MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Game Engineering and Simulation Technology

Modular game engine in Rust - Comparing performance and memory usage of subsystems to C++

By: Lukas Vogl, BSc.

Student Number: gs16m007

Supervisors: Dipl.-Ing. Stefan Reinalter
Mag.rer.nat. Dr.techn. Eugen Jiresch

Wien, April 21, 2018

FH University of Applied Sciences
TECHNIKUM WIEN

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Wien, April 21, 2018 Signature

Kurzfassung

Modulare Spieleengines zeichnen sich dadurch aus, dass sie intern aus verschiedenen Subsystemen bestehen die unterschiedlichste Aufgaben abarbeiten. Beispielhafte Systeme sind unter anderem Speichermanagement, Rendering oder Physiksimulation. Die Gemeinsamkeit zwischen den Systemen, unabhägig davon wie hardwarenahe oder abstrakt diese sind, sind Aspekte wie Performance und Speicherverbrauch. Um möglichst viel Kontrolle über diese Bereiche zu haben entscheiden sich viele EntwicklerInnen für Systemprogrammiersprachen wie C++ als Entwicklungswerkzeug. Im Zuge dieser Arbeit wird der Autor die seit 2015 existierende Programmiersprache Rust verwenden um ausgewählte Subsysteme einer modularen Spieleengine zu implementieren. Ziel der Arbeit ist es zu untersuchen, ob Rust durch seine neuen Konzepte gängige Schwierigkeiten bei der C++ Entwicklung vermeiden und gleichzeitig eine gleichwertige Performance liefern kann. Dafür werden die in Rust implementierten Systeme zusätzlich in C++ implementiert und anschließend in verschiedenen Szenarien vermessen und verglichen. Aus den Ergebnissen wird evaluiert ob Rust als Programmiersprache für Spieleengines in Frage kommt. Zusätzlich werden die Implementierungsdetails der verschiedenen Sprachen und Systeme behandelt, wodruch aufgezeigt wird welche Unterschiede zwischen den beiden Sprachen bestehen.

Schlagworte: Rust, C++, Engine, Speichermanagement, Performance

Abstract

Modular game engines are defined by the fact that they are composed of different subsystems working on many distinct tasks. Exemplary systems are, inter alia, memory management, rendering or physics simulation. The similarity between the systems, regardless of how low-level or abstract they are, are performance and memory consumption. To gain control over these fields most programmers choose system programming languages such as C++ as development tool. In this thesis the the author chose the programming language Rust to implement selected subsystems of a modular game engine. Is it the goal of the thesis to investigate whether Rust can avoid common difficulties known from C++ due to its new concepts while maintaining C++ like performance. For this purpose the selected systems will also be implemented in C++. They are then surveyed in different scenarios and compared to each other. The results are evaluated to see whether it is worth considering using Rust as a language for game engine programming. Furthermore the implementation details ofthe different languages and systems are discussed whereby the differences between the two languages are outlined.

Keywords: Rust, C++, Engine, Memory management, performance

Acknowledgements

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Contents

1 Introduction

Game engines are an essential part of the gaming industry. Todays state-of-the-art game engines have committed themselves to the goal of creating visually appealing games while providing reasonable performance. Achieving this requires the engine engineers to invest a great amount of time and know-how of underlying hardware. Many of these engines, choosing Unity and Unreal Engine 4 as example, are using C++ as underlying technology. C++ is the language of choice due to it's capabilities of managing memory manually without the limitations of a garbage collector. These capabilities are the foundation for high performance software and essential to game engines. But while the benefits of manual memory management are indisputable it also comes with common pitfalls.

This thesis aims to examine whether the system programming language Rust can be used as a replacement for C++. Rust claims to avoid pitfalls made in C++ while maintaining similar performance. As a basis for discussion the author will implement selected engine subsystems: memory management, containers and an Entity Component System (ECS). All systems, except for the ECS which already exists in Rust, are written in Rust and C++ to later measure and compare their performance in different scenarios. The results of the measurements shall then serve as the basis for the discussion if Rust can be considered as a viable language for game engine programming.

Chapter 2 will introduce the reader to the history and evolution of game engines. It will also outline state-of-the-art products and shortly describe them. In chapter 3 important tools that are tightly related to the underlying engine and the concept of an asset pipeline are discussed, concluding the chapter with a section presenting the theory and examples of selected engine subsystems. The next chapter provides the reader with an overview of the Rust programming language. It describes the current state of Rust and creates a basic understanding of it by introducing the most important concepts and patterns. It will then compare common and well-known C++ problems and pitfalls with corresponding code in Rust. At the end the author outlines encountered difficulties that can occur when working with Rust. In chapter 5 the author talks about the implementation details of the implemented subsystems and where the differences between Rust and C++ are visible. The development process, architecture and project setup of the Spark engine (the implemented submodules will serve as a basis for this engine in future work) will be discussed. The performance measurement results and observed scenarios are then compared and discussed in chapter 6. Chapter 7 will then finish the thesis with the conclusion.

2 Game engines

Game engines are tightly connected to the evolution of video games themselves. Where 50 years ago a game was build out of hardware the rapid development of a computer's processing power and storage capabilities changed the process of how games are made. Today a game runs on machines assembled from multiple cores, several Gigabytes (GBs) of Random-Access-Memory (RAM) and a powerful graphics processing unit (GPU). But whether the target platform is a PC or a specialized gaming console every modern game has to fulfill certain constraints to be a viable product. This chapter will highlight some of the most important milestones in the history of video games and game engines. It will also give an overview of well-known products on the game engine market and will conclude with the description of selected submodules, the underlying building blocks of a game engine.

2.1 Evolution of game engines

When the first developers started to create video games, the term *game engine* was non existent. At this time the software that ran the simulation a player experienced was tailored to the needs of a specific genre, hardware and game. It was then in the 1990s and with the rise of games like *Doom (1993)* and *Quake (1996)* that certain software was referred to as a *game engine*. The mentioned games separated their technical backbones into different components, creating an architecture that distinguishes between core software modules and game specific entities such as art assets, levels and the general rules of the game. Due to the well-designed architecture and separations the effort to create a new game, where the general concept is the same, was reduced from writing every system and piece of code to creating new art and only tweak and configure the software of previous games. This was also the birth of the *modding* community, where individuals and also studios modify existing games or engine software to create new content or whole games.

From that time on the developers created their games with modding and future extension in mind. Smaller studios started to license parts of the engine software they could not afford to create by themselves, be it money, time or man power. With the concept of licensing, studios, that built the extensible and reusable software packages, created another source of income. But while the goal of an extensible and reusable software collection is a desirable one, the line between a game and an engine is often softer than desired. Because of the games nature and their specific genre rules, it is hard, if not even impossible, to develop a generic engine that can serve as a template for every game. It became the responsibility of engine developers to find

the balance between general-purpose functionality and game or platform tailored optimizations. This trade-off has to be made because the developer can only assume how the software will be used. And so a game engine developed and optimized for rendering frames per seconds (FPSs) will probably not run a real-time strategy (RTS) game with maximum performance due to the different rule sets and features both genres require.

Empowered by the wish of creating games that can be modified and licensed, bigger studios started to create commercial engines. *Id Software*, the company that created *Doom* and the *Quake* trilogy, opened up the field with their FPS engines in the early 1990s. *Id Software* was then followed by *Epic Games, Inc.*, creator of the Unreal Engine, which powered their well-known game *Unreal* and later the *Unreal Tournament* series. The current version, the Unreal Engine 4 (UE4), is one of the most popular game engines of our time and will be described in the next section. Other game engines released in the same period and which should not be left unmentioned are *CryENGINE* (Crytech), *Source Engine* (Valve), *Frostbite* (DICE) and *Unity*.

For a long time many of the mentioned engines, if not all, followed a model of selling licenses to developers for accessing the engine and its source code. But it was around 2009 and again in 2015 that big engine developers, including Unity and Epic Games, decided to rework their business model and let developers use their software for free. The license terms often include a revenue share if the game should be successful but the basic usage of the engine is free most of the time. Although this certainly had a huge impact on smaller products and teams, that could not compete with the teams working at Epic or Unity, this decision lowered the entrance barrier for game developers and students. This transition to accessible license models and free access can be seen as one of the biggest milestones in the modern history of game engines.

Speaking of Unity and Epic, the next section will describe modern game engines on the market, how they work and what they are used for. In contrast to Unity and Unreal Engine 4, two smaller engines will be described to show the difference between approaches and why the flexibility of smaller engines and teams can also be an advantage over big software projects.

2.2 Modern commercial game engines

As described in the previous section game engines evolved from extensible game or genre tailored software to standalone viable products used to create different games. The market is actually dominated by two big engines, Unity and Unreal Engine 4, whereas the rest is separated between custom in-house and several medium to small engines and open source projects. The author is going to describe the features and license models of the two big solutions as well as of two smaller but more flexible ones.

2.2.1 Unreal Engine 4

As already known *Epic Games, Inc.* released the first version of the Unreal Engine with their game *Unreal*. The solution was quickly adopted and Epic developed two more generations,

Unreal Engine 2 and 3, before the current generation, called UE4, was released. All generations powered well-known games such as Deus Ex (UE1), Tom Clancy's Splinter Cell: Pandora Tomorrow (UE2), Gears of War (UE3) or Fortnite (UE4), to name a few.

While previous generations could be licensed by developers for a license fee, Epic first changed their licensing model to a monthly fee and later lowered the access barrier even more by getting rid of any license fee at all. The current version of UE4 is for free and source code access can be requested at GitHub (https://github.com/EpicGames/UnrealEngine). This move allowed many developers to create their projects with UE4 and Epic is eligible for a 5% share if the game makes more than 3000$ per calendar quarter in revenue.

When working with UE4 the developer can choose to build everything from source or work with distributed pre-built binaries. Due to the fact that the Unreal family of engines was developed with first- and third person shooters in mind, the source code access is often appreciated by developers to tweak and rework the engine in a way to better run games from other genres. Regardless whether the engine was built from source or not, when working with it the user interacts with the integrated editor, often also referred to as *UnrealEd*. It is the entry point to nearly every tool that ships with **U4!** (**U4!**).

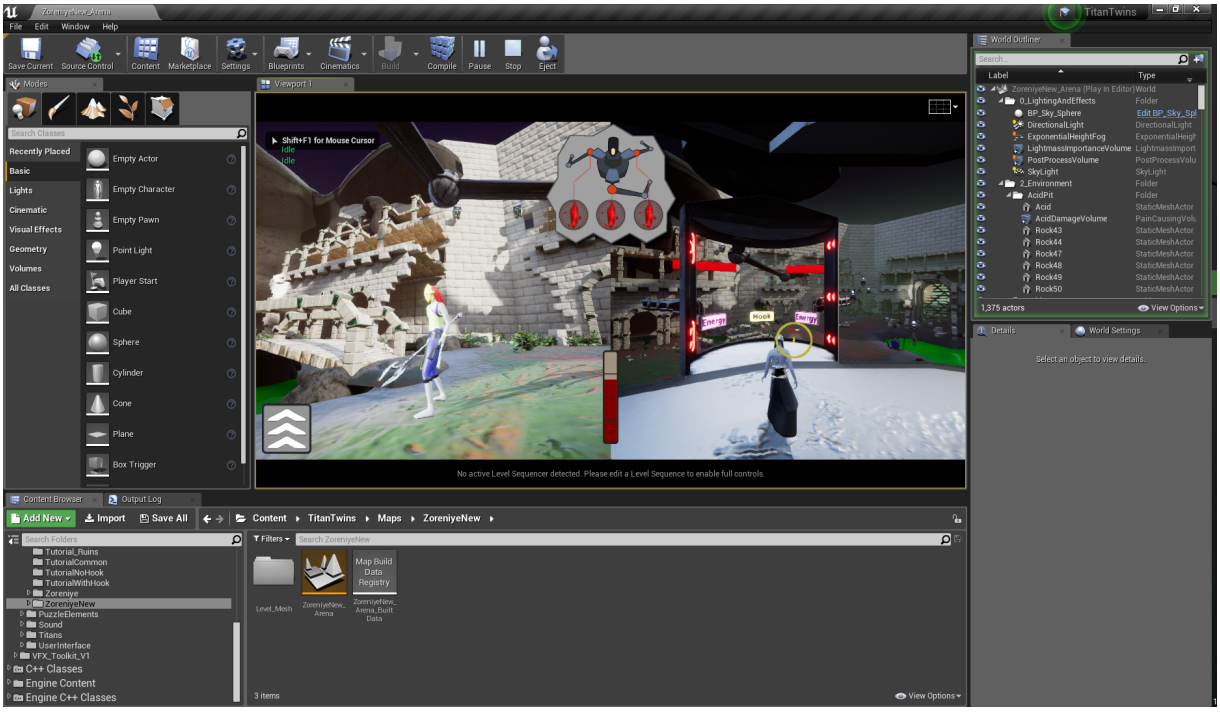

Figure 1: The editor shipped with UE4

Going from left to right and top to bottom, Figure 1 shows a selection for actors (objects that can be placed in the game world), a game world view, the world outliner (hierarchy of actors) and the content browser, that lists all folders and contained assets. The editor is a place where designers can create the game world, where artists can author special effects and their assets directly in the game and where designers and programmers can develop the rules and logic

needed to drive the world. For creating this logic UE4 offers two possible ways: scripting via the *Blueprint* system or directly in C++. UE4 comes with the integrated *Blueprint Visual Scripting* system that allows for gameplay programming using the concept of a node-based graph. This system is versatile and if it has reached it's capacities a programmer can still implemented the necessary or performance critical parts in C++ and expose an interface for the designer to use the component together with built-in blueprints.

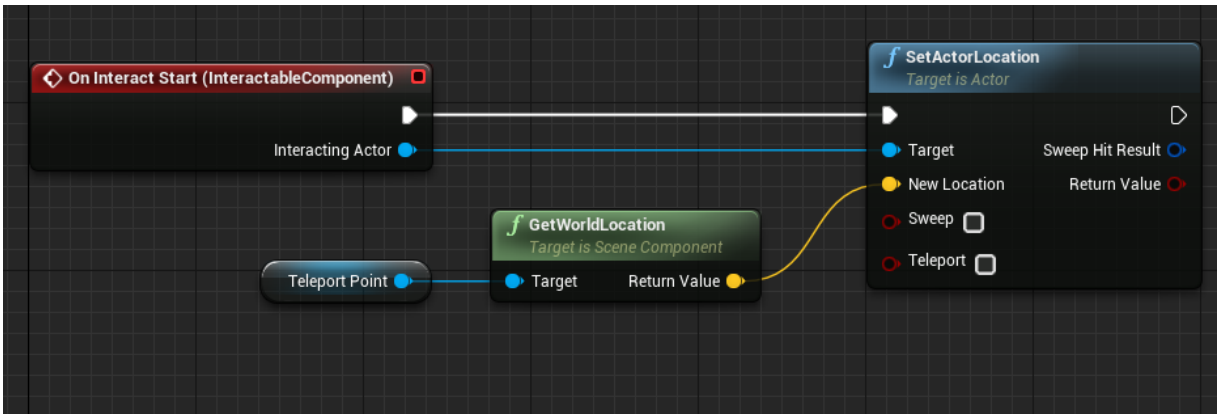

Figure 2: A very simple example that shows how the blueprint visual scripting looks like in UE4

A very similar system of node-based graphs is used for building complex materials in UE4. A material describes how an object, that uses it, looks like. It can for example define how it reacts upon light, whether the object appears to be rough or specular. The advantage of UE4's approach, using a node-based abstraction for materials, is that it makes life easier for developers when creating and authoring styles or effects. Without this system in place it would require a graphics programmer to write a shader. There are many different kinds of shaders but to keep it simple and because it is not necessary for now it is enough to say that a shader is a program that is executed on the GPU and defines which color a pixel shall display at the end. Writing such shaders is not a trivial task. The node-based approach puts a layer of abstraction upon this process that allows developers to work with materials without needing to know the low-level internals of shaders.

It should not be unmentioned that working with UE4, especially when needing very specific features or techniques that are not exposed to the abstraction systems, it requires a fond knowledge of C++ and the internal engine systems to fulfill the task. UE4 has grown in the past years and due to the fact that its source code is authored by many developers, both Epic engineers and open source contributors, bug reports are likely prioritized in relation to the needs Epic has for its own products developed in UE4. The documentation is well written for engine tools and visual scripting but is a little weak on the C++ side. When working directly in C++ sometimes compile-times can decrease iteration times in UE4 which is due to the Unreal Build Tool (UBT) and some default configurations on how to deal with precompiled headers and unity builds.

To conclude this paragraph about UE4 it can be said that Epic created an engine that is suitable for creating games from any genre, which sometimes require tweaking the engine's

source code. UE4 offers a variety of tools and integrations for asset authoring software which makes it easy for developers to start working with it. The abstraction systems paired with the revenue share licensing model lowered the entrance barrier for new game developers and are a reason for why the engine is in such a good market position. The development experience and fast iteration times fade away the more direct C++ coding is involved but again a trade-off between developing every system in-house and dealing with problems that rise has to be made.

2.2.2 Unity

The biggest competitor on the market for UE4 is Unity, and there are several reasons why there is a second product that viable. Unity was born in 2004 and created by the company called *Over the Edge* which was later re-branded to *Unity Technologies*. Contrary to the evolution of UE4 that evolved from moddable games and with easy extension in mind, Unity was created to lower the access barriers for 2D and 3D game development. After their first game, GooBall, failed commercially the founders discovered how valuable engine software and tools are. This led to their decision of building an engine that is accessible to as many people as possible and that promises ease of development and cross-platform support. With these principles in mind Unity became a product that was adopted by many developers and nowadays is used by many studios and small developers.

In contrast to UE4 the source code of Unity is not freely accessible but it can be acquired by acquiring an enterprise subscription. If source code access is not needed, which is the case more a major percentage of the developer products, Unity offers a very fair licensing model. A personal license can be registered for free, with the constraint that a product does not generate more revenue than 100.000$ annually. This version includes all Unity core features as well as continuous upgrades and access to Unity beta versions. The next tier, called Unity Plus, includes everything of the previous one and adds more flexible customizations (custom splash screen, pro editor UI), more in depth analytics and it alters the cap of concurrent players on multiplayer games hosted by Unity. The Plus tier costs 35$ per month and is constrained by a revenue cap of 200.000$ per year. If the income exceeds that limit, the Unity Pro tier comes without any revenue limit at all for 125$ per month. It adds again more in-depth analytics and alters the concurrent players cap once more. Independent of the selected tier, Unity includes an editor to interact with its tools and the game world.

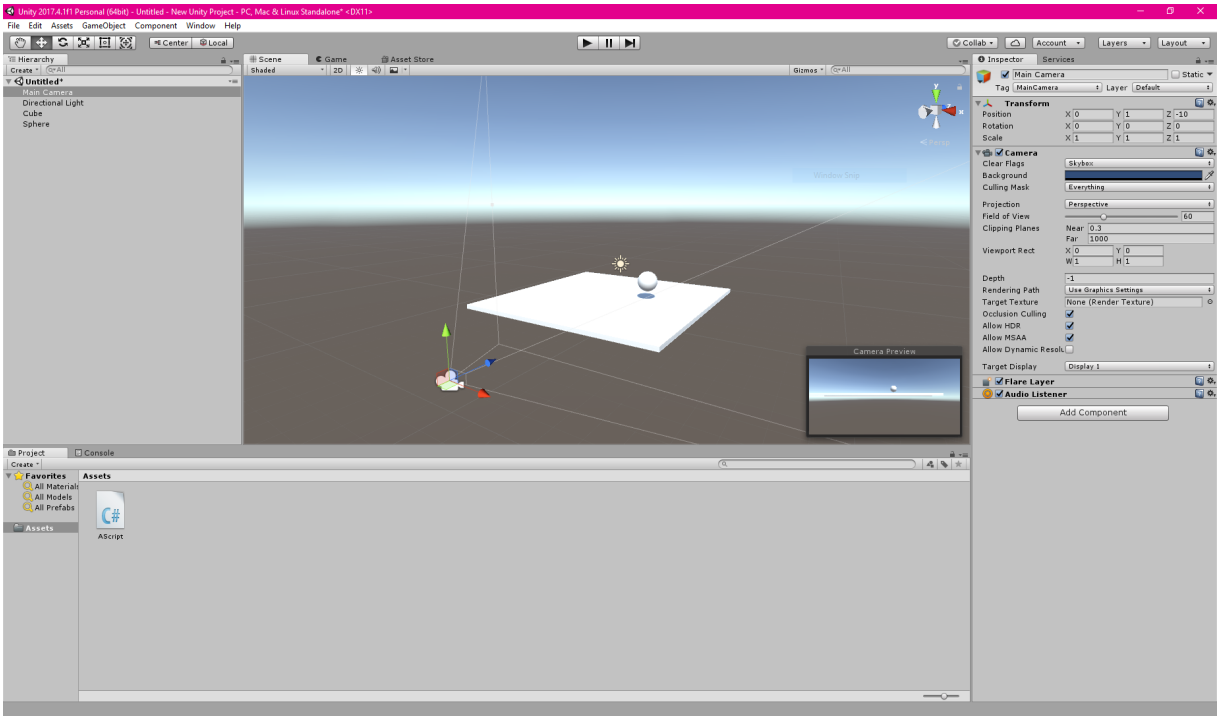

Figure 3: The default layout of the editor shipped with Unity

The editor is quite similar to the one of UE4 but the views and tools differ in their naming. Going from left to right and top to bottom Figure 3 shows the Hierarchy, collection of game objects that are placed in the game world, the scene or current world, the Inspector, a list of components attached to a single game object and the project structure showing folders and assets. What was called an actor in UE4 can be compared to a game object in Unity. A game object is an entity that can be placed in the world but does not contain any logic itself. Rather it is a container that holds a collection of components, each holding data or logic. This system of entities and components is called an Entity Component System and will be described in detail at the end of this chapter. Because of the ECS components can be shared and reused among projects which simplifies the development process and cuts iteration times when done properly. It was already mentioned that components can also contain logic which already describes the basics on how game rules and logic are created in Unity. Where UE4 provides a visual scripting system Unity does not have anything similar. In Unity scripting is done in C#, a high-level programming language running in the MonoDevelop/.Net runtime. The work flow of creating gameplay logic often starts with a programmer creating a new scripting component in C# which then later can be used by designers to assemble new game objects without needing to touch any C# code. This is ensured by a system that allows programmers to expose certain properties of the script to the editor, where values can be tweaked by designers to fit the needs of the game. Although Unity does not have a visual scripting system it is easy to create and run scripts because they are contained withing a single component and an error inside of them will not crash the whole engine but is guarded by the scripting runtime.

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class AScript : MonoBehaviour {
6     // Use this for initialization
7     void Start () {
8     // Execute code at component start ...
9     }
10
11    // Update is called once per frame
12    void Update () {
13    // Execute code once per frame ...
14    }
15 }
```

Code 1: Example of an empty C# script in Unity

While previous versions of Unity supported two additional scripting languages, Boo and Javascript, C# was the most dominant scripting language used among Unity games.

A long time it was the node-based material graph that separated Unity and UE4, but since version 2018.1 (that is in beta-state at the time of writing) Unity introduces the *Shader Graph*, seen in Figure 4, which serves as a visual interface for building shaders. Together with the *Scriptable Render Pipeline* and the new job system, programmers gain more access over low-level and performance critical parts of the engine which closes the gap to UE4 a bit more regarding performance and control.

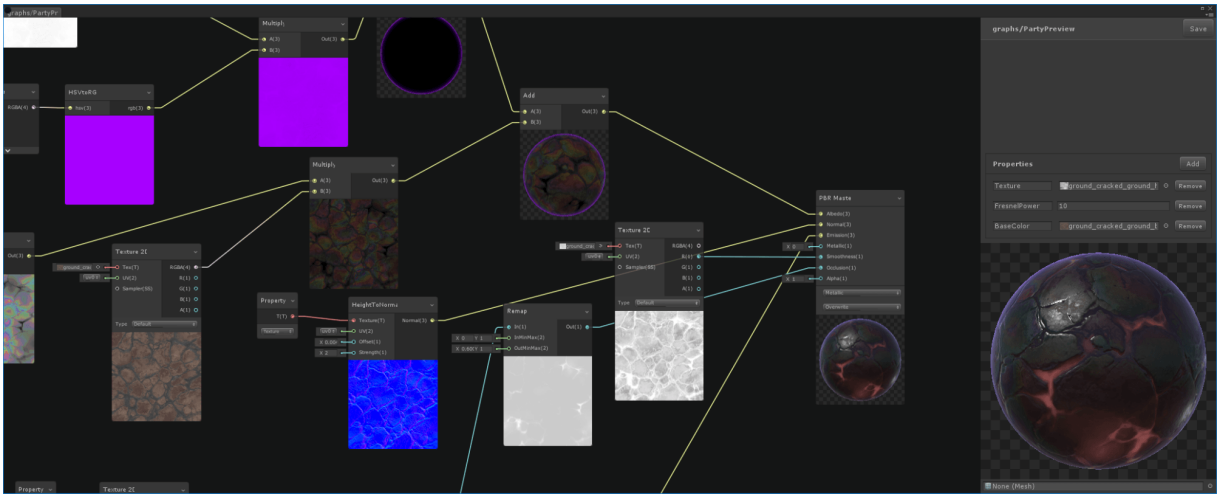

Figure 4: The new visual shader graph editor that ships with Unity 2018.1

Together with all these features and the asset store, a marketplace hosted by Unity where developers can upload, download and sell components, art and different plugins Unity is a tool suitable for games from any genre. Its low entrance barriers and support for many different platforms make it a powerful competitor to UE4. It is also due to these aspects that Unity is the

tool of choice for game developers that plan to release on mobile platforms, such as Android or IOS. Unity is also the engine often chosen for rapid prototyping due to good iteration times and the asset store.

2.2.3 Molecular

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

2.2.4 Tombstone

The Tombstone engine is developed by *Terathon Software*, which was founded by Eric Lengyel in 2001. It is a cross-platform engine written in C++ that runs on Windows, Playstation 4, Linux and MacOS. This engine is mentioned in this chapter because it features an interesting architecture and invented and uses several innovative and robust techniques. The Tombstones engine architecture is separated into several layers of managers combined with general utility libraries and a plugin system. General utilities such as memory management and a container library build the bases for managers of different systems. These system manager include for example the resource, thread and graphics manager. Upon this layer more high level managers are built that handle the game world, the scene hierarchy or the animation system. To work with these systems Tombstone offers several plugins, shipped with the engine.

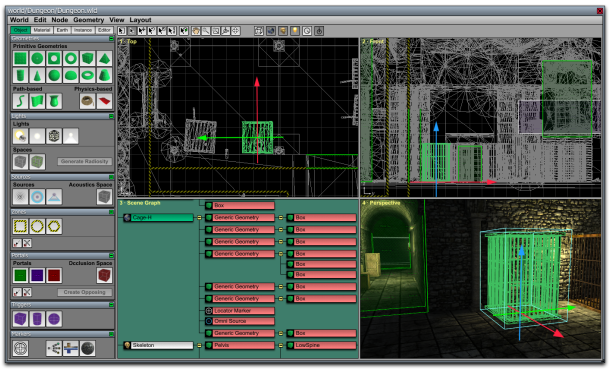

Figure 5: The world editor plugin used to edit worlds in the Tombstone engine

Among these there is also the world editor that, similar to the bigger engines, allows for creating worlds and levels. Beside the world editor other plugins are a visual shader editor, several tools for importing assets and user interface (UI) builder to generate panels and wid-

gets. Beside the well-defined architecture Tombstone features two standards also created by Eric Lengyel, named the Open Data Description Language (OpenDDL) and the Open Game Engine Exchange Format (OpenGEX). These protocols are used for exchanging data with asset authoring software like Maya or Photoshop and to store serialized data. The design of the architecture if very well described in a visual form directly at the Tombstone homepage (http://tombstoneengine.com/architecture.php).

Contrary to the bigger companies the Tombstone engine's licensing model does not offer free versions. To acquire a lifetime license for all platforms (Windows, Linux & MacOS) a fee of 495$ per person has to be paid. This license does not include any revenue share nor is it limited to a specific amount of shipped games.

The Tombstone engine is an example of a quality piece of software and showcases in what direction an engine can develop if the design goals are carefully thought out. It also tries to establish open standards for exchanging data between different parts of the game development toolchain. If such standards would be implemented and obeyed by more engines it would ease the process of migrating the workflow of engineering teams to another engine or software that would better fit their current project.

2.3 Rust game engines

Because the goal of this thesis is to research whether Rust is a potential choice for a game engine's main language, the author wants to highlight work already done in that field. All previously described solutions are driven by C++. While some of them use it for the entire engine, others built only their core systems in C++ and use higher level languages on top of it. To showcase what was already done in the Rust ecosystem and to better understand some decisions the author made when implementing the submodules, this section is dedicated to the two biggest engines written in Rust - Piston and Amethyst.

2.3.1 Piston

Piston is a modular game engine written in Rust that is now maintained as an open source project on GitHub. It was created in 2014 byt developer Sven Nilsen. Having been a testing field for 2D graphics in Rust, working with different back-ends, the project evolved into what is called Piston today. The engine is separated into different modules, each handling a specific task. The contributors are currently developing solutions for 2D and 3D rendering, window management, plugins for integrated development environments (IDEs) like Visual studio and other systems connected to games. The Piston project aims to generate an ecosystem that reduces development cost for games. Due to this desire the functionality is separated into the distinct modules to allow them to be reused between multiple projects. Contrary to Unity or UE4 Piston does not include any editor to work directly in the 3D or 2D world. It is up to the

developer of a game or an open source contributor to implement such a tool, that can be put on top of the already existent ecosystem libraries.

2.3.2 Amethyst

The second engine commonly mentioned in the Rust ecosystem in Amethyst. It is a data-oriented game engine written in Rust. On the project page the developers describe that Amethyst is inspired by the *Bitsquid Engine*, that is now called *Autodesk Stingray*. Being inspired by the Bitsquid the engine's goal include a parallel architecture featuring an ECS and an optimized renderer using modern application programming interfaces (APIs) such as Vulkan or Direct3D 12 and greater. On the tool side it plans to split the commonly known world editor into several distinct tools, but at the moment just a single tool, used for creating and deploying projects, is implemented.. Just as Piston, Amethyst is an open source project under the MIT/Apache2 license that is hosted and maintained at GitHub.

Due to the infancy of both projects and the language in general neither Piston nor Amethyst can be compared to commercial engines like Unity or UE4. But these projects create a fond basis for game development in Rust and both already built a strong community that is eager to push the boundaries of Rust.

3 Engine architecture overview

Now, after the last chapter described many different kinds and sizes of engines, this section is going to examine the architecture that powers an engine. At the beginning an overview of a modern game engine's runtime architecture is given. That overview is then followed by a description of several selected submodules. These power different systems of the engine and are responsible for its performance and functionality. The submodules were chosen based on the author's opinion of their relevance and importance to the backbone of an engine.

3.1 General runtime architecture

When talking about software parts of an engine in a high level fashion it can be separated into two clearly diverging parts, tools and the runtime component. This section will almost entirely discuss the runtime part while only mentioning the most important tools at the end. An engine consists from many modules which are separated into different layers. Game engines share this architectural design decision with many other software projects that reach a specific size. Layers group modules together with other ones operating in the same order of magnitude as their siblings. A layer often depends upon lower ones but shall not have any dependencies to upper ones. This ensures that coupling between layers is loose which leads to more stable software solutions. An exemplary illustration of a generic engine's runtime architecture can be seen in Figure 6.

The complexity of modules in a layer grows when ascending through them from bottom to top. Where the lowermost layers include low-level systems essentially drivers [1], platform-dependent 3rd party software development kits (SDKs) or platform-independent abstraction components. Traversing the hierarchy upwards from core systems, over resource management, to general rendering and gameplay foundation modules, at one point the uppermost layers are reached. Those encompass game specific subsystems that can vary from one game to another. With the crude knowledge of the different layers involved in an engine it can be emphasized that a game engine is a highly complex software and building one is an endeavor that requires expertise, experience and time. It is important to properly reason out the architecture and uphold the focus onto the engine's goal. Maintaining focus while following the sketched out architecture will help to avoid unnecessary coupling and the implementation of irrelevant features or systems. The rest of this section will describe some modules from Figure 6 in-depth to investigate how they work and how they contribute to the combined whole.

[1] A program that controls or communicates with a hardware device

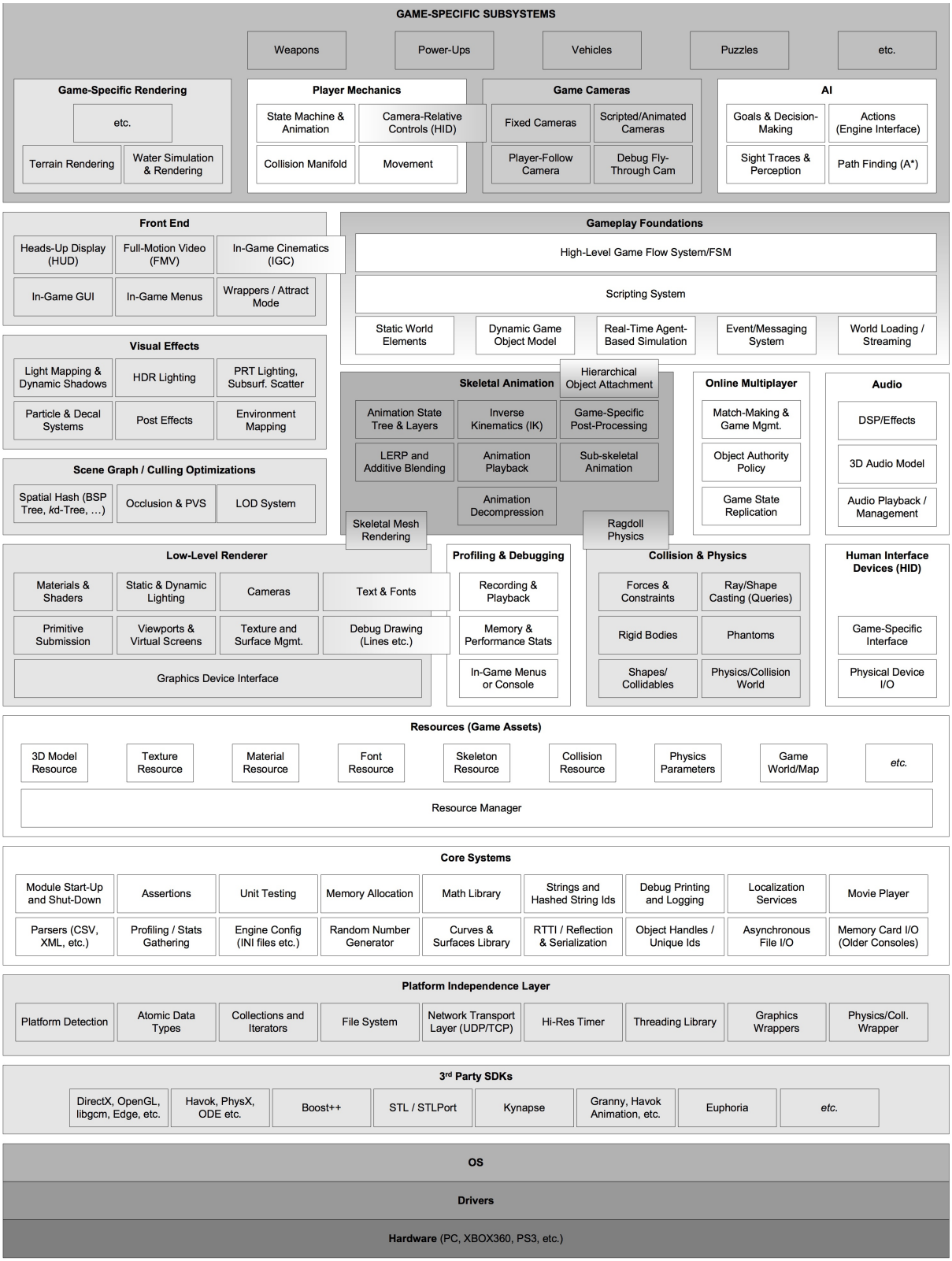

Figure 6: Illustration showing common modules grouped into distinct layers of a large scale engine solution

3.2 Memory Management

One key constraint nearly every engine ha+s to fulfill is running games with a high frame rate. Because games are real-time simulations the time window for running gameplay logic and rendering a single frame is very limited. To complement this with discrete numbers for a game, running with 60 FPS a slice of 16.6 ms can be used per single frame. To stay into this limits game developers came up with optimization techniques and algorithms that speed up calculations and processing. But the performance of code is not only dependent upon the efficiency of an applied algorithm but also how the program manages and uses its resources, especially memory. Controlling how an engine utilizes the RAM is mandatory for guaranteeing high performance. The two most commonly applied memory usage optimizations are either reducing the amount of dynamic allocations at a game's runtime or allocating bigger sections of memory to store data in contiguous blocks. To solve these problems engines often implement custom memory allocators that have a better runtime performance then using the existing system allocator.

3.2.1 Custom allocators

Custom allocators are facilities that optimize dynamic memory allocations and reduce the performance penalty introduced by them. Its an allocator's main purpose to provide a source of memory the requester can work with. The process of finding a block of memory that fits the request is called an allocation. When requested memory is not needed anymore, the allocator takes it back and reuses it for another request if possible. Almost all system programming languages come with a heap allocator, that can handle requests for memory at runtime. These so called *heap allocations* are made by either calling `malloc() & free()` (C) or `new & delete operators` (C++). But caused by different aspects these allocations are rather slow. The performance penalty is mainly caused by two factors. At first, because any size of allocation has to be fulfilled by a heap allocator, it needs a lot of internal allocation management which introduces an overhead, making heap allocations costly. The second factor that contributes to the cost of dynamic allocations is that a context switch from user-mode to kernel-mode takes place before any allocation.

To better understand why this introduces a performance penalty one has to know what divides the user- and kernel-mode. Many modern operating systems (OSs) distinguish between these two modes. A process, that is running in user-mode, often has no capabilities to directly talk with the underlying hardware, both for security and portability reasons. The way a user-mode process communicates with lower level systems is by issuing a *system call*. Such calls trigger routines in the hardware that switches into the privileged kernel-mode, executing the requested kernel procedure while strictly controlling what is happening. After it terminates, the procedure forces a switch back into user-mode and the process continues at the point right after the system call.[1]

And it is that context switching on system calls that slows down dynamic memory allocations. Whenever a call to `malloc()` or similar is issued, under the hood it forwards the request to a system call and the allocation is fulfilled in kernel-mode. But because every game engine needs some kind of dynamic allocations in some places, they have to be optimized. The common solution for this problem is a collection of customer allocators implemented in the engine's core systems. They resolve the problems bound to dynamic allocations by minimizing context switches and knowing the allocation patterns. To reduce the overhead of system calls custom allocators satisfy allocation requests from a preallocated block of memory. This block can be acquired by the system's heap allocator (or with another approach that is described in the implementation section in chapter 6). After that block was allocated every further request for memory will stay in user-mode. Assuming the usage patterns of itself, a custom allocator can handle requests more efficiently than the general purpose heal allocator. Several different kinds of allocators are implemented and the user is responsible for knowing how the memory is used and which allocator fits the job best. That allows for reducing book keeping overhead and for better performing allocations.

The memory management module of a modern engine has a huge impact onto the general performance. It serves as the basis for many other modules and more high level ones, such as for example a container library, can be built on top of it. Being one of the modules that were implemented in the thesis' project, an even more in-depth insight into the internals of such a system and how it is implemented in Rust and C++ will be given in section 5.4. It will also describe how the different kinds of allocators work internally and how they can be easily combined with other utilities to form a versatile and well-performing system.

3.3 Rendering

One of the most complex parts of a game engine is the renderer. It encompasses many disciplines from graphics programming to algorithmic knowledge and there are many different architectures to choose from. This section will give a basic overview of how a renderer can be built but going into detail would quickly exceed the intention of this section. As many software solutions that become complex over time, a renderer is often separated into different layers. To recap them in a visually form, one can find them in Figure 6 under section 3.1. At the bottom of the architecture there is the low-level renderer. Its purpose is to render geometric primitives [2] as quickly as possible without performing any redundant passes. To display the primitives onto a screen a renderer uses Graphics-SDKs such as OpenGL or DirectX. Cross-platform engines abstract the graphics API used on a specific platform behind a layer called the *graphics device interface*. This name is not a standard and can be any other one in the source code of different engines. It is the task of this interface to enumerate the available graphics devices and setup surfaces [3]. Other components of the low-level renderer are for example a system to calcu-

[2] Common primitives are points, lines or triangles

[3] Buffers on the GPU into which the renderer stores its generated values

late dynamic and static lighting of the scene or a material system for managing shaders and hardware state on a per primitive basis. Shading and lighting are broad topics and an in-depth description is omitted on purpose due to the complexity of these fields. On top of the low-level renderer another layer is often implemented to decide, which primitives to process and submit. This higher level layer manages what primitives should be rendered based on visibility determinations. Some components that are included in this layer are a level of detail (LOD) system. This systems is an optimization often used to reduce the amount of primitives for a mesh based on its distance to the camera. A model is replaced by a lesser detailed one the further away the camera is moving. If the distance is reduced again, the model is switched back to the more detailed version. Other exemplary techniques are *frustum culling*, where objects that are not 'seen' by the camera are removed, or some kind of *spatial subdivision*, where the virtual world is divided into smaller chunks which makes it more easy to reason about parts of the world that cannot be seen from a specific camera location. Additionally to the described components a rendering engine can include systems to handle various special effects and post processing effects, such as dynamic shadows, particle systems and more realistic lighting techniques. It is often also capable of rendering some kind of UI on top of the game's 3D scene.

3.4 Gameplay systems

Beside the low-level systems an engine has to provide a possibility for game developers to define the rules and attributes of the game world and what abilities a player has. This action and logic, often called gameplay, is implemented in the engine's native or a more higher level scripting language. To allow gameplay code to interact with lower level systems from the engine's core, a layer called *gameplay foundations* is introduced. What exactly is grouped into that layer differs from engine to another but two components that are found in any engine are some kind of scripting environment and a system for handling game objects and components.

3.4.1 Game objects & components

Every game engine has some facility for representing an object in the virtual world. These so called *game objects* can be crated and configured by game designers using the world editor or similar tools. But the implementation details on how these objects are modeled in code is quite complex and several different architecture techniques exist.

The object-centric approach

The first and most straight forward approach is the object-centric one. With this technique every object in the game world is an instance of a class. This class contains both, attributes and logic of the object and one instance is created for every logical game object that is added to the world. That idea of grouping data and logic together into a single package is inherently similar to the

way object oriented languages work. It is due to this similarity that leads game developers to often implement a hierarchy of game objects using inheritance. Such class hierarchies usually define a root class called `GameObject` that serves as a parent for every object in the world. The exemplary sub classes `PhysicsObject` and `RenderableObject` both are implemented as child classes to `GameObject`, having common functionality already defined in the parent class. While such a system can be intuitive and simple for a small hierarchy, as soon as complexity increases, the hierarchy starts getting deeper and its advantages fade away. With growing complexity the disadvantages of a class based solution become visible. A deep hierarchy of classes is inflexible to change requests. If a feature is requested, that does not fit into the already existing taxonomic system, it requires a lot of work or a non optimal workaround, to force the new requirement into the current model. A different problem that often arises with such a design is called the *Bubble-Up effect*. When the number of different classes grows it is more likely that similar routines are implemented in several classes. In order to reuse existing code and to provide the designers with a possibility of applying the logic to other objects of the hierarchy, some routines and data are moved into the parent class. This is a problem because in the moment when the functionality is moved into the base class it is automatically available to all children of it, even to those who were never meant to gain that certain behavior at all. Duplication of code or changing the fundamental design of the hierarchy is seen as a bigger problem than having unused logic and data associated with objects. An example for that effect can even be found in well known solutions looking at the design of UE4's `Actor` class hierarchy.

The component-based approach

If the hierarchy in an object-centric approach is assumed to become very deep or problems became big enough to redesign the system, the component-based approach can be used to simplify the hierarchy of game objects. The main difference is the usage of composition rather than inheritance. Using a design based on composition redefines how the `RenderableObject` from the last section would be implemented. Instead of being a child class of `GameObject` a `RenderableObject` is not an own class but more an instance of `GameObject` that owns a component which includes the logic on how an object is rendered. The previously child classes of the root object class are removed and their logic is encapsulated into independent classes, providing logic and data for a distinct functionality. These classes are then often referred to as *components*. A `GameObject` instance now serves as a container that holds pointers to other components or, in an even more advanced version of a composition-based approach, a list of generic components which then can be easily added, removed and queried during the runtime. But although an component-based approach has many advantages about the object-centric one it also has its share of problems. When the number of different components grows the task of handling communication between them grows more and more complex.

Beside the two described approaches, there are still other ones that are used. Some getting rid of the game object container class at all *(Pure Components)* and others using property-based

approaches, which remind of relational databases, using object ids to identify one's properties. Independent of the chosen approach every engine implements some kind of entity management and an exemplary implementation of such an Entity Component System can be seen in section 5.6.

3.4.2 Scripting

Many engines allow the designers and programmers to work in a scripting language when implementing game specific rules and content. This languages can be more high level than the engine's core one, allowing for faster iteration times and more user friendly systems. While working in the engine's native language should not post a challenge to a programmer, using a scripting language encompasses several advantages. One of them is the fact, that iteration times are cut down by a good part. Because a scripting language often removes the need of recompiling and relinking the binaries of a game or engine, idle times and well-known compilation breaks can be avoided. A scripting language is therefore in many cases an interpreted one, having its runtime integrated into the engine's core systems. It shall ne be unmentioned that there exist several solutions to use languages like C++ as scripting languages using techniques like hot-reloading (immediate solutions or via .dll reloading). Such techniques can be found implemented into the UE4 or with using *Live++* [4], a solution for C++ hot-reloading, developed by the author of the Molecule Engine, described in section 2.2.3. The most important benefit scripting languages provide is the fact that non-programmers, speaking designers and artists, can create and tweak custom logic without the need of a programmer. With that in place, programmers can focus onto more complex gameplay systems and the workload of creating new content is moved to other departments of the development team. Commonly used scripting languages that provide these benefits are, inter alia, Lua, Python, C#, Javascript and custom in-house ones.

3.5 Job System

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

[4] The interested reader can find further information at https://molecular-matters.com/products_livepp.html

3.6 Tools

Although the design and architecture of the engine's runtime systems is important for the quality of games it powers, there is another factor, namely the tool-set it incorporates, that has an impact on quality and productivity. Those tools are responsible for interacting with the engine's systems and for managing, or sometimes even creating, data, for instance scripts, 3D models or other assets. Some of those tools are used to interact and modify the game world, commonly known as world editors, while others ensure that assets are served to the engine in a specific format that better fits its needs. The variety of tools differs by every engine but mostly all of them include ad least a world editor and some kind of asset conditioning pipeline. And because the appearance and controls of the tools are different, there are also several ways how they interact with the engine and how they are designed. The following to subsections will shortly describe several tool architectures and their differences as well as the purpose of an asset conditioning pipeline.

3.6.1 Different approaches

The approaches on how deeply the tool suite is integrated into the core engine itself are quite diverse. One approach sees tools as stand-alone software applications. Tools built with this approach in mind do not use any part of the engine's core systems, having the advantage of not being dependent or coupled onto any of these parts. While the separation of engine and tools may have its benefits in another approach uses the idea of building tools on top of core systems that are then shared with the runtime. The advantage of this approach is that functionality already built for the engine's runtime can be reused by the tools and also representations of entities, speaking of game objects or components, can be shared and simplify the process of serializing them to and from the tool's environment.

(a) Standlone tool-suite (b) Integrated tool-suite on top of core systems

Figure 7: Illustrated tool-suite architecture approaches commonly found among most engines

Beside these two kinds of architecture there is a third one that is almost uniquely used by UE4 and its previous versions. The developers at Epic went one step further and integrated the *UnrealEd* directly into the engine's runtime. Having the benefit of total access to the systems and data structures, it simplifies the situation if having multiple representations of the same objects even more. Furthermore it is faster than other solutions when the game is run from

withing the editor. That is due to the fact the part of the game is already running when the editor is integrated deeply into the engine's runtime. But with the power that approach established comes a great drawback. Because the tools are tightly coupled with the runtime, as soon as the game crashes or runs into an undefined state it also impacts the stability of the tools and sometimes even crashes them too, leading to a decrease of productivity and iteration times.

3.6.2 Asset conditioning pipeline

Being one of the tools that is included in almost every engine, the asset conditioning pipeline is responsible for formating the assets into a format that is better suited for the engine or a specific platform. Assets are normally generated using digital content generation (DCC) tools, including well-known solutions like Maya (3D modeling), Photoshop (Textures) or Audacity (Sound and Audio). The data that is created withing the DCC tool is then exported into a intermediate format which often needs further processing before it can be sent to the game engine. The additional work is needed because the formats are often verbose and the asset pipeline can apply optimizations that are helping the performance of the runtime engine. Some of these techniques convert intermediate formats into binary ones, allowing them to be parsed faster at runtime, or group together assets with similar properties to reduce the size of files that have to be read. If the engine works on multiple platforms it is also often the job of the asset pipeline to convert the files to formats better suiting the target platform of the current build.

4 Rust

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

4.1 Current state

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

4.2 Rust ecosystem

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

4.3 Concepts

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there

no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

4.3.1 Borrow checker

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

4.3.2 Traits

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

4.3.3 Hygienic macros

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

4.4 Pitfalls

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there

no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5 Subsystem implementation

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.1 Spark engine architecture

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.2 Development environment

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.3 Rendering framework

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there

no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.4 Memory Management

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.4.1 API

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.4.2 C++ implementation

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.4.3 Rust implementation

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there

no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.5 Containers

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.5.1 API

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.5.2 C++ implementation

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.5.3 Rust implementation

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there

no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.6 Entity Component System

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.6.1 API

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.6.2 C++ implementation

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.6.3 Rust implementation

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there

no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

6 Submodule benchmarks

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

6.1 Benchmark setup

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

6.1.1 Cross-language benchmark process

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

6.1.2 Environment

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest

gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

6.1.3 Scenarios

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

6.2 Results

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

6.2.1 Memory Management

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

6.2.2 Container

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest

gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

6.2.3 Entity Component System

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

6.3 Conclusion & Discussion

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

7 Conclusion

Bibliography

[1] C. Bovet, Daniel, *Understanding the Linux Kernel*, 3rd ed. OReilley, 2005. [Online]. Available: https://www.safaribooksonline.com/library/view/understanding-the-linux/0596005652/index.html

[2] J. Gregory, *Game Engine Architecture*, 2nd ed. 6000 Broken Sound Parkway NW: CRC Press, 2014.

[3] B. Stroustrup, *The C++ Programming Language*, 4th ed. Upper Saddle River, NJ: Addison-Wesley, 2013.

[4] O. Blandy, Jim, *Programming Rust - Fast, Safe Systems Development*, 1st ed. OReilley, 2017.

[5] Chapter-wise on usage, *Game Engine Gems 3*, 4th ed. 6000 Broken Sound Parkway NW: CRC Press, 2016.

[6] D. Portisch, “Multitasking using a job sytem with fibers,” Master’s thesis, Fachhochschule Technikum Wien, Höchstädtplatz 5, 1200 Wien, 2017.

List of Figures

List of Tables

List of Code

List of Abbreviations

ECS Entity Component System

GB Gigabyte

RAM Random-Access-Memory

GPU graphics processing unit

FPS first person shooter

RTS real-time strategy

UE4 Unreal Engine 4

UI user interface

UBT Unreal Build Tool

UHT Unreal Header Tool

OpenGEX Open Game Engine Exchange Format

OpenDDL Open Data Description Language

IDE integrated development environment

API application programming interface

SDK software development kit

FPS frames per second

OS operating system

LOD level of detail

DCC digital content generation