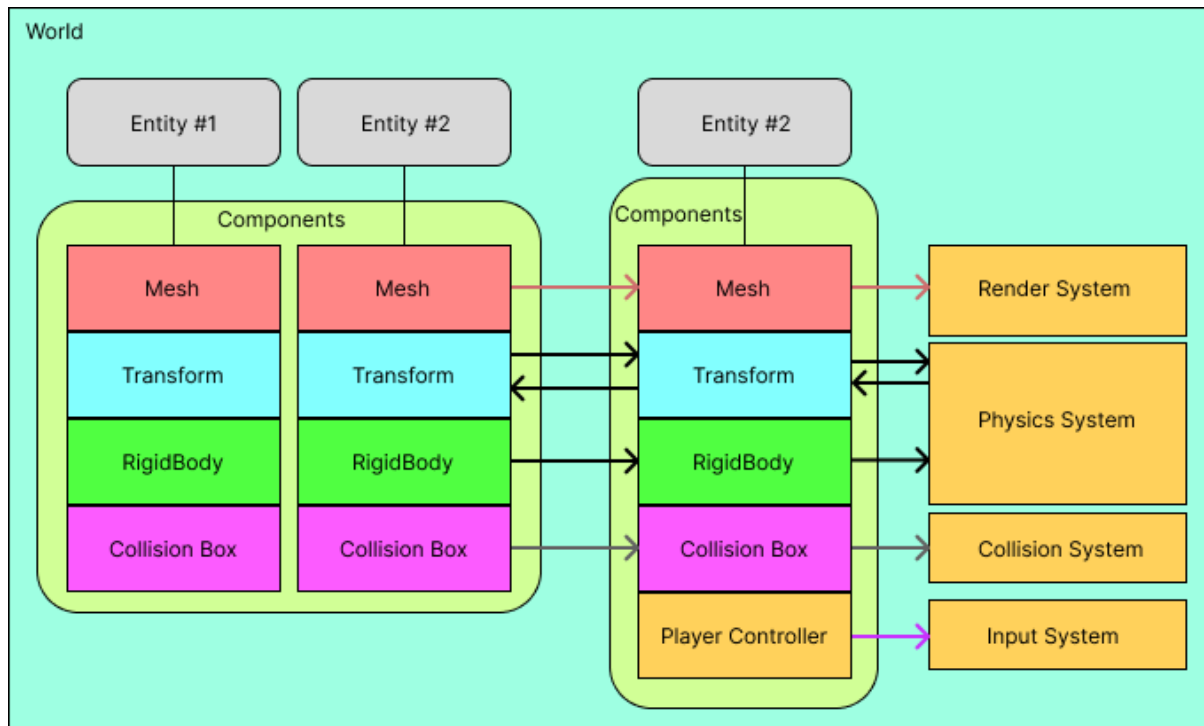


Figure 28, diagram of the designed ECS, extended from Unity Docs ECS diagrams.

Entities:

An entity is a general-purpose object that represents a single item within the game world, such as characters, items, or environmental features. Entities are lightweight and flexible; they don't inherently possess any behaviour or data. Instead, they serve as unique identifiers to which components are attached, defining their properties and behaviours. (54) This separation of identity from functionality allows for a highly modular and dynamic system where entities' characteristics and behaviours can be altered at runtime by adding, removing, or modifying their associated components, enabling developers to create complex and varied interactions within the game's ecosystem.

Components:

A component represents a discrete chunk of data or state without any associated behaviour. Components are used to attach specific attributes or properties to entities, effectively defining their characteristics and the way they should be processed by systems. For instance, a "Position" component might store the coordinates of an entity in the game world, while a "Health" component might keep track of an entity's health points. By attaching different sets of components to entities,

developers can compose varied and dynamic game objects on the fly, enabling a flexible and data-driven approach to game design where behaviour is determined by the systems that process these components.(55)

Systems:

Manages systems responsible for processing specific aspects of game logic, such as player controls and movement, collisions, physics updates and the use of archetypes to improve cache locality(56)

Figure 28, diagram of the designed ECS, extended from Unity Docs ECS diagrams. shows the implementation of the ECS and its proposed methodology. An entity exists as a numerical reference to its location in a list, where components are associated with these entities. Systems interact with entities which contain certain components and join those entities to run systems.(57)

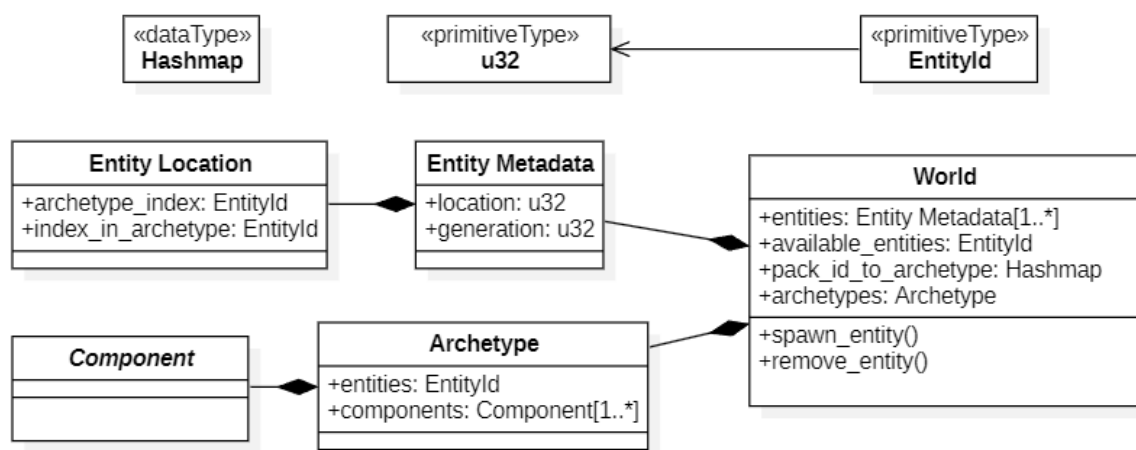


Figure 29, Diagram of ECS world and layout.

Archetype:

The ECS will utilize an archetype pattern, where an Archetype is a container of component arrays. (58) The archetype holds a signature of the component's types, which allows for faster searching of components within the World space. A world object then stores this data within lists, allowing for the index / location of the entity, and the entities components to store within an archetype. A generation system will be used so that reused entities can hold unique values.

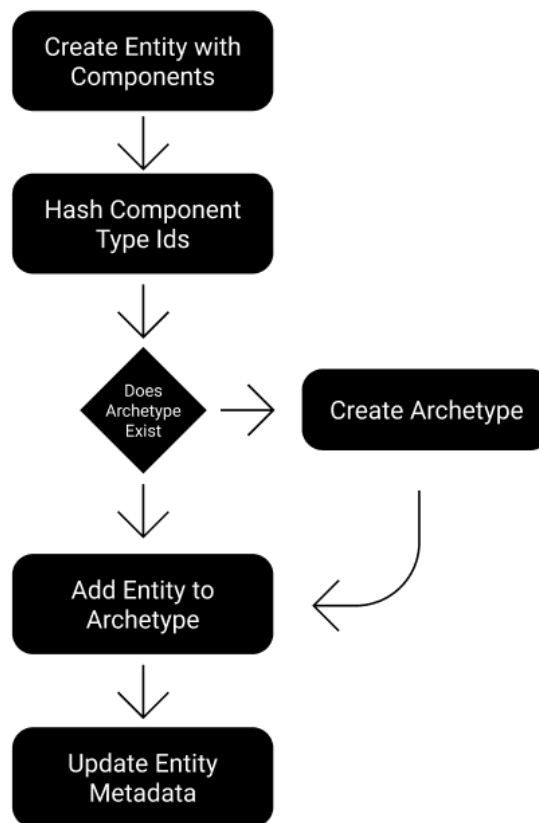


Figure 30, Diagram of adding Components and Entities to Archetypes

Figure 30, Diagram of adding Components and Entities to Archetypes describes the process of adding an entity to the world, where an archetype is created if it does not exist. The design is optimized to store related sets of information closer together within heap memory, which increases the chances of the CPU successfully “hitting” its cache.(59)

Cache locality refers to the tendency of a computer program to access data locations within proximity to one another during execution. There are two primary types of cache locality:

1. Temporal Locality: This occurs when the same memory location is accessed repeatedly within a short time span.
2. Spatial Locality: This happens when memory locations close to each other are accessed sequentially.

The improvement in cache locality translates directly to performance gains:

- Faster Data Access: Improved cache hits mean faster data access times, as accessing the cache is significantly quicker than accessing main memory. (60)
- Enhanced Parallelism: With data organized contiguously, it's easier to parallelize operations across multiple entities, leveraging modern CPU architectures more effectively.

- **Reduced Overhead:** The archetype pattern reduces the overhead associated with managing scattered components, thereby streamlining the execution of systems that operate on these

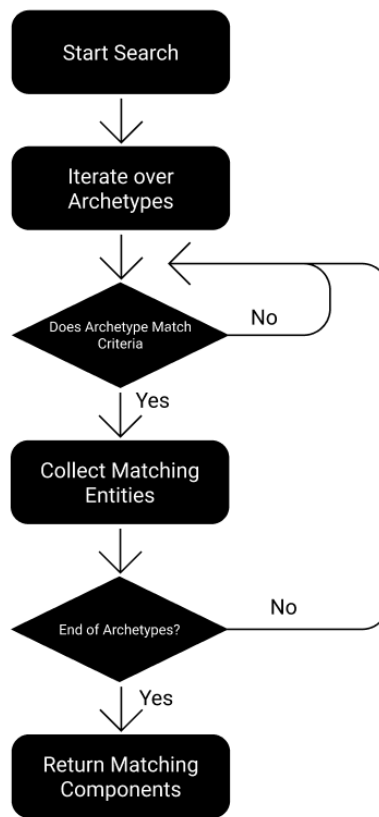


Figure 31, Diagram of searching the World.

components.

- **Input System:**
 - **Input Handling:** Manages user input from keyboard and mouse.
 - **Input Mapping:** Provides an input mapping system to customize and map user inputs to in-game actions.

The game state will be stored on two worlds, like the swap chain approach of a graphics card. One mutable state which is written to and prepared for next frame, and the current which is immutable. Both states pointers will swap locations in memory, meaning that state for memory is not allocated on the heap.

Multithreaded communication will happen through the Go language style. “Don't communicate by sharing memory, share memory by communicating.”(61) Due to Rust's borrow checker it is important that mutable memory is not shared cross thread (this is impossible without explicit guards removed within the language). The system will communicate events through a channel approach, where the central event handle can then process events processed and created by multiple threads.

Physics and Collisions

A physics system in this engine is responsible for simulating the physical behaviour of objects within a game world. It handles aspects such as movement, forces, velocities, and collisions. The physics system ensures that objects in the game world behave realistically according to the laws of physics, enhancing the immersion and realism of the gaming experience. Within the ECS architecture, the physics system is responsible for managing the movement and interactions of entities within the game world. Leveraging Euler integration, this system calculates the new positions and velocities of entities based on forces applied to them. Euler integration is a simple yet effective numerical method for approximating the continuous motion of objects over time. By applying forces such as gravity, friction, and user input, the physics system computes the resulting motion of entities, updating their positions and velocities accordingly. This system ensures that objects in the game world behave realistically and respond dynamically to external influences, enhancing the overall immersion and believability of the game environment.

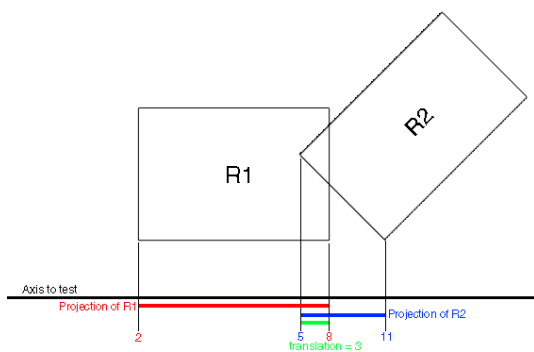


Figure 33, Separating Axis Theorem

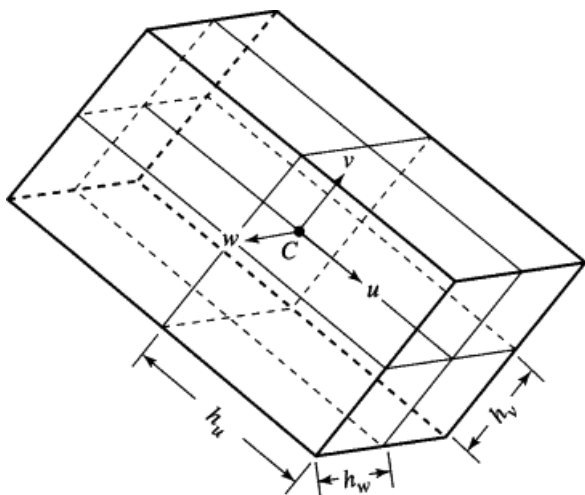


Figure 32, Orientated Bounding Box

In the ECS architecture, the collision system is tasked with detecting and resolving collisions between entities using Oriented Bounding Boxes (OBBs) and the Separating Axis Theorem (SAT). OBBs provide a flexible and accurate representation of the bounding volumes of entities, allowing for precise collision detection even when objects are rotated or scaled. The Separating Axis Theorem is a powerful algorithm used to determine whether two convex shapes overlap along a given axis. By applying the SAT to pairs of OBBs, the collision system efficiently identifies potential collisions between entities and computes the collision points and normal.(62–64) Subsequently, it resolves these collisions by adjusting the positions and velocities of the colliding entities, ensuring that they interact realistically and prevent objects from intersecting with each other. This collision system enables developers to create complex and dynamic interactions between entities, enhancing the depth

and realism of the game world. (65,66) Figure 33 was taken from [this](#) stack overflow thread. (67)

Figure 32 was taken from [this page](#). (65)

4. Collaboration:

Communication Between Components

Cooper serves as the orchestrator, facilitating communication between Lynch and Frost. Lynch provides Cooper with graphical information, while Frost contributes to the overall game logic and input processing. Lynch and Frost may collaborate directly for tasks like rendering ECS-managed entities or responding to user input during graphical interactions.

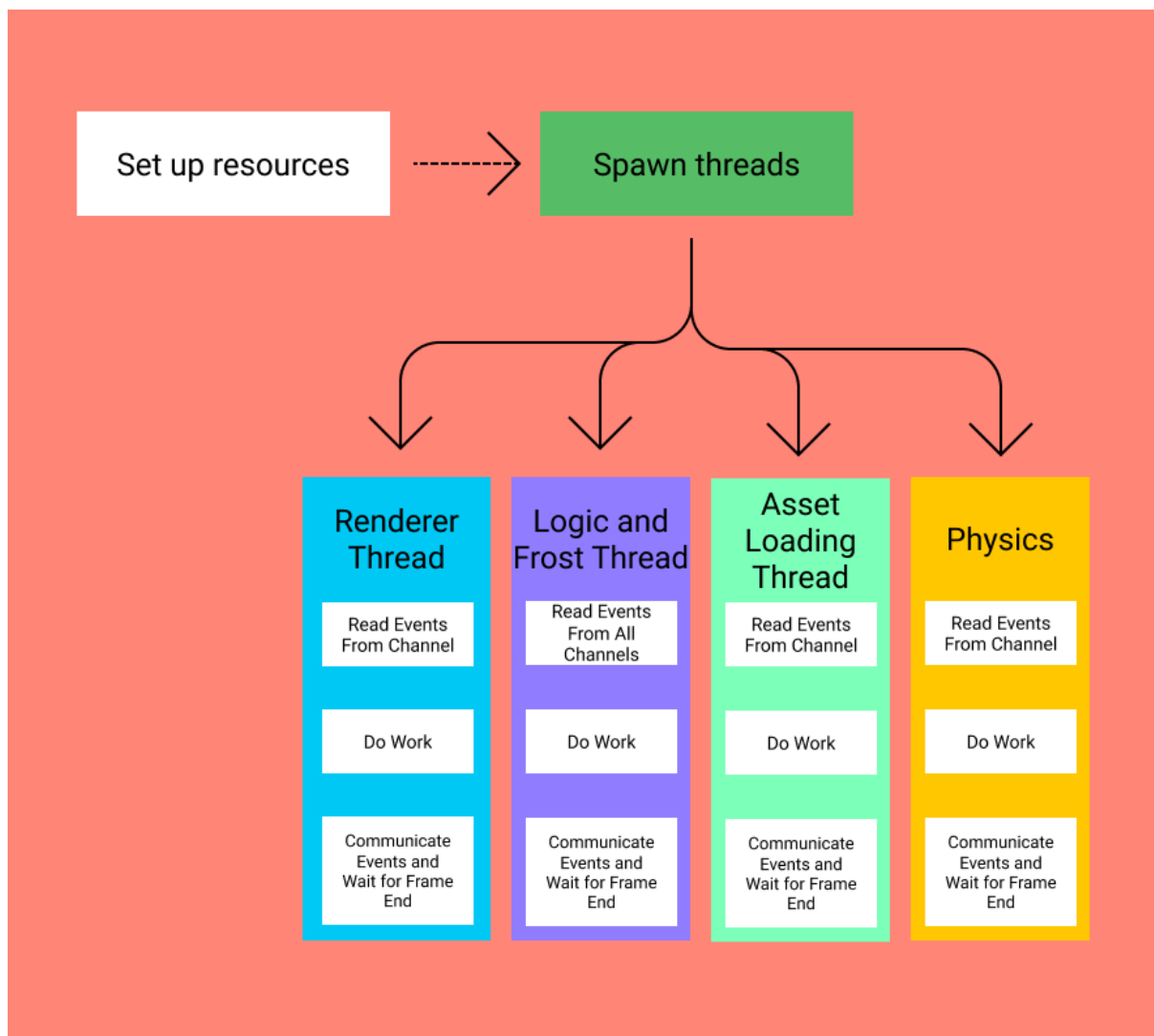


Figure 34, diagram of the designed architecture with communication.

With a focus on sharing memory by communicating, we will use producer and receiver architecture to synchronize and share data through threads. This will use tagged structs (Rust enums provide this

feature) we can communicate events with data, without ever violating the Rust borrow checker structure.

5. Extensibility and Customization:

Cooper will support using Rust's crates for extensibility, allowing developers to extend functionality by integrating additional modules and custom systems. (68)

3.4. Conclusions

In summary, the Cooper game engine system comprises the central engine (Cooper), responsible for overall game management, a dedicated graphics engine (Lynch) for rendering, and an ECS, state management, and input system (Frost) to handle entity behaviour, game states, and user inputs. The collaboration between these components creates a flexible and extensible framework for game development.

4. Testing and Evaluation Plan

4.1. Introduction

Testing a game engine is a critical aspect of ensuring its robustness, reliability, and adherence to design specifications. As a foundational component of your thesis, the testing phase serves to validate the correctness of the implemented functionalities, assess performance under various conditions, and identify potential areas for optimization and improvement. This introduction provides an overview of the key considerations and methodologies involved in testing a game engine.

Testing Objectives:

The primary objectives of testing a game engine are to verify that it behaves as intended, meets specified performance benchmarks, and remains stable under diverse scenarios. This involves scrutinizing various aspects, including graphics rendering, physics simulations, and any additional systems integrated into the engine. Testing also evaluates the engine's compatibility across different platforms and ensures that it gracefully handles unexpected inputs or edge cases.

Testing Methodologies:

- Integration Testing
 - Move on to integration testing, which assesses the interactions between different modules or systems within the game engine. This phase evaluates how well these components collaborate and whether they produce the desired outcomes when combined.

- System Testing
 - Conduct system-level testing to evaluate the game engine. This includes assessing its overall behaviour, performance, and adherence to the defined requirements. System testing also involves scenarios that simulate real-world usage conditions, helping identify potential bottlenecks or issues that might arise during actual gameplay. This will be done through playtests and user experience.
- Performance Testing
 - Measure the engine's performance under varying conditions, including stress tests to assess its limits. This involves evaluating frame rates, resource utilization, and responsiveness to ensure optimal performance across different hardware configurations.
- User Acceptance Testing (UAT)
 - If applicable, involve potential end-users or stakeholders in user acceptance testing. This provides valuable feedback on the engine's usability, user interface, and overall user experience.

4.2. Plan for Evaluation

The project will be evaluated based on the previously stated requirements, as well as the features of each individual component. With one of the goals of the project being performance, and memory management, profiled runs and benchmarks will be used to evaluate the goals of the project.

Examples of this would include benchmarking of the ECS, testing insertions and searches, benchmarking the rendering engine, testing frame rates compared to the number of mesh instances created, or passes enabled, and testing physics, scaling the number of colliders and rigid bodies up.

I will look to get feedback from my supervisor, and game design students within my college as well as other design students in other universities, on their evaluation of engine features and where they think the focus of future expansion should continue. Give it to a game developer and make something with the project.

4.3. Conclusions

Testing a game engine using traditional unit tests can prove tricky due to the interactive nature of its outputs. Features which can be tested, such as render initialization and struct creation can be tested traditionally, as well as aspects of the components and related systems, such as physics and collisions, which, process by process are determinative.

5. Prototype Development

5.1. Introduction

Prototyping is a fundamental phase in the development of any software, including game engines. It allows developers to test ideas, experiment with different architectures, and identify potential challenges early in the development process. In the context of a Rust game engine, prototyping becomes a crucial step for exploring the language's capabilities and ensuring that it aligns with the specific requirements of game development.

Key Stages of Prototype development:

1. Conceptualization
 - a. Define the scope and goals of the game engine.
 - b. Identify the target platforms and performance requirements.
 - c. Outline the core features and functionalities required for a minimal viable product.
2. Environment Setup
 - a. Establish the development environment with the necessary tools and libraries.
 - b. Select relevant Rust crates and dependencies for graphics, input handling, and other essential functionalities.
3. Architecture Design
 - a. Define the overall architecture of the game engine, considering how Cooper, Lynch and Frost will work in tandem.
 - b. Determine how Rust's ownership and borrowing mechanisms will be leveraged to ensure memory safety and performance.
4. Implementation of Core Components
 - a. Define the overall architecture of the game engine, considering factors such as entity-component systems, rendering pipelines, and physics simulations.
 - b. Determine how Rust's ownership and borrowing mechanisms will be leveraged to ensure memory safety and performance.
5. Iterative Testing and Refinement
 - a. Conduct regular testing to identify and address bugs and performance bottlenecks.
 - b. Gather feedback from friends, family, and supervisor to refine the engine's design and features.
6. Documentation
 - a. Document the codebase thoroughly to aid future development and collaboration.
 - b. Create tutorials and guides for developers who will work with or contribute to the game engine.

5.2. Prototype Development

5.2.1 Window creation

Prototype Development began with creating simple representations of engine features as “proofs-of-concepts”. Each section of the engine would be written and tested feature by feature, starting from the ground up. Each iterative change should provide clarity of features.

The earliest positive result seen was through the opening of a window using the Rust crate `winit`.

```

fn main() -> Result<(), impl std::error::Error> {
    let event_loop = EventLoop::new().unwrap();

    let window = WindowBuilder::new()
        .with_title("Cooper")
        .with_inner_size(winit::dpi::LogicalSize::new(1280., 720.0))
        .build(&event_loop)
        .unwrap();

    event_loop.run(move |event, elwt| {
        println!("{event:?}");

        match event {
            Event::WindowEvent { event, window_id } if window_id ==
window.id() => match event {
                WindowEvent::CloseRequested => elwt.exit(),
                WindowEvent::RedrawRequested => {
                    window.pre_present_notify();
                }
                _ => (),
            },
            Event::AboutToWait => {
                window.request_redraw();
            }
            _ => (),
        }
    })
}

```

Figure 5.2.1.1 code showing window creation and event handling.

Which produced the following output:

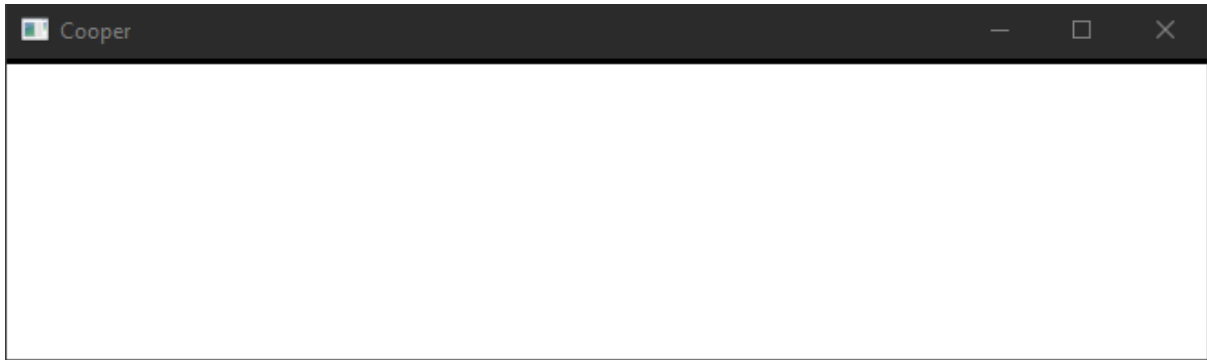


Figure 5.2.1.2 Screenshot showing window creation.

This shows some features of Rust which are suited to game design. Aspects Rust inherits from OCaml (primarily functional aspects) can feel more natural for event handling, as well as the `match` key word, which requires each match arm to be satisfied, and has low overhead abstractions using the tagged union enumeration pattern.

The next step was to begin learning and following the official Vulkan Tutorial to learn the basic patterns of Vulkan.⁽⁶⁹⁾ While there were alternatives written for Rust, I decided to convert the existing C++ code example as a base using the library Ash, which helped speed along both my understanding of the language and the library.

5.2.2 Vulkan integration



Figure 5.2.2.1, a rendered triangle

The triangle rendered here shows some of the first steps of communication between the program and the GPU.

Vulkan insists this code is verbose, as you must explicitly set every behaviour expected in rendering. This is a feature and not a bug but makes showcasing this code difficult. Below are some snippets of code which establishes connections to the graphics card:

```

let queue_info = vk::DeviceQueueCreateInfo::default()
    .queue_family_index(queue_family_index)
    .queue_priorities(&priorities);

let device_create_info = vk::DeviceCreateInfo::default()
    .queue_create_infos(std::slice::from_ref(&queue_info))
    .enabled_extension_names(&device_extension_names_raw)
    .enabled_features(&features);

let device: Device = instance
    .create_device(pdevice, &device_create_info, None)
    .unwrap();

```

Figure 5.2.2.2, some code to show queue and device creation.

This shows the general design pattern of Ash and rust. You create a create device queue info builder, you assign necessary values to it (such as the queue family requested), you pass that to a device create info builder, which is then passed to the Vulkan instance to create a device handle. This pattern repeats itself through the processes laid out in Figure 3.1.2.

This code was converted from the Vulkan tutorial C++ project to learn how to read and transfer APIs.(69)

5.2.3 Graph Structure, Creation and Building a render pass.

```

pub struct RenderGraph {
    pub passes: Vec<Vec<RenderPass>>,
    pub resources: GraphResources,
    pub descriptor_set_camera: Vec<DescriptorSet>,
    pub pipeline_descs: Vec<PipelineDesc>,
    pub current_frame: usize,
    pub device: Arc<Device>,
}

```

Figure 5.2.3.1, Shows render graph struct.

```

pub struct RenderPass {
    pub pipeline_handle: PipelineId,
}

```

```

pub render_func:
    Option<Box<dyn Fn(&Device, &vk::CommandBuffer, &VulkanRenderer,
&RenderPass, &GraphResources)>>,
pub reads: Vec<Resource>,
pub writes: Vec<Attachment>,
pub depth_attachment: Option<DepthAttachment>,
pub presentation_pass: bool,
pub read_resources_descriptor_set: Option<DescriptorSet>,
pub name: String,
pub uniforms: HashMap<String, (String, UniformData)>,
pub uniform_buffer: Option<BufferId>,
pub uniforms_descriptor_set: Option<DescriptorSet>,
pub copy_command: Option<TextureCopy>,
pub extra_barriers: Option<Vec<(BufferId, vk_sync::AccessType)>>,
device: Arc<Device>
}

```

Figure 5.2.3.2, Shows render struct.

```

pub fn setup_gbuffer_pass(
    graph: &mut RenderGraph,
    renderer: &VulkanRenderer,
    gbuffer_position: TextureId,
    gbuffer_normal: TextureId,
    gbuffer_albedo: TextureId,
    gbuffer_pbr: TextureId,
) {
    graph
        .add_pass_from_desc(
            "gbuffer_pass",
            PipelineDesc::builder()
                .vertex_path("assets/shaders/gbuffer.vert")
                .fragment_path("assets/shaders/gbuffer.frag")

```

```

        .default_primitive_vertex_bindings()
        .default_primitive_vertex_attributes(),
    )
    .write(gbuffer_position)
    .write(gbuffer_normal)
    .write(gbuffer_albedo)
    .write(gbuffer_pbr)
    .external_depth_attachment(renderer.depth_image.clone(),
vk::AttachmentLoadOp::CLEAR)
    .record_render(move |device, command_buffer, renderer, pass,
resources| {
        let pipeline = resources.pipeline(pass.pipeline_handle);

        renderer.internal_renderer.draw_meshes(
            device,
            *command_buffer,
            pipeline.pipeline_layout,
        );
    })
    .build(graph);
}

```

Figure 5.2.3.3 shows the GBuffer render pass static function.

```

pub fn build_render_graph(
    graph: &mut RenderGraph,
    device: Arc<Device>,
    base: &VulkanRenderer,
    view_data: &ViewUniformData,
    camera: &Camera,
) {
    let width = base.surface_resolution.width;
    let height = base.surface_resolution.height;
}

```

```

let (gbuffer_position, gbuffer_normal, gbuffer_albedo, gbuffer_pbr) =
    create_gbuffer_textures(*);

let shadow_map = create_shadowmap_texture(graph, device.clone());
let (cascade_matrices, cascade_depths) = shadow::setup_shadow_pass(
    *
);

let image_desc = ImageDesc::new_2d(*);

let deferred_output = graph.create_texture(*);

let image_desc = ImageDesc::new_2d(*);
let ssao_output = graph.create_texture(*);
setup_gbuffer_pass(*);
let (environment_map, irradiance_map, specular_map, brdf_lut) =
    setup_cubemap_pass(*);
setup_ssao_pass(*);
setup_deferred_pass(*);
setup_atmosphere_pass(*);
setup_present_pass(*);
}

```

Figure 5.2.3.4 shows the render graph pass setup.

These are cutouts of the core components, and how this code interacts. This allows us to create passes on when we want to and will in the future allow the code to dynamically introduce and remove these passes with flags to disable specific passes.

5.2.4 Main program and event loop

```
pub fn run(mut self: Self, mut start: E, mut update: F, mut fixed_update: G,
mut finally: H) where

    E: FnMut( &Sender<GameEvent>),
    F: FnMut( &Sender<GameEvent>, f32),
    G: FnMut( &Sender<GameEvent>, f32),
    H: FnMut( &Sender<GameEvent>),
{
    let _frame_count = 0;
    self.create_scene();
    let mut input : Input = Input::default();
    let _events : Vec<WindowEvent> = Vec::new();
    self.event_loop.run(
        |event, _elwt|{
            match event{
                Event::WindowEvent {event, .. } => match event {
                    WindowEvent::RedrawRequested=> {
                        self.camera.update(&input);
                        self.renderer.render(&mut self.graph,
&self.camera);

                        input.reset_mouse()
                    },
                    WindowEvent::CloseRequested => {
                        self.graph.clear();
                        _elwt.exit();
                    },
                    WindowEvent::Resized(resize_value) => {
                        self.renderer.resize(resize_value);
                    },
                    WindowEvent::MouseInput {..} |
WindowEvent::CursorMoved {..}| WindowEvent::KeyboardInput {..}|
WindowEvent::MouseWheel {..} => {
                        input.update(&event);
                    },
                }
            }
        }
    )
}
```



```

        _ => {

        }

    },

    Event::LoopExiting => debug!("Main program loop
exiting."),

    Event::AboutToWait =>
    {self.window.window.request_redraw();},

    _ => {}

    }

    }

    ).unwrap()

}

```

Figure 5.2.4.1, showing the main loop of the engine.

Shows the engines main loop call. This is where user defined main loop data will be placed, and where Frost's ECS components will be orchestrated and executed. The user can communicate with this in the future through closures.

5.2.5 Event Loop and user functions

```

let (event_transmitter,event_receiver) = mpsc::channel();
start(&event_transmitter.clone());

let update_transmitter = event_transmitter.clone();

let fixed_update_transmitter = event_transmitter.clone();

let finally_transmitter = event_transmitter.clone();

```

Figure 5.2.5.1 shows creation of event transmitters.

```

update( &update_transmitter, delta);
// call fixed_update fixed_update_rate times per second
while lag >= self.engine_settings.fixed_update_rate.as_secs_f32() {
    // user fixed update call
    fixed_update(&fixed_update_transmitter,
self.engine_settings.fixed_update_rate.as_secs_f32());
    lag -= self.engine_settings.fixed_update_rate.as_secs_f32();
}

```

Figure 5.2.5.2 shows the handling of update and fixed update.

```

finally(&finally_transmitter);
loop{
    let event = event_receiver.recv();
    match event {
        Ok(event) => {
            match event {
                GameEvent::Input=>{
                    // TODO DO SOMETHING
                },
                GameEvent::MoveEvent(instance, matrix)=>{
                    // TODO DO SOMETHING
                    self.renderer.internal_renderer.instances[instance].transform = matrix;
                },
                GameEvent::Spawn(_path) =>{
                    // TODO handle spawn event
                }
                GameEvent::NextFrame=>{
                    // MARK FRAME COMPLETE
                    break
                },
            }
        },
    }
}

```

```

    Err(_err) => {},
  }
}

```

Figure 5.2.5.3 shows how user communicated events are read.

```

CooperApplication::create().run(
  // creates 3 cubes
  |event_stream: &Sender<GameEvent>| {
    (0..3).into_iter().for_each(|_|{
      event_stream.send(GameEvent::Spawn("models/cube.gltf".to_string())).unwrap();
    })
  },
  |renderer_event_stream, delta|
  {
    let (s,mut r,mut t) = amount.to_scale_rotation_translation();
    t.x = 5. * f32::cos(f32::to_radians(counter));
    t.z = 5. * f32::sin(f32::to_radians(counter));

    r = look_at(t, CENTRE);
    amount = Mat4::from_scale_rotation_translation(s,r, t);
    renderer_event_stream.send(GameEvent::MoveEvent(0,amount)).unwrap();

    counter += 20. * delta;
  },
  |renderer_event_stream, _delta|
  {
    let (s,mut r,mut t) =
fixed_amount.to_scale_rotation_translation();
    t.x = 5.;
    t.z = 5.;
    t.y = 1. * f32::sin(f32::to_radians(fixed counter));
    println!("fixed update: {}", t.y);
    r = look_at(t, CENTRE);
  }
);

```

```

        fixed_amount = Mat4::from_scale_rotation_translation(s,r, t);
        renderer_event_stream.send(GameEvent::MoveEvent(3,fixed_amount))
        .unwrap();
        fixed_counter += 50. * _delta;
    },
    |event_stream| {
        event_stream.send(GameEvent::NextFrame).unwrap();
    }
);

```

Figure 5.2.5.4 shows how user communicated events are read.

While there is a lot of code here, we can see the benefits of this code. We can communicate in a way where memory is never shared – only communicated to the engine user. We supply closures with unmoved data (this means that the caller retains ownership in the rust context). In future iterations these functions will work in tandem with the ECS created in Frost.

This style of communication provides guarantees a message will be consumed, transfers the ownership of the message to the thread which needs to assume ownership and maintains stack speed of access and modification. A comprehensive and extensive event system allow communication to many parts of the engine without worrying about concepts like race conditions. T

6. Software Development

This chapter highlights the process of the development of the engine's features. There are three sections described, the application layer (titled Cooper), the rendering layer (titled Lynch) and the state management layer (titled Frost).

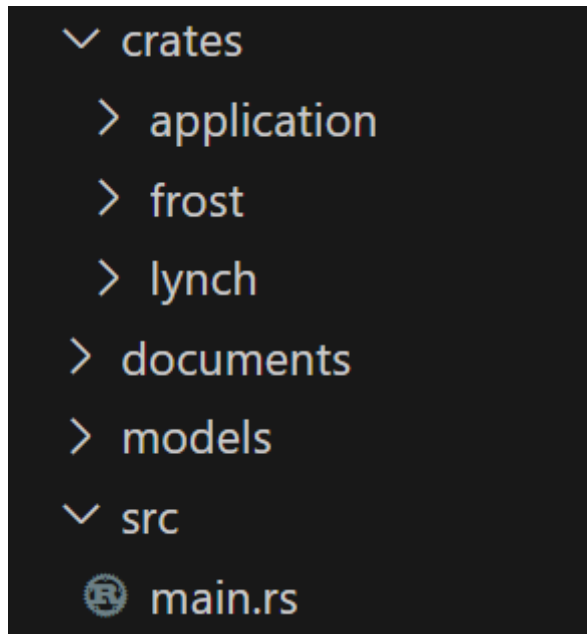


Figure 35, Layout of Project

6.1 Cooper

Cooper is the application layer, meaning it handles creating the window and is the main interface that a user of the engine would interact with, and is the “orchestration” layer for this project.

6.1.1 The Cooper Application

The application created contains references to the window handle, the renderer, the render graph, the ECS and a camera:

```
pub struct CooperApplication {  
    window: Window,  
    pub renderer: VulkanRenderer,  
    graph: RenderGraph,  
    pub camera: Camera,  
    event_loop: EventLoop<()>,  
    engine_settings: EngineSettings,  
}
```

The application is then instantiated using the builder pattern, which is used in many places throughout the code base.

The window struct is a wrapper for [Winit](#), and only contains the handle, to allow for more dynamic borrowing.

6.1.2 Update

In a game engine, the update function is commonly responsible for handling various game logic and state updates. This can include things like player input processing, updating game object positions and animations, managing AI behavior, handling collisions, and more.

```
update(&update_transmitter, render_statistics.full_render_time);
```

6.1.3 Fixed Update

In game development, a fixed update is a mechanism used to ensure consistent behavior across different hardware configurations and frame rates. Unlike regular updates, which are tied to the rendering frame rate and can vary depending on the system's performance, fixed updates occur at a fixed interval regardless of the frame rate.

```
while lag >= self.engine_settings.fixed_update_rate.as_secs_f32()
    || physics_control.step
{
    if let Some(onfixed update system) =
        self.systems.get_mut(&Schedule::OnFixedUpdate) {
        for onfixed update system in onfixed update system.iter_mut() {
            onfixed update system
                .as_mut()
                .run_fixed( &mut world,
                    self.engine_settings.fixed_update_rate.as_secs_f32()) .unwrap();
        }
    }
}
```

The loop runs if the accumulated lag (representing the time between fixed updates) is greater than or equal to the fixed update rate defined in `self.engine_settings.fixed_update_rate.as_secs_f32()` or if `physics_control.step` is true. If `physics_control.should_update_physics()` returns true, the physics simulation is updated using the `physics_system.run()` method. This ensures that the physics simulation progresses according to a fixed time step, regardless of the frame rate.

```
let mut search = world.search::(&RigidBody, >()).unwrap();
for (i, rb) in search.iter().enumerate() {
    if i == rigidbody_list.len() {
        rigidbody_list.push(rb.clone())
    } else {
        rigidbody_list[i] = rb.clone()
    }
}
```

The positions and orientations of graphical objects (represented by `gfx`) are updated to match the corresponding rigid body's transformation after the physics simulation step. Systems registered for fixed update (`Schedule::OnFixedUpdate`) are executed. These systems perform operations that need to occur at fixed intervals, such as AI behavior updates, physics interactions, or other game logic.

6.2 Lynch

The rendering engine used in this project. The *lynch* module is designed to handle complex rendering tasks, with a focus on flexibility and memory safety. The module includes a variety of submodules, such as *mesh_loader*, *render_graph*, *vulkanrenderer*, and *vulkan*, each responsible for different aspects of the rendering process. It also includes a *render_tools* submodule, which contains a collection of tools for specific rendering tasks like atmospheric effects, deferred rendering, forward rendering, and shadow mapping. The lynch module is built utilizing Vulkan, a modern graphics API, for efficient hardware-accelerated rendering.

The library supports loading of GLTF models, which include materials, textures and models within them, and there are many examples provided by Intel and Nvidia to test the ability of the renderer to load complex scenes and models.

6.2.1 Vulkan Device

A Vulkan device represents a connection to the graphics processing unit (GPU), serving as the primary means through which we will interact with the GPU. Once a Vulkan instance is created and a physical device is selected, the Vulkan device is initialized to create a logical interface for rendering tasks. This device handles a wide array of tasks, including memory management, command buffer execution, and synchronization. It facilitates the creation and management of various Vulkan resources like queues, images, buffers, and shaders, enabling applications to submit commands and perform computations or rendering operations.

Our `Device` is defined like this,

```
pub struct Device {
    pub ash_device: ash::Device,
    pub physical_device: vk::PhysicalDevice,
    pub queue: vk::Queue,
    pub cmd_pool: vk::CommandPool,
    pub setup_cmd_buf: vk::CommandBuffer,
    pub device_memory_properties: vk::PhysicalDeviceMemoryProperties,
    pub queue_family_index: u32,
    pub gpu_allocator: Arc<Mutex<gpu_allocator::vulkan::Allocator>>,
    pub debug_utils: Option<ash::extensions::ext::DebugUtils>,
}
```

And is created using the standard means. When we enable device features, we enable `ExtDescriptorIndexingFn` and `KhrDynamicRenderingFn`. Vulkan extensions and features are additional functionalities provided by the Vulkan graphics API beyond its core specifications. `ExtDescriptorIndexingFn` is used to allow shaders to access large descriptor sets with dynamic

indexing, reducing the overhead associated with binding individual descriptors, resulting in improved performance.(70) KhrDynamicRenderingFn enables dynamic rendering, allowing our application to dynamically adjust rendering parameters and techniques during runtime. (71)

```
let device_extension_names_raw = vec![
    Swapchain::name().as_ptr(),
    vk::ExtDescriptorIndexingFn::name().as_ptr(),
    vk::KhrDynamicRenderingFn::name().as_ptr(),
    vk::KhrMaintenance1Fn::name().as_ptr(),
    vk::KhrMaintenance2Fn::name().as_ptr(),
    vk::KhrMaintenance3Fn::name().as_ptr(),
];
```

While this was planned to be abstracted out to a function, allocating the name to pointers correctly proved challenging, as Rust eagerly frees memory the moment it exits scope.(72)

We then enable those features on the card itself.

```
let mut descriptor_indexing_features =
    vk::PhysicalDeviceDescriptorIndexingFeaturesEXT::default();
let mut buffer_device_address_features =
    vk::PhysicalDeviceBufferDeviceAddressFeaturesKHR::default();
let mut scalar_block_layout_features =
    vk::PhysicalDeviceScalarBlockLayoutFeatures::default();
let mut dynamic_rendering_features =
    vk::PhysicalDeviceDynamicRenderingFeatures::default();

let mut features2 = vk::PhysicalDeviceFeatures2::builder()
    .push_next(&mut descriptor_indexing_features)
    .push_next(&mut buffer_device_address_features)
    .push_next(&mut scalar_block_layout_features)
    .push_next(&mut dynamic_rendering_features)
    .build();
instance.get_physical_device_features2(physical_device, &mut features2);
```

```
let gpu_allocator = Allocator::new(&AllocatorCreateDesc {
    instance: instance.clone(),
    device: device.clone(),
    physical_device,
    debug_settings: AllocatorDebugSettings {
        log_leaks_on_shutdown: true,
        log_memory_information: true,
        log_allocations: true,
        log_stack_traces: true,
        ..Default::default()
    },
    buffer_device_address: true,
    allocation_sizes: Default::default()
});
```



```

    })
    .expect("Failed to create GPU allocator");

```

A GPU allocator is then created. This is provided by the library `gpu_allocator`, and is a useful tool to manage memory allocation within Vulkan. (73)

6.2.2 Vulkan Context

In Vulkan, the term "context" isn't officially used in the same way it is in OpenGL; however, when referring to Vulkan context, we're typically talking about the combination of several key objects and states that enable interaction with the GPU. The instance is the first object we create when initializing a Vulkan application, it holds information about the application itself and the Vulkan extensions and layers that are available and enabled. It's the connection between our engine and the Vulkan library. I utilize an Arc surrounding the Device. Arc is an "atomic reference count" which is a practical way for the device to be shared safely across threads. When the Arc is dropped (goes out of scope), a drop function is called, which checks the number of references still existing – if this reaches 0, the memory is freed. This is very similar to garbage collected behaviour but maintains predictability of clean up (after the engine has stopped). The surface is a reference to the operating system level handle which we will later present frames to.

```

pub struct VkContext {
    _entry: vk::Entry,
    instance: vk::Instance,
    surface: vk::Surface,
    surface_khr: vk::SurfaceKHR,
    device: Arc<Device>,
}

```

6.2.3 Render Pass

A render pass in this project is a datatype which contains instructions used to render a specific shader, as well as the data designed to be passed to it. The associated pipeline handle is a value which will be created later [here](#). The associated name is used both for debugging (within the Vulkan framework) and to retrieve the pass itself and the pipeline for the renderer, so it is not unnecessarily recreated. The `render_func` is the closure which will send render commands to the device along its command buffer, as well the referenced attachment to write to or buffer to read from. The `depth_attachment` is an optional attachment which describes a depth image used in certain parts of render pipelines.

```

pub struct RenderPass {
    pub name: String,
    pub is_pres_pass: bool,
    pub pipeline_handle: PipelineId,
    pub render_func: Option<

```

```

        Box<dyn Fn(&Device, &vk::CommandBuffer, &VulkanRenderer, &RenderPass,
&GraphResources)>,
        >,
        pub writes: Vec<Attachment>,
        pub reads: Vec<Resource>,
        pub depth_attachment: Option<DepthAttachment>,
        pub uniforms: HashMap<String, (String, UniformData)>,
        pub read_resources_descriptor_set: Option<DescriptorSet>,
        pub uniform_descriptor_set: Option<DescriptorSet>,
        pub uniform_buffer: Option<BufferId>,
        pub extra_barriers: Option<Vec<(BufferId, AccessType)>>,
        device: Arc<Device>,
    }

```

A render pass is created using the builder pattern, and its interface will be explained during the process showing the creation of actual render passes [here](#).

6.2.4 Render Graph

The `RenderGraph` describes a resource which contains a list of render passes to be executed, and is the main interface for starting new frames, transitioning resources, presenting images, swapping the swap chain, and adding resources to the renderer to be rendered. Currently the graph also holds the descriptor set reference to the camera, although in the future this would need to be moved to allow multi view rendering to be implemented in a more idiomatic manner.

```

pub struct RenderGraph {
    pub passes: Vec<Vec<RenderPass>>,
    pub resources: GraphResources,
    pub descriptor_set_camera: Vec<DescriptorSet>,
    pub pipeline_descs: Vec<PipelineDesc>,
    pub current_frame: usize,
    pub device: Arc<Device>,
}

```

`GraphResources` contains the resources used by the graph during stages of rendering. Each of these resource types is explained in the document in the related sections, [buffers](#), [textures](#) and [pipelines](#).

```

pub struct GraphResources {
    pub buffers: Vec<Buffer>,
    pub textures: Vec<Texture>,
    pub pipelines: Vec<Pipeline>,
}

```

6.2.5 Descriptor Set

Descriptor sets are collections of descriptors that define resources accessible to shaders within a rendering pipeline. These descriptors specify resources such as buffers, images, and samplers, which

are bound to shader stages (like vertex, fragment, or compute shaders) to enable rendering or compute operations. It's important in a game engine for Descriptor Sets to be malleable and as dynamically assignable as possible.

```
pub struct DescriptorSet {
    pub handle: vk::DescriptorSet,
    pub pool: vk::DescriptorPool,
    layout: vk::DescriptorSetLayout,
    binding_map: BindingMap,
    device: Arc<Device>,
}
```

To use descriptor sets, we first allocate them from a descriptor pool with specified bounds as well as types and the number of descriptors needed. We then populate these sets with specific resources by updating their contents. Once populated, we will bind the descriptor sets to the appropriate shader stages before rendering operations.

6.2.6 Pipeline Descriptions

A `PipelineDescription` refers to the configuration and setup of various stages involved in the rendering process. This includes the shader stages, such as vertex and fragment shaders, vertex binding descriptions (contains binding (location of vertex buffer in memory), stride (size of vertex pushed) and input rate), attribute descriptions (location, binding format and offset) and the necessary attachment formats. Colour attachment formats are the format of the images associated with a pipeline (those are described [here](#)).

```
pub struct PipelineDesc {
    pub vertex_path: Option<&'static str>,
    pub fragment_path: Option<&'static str>,
    pub vertex_input_binding_descriptions:
Vec<vk::VertexInputBindingDescription>,
    pub vertex_input_attribute_descriptions:
Vec<vk::VertexInputAttributeDescription>,
    pub colour_attachment_formats: Vec<vk::Format>,
    pub depth_stencil_attachment_format: vk::Format,
}
```

To allow pipelines to be more dynamically related to the shaders which will be bound to them, a pipeline description builder pattern is used:

```
pub struct PipelineDescBuilder {
    descriptor: PipelineDesc,
}
```

Which allows pipelines to be created intuitively using this pattern:

```
PipelineDesc::builder()
    .vertex_path("assets/shaders/gbuffer.vert")
```

```
.fragment_path("assets/shaders/gbuffer.frag")
.default_primitive_vertex_bindings()
.default_primitive_vertex_attributes()
.build();
```

This example shows the inclusion of “default_primitive_vertex_bindings” and “default_primitive_vertex_attributes”. These are bindings taking from the [GPU Vertex context described later](#). These map the vertex’s attributes to relevant Vulkan format values. This lets the shader know where to look for data within the vertex buffer, and how big to expect this each piece of data.

```
pub fn get_vertex_input_attribute_descriptions() ->
Vec<vk::VertexInputAttributeDescription> {
    vec![
        vk::VertexInputAttributeDescription {
            location: 0,
            binding: 0,
            format: vk::Format::R32G32B32A32_SFLOAT,
            offset: offset_of!(Vertex, position) as u32,
        },
    ],
}
```

This stub shows how we assign a location, binding position, format and offset to this vertex position. Using format: `vk::Format::R32G32B32A32_SFLOAT` means that the format of the location binding is a vector of 4 32-bit float values, which in Rust we represent as a `Vec4<f32>`. `offset_of!` Allows us to, at compile time, receive the memory alignment position of the position attribute within the struct.

```
pub struct Pipeline {
    pub handle: vk::Pipeline,
    pub pipeline_layout: vk::PipelineLayout,
    pub descriptor_set_layouts: Vec<vk::DescriptorSetLayout>,
    pub shader_reflection: ShaderReflect,
    pub pipeline_desc: PipelineDesc,
    pub pipeline_type: PipelineType,
}
```

The pipeline itself is a handle to the pipeline created, as well as related layouts, the description of the pipeline, its type (almost always graphical) and then a reflection of the shader used to create it. The `rpsirv_reflection` crate is used to dynamically read shaders, and to parse necessary data from them, so that the `PipelineDesc` can be populated appropriately. (74)

6.2.7 Buffers

A buffer is a region of memory used to store raw data, such as vertex data, index data, uniform data, or other types of data required for rendering or computation. This is our main mechanism for communicating material, indices, and vertex data to the graphics card. This struct serves as a

comprehensive abstraction for managing buffers, enabling seamless integration and manipulation of buffer resources within Vulkan rendering and compute pipelines.

```
pub struct Buffer {
    pub vk_buffer: vk::Buffer,
    pub allocation: Allocation,
    pub memory_req: vk::MemoryRequirements,
    pub memory_location: MemoryLocation,
    pub size: u64,
    pub debug_name: String,
    device: Arc<Device>,
}
```

The buffer can have been data modified by any type that enables copying, and can have data sent to its location by interfacing with this function:

```
pub fn update_memory<T: Copy>(&mut self, data: &[T])
```

This uses unsafe code to copy the data within `data` which is some type `T` which implements the built-in `Copy` trait. We start by getting a raw pointer to the data, and the length of that data (C through C style sizeof call):

```
let src = data.as_ptr() as *const u8;
let src_bytes = data.len() * mem::size_of::<T>();
```

We then create a staging buffer on the device, with TRANSFER_SRC type, and we copy the data from the source (the data pointer) to the staging buffer:

```
let staging_buffer = Buffer::create_buffer(
    vk::BufferUsageFlags::TRANSFER_SRC,
    MemoryLocation::CpuToGpu
);
let dst = staging_buffer.allocation.mapped_ptr().unwrap().as_ptr() as *mut u8;
let dst_bytes = staging_buffer.allocation.size() as usize;
copy_nonoverlapping(src, dst, std::cmp::min(src_bytes, dst_bytes));
```

We then send a command to the device copying the data from this staging buffer (CPU accessible) to the destination Vulkan buffer (GPU only accessible (faster)). Once complete, we deallocate the staging buffer. (75)

```
self.device.ash_device.cmd_copy_buffer(
    cb,
    staging_buffer.vk_buffer,
    self.vk_buffer,
    &[regions],
);
self.device
    .ash_device
```

```
.destroy_buffer(staging_buffer.vk_buffer, None);
```

This process is used anytime we add or remove data from the buffer but is an expensive operation. Copying data from the CPU to the GPU is ideally a task we want to minimize, so the code wrapping this will need to make sure the buffer's memory is only updated when needed.

6.2.8 Images

In Vulkan, images represent two or three-dimensional arrays of texels (texture elements), which can be used for various purposes such as textures, framebuffers, or attachments for rendering operations. The `ImageDesc` struct provides a description of an image's properties, including its dimensions (width, height, depth), number of array layers, format, type aspect, usage flags, and the number of mip levels.

```
pub struct ImageDesc {  
    pub width: u32,  
    pub height: u32,  
    pub depth: u32,  
    pub array_layers: u32,  
    pub format: vk::Format,  
    pub image_type: ImageType,  
    pub aspect_flags: vk::ImageAspectFlags,  
    pub usage: vk::ImageUsageFlags,  
    pub mip_levels: u32,  
}
```

The Image struct encapsulates an image object along with its associated resources. It includes fields such as image representing the Vulkan image object, image_view providing a view of the image, layer_views containing views for each layer of the image (useful for array textures or cube maps), device_memory storing memory allocated for the image, current_layout indicating the current layout of the image, desc containing the description of the image properties, debug_name for debugging purposes, and device referencing the Vulkan device used to interop.

```
pub struct Image {  
    pub image: vk::Image,  
    pub image_view: vk::ImageView,  
    pub layer_views: Vec<vk::ImageView>,  
    pub device_memory: vk::DeviceMemory,  
    pub current_layout: vk::ImageLayout,  
    pub desc: ImageDesc,  
    pub debug_name: String,  
    pub device: Arc<Device>,  
}
```

The `new_from_desc` function is responsible for creating a new image in a Vulkan application based on the provided ImageDesc description. The function begins by defining the parameters

required to create a Vulkan image using the `vk::ImageCreateInfo` structure (shortened for brevities sake):

```
let image_create_info = vk::ImageCreateInfo {
    image_type: vk::ImageType::TYPE_2D,
    format: desc.format,
    extent: vk::Extent3D {
        width: desc.width,
        height: desc.height,
        depth: 1,
        .....
```

After defining the image creation parameters, the function retrieves memory requirements for the image using ``get_image_memory_requirements()``. We then find a suitable memory type index for the image's memory allocation based on the required memory properties

(``vk::MemoryPropertyFlags::DEVICE_LOCAL``). Once the memory type index is found, memory is allocated for the image using ``device.ash_device.allocate_memory()``.

```
let image = device.create_image(&image_create_info, None)
let image_memory_req = device.ash_device.get_image_memory_requirements(image);
let device_memory = device.allocate_memory(&image_allocate_info, None)
device.ash_device.bind_image_memory(image, device_memory, 0)
let image_view = Image::create_image_view(
    &device,
    image,
    desc.format,
    desc.aspect_flags,
    view_type,
    0,
    desc.array_layers,
    desc.mip_levels,
);
```

Images play a pivotal role in presenting the outputs of render passes to either other passes, or when they are finally transitioned to presentation layers during the last pass.

6.2.9 Textures

A texture is a type of image used primarily for storing and representing graphical data, such as colours, patterns, or surface properties, which can be accessed and sampled by shaders during rendering operations. Textures are represented as images with specific formats (e.g., RGBA8 for 32-bit colour images) and dimensions (width, height, depth). They can contain one or more mipmap levels representing the same image at different levels of detail, enabling efficient rendering of textures at various distances from the viewer. Textures are used for a variety of purposes in the engine, for colour textures (surface patterns and images), normal maps (for efficient bump and


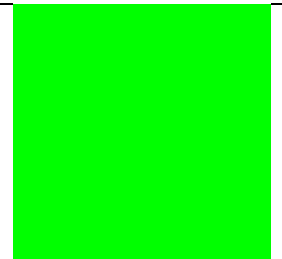
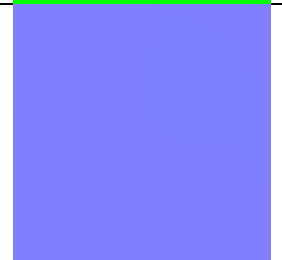
surface details), depth textures (used for shadow mapping and stencilling), cube maps (used for skyboxes and potentially reflections in the future) and texture arrays.

```
pub struct Texture {  
    pub device: Arc<Device>,  
    pub image: Image,  
    pub sampler: vk::Sampler,  
    pub descriptor_info: vk::DescriptorImageInfo,  
}
```

Textures are accessed and sampled by shaders during rendering operations to retrieve texel (texture element) data for use in calculations, such as lighting, shading, and texture mapping. Sampling involves specifying texture coordinates within the range [0,1] to retrieve texel values from the texture image.

Textures are bound to texture units within the graphics pipeline, allowing shaders to access them during rendering. Each texture unit can hold one or more textures, depending on the hardware and implementation.

We also have “default” textures which can be used in place of defined textures when needed for placeholder data:

White Texture:	
Default roughness map (metallic):	
Default normal map:	

These textures are loaded from here and added to the devices memory so they can be accessed by pipelines:


```

let default_diffuse_map = Texture::load(device.clone(),
"assets/textures/def/white_texture.png");
let default_normal_map = Texture::load(device.clone(),
"assets/textures/def/flat_normal_map.png");
let default_metallic_roughness_map = Texture::load(device.clone(),
"assets/textures/def/metallic_roughness.png");

self.default_diffuse_map_index = self.add_bindless_texture(&device,
&default_diffuse_map);
self.default_normal_map_index = self.add_bindless_texture(&device,
&default_normal_map);
self.default_metallic_roughness_map_index =
    self.add_bindless_texture(&device, &default_metallic_roughness_map);

```

6.2.10 The View Uniform

In Vulkan, a uniform is a type of resource that provides a way to pass constant data from the CPU to the GPU shaders. Uniforms are typically used to pass data that remains constant across a single draw call or dispatch operation, such as transformation matrices, material properties, or lighting information. In Vulkan, uniforms are commonly accessed in shaders through uniform buffer objects (UBOs). (76)

The view uniform is a type of uniform data that typically contains the view matrix or other related information that represents the camera's position and orientation in the scene. The view matrix is used to transform vertices from world space to view (or camera) space. The view matrix is a transformation matrix that represents the inverse of the camera's transformation in the world. It's used to transform the coordinates of objects in the scene from world space to view space, aligning them relative to the camera's perspective. (77)

```

#[repr(C)]
pub struct ViewUniformData {
    pub view: glam::Mat4,
    pub projection: glam::Mat4,
    pub inverse_view: glam::Mat4,
    pub inverse_projection: glam::Mat4,
    pub eye_pos: glam::Vec3,
    pub sun_dir: glam::Vec3,
    pub viewport_width: u32,
    pub viewport_height: u32,
}

```

This view uniform can then be represented within shaders with this layout. It's important to note the use of `#[repr(C)]`. Rust, default struct layout is not specified to be consistent across compilations or platforms; the compiler may reorder fields for optimization purposes. However, when interfacing with graphics APIs, which are typically C-based, we need a predictable, consistent memory layout for data structures to ensure the data is correctly understood by the GPU. Vulkan has specific

requirements for the alignment of data in buffers. `#[repr(C)]` ensures that the struct's fields are aligned in a way that is compatible with C (and by extension, compatible with the graphics APIs), including the correct padding between fields when necessary. (78)

```
layout (std140, set = 1, binding = 0) uniform view_uniform
{
    mat4 view;
    mat4 projection;
    mat4 inverse_view;
    mat4 inverse_projection;
    vec3 eye_pos;
    vec3 sun_dir;
    uint viewport_width;
    uint viewport_height;
}
```

6.2.11 GPU Vertex

A vertex represents a point in space, and the mechanism for sending this to the GPU was described within the section on [buffers](#). This data will be loaded in from models (gLTF only) and populated with representative data. Vertices simply represent a point in space.

```
#[repr(C)]
pub struct Vertex {
    pub pos: Vec4,
    pub normal: Vec4,
    pub uv: Vec2,
    pub color: Vec4,
    pub tangent: Vec4,
}
```

Here is how this vertex is represented within the shader itself, and as we can see the alignment of variables here matches where position, normal, uv, color and tangent match both datatypes and sizes. (Vec4 stores data within f32, or a 32-bit float, the same used within Spir-V shaders.)

```
struct Vertex
{
    vec4 pos;
    vec4 normal;
    vec2 uv;
    vec4 color;
    vec4 tangent;
};
```

These vertices are then stored within a shader storage buffer object which contains them defined like:

```
layout (std430, set = 0, binding = 1) readonly buffer verticesSSBO
{
```

```
Vertex vertices[];
} verticesSSBO[];
```

Anytime vertex data needs to be accessed, this bindless description is passed into the function, and is indicated at the pipeline description stage. When iterating over this data within the shader, we can access the specific vertex like so:

```
Vertex vertex = verticesSSBO[vertex_buffer].vertices[gl_VertexIndex];
```

6.2.12 GPU Materials

Vulkan shaders, written in GLSL, define how an object's appearance is calculated. There are two main types of shaders: vertex shaders, which transform vertices from object space to screen space, and fragment shaders (also known as pixel shaders), which calculate the color of individual pixels on the object's surface. A Material in this project provides diffuse, normal, metallic / roughness and a base colour. The different maps (or textures) are identified by their handle within the GPU. This data structure is then sent directly to the shader.

```
#[repr(C)]
struct GpuMaterial {
    diffuse_map: u32,
    normal_map: u32,
    metallic_roughness_map: u32,
    occlusion_map: u32,
    base_color: Vec4,
    metallic_factor: f32,
    roughness_factor: f32,
    padding: [f32; 2],
}
```

The shader represents this data in this format:

```
struct Material
{
    uint diffuse_map;
    uint normal_map;
    uint metallic_roughness_map;
    uint occlusion_map;
    vec4 base_color_factor;
    float metallic_factor;
    float roughness_factor;
    vec2 padding;
};
```

It's important to again note the layout. This "default shader" can further be expanded to enable more features and store more relative data. The related maps are by default assigned their materials IF the model provided does not indicate this data. Materials are stored in another buffer such that they

can be accessed via their assigned index by our deferred shader, or some forward shader if that were to be used.

```
layout (scalar, set = 0, binding = 3) readonly buffer MaterialsSSBO
{
    Material materials[];
} materialsSSBO;
```

6.2.13 GPU Mesh

A GPU Mesh, in this project, is simply a pointer to a vertex, index and material buffer, which contains the data used by the model to be rendered. A mesh comes from a loaded model, created by the engine user, and defines this data. The buffer values are assigned when the related buffer is created on the graphics card as described in the [buffer section](#).

```
#[repr(C)]
struct GpuMesh {
    vertex_buffer: u32,
    index_buffer: u32,
    material: u32,
}
```

And as the same with the rest, the mesh is then described shader side with a matching structure.

```
struct Mesh
{
    uint vertex_buffer;
    uint index_buffer;
    uint material;
};
```

And these are stored on their own shader storage buffer object, which again get passed to any render pass which needs access to vertex / index / mesh data.

```
layout (scalar, set = 0, binding = 4) readonly buffer MeshesSSBO
{
    Mesh meshes[];
} meshesSSBO;
```

6.2.14 Renderer Internals

The renderer has an internal structure, which holds references to some of the required data which will be rendered by the program. We hold references to instances of models, the created buffers for materials and meshes, some default textures and maps as well as image counts. This differs from Graph Resources, which are the pointers to the buffers used within render passes.

```
pub struct RendererInternal {
    pub bindless_descriptor_set_layout: vk::DescriptorSetLayout,
    pub bindless_descriptor_set: vk::DescriptorSet,
```

```

pub instances: Vec<ModelInstance>,
pub gpu_materials_buffer: Buffer,
pub gpu_meshes_buffer: Buffer,
gpu_materials: Vec<GpuMaterial>,
gpu_meshes: Vec<GpuMesh>,
default_diffuse_map_index: u32,
default_normal_map_index: u32,
default_occlusion_map_index: u32,
default_metallic_roughness_map_index: u32,
next_bindless_image_index: u32,
next_bindless_vertex_buffer_index: u32,
next_bindless_index_buffer_index: u32,
}

```

This data structure is during the actual renderer's initialization, and is the interface used behind the scenes to manage vertex and material buffers, to load and allocate memory for resources used within rendering and is the and any necessary data to the renderer. This handles the drawing of meshes, wherein push constants are sent to the device during rendering. This is where the models transform matrix is applied, and where the vertex, and index buffers are bound.

```

pub fn draw_meshes(
    &self,
    device: &Device,
    command_buffer: vk::CommandBuffer,
    pipeline_layout: vk::PipelineLayout,
) {
    for instance, in &self.instances {
        for (i, mesh) in instance.model.meshes.iter().enumerate() {
            device.cmd_push_constants(
                command_buffer,
                pipeline_layout,
                (
                    instance.transform * instance.model.transforms[i],
                    glam::Vec4::new(1.0, 0.5, 0.2, 1.0),
                    mesh.gpu_mesh,
                    [0; 3],
                ),
                device.device().cmd_bind_vertex_buffers(

                    &[mesh.primitive.vertex_buffer.vk_buffer],
                    &[0],
                );
                device.device().cmd_bind_index_buffer(
                    mesh.primitive.index_buffer.vk_buffer,
                    vk::IndexType::UINT32,
                );
                device.device().cmd_draw_indexed(

```

```

        command_buffer,
        mesh.primitive.indices.len() as u32
    );
}

```

This functionality will typically be called within specific render passes themselves and cannot be called outside of that context (running draw or any command buffer commands outside of a render pass results in undefined behaviour and from many frustrations, crashes).

6.2.15 The Actual Renderer

```

pub struct VulkanRenderer {
    pub vk_context: VkContext,
    pub sync_frames: Vec<Frame>,
    pub command_pool: vk::CommandPool,
    pub image_count: u32,
    pub present_images: Vec<Image>,
    pub depth_image: Image,
    pub surface_format: vk::SurfaceFormatKHR,
    pub surface_resolution: vk::Extent2D,
    pub swapchain: vk::SwapchainKHR,
    pub swapchain_loader: ash::extensions::khr::Swapchain,
    pub debug_utils_messenger: Option<vk::DebugUtilsMessengerEXT>,
    pub internal_renderer: RendererInternal,
    pub current_frame: usize,
    pub num_frames_in_flight: u32,
    pub swapchain_recreate_needed: bool,
    pub camera_uniform_buffer: Vec<Buffer>,
    pub view_data: ViewUniformData,
    pub last_frame_end: Instant,
    pub gui_renderer: imgui_rs_vulkan_renderer::Renderer,
}

```

The renderer is what the application itself interfaces with to start and end frames. It contains all data necessary to complete a rendering task, and a reference to the internal renderer mentioned before. It is also the interface which we pass GUI data, and camera data to. This initializes data structures and is passed to the Graph itself to run rendering tasks.

6.2.16 Render Passes

Render passes are complex and took a lot of debugging to figure out. RenderDoc was used during this process as it provides a comprehensive platform for capturing and analysing frame-by-frame rendering in Vulkan (and other APIs). It allows me to inspect each rendering step, view states resources and debug shaders. (79)

Shadow Pass

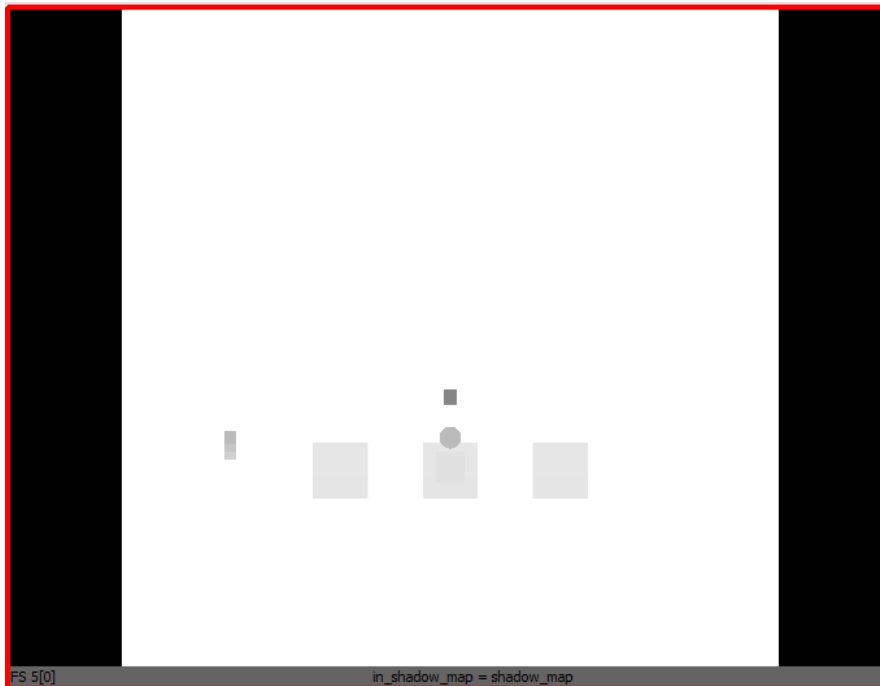


Figure 36, Generated Shadow Map

Shadows were created using the deferred rendering technique and involves rendering the scene from the perspective of the light source(s) to create a depth map. This depth map represents the distances from the light source to the surfaces in the scene. Later, during the main rendering pass, this shadow map is sampled to determine whether a pixel on a surface is in shadow or not, and to calculate the appropriate lighting effect. The technique used is called "cascaded shadow mapping" (CSM). It is used to enhance the quality and accuracy of shadows by dividing the view frustum into multiple cascades and rendering separate shadow maps for each cascade.

The shadow map pass is constructed on the render graph and is completed first. The number of cascades is dynamic but currently set at 4.

graph

```
.add_pass_from_desc(  
    format!("shadow_pass_{i}").as_str(),  
    PipelineDesc::builder()  
        .vertex_path("assets/shaders/shadow.vert")  
        .fragment_path("assets/shaders/shadow.frag")  
        .default_primitive_vertex_bindings()  
        .default_primitive_vertex_attributes(),  
)  
    .uniforms("cascade_view_projection", &view_projection_matrix)  
    .depth_attachment_layer(shadow_texture, i as u32)  
    .build(graph);
```

We pass the scene vertices, and an additional uniform “cascade_view_projection” which is like the view from the camera, instead originating for the light source.

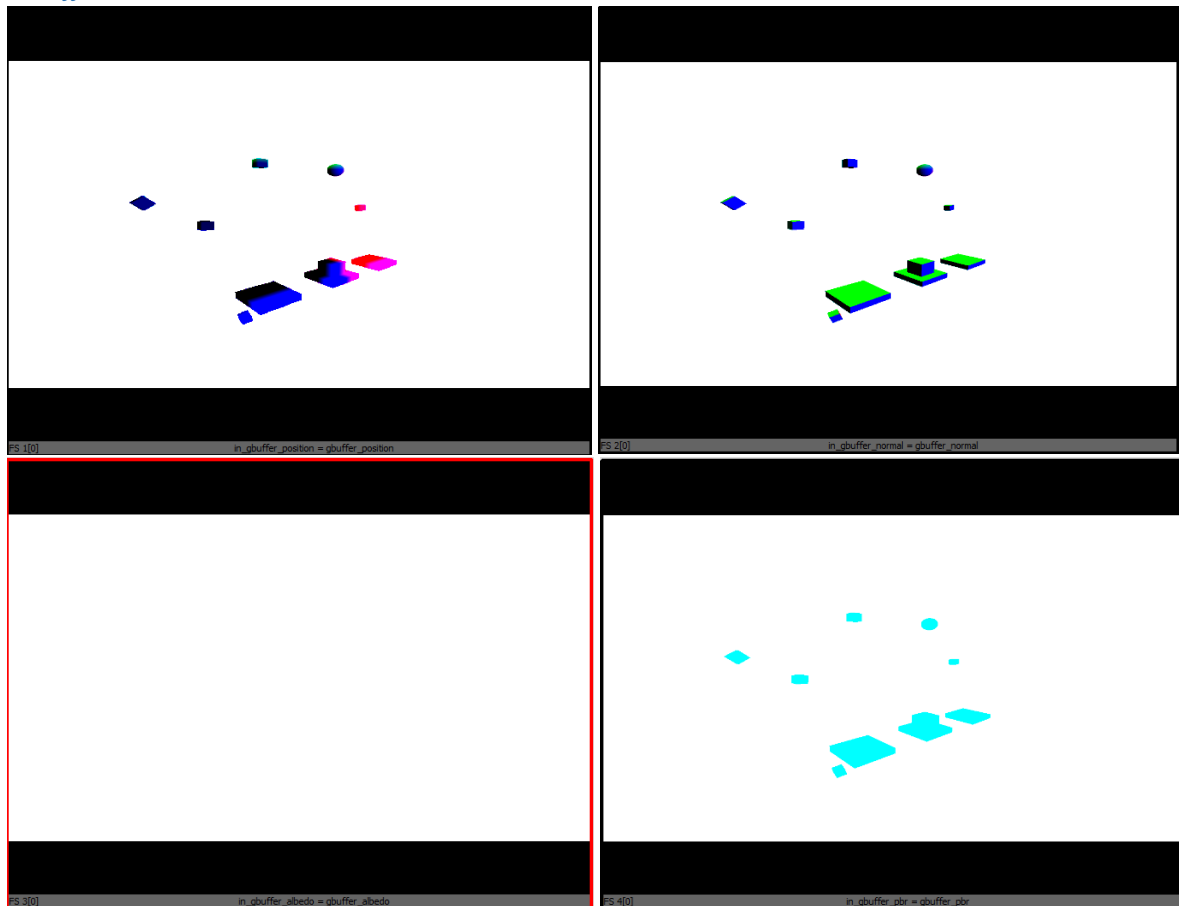
We iterate over a predefined number of cascades, calculating split distances based on the depth ranges of the view frustum. For each cascade, we construct a light view-projection matrix using an orthographic projection matrix centred around the frustum's bounding volume. These matrices are then used to render depth maps for each cascade, updating the render graph with appropriate render passes. Additionally, we calculate and return the cascade matrices and split depths for later use in rendering shadows.

```
pub fn calculate_cascade_splits(near_clip: f32, far_clip: f32) -> [f32;
SHADOW_TEXTURE_CASCADE_NO] {
    let mut cascade_splits = [0.0; SHADOW_TEXTURE_CASCADE_NO];
    let clip_range = far_clip - near_clip;
    let cascade_split_lambda = 0.927;

    for i in 0..SHADOW_TEXTURE_CASCADE_NO {
        let p = (i + 1) as f32 / SHADOW_TEXTURE_CASCADE_NO as f32;
        let log = near_clip * (far_clip / near_clip).powf(p);
        let uniform = near_clip + clip_range * p;
        cascade_splits[i] = (cascade_split_lambda * (log - uniform) + uniform
- near_clip) / clip_range;
    }
    cascade_splits
}
```

`calculate_cascade_splits` to compute the cascade split distances based on the near and far clipping planes of the camera's perspective projection. We calculate split distances that distribute shadow map resolution across the depth range of the scene, enhancing shadow quality for both near and distant objects.

G-Buffer Pass



`setup_gbuffer_pass` is responsible for configuring the rendering pipeline to generate a G-buffer, which is used in deferred rendering to store information about the scene geometry. We add a render pass to the render graph with specified vertex and fragment shader paths, defining the necessary bindings for vertex data and attributes. The function sets output layouts for the G-buffer textures, including positions, normals, albedo, and physically-based rendering (PBR) parameters. Additionally, we attach an external depth attachment to the render pass, clearing it for subsequent rendering operations. Within the render pass, we record rendering commands to draw meshes using the specified pipeline layout and resources.

```
pub fn setup_gbuffer_pass(  
    graph: &mut RenderGraph,  
    renderer: &VulkanRenderer,  
    gbuffer_position: TextureId,  
    gbuffer_normal: TextureId,  
    gbuffer_albedo: TextureId,  
    gbuffer_pbr: TextureId,  
) {  
    graph  
        .add_pass_from_desc(  
            "gbuffer_pass",
```

```

        PipelineDesc::builder()
            .vertex_path("assets/shaders/gbuffer.vert")
            .fragment_path("assets/shaders/gbuffer.frag")
            .default_primitive_vertex_bindings()
            .default_primitive_vertex_attributes(),
    )
    .layout_out(gbuffer_position)
    .layout_out(gbuffer_normal)
    .layout_out(gbuffer_albedo)
    .layout_out(gbuffer_pbr)
    .external_depth_attachment(renderer.depth_image.clone(),
vk::AttachmentLoadOp::CLEAR)
    .record_render(move |device, command_buffer, renderer, pass,
resources| {
        let pipeline = resources.pipeline(pass.pipeline_handle);

        renderer.internal_renderer.draw_meshes(
            device,
            *command_buffer,
            pipeline.pipeline_layout,
        );
    })
    .build(graph);
}

```

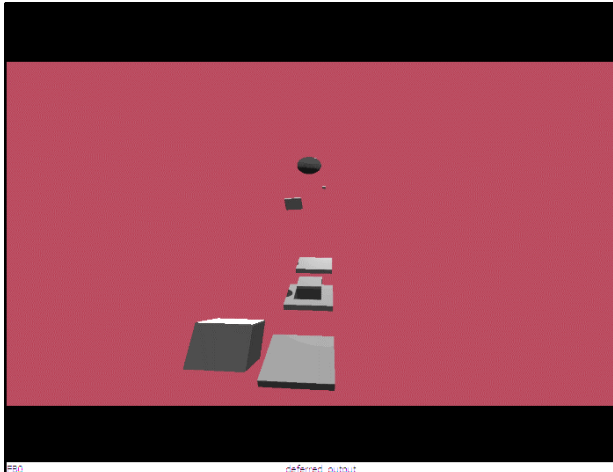
The vertex shader computes the transformed vertex attributes into the required formats for G-buffer generation. The tangent space basis (TBN) matrix is calculated based on vertex tangents, normals, and bitangents. The vertex position is transformed into world space, and texture coordinates, vertex color, normal, and tangent are computed accordingly. Finally, the transformed vertex position is projected onto the screen using view and projection matrices.

```

void main() {
    Mesh mesh = meshesSSBO.meshes[pushConsts.mesh_index];
    Vertex vertex = verticesSSBO[mesh.vertex_buffer].vertices[gl_VertexIndex];
    vec3 bitangentL = cross(vertex.normal.xyz, vertex.tangent.xyz);
    vec3 T = normalize(mat3(pushConsts.world) * vertex.tangent.xyz);
    vec3 B = normalize(mat3(pushConsts.world) * bitangentL);
    vec3 N = normalize(mat3(pushConsts.world) * vertex.normal.xyz);
    out_tbn = mat3(T, B, N);
    out_pos = (pushConsts.world * vec4(vertex.pos.xyz, 1.0)).xyz;
    out_uv = vertex.uv;
    out_color = vertex.color;
    out_normal = mat3(transpose(inverse(pushConsts.world))) * vertex.normal.xyz;
    out_tangent = vertex.tangent;
    gl_Position = view.projection * view.view * pushConsts.world *
    vec4(vertex.pos.xyz, 1.0);
}

```

Deferred Pass



```
pub fn setup_deferred_pass(
    graph: &mut RenderGraph,
    ...
) {
    graph
        .add_pass_from_desc(
            "deferred_pass",
            PipelineDesc::builder()
                .vertex_path("assets/shaders/fullscreen.vert")
                .fragment_path("assets/shaders/deferred.frag"),
        )
        .layout_in(gbuffer_position)
        .layout_in(gbuffer_normal)
        .layout_in(gbuffer_albedo)
        .layout_in(gbuffer_pbr)
        .layout_in(shadow_map)
        .layout_in(ssao_output)
        .layout_in(irradiance_map)
        .layout_in(specular_map)
        .layout_in(brdf_lut)
        .layout_out(deferred_output)
        .uniforms("shadowmapParams", &(cascade_data))
        .record_render(
            move |device, command_buffer, _renderer, _pass, _resources| unsafe {
                device.device().cmd_draw(*command_buffer, 3, 1, 0, 0);
            },
        )
        .build(graph);
}
```

The `setup_deferred_pass` function orchestrates the core phase of the deferred rendering technique, following the setup of the G-buffer. This function leverages the pre-populated G-buffer

textures, comprising positions, normals, albedo, and PBR parameters, to perform lighting calculations in screen space. It adds a new render pass to the render graph, binding the G-buffer textures as inputs.

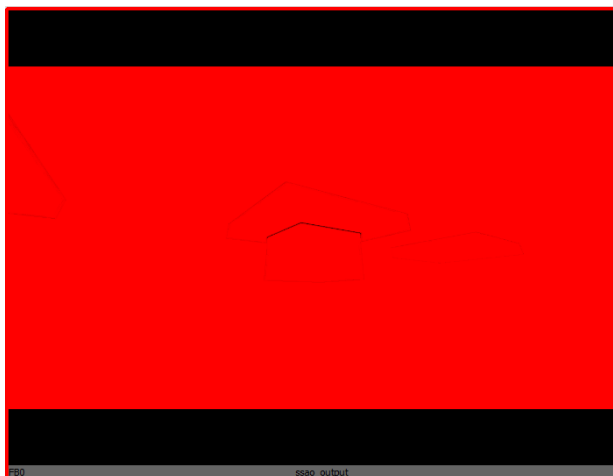
During this stage, we specify shader paths for deferred lighting calculations, often including point, spot, and directional light contributions. The shaders utilize the information from the G-buffer to compute lighting for each pixel on the screen, significantly reducing the computational load compared to forward rendering methods, especially in scenes with many light sources.

The function defines the output targets for the deferred pass, a single texture that accumulates the lighting information, referred to as the illumination buffer. The blending states within this pass are carefully configured to correctly combine the contributions of various lights.

In addition, the `setup_deferred_pass` handles the setup for potential post-processing inputs, such as shadow maps, which are incorporated into the lighting calculations to enhance visual fidelity. We manage the depth attachment's lifecycle, ensuring it's available for depth tests to maintain proper occlusion and layering of scene elements during lighting.

The rendering commands recorded within the pass invoke draw calls that typically render a full screen to dispatch work across the screen pixels. These draw calls are orchestrated such that each pixel's lighting is computed using the G-buffer's data, outputting the final shaded image to the lighting buffer.

SSAO



```
const SSAO_BIAS : glam::Vec2 = glam::const_vec2!([0.35, 0.3]);
pub fn setup_ssao_pass(
    graph: &mut RenderGraph,
    . . .
) {
    graph
        .add_pass_from_desc(
            "ssao_pass",
            PipelineDesc::builder()
                .vertex_path("assets/shaders/fullscreen.vert")
                .fragment_path("assets/shaders/ssao.frag"),
        )
}
```

```

        .layout_in(gbuffer_position)
        .layout_in(gbuffer_normal)
        .layout_out(ssao_output)
        .uniforms("settings_ubo", &(SSAO_BIAS))
        .record_render(
            move |device, command_buffer, _renderer, _pass, _resources| unsafe
        {
            if enabled {
                device.device().cmd_draw(*command_buffer, 3, 1, 0, 0);
            }
        },
    )
    .build(graph);
}

```

The `setup_ssao_pass` function configures a render pass within the render graph system for screen space ambient occlusion (SSAO), SSAO settings are passed as uniform values within a uniform buffer object (UBO), which contains fine-tuning parameters for the SSAO calculation to avoid artifacts like over-occlusion. A draw call is issued to render a full-screen quad, invoking the SSAO computation for each pixel. We then render 3 vertices to cover the screen buffer, described within the `fullscreen.vert` shader.

Atmospheric Pass

```

const WORLD_SCALE : Mat4 = const_mat4!([1000.,0.,0.,0.], [0.,1000.,0.,0.],
[0.,0.,1000.,0.], [0.,0.,0.,1.0]);
pub fn setup_atmosphere_pass(
    graph: &mut RenderGraph,
    renderer: &VulkanRenderer,
    atmosphere_output: TextureId,
    environment_map: TextureId,
    camera: &crate::camera::Camera
) {
    let projection = camera.get_projection();
    graph
        .add_pass_from_desc(
            "atmosphere_pass",
            PipelineDesc::builder()
                .vertex_path("assets/shaders/atmosphere.vert")
                .fragment_path("assets/shaders/atmosphere.frag")
                .default_primitive_vertex_bindings()
                .default_primitive_vertex_attributes(),
        )
        .load_write(atmosphere_output)
        .layout_in(environment_map)
        .uniforms("ubo_constants", &(projection, WORLD_SCALE))
        .external_depth_attachment(renderer.depth_image.clone(),
vk::AttachmentLoadOp::LOAD)
        .record_render(

```

The atmospheric pass uses a shader taken from Felix Westin's publically available MinimalAtmosphere, designed for Unity shaders but implemented within glsl. This is added like all other passes. This does rely on the first instance of a created model, and as such this would need to be resolved within context of this project later down the line.

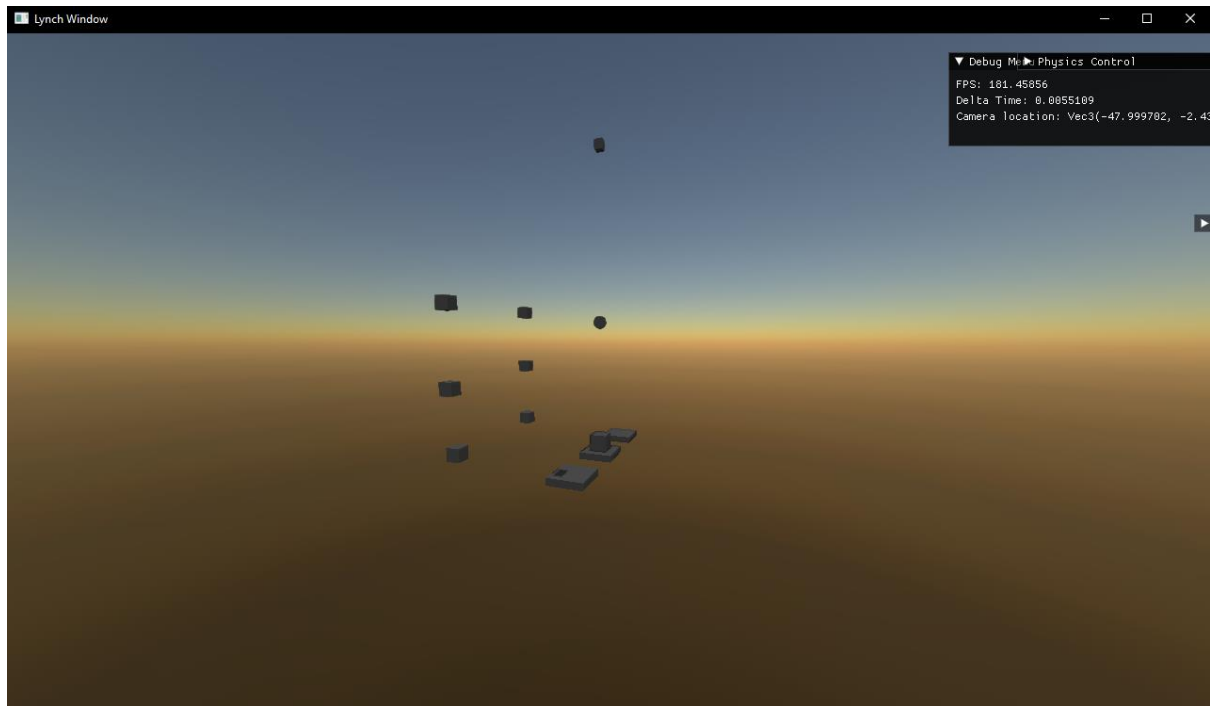


Figure 37, Example of atmosphere pass running within the renderer.

6.2.17 The Camera

The camera holds values which allows us to receive the projection and view of the user.

```
pub struct Camera {  
    camera_rig: CameraRig,  
    fov_degrees: f32,  
    aspect_ratio: f32,  
    z_near: f32,  
    z_far: f32,  
    speed: f32,  
}
```

I am using the dolly library which includes some features to implement camera movement smoothing. The camera is built by the user using its builder function, which allows for optional parameters, and more flexible type usage.

```
pub fn build(self) -> Camera {
```

```

        let camera_rig = self.camera_rig.unwrap_or_else(|| {
            CameraRig::builder()
                .with(Position::new(self.position))
                .with(YawPitch::new().rotation_quat(self.rotation))
                .with(Smooth::new_position_rotation(0.9, 0.9))
                .build()
        });
        let fov_degrees = self.fov_degrees;
        let aspect_ratio = self
            .aspect_ratio
            .expect("Aspect ratio is a required property for camera
builder.");
        let z_near = self.z_near;
        let z_far = self.z_far;
        let speed = self.speed;

        Camera {
            camera_rig,
            fov_degrees,
            aspect_ratio,
            z_near,
            z_far,
            speed,
        }
    }
}

```

6.3 Frost

Frost is the state management system for this project. We have a world struct where archetypes, entities and their metadata are stored. The world exists as a list of entities, which are simply two numbers, an ID and a generation, available entities, which are entities which have not yet been assigned, archetypes, which are collections entities and components, and the hash of component packs to their related archetypes.

```

pub struct World {
    pub(crate) entities: Vec<EntityMeta>,
    available_entities: Vec<EntityId>,
    pack_id_to_archetype: HashMap<u64, usize>,
    archetypes: Vec<Archetype>,
}

```

6.3.1 Entities

In my implementation, an entity is composed of two pieces of metadata, an ID, and a generation. An ID is simply the entities' location within some array, and its generation is used to see if an old entity matches a previous one. So, entities are a Vector of EntityMeta's, which look like this:

```

pub struct EntityMeta {
    pub generation: Generation,
}

```

```
pub location: EntityId,
}
```

As Rust practically forbids circular references, we cannot directly reference the `EntityMeta`'s location in its parent Vector, as the lifetime of `EntityMeta` will **always** subceed that of the reference to it stored within World, so therefore its location parameter *cannot guarantee* the position it points to *even exists*, and so therefore cannot safely be used. Instead, we store the index of the entity itself within the entity meta, forcing us to safely check if that entity exists in its parent structure before we can run operations on it. The `Generation` type is an alias of `u32`.

```
pub type EntityId = u32;
pub type Generation = EntityId;
```

6.3.2 Components

A component, in this case, is some type (a struct, or tuple struct), which contains data, and has some constraints placed on it. A component could look like this:

```
pub struct Name(String);

pub struct SampleComponent {
    pub name: Name,
    pub age: u32,
    pub brith_date: DateTime
}
```

A struct used as a component will need to have some limitations placed on it so we can safely coerce the struct into a component.

Here, we declare an empty trait `Component` which must implement Sync (can safely be used across threads) and Send (can safely be sent across threads) and that the struct itself exist for the static length of the program itself.

```
pub trait Component: Sync + Send + 'static {}
impl<T: Sync + Send + 'static> Component for T {}
```

Note: while these structs could technically exist for a 'world borrow, this would provide little benefit, as it seems uncouth to define datatypes that would be used in systems within functions themselves, and not within the context of a broader program

Take this definition as a reusable set of traits we can coerce other objects into, without needing to respecify them constantly. We can now define a trait which can be used to implement additional features for types and wrappers:

```
trait ComponentVec: Sync + Send {
    fn to_any(&self) -> &dyn Any;
    fn to_any_mut(&mut self) -> &mut dyn Any;
    ...
}
```



```
}
```

We can now implement this trait over safe types provided by Rust's standard library wherein we can guarantee that there is only ever one mutable writer, or many immutable readers for the type, satisfying type and safety checks:

```
impl<T: Component> ComponentVec for RwLock<Vec<T>> {  
    fn to_any(&self) -> &dyn Any {  
        self  
    }  
}
```

And this concludes the general mechanism for us to coerce regular structs into this Component trait and allows us to store this component in a vector of instances of that component. Now we need to store the components.

6.3.3 Component Stores

A component store is a data structure which can contain *one* type of component, storing its type of ID, and a vector of that component on the heap. As the size of the type of the component is technically unknown at compile time, we must store this component within a `Box` (essentially a pointer to heap which is guaranteed to point to a value which is uniquely owned by its caller)[CITATION NEEDED?], which can then place the type on the heap. (equivalent of handling a malloc and free for the user in a safer manner than C / C++).

```
struct ComponentStore {  
    pub type_id: TypeId,  
    data: Box<dyn ComponentVec + Send + Sync>,  
}
```

Here is how we can implement this:

```
impl ComponentStore {  
    pub fn new<T: 'static + Send + Sync>() -> Self {  
        Self {  
            type_id: TypeId::of::<T>(),  
            data: Box::new(RwLock::new(Vec::<T>::new())),  
        }  
    }  
}
```

6.3.4 Archetypes

An archetype serves as a fundamental organizational structure within the ECS, providing a systematic approach to managing entities and their associated components. At its core, an archetype represents a blueprint or template for a specific combination of components that entities can possess within the game world. By grouping entities with similar component compositions into archetypes, we can efficiently optimize memory usage, data storage, and processing, enhancing performance.

Simple put an archetype is a list of entities which share a list of related components. When we want to search for any entities, with say, a type A and a type B, ideally we only want to iterate over those components, and if we have an entity with Type A, Type B, and Type C, we want to be able to find archetypes which match with $O(n)$.

```
pub struct Archetype {
    pub(crate) entities: Vec<EntityId>,
    pub(crate) components: Vec<ComponentStore>,
}
```

With this layout, we optimize the memory layout of entities and their related components by storing related components in contiguous heap memory, optimizing for cache locality.

Also, we will need to change our entity id, to include which archetype the entity is stored, and where in that archetype it is stored:

```
pub struct EntityLocation {
    archetype_index: EntityId,
    index_in_archetype: EntityId,
}
```

Which will be propagated to EntityMeta in place of the previously stored EntityId.

6.3.5 ComponentPack (the glue which holds archetypes, entities, and components together)

We first (as with nearly every step) define a trait which holds a pack of components which can be spawned into the world. The “*ComponentPack*” is a type which the user provides and is coerced into a “*ComponentStore*” type.

```
pub trait ComponentPack: 'static + Send + Sync {
    fn new_archetype(&self) -> Archetype;
    fn spawn(self, world: &mut World, entity_index: EntityId) -> EntityLocation;
}
```

When a user wants to spawn an entity with components, they must provide actual instances of those components (or generic structures).

We also need to hash type ids of components within component packs, so that we can use a hashmap to look up the stored position of archetypes which contain them.

We now need to be able to spawn N amount of components, which will be passed along in this case on the stack, to be later moved to component stores within the related archetype, if it exists. This is the guts of the ability to add Components to the ECS. Let's implement this `ComponentPack` for some abstract type, `A`.

```
impl <A:'static+Send+Sync>ComponentPack for(A,){
fn spawn(self,world: &mut World,entity_index:EntityId) -> EntityLocation {
let mut types = [(0, TypeId::of::<A>())];
types.sort_unstable by(|a, b| a.1.cmp(&b.1));
let mut order = [0; 1];
```

6.3.6 Searching the World for Entities with Components

In ECS systems, efficient searching of components is crucial, however, my initial solution faced limitations in handling searches involving multiple components.

Searching the world in the ECS is done by “using” (or importing) the search struct. We can then, at compile time, coerce these search parameters into types which we can then generate retrieve methods over.

A search can be defined like this:

```
Search<(&StructA, &StructB,...)>
```

Where the search structs, decorated by a type “T”:

```
pub struct Search<'world, T: SearchParameters> {
    pub data: <T as SearchParameterGet<'world>>::RetrieveItem,
}
```

This struct holds “data” (which will be an array of components, where the entity has all the supplied search types). RetrieveItem is a trait which describes a function which allows the inner component type to be returned.

```
pub trait RetrieveItem<'a> {
    type InnerComponent;
    fn inner(&'a mut self) -> Self::InnerComponent;
}
```

The type T, must implement the trait “SearchParameters”:

```
pub trait SearchParameters: for<'a> SearchParameterGet<'a> {}
```

SearchParameters is represented as a tuple of structs which implement a specific set of traits. With some extra implementations, we can now coerce the tuple of structs (which is a type itself) within the Search query, Struct A and Struct B, into this SearchParameters type:

```
impl <'world,A:SearchParameter, B:SearchParameter,>SearchParameters for(A,
B){}
```

SearchParameter is a type which can be used to match a component to its archetype:

```
pub trait SearchParameter {
    type SearchParameterGet: for<'a> SearchParameterGet<'a>;
    fn matches(archetype: &Archetype) -> bool;
}
```

We use this trait in our last, longest, and most important implementation regarding searching:

```
impl <'world,A:SearchParameter,B:SearchParameter,
>SearchParameterRetrieve<'world>for(A,B){
    #[allow(unused_parens)]
```

```

type RetrieveItem = Vec<(<A::SearchParameterRetrieve as
SearchParameterRetrieve<'world>>::RetrieveItem,<B::SearchParameterRetrieve as
SearchParameterRetrieve<'world>>::RetrieveItem)>>;
fn retrieve(world: &'world World, _: usize) ->
Result<Self::RetrieveItem, RetrieveError>{
    let mut archetype_indices = Vec::new();
    for(i, archetype) in world.archetypes.iter().enumerate(){
        let matches =
A::matches_archetype(&archetype)&&B::matches_archetype(&archetype);
        if matches {
            archetype_indices.push(i);
        }
    }let mut result = Vec::with_capacity(archetype_indices.len());
    for index in archetype_indices {
        result.push((<A::SearchParameterRetrieve as
SearchParameterRetrieve<'world>>::retrieve(world, index)?
,<B::SearchParameterRetrieve as
SearchParameterRetrieve<'world>>::retrieve(world, index)?));
    }
    Ok(result)
}
}

```

This collects data from our world, checking where the types provided as `SearchParameters` match within the archetype provided, returning either an ok result containing the value, or an error describing the issue, in a way where the types are coerced at compile time, and the code can then be optimised during the compilation and borrow checking phase, and not during run time, ensuring necessary searching is done.

There is a problem though, as this code only works for searches containing 2 components (A and B), our code can't currently coerce a search which contains a tuple of 1, or 3, or n components. The first solution to this problem is to write a lot of repetitive code where each necessary component is added, and the second would be to instead rely on runtime checks to verify type size and safety, which would ultimately limit our ability to manage data on the stack and would require all component manipulation to occur within Boxes on the heap. This is where Rust's macros saved the day.

6.3.7 Macro Utilization

In Rust, macros can be used as a mechanism to engage in meta programming. This could allow me to generate complex type relationships between arbitrary types, without the need to manually create each implementation.

The above definition of can be better represented as:

```
impl<'world, N amount of Type which impl SearchParameters*>
SearchParameterRetrieve<'world> for (Tuple of N Types, defined before) {
    type RetrieveItem = Vec<Tuple of N Types which can safely
retrieved>;

    fn retrieve ...
        let mut archetype_indices = Vec::new();
        for (i, archetype) in world.archetypes.iter().enumerate() {
            let matches = ;
            if Type of I from N Types matches some archetype {
                archetype_indices.push(i);
            }
        }

        let mut result = Vec::with_capacity(archetype_indices.len());
        for index in archetype_indices {
            result.push(Retrieve value from Type I of N Types );
        }

        Ok(result)
    }
}
```

Macros accomplish this by defining some inputs:

A) a condition for our input to match against:

```
macro_rules! search_params {
    ($($name: ident),*) => {
```

This defines a macro, `search_params`, which matches any zero or more list of values to a rule. We define “*name*”, which is “*ident*” (a variable name, or in this case, a generic type identifier (A/B/C)), and then we start to dynamically write code which would otherwise be repetitive. (the `=> {` indicates a “closure” or section of code these variables “name” will be passed into).

I’ll describe the process of using `search_params` for two types `A` and `B`:

```
search_params!{A,B}
```

This matches:

```
($ (name=A, name=B))
```

So now we can define the implementation for search parameters to take this data:

```
impl<'world, $($name: SearchParameter,)*> SearchParameters
for ($($name,)*){}
```

which will expand `($ (name=A, name=B))` into:

```
impl<'world, $(A: SearchParameter, B: SearchParameter,)*> SearchParameters
for (A, B){}
```

The same principle applies to the longer function, but for the sake of brevity will leave that out of here.

Now there is another, smaller problem. We must define N number of `search_params` to create usable search queries. To accomplish this, we will need to define each required amount of search parameters. I have arbitrarily chosen 26, as that's how many letters exist in the alphabet which can be used for generics. This looks like this:

```
search_params!{A}
search_params!{A,B}
search_params!{A,B,C}
search_params!{A,B,C,D}
search_params!{A,B,C,D,E}
```

This repetition is a little ugly, and unnecessary, so I defined a second macro which recursively calls itself, removing the head of the list from its tail and recursively executing this:

```
macro_rules! search_paramsr{
  ($x: ident) => {
    search_params!{$x}
  };
  ($x: ident, $($y: ident),*) => {
    search_params!{$x, $($y),*}
    search_paramsr!{$($y),*}
  };
}
```

We use more list comprehension to now generate each `search_params`, removing from the head during each call. The base case is when there exists only one type remaining the macro-based solution provides a more efficient and flexible approach to handling ECS component searches. By leveraging Rust's macro capabilities, we can reduce code repetition, improve maintainability, and enhance overall system scalability.

6.3.8 Systems

A system can be defined as this trait bound:

```
pub trait System<P> {
  fn run(&mut self, world: &World, delta_time: f32) -> Result<(),
RetrieveError>;
}
```

The `System` trait defines a common interface for systems in the ECS architecture. It takes a generic parameter `P`, which represents the type of parameters required by the system to run. The run

method within the trait signature is responsible for executing the system's logic. It takes a reference to the World structure, representing the game world, and the delta time, which indicates the time elapsed since the last update. The method returns a `Result<(), RetrieveError>`, where `RetrieveError` represents a potential failure in retrieving some search result for a component.

```
impl <FUNC, ABC: SysParam> System<(ABC,)> for FUNC where
FUNC: FnMut(ABC, f32) + for <'a, 'b> FnMut(InnerComponent<'a, 'b, ABC>, f32) {
    fn run<'world>(&mut self, world: &'world World, delta_time: f32) ->
Result<(), RetrieveError> {
    let mut ABC = ABC::Retrieve::retrieve(world)?;
    self(ABC.inner(), delta_time);
    Ok(())
}
}
```

The `impl` block provides an implementation for the `System` trait. It defines how the `run` method should behave for any function (`FUNC`) that matches certain trait bounds. These bounds specify that the function must take a parameter of type `ABC` (which implements the `SysParameter` trait) and an `f32` representing delta time. The function can also accept an `InnerComponent` type, which represents a component retrieved from the world. The implementation ensures that the function is capable of processing components and delta time.

```
struct SampleStruct {
    a: i32,
    b: f32,
    c: i64,
}

fn world_system(mut search: Search<(&SampleStruct,)>, _delta_time: f32) {
    for i in search.iter() {
        println!("{}", i.a, i.b, i.c)
    }
}
```

The code includes an example function (`world_system`) and a sample structure (`SampleStruct`) to demonstrate how systems are defined. The `world_system` function iterates over a search result for components of type `SampleStruct` and prints their fields. This function conforms to the trait bounds specified in the system implementation, making it compatible with the `run` method provided by the `System` trait.

```
world_system.run(&mut world, 0.0126)
```

The `run` method of a system can be called by invoking it on the system object, passing the game world and delta time as arguments. For example, `world_system.run(&mut world, 0.0126)` executes the `world_system` function as a system within the ECS architecture, processing components retrieved from the world and using a delta time of 0.0126.

6.3.9 Rigid Body

The `RigidBody` struct encompasses various attributes crucial for simulating physical objects within the game world. These attributes include `inverse_mass`, representing the reciprocal of the object's mass, facilitating efficient computations involving acceleration and force. The `transform` struct encapsulates the spatial properties of the rigid body, including its position, orientation, and scale. Acceleration and velocity vectors denote the linear motion of the object, while `angular_velocity` describes its rotational motion. The `inverse_inertia_tensor` matrix defines the object's resistance to rotational motion, aiding in calculations of angular acceleration and torque. `force_accumulator` and `torque_accumulator` accumulate external forces and torques acting on the object, respectively. Boolean flags like `gravity` determine if gravity affects the object, while `angular_drag` and `restitution` coefficients regulate angular drag and collision elasticity. Finally, `is_static` distinguishes between static and dynamic objects.

```
pub struct RigidBody {
    pub inverse_mass: f32,
    pub transform: Transform,
    pub acceleration: Vec3,
    pub velocity: Vec3,
    pub angular_velocity: Vec3,
    pub inverse_inertia_tensor: Mat3,
    pub force_accumulator: Vec3,
    pub torque_accumulator: Vec3,
    pub gravity: bool,
    pub angular_drag: f32,
    pub restitution: f32,
    pub is_static: bool,
}
```

6.3.10 Orientated Bounding Box (OBB)

An Oriented Bounding Box (OBB) is a geometric shape used in collision detection algorithms. Unlike an Axis-Aligned Bounding Box (AABB), which is aligned with the coordinate axes and cannot be rotated, an OBB can be rotated arbitrarily in three-dimensional space.

An OBB is defined by a set of parameters that specify its position, orientation, and size:

```
pub struct DynamicOBB {
    pub center: Vec3,
    pub half_extents: Vec3,
    pub orientation: Quat,
    pub vertices: [Vec3; 8],
    pub faces: [PolygonPrimitive; 6],
}
```


The `DynamicOBB` struct represents an oriented bounding box (OBB) in three-dimensional space. It comprises several properties crucial for defining the box's geometry and orientation within a game or simulation environment. The `centre` property denotes the central point of the OBB, determining its position in space. Complementing the `centre`, the `half_extents` property defines the dimensions of the box along each axis, essentially representing half of the width, height, and depth. This property influences the size of the OBB, allowing for flexible adjustments to its scale.

Additionally, the `vertices` array holds eight `Vec3` vectors, each representing one of the box's corners. These vertices delineate the boundaries of the OBB and are instrumental in collision detection and spatial computations. The `faces` array contains instances of the `PolygonPrimitive` struct, which represent the individual faces of the OBB. Each `PolygonPrimitive` encapsulates information about the vertices, edges, and face ID associated with a particular face of the OBB, facilitating detailed geometric analysis and interaction within the physics system.

```
pub struct PolygonPrimitive {
    pub num_vertices: usize,
    pub vertex_ids: [WrappedPrimitiveId; 4],
    pub edge_ids: [WrappedPrimitiveId; 4],
    pub face_id: WrappedPrimitiveId,
}
```

The `PolygonPrimitive` struct, on the other hand, complements the `DynamicOBB` by representing individual polygons that comprise its faces. This struct contains properties and methods necessary for describing and manipulating polygons within the context of collision detection.

```
impl WrappedPrimitiveId {
    fn assert_mask<T>(code: T)
    where?
        T: std::fmt::Debug + std::ops::BitAnd<u32> + Binary +
std::marker::Copy,
        <T as std::ops::BitAnd<u32>>::Output: PartialEq<u32>
    {
        if (code & Self::BIT_MASK) != 0u32 {
            panic!("Primitive does not have required flag '{:#032b}', actual
value: {:#032b} ", Self::BIT_MASK, code);
        }
    }
    pub const UNKNOWN: Self = Self(0);

    const CODE_MASK: u32 = 0x3fff_ffff;
    const BIT_MASK: u32 = !Self::CODE_MASK;
    const VERTEX_BIT: u32 = 0b01 << 30;
    const EDGE_BIT: u32 = 0b10 << 30;
}
```

```
pub enum PrimitiveId {
    Vertex(u32),
    Edge(u32),
    Face(u32),
    Unknown
}
```

This code defines two related structures, `PrimitiveId` and `WrappedPrimitiveId`, which can be interchanged.

The `PrimitiveId` enum enumerates different types of primitives that can exist within a geometric context, namely vertices, edges, and faces, along with an 'Unknown' variant. This enum allows for easy identification and classification of geometric primitives within algorithms or data structures.

On the other hand, the `WrappedPrimitiveId` struct serves as a wrapper around a primitive identifier, incorporating additional functionality for handling and manipulating these identifiers within a specific context. It provides methods for constructing primitive identifiers with additional metadata, including whether the identifier represents a vertex, edge, or face. It also offers utilities for converting between `PrimitiveId` and `WrappedPrimitiveId`.

6.3.11 Collision System

The collision system runs over any entity which contains a collider (in this project this only includes object orientated bounding boxes) and runs the separating axis theorem to determine if a chosen bounding box collides with another.

```
pub fn collision_system<'a>(<
    mut search: Search<(&'a mut obb::DynamicOBB,)>,
    fixed_update: f32,
) where
    'a: 'static,
{
    let bodies_and_boxes = search.iter().collect::<Vec<_>>();

    let mut collision_details = Vec::new();
    for i in 0..bodies_and_boxes.len() {
        for j in (i + 1)..bodies_and_boxes.len() {
            let (rb, obb1) = &bodies_and_boxes[i];
            let (rb2, obb2) = &bodies_and_boxes[j];
            if rb.is_static && rb2.is_static {
                continue;
            }

            if let Some(collision_point) =
                obb1.get_collision_point_normal(*obb2) {
```

```

        collision_details.push((i, j, collision_point));
    }
}
}

```

We check for potential collisions between pairs of objects represented by these bounding boxes. For each pair of objects, we evaluate whether there is a collision between their respective bounding boxes. If a collision is detected, we record and store details about the collision, such as the indices of the colliding objects and the collision point. We also skip collision detection between two static objects.

We use the function `handle_collision` to manage collisions between two rigid bodies within this simulation. This correction is applied proportionally to the bodies' inverse masses. Subsequently, we compute the relative velocity at the collision point and calculate an impulse to resolve the collision, considering the coefficient of restitution for both bodies. Finally, we apply the calculated impulse to each body to enforce conservation of momentum and restitution, ensuring realistic collision behavior within the simulation.

6.3.12 Physics System

In the `physics_system` function, we process physics simulation for a collection of rigid bodies and dynamic oriented bounding boxes (OBBs). We iterate through all pairs of bodies and boxes to detect potential collisions. If at least one of the bodies involved is not static, we check for collisions between their corresponding OBBs using the `get_collision_point_normal` method. Detected collision details are stored for further processing. Afterward, we iterate through the stored collision details and resolve collisions between the affected bodies using the `handle_collision` function.

```

pub fn physics_system<'a>(
    mut search: Search<(&mut RigidBody, &'a mut obb::DynamicOBB)>,
    fixed_update: f32,
) where
    'a: 'static,
{
    bodies_and_boxes.iter_mut().for_each(|b_b| {
        let (rb, obb) = b_b.borrow_mut().get_mut();

        if !rb.is_static {
            rb.apply_gravity();
            rb.apply_angular_drag(fixed_update);
            rb.integrate(fixed_update);
        }
        obb.center = rb.transform.position;
        obb.orientation = rb.transform.rotation;
        obb.half_extents = rb.transform.scale * 0.5;
    });
}

```

```

        obb.update_vertices();
    });
}

```

We update the properties of each rigid body and its associated OBB based on the simulation parameters. For each body-box pair, we apply gravitational forces, handle angular drag, and integrate the body's position and velocity over the fixed update time step. Additionally, we update the centre, orientation, and half extents of each OBB to match the corresponding rigid body's transformation, ensuring synchronization between the physical representation of objects and their bounding volumes within the simulation.

6.3.13 Integration Function

The integration function takes the output of the force and torque accumulators and applies them to the rigid body.

```

pub fn integrate(&mut self, fixed_time: f32) {
    debug_assert_ne!(fixed_time, 0.0, "Fixed time step cannot be zero");
    debug_assert(!self.is_static, "Static rigid bodies cannot be
integrated");

```

The acceleration (accel) of the rigid body is computed by multiplying the accumulated force (self.force_accumulator) with the inverse mass of the rigid body (self.inverse_mass). This step calculates how much the velocity of the rigid body should change over the time step. The velocity of the rigid body is updated by adding the computed acceleration (accel). This step applies the accumulated forces to the rigid body, affecting its linear motion. The position of the rigid body is updated by multiplying its velocity by the fixed time step (fixed_time) and adding it to the current position (self.transform.position). This step integrates the linear motion of the rigid body over the specified time interval.

```

    let accel = self.force_accumulator * self.inverse_mass;
    self.velocity += accel;
    self.transform.position += self.velocity * fixed_time ;

```

The angular acceleration (angular_accel) of the rigid body is computed by multiplying the accumulated torque (self.torque_accumulator) with the inverse inertia tensor (self.inverse_inertia_tensor). This step calculates how much the angular velocity of the rigid body should change over the time step. The angular velocity of the rigid body is updated by adding the computed angular acceleration (angular_accel). This step applies the accumulated torques to the rigid body, affecting its rotational motion.

```

let angular_accel = self.inverse_inertia_tensor * self.torque_accumulator;
self.angular_velocity += angular_accel * fixed_time;
let angle = self.angular_velocity.length() * fixed_time;

```

The orientation of the rigid body is updated based on its angular velocity (`self.angular_velocity`). The rotation is computed using a quaternion (`delta_rotation`) derived from the angular velocity and the fixed time step. This step integrates the rotational motion of the rigid body over the specified time interval.

```
let axis = if angle > 0.0 { self.angular_velocity / angle.abs()
} else {Vec3::X };
let delta_rotation = Quat::from_axis_angle(axis, angle);
self.transform.rotation = (self.transform.rotation *
delta_rotation).normalize();
```

Finally, the function clears the force and torque accumulators to prepare for the next integration step.

```
self.clear_accumulators();
```

6.3.14 The UI

The UI was implemented using `imgui-rs`, a Rust port of the popular C++ `imgui` library. To enable this, a render pass needed to be added to the renderer, and then the created UI can be passed to the engine user each frame. The UI is stored in this component:

```
struct CooperUI {
    pub gui: imgui::Context,
    pub platform: imgui_winit_support::WinitPlatform,
}
```

A gui frame is then started:

```
let mut gui_frame = self.ui.gui.frame();
```

Which can then be written over the “`ui_func`” by the user of the engine, then rendered by `IMGUI` and passed to the renderer.

```
ui_func(&mut run, &mut gui_frame);
let draw_data = self.ui.gui.render();
self.renderer.render(
    &mut self.graph,
    &self.camera,
    draw_data,
    &mut render_statistics,
);
```

The UI is then drawn using this function on the rendering end, done as a render pass:

```
let color_attachment_info = vk::RenderingAttachmentInfo::builder()
    .image_view(image.image_view)
    .image_layout(vk::ImageLayout::COLOR_ATTACHMENT_OPTIMAL)
    .load_op(vk::AttachmentLoadOp::LOAD)
    .store_op(vk::AttachmentStoreOp::STORE)
    .build();
```

```

let rendering_info = vk::RenderingInfo::builder()
    .render_area(vk::Rect2D {
        offset: vk::Offset2D { x: 0, y: 0 },
        extent: vk::Extent2D{width: image.width(), height: image.height()},
    })
    .layer_count(1)
    .color_attachments(std::slice::from_ref(&color_attachment_info))
    .build();
self.ash_device().cmd_begin_rendering(command_buffer, &rendering_info);
self.gui_renderer.cmd_draw(command_buffer, draw_data).unwrap();
self.ash_device().cmd_end_rendering(command_buffer);

```

The UI then can be used to modify and interact with data within the engine, providing user input and important debug data back to developers.

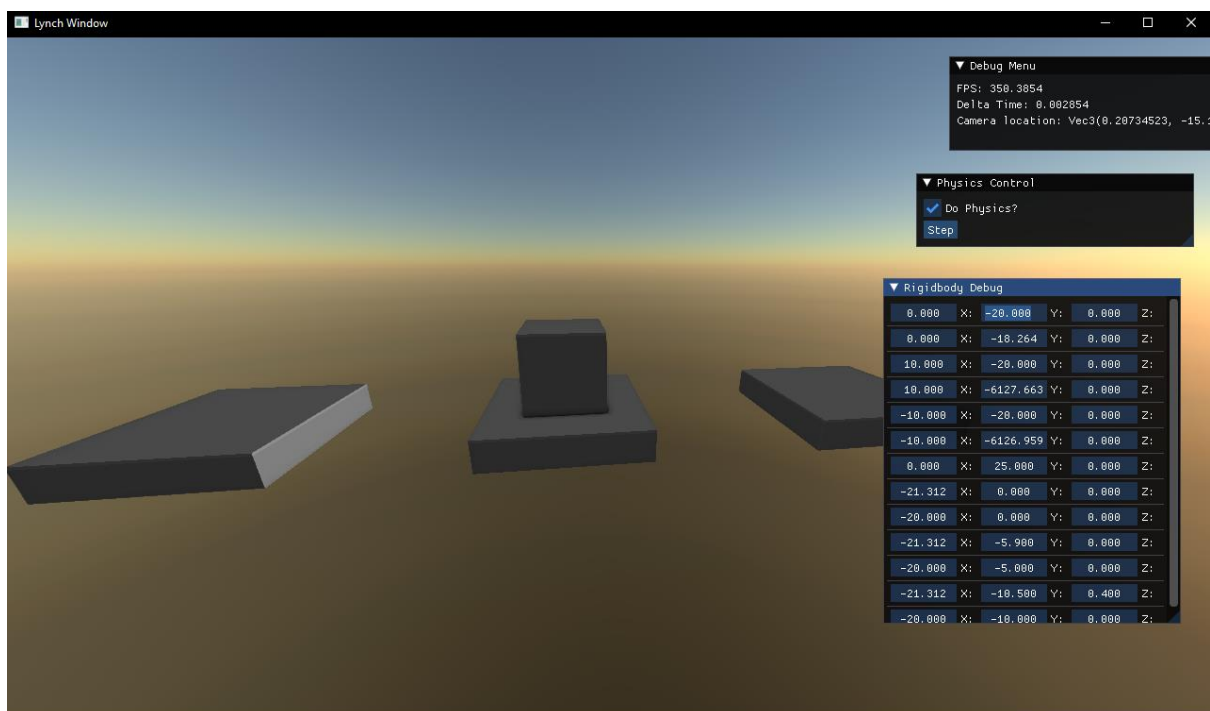


Figure 38, Sample of UI running within the engine.

7. Testing and Evaluation

7.1.1 Introduction

Testing and evaluation of a game engine is a complex topic. Testing was initially planned to be done using UAT(user acceptance testing), although this became unfeasible by the end, as the engine was not in a state where users could create games in a flexible state. This was a serious point of learning. I have maintained the use of benchmarking, and most of the testing is through either unit tests of ECS, and physics functions, as well as a benchmarking and stress testing the ECS, physics and graphical library by themselves.

System Testing

This is to benchmark the entire system, which will test the ECS (with physics and collision systems running) as well as rendering and will compare the availability of the aims and objectives. The examples provided below show the whole system executing successfully, running around 400 fps. These show each step as the system handles rendering, collisions, physics, ECS and UI rendering.

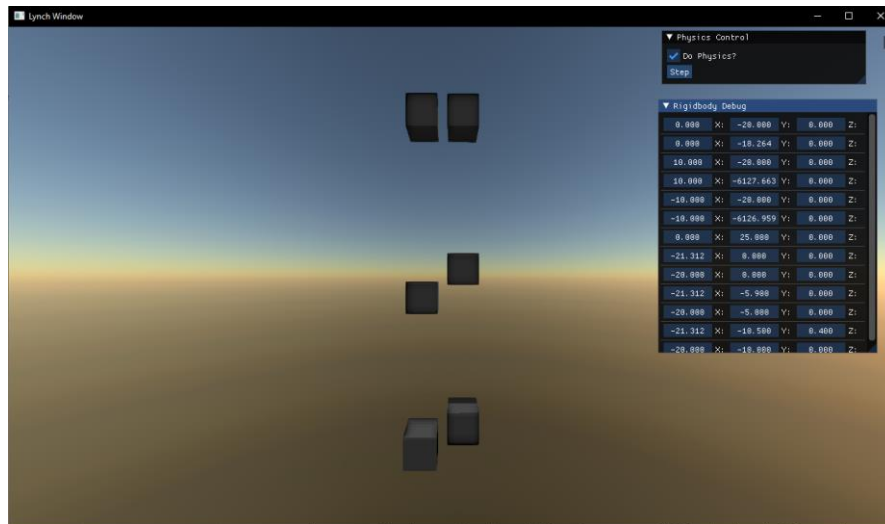


Figure 39, Samples of Cubes about to Collide.

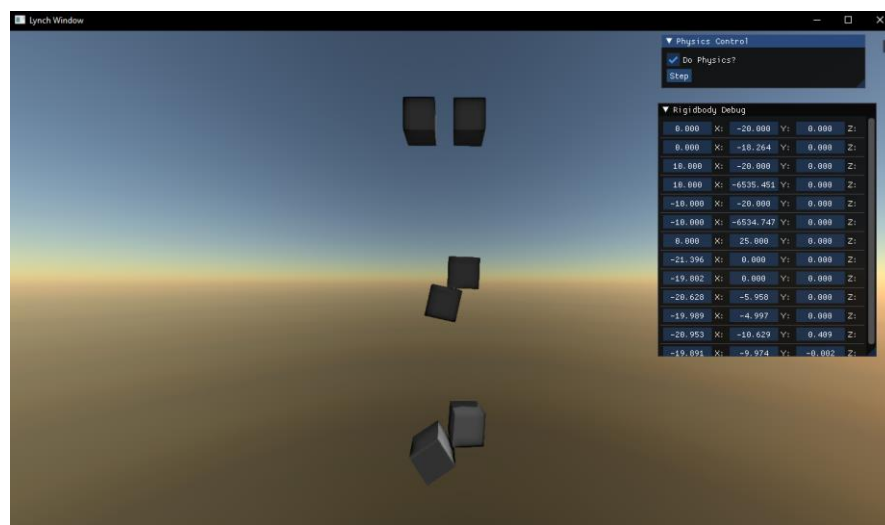


Figure 40, Sample of Cubes of Different Weights Colliding

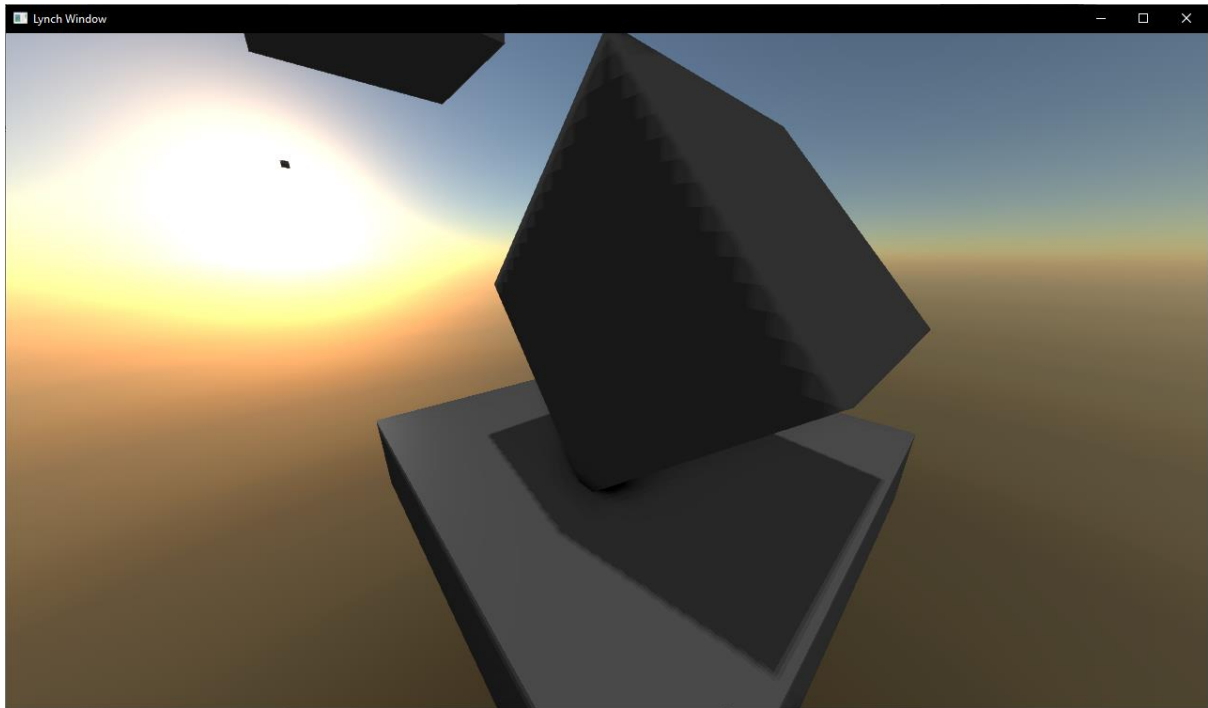


Figure 41, Demonstration of shadows and default textures.

The Methal Rough Sphers scene is a .glTF file provided by nVidia to test the format and show how it stores different materials. This glTF demonstrates the different materials which can be loaded from data within the engine.

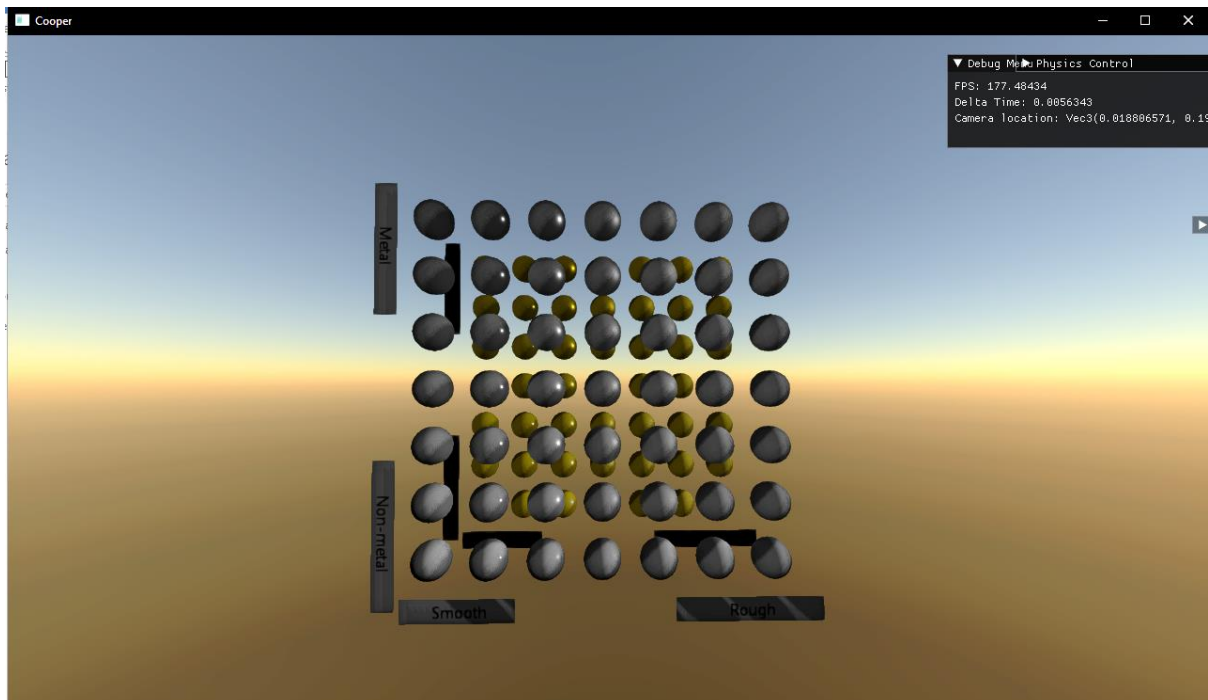


Figure 42, Example of glTF Metal / Non-Metal and Smooth / Rough materials.

The Sponza scene is a .glTF file provided by nVidia to test the format and show how it stores different materials. This scene shows how the renderer, when loading complex models which are

batched correctly, can load complex models, materials, and display shadows on those models with high levels of detail, and running around 131 frames per second.

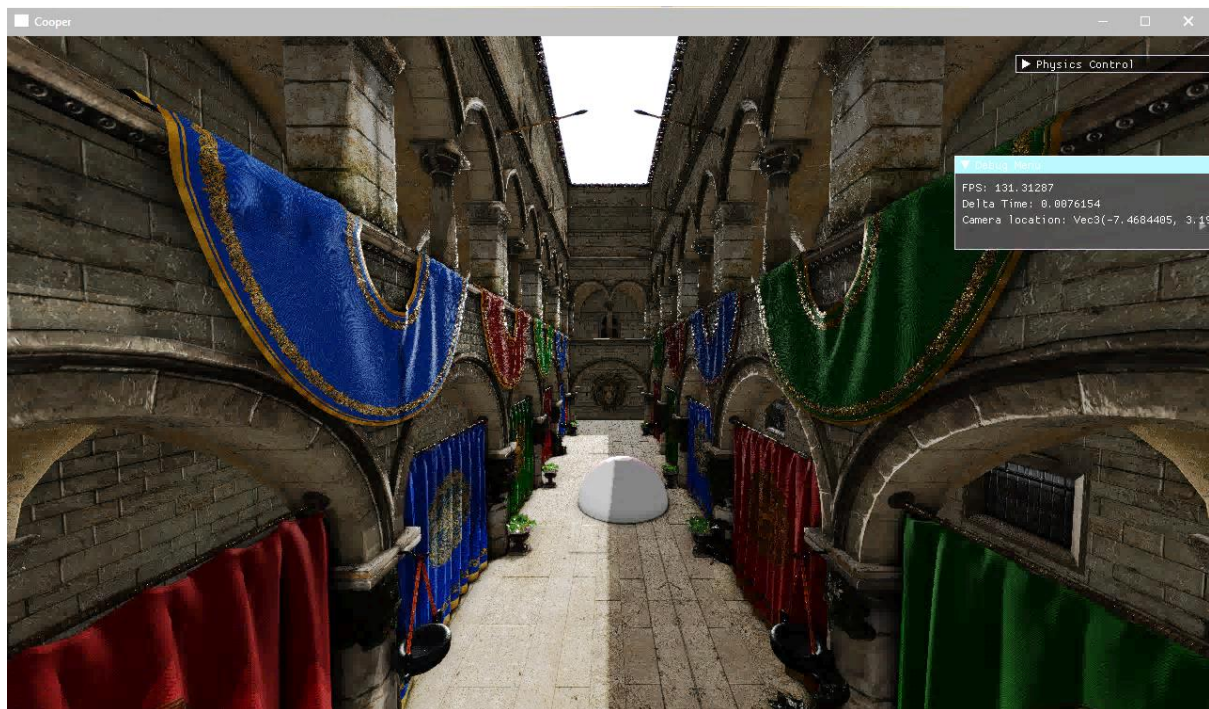


Figure 43, The Sponza scene being rendered at 131 FPS.

7.1.2 Benchmarking

Benchmarking was done against metrics in the design area, frame times, render pass times, and general ECS performance. This was done to compare the performance of the engine in different contexts, and how it handles increased load.

Lynch

The first part measured on Lynch was how long the system took to start, and to complete the first frame. This was important, as shader compilation and pipeline compilation happen within this frame, as well as the creation of most buffers. On a scene with 200 cubes, it took ~2.5 seconds to start and render the first frame.

Time to Start	509.482ms
Time to complete first frame	2.4629997s

Figure 44, Time to First Frame

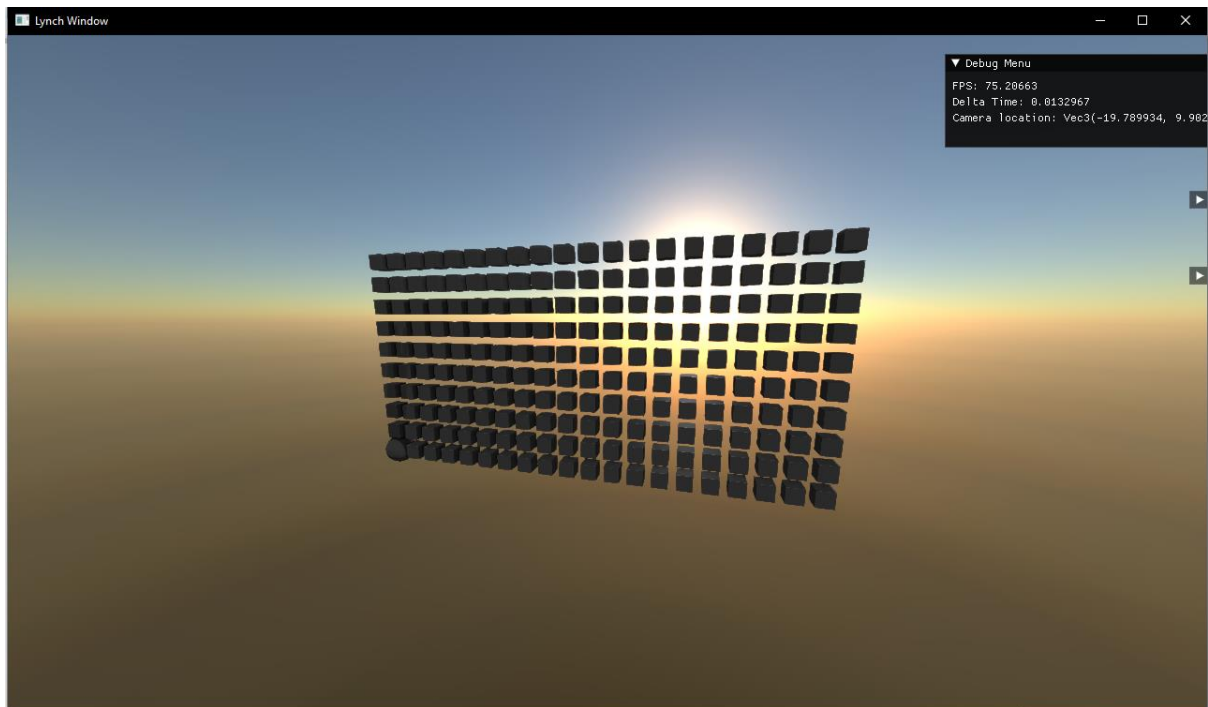


Figure 45, Render of 200 instances.

The most important test of Lynch was comparing the number of spawned instances of a model, and its effect on frame time, more importantly, on the length of time taken to render each pass of the frame. As we can see, any pass which required meshes to be processed for rendering (GBuffer and shadow maps) increased linearly ($O(n)$) with the number of instances. While I am unhappy with this result, as frame times become unreasonable ~ 200 instances, it is a problem that can be solved through multithreading and optimizing the mechanism of sending vertex data to the graphics card.

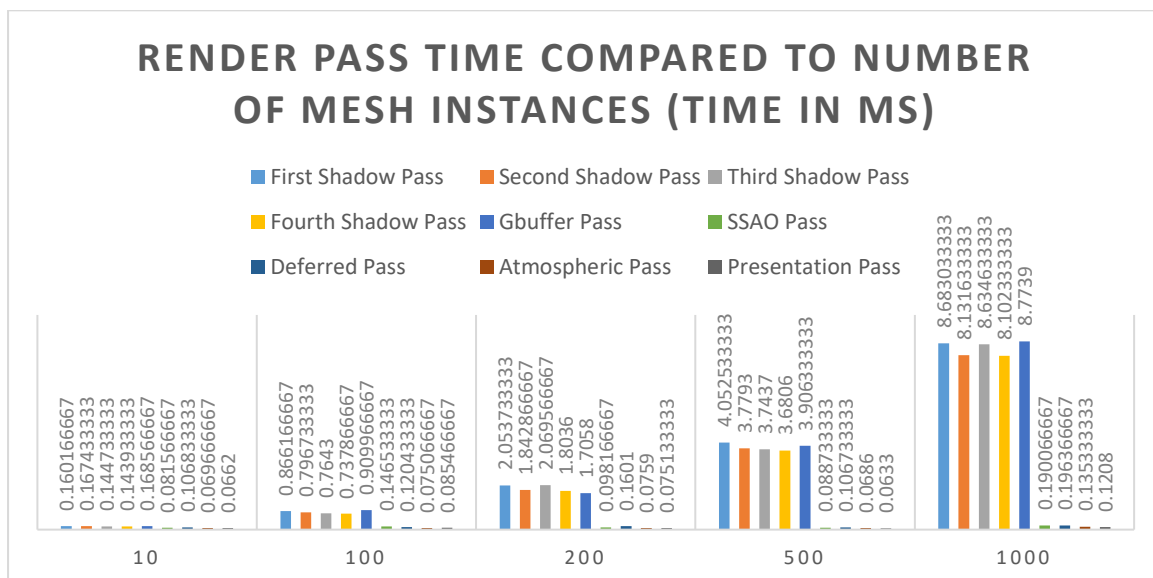


Figure 46, measurement of render pass time to mesh instances.

Frost

The benchmarking of frost was done using the “criterion” Rust library, which provides tools to effectively run benchmarks over code. This was used in conjunction with Gnuplot to provide 100 sample runs of different benchmark scenarios of the ECS.

```
fn criterion_benchmark(c: &mut Criterion) {
    let sizes =
[10,20,50,100,200,400,500,1000,1500,2000,5000,10000,20000,50000,100000,200000,
500000,1000000];
    for size in sizes {
        c.bench_with_input(BenchmarkId::new("Insert New Entity", size), &size,
|b, &s| {
            b.iter(||{world_insertions(s)});
        });
        let mut world = World::new();
        insertions(&mut world, size);
        c.bench_with_input(BenchmarkId::new("Search Entity", size), &size, |b,
&s| {
            b.iter(||{world_search.run(&world, 0.0)});
        });
    }
}
```

The benchmark was run like this, and as such is also a measure of the System aspect of the Entity component system.

The following graphs show how consistent the insertion and search time of the ECS really is, with outliers and severe outliers marked. These were running 100 times, to mitigate the effects of outliers in the measurement data, and to reduce bias.

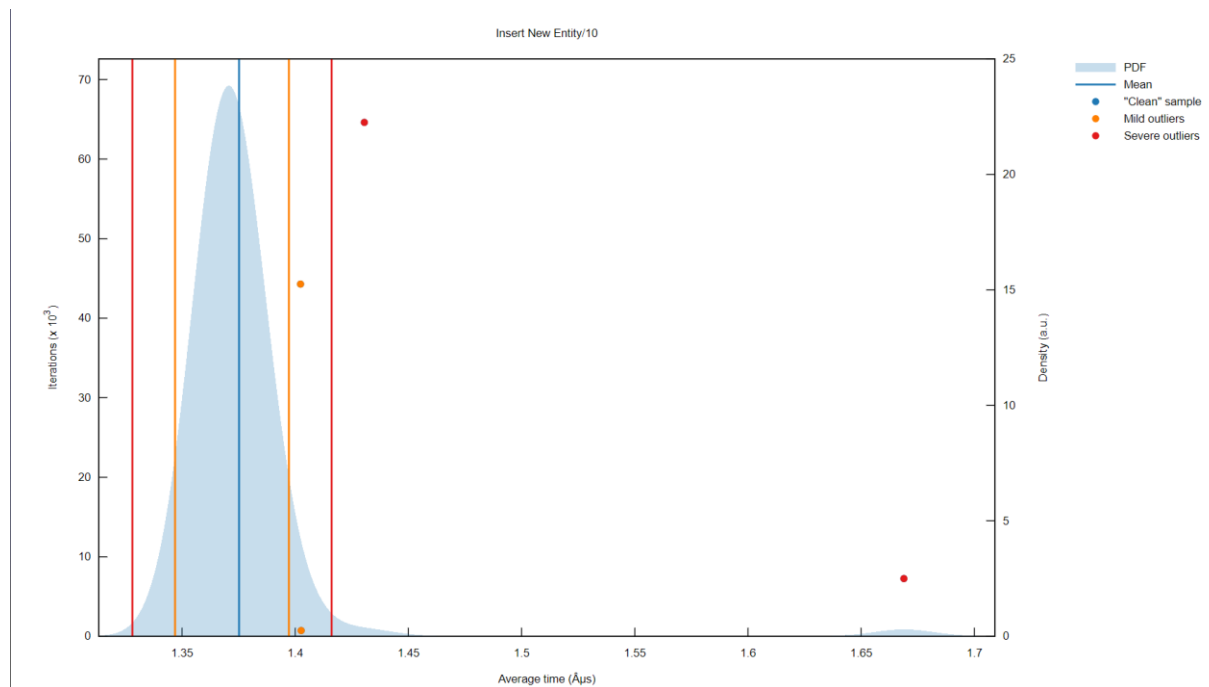


Figure 47, Inserting 10 entities.

Note: each of these benchmarks also include the time to create the “world” resource within them, and so realistically can be more accurately represented with lower values, although time constraints meant I had to leave it this way.

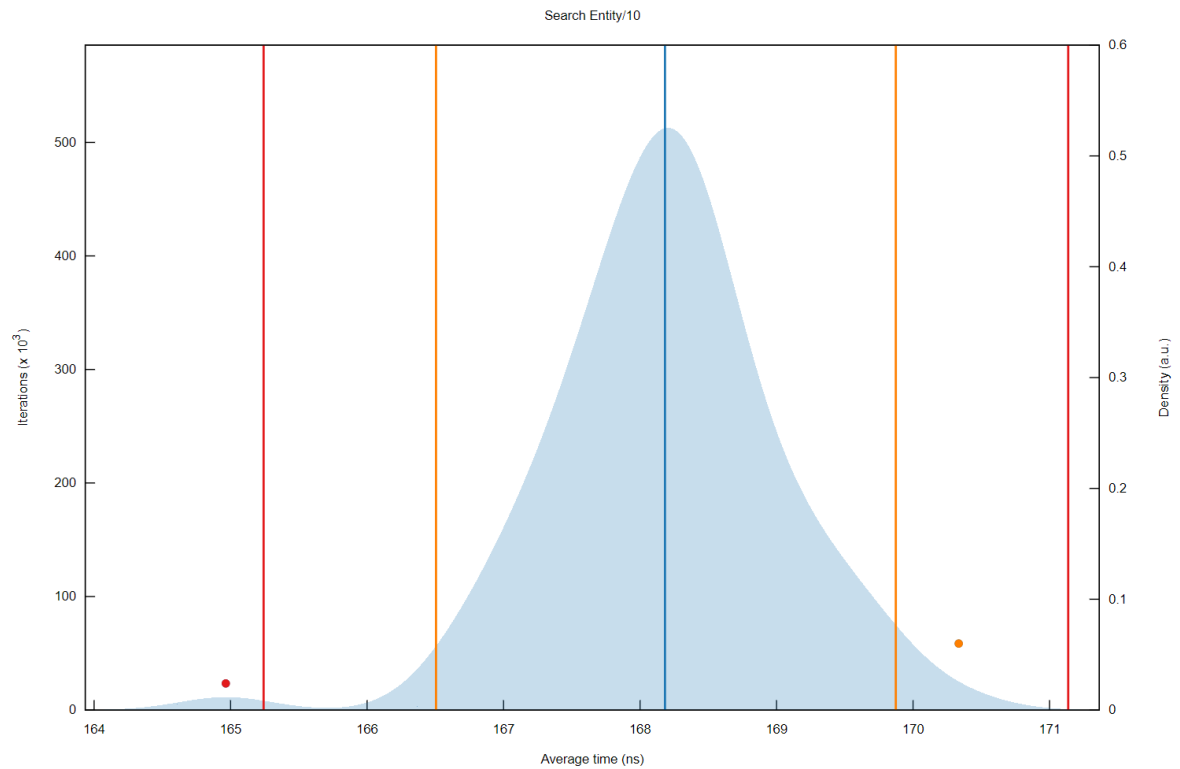


Figure 48, Searching 1000 entities.

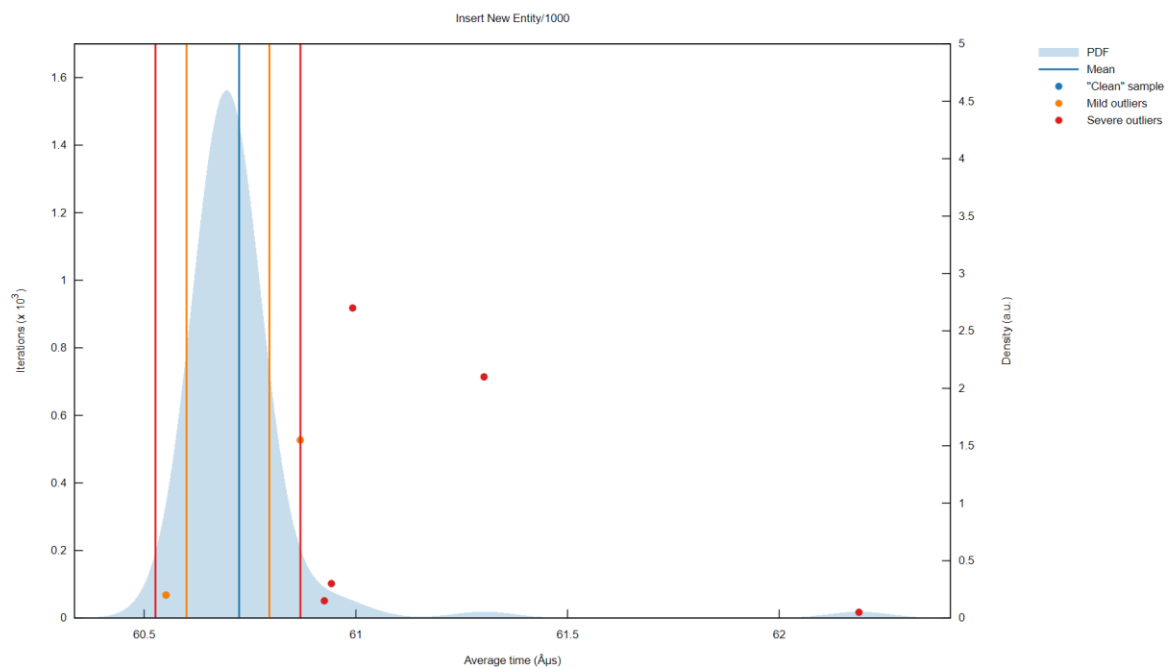


Figure 49, Inserting 1000 entities.

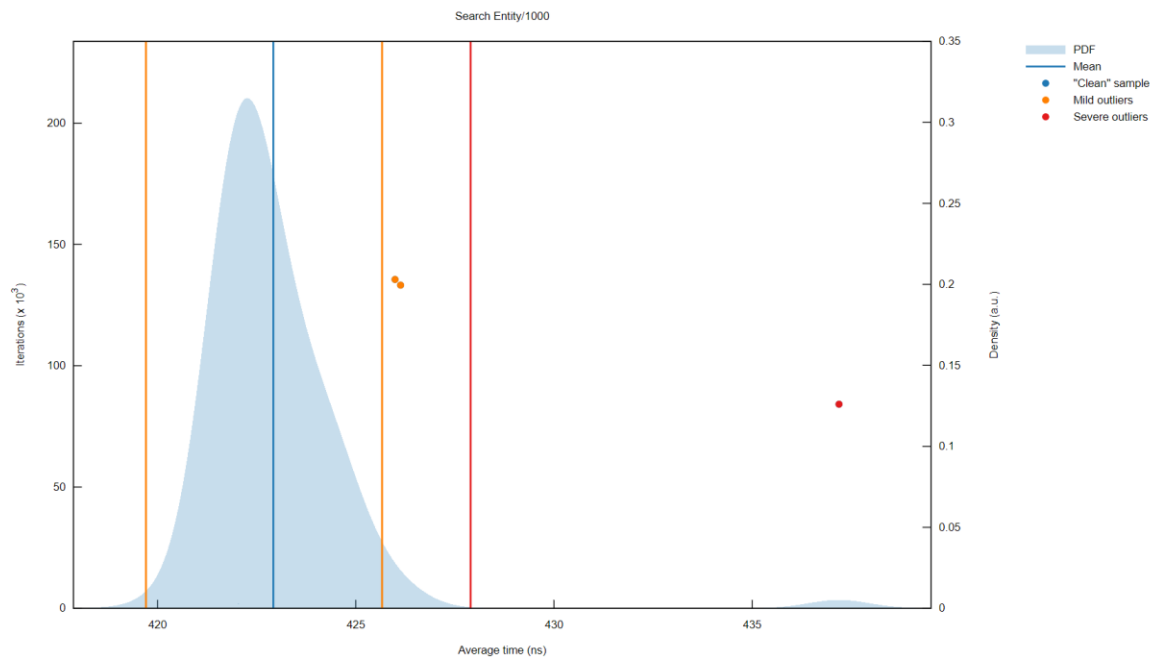


Figure 50, Searching 1000 entities.

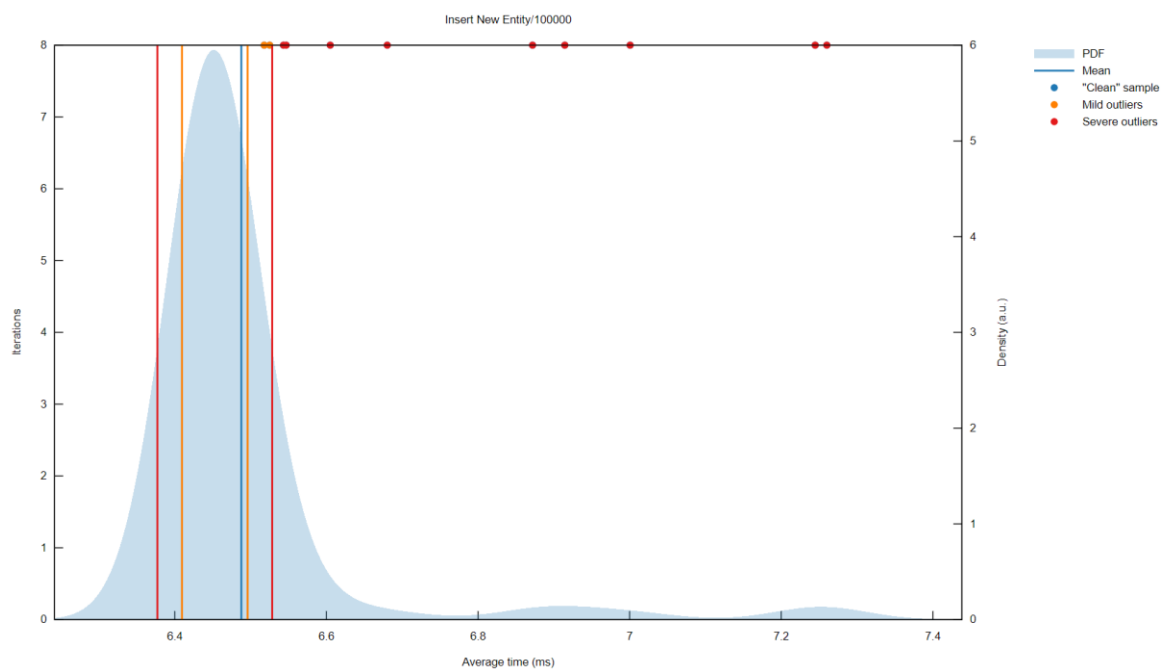


Figure 51, Inserting 100000 entities.

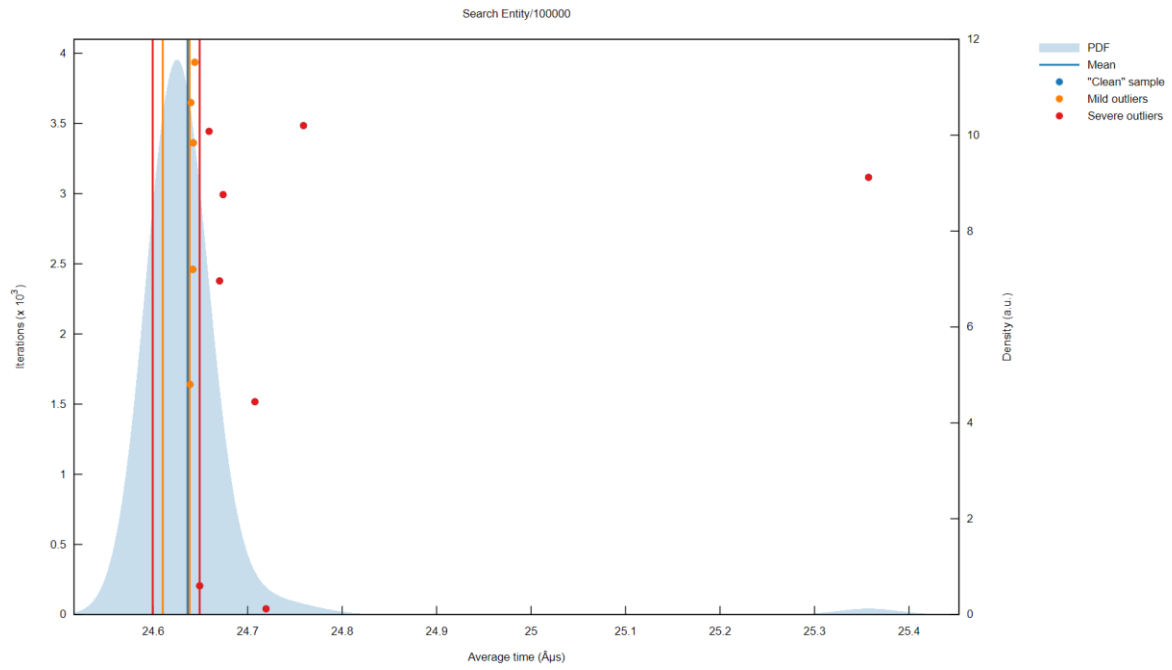


Figure 52, Searching 100000 entities.

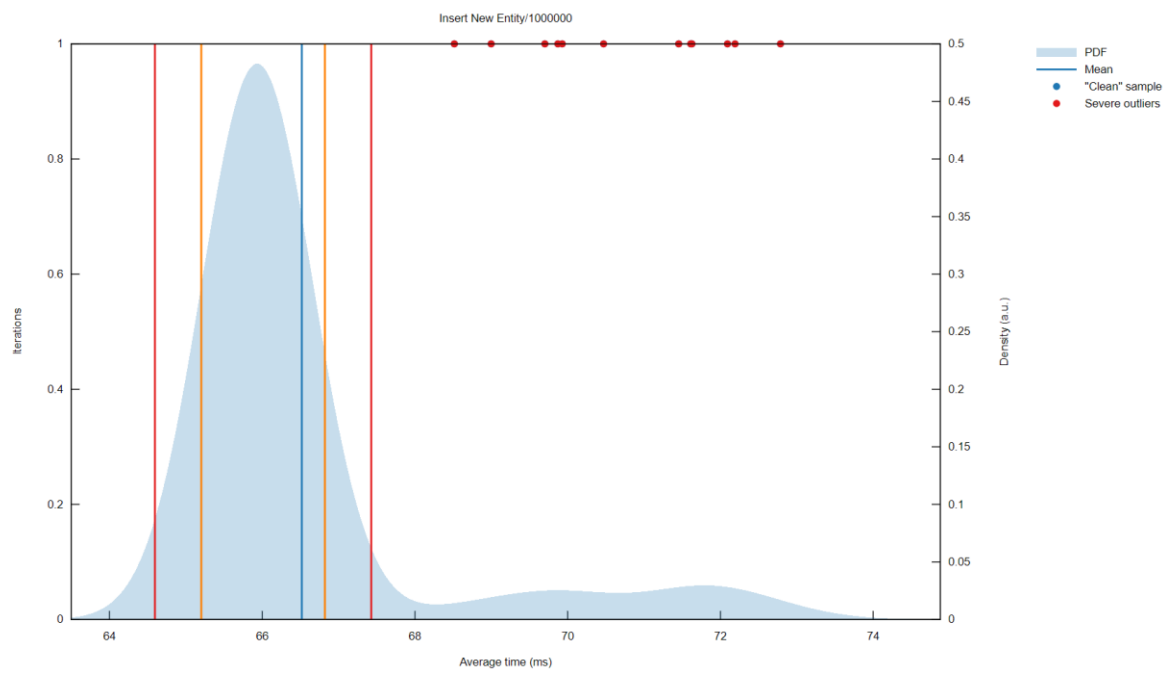


Figure 53, Inserting 1,000,000 entities.

NOTE: These benchmarks can be explored within the frost project's "target" folder.

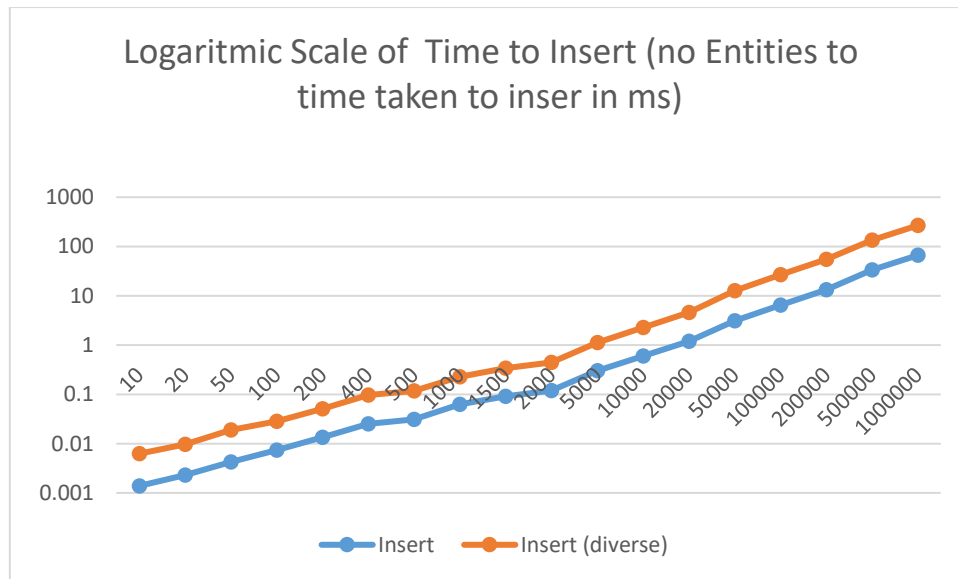


Figure 54, Time to Insert Simple vs Complex Data types into World.

Figure 44 presents a comparison between the insertion times of simple and complex data types into the game engine's world structure. The graph employs a logarithmic scale to better display the relationship between the number of entities and the time taken to insert them, measured in milliseconds (ms). Two different insertion methods are charted: a standard insertion and a "diverse" insertion, which involves inserting more varied and complex components into the structure.

The data suggests a linear relationship on a logarithmic scale, indicating that the time complexity of the insertion operation is $O(n)$. This means that the time taken to insert entities grows proportionally with the number of entities. For example, increasing the insertion from 10 to 100 entities results in a time increase would be around 4 nanoseconds. The near-parallel trend lines for both simple and diverse insertions imply that the ECS handles various data types with comparable efficiency, showcasing its ability to process different kinds of data without significant performance degradation.

The next data explores the length of time to search. This was done by running a system, and as such acted as a benchmark of the system component as well.

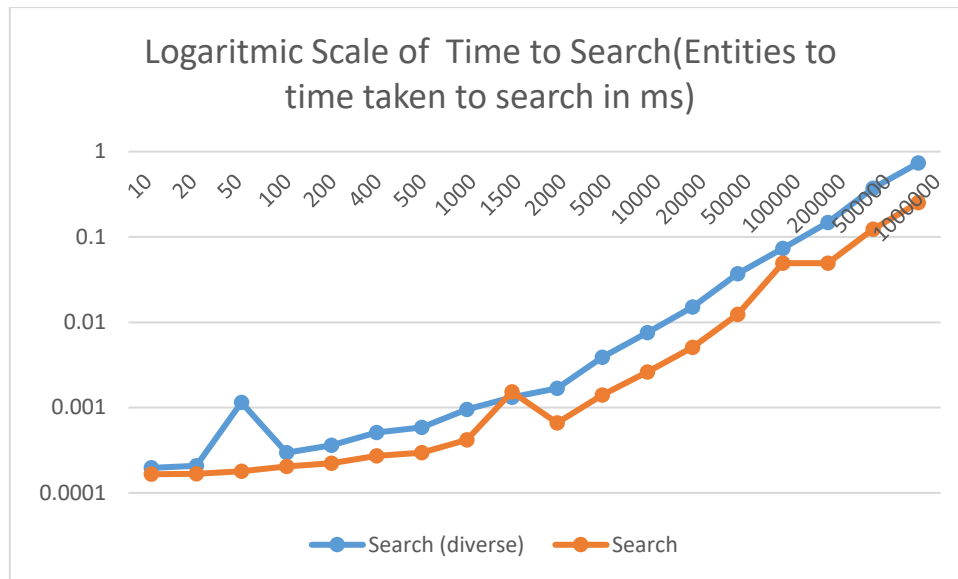


Figure 55, Time to search the world, graphed in ms.

Figure 45 explores the time performance characteristics of searching within the ECS. Here, the graph indicates the time taken to search in milliseconds (ms) on a logarithmic scale, relative to the number of entities or components within the system. Like the insertion graph, two search methods are being compared: a standard search and a "diverse" search, which searches over the recently inserted diverse data.

The search performance also exhibits a linear relationship on the logarithmic scale, which suggests a time complexity of $O(n)$. The graph shows that as the number of entities increases, the search time also increases proportionally. The overlapping of the two lines for standard and diverse searches shows that the ECS's search mechanism is optimized for both uniform and complex queries, demonstrating robust and scalable search capabilities across various types of data. The minor deviation at the lower entity counts may be due to the overhead of the search initialization or anomalies in the benchmark setup, but overall, the search time increases are consistent with the growth of the dataset.

In both figures, the ECS demonstrates efficient handling of operations across simple and complex data types. The insertion and search processes both follow a linear scalability pattern, which underscores the system's ability to maintain performance as the number of entities grows. This behaviour is critical for game engines that must handle dynamic and sizeable worlds, ensuring that gameplay remains smooth and responsive as the game's complexity increases. These performance characteristics are essential for developers who rely on the ECS to provide a stable and efficient foundation for game mechanics and entity management.

7.1.3 Evaluation

The Application Layer

Strengths

- Intuitive interface utilizing the builder pattern, allowing for smart defaults.
- Simple single package export
- Callbacks provided to interact with and run systems on Update, FixedUpdates, OnStart and OnClose
- UI can be accessed by caller and can display data directly from Frost.
- Fast and reliable implementation of fixedupdate and update calls

Weaknesses

- Systems need to be called manually and cannot be registered by the user at launch.
- Limited exposure of renderer to the engine user
- Ui interaction can be tedious as accessing render and ECS data is complex.

Issues

- Does not cleanly close and return to operating system.
- Slow start up taking roughly 3 seconds, not including compile

Lynch (Renderer)

Strengths

- Supports complex render pass combinations.
- Allows for addition or removal of passes dynamically.
- Efficient storage and retrieval of pipelines.
- Default textures allow easier prototyping.
- Rendering looks quite nice.

Weaknesses

- Does not handle many separate instances well. Could benefit from improved batching of calls, as well as better Descriptor Set allocation, instead of block allocating all sets.
- Rendering is a little bit dark now and looks much better when external exposure is increased.

Issues

- Removing models currently is nonfunctional, as the memory is not correctly reordered.
- Spawned models must be referenced via their index within the instances list.

- Shaders are recompiled every launch – could check to see if shader cache has been invalidated instead.
- Resizing swap chains and pipelines is not supported.
- Clean up of some textures and images on the graphics cars is not clean.

Frost (State Management)

Strength

- The ECS is fast and utilizes idiomatic Rust to complete its requirements.
- Creating custom Components and Systems is intuitive and unintrusive.
- Utilization of Macros to reduce amount of code repetitions.
- The physics simulation accurately calculates collisions and the location / primitive they occurred on
- Searching small sets is fast and returns mutable data, which is guaranteed to be thread safe (rwlock guarantees this).
- Data can be accessed directly allowing for special interactions (between input and camera, for example)
- Incredibly fast with many of the same archetype
- Hashing means search times are generally very fast.
- Insertion is adequately fast.

Weaknesses

- Lack of multithreading support
- Difficult to profile and provide references to data within the work.
- Searching multi-hundred components set is expensive (where the world has many components and many diverse archetypes)
- Slow when searching a diverse set of different archetypes.
- No ability to serialize or deserialize scenes.

Issues

- Referencing other entities from one entity is difficult and unintuitive.
- Parenting can only hold child structures but cannot hold an arbitrary amount of these (this is also a part of the Rust linked list issue).
- Searching for an entity itself and requires some more work from the developer.

8. Issues and Future Work

8.1 Introduction

Game engines will perpetually have features and issues that need improvement – as is the nature of any software with the scale and scope of a game engine. Below I detail the issues with the project as well as some of the risks surrounding its completion, as well as the plans for future development I have with the project.

8.2 Issues and Risks

One of the biggest challenges and issues with this project is with the architecture. Rust's borrowing system is deep, and complex and correct utilization of it is necessary to get the most out of the language, and to write idiomatic rust. An example of this is using Arc to provide Vulkan device references to resource that must be cleaned from the graphics card. Appropriate lifetime usage is both more efficient (the device is stored on the stack) and provides a clearer structure for the existence of objects (and who owns them).

The clean-up process of Vulkan is currently very messy. Many resources created are done in a fire and forget way. This goes against the principles of design for this project regarding memory ownership and resource handling. When closing the program, and destroying the Vulkan device and instance, a cascade of errors is returned from the Vulkan validation layer, regarding cleanup. The validator is the best tool in debugging these issues and ideally would be returning very little.

Currently the system is quite complex with little abstractions, and this will need to improve.

The system is single threaded and will benefit from appropriately implemented multi-threading, with channel-based communication.

8.3 Plans and Future Work

I hope to continue working on this project for the years to come, with a refactor of the rendering system planned, allowing for more engine like integration, with a customizable GUI, easy to extend engine level features, audio components, support for multiple cameras and more diverse UI planned. It has already greatly improved my abilities as a programmer and I will continue to use it as a benchmark against my own knowledge of algorithms, libraries and writing code.

9. Conclusion

Embarking on the development of this game engine represented a significant personal and professional growth opportunity for me. Initially, the challenge was not merely the intricacies of

engine design and development but also the steep learning curve associated with mastering Rust—a language known for its strict safety guarantees and complex interplay of features.

My foray into game engine architecture began with a fundamental understanding gleaned from prior usage. Concepts such as Entity Component Systems (ECS), the Separating Axis Theorem, and Vulkan API were foreign to me, as were the advanced features of Rust. The project's scope, initially underestimated, quickly expanded as the planned inter-thread communication via channels proved infeasible due to my ECS and renderer's architecture. This unforeseen complexity required a strategic pivot in my approach.

The construction of the ECS is a cornerstone achievement in this journey. It is not only fast but was also an immensely satisfying endeavour—far more so than the renderer, which, while time-consuming, offered less gratification. Rust's intricate type system presented a formidable challenge, yet harnessing its potential allowed me to forge an ECS that met my design ambitions, smoothly integrating with external libraries due to the intrinsic nature of Components and Systems as Rust types.

The renderer stands as a testament to my persistence and willingness to delve into Vulkan's profound complexity. Although I lament not allocating more time to study Vulkan thoroughly, the knowledge I've acquired on rendering and GPU communication is invaluable.

This project has underscored the criticality of effective time management. Juggling this engine's development with work, academic responsibilities, and personal life was a delicate act, and misjudgements in this area were my most significant setbacks. My initial optimism about rapid feature implementation encountered reality in complex domains like the rendering system, where time management was paramount.

Regrettably, certain aspects like multi-threading, entity serialization, and advanced physics remain underdeveloped. However, my commitment to this project remains unwavering, and I intend to continue refining these features. The absence of user acceptance testing was a disappointment, yet it has provided a rich learning experience on project pacing and user engagement.

In reflection, this endeavour, with all its imperfections, has been a profound learning experience. Navigating uncharted territory—from Rust's peculiarities to the intricacies of Vulkan, from the nuances of ECS to the principles of physics simulation—has imparted knowledge that will be enduring and instrumental in my future endeavours. I take pride in choosing a project that set me apart from my peers, one that was as unconventional as it was challenging, and I look forward to the continued evolution of this engine and the skills I have honed in the process.

10. Table of Figures

Figure 1	4
Figure 2	4
Figure 3	6
Figure 4	7
Figure 5	8
Figure 6	9
Figure 7	10
Figure 8	12
Figure 9	13
Figure 10	14
Figure 11	15
Figure 12	18
Figure 13	19
Figure 14	19
Figure 15, Kanban of Renderer	24
Figure 16, Kanban of ECS	24
Figure 17, Kanban of Engine	24
Figure 18, Cargo Crates.....	25
Figure 19, diagram of the Renderer initialization.....	28
Figure 20, diagram of the Renderer creation	28
Figure 21, shadow map generation.	30
Figure 22, Four typical G-Buffer passes.	30
Figure 23, Forward Vs Deferred.....	31
Figure 24, Nvidia FXAA demo.....	32
Figure 25, SSAO in the Source Engine	32
Figure 26, diagram of the Renderer Main loop.	33
Figure 27, Quote from Guru 99.....	33
Figure 28, diagram of the designed ECS, extended from Unity Docs ECS diagrams.....	34
Figure 29, Diagram of ECS world and layout.....	35
Figure 30, Diagram of adding Components and Entities to Archetypes.....	36
Figure 31, Diagram of searching the World.	37
Figure 32, Orientated Bounding Box.....	38
Figure 33, Separating Axis Theorem	38
Figure 34, diagram of the designed architecture with communication.	39
Figure 35, Layout of Project	54
Figure 36, Generated Shadow Map	72
Figure 37, Example of atmosphere pass running within the renderer	79
Figure 38, Sample of UI running within the engine	95
Figure 39, Samples of Cubes about to Collide	96
Figure 40, Sample of Cubes of Different Weights Colliding	96
Figure 41, Demonstration of shadows and default textures	97
Figure 42, Example of glTF Metal / Non-Metal and Smooth / Rough materials.....	97
Figure 43, The Sponza scene being rendered at 131 FPS	98
Figure 44, Time to First Frame	98
Figure 45, Render of 200 instances.....	99
Figure 46, measurement of render pass time to mesh instances.	99

Figure 47, Inserting 10 entities	100
Figure 48, Searching 1000 entities.....	101
Figure 49, Inserting 1000 entities.	101
Figure 50, Searching 1000 entities.....	102
Figure 51, Inserting 100000 entitites.....	102
Figure 52, Searching 100000 entities.....	103
Figure 53, Inserting 1,000,000 entities.	103
Figure 54, Time to Insert Simple vs Complex Data types into World.	104
Figure 55, Time to search the world, graphed in ms	105

Bibliography

1. Endler M. mre/idiomatic-rust [Internet]. 2023 [cited 2023 Dec 4]. Available from: <https://github.com/mre/idiomatic-rust>
2. Technologies U. Unity - Manual: Feature sets [Internet]. [cited 2023 Nov 25]. Available from: <https://docs.unity3d.com/Manual/FeatureSets.html>
3. Gomez N. Headsem.com. 2017 [cited 2023 Nov 25]. Godot Engine, Motor de Videojuegos Open Source. Available from: <https://www.headsem.com/godot-engine-el-motor-de-videojuegos-open-source-mas-completo/>
4. Ampomah E, Mensah E, Gilbert A. Qualitative Assessment of Compiled, Interpreted and Hybrid Programming Languages. Commun Appl Electron. 2017 Oct 25;7:8–13.
5. Delporte F. Azul | Better Java Performance, Superior Java Support. 2022 [cited 2023 Dec 8]. JIT Performance: Ahead-Of-Time versus Just-In-Time. Available from: [https://www.azul.com/blog/jit-performance-ahead-of-time-versus-just-in-time/](https://www Azul.com/blog/jit-performance-ahead-of-time-versus-just-in-time/)
6. Oyez [Internet]. [cited 2024 Apr 5]. Google LLC v. Oracle America Inc. Available from: <https://www.oyez.org/cases/2020/18-956>
7. Why is operator overloading not supported by java? [Internet]. [cited 2023 Nov 25]. Available from: <https://www.tutorialspoint.com/why-is-operator-overloading-not-supported-by-java>
8. The Go Programming Language [Internet]. [cited 2023 Nov 29]. Available from: <https://go.dev/>
9. Pakshina N. Medium. 2023 [cited 2023 Nov 29]. Memory Optimization and Garbage Collector Management in Go. Available from: <https://betterprogramming.pub/memory-optimization-and-garbage-collector-management-in-go-71da4612a960>
10. Bitfield Consulting [Internet]. 2024 [cited 2024 Apr 5]. Rust vs Go in 2024. Available from: <https://bitfieldconsulting.com/golang/rust-vs-go>
11. Golang Performance: Go Programming Language vs. Other Languages [Internet]. [cited 2024 Apr 5]. Available from: <https://www.orientsoftware.com/blog/golang-performance/>
12. Tyson M. InfoWorld. 2023 [cited 2023 Nov 28]. Meet Zig: The modern alternative to C. Available from: <https://www.infoworld.com/article/3689648/meet-the-zig-programming-language.html>
13. Jana S. ValorZard/awesome-zig-gamedev [Internet]. 2024 [cited 2024 Apr 5]. Available from: <https://github.com/ValorZard/awesome-zig-gamedev>
14. Aibyn A, Maikarina A. The Role of C++ in Game Development: Case Studies and Performance Analysis. 2023 Nov 9;
15. Grant H, TechBullion ASB. The History of C++ Coding in Video Games [Internet]. TechBullion. 2023 [cited 2024 Apr 5]. Available from: <https://techbullion.com/the-history-of-c-coding-in-video-games/>

16. Zero Cost Abstractions - The Embedded Rust Book [Internet]. [cited 2024 Apr 5]. Available from: <https://doc.rust-lang.org/beta/embedded-book/static-guarantees/zero-cost-abstractions.html>
17. Arindam. DebugPoint.com. 2024 [cited 2024 Apr 5]. Microsoft's Strategic Move: C# to Rust, But Why? [Opinion]. Available from: <https://www.debugpoint.com/rust-c-microsoft/>
18. ZDNET [Internet]. [cited 2024 Apr 5]. Rust in Linux: Where we are and where we're going next. Available from: <https://www.zdnet.com/article/rust-in-linux-where-we-are-and-where-were-going-next/>
19. Stack vs Heap Allocation in C: Pros and Cons [Internet]. [cited 2023 Nov 28]. Available from: <https://www.matecddev.com/posts/c-heap-vs-stack-allocation.html>
20. Group ISR. Prossimo. [cited 2023 Nov 28]. What is memory safety and why does it matter? Available from: <https://www.memorysafety.org/docs/memory-safety/>
21. November 05 MKB, 2015. Game Developer. 2015 [cited 2023 Nov 28]. Writing a Game Engine from Scratch - Part 2: Memory. Available from: <https://www.gamedeveloper.com/programming/writing-a-game-engine-from-scratch---part-2-memory>
22. Berg Marklund B, Engström H, Hellkvist M, Backlund P. What Empirically Based Research Tells Us About Game Development. *Comput Games J*. 2019 Dec 1;8(3):179–98.
23. OpenGL 4.0 (Core Profile) - March 11, 2010. 2010;
24. Lujan M, Baum M, Chen D, Zong Z. Evaluating the Performance and Energy Efficiency of OpenGL and Vulkan on a Graphics Rendering Server. In: 2019 International Conference on Computing, Networking and Communications (ICNC) [Internet]. 2019 [cited 2023 Dec 8]. p. 777–81. Available from: <https://ieeexplore.ieee.org/document/8685588>
25. OpenGL vs. DirectX - what really happened? | Back2Gaming [Internet]. 2019 [cited 2023 Dec 8]. Available from: <https://www.back2gaming.com/review/pc-games/opengl-vs-directx-what-really-happened/>
26. Discuss btarunr. TechPowerUp. 2023 [cited 2023 Dec 8]. Is DirectX 12 Worth the Trouble? Available from: <https://www.techpowerup.com/231079/is-directx-12-worth-the-trouble>
27. MoltenVK | Run Vulkan on iOS and OS X [Internet]. Molten. [cited 2023 Dec 8]. Available from: <https://moltengl.com/moltenvk/>
28. Blog post: The Benefits of Vulkan - Imagination [Internet]. [cited 2023 Dec 8]. Available from: <https://blog.imaginationtech.com/stuck-on-opengl-es-time-to-move-on-why-vulkan-is-the-future-of-graphics/>
29. Robertson K. MUO. 2022 [cited 2023 Dec 8]. DirectX 11 vs. DirectX 12: What Are the Differences and Which Should You Use? Available from: <https://www.makeuseof.com/directx-11-vs-directx-12-differences/>
30. wgpu [Internet]. Rust Graphics Mages; 2023 [cited 2023 Nov 28]. Available from: <https://github.com/gfx-rs/wgpu>
31. amethyst/legion [Internet]. Amethyst Foundation; 2024 [cited 2024 Apr 4]. Available from: <https://github.com/amethyst/legion>

32. Specs and Legion, two very different approaches to ECS [Internet]. [cited 2024 Apr 4]. Available from: <https://csherratt.github.io/blog/posts/specs-and-legion/>
33. gebbdoo.pdf [Internet]. [cited 2023 Nov 28]. Available from: <https://fabiansanglard.net/b/gebbdoo.pdf>
34. Gillin P. Mendix. 2024 [cited 2024 Apr 5]. What is Component-Based Architecture? Available from: <https://www.mendix.com/blog/what-is-component-based-architecture/>
35. Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP) – Games from Within [Internet]. 2009 [cited 2024 Apr 4]. Available from: <https://gamesfromwithin.com/data-oriented-design>
36. Single Instruction Multiple Data - an overview | ScienceDirect Topics [Internet]. [cited 2024 Apr 4]. Available from: <https://www.sciencedirect.com/topics/computer-science/single-instruction-multiple-data>
37. Features :: Vulkan Documentation Project [Internet]. [cited 2024 Apr 7]. Available from: <https://docs.vulkan.org/spec/latest/chapters/features.html>
38. Vulkan Guide [Internet]. [cited 2024 Apr 4]. Executing Vulkan Commands. Available from: https://www.vkguide.dev/docs/chapter-1/vulkan_command_flow/
39. Vulkan Guide [Internet]. [cited 2024 Apr 4]. The render pipeline. Available from: https://www.vkguide.dev/docs/chapter-2/vulkan_render_pipeline/
40. Vulkan 1.2.197 Released With Dynamic Rendering Extension [Internet]. [cited 2024 Apr 4]. Available from: <https://www.phoronix.com/news/Vulkan-1.2.197-Released>
41. Loggini R. Riccardo Loggini. 2021 [cited 2024 Apr 5]. Render Graphs. Available from: <https://logins.github.io/graphics/2021/05/31/RenderGraphs.html>
42. MJP. Light Indexed Deferred Rendering [Internet]. The Danger Zone. 2012 [cited 2024 Apr 4]. Available from: <https://mynameismjp.wordpress.com/2012/03/31/light-indexed-deferred-rendering/>
43. Working with lights and shadows – Part II: The shadow map – Developer Log [Internet]. 2017 [cited 2024 Apr 4]. Available from: <https://blogs.igalia.com/itoral/2017/07/30/working-with-lights-and-shadows-part-ii-the-shadow-map/>
44. LearnOpenGL - Shadow Mapping [Internet]. [cited 2024 Apr 5]. Available from: <https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping>
45. LearnOpenGL - Deferred Shading [Internet]. [cited 2024 Apr 5]. Available from: <https://learnopengl.com/Advanced-Lighting/Deferred-Shading>
46. Lettier D. Deferred Rendering | 3D Game Shaders For Beginners [Internet]. [cited 2024 Apr 5]. Available from: <https://lettier.github.io/3d-game-shaders-for-beginners/deferred-rendering.html>
47. VkPresentModeKHR(3) [Internet]. [cited 2024 Apr 5]. Available from: <https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VkPresentModeKHR.html>
48. G2A.COM. G2A News. 2023 [cited 2024 Apr 5]. FXAA vs MSAA: Which is Better for Gaming? Available from: <https://www.g2a.com/news/features/msaa-vs-fxaa/>

49. Choi. choi303/FXAA [Internet]. 2024 [cited 2024 Apr 5]. Available from: <https://github.com/choi303/FXAA>
50. What is Ambient Occlusion? Does it Matter in Games? [Internet]. 2021 [cited 2024 Apr 5]. Available from: <https://thewiredshopper.com/ambient-occlusion/>
51. LearnOpenGL - SSAO [Internet]. [cited 2024 Apr 5]. Available from: <https://learnopengl.com/Advanced-Lighting/SSAO>
52. Alexey Naumov [Internet]. 2020 [cited 2024 Apr 5]. Separation of Concerns in Software Design. Available from: <https://nalexn.github.io/separation-of-concerns/>
53. Entity Component System [Internet]. 2024 [cited 2024 Apr 7]. Available from: <https://www.guru99.com/entity-component-system.html>
54. Simplilearn.com [Internet]. 2022 [cited 2024 Apr 5]. Entity Component System: An Introductory Guide | Simplilearn. Available from: <https://www.simplilearn.com/entity-component-system-introductory-guide-article>
55. Codecademy [Internet]. [cited 2024 Apr 5]. Entity-Component-System. Available from: <https://www.codecademy.com/article/a-frame-entity-component-system>
56. ECS concepts | Package Manager UI website [Internet]. [cited 2024 Apr 4]. Available from: https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/ecs_core.html
57. What's an Entity System? - Entity Systems Wiki [Internet]. [cited 2024 Apr 4]. Available from: <http://entity-systems.wikidot.com/>
58. Alex Jobe [Internet]. [cited 2024 Apr 4]. Available from: <https://www.alexjobe.net/posts/ecs>
59. Raygun Blog [Internet]. 2023 [cited 2024 Apr 4]. The hidden impact of cache locality on application performance. Available from: <https://raygun.com/blog/cache-locality-impact-application-performance/>
60. Compton BV. An Investigation of Data Storage in Entity-Component Systems.
61. Gopherfest 2015 | Go Proverbs with Rob Pike - YouTube [Internet]. [cited 2023 Nov 30]. Available from: <https://www.youtube.com/watch?v=PAAkCSZUG1c&t=168s>
62. Programmer Art [Internet]. [cited 2024 Apr 10]. Separating Axis Theorem. Available from: <http://programmerart.weebly.com/separating-axis-theorem.html>
63. Code Envato Tuts+ [Internet]. 2012 [cited 2024 Apr 10]. Collision Detection Using the Separating Axis Theorem | Envato Tuts+. Available from: <https://code.tutsplus.com/collision-detection-using-the-separating-axis-theorem--gamedev-169t>
64. Separating Axis Theorem :: K-State CIS 580 Textbook [Internet]. [cited 2024 Apr 10]. Available from: <https://textbooks.cs.ksu.edu/cis580/04-collisions/04-separating-axis-theorem/index.html>
65. Oriented Bounding Box - an overview | ScienceDirect Topics [Internet]. [cited 2024 Apr 10]. Available from: <https://www.sciencedirect.com/topics/computer-science/oriented-bounding-box>
66. chadogome/OptimalOBB [Internet]. chadogome; 2024 [cited 2024 Apr 10]. Available from: <https://github.com/chadogome/OptimalOBB>

67. Slemgrim. Translation direction in separating axis theorem [Internet]. Stack Overflow. 2012 [cited 2024 Apr 10]. Available from: <https://stackoverflow.com/q/10396068>
68. Packages and Crates - The Rust Programming Language [Internet]. [cited 2024 Apr 7]. Available from: <https://doc.rust-lang.org/book/ch07-01-packages-and-crates.html>
69. Introduction - Vulkan Tutorial [Internet]. [cited 2023 Nov 28]. Available from: <https://vulkan-tutorial.com/>
70. VK_EXT_descriptor_indexing(3) [Internet]. [cited 2024 Apr 7]. Available from: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_EXT_descriptor_indexing.html
71. VK_KHR_dynamic_rendering(3) [Internet]. [cited 2024 Apr 7]. Available from: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_KHR_dynamic_rendering.html
72. Running Code on Cleanup with the Drop Trait - The Rust Programming Language [Internet]. [cited 2024 Apr 8]. Available from: <https://doc.rust-lang.org/book/ch15-03-drop.html>
73. gpu-allocator - crates.io: Rust Package Registry [Internet]. 2024 [cited 2024 Apr 7]. Available from: <https://crates.io/crates/gpu-allocator>
74. KhronosGroup/SPIRV-Reflect [Internet]. The Khronos Group; 2024 [cited 2024 Apr 8]. Available from: <https://github.com/KhronosGroup/SPIRV-Reflect>
75. Staging buffer - Vulkan Tutorial [Internet]. [cited 2024 Apr 8]. Available from: https://vulkan-tutorial.com/Vertex_buffers/Staging_buffer
76. Uniform Buffer Object - OpenGL Wiki [Internet]. [cited 2024 Apr 5]. Available from: https://www.khronos.org/opengl/wiki/Uniform_Buffer_Object
77. Uniform (GLSL) - OpenGL Wiki [Internet]. [cited 2024 Apr 5]. Available from: [https://www.khronos.org/opengl/wiki/Uniform_\(GLSL\)](https://www.khronos.org/opengl/wiki/Uniform_(GLSL))
78. Collabora | Open Source Consulting [Internet]. [cited 2024 Apr 5]. Exploring Rust for Vulkan drivers, part 1. Available from: <https://www.collabora.com/news-and-blog/blog/2023/02/02/exploring-rust-for-vulkan-drivers-part-1/>
79. Karlsson B. baldurk/renderdoc [Internet]. 2024 [cited 2024 Apr 5]. Available from: <https://github.com/baldurk/renderdoc>