Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Exploring Game Engine Architecture and building an experimental Voxel Renderer with Rust and Vulkan

Bachelor's Project

Moritz F. Kuntze

mkuntze@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Quentin Kniep
Prof. Dr. Roger Wattenhofer

December 2022

Acknowledgements

First and foremost, I would like to thank my family — my mother Martina, my father Ulrich, and my brothers Marius and Jan — for their continued support throughout my life and my studies, always enabling me to pursue my dreams and explore my interests.

Secondly, I would like to extend my gratitude to my supervisors, Prof. Dr. Roger Wattenhofer and Quentin Kniep for allowing me to undertake this project as a part of their group.

Finally, I would like to acknowledge Yan Chernikov, who singlehandedly re-ignited my interest in game engine development through his YouTube channel The Cherno, and Victor Blanco, whose Vulkan Guide allowed my to better understand the Vulkan API, allowing me to implement this project in the first place.

Abstract

This project implements *Orion*, a partial modular game engine fully written in the Rust programming language and using the Vulkan graphics API as its rendering backend. Additionally, this project demonstrates how *Orion* can be extended by implementing an experimental rendering system for voxel-based geometry. *Orion* is shown to be capable of rendering scenes with many large triangle meshes and quickly converting large voxel objects into mesh-based geometry through the use of compute shaders.

Contents

Chapter 1

Introduction

Rust is a general-purpose programming language, originally started by Graydon Hoare at Mozilla [12] and now developed by volunteers under the umbrella of the Rust Foundation. It has gained significant traction over the past years, rising to no. 20 in the TIOBE index in Dec. 2022 [13] and being rated "Most loved programming language" in the StackOverflow developer survey for seven years in a row [14]–[20]. It claims to be a high-performance language [21], being a natively compiled language, making heavy use of zero-cost abstractions and ensuring memory safety without the use of a garbage collector through a unique compile-time borrow checker system. It is heralded by some as a potential replacement for C and C++ [22] and was recently made the second official language of the Linux kernel [23].

Game engines are the backbone of the modern games industry, an industry that is estimated to have a global revenue of 196 billion US$ [24], making it larger than the global film industry [25]. As one of the most complex software systems in wide deployment that are typically written in a low-level language like C++ for performance reasons [1] it is a natural question whether Rust would be a suitable replacement for C++ in this domain of software development. In addition to the question of whether the language and its features by itself are a suitable match for common abstractions and patterns found in game engine development, the question arises, whether the Rust ecosystem (in particular the available libraries) are in a state that allows the development of a game engine without implementing everything from scratch or having to resort to the C foreign function interface too often.

The goal of this project was to answer these questions by attempting to implement a (partial) game engine in Rust and making use of available libraries where it makes sense, noting the quality of these libraries and situations where a library would be useful but none (or none that is sufficiently advanced) exists.

The Vulkan graphics API was chosen to enable the rendering and general-purpose GPU needs of this work, as it is the most modern cross-platform graphics API available (with OpenGL being the alternative).

1.1 Structure of this report

This report is split into the following parts: Chapter 2 introduces some background information on game engines in general and their architecture in particular, in addition to introducing some background on the GPU rasterization rendering pipeline and projective geometry.

Chapter 3 describes the architecture of the *Orion* game engine implemented in this project and some implementation details, including details on the triangle mesh renderer that is part of Orion.

Chapter 4 implements an experimental voxel renderer within the framework of the Orion engine

described in chapter 3. This voxel renderer is evaluated by its own merit as a voxel renderer but also serves as an evaluation tool for the architecture of the Orion engine by reporting the ease of adding additional functionality to the base engine.

Finally, chapter 5 evaluates the current state of the Rust ecosystem and some of the aspects of the Rust programming language with regard to the ease of game engine development in Rust.

Chapter 2

Background

2.1 Fundamentals of Game Engines

The Oxford English Dictionary defines a game engine as "a software framework underlying a video game, which performs basic functions such as graphics rendering and sound playback" [26]. This definition is correct, although it omits the manifold other subsystems and tools that define modern commercial game engines like Epic Games' *Unreal Engine* and make them not only the software framework underpinning video games but also the toolset enabling their development in the first place and fulfilling functions that are more and more complicated as time and the industry moves on. *Game Engine Architecture* [1, p. 39] shows an overview of common systems found in modern game engines.

In terms of architecture, a game engine can be split into three kinds of subsystems:

1. *Supporting libraries*, providing standalone functionality that is independent of other engine functions.
2. *Supporting subsystems*, being subsystems that are responsible for providing data or functionality that is used by other subsystems and enable those subsystems to interact with each other.
3. *Core subsystems*, being subsystems that are central to the gameplay experience and the production of games in general.

Supporting libraries include functionality such as providing fast mathematics, windowing and other operating system interfaces.

Supporting subsystems include functionality for debugging or logging, event handling, asset management and game state management. Game state management is a particularly important supporting system as it interacts with almost all other components of the engine, most of which depend on or modify the state of the game world.

Core subsystems are systems like the graphics renderer, the audio engine, the physics engine or the animation system. These systems each enable vital parts of the user experience but usually don't directly interact with each other but instead interact via a supporting system (the notable exception being the animation system which usually interacts heavily with both the renderer and the physics engine).

Game engines arose with the growth of the video game industry in the late 1980s through the desire to re-use existing codebases for game development and as an additional revenue stream for game studios licensing their engine to other studios. Many of the most widely used game engines in industry have a long development history reaching back into the 1990s for some (such as the aforementioned *Unreal Engine*, which was originally released in 1998 and Valve

Software's *Source* engine, which derives from the *Quake* engine, which was used in *id Software's* 1996 release *Quake*).

2.2 Fundamentals of Rust

Rust is a multi-paradigm, general-purpose programming language with a focus on high-performance and safe code. In particular, Rust is compiled to machine code through LLVM and attempts to ensure memory safety, type safety and thread safety through various mechanisms in its type system. Rust's *borrow checker* allows the compiler to track an object's lifetime and scope at compile time to ensure memory safety and prevent unsafe concurrent accesses without the use of a garbage collector or automatic reference counting.

Rust is syntactically similar to other C-like languages like C, C++ or Swift but makes heavy use of type inference, syntactic sugar and zero-cost abstractions to make its code seem more like higher-level, dynamically typed languages like JavaScript. Rust's standard library makes heavy use of generics which are monomorphized at compile time.

While Rust has features of object-oriented languages, such as the ability to attach methods to a data structures, it lacks some features common in object-oriented languages, in particular the concept of inheritance. Common functionality is described through *traits* which are similar to Java's *interfaces* but cannot inherit from each other.

One of the core principles of Rust is the notion of "safe" and "unsafe" code, aimed at preventing undefined behavior (UB) from ocurring in Rust. Safe code should not be able to cause UB and thus cannot perform certain actions that may cause UB, such as dereferencing a raw pointer (which may be null or out of bounds). Unsafe code (marked using the `unsafe` keyword) can perform these actions and it is up to the developer to guarantee that no UB occurs, creating a safe wrapper.

2.3 Fundamentals of Real-Time 3D Graphics

Real-time 3D computer graphics — specifically real-time rendering — is the process of computationally producing depictions of a three-dimensional scene at a frame rate that allows interactivity. The common boundary for real-time rendering is set at 24 or 30 frames rendered per second (FPS). Rendering complex scenes in a realistic manner at these frame rates requires the use of hardware graphics accelerators, commonly known as GPUs. This section introduces the *graphics pipeline*, the set of fixed-function, configurable and programmable stages implemented by modern GPUs to enable real-time rasterization rendering.

Rasterization rendering is the commonly used term for the mechanism by which modern graphics hardware produces images and is often contrasted with *ray tracing*, the technique often used for photorealistic rendering in non-real time rendering applications. Ray tracing techniques work by simulating the behavior of light rays bouncing around in a virtual scene and interacting with materials in the scene to produce an image via the interaction of these rays with a virtual camera or eye. Rasterization techniques, on the other hand, do not fully simulate light rays but project the scene's geometry onto a virtual plane, which is then rasterized, i.e. converted to pixels, sometimes also called *fragments*, in rasterization rendering [2, p. 21].

2.3.1 Coordinates and Transforms

Homogeneous Coordinates

Most 3D rasterization renderers operate using four-dimensional *homogeneous coordinates* instead of the usual three-dimensional Euclidean coordinate system. This coordinate system confers the advantage that affine transformations, i.e. transformations that include a translation, can be represented using matrices and applied through regular matrix-vector multiplication [3, p. 74]. Additionally, homogeneous coordinates allow us to represent projective transformations as matrices as well.

In homogenous coordinates, we add a fourth real-valued coordinate to our vectors, usually denoted as w. Each point (x, y, z) in 3D Euclidean space is represented by a line in 4D homogeneous space defined by (xw, yw, zw, w) where $w \neq 0$. Points in homogeneous space where $w = 0$ do not have a direct equivalent in 3D Euclidean space and are considered "points at infinity".

Conceptually, we can differentiate between *points* and *vectors* in 3D space using homogeneous coordinates: Vectors are not affected by translation and represented using points at infinity with a w coordinate set to 0 (i.e. as $(x, y, z, 0)$). Points in 3D space are affected by translation and feature nonzero w coordinates in homogeneous space (usually $w = 1$, as this avoids multiplications or divisions of the first three coordinates when converting from or to Euclidean space).

Common Transforms

To further illustrate how homogeneous coordinates are useful, we examine a few common transformations we can perform using homogenous coordinates. Firstly, we can perform any rotation, scaling or shearing transformations (or any general 3D linear transformation) represented by a 3×3 matrix L using homogenous coordinates by augmenting the matrix L as follows to produce a 4×4 matrix

$$L_H = \left(\begin{array}{c|c} L & \mathbf{0} \\ \hline \mathbf{0}^T & 1 \end{array}\right)$$

where $\mathbf{0}$ represents the three-dimensional zero vector.

A translation by the vector (t_x, t_y, t_z) can be represented using the 4×4 matrix

$$T_H = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

One can easily observe that multiplying this matrix by any homogenous point (xw, yw, zw, w) yields the point $(xw + t_x w, yw + t_y w, zw + t_z w, w)$. Dividing the x, y and z coordinates by w to retrieve the corresponding 3D point yields $(x + t_x, y + t_y, z + t_z)$, showing the desired effect. One can similarly observe that the translation matrix has no effect on points at infinity.

Finally, we examine how a perspective transformation can be expressed as a matrix using homogeneous coordinates. Perspective transformations transform points inside the so-called *view frustrum* into the *canonical view volume*, expressed in *normalized device coordinates*. Perspective transforms must not be confused with perspective projections, the latter being non-invertible transformations that project points (expressed in homogeneous coordinates) from 3D space onto a 2D plane. Perspective projection is the result of applying a perspective transform and then discarding the z (and w) coordinate.

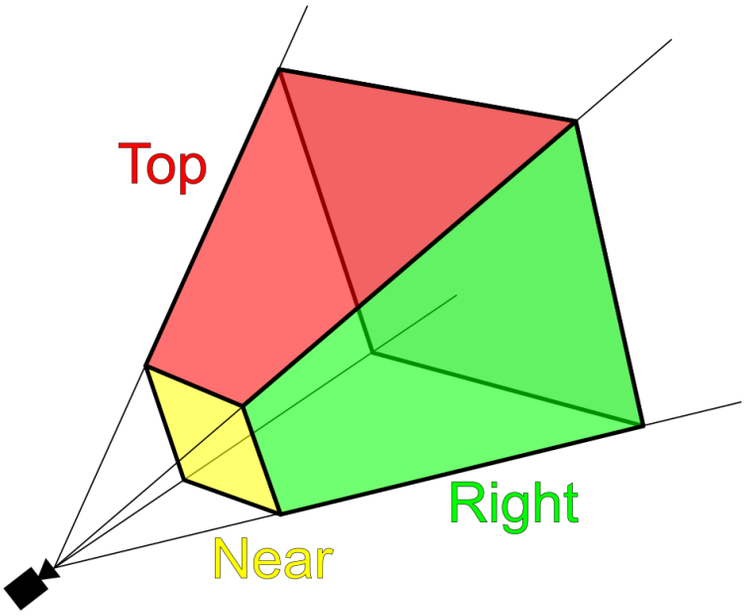

Figure 2.1: An exemplary view frustrum with the near, top and right clipping planes highlighted.

The view frustrum, as depicted in figure 2.1, can be defined through several equivalent sets of parameters; in this case we will use the parameters n, being the distance of the *near clipping plane*, f, being the distance of the *far clipping plane*, ϕ being the *horizontal field of view* and $a = \frac{\text{width}}{\text{height}}$ being the *aspect ratio* of the view frustrum. We choose our source coordinate system such that the camera is located at the origin, the y-axis is pointing down and the camera is facing the z direction[1], call this coordinate system *view space* and denote coordinates in view space using a subscript v. The near clipping plane is given as the rectangle at

$$z_v = n$$

with

$$x_v \in \left[-n \cdot \tan\frac{\phi}{2}, n \cdot \tan\frac{\phi}{2}\right]$$

and

$$y_v \in \left[-\frac{n}{a} \cdot \tan\frac{\phi}{2}, \frac{n}{a} \cdot \tan\frac{\phi}{2}\right].$$

The far clipping plane is defined similarly, replacing n by f. The side clipping planes are defined by connecting the edges of the near and far clipping plane. The goal of the perspective transform is to transform this view frustrum into the canonical view volume, which is a cuboid given by $x_d, y_d, z_d \in [-1, 1] \times [-1, 1] \times [0, 1]$ in Vulkan [4, Sec. 26 .9]. We call this final coordinate system *normalized device coordinates* (NDC) and denote NDC coordinates using a subscript d.

To derive the transformation matrix, we examine the transformations of each coordinate of our source point (x_v, y_v, z_v) individually: The x_v coordinate of our source point is transformed according to

$$x_d = \frac{1}{z_v} \cdot \frac{x_v}{\tan\frac{\phi}{2}}.$$

[1] this coordinate system is chosen for ease of derivation, as it matches Vulkan's conventions; note that *Orion* uses a slightly different view space coordinate system as described below.

Similarly, the y_v coordinate is transformed according to

$$y_d = \frac{1}{z_v} \cdot \frac{y_v \cdot a}{\tan \frac{\phi}{2}}.$$

Finally, for the z_v coordinate we seek a transformation of the form

$$z_d = \frac{1}{z_v} (Az_v + B)$$

that maps $z_v = n$ to $z_d = 0$ and $z_v = f$ to $z_d = 1$. Solving this system of equations yields

$$A = \frac{f}{f - n} \quad \text{and} \quad B = \frac{n \cdot f}{n - f}.$$

As we can clearly see, each of these transformation equations consist of some affine transformation, followed by a division by z_d. As the conversion from homogeneous coordinates to Euclidean coordinates involves a division by the w coordinate, we can implement this division by z_d by placing z_d into the w coordinate of the result of our transformation. We also assume that the inputs to our transformation matrix have $w = 1$. This yields the perspective transformation matrix

$$P = \begin{pmatrix} \frac{1}{\tan \frac{\phi}{2}} & 0 & 0 & 0 \\ 0 & \frac{a}{\tan \frac{\phi}{2}} & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & \frac{n \cdot f}{n-f} \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Coordinate Systems

In 3D rasterization rendering, we use multiple coordinate systems, related through homogeneous coordinate transforms. The coordinate systems used by *Orion* are the following:

Each mesh is defined in a coordinate system called *model space*. This is the coordinate system in which a model's vertices are defined when loading that particular mesh from disk and it is defined with respect to some arbitrary *model origin*, e.g. the model's center of gravity.

The game's world is defined in a coordinate system called *world space*, defined by some arbitrary *world origin*. All objects, including the camera, are placed in world space. A mesh's model space coordinates can be transformed into world space through the *model transform* M which is given by

$$M = T \cdot R \cdot S$$

where S is a scale transform, R is a rotation and T is a translation.

For rendering, all objects are transformed into *view space*, which places the camera at the origin. In *Orion*, view space is defined as a right-handed, y-up coordinate system, i.e. the camera's "right" direction is the x-axis, the "up" direction is the y-axis and the camera is looking in the negative z direction. World space coordinates can be transformed into view space with the view transform V.

For rendering, points are transformed from view space into (four-dimensional) *clip space* using a perspective projection matrix P. Perspective division (division by the w coordinate) transforms points from clip space into 3D normalized device coordinates.

Finally, the *viewport transform* transforms normalized device coordinates into pixel coordinates inside the game's viewport before *screen mapping* places the viewport on the user's monitor.

2.3.2 The Graphics Pipeline

The graphics pipeline, for the purposes of this discussion, can be split into four major stages [2, p. 13]:

1. the application stage
2. the geometry processing stage
3. the rasterization stage
4. the pixel/fragment processing stage

This general pipeline, including substages, is shown in figure 2.2.

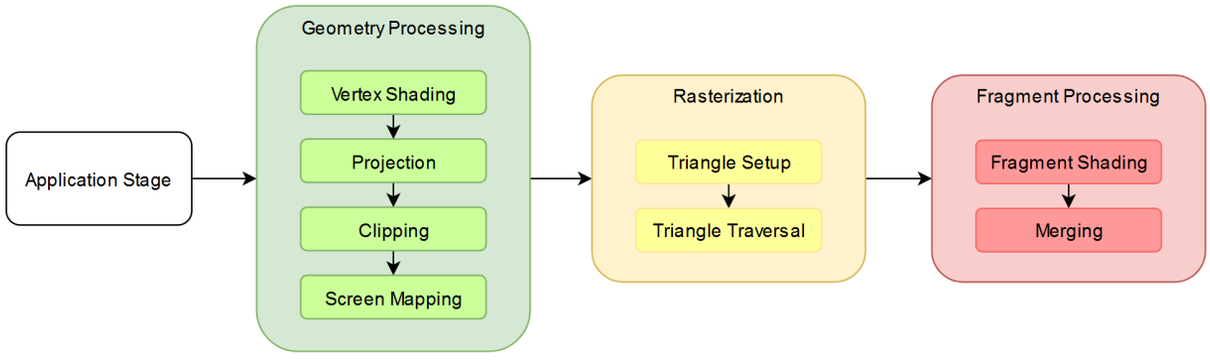

Figure 2.2: The general graphics pipeline, as implemented by most graphics APIs. Adapted from [2, pp. 12ff .].

The Application Stage

The application stage is usually executed on the CPU (but can be partially executed on the GPU through the use of *compute shaders*, as demonstrated in the implementation of the voxel renderer in Chapter 4) and is tasked with the preparation of a set of *primitives* (points, lines or triangles) to be fed into the geometry processing stage [2, pp. 13f.].

The Geometry Processing Stage

The geometry processing stage operates *per-vertex* or *per-triangle* and can be split into the *vertex shading*, *projection*, *clipping* and *screen mapping* sub-stages[2]. The vertex shading sub-stage is usually fully programmable in modern graphics hardware through the use of a *vertex shader*. The vertex shader is a program that is designed to be run per-vertex and is tasked with computing per-vertex attributes and transforming vertex positions from model space into view space [2, pp. 15f.].

The projection stage is usually executed as part of the vertex shader and transforms view space into clip space through the use of some sort of projection transform, usually a perspective projection in game development.

Following the projection stage, primitives are clipped at the edges of the canonical view volume, meaning that vertices outside the view volume are discarded and primitives that intersect the boundaries of the volume are cut at the boundary. At the end of the clipping stage, all geometry that remains is fully contained within the canonical view volume [2, pp. 19f.].

[2]The tesselation and geometry shader stages have been omitted as they are optional and not used in this project.

The final stage of geometry processing is screen mapping, which transforms vertex coordinates to *screen space* coordinates, i.e. coordinates on the user's display.

The Rasterization Stage

The rasterization stage can be split up into the *triangle setup* and *triangle traversal* sub-stages [2, pp. 21f.].

The triangle setup stage is a fixed-function stage that generates per-primitive data such as edge equations that is needed for triangle traversal. Triangle traversal generates a *fragment* for each pixel in screen space that is covered by a primitive. Per-vertex attributes generated by the vertex shader are interpolated across primitives in this step [2, p. 22].

The Fragment Processing Stage

The fragment processing stage is composed of the *fragment shading* and the *merging* sub-stage.

The fragment shading stage is fully programmable and executes a user-defined *fragment shader* that is executed per-fragment and is given the interpolated vertex attributes as inputs and tasked with generating a color for each fragment.

The merging stage combines the fragment colors generated by the fragment shading stage with the data already present in the output framebuffer in some usually user-configurable way. It also performs visibility testing by discarding fragments that are covered by other fragments; the exact conditions under which fragments are discarded is also usually user-configurable [2, pp. 24f .].

Parallel computations

As can be seen from the above description of the graphics pipeline, several stages involve *per*-something computations (e.g. the per-vertex calculations of the vertex shader) which are easily parallelized. As machines designed and optimized to perform these specific parallel calculations, GPUs usually operate under a *single instruction, multiple data* (SIMD) paradigm [2, p. 31]. This places restrictions on the complexity of shader programs, as programs with complex control flow greatly reduce the effectiveness of SIMD data processing.

Common Data

Throughout the graphics pipeline, in particular the programmable vertex shader and fragment shader stages, the programmer may want to access data besides the per-vertex or per-fragment data that is provided to these shader programs by default. This data access is facilitated through the use of *push constants*, *images*, *uniform buffers*, and *storage buffers* in the Vulkan graphics API [5, p. 5]. Push constants are a usually small set of constants that are directly submitted to the GPU with each invocation of the graphics pipeline (a so-called *draw call*). Sampled images and uniform buffers are buffers which can be read by the shader stages but not written to, whereas storage images and storage buffers can, in principle, be written to by the shader stages. Due to the highly parallel nature of graphics pipeline computations, care must taken when writing to shared buffers in order to prevent data races.

Chapter 3

The Engine

3.1 Goals of this Engine

Commercial game engines are immensely complex projects usually totalling millions of lines of source code [1]. It is clearly infeasible for a single developer to implement a fully featured engine in any reasonable amount of time and as such beyond the scope of this project.

Instead, the goal of this project was to implement a partial game engine, called *Orion*, focusing on a few core subsystems, but allow continued development after the end of this thesis and the future integration of further features to make *Orion* a more viable engine for actual game development use. In line with this goal, a central architectural guideline during the development of *Orion* was the desire for modularity where different, parallel, implementations of a particular subsystem were deemed likely in the future.

To facilitate the development within the timeframe of a Bachelor's project, the list of implemented subsystems was cut down to the absolutely essential supporting subsystems, namely game state management, input handling and asset management, and a single core subsystem, namely the graphics renderer. Nevertheless, the implementation of further subsystems is planned in the future and therefore the engine architecture must facilitate the addition of further subsystems.

An additional goal of the engine was the desire to be *multiplatform*, i.e. to run on Windows, Linux and macOS operating systems. Game consoles were excluded from this, as they often require the use of proprietary APIs, e.g. for graphics functionality, and there is no cross-platform graphics API that covers all current generation consoles.

3.2 Engine Architecture

3.2.1 Game State Management and Runtime Object Model

The most basic architectural layer of the *Orion* engine is its game state management system. The game state management system connects all other subsystems, manages the game world, and executes the game loop. In order to decide on an architecture of the game state management system, one has to decide on a runtime object model for all objects or *entities* in the game world. *Game Engine Architecture* [1] lists two different approaches to the runtime object model: *Object-centric* and *property-centric* architectures. In an object-centric architecture, each game entity is a class instance and its behaviors are derived from the class and its superclasses, whereas in a property-centric architecture each entity consists of a set of properties which implicitly define that entity's behavior [1, pp. 1043ff.]. To select an approach, we examine how one might implement each approach in Rust.

[1] The *Unreal Engine* totals 33 027 271 lines of C and C++ code in its 5.1.0 release.

Considering the object-centric approach, one must notice that Rust does not support inheritance in the classical object-oriented programming sense, which makes complex class hierarchies difficult to implement. As an alternative, one might attempt to pursue an object-centric architecture by implementing entities as structs with shared behaviors implemented as traits. Instead of inheritance, blanket trait implementations provide a way to automatically implement shared behavior for all structs that implement some other trait. However, this approach naturally pushes us to reason about shared behaviors which is more akin to a property-centric design and does not really reflect a *true* object-centric architecture and object hierarchy.

A property-centric approach seems to be the more natural approach for an engine implemented in Rust, as it maps naturally to the *composition over inheritance* principle that is part of Rust's design. Listing 1 shows how one might implement a trait-based property-centric architecture in Rust.

```
1  trait Render {
2      fn render(&self, renderer: &mut Renderer);
3  }
4
5  trait Collide {
6      fn collision_step(&mut self, world: &World);
7  }
8
9  struct Cube {
10     // ...
11 }
12
13 impl Render for Cube { /* ... */ }
14
15 impl Collide for Cube { /* ... */ }
```

Listing 1: Example of a trait-based property-centric architecture

This trait-based approach has its drawbacks however, when it comes to querying the game world for specific properties: It is not a simple operation in Rust to ask for all objects which implement a certain trait, e.g. all objects implementing `Render` in order to call their respective `render()` method. It also presents issues if one wants to implement a function that takes any possible implementor of a specific trait as a parameter. To implement such a behavior, the function would either have to be generic over all types that implement that specific trait or accept a trait object, which is a wide pointer consisting of a pointer to the base object and a vtable pointer to enable dynamic function dispatch. Generic functions could result in a significant increase in binary size due to monomorphization, if the trait is implement by a large collection of structs and trait objects may result in decreased performance due to dynamic dispatch, leaving both of these options suboptimal.

Instead of attempting a property-centric architecture using traits, one might attempt to follow the *Entity Component System* (ECS) pattern where entities are composed of *components*, again following the composition over inheritance pattern. However, as opposed to the trait-based approach, instead of having a struct per type of entity and composing traits to implement shared behavior, each component is represented by a struct and shared behavior is implemented through *systems* which act on all entities with specific components. In this approach, an entity is *solely* defined by its components and can simply be represented as a tuple of its constituent components.

In an ECS architecture, the components may be stored in a struct-of-arrays approach, meaning

that the game world is represented by a struct containing a list for each component encountered in the game world (e.g. a list of all positions, a list of all orientations etc.) and each entity is simply a unique ID with pointers into these lists. This approach makes certain queries, e.g. for all entities having a certain component, very fast as one can simply iterate over the list for that particular component. Additionally, an ECS architecture may optimize certain more complex queries by recognizing common *archetypes*, i.e. common combinations of components.

There are several ECS implementations available as Rust crates, the three most popular being `bevy_ecs`, a part of the *Bevy* game engine, `specs` and `legion`. In line with the goal to test the Rust ecosystem's readiness for game engine development, we chose to make use of one of these available libraries, specifically the `legion` ECS as the game state management system.

3.2.2 Asset Management

The asset management subsystem is tasked with loading and preparing various resources, such as level data, meshes, voxel chunks, materials, etc. for use by other engine components. It is also responsible for ensuring that assets stay loaded as long as they are needed.

The first task of the asset management subsystem is accomplished through the use of generics and the `Asset` trait, as shown in listing 2. Any struct implementing this trait is required to provide a function that loads the asset given a file path to the asset and a second function to create said file path given a base directory and asset ID.

```
pub trait Asset: Any + Send + Sync + 'static {
    type CreateContext;

    fn load_from_file(path: &Path, ctx: impl Deref<Target = Self::CreateContext>)
        -> Result<Self, EngineError>
    where
        Self: Sized;

    fn construct_path(id: &SID, base: &Path) -> PathBuf;
}
```

Listing 2: `Asset` trait, taken from `assets/asset_manager.rs`.

Assets are identified by their type and a hashed string ID (represented by the `SID` type). Internally, the asset manager leverages Rust's safe dynamic typing capabilities through the `Any` trait, provided by the standard library and simply stores a hash map mapping hashed string IDs to pointers to assets.

Hashed string IDs consist of a string and a numerical hash of that string. By keeping track of the hash of the string, expensive hashing operations when searching for the ID in a hash map can be avoided while retaining the user-friendliness of string identifiers.

When requesting an asset from the asset manager, the user simply calls the appropriate specialization of the asset manager's `get()` function with the hashed string ID of the desired asset. The asset manager then performs the following tasks:

1. It checks its internal hash map if the asset is already loaded.
2. If the asset is loaded, it returns a reference to the requested asset.
3. If the asset is not loaded, it calls the asset type's `load_from_file()` function, loading the asset into memory. The asset is then inserted into the internal hash map and a reference is returned to the user.

To fulfill the second task of the asset manager, namely ensuring that assets stay loaded for the right amount of time, the asset manager relies on reference-counting smart pointers. Instead of "raw" pointers, whose use is discouraged in Rust, the `Arc<T>`[2] and `Weak<T>` smart pointers provided by the standard library are used for storing references to assets in the internal hash map. When loading an asset, the user can specify whether the asset should be loaded permanently, in which case an `Arc` holding the asset is used, or loaded only temporarily, in which case a `Weak` pointer is used. Use of `Weak` ensures that the refrence held by the asset manager does not prevent the asset from being dropped when all other references to the asset go out of scope, while still allowing the asset manager to avoid second loads of an asset if it is still available.

Finally, the asset manager exposes the `AssetHandle` trait, which allows the implementation of thin wrappers around string IDs (associating the ID with a specific asset type). These thin wrappers are held by the game state management systems instead of references to the assets themselves, to allow for easy serialization of scene data.

3.2.3 Input Handling

The final supporting subsystem implemented in this project is the input handling subsystem. It is tasked with taking the "raw" inputs from interface devices or the operating system and transforming them into game actions. These game actions are pushed to a queue, which is a part of the `GameEventQueues` structure, for handling in the main game loop. The `GameEventQueues` structure is intended as a general mechanism for passing event-like information between subsystems, even across frames.

While the event queue system may be sound for passing information between subsystems, it has shown itself to not be an ideal abstraction for input events, particularly for common actions like translational movement. To illustrate this, we consider the "move forward" action as an example, commonly bound to the "W" key on a keyboard. When the user presses the "W" key, the operating system issues three kinds of events[3]: *key down*, *key repeat* and *key up*. For games, we usually don't care about these events but instead care about the current state of the button, which would be most easily abstracted as a global input map, that can be queried in the game loop. The event queue abstraction works well for mouse movement events, as mouse movements are given as a sequence of small steps, which all need to be processed in-order to reconstruct the correct movement.

The input handling system clearly needs more work and should probably be restructured to differentiate between "state-like" actions, like a button being pressed, and "event-like" actions, like mouse movements or the completion of a specific button sequence, and handle each type of input in a way that is fitting for that input.

3.2.4 Renderer

One of the most complicated and important core systems of a modern game engine is often its (graphics) *renderer*, i.e. that subsystem which is tasked with producing images, in real-time, to be displayed to the user. The renderer is often constructed on top of an abstraction layer, allowing the use of different graphics APIs (e .g. DirectX, Vulkan, OpenGL, Metal) backing the rendering process. As the Vulkan API is sufficiently cross-platform for PC systems and consoles were not a focus of this project, no such abstraction layer is implemented as part of *Orion*. However, as extensibility is one of the major design considerations in *Orion*'s architecture,

[2] *Arc* stands for atomically reference counted.

[3] a similar thing occurs when operating on a raw interface basis and interpreting signals from the hardware directly

care was taken to allow the integration of a different renderer, potentially featuring multiple-API support, in the future.

To achieve the desired extensibility, renderer functionality is split into traits providing backend-agnostic interfaces and any application requiring a specific set of functionality is required to pick a renderer implementation that implements the necessary traits. The complete set of traits currently available and implemented by the "reference" `VulkanRenderer` is shown in listing 3.

```rust
pub trait GraphicsRenderer {
    fn begin_frame(&mut self, cam: &Camera,
                              dir_lights: &[(Rotation, Color, DirectionalLight)],
                              point_lights: &[(Position, Color, PointLight)])
                    -> Result<(), EngineError>;

    fn finish_frame(&mut self) -> Result<(), EngineError>;
}

pub trait EguiRenderer {
    fn render_gui(&mut self, gui: &mut Gui) -> Result<(), EngineError>;
}

pub trait MeshRenderer {
    fn render_mesh_solid(&mut self, transforms: &[(&Position, &Rotation, &Scale)],
                                    mesh: Arc<Mesh>,
                                    mat: Arc<SolidMaterial>)
                        -> Result<(), EngineError>;

    fn render_mesh_textured(&mut self, transforms: &[(&Position, &Rotation, &Scale)],
                                       mesh: Arc<Mesh>)
                            -> Result<(), EngineError>;
}

pub trait SkyRenderer {
    fn render_sky(&mut self, rotation: &Rotation,
                             mat: &SkyMaterial,
                             cam: &Camera)
                  -> Result<(), EngineError>;
}

pub trait VoxelRenderer {
    fn meshify_voxel_chunk(&mut self, chunk: Arc<VoxelChunk>)
                           -> Result<(), EngineError>;

    fn render_voxel_chunk(&mut self, position: &Position,
                                     rotation: &Rotation,
                                     scale: &Scale,
                                     chunk: Arc<VoxelChunk>)
                          -> Result<(), EngineError>;
}
```

Listing 3: Renderer traits, taken from `renderer.rs`.

The `GraphicsRenderer` trait is the basic trait that should be implemented by all renderers and provides functions to mark the beginning and end of a frame, which must be called before and after a sequence of `render_*` functions respectively. This allows the renderer implementation to

internally prepare state that can be modified by the `render_*` functions and finally submit the desired list of rendering commands to the GPU with the `finish_frame` function.

3.3 Mesh Renderer Implementation

As mentioned before, the default mesh renderer of *Orion* is implemented using the Vulkan graphics API. There are several options for integrating Vulkan with Rust, most notably perhaps the `ash` and `vulkano` crates. Ash provides auto-generated bindings to the Vulkan API and exposes Vulkan in it's full complexity and feature set. As Ash directly exposes bindings to a C-like API and does not validate API usage, all functions provided by Ash are `unsafe`. Vulkano, on the other hand, provides hand-written safe wrappers around Ash's unsafe bindings, extending Rust's idea of safe code to Vulkan API calls, ensuring that all API calls are standard-compliant. In order to accomplish this safe wrapping, Vulkano introduces some further abstractions that are not present in the Vulkan API specification (such as the idea of a GPU future, representing the result of some action on the GPU that may or may not be ready yet). However, in most cases, Vulkano stays relatively close the the abstractions presented in the Vulkan specification. For *Orion*'s renderer implementation, Vulkano was chosen for its safe wrappers around Vulkan.

3.3.1 Some Vulkan Background

In order to understand the mesh renderer and voxel renderer implementations, we must examine a set of Vulkan objects that are central to Vulkan's operation. These objects are the *Device*, *Queue*, *Command Buffer*, *Pipeline* and *Descriptor Set*.

Devices and Queues

In Vulkan, "Device objects represent logical connections to physical devices. Each device exposes a number of *queue families* each having one or more *queues*" [4, Sec. 5.2].

The device and queue handles are used to submit operations to be executed on the GPU. In order to define their roles more accurately, we need to distinguish between two different kinds of operations: *Highly parallelized operations*, usually called *commands* in the context of Vulkan, and *non-parallelized operations* such as the allocation of buffers or configuration of graphics or compute pipelines.

Non-parallelized operations are submitted to the GPU via the device object, while commands are submitted to a queue. The type of operation that can be submitted to a queue depends on that queue's *family*. Device and queue objects are created as a single step and are closely tied to each other. Multiple queues may be created on a single device to allow further explicit concurrency of GPU command execution, as work on separate queues may be performed concurrently, while work on a single queue obeys at least some weak ordering guarantees [4, Sec. 7.2].

Command Buffers

Submitting work to be executed by a queue is an expensive operation [4, Sec. 6.5], as it incurs overheads on the allocation of command storage on the GPU, transfer of commands to the GPU and the ramp-up and ramp-down of the parallel processing pipelines themselves. To minimize this overhead, commands are grouped into command buffers which are then submitted to the GPU as a batch.

In Vulkan, the command buffer itself is an opaque object and gets filled by *recording* commands. Recording is performed by calling specific functions that modify the state of the command buffer, each representing a specific command or set of commands to be executed.

Pipelines

As discussed in section 2.3.2, the graphics pipeline on the GPU consists of several fixed function, configurable or fully programmable stages. Compute operations are executed in similar stages, although the number of stages is greatly reduced in a compute pipeline compared to a graphics pipeline. In order to perform work on the GPU, the compute or graphics pipeline needs to be configured to allow that work to be performed.

In Vulkan, *Pipeline* objects represent a particular configuration of the compute or graphics pipeline. They encapsulate the configuration of the configurable stages and the shader code for any fully programmable stages in addition to the structure of external inputs and outputs of pipeline [4, Sec. 10]. This I/O configuration of the pipeline is referred to as it's *layout* [4, Sec. 14.2.2]. An overview of the stages of the Vulkan pipeline is shown in figure 3.1.

When recording command buffers, one of the first operations consists of *binding* a pipeline object, providing the context in which subsequent commands are executed.

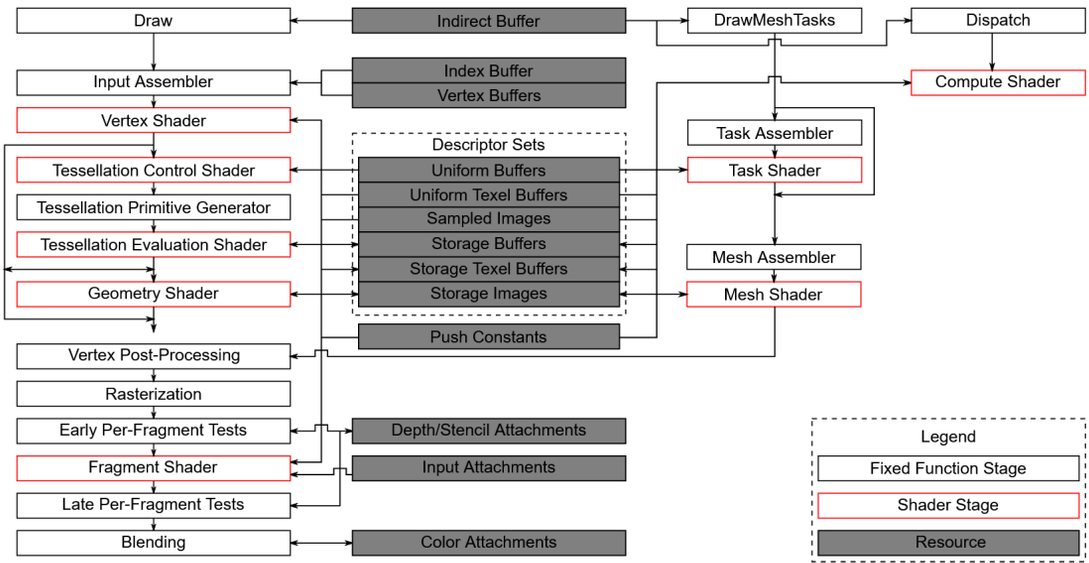

Figure 3.1: An overview of fixed function and configurable stages in Vulkan pipelines, including resources used by them. Taken from [4, Sec. 10], © The Khronos Group Inc.

Descriptor Sets

Quoting from the specification, "A *descriptor* is an opaque data structure representing a shader resource such as a buffer, buffer view, image view, sampler, or combined image sampler. Descriptors are organized into *descriptor sets* [. . .]" [4, Sec. 14]. Like pipelines, descriptor sets have a layout and are bound during command buffer recording. When binding descriptor sets, their layout must be compatible with the layout of the currently bound pipeline and the descriptor set layout/pipeline layout can be viewed as the formal specification of the interface between a shader program and its resources. When writing shader code, resources are annotated with a descriptor set number and descriptor binding number (within that specific set) to clearly indicate where the GPU can access the needed resource.

3.3.2 Application Stage

When rendering meshes, the application stage must perform several tasks: First, the application stage must prepare the GPU for rendering by initializing a command buffer to record into, then uploading all necessary resources to the GPU (or ensuring resources are already uploaded),

binding the required graphics pipeline and descriptor sets and finally issuing the rendering command, usually referred to as *draw*.

When discussing the resources needed by the shaders, we must differentiate between *vertex input data* and *uniform data*. Vertex input data is data that is different for each vertex of a mesh (or mesh instance), while uniform data remains constant for all shader invocations. Vertex data is passed to the shader through vertex buffers, while uniform data is accessed through descriptor sets.

Vertex and Index Buffers

A data structure must be chosen to store the mesh data, such that it is available to the GPU for rendering. In rasterization rendering it is common to store this mesh data in the form of vertex buffers, where attributes are associated with each vertex of the mesh and only a list of vertices and their attributes is stored. This list of vertices is referred to as the vertex buffer. [4, Sec. 22.1]

In order to reconstruct triangles from the vertex buffer, the GPU must interpret the data contained in the vertex buffer according to a pre-defined ruleset. This ruleset is known referred to as the *primitive topology* in Vulkan [4, Sec. 21.1].

Vulkan supports five primitive topologies for triangle meshes, three of which we will examine here: *Triangle lists*, *triangle strips* and *triangle fans*.

Triangle lists are perhaps the simplest way of interpreting vertex data, where "each consecutive set of three vertices defines a single triangle primitive" [4, Sec. 21.1.5]. The i-th primitive (i.e. triangle) is defined through the equation

$$p_i = \{v_{3i}, v_{3i+1}, v_{3i+2}\}.$$

Triangle lists have the obvious disadvantage that, when rendering connected meshes, vertices must be duplicated as each triangle is represented by its own group of three vertices and there is no way of sharing vertices between triangles. *Indexed rendering* is one method of reducing the impact of this restriction (see below) but other topologies can also be chosen to re-use vertex data in connected meshes.

For triangle strips, primitives are generated through the following construction: We begin with a single triangle, defined through three consecutive vertices in the vertex buffer. Each successive vertex in the vertex buffer generates a new triangle with the two vertices that precede it in the buffer. Equivalently, "one triangle primitive is defined by each vertex and the two vertices that follow it" [4, Sec. 21.1.6]. This can be expressed through the equation

$$p_i = \{v_i, v_{i+(1+i\%2)}, v_{i+(2-i\%2)}\}$$

where % is the modulo operator. Triangle strips can also suffer from vertex repetition, like triangle lists.

Finally, when using triangle fans, each triangle is defined by a vertex, its preceding vertex and a shared "central" vertex [4, Sec. 21.1.7]. This is expressed in the equation

$$p_i = \{v_{i+1}, v_{i+2}, v_0\}.$$

Indexed rendering is a technique that reduces the impact of vertex repetition, which makes it particularly useful for rendering with triangle list or triangle strip topologies. In indexed rendering, instead of defining the topology through the vertex buffer directly, a secondary buffer called the *index buffer* is used to define the triangle topology. All vertices and their attributes

are still stored in the vertex buffer but each vertex need only be stored once. The index buffer contains indices into the vertex buffer and defines the mesh topology according to the chosen primitive topology ruleset.

In *Orion* only indexed triangle list rendering is used, as this is the most general format and default representation for many 3D formats, including the *Wavefront OBJ* format used for mesh assets in *Orion*.

Regardless of topology, the vertex shader is executed once for each vertex contained in the vertex buffer [4, Sec. 9.8].

To enable further reuse of mesh data, Vulkan supports *instanced rendering*, where multiple *instances* of each mesh can be rendered, each having a potentially different set of attributes. Instance attributes are also passed to the vertex shader through vertex buffers, although these particular attributes are marked with a special *input rate* flag [4, Sec. 22.2] to inform Vulkan that these vertex buffers are to be accessed based on the instance index instead of the vertex index.

Descriptor Sets

Data that is shared between vertex shader invocations, i.e. *uniform data*, is accessed through descriptor sets. Vulkan specifies that hardware must support a minimum of four descriptor sets being bound to a pipeline at once [4, Sec. 45.1]. This limit forces us to reduce the number of descriptor sets used in our application as much as possible. Additionally, binding descriptor sets is an expensive operation, so we would like to minimize the number of re-binding operations we need to perform.

To reduce the overall number of simultaneously bound descriptor sets and minimize binding operations, *Orion* groups resource descriptors into sets by *binding frequency*. Specifically, *Orion* binds at most two descriptor sets at once, binding set 0 per-frame and set 1 per-mesh.

Set 0 thus contains descriptors for all resources that are constant for each frame being rendered, e.g. the scene lights or camera information. Set 1 contains material information.

Buffer Preparation

Before submitting *draw* commands to the GPU, the application stage must ensure that all necessary resources are available and fill the used descriptor sets appropriately.

At the start of each frame, camera and light data is placed into GPU-accessible buffers in CPU memory and descriptor set 0 is filled with descriptors pointing to these buffers.

For each `render_mesh_solid` call (as described in Listing 3) the application ensures that the mesh data is loaded (mesh data is uploaded to the GPU asynchronously, so the mesh data may not be ready yet) and skips rendering of that particular mesh if it's data is not available yet. Additionally, buffers are allocated for material data and the set of transforms supplied to the render call.

Draw Call

Finally, the application binds the necessary pipeline, vertex buffers, index buffer and descriptor sets, if they aren't bound already, and issues a `vkDrawIndexed` command.

3.3.3 Vertex Shader

The vertex shader used in the mesh rendering pipeline[4] computes the clip space and world space positions of each vertex as well as the world space normal vectors at each vertex. The clip space positions are passed to the GPU for perspective division, clipping and viewport transformation through a special pre-defined output variable (`gl_Position` in GLSL). The world space positions and normals are passed to the fragment shader as regular vertex shader outputs, which are interpolated across each triangle by the rasterizer.

3.3.4 Fragment Shader

The fragment shader is tasked with computing the color of each pixel, given the pixel's position in world space, its normal vector, the position and properties of each light and the material properties of the pixel in question.

Orion's fragment shader implementation is based heavily on the fragment shader of Google's *Filament Engine*, described in [27], which is in turn based on the physically-based shading model used at Disney and described in [6].

In physically-based shading, the interaction of light with a surface is described through the *Bidirectional Scattering Distribution Function* (BSDF), which is a probabalistic measure of how rays hitting the surface are scattered. The BSDF can be split into two components: The *Bidirectional Reflectance Distribution Function* (BRDF), describing the behavior of reflected rays, and the *Bidirection Transmittance Distribution Function* (BTDF), describing the behavior of rays transmitted into the surface. Our shading model does not deal with rays transmitted into objects and thus our BSDFs are only composed of the BRDF component.

Our BRDF model can be further broken down into two terms: One term describing *diffuse* interactions and one term describing *specular* interactions.

Our diffuse term models ideal Lambertian reflectance, i.e. the diffuse reflectance follows Lambert's cosine law stating that the reflected luminous intensity is proportional to the cosine of the angle between incident ray and surface normal. This is an approximation, as real materials do not exhibit ideal Lambertian diffuse reflectance.

Our specular term follows the Cook-Torrance specular reflection model, described in [7] and given by

$$f_r(v, l) = \frac{D(h, \alpha)G(v, l, \alpha)F(v, h, f_0, f_{90})}{4(n \cdot v)(n \cdot l)}$$

where v is a unit vetor pointing towards the camera, l is a unit vector pointing towards the light, n is the surface normal vector, h is the half-angle vector between the incident light vector l and the view vector v, α is a roughness parameter, f_0 is the Fresnel reflectance at normal incidence and f_{90} is the Fresnel reflectance at a grazing angle. D is the microfacet normal distribution function, G is the geometric shadowing function and F is the Fresnel reflectance function.

This reflection model is based on microfacet theory, which models rough surfaces through a collection of microscopic flat surfaces called *microfacets*, each with a *microfacet normal* m. D describes the statistical distribution of these microfacet normals. In the microfacet model, not all microfacets take part in a light interaction, as some may be shadowed by other features of the microscopic surface geometry. This shadowing is described through the geometric shadowing function G. The product $D \cdot G$ describes the distribution of visible microfacet normals [2, pp. 331ff.]. The Fresnel reflectance function F describes the intensity of the specular reflection, which is based on the optical properties of the material in question and the viewing angle.

[4] defined in `shaders/glsl/SimpleMesh/main.vert`

Several models exist for the microfacet distribution D, including the Beckmann distribution, Phong distribution and GGX distribution. *Orion* uses the GGX distribution, described in [8], which is an empirical model.

As the geometric shadowing function F, the Smith model [9] is used, which sets

$$G(v, l, \alpha) = G_1(v, \alpha)G_1(l, \alpha)$$

where G_1 depends on the microfacet model used. [8] gives a G_1 that matches their GGX microfacet distribution.

For the Fresnel term F an approximation by Schlick [10] is used, that is fast to compute.

The exact implementation of this model is given in `shaders/glsl/SimpleMesh/main.frag` and `bsdf.glsl`.

The given model describes *local illumination*, i.e. the illumination that is caused by light following the path light source – surface – camera. *Orion* does not implement *global illumination* effects caused by lights bounding multiple times, nor does it implement the casting of shadows.

3.4 Benchmarks

All benchmarks were performed on a machine equipped with an AMD Ryzen 9 5900X CPU, an Nvidia RTX 3080 GPU and 32 GB of RAM operating at 3200 MT/s. Apart from the Debug vs. Release mode comparison, all benchmarks were performed using the Release build configuration. All benchmarks were rendered at a resolution of 1920x1080 pixels.

3.4.1 Debug vs. Release Mode Builds

In order to test the performance differences between debug and release configuration builds, we render the "Sponza" mesh in both build configurations. The Sponza mesh is based on the atrium of the Sponza Palace in Dubrovnik, Croatia and is a widely used reference mesh in computer graphics; it consists of 262'267 triangles. The results of this benchmark are shown in figure 3.2, the rendered scene is shown in figure 3.3.

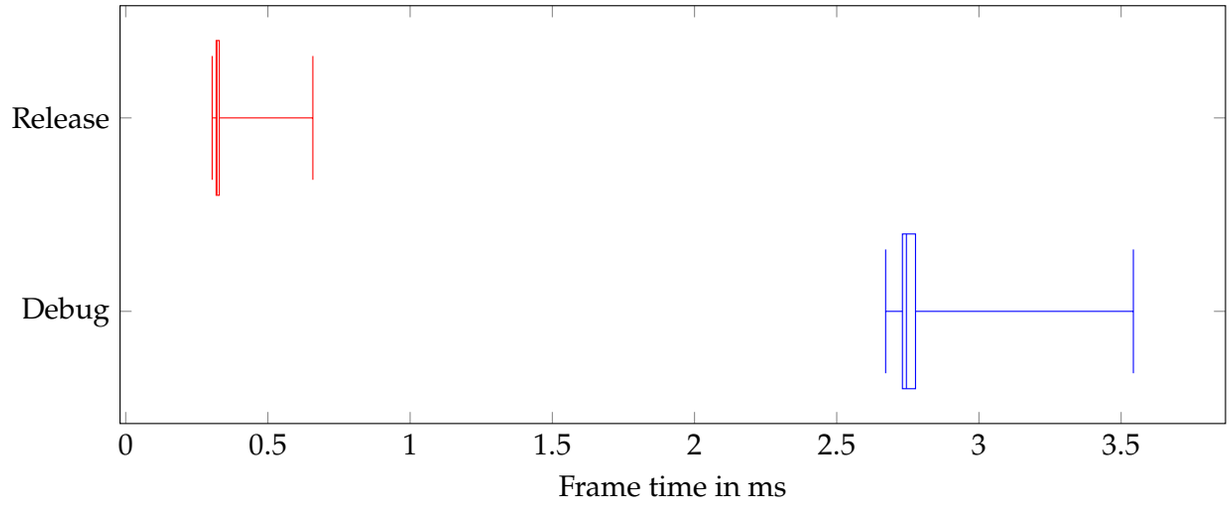

Figure 3.2: Comparison of frame times in Debug and Release build profile, whiskers show maximum and minimum. The median frame time in Debug mode is 2.78 ms, the median in release mode is 0.33 ms, indicating that frame times are approximately 8.4x higher in Debug mode.

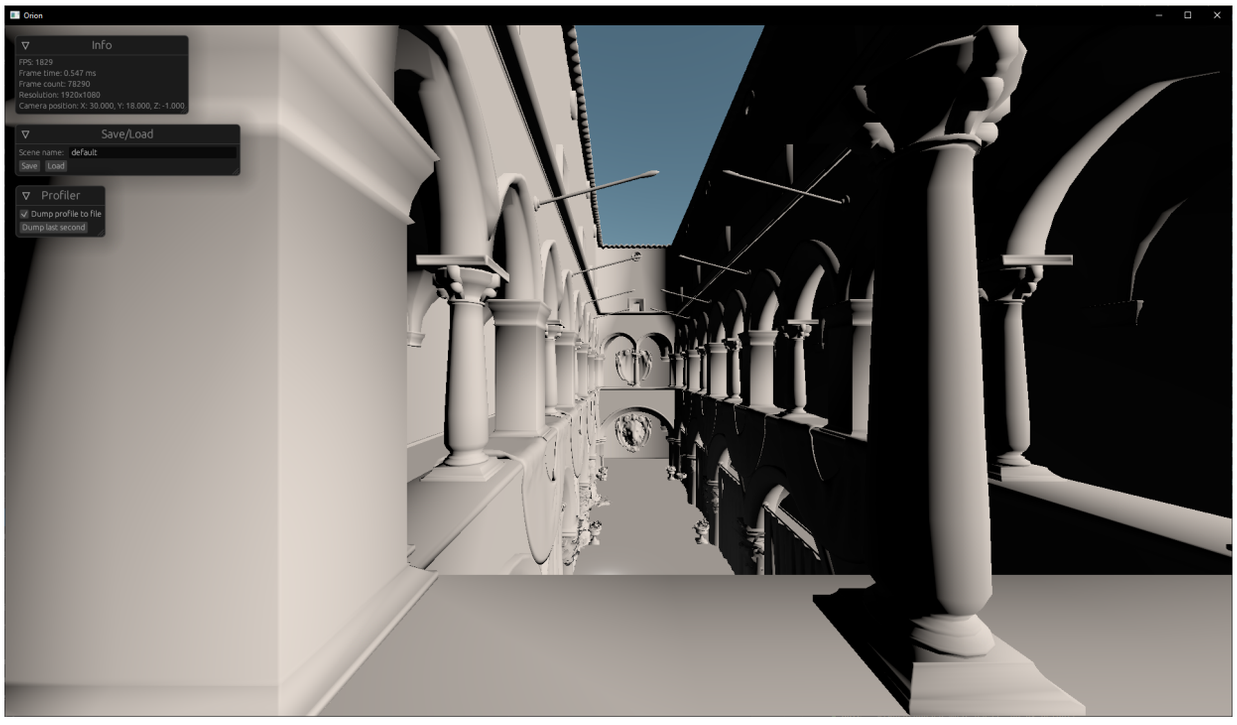

Figure 3.3: The Sponza model rendered with a uniform grey material.

In order to gain more insight into the cause of these differences, we investigate the time composition of an exemplary frame, this is shown in figure 3.4.

3.4.2 Rendering Multiple Meshes

As a next benchmark, we investigate how the engine performance scales when rendering multiple meshes. The benchmark scene is set up to render n copies of the "Suzanne" monkey head[5] placed at random positions and orientations within the view frustrum. Note that this example does not use instanced rendering but renders each mesh using a separate draw call to simulate rendering different meshes. Figure 3.5 shows the results of this benchmark. Figure 3.6 shows the scene with 200 monkey heads.

3.4.3 Large Meshes

As a final benchmark of the mesh renderer we investigate how performance scales with the number of triangles in a mesh. For this benchmark, we use the result of the voxel based mesh generation described in section 4.2 and render various versions of a voxel torus, excluding the time to generate the triangle mesh from the voxel source data. The results of this benchmark are shown in figure 3.7.

[5] "Suzanne" is a default mesh in the Blender graphics software.

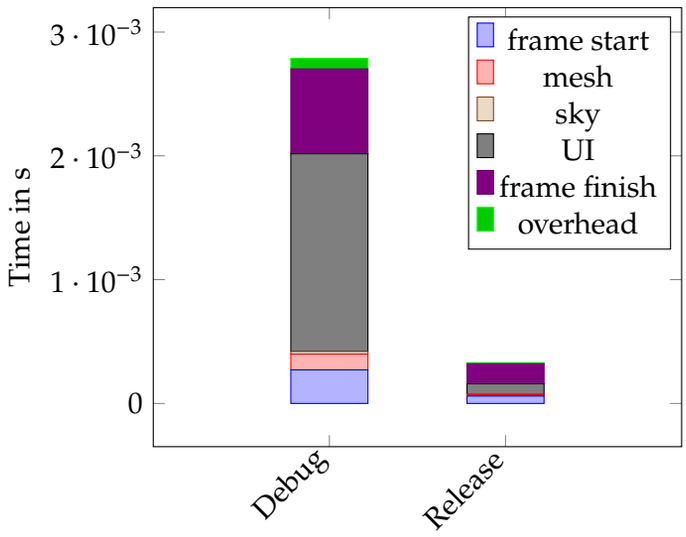

Figure 3.4: Frame time composition when rendering the Sponza scene in the Debug and Release build profile. UI rendering takes up a considerable amount of time when rendering using the Debug profile. The shown frame exhibits a total frame time close to the median.

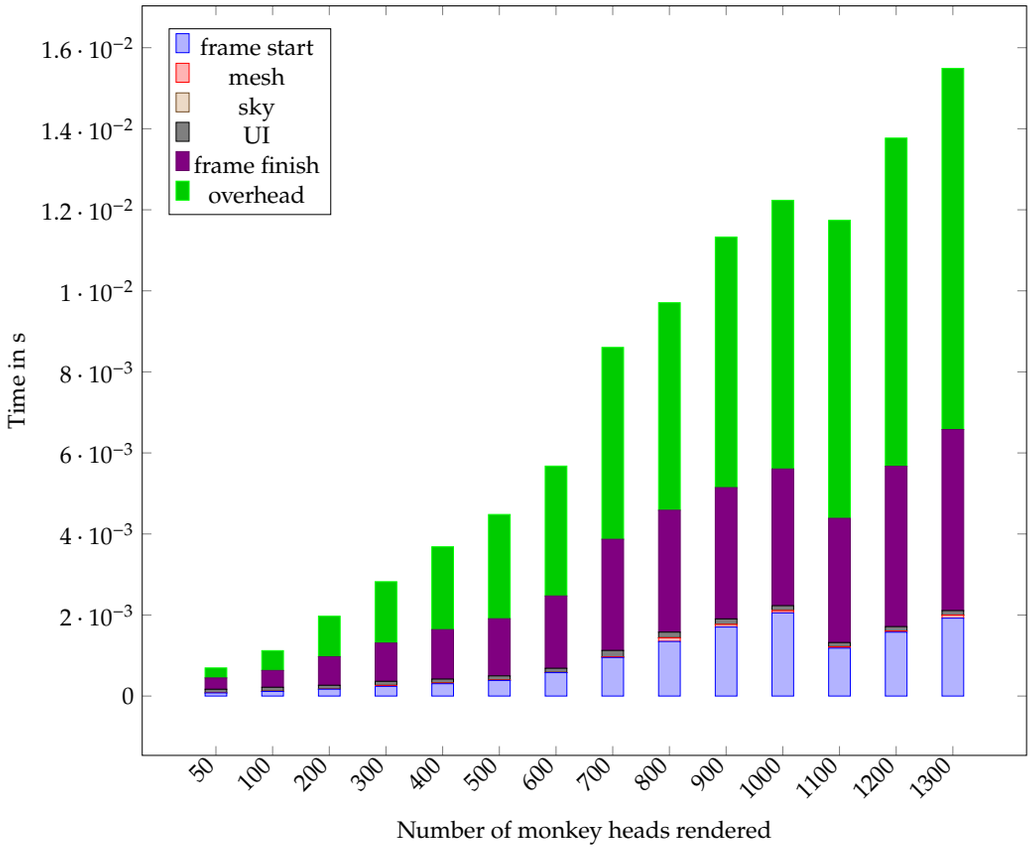

Figure 3.5: Frame time composition of a close-to-median frame when rendering multiple copies of the "Suzanne" monkey head. We can easily see that frame start, frame finish and unprofiled overhead are increase most dramatically. This is most likely due to a) the need to wait longer for the GPU when starting a frame, b) the increase in command buffer size when submitting data to the GPU when finishing the frame and c) increased overhead in the game state management system.

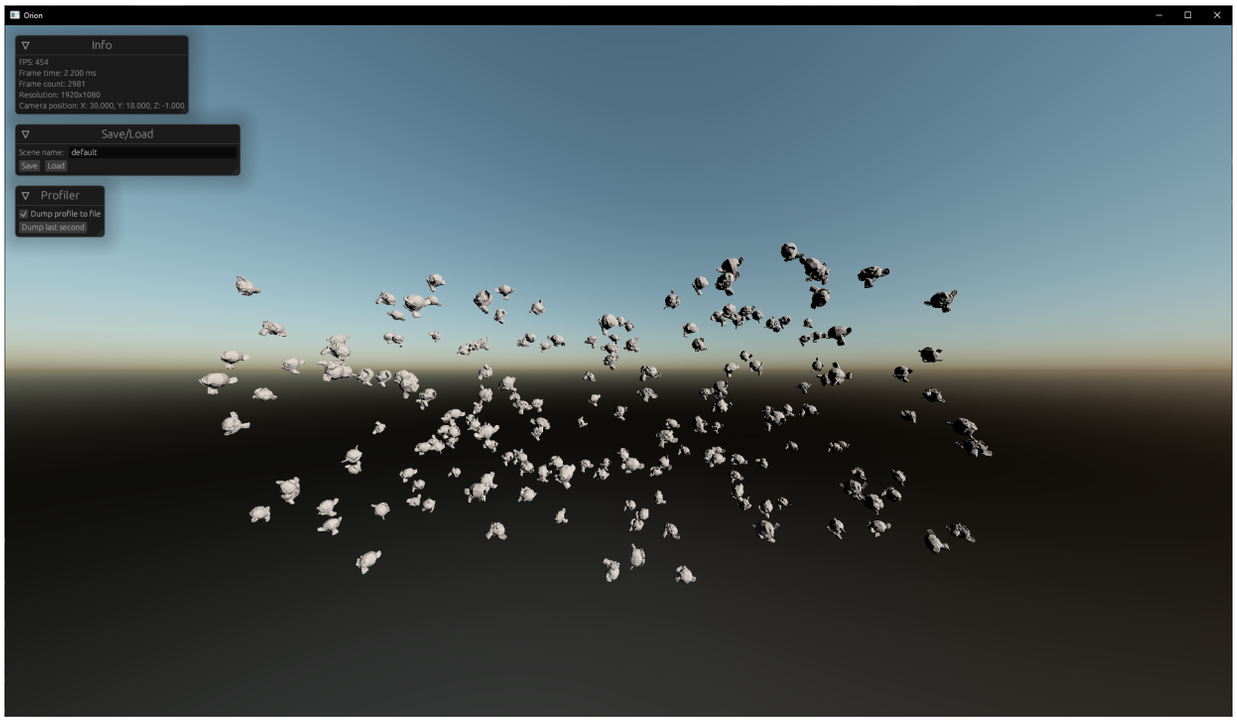

Figure 3.6: Rendering 200 randomly positioned and oriented monkey heads.

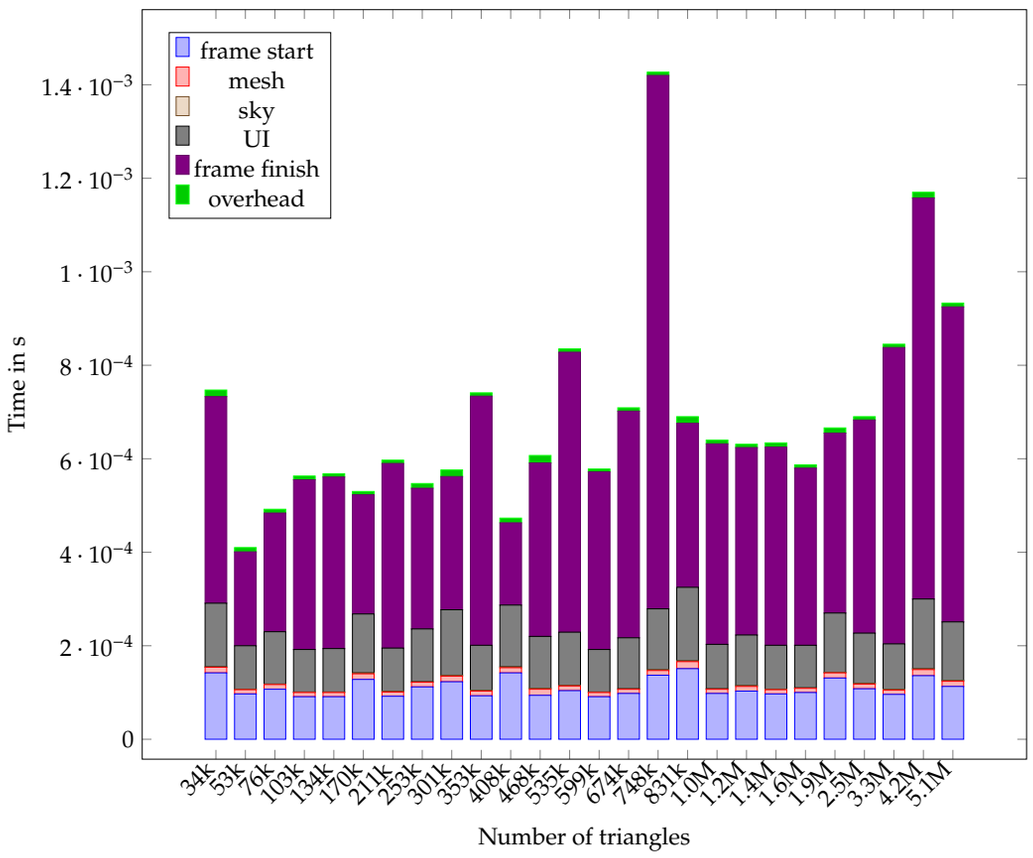

Figure 3.7: Frame time composition of a close-to-median frame when rendering voxel-based torus meshes. This benchmark shows at most a slight upwards trend in frame time when varying the number of vertices in the scene over multiple orders of magnitude. This clearly demonstrates that our application is not GPU-constrained as the increase in triangle count purely affects the GPU load and has minimal impact on the CPU load.

Chapter 4

The Voxel Renderer

In order to evaluate whether the goal of implementing an extensible engine architecture is met, the engine is extended with the capability to render voxel-based geometry. This voxel renderer is designed around the observations that a) modern GPUs can perform parallelized computations very effectively using compute shaders and b) the memory available on GPUs has grown significantly in recent years, enabling techniques that focus less on memory efficiency.

4.1 Traditional Voxel Rendering Techniques

When rendering voxel-based geometry, one can either render the geometry directly through ray tracing techniques or construct a triangle mesh from the voxel data that can then be rendered using any mesh rendering system, such as the renderer described in section 3.3. In this case, we focus only on the second option, as our goal is to render voxels using modern GPUs, which are fundamentally designed for rendering triangle meshes.

A commonly used technique for extracting triangles from voxel-based geometry is the Marching Cubes algorithm described in [11]. Marching Cubes can be parallelized using techniques similar to the one implemented in this renderer, however, this algorithm yields an approximation of a smooth surface arond the voxel geometry, which may be undesirable for artistic purposes.

A different algorithm is described in [28], which attempts to create a surface that looks identical to the voxel model while minimizing the total vertex count. This algorithm suffers from the fact that it cannot easily be parallelized.

4.2 Parallel Triangle Mesh Generation

The voxel renderer implemented in *Orion* is fundamentally similar to the culling mesh generation technique shown in [28] but generates triangles in parallel on the GPU using a compute shader instead of generating the triangle mesh on the CPU. This produces a suboptimal mesh, as surfaces may contain superfluous vertices that may not be necessary but it avoids a mesh copy operation from CPU to GPU memory, as the vertex data is directly generated on the GPU as well as enabling parallelization of the surface generation.

Fundamentally, the algorithm performs the following steps for each of the six faces of each voxel:

1. Check if the neighboring voxel on that side is opaque. If it is, do nothing for this face, otherwise continue.
2. Compute the four vertices of that face and their normals.
3. Increment an atomic counter that is shared between all invocations of the shader.

4. Insert the four vertices and normals into a buffer at an index based on the value of the atomic counter.

This method only generates vertex data for faces that are potentially visible and produces a buffer that may be used as the vertex buffer in traditional triangle mesh rendering.

4.2.1 Implementation Details

The algorithm described above is implemented in `shaders/glsl/Voxels/meshify.comp`. Voxels are stored as a 3D texture, giving each voxel a unique color. As the output buffer of the meshing operation needs to be pre-allocated, we run the algorithm twice, simply counting the number of generated vertices in the first iteration, then allocating the buffer and finally running the algorithm again to actually generate the vertex data.

4.3 Improvements and Adaptations

There are several ways in which this simple voxel rendering algorithm could be extended to different use cases or to allow more efficient calculations.

First, this algorithm could be augmented to use marching cubes for surface generation instead of generating flat *Minecraft*-esque surfaces. Second, the 3D texture could contain texture information instead of colors, to enable the rendering of textured voxels instead of voxels with flat colors. Third, mipmapping could be used to create lower-resolution versions of the voxel texture, allowing the generation of lower-resolution meshes, e.g. for rendering objects that are far from the viewer.

Finally, mesh shaders [4, Sec. 25], could be used to directly generate the vertex data in the graphics pipeline, skipping the compute and vertex shader stages and potentially saving buffer allocations and the double execution of the compute shader. However, mesh shaders would not allow caching of the generated mesh, so all meshes would need to be generated each frame.

4.4 Benchmarks

To test whether the triangle mesh generation approach above is sufficiently fast for use in a game, we tested the system for voxel meshes of several sizes. The results of this test are shown in Figure 4.1. For these experiments a thin-walled torus of various sizes was used as the test object. The number of rendered voxels compared to the total grid size is shown in figure 4.2. The shown relationship approaches a linear relationship, as would be expected: The number of rendered voxels scales with the surface area of the torus, which is proportional to the product of the major radius and minor radius of the torus, while the total size of the grid is proportional to the product of the major radius squared and the minor radius. A screenshot of one of the rendered tori is shown in figure 4.3.

4.5 Architecture Evaluation

Integrating the voxel renderer into the rest of the engine did not require major changes to the codebase. The ECS and asset management system were extended with the `VoxelChunk` component as an alternative to the `Mesh` component. A new `VoxelRenderer` trait was introduced and implemented by the `VulkanRenderer` renderer system. Notably, implementing the `VoxelRenderer` trait did not require major changes to the architecture of `VulkanRenderer`. As

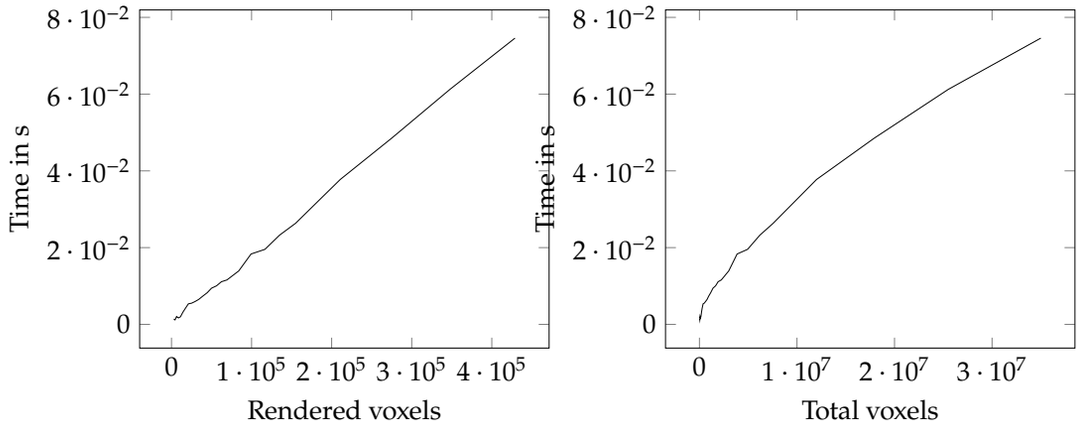

Figure 4.1: Time to convert a voxel model to a triangle mesh. As we can see, the time to convert a voxel grid to a triangle mesh seems to scale linearly with the number of "surface" voxels, i.e. voxels that are actually rendered/included in the triangle mesh. The fact that the time scales sublinearly in the total size of the grid seems to suggest that the limiting factor is the atomic add operation that only gets carried out if a voxel is actually rendered.

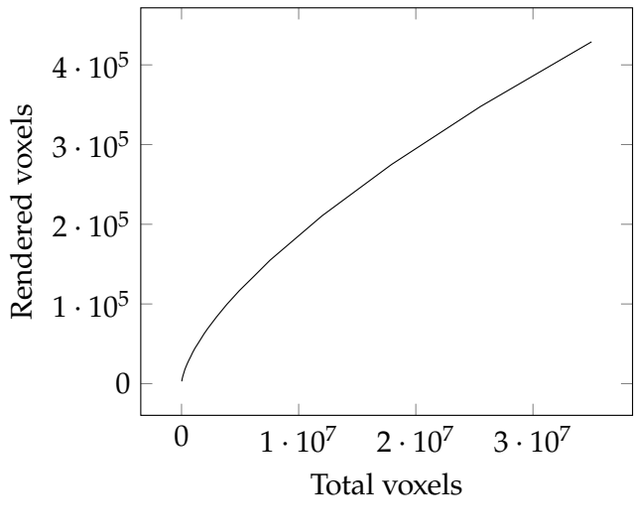

Figure 4.2: Number of rendered voxels vs. grid size/total number of voxels for the example object of a thin-walled torus.

the final piece of integrating the voxel renderer, the ECS was extended with a system acting on all entities with a `VoxelChunkHandle` component.

As a result of this, the *Orion* engine is now capable of rendering voxel-based geometry, while still retaining all previous functionality (e.g. rendering mesh based geometry). An *Orion* scene may contain both voxel-based and mesh-based geometry, enabling maximal flexibility in the use of both of these technologies for the game developer.

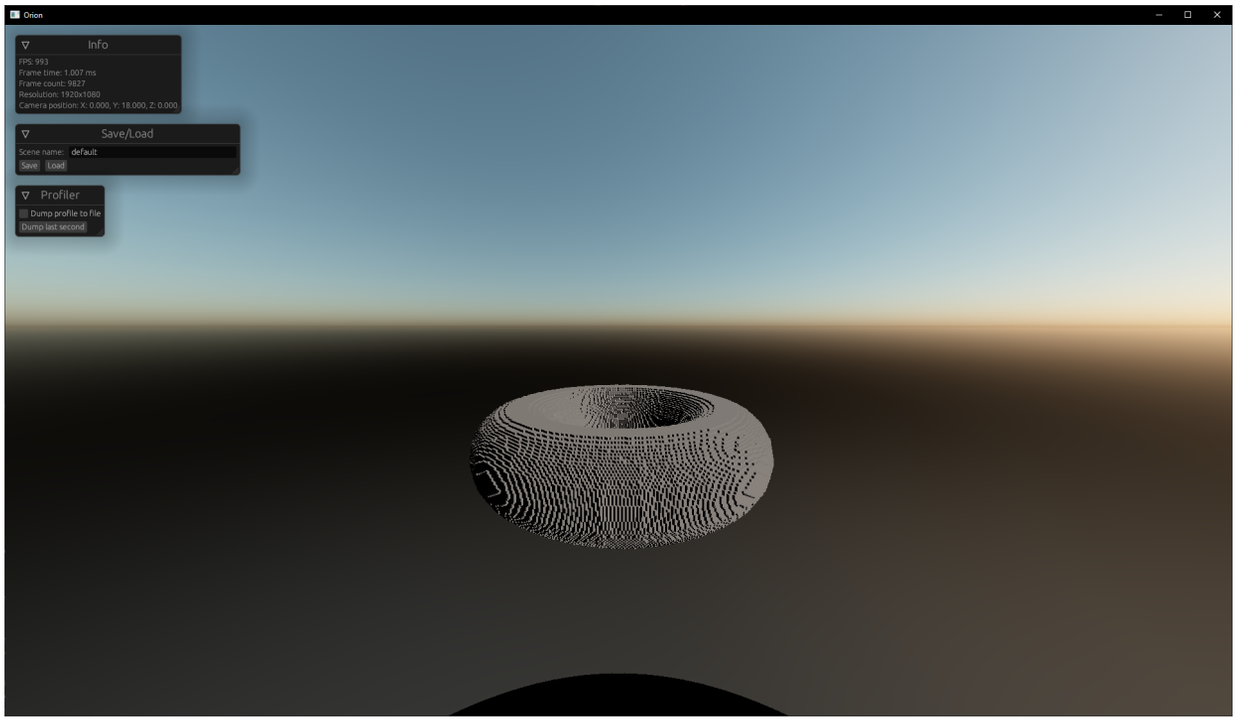

Figure 4.3: A voxel-based torus measuring $170 \times 48 \times 170$ voxels in size.

Chapter 5

Conclusion

5.1 Viability of Rust for Game Engines

This project shows that the Rust programming language is a viable candidate for the development of game engines. The benchmarks in section 3.4 clearly demonstrate that rendering relatively complex scenes at more than 60 FPS is feasible, even without significant hand optimization in the engine codebase. However, the overhead seen in section 3.4.2 shows that there still is potential for further optimization potential in the CPU-side application code.

The use of a "safe" wrapper around the Vulkan API in the form of the `vulkano` library did not seem to impact performance in a significant way, showing that using Vulkan without a large number of `unsafe` code blocks is possible.

The libraries for foundational tasks, such as windowing, mathematics and graphics are available in a relatively polished state, as can be seen by the `winit`, `ultraviolet` and `ash` and `vulkano` libraries used by *Orion*. Libraries for higher-level systems such as an Entity Component System are available but do not seem as polished or optimized as the aforementioned foundational libraries.

Architecturally, Rust's idioms map particularly well to the *composition over inheritance* style of programming, that is employed in an ECS based game engine architecture. Rust does not map well to inheritance-based structures, making the implementation of a hierarchichal entity system in a game engine difficult.

In conclusion, one can assert that Rust is definitely capable of being used as a language for game engine development. Projects such as the *Bevy* game engine, which present more mature engines than *Orion*, lead to the same conclusion, although we are not aware of any large Rust-based game engine projects that use Vulkan as a backend. As a caveat, it is unlikely that we will see game engine development based on Rust in the games industry at-large, as fully from-scratch engine projects are very rare, especially at big game developement studios. This is due to the fact the development of a new engine would presents a significant financial burden, regardless of the programming language used. However, we might see hybrid development approaches similar to the approach taken by the Linux kernel, in that rewrites of certain old or the development of new modules could take place using Rust.

5.2 Outlook

After the conclusion of this project, the source code of *Orion* will be made publicly accessible under an open-source license. We hope to continue development of *Orion*, bringing in more contributors, and moving towards a more feature-complete engine.

In the immediate future, we plan to extend Orion with a physics engine and move the renderer

towards a deferred rendering-based system and adding some global illumination features including shadowing. In conjunction with this major renderer change, we might move from `vulkano` to `ash` as the backing graphics library in order to gain access to bleeding-edge Vulkan features such as hardware ray tracing or mesh shaders, which are not available using `vulkano` yet.

Appendix A

Bibliography

General references

[1] J. Gregory, *Game Engine Architecture*, 3rd, M. W. Jeff Lander, Ed. CRC Press, Aug. 2019, ISBN: 978-1138035454. [Online]. Available: https://www.gameenginebook.com/.

[2] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire, *Real-Time Rendering*, 4th. Boca Raton, FL, USA: A K Peters/CRC Press, 2018, ISBN: 978-1-13862-700-0. [Online]. Available: https://www.realtimerendering.com/.

[3] E. Lengyel, *Foundations of Game Engine Development, Volume 1: Mathematics*. Terathon Software LLC, Sep. 2016, ISBN: 978-0-9858117-4-7.

[4] The Khronos Vulkan Working Group, *Vulkan 1.3.236 - A Specification (with all registered Vulkan extensions)*, Contains extensions not ratified by Khronos, and is therefore not official yet, The Khronos Group, Inc., Dec. 1, 2022. [Online]. Available: https://www.khronos.org/registry/vulkan/specs/1.3-extensions/html/vkspec.html (visited on Dec. 7, 2022).

[5] The Khronos Vulkan Working Group, *Vulkan 1.1 Quick Reference*, 2018. [Online]. Available: https://registry.khronos.org/vulkan/specs/1.1/refguide/Vulkan-1.1-web.pdf (visited on Dec. 7, 2022).

[6] B. Burley, "Physically Based Shading at Disney," *Walt Disney Animation Studios*, Aug. 2014. [Online]. Available: https://media.disneyanimation.com/uploads/production/publication_asset/48/asset/s2012_pbs_disney_brdf_notes_v3.pdf (visited on Dec. 7, 2022).

[7] R. L. Cook and K. E. Torrance, "A Reflectance Model for Computer Graphics," *ACM Trans. Graph.*, vol. 1, no. 1, pp. 7–24, Jan. 1982, ISSN: 0730-0301. DOI: 10.1145/357290.357293.

[8] B. Walter, S. R. Marschner, H. Li, and K. E. Torrance, "Microfacet Models for Refraction through Rough Surfaces," in *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, ser. EGSR'07, Grenoble, France: Eurographics Association, 2007, pp. 195–206, ISBN: 9783905673524.

[9] B. Smith, "Geometrical shadowing of a random rough surface," *IEEE Transactions on Antennas and Propagation*, vol. 15, no. 5, pp. 668–671, 1967. DOI: 10.1109/TAP.1967.1138991.

[10] C. Schlick, "An Inexpensive BRDF Model for Physically-based Rendering," *Computer Graphics Forum*, vol. 13, no. 3, pp. 233–246, 1994. DOI: 10.1111/1467-8659.1330233.

[11] W. E. Lorensen and H. E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 163–169, Aug. 1987, ISSN: 0097-8930. DOI: 10.1145/37402.37422.

Online references

[12] G. Hoare. "Project Servo: Technology from the past come to save the future from itself." (Jul. 2010), [Online]. Available: http://venge.net/graydon/talks/intro-talk-2.pdf (visited on Dec. 7, 2022).

[13] "TIOBE Index." (Dec. 2022), [Online]. Available: https://www.tiobe.com/tiobe-index/.

[14] StackOverflow. "StackOverflow Developer Survey." (2016), [Online]. Available: https://insights.stackoverflow.com/survey/2016 (visited on Dec. 8, 2022).

[15] StackOverflow. "StackOverflow Developer Survey." (2017), [Online]. Available: https://insights.stackoverflow.com/survey/2017 (visited on Dec. 8, 2022).

[16] StackOverflow. "StackOverflow Developer Survey." (2018), [Online]. Available: https://insights.stackoverflow.com/survey/2018 (visited on Dec. 8, 2022).

[17] StackOverflow. "StackOverflow Developer Survey." (2019), [Online]. Available: https://insights.stackoverflow.com/survey/2019 (visited on Dec. 7, 2022).

[18] StackOverflow. "StackOverflow Developer Survey." (2021), [Online]. Available: https://insights.stackoverflow.com/survey/2020 (visited on Dec. 7, 2022).

[19] StackOverflow. "StackOverflow Developer Survey." (2020), [Online]. Available: https://insights.stackoverflow.com/survey/2021 (visited on Dec. 7, 2022).

[20] StackOverflow. "StackOverflow Developer Survey." (2022), [Online]. Available: https://survey.stackoverflow.co/2022/ (visited on Dec. 7, 2022).

[21] "Rust." (Dec. 7, 2022), [Online]. Available: https://www.rust-lang.org/ (visited on Dec. 7, 2022).

[22] M. Russinovich. "Twitter post." Mark Russinovich is the CTO of Microsoft Azure at the time of writing. (Sep. 20, 2022), [Online]. Available: https://twitter.com/markrussinovich/status/1571995117233504257 (visited on Dec. 7, 2022).

[23] S. Vaughan-Nichols. "Linus Torvalds: Rust will go into Linux 6.1." (Sep. 19, 2022), [Online]. Available: https://www.zdnet.com/article/linus-torvalds-rust-will-go-into-linux-6-1/ (visited on Dec. 7, 2022).

[24] Statista. "Games market revenue worldwide in 2022, by device." (Nov. 11, 2022), [Online]. Available: https://www.statista.com/statistics/278181/global-gaming-market-revenue-device/ (visited on Dec. 7, 2022).

[25] Statista. "Global box office revenue from 2004 to 2021, by region." (Mar. 16, 2022), [Online]. Available: https://www.statista.com/statistics/264429/global-box-office-revenue-by-region/ (visited on Dec. 7, 2022).

[26] Oxford English Dictionary. "Game." (Dec. 2022), [Online]. Available: https://www.oed.com/view/Entry/76466 (visited on Dec. 7, 2022).

[27] R. Guy and M. Agopian. "Physically Based Rendering in Filament." (Feb. 2019), [Online]. Available: https://google.github.io/filament/Filament.html (visited on Dec. 7, 2022).

[28] M. Lysenko. "Meshing in a Minecraft Game." (Jun. 2012), [Online]. Available: https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/ (visited on Dec. 7, 2022).

Appendix B

Source Code

The source code of the *Orion* engine and this LaTeX document is available here for ETH members. The commit hash at the end of the Bachelor's project is `eb4149af...`, tagged `v1.0.0`. The source code repository is mirrored at `https://gitlab.com/Zortac/orion` for access to the general public.

Appendix C

List of Rust Libraries used

Rust libraries are permanently listed on crates.io, which should be consulted for further information.

- `atomic_refcell`, an atomic variant of the RefCell container
- `bytemuck`, a library containing utilities for safely converting between "plain old data" types
- `chrono`, a library containing various time and date utilities
- `egui`, a pure-Rust GUI library inspired by Dear ImGui
- `egui_winit_vulkano`, glue code for using `egui` with `winit` and `vulkano`.
- `figment`, a configuration handling library
- `legion`, an Entity Component System library, used as the main game state management system in *Orion*
- `nohash-hasher`, a hash function that does nothing
- `nom`, a parser combinator library
- `obj-rs`, a Rust implementation of the Wavefront OBJ file format
- `once_cell`, a container that can only be written to once
- `qoi`, a Rust implementation of the Quite OK Image Format (QOI)
- `qoi-3d`, a custom extension of the QOI format to handle voxel data/3D images, source code is available here
- `quote`, a library containing helpers for writing procedural macros in Rust
- `rand`, a library containing utilities for random number generation
- `rustc-hash`, a library containing a fast, non-cryptographic hash function
- `serde`, a framework for serializing and deserializing Rust data structures
- `serde_json`, the JSON backend for `serde`
- `syn`, a library containing helpers for writing procedural macros in Rust
- `toml`, the TOML backend for `serde`
- `ultraviolet`, a high performance mathematics library for computer graphics applications
- `vulkano`, a safe wrapper library of the Vulkan graphics API

- `vulkano-shaders`, macros for compiling GLSL shaders to SPIR-V at compile time for use with `vulkano`
- `vulkano-win`, integration of `vulkano` with `winit`
- `winit`, a cross-platform windowing library