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: 1610585007

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

Wien, May 6, 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, May 6, 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 UE4.

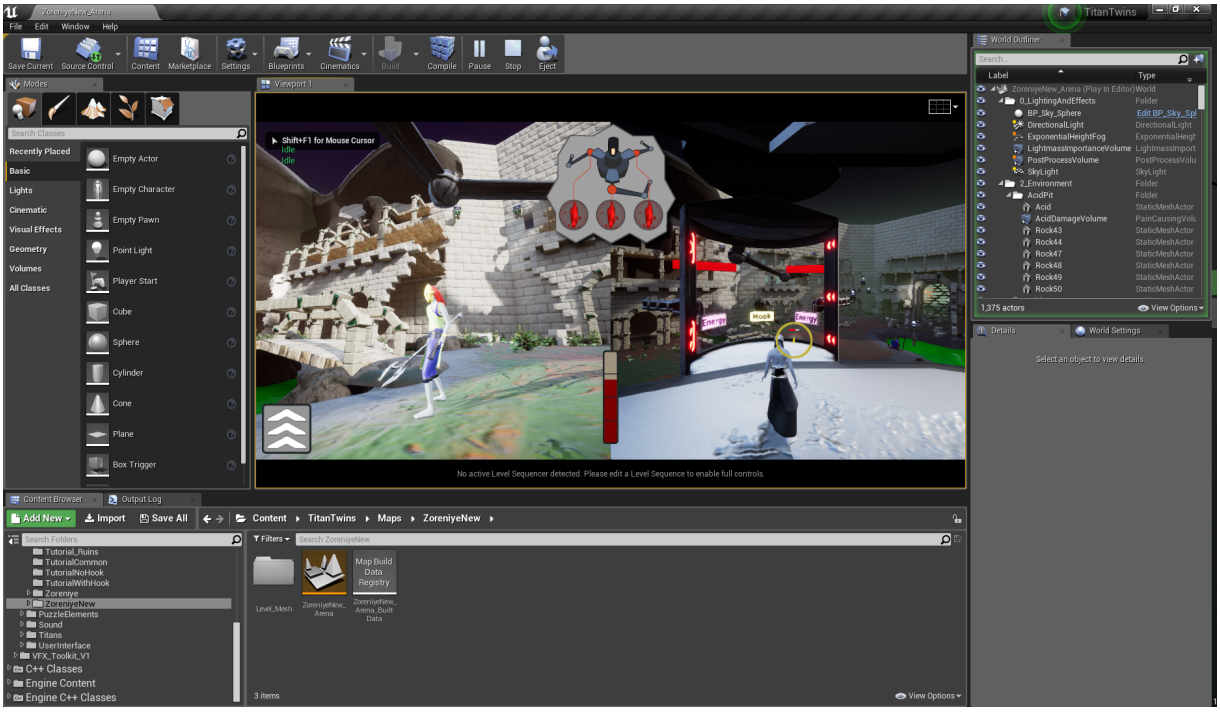


Figure 1: The editor shipped with UE4

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.

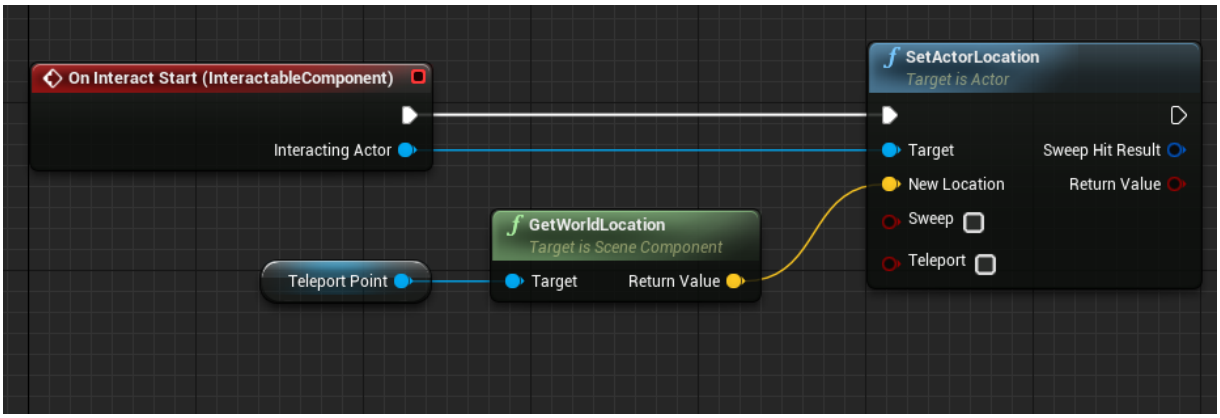


Figure 2: A very simple example that shows how the blueprint visual scripting looks like in UE4

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.

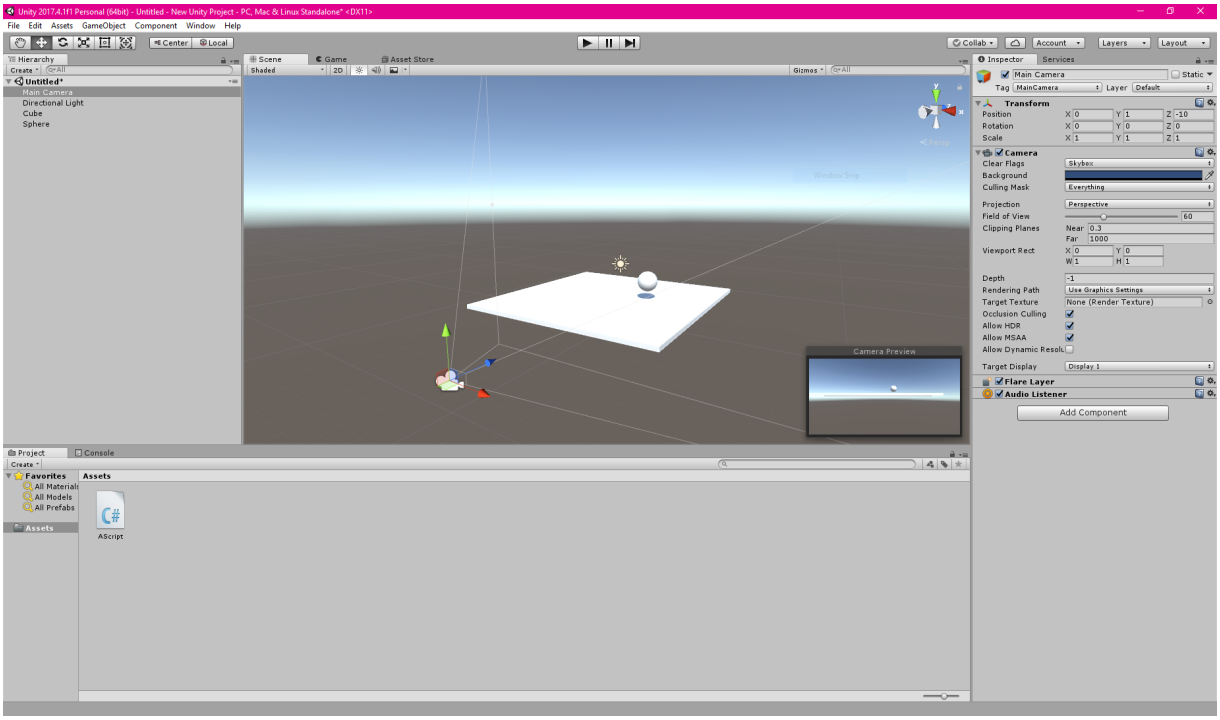


Figure 3: The default layout of the editor shipped with Unity

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.

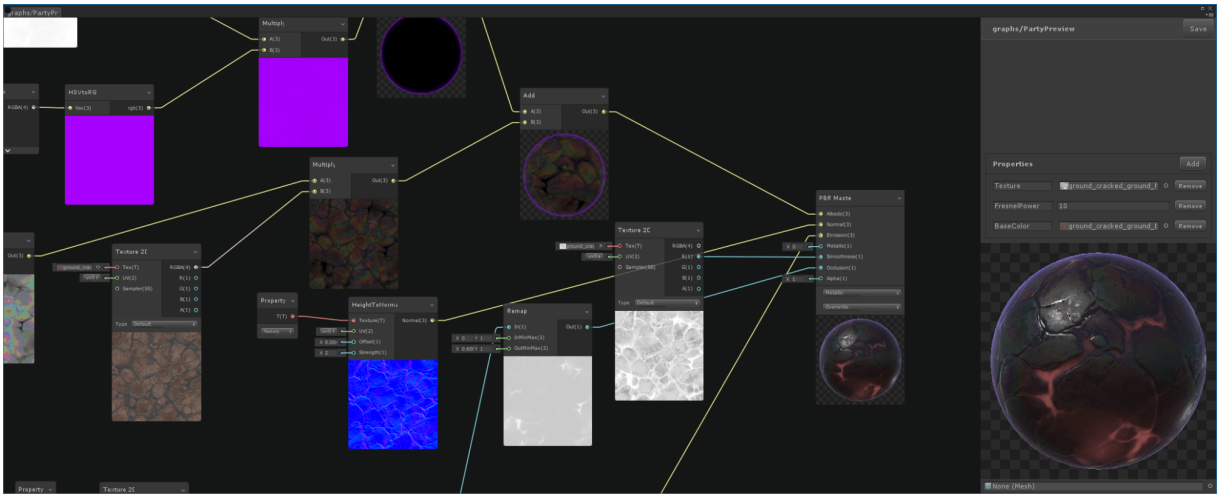


Figure 4: The new visual shader graph editor that ships with Unity 2018.1

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.

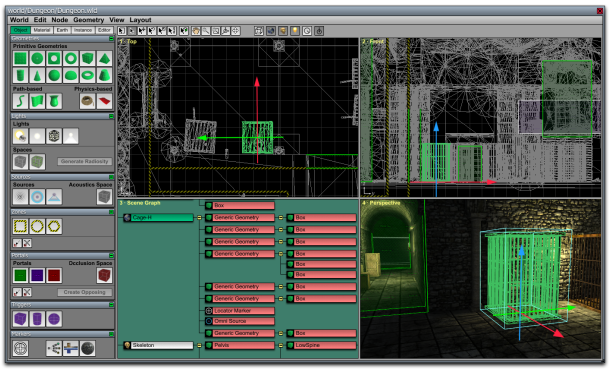


Figure 5: The world editor plugin used to edit worlds in the Tombstone engine

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

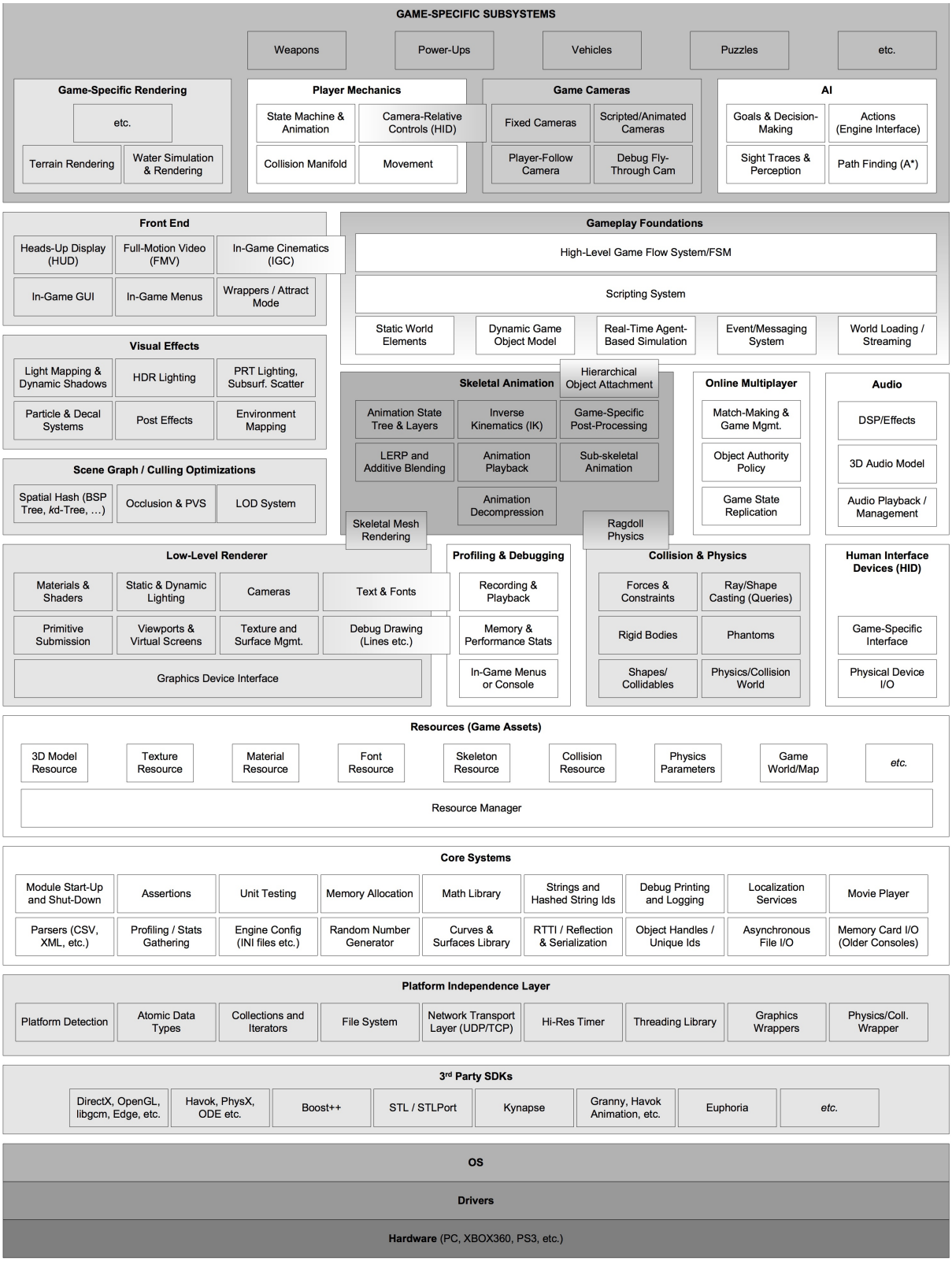


Figure 6: Illustration showing common modules grouped into distinct layers of a large scale engine solution

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 5.3. 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 5.5.

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

With the rise of multi-core systems, game engines started to implement techniques for using them efficiently. Some of the tasks an engine has to process are dependent upon each other but there are systems that can be run in parallel. To improve the performance of modern engines, the architecture's design focused more and more onto parallelism and multi-threading. Because game engines often execute routines many times for data of the same kind, one approach for a better utilization of the machine's cores was called *Fork and Join*. This approach was inspired by the field of divide-and-conquer algorithms and works with the idea of distributing the workload of a task to multiple processing units or threads. A thread is described as a sequential program hosted by an operating system process. All threads of a single process share the same memory

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

space and run in an asynchronous manner, speaking they can run at different speeds and can get interrupt at any time. Each thread that is created by its host program is then scheduled for execution on a dedicated processing unit of the machine's hardware. After every thread has finished, the data they were processing is merged together by the main thread. With this approach an engine can speedup the processing of big data chunks by processing them in parallel. While the previous approach is speeding up the processing of data at specific points, the following one tries to enhance the runtime speed of several engine systems throughout the whole execution time. It was already introduced that an engine is often separated into subsystems and that some of them can operate independently from others. That fact allows for an optimization where each subsystems runs on an own thread. The systems now can run in parallel, being managed by a master thread that handles synchronization points between the different systems. This allows machines with more cores to better utilize them and run the systems in parallel, leading to better performance. Although the general performance increased with such an approach there is an overhead that occurs when a thread has no work to do. If for example the animation system is currently idle and has nothing to to, its thread is suspended and sleeps until data arrives for being processed. This suspension time can not be used by any other system and is therefore lost. To overcome that disadvantage engines started to implement job systems. Using such a systems requires the engine's workload to be divided into more granular routines which are then often called *jobs* or *tasks*. A job can be though of as a package containing some data and the code that shall be run to work with that data. Most of the time these jobs are rather independent from each other, although there are job systems that can also handle jobs with certain dependencies. These jobs are then submitted to a queue by the main thread where they stay until being picked up by one of several worker threads. The worker threads are responsible for fetching a job from the queue and executing it. This approach has the advantages of better utilizing the machine's cores while providing an abstraction for the complicated multi-threaded parts of the engine. Together with more advanced techniques *(work stealing, multiple queues, ec.)* job systems increase the performance of a game engine even more by better utilizing the underlying hardware.

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, that can be seen in Figure 7a, 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, 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, which can be seen in Figure 7b, 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 - a new system level programming language

To better understand where Rust, a new system level programming language, is placed among others of its kind, it is necessary to understand the difficulties that can arise by using them. Described by Bjarne Stroustrup, the creator of C++, in his book *'The C++ programming language'*, one purpose of a programming language is to provide "a vehicle for the programmer to specify actions to be executed by the machine". That is even more true for a language that is used to create software that needs to utilize the underlying hardware very well to run at a good performance level. C++, and now Rust, are languages that provide the tools a programmer needs to write code that runs well on the hardware. But because that tools grant great power and control, any error done when implementing an application can be costly and have severe consequences. And there are two problems that, based on history and experience, prove quite difficult to solve. That problems are writing *secure* and/or *multi-threaded* code. Secure code is often related to memory management and with languages that allow for manually controlling memory operations, the complexity of this issue increases. Consequences of these difficulties are security vulnerabilities and exploits, that also affected bigger companies without mentioning any names. Although multi-threaded code does not that often cause security holes, the complexity, introduced by its parallel and asynchronous fashion, is the reason for errors that are hard to identify or even reproduce.

Rust tries to solve exactly this issues while claiming to maintain performance similar to the one of C or C++. Rust is developed by Mozilla and an open source community. It allows the programmer to manage the memory used by the application manually and tries to uphold a close relationship between language operations and the machine's hardware. While trying to solve problems that can occur in code written in C++, Rust shares several common principles with it. One of them being the ambition Bjarne Stroustrup has expressed for C++ in his paper *"Abstraction and the C++ Machine Model"*:

> In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better.

While following this and other principles shared with C++, Rust adds own standards it wants to uphold, including memory safety or simplified and trustworthy concurrency. To meet these promises, Rust relies heavily on compile-time checks that collaborate well with its static type system. In statically typed languages, the types of variables are checked at compile-time by

a part of the compiler called the type-checker. Because those checks are not performed at the program's runtime, these languages are said to be *statically type checked*. This comes with the advantage of allowing certain runtime checks to be omitted, which removes execution overhead and reduces the binary size. To prevent a common misunderstand when talking about statically typed languages, it has to be mentioned that *type inference* does not prevent static type checking. Type inference is a language feature, that allows the programmer to omit any specific type when declaring a variable. The type is chosen by the compiler based on the context within a variable was declared and by knowing the type of the value that was assigned to it.

Combined with its novel ownership system, that allows the definition of lifetimes for used values, it ensures memory safety without the need of a garbage collector. A garbage collector is a system that tracks memory allocations at runtime and manages their lifetime instead of the programmer. It is commonly used in higher level languages such as Java or C#, but the safety comes with a cost in performance. When ownership shall be transfered from one owner to the new one, Rust uses the concept of *moving* and defines *borrows*, that allow temporary usage of values without affecting their ownership. These techniques, which are later described in more details, build the basis for the memory safety in Rust. They also provide the foundation Rust's concurrency model is built upon. Again, by using compile-time checks, Rust is able to detect certain problems related to multi-threaded code before the program is run once.

This section continues with a description of Rust's current state and its ecosystem, followed a detailed description of the concepts that defined the language. They are explained and illustrated with code examples to built a better understanding. The shown code samples are compared to corresponding ones in C++, to show the difference between those languages.

4.1 Language's current state

With version 1.0 being released in 2014, Rust is a rather young programming language. The current version is 1.25.0, which was shipped on March 29, 2018. Examining the release notes and corresponding dates, it can be observed that about every six weeks a new major version upgrade is released. Version upgrades include request for changes (RFCs) issued by the community or by dedicated working groups. Another source of feedback for the language comes from the developers of *Servo*, a new web browser engine entirely developed in Rust, which is developed by Mozilla. Servo powers the newest version of the Firefox browser and serves as a real-world test of Rust. Beside the concepts already mentioned above, Rust is embedded into an ecosystem that includes many tools that simplify the live of a developer. The parts of the ecosystem are described in the following section.

4.2 Rust ecosystem

During the development process of Rust several tools, alongside the Rust compiler *rustc*, were built to enhance Rust's development process. Together with libraries, or *crates*, created and distributed by Rust developers all around the globe, they form the Rust ecosystem. Two of these tools, every Rust developer will use at least once, are called *Rustup* and *Cargo*.

4.2.1 Rustc - The Rust compiler

The Rust compiler went through many iteration steps until it reached its current state. The first version of the Rust compiler was written in OCaml, a different programming language using a functional, imperative and object-oriented style [OCAML]. It was the purpose of that compiler to compile a state of Rust that is capable of build a good compiler on its own, paving the way to a self-hosted compiler. A compiler that is self-hosted is written in the same language it normally parses. After Rust has reached the quality needed to serve that purpose, the legacy OCaml compiler was deleted from the language repository, which is proved by traveling back in time in the Rust GitHub repository commit history. At https://github.com/rust-lang/rust/commit/6997adf76342b7a6fe03c4bc370ce5fc5082a869 it can be seen that the OCaml compiler part was removed. Beside the fact that the Rust compiler is self-hosted another interesting fact is how the compilation process works and what tools are included withing it. While *rustc* serves as the compiler front-end, low-level virtual machine (LLVM) is used in the compilation process and as a back-end. LLVM is an open source tool for building programming languages and compilers. It includes many different tools and is a fundamental part of the compilation process of Rust programs.

Compiler design & LLVM

Because LLVM is such a fundamental part of the Rust compiler, this section is going to shortly describe how a classical compiler is designed and how the LLVM project works. The most popular design of a static compiler separates itself into three major components: the front-end, the optimizer and the back-end. It is the front-end's job to parse the code written in the source language, building an Abstract Syntax Tree (AST) out of it and reporting errors encountered during the processing. The optimizer is executed after the front-end finishes and applies transformations based an rules to enhance the performance of the code fed to it. After being optimized the code is then passed to the back-end, which is responsible to emit instructions matching the target architecture. Due to that reason the back-end can be also called *code generator* in other literature. An illustration of such a three-pass-compiler approach can be seen in Figure 8.

The biggest benefit from such an approach is that in theory it is simple to support different programming languages or machine architectures. If another language should be supported, this can be achieved by implementing a front-end for it while the optimizer and all existing back-ends just work without any big changes. The same applies for new target architectures, each

Figure 8: The stages of a three-pass compiler showing the way from source code to machine code

requiring a new back-end, that supports every existing front-end out of the box. A good example for an open source compiler that supports several front-ends and back-ends is the GCC. But although the three-pass compiler design is well documented in various literature, in practice it is rather hard to uphold the separation all the time and several well-known open source projects did not do so. Due to that fact, LLVM was created to unify and simplify the process of building languages and compilers. It shall also enhance the development of already existing languages. [2]

One of the most important parts of LLVM is its intermediate representation (IR). The LLVM IR is how code is represented in the compiler. The purpose of the IR is to allow the compiler's optimizer to run mid-level transformations and analyses, while being a first class language with well-defined semantics on its own. The IR in LLVM serves as a perfect working environment for an optimizer, which is not constrained to any language or target specification. Beside the IR, LLVM uses a classical three-pass design. The front-end parsed the source code and generates LLVM IR from it, which is then handed to the optimizer for running several analysis and transformation passes. At the end all IR code is passed to the back-end, where native machine code is generated from it. The implementation of LLVM's three-pass design can be seen in Figure 9.

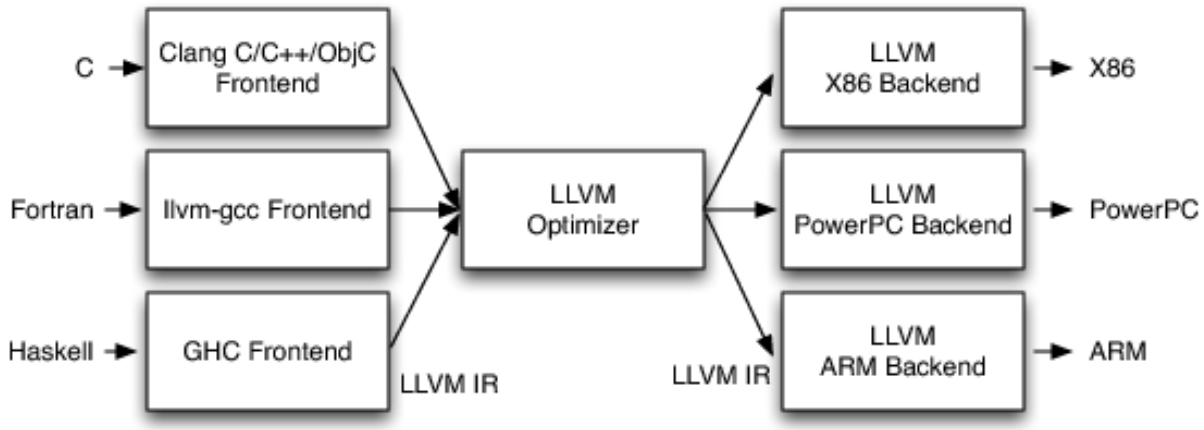


Figure 9: LLVM's implementation of the classical three-pass compiler design

By using LLVM during the compilation process, Rust gets all the benefits from already implemented optimizations in the LLVM project. It is also responsible for the cross-platform abilities of *rustc*, that is able to build binaries for different platforms specified by the compiler's toolchain.

4.2.2 Rustup

Rustup is a tool that helps installing the Rust toolchain. It allows the installation of different configurations and makes it possible to easily switch between several states of the Rust compiler. Beside being able to switch between nightly, beta and stable compiler versions, it is responsible for keeping them updated. It is currently able to run in all platforms, that are supported by Rust. With *rustup* it is possible to install and manage several different Rust toolchains, managing them by a single set of tools. A toolchain in that context describes a single installation of *rustc*.

Another important task of *rustup* is the possibility to install additional targets for cross-compilation. Cross-compilation describes the process of generating compilation binaries for one machine (*host platform*) on another, different one (*build platform*). Because *rustup* only installs the standard library binaries for the current platform, it is necessary to download them for the *host platform* as well if the binaries shall be cross-compiled.

4.2.3 Cargo

Cargo is the package manager of the Rust programming language. Some of its features include downloading project dependencies, building the application, packaging crates and uploading them for distribution. It can be compared to package managers from other languages such as NPM from NodeJs or Nuget and C# that are also capable of handling project dependencies in an automated fashion. But what makes Cargo rather unique and powerful are the capabilities it provides beside dependency management. It incorporates a build environment for Rust applications that is powerful and enhances the development process. Beside being able to invoke *rustc* or a custom build script/tool, it is possible to test and benchmark a Rust application without any external tools. While other languages need to integrate that tools themselves , which often requires some amount of extra work, Cargo provides this important utilities out of the box.

Cargo is also the tool used to upload packages, or crates, to *crates.io*, the Rust community;s package registry. Crates.io serves as a hub for many different libraries that can be used by any Rust application by simply depending onto them in the application's configuration file.

4.3 Basic syntax

Rust's syntax is quite similar to the one of C and even C++ in some parts. This section is going to showcase the its basic syntax based on examples to better understand the following concepts and code samples. It will also introduce some basics that are part of Rust's safety guarantees.

Variables, types & mutability

Because Rust is a statically typed programming language, every variable needs to have a distinct type specified at compile-time. As already mentioned the ability of type inference is not

contradictory to statically typing. In Rust, the type of a variable can be annotated explicitly or inferred by the compiler. A variable declaration beginning with the `let` keyword do not require the programmer to specify the type, although it is not restricting an explicit declaration. Other variable declarations using `const` or `static` require a type to be annotated.

```
1   ///
2   /// Several variable bindings where both versions, type inference and type
        annotations, are showcased.
3   /// The third binding will emit a warning because Rust sees that the literal
        exceeds the range of a 'u8'
4   ///
5   let inferred  = 10;
6   let annotated: u32   = 42;
7   let annotated_second: u8 = 1024;
8
9   ///
10  /// Variable bindings with different mutability annotations
11  ///
12  let immutable_var = 10;
13  // immutable_var = 11 --> compilation error
14  let mut mutable_var = 41;
15  mutable_var = 42;
16
17  ///
18  /// Rust does not allow implicit conversions between numerical types,
19  /// hence they have to be cast explicitly
20  ///
21  let small_int: u32 = 100;
22  let bigger_int: u64 = small_int as u64;
```

Code 2: Variable bindings and mutability declarations in Rust

As it can be seen in listing 2, Rust defaults all variable bindings to be immutable. Contradictory to C++, where immutable variables have to be marked as `const`, Rust requires the programmer to specify which variables are mutable by using the `mut` keyword. Another situation where Rust prefers explicitness over implicitness is when casting between different types. Where in C++, numerical types can be implicitly converted to each other, Rust does not include such capabilities on purpose. If a type conversion shall be performed the programmer has to cast one type to the other by using either the `as` keyword, for a safe cast, or the `transmute` function, for an unsafe cast working quite similar to the `reinterpret_cast` in C++.

Pattern matching

Another syntactical and functionally interesting feature of Rust are its pattern matching structures. Rust provides the `match` keyword, that works quite similar to a `switch` statement in C++, but is more powerful.

```
1
2   ///
3   /// 1) A match statement with range-based matches
4   ///
5   let arr = [1, 2, 3, 4, 5, 6];
6
7   match arr {
8      [2, ..]  => { println!("Match␣if␣value␣at␣[0]␣is␣2") },
9      [1 .. 4] => { println!("Starting␣with␣1␣and␣ending␣with␣4") },
10     _ => { println!("Needed␣to␣handle␣every␣other␣case") }
11  };
12
13  ///
14  /// 2) A match statement with conditional patterns that returns from every arm
15  ///
16  let x = 10;
17
18  let res = match x {
19     1 | 9 | 10 => { println!("Match␣if␣x␣is␣either␣1␣or␣9"); true },
20     _ => { println!("Match␣if␣x␣is␣10"); true},
21  };
22
23  ///
24  /// 3) A match statement using destructuring
25  ///
26
27  struct Data {
28     id: u32,
29     amount: u64,
30  }
31
32  let workload: Data = Data { id: 1, amount: 10 };
33
34  match workload {
35     Data { id: bind_one, amount: bind_two } => println!("Processing␣id:␣{}␣
           with␣amount:␣{}", bind_one, bind_two),
36  }
```

Code 3: Match statement in Rust, allowing for complex pattern matching in each arm

Listing 3 shows several use cases of the match statement. The first match showcases the ability of complex patterns where the arms match only arrays that values are in a specific order. These complex patterns can be combined with conditional expressions, shown in the second match, making a pattern matching a powerful tool. In the third match, the expression of the first arm uses *destructuring* in order to bind the fields of the structure to variables, that can be used in the body of that arm.

4.4 Ownership system

Regarding ownership Rust promises that the decision over a value's lifetime is up to the user and that the program, in spite of the given freedom, will never use a pointer to an already freed object (dangling pointer). While C++ allows for flexible lifetime decisions it cannot guarantee that the program does never hit a dangling pointer because it's the user's responsibility to ensure that no pointer to an already freed object is used. Some highlevel languages such as Java or C# solve the dangling pointer problem by introducing a garbage collector that controls when exactly objects are freed. This technique however also introduces an impact in performance. To cope with this problem and to avoid garbage collection Rust came up with its own concept of ownership. In C++ and other languages it is said that an object 'owns' some other value that it points to and the owner therefore has control over the value's lifetime. In Rust this implicit ownership rule was turned to an explicit one and the language enforces that a value just has a single owner that controls its lifetime. As soon as the owner is freed, or dropped in Rust terminology, the owned value is dropped too. One example of Rust's ownership model is its `Box<T>` type that is a pointer to a value of type *T* that is stored on the heap. When a *Box* goes out of scope and is dropped, it frees the space it has allocated too, avoiding an otherwise introduce memory leak. [**?**, Chapter 4. Ownership] Whereas C++ also has constructs, called smart pointers, to express ownership, Rust's model comes without the overhead of storing additional meta data at runtime. Smart pointers have to store data for knowing when it is valid to drop the resource. Instead of this runtime overhead, Rust's system has to advantage of avoiding this overhead at all by moving the ownership checks to the compilation stage.

4.4.1 Moves

In C++, when a user assigns a variable to another or passes it to a function by-value, a copy of the value is created. Rust has chosen a different approach and instead of copying values, they are moved. When a move occurs, the previous owner transfers ownership to the destination and gets uninitialized. The destination now controls the value's lifetime and becomes the new owner. Move semantics also exist in C++ and were introduced in the C++11 standard allowing the programmer to define move constructors and move assignment operators. [**?**] With the rule of using moves as the default behavior for assignment Rust can allow cheap assignments and keep the ownership clear as well. One trade-off the developer has to pay is that a copy now has to be requested explicitly. To do so in Rust one can implement either the `Copy` or `Clone` trait. What exactly a trait is and how they interact with other part of Rust is described in a later section. For now a trait can be though of as an extension, that can be implemented for other types, allowing them to be use by other code parts only knowing the trait's API. The first one describes its implementor as trivially copyable by a plain `memcpy` where the second option requires the caller to invoke the `clone` method that returns a new object. [**?**, Chapter 4. Ownership]

4.4.2 Borrows

Beside owning pointers, such as a `Box<T>`, Rust also defines non-owning pointer types that are called *references*. The major difference to owning-pointers is that a reference has no effect on the lifetime or ownership of the value it is pointing to. In Rust terminology the act of referencing a value is called *'borrowing'* and the compiler enforces that no reference outlives its referent. There are two kinds of references in Rust – *shared* and *mutable* ones. A shared reference allows to read the value but not modify it and there can be as many shared references as the programmer needs. A mutable reference however is allowed to be read and modified but no other reference is allowed to be active at the same time. This concept introduces the compile-time rule of either several readers or one writer which is essential to the memory safety of Rust. One thing that shall not stay unmentioned is a big difference between C++ and Rust references. While under the hood both are just addresses, Rust references are allowed to be reseated after initialization whereas C++ references just alias the object they have been initialized with and cannot be reassigned afterwards. [**?**, Chapter 5. References]

4.4.3 Lifetimes

With the knowledge on how Rust handles moves and references it is now possible to describe a more advanced feature of Rust's ownership model. To fulfill the promise of never using a dangling pointer in the entire program, Rust needs a way to tell, how long a value is valid or, in other words, alive. Although the Rust compiler is often able to deduce the lifetimes of objects or references from the context, there are some situations that require the programmer to opt-in. If such a situation occurs *lifetime parameters* in the form of `'a`, where `a` is describing a distinct lifetime. Lifetimes enable the compiler to reason about how values are used and allow it to reject code, that would produce a dangling reference. One example where explicit annotation of lifetime parameters can be necessary is when returned references from functions. Such an example can be seen in the following code listing.

```
1 fn get_element_from_slice<'a>(slice:␣&'a [u32], index: usize) -> &'a␣u32␣{
2 ␣␣␣&slice[index]
3 }
4
5 ///␣...␣Somewhere␣in␣the␣main␣function
6
7 let␣ref_to_element;
8 {
9 ␣␣␣let␣a_slice␣=␣[10,␣20,␣30,␣42,␣50];
10 ␣␣␣ref_to_element␣=␣get_element_from_slice(&a_slice);
11 }
12 assert_eq!(*ref_to_element,␣42);
```

Code 4: Returning a reference from a function, needing an explicit lifetime annotation. Compilation error because of a possible dangling reference.

As we can see, the function signature of `get_element_from_slice` is providing a specific lifetime parameter `<'a>`, that is then connected to the parameter and return value, describing that they both share the same lifetime. With that constraint setup, it is said, that the reference returned by the function is not allowed to live any longer than the slice passed to the function. If the compiler now parses the part of the code that calls the function and then later uses the returned reference after `a_slice` left the scope, the program would be rejected with an error and a hint that the slice the reference is pointing into is dropped while there was still a reference into it. The programmer is notified of this possible dangling reference at compile-time, which prevents application errors or undefined behavior at runtime. To solve the problem and make the code in Listing 4 compile, only one line has to be changed. By moving the `assert_eq!(*ref_to_element, 42)` two lines up into the scope, the reference returned does not live longer than the referee and the compiler will create a valid program.

Beside being needed for returned references explicit lifetime annotations are also necessary when creating custom types that hold references. Because the field of the type is still a reference, the Rust compiler needs to ensure its validity throughout the program the same way every non contained reference is checked. Due to that rule a struct that contains a reference to a value needs an explicit lifetime. The following listing displays how a simple example could look like.

```
1 struct<'a>␣SomeData␣{
2 ␣␣␣␣pub␣reference:␣&'a u32,
3 }
4
5 /// ...
6
7 let instance;
8 {
9     let x: u32 = 100;
10    instance = SomeData { reference: &x };
11 }
12 assert_eq!(*instance.reference, 100);
```

Code 5: Struct containing a reference to some value, needing a lifetime annotation to not create a dangling reference.

The code shown in Listing 5 does not compile. The interested reader may already guess that the compilation error is related to a lifetime issue, which is in fact the reason. Because `SomeData`'s field `reference` needs a lifetime parameter and the instance was initialized with a reference to x, the struct's lifetime `'a` is bound to the one of `x`. It is then following the rules of every other reference and is not allowed to outlive its referent. Again to solve the issue with the code above, the usage of the reference has to be moved into the scope and therefore into the lifetime of `x`.

4.5 Fearless concurrency

The ownership concepts described in the previous section are very important for every Rust program. Because concurrent code tends to get complex quite fast if the number of threads rises, Rust's ownership concepts help reducing the complexity. Another advantage is the avoidance of concurrent bugs, which are hard to reproduce and debug. Because Rust wants to make concurrent programming more accessible by programmers it provides several features that are built upon design choices already used in concurrent programs. They allow a programmer to chose what style of parallelism is best for the current task and does not enforce the usage of a specific implementation. The following section will describe how one can create threads in Rust and what possibilities for communication and shared state there are.

4.5.1 Threads & immutable shared memory

As almost every other system level programming language Rust offers an API to create and manage threads. It is integrated into the standard library by the `thread` module. When a programmer needs to create a new thread this can be done by calling the `spawn` method and providing the function or closure, an anonymous function, it shall execute. The invocation of that function create an operating system thread, having its own thread local stack to store values. The handle returned by that function is later needed to `join` the thread. Joining is a common term for waiting until execution has terminated. That is needed to ensure that every thread is finished before the main thread terminates, which is ending the program.

Because a thread often also needs data it will operate on, there needs to be a way to provide both, mutable and immutable data. While writing to memory locations in a concurrent fashion needs some extra work for synchronization, read only access is simpler to control. Rust offers several abstractions to model a situation of shared ownership by providing reference counted types such as `Rc` or `Arc`. These special types are necessary due to the ownership concepts already introduced where a value can only have one owner at a time. But because moving and simple references do not work or cannot be checked for validity, reference counting is the way Rust implemented for shared ownership. So if a resource has to be shared among threads, it often is wrapped into an `Arc`, standing for atomic reference counted. Whenever another thread needs an instance of the wrapped data, the whole `Arc` is cloned, bumping its reference count by one. If all threads terminated and the reference count drops below one, it is dropped. These behavior is similar to smart pointer types, such as `shared_ptr` or `unique_ptr`, used in C++. This also prevents common concurrency bugs because the data inside an `Arc` is immutable. How resources are shared in a mutable way is described in the following section.

4.5.2 Mutable shared state

Multiple concurrent threads working on a mutable resource is a situation, that requires fine-grained access control to avoid critical bugs. Rust provides a system programmer with familiar concepts naming mutexes, conditional variables and atomics. These, so called *synchronization primitives*, allow the programmer to synchronize the access of a mutable resource. In order to show some differences between the Rust approach and the one of other languages the *mutex* primitive is described in more detail.

Mutex

A mutex is a synchronization primitive that guards resource or section of code from being accessed by more than one thread. The section that is protected by a mutex, or a lock in general, is referred to as a *critical section*. Beside being used for thread synchronization, mutexes can be used for inter process communication (IPC) by using their named variants. A named mutex is visible across all processes of the OS allowing them to communicate by changing its state. Using such a primitive in a language is nothing new and C++ also has an implementation in its standard library. A basic example of a C++ mutex can be seen in Listing 6 below. The code sample is simplified on purpose to show the usage of a mutex, though the global variables and meaningless mutation function.

```
1 std::mutex g_resource_mutex;
2 std::vector<int> g_shared_resource;
3
4 void mutate_shared_state()
5 {
6    g_resource_mutex.lock(); // Enter the critical section
7    g_shared_resource.push(42);
8    g_resource_mutex.unlock(); // Leave the critical section
9 }
10
11 int main()
12 {
13    std::thread thread_1(mutate_shared_state);
14    std::thread thread_2(mutate_shared_state);
15
16    thread_1.join();
17    thread_2.join();
18 }
```

Code 6: Usage of a std::mutex to guard a critical section in C++

As the code sample above shows the mutex has to be locked and unlocked explicitly by the programmer whenever a shared resource is mutated. That section is only allowed to be run by a single thread at a time and every other one that tries to acquire the lock will be suspended

and wait until the mutex is unlocked. Because a mutex is often strictly associated with an object or a resource, Rust came up with a slightly different API for it, again turning an already commonly used implicit pattern into an explicit one. In Rust, the mutex wraps the data it protects and it can only be retrieved through the acquired lock. Therefore there is know way that the lock acquisition can be forgotten when the mutable resource is accessed. Listing 7 shows how a mutex is created in Rust and how the data is retrieved by locking it.

```
1 let guarded_vec: Mutex<Vec<i32>> = Mutex::new(vec![]);
2
3 // Somewhere in the concurrent function, accessing guarded_vec
4 let mut lock_guard = guarded_vec.lock().unwrap();
5 lock_guard.push(42);
```

Code 7: Usage of a std::sync::Mutex to guard a shared resource in Rust

Contrary the `Mutex` in Rust is tightly bound to the resource it protects by wrapping it. Whenever `lock()` is called in the mutex,it returns a `MutexGuard` that is just a simple abstraction over a mutable reference to the wrapped type. Due to explicit implemented coercion traits on the mutex, it is possible to directly call methods of the wrapped type on the guard itself. A lock is then automatically released whenever the guard goes out of scope or if it is dropped manually, after access to the resource is not needed anymore.

4.6 Polymorphism

Polymorphism is a concept that is part of many programming languages for many years. In Rust it is implemented by *traits* and *generics*. While traits are a concept similar to interfaces or abstract base classes in C# or C++ and allow for run-time polymorphism, generics are similar to C++'s templates, hence allowing for polymorphic behavior at compile-time. The following two sections are going to show how traits and generics work and how they are used in Rust.

4.6.1 Traits

A trait can be described as a set of functionality or features that a type, that is implementing it, supports. Traits are a powerful object to describe the capabilities of a type and combined with generics they can also be used to restrict the types that can be used with generic functions.

```
1
2 trait ReadBuffer {
3     fn read(&self, len: usize) -> Vec<u8>;
4     fn valid(&self) -> bool;
5 }
6
```

```
7 struct FileBuffer {
8    buffer: [0; 100],
9 }
10
11 impl ReadBuffer for FileBuffer {
12    fn read(&self, len: usize) -> Vec<u8> {
13       // details omitted ...
14    }
15
16    fn valid(&self) -> bool {
17       // details omitted ...
18    }
19 }
```

Code 8: Example usage of a trait and a type implementing it

Listing 8 shows the trait `ReadBuffer` that exposes an interface to read values and to query for validity. The trait is then implemented for the distinct type `FileBuffer`, which implements the trait's methods to fulfill the purpose they describe. Now every instance of a `FileBuffer` provides the interface of the implemented trait. If another type would also implement the trait for itself, a function could take references to types implementing it while not needing to care what type exactly was passed to it. The code sample below shows a function that can work with any type that implements the specified trait. If a reference to any type implementing a trait is used it is called a *trait object*.

```
1 using ReadBuffer;
2
3 // 1) Calling read() on a trait object, every type that implements ReadBuffer
      (dynamic dispatch)
4 fn read_bytes(read_buffer: &ReadBuffer, length: usize) {
5    let content = read_buffer.read(length);
6 }
7
8 // 2) Callind read() on a distinct type implementing ReadBuffer (static dispatch)
9 let file_buffer: FileBuffer = FileBuffer::new(file);
10 let content = file_buffer.read(100);
```

Code 9: A function, taking a trait object

Listing 9 shows two ways to invoke the trait's method on a type implementing it. In section one a function is taking a `&ReaderBuffer`, called a *trait object*, as argument. Trait object is the common term for a reference to a trait type. Because when using this approach the type behind the reference cannot be known at compile-time, Rust needs to store some additional information in order to call the right function. Behind the scenes, when using a trait object, a *dynamic dispatch* is performed. To do a trait object is represented as a *fat pointer*, containing the data and a so called *virtual table pointer*, in memory. The virtual table pointer, or *vptr* for short, contains information about the type behind the object. The concept is rather similar the

one of C++ beside the fact that the *vptr* is not part of the struct itself. This allows even types, that would otherwise be too small to hold an additional pointer, to implement traits.

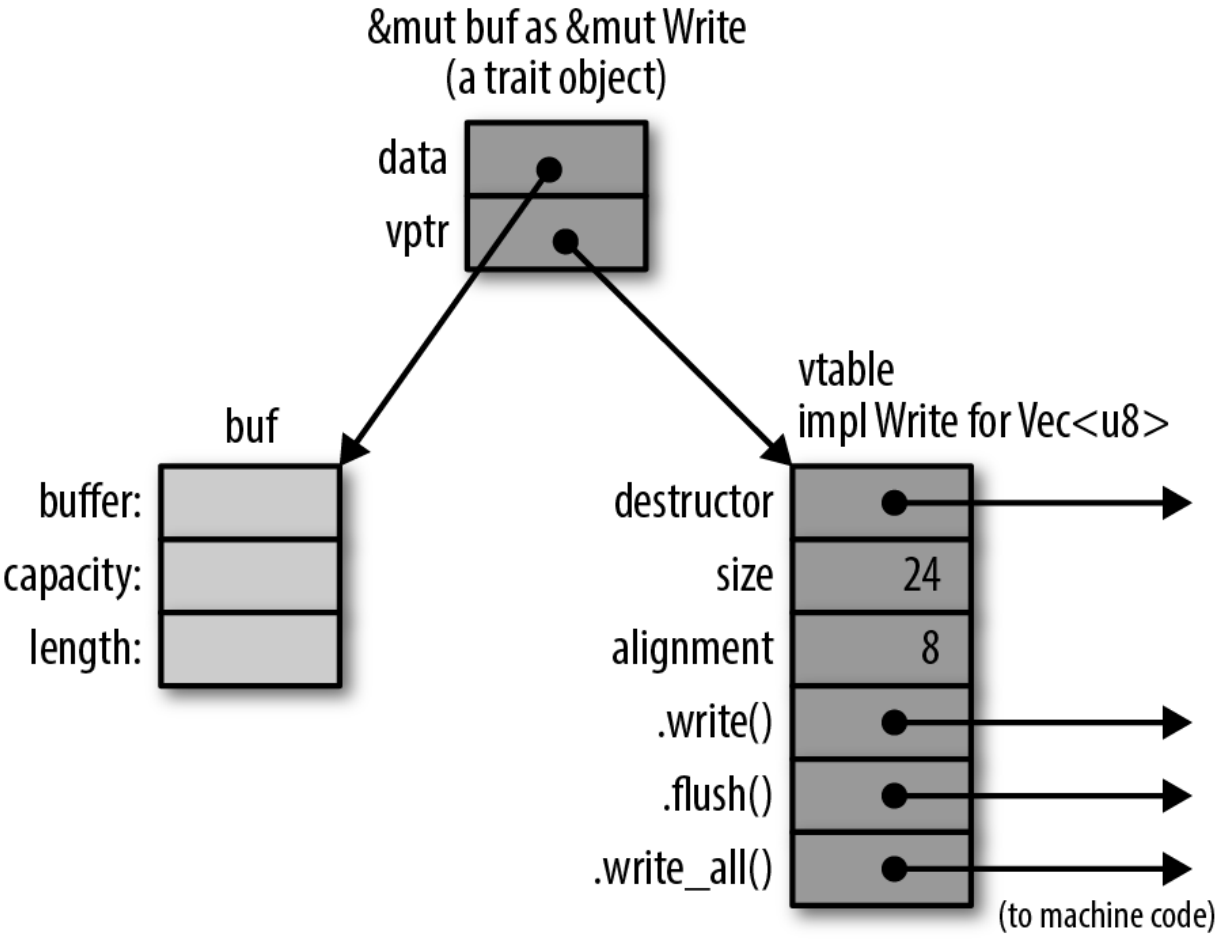


Figure 10: Internal layout of a trait object in Rust. Dark grey parts are part of the vtable and only created once at compile-time

The above figure visually shows the described layout of a trait object. The image uses a trait called `Write` instead of the previously used `ReadBuffer`. The dark gray parts mark the parts that are hidden form the programmer. The vtable is created once for each type at compile-time and whenever a trait method is called via a trait object a lookup into the vtable is performed to find the location of the distinct type's implementation. As the interested reader now might already has guessed, calling methods through trait objects introduce an overhead. That overhead can be avoided whenever the type of the trait's implementor is known upfront. If that is the case, as it is in Listing 9 in the second part of the code, the trait method can be called directly which is as fast as calling any other method.

4.6.2 Generics

Whereas traits and trait objects allow for runtime polymorphism, generics, similar to C++ templates, do the same at compile-time. By using *type parameters* functions and structs can be turned into generic versions, usable with different types. One of the most popular examples of generic code is `Vec<T>` that works with many different types. The `T` in the type definition can be replaced by others, such as `Vec<u8>` or `Vec<String>`, allowing the vector to be adapted to the programmer's needs. Whenever the compiler discovers a type parameter for the vector it has now previously seen, it copies all of the vector's code with the T being replaced by the distinct type. This process is called *Monomorphization* and also applies to generic functions.

```
1 using ReadBuffer;
2 using std::fmt::Debug;
3
4 // Generic function with simple trait bound
5 fn read_from_buffer<T: ReadBuffer>(buffer: &mut T, len: usize) {
6    let content = buffer.read(len);
7    // ...
8 }
9
10 // Generic function, more complex trait bound
11 fn read_from_buffer_debug<T: ReadBuffer + Debug>(buffer: &mut T, len: usize) {
12    let content = buffer.read(len);
13    // ...
14 }
```

Code 10: A generic function in Rust showcasing trait bounds

Listing 11 above shows how generic functions can be declared. A feature not mentioned until now is the possibility of adding *trait bounds* to a type parameter. A trait bound serves the purpose of requiring certain functionalities the types supplied to the generic function has to provide. Here can be a single trait or multiple once concatenated by a `+`. Whenever a type does not fulfill the bound it will raise a compiler error notifying the programmer about an unsatisfied trait boundary.

4.7 Crates & Modules

The last feature of Rust that is going to be described in this chapter is the way Rust handles dependencies and structures code, both between different projects and in a single one. When coming from a C++ background the common way to structure a project is to generate different header files and include them wherever they are needed. While techniques such as include guards, checking that no file is included more than once per compilation unit, reduce errors made by programmers, adding several libraries to a project can get cumbersome quite fast. In the C++ world, when a library shall be added to the project, a programmer needs to add

the include files first and then link against the necessary libraries. Contrary to this, Rust uses *crates*, for sharing code between projects, and *modules* for structuring code in one project.

Crates

A crate is described as the collection of everything related to a single Rust project containing all source code, tests, examples, tools and so on. A Rust project can be dependent onto other crates that can be either come from crates.io or the same project. Then, when a Rust project is built with cargo, it fetches all dependencies first, compiling them with rustc into *.rlib* files and then later statically linking them into the program. When using an external crate this is simply done by adding a `extern crate CRATE_NAME` directive to the top of the project's main file and bring the necessary module into scope with the `use` keyword.

Modules

Where crates are used to share and manage dependencies between projects, withing a single one there are *modules* to organize the code. A project can include multiple modules that can contain other ones or items, such as functions, types, variables and others. Modules can be compared to namespaces of other languages, serving as containers for logically grouped items. Modules are created with the `mod` keyword and can be nested into other ones. Additionally to the logical grouping, modules explicitly specify the visibility of contained items with either marking them as `pub` or not. Being marked as publicly visible allows the parts of the module to be used when the crate it contains is used, otherwise the non marked parts are just visible to the current module. Because modules can also be separated into several files, they are of great use when the size of a project grows and to design a meaningful API for the user of the crate.

4.8 Missing or unstable features

Although Rust already provides many useful and important features a system programming language needs, the author recognized two features, that would have been rather important for the implementation part of the thesis, are currently not implemented or in an unstable stage. This section is going to shortly describe these missing features. It has to be mentioned that the selection is solely subjective in the author's opinion.

4.8.1 Const generics

While coming from a C++ background the author missed a feature related to generic programming and that is often useful when working with templates in C++. Listing **??** shows how that feature looks like in the world of C++, where it is called *non-type template parameter*.

```
template<typename T, size_t N>
class FixedSizeArray {
public:
    T* data();

private:
    T m_array[N];
}

int main()
{
    FixedSizeArray<int, 100> hundredIntsArr;
    return 0;
}
```

Code 11: Showcasing template non-type parameters in C++

This above sample shows, how a template parameter, that is not a type, can be used to control the size of the array member. Non-type template parameters are a useful concept in generic programming when a variable parameter is known at compile-time. Unfortunately resembling such a behavior in Rust is currently not possible. There already is a RFC and work is done in that field but at the moment there is no workaround to implement a similar approach. A description of that RFC and a summary of related problems with the implementation can be found at GitHub under https://github.com/rust-lang/rfcs/blob/master/text/2000-const-generics.md.

4.8.2 Placement-new functionality

While the lack of const generics did not have a great impact onto the implementation of the submodules, the lack of a stable and working placement-new feature is rather bad. Placement construction describes a capability where it is possible for the compiler to construct aa value into a provided memory location. In C++, the `operator new` has a placement syntax, that allows the programmer to specify into which memory region the object shall be constructed.

```
struct Data { ... };

char memory_buffer[sizeof(Data)];
Data* data = new (&memory_buffer) Data;
```

Code 12: Using placement-new to construct an object in a pre-allocated memory location

In Listing 12 an instance of `Data` is constructed into a memory region on the stack by using the operator new with placement syntax. Its existence allows for efficient implementations of memory pools, garbage collectors and certain collection types.

At the beginning of the implementation period for this thesis, Rust had a nighlty feature for providing places where values could be constructed into. But during the first implementation of

the memory system (5.3) it was decided to stop development of it and remove all of its parts from the language. It is the author's opinion that the decision to remove the unstable version was a good one because beside of not being very ergonomic, the feature failed wo work in more cases than it was successful. Where it was capable of constructing a large array in-place it failed as soon as the array was the member of a struct. With this feature being removed from the language it was necessary to redesign the memory system and live with the drawbacks that introduced. One of the biggest disadvantages, that will be described in more detail in the next chapter, is the inability of allocating types that are too big for the stack and the overhead needed to move an object into the provided place instead of it being constructed there.

4.9 Conclusion

While being a rather young language, Rust already provides many features necessary for system programming. Combined with the design of *rustc* and LLVM as compiler back-end it is also useful for creating applications on multiple platforms. These two benefits are even more strengthened by the ecosystem and tool-suite that is build to supplement them. With Cargo and crates.io building applications and sharing them is a well-defined and documented process. With the basics of Rust and its ecosystem described, the next chapter will discuss the implementation details of the engine subsystems that were developed by the author during the process of this thesis.

5 Submodule implementations

In order to test the viability and usefulness of Rust in the field of game engine development, the author implemented three common submodules of an engine. Because measurements of the Rust implementations alone would not be enough to tell how well or badly Rust performs in certain disciplines, all modules were also implemented in C++, serves as a baseline of the Rust module benchmarks. The systems that were selected for implementation were based on their importance and impact onto the performance of an engine and its tool suite. The first module that was implemented is a memory management system featuring rather low-level custom allocators, debugging facilities and an abstraction type that allows for combining them for more ergonomic and flexible use. The second one includes three different types of containers usable by an engine or its tools, showing how Rust handles mid-level systems, shared across an engine's core system and tool applications. The last module that was implemented features an ECS, testing Rust's capabilities and ergonomics in a high-level system.

This chapter is going to describe the general architectural overview of the research engine called *Spark* [1]. It will also shortly give an insight into the development environment of the C++ and Rust projects and is then followed by detailed descriptions of the systems APIs. This will include both, general implementation details shared among the languages and differences between them. Basic code samples showcasing use cases of the systems will highlight advantages of the implementations and pitfalls that can occur when being used wrong.

5.1 Spark engine architecture

Following the concept of separated modules the current design implements each system as a standalone library. The basis of all three modules is built by a collection of common functionality grouped inside the *core* library. It includes mostly functions often used in all modules. On top of the *core* library, the other three engine systems are built, every single one being a standalone library without any dependencies to their siblings. This lack of dependency grants the benefit of being able to use every single one of these modules on their own. This concept is useful in the thesis's context to ensure that every system can be measured on it's own, without suffering any penalty when a depending module was implemented in a non optimal way. In a real world example the container system may be dependent upon the memory management to allow a programmer to choose the allocators used for a collection. And then again the ECS, because it

[1] Name is based on the fact that it can either turn into a wildfire spreading if Rust's performance meets expectations or stop burning if it shows that it is not a competitor of C++ in any field.

is more high-level and therefore it is allowed to depend onto lower level modules, could be built on top of the container library to use already implemented collections for holding components. But due to the need for independent measurements the author decided to follow the architecture that can be seen in Figure 11.

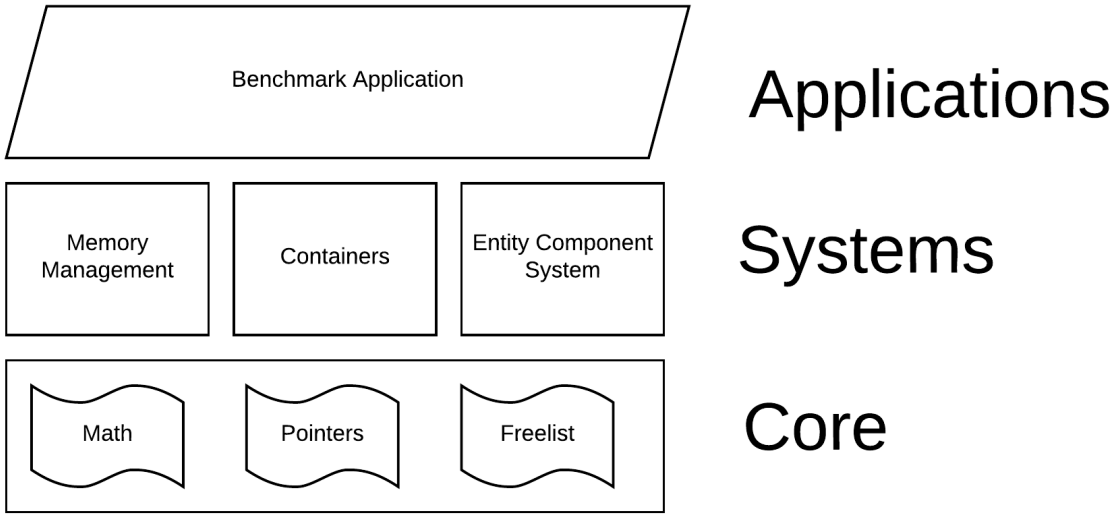


Figure 11: Layers of the module architecture implemented during this thesis. Higher layer can depend onto lower ones.

As it can be seen in the illustration above, the application used for benchmarking the different modules is built on top of them. This architecture is the same for both projects. Beside the shown components every project also contains a test suite, containing unit-tests for the features of all systems. How the tests are written and why the author decided to implement them additionally to the systems is described in the next section.

5.2 Development process

After the global architecture of the implementation was described this sections gives an insight into the development process of it. During the whole implementation period the author followed a test-driven approach for implementing features. Without going into much detail right now, it can be said that for every feature created there were several test cases developed beforehand describing the preferred usage and expected behavior of the API. Exact details of this are described in section 5.2, which also describes what frameworks were used and what benefit can be gained by implementing tests. Because Rust is rather young and so is the IDE world around it, the first part of this section will shortly explain which tools were used and how they compare to the ones used in C++, a mature language with a well-grown tooling landscape.

Spark++ - the C++ project

With Windows as the development OS the straight forward IDE choice for the C++ project was Visual Studio 2017. Because it is planned to develop the systems for multiple platforms in the future, *premake* was used to create IDE projects for the different platforms. *Premake* is a build configuration system used for generating project files for Visual Studio, XCode or other IDEs. When creating configuration files with premake, one uses *Lua*, a well-known scripting language in the gaming industry. An excerpt of the build configuration used for the C++ project can be seen in Listing 13.

```
1 project "MemorySystem"
2 kind "StaticLib"
3 language "C++"
4 targetname "mem-sys"
5 targetdir "build/mem_sys/%{cfg.buildcfg}"
6
7 files { "src/MemorySystem/**.h", "src/MemorySystem/**.cpp" }
8
9 links { "Core" }
10 includedirs { "src/Core/"}
11
12 filter "configurations:Debug"
13 symbols "On"
14 flags { "StaticRuntime" }
15
16 filter "configurations:Release"
17 symbols "Off"
18 flags { "StaticRuntime" }
```

Code 13: Part of the premake.lua file used to generate the Spark C++ project files

The above script shows how the project for the memory system is setup. The description defines what kind of binary out shall be created and the most common one can be an executable, a static or dynamic library. A project description also contains binary dependencies and needed include files, as well as a location for all files that are part of the generated project. By using filter it is also possible to define certain settings that only take effect in specific build configurations. In the context of this thesis premake was only used to generate the Visual Studio solution but simplifies the process of porting the systems to other platforms. While *premake* was only used for the C++ part of the implementation, the Rust one is based upon *Cargo* and its setup is described in the next section.

RustySpark - the Rust project

Before the author settled for Visual Studio Code as the IDE for Rust, there was an attempt on using Visual Studio too. But after a view hours of trying to work with the Rust plugin for

it, developed by parts of the Piston engine community, the author decided to switch to *VS Code*. That decision was based upon experiences of other Rust users and due to the author's experience with the Rust plugin. When using *VS Code* a programmer benefits from useful extensions for Rust development and the usage of a tool called *Racer*. Racer can be found at https://github.com/racer-rust/racer and is a utility for Rust code completion, boosting productivity and helps Rust newcomers. The build process was then totally done via *Cargo's* exposed functionality.

Test-driven development

Another important part of the development process, apart tooling that supports the programmer, was the decision to follow a test-driven approach. The term *test-driven development* in the context of the subsystem implementation describes the process of defining the wanted behavior in test cases and then implement functionality to pass these tests. For every feature several unit-tests were implemented to ensure that the behavior of small, separated parts of the software meet expectations. Beside the benefit of a structured development process another advantage of a well-maintained test-suite is the possibility to catch regressions introduced when altering or refactoring code. This safe-guard proved useful more than once during the development process and ensured that the behavior did not change when refactoring code or moving reusable functionality into the core library. The tests for the C++ project were implemented using the *Google Test Framework* that proved to be quite easy to integrate while providing direct integration into Visual Studio. The tests of the Rust project where implemented directly in the different module files and were executed via the `cargo test` command. The fact that Rust projects come with an unit-test runner out-of-the-box is a great benefit for productivity and maintainability of code.

5.3 Memory Management

The first implemented submodule that is going to be described is the memory management system. As it was already mentioned in section 3.2, an engine's memory management system is responsible to provide control over how resources are used at runtime. It tries to reach peek performance by assuming access patterns and providing different allocation strategies. Beside that is important to provide debugging facilities, especially when working with pointers and raw memory. This section will first describe the general API and from which parts the module is composed of. It will then highlight some interesting implementation details of both languages with code samples.

The design decisions for parts of the memory system are based on resources hosted at *Stefan Reinalter's* blog *Molecular Musings* [3] [4] [5] [6] [7] and on allocation strategies described in the book *Game Engine Architectures* [8]. The implementation for both languages share a common API and at least the same system members. On top of that the Rust implementation

features an additional abstraction layer to showcase how a safe interface for unsafe code can be built in Rust. The system is structured into the following parts:

allocator — facility to allocate and deallocate memory - providing different kinds of allocation policies

bounds checker — debugging facility to ensure the validity of an allocated memory area

memory realm — a memory allocation facilty that combines different debugging facilities with allocation policies - flexible and policy-based approach

5.3.1 Virtual memory

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.

5.3.2 Allocators

The different allocation strategies implemented serve as the module’s basis. In order to allow a user to store them in a polymorphic way, all of them share a common interface. This interface is implemented as a pure virtual class in C++ and as a trait in Rust. The API all of them share is shown in Listing 14. Although it showcases the C++ methods, the Rust trait only differs syntactically.

```
1 class AllocatorBase
2 {
3 public:
4 virtual void* Alloc(size_t size, size_t alignment, size_t offset) = 0;
5 virtual void Dealloc(void* memory) = 0;
6 virtual void Reset() = 0;
7 virtual size_t GetAllocationSize(void* memory) = 0;
8 virtual ~AllocatorBase() = 0;
9 };
```

Code 14: Common interface among all allocator implementation. Rust trait only differs syntactically from this C++ sample.

Every allocator can fulfill an allocation request, that provides size, alignment and an optional offset of the allocation. The term *alignment* is related to an address a certain object resides

at in memory. A certain value is said to be aligned to X, where X is a power of two and the address is a multiple of X. Normally alignment is not a concern to the everyday programmer but in certain situations and on specific hardware explicit alignment of values can yield better performance. The offset parameter describes how much memory the allocator shall provide **before** the aligned pointer. This is an important capability when an allocator is combined with a debugging facility that needs to store meta information in front of the aligned user pointer.

The first allocation policy, that is also the most simple one, is a *linear allocator*. It provides allocations with nearly zero overhead by not allowing the deallocation of single memory blocks. When using a linear allocator, the only possibility to free allocated blocks is to reset the whole buffer. Because of this fact it is not necessary to store the order of allocations and additional meta data needed when the internal pointer has to be rearranged on deallocations. Whenever an allocation is issue to the linear allocator, internally it ensures alignment and offset requirements and then bumps a pointer, pointing to the next free space. The only meta information that needs to be stored by the allocator is a 32-bit unsigned integer, storing the allocation's size. This is needed to properly work when used in the memory realm, which is described later. Figure 12 visualizes the change in a linear allocator's internal state after an allocation request was issued.

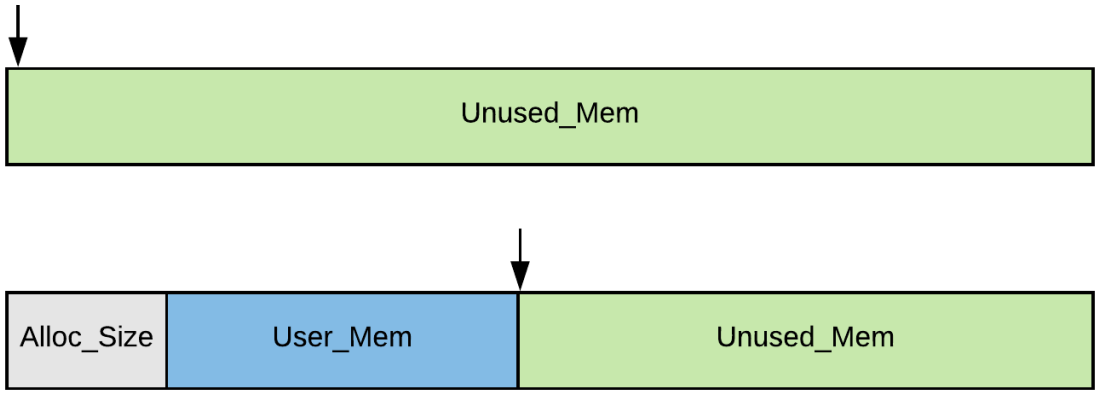


Figure 12: A linear allocator's internal state before and after an allocation request.

A common use case for such an allocation strategy is allocating per-frame temporary values that can be freed all at once with the start of the next frame.

If there is the need to free individual allocations another low overhead allocator can be used. A *stack-based allocator* issues allocation requests by advancing the internal pointer forward, but it also provides the possibility to free single allocations again. One constraint though, assuming the usage pattern for better performance, is the order in which allocations can be freed. A stack-based allocator only allows deallocations to happen in the reverse order of the allocations, meaning it adheres to a last-in-first-out (LIFO) approach. In order to set the internal pointer back to a previously done allocation, the meta data size grows and so the overhead for any allocation

is slightly larger than with a linear allocator. Figure 13 shows the internal state of a stack-based allocator and how it is changed after an allocation and a deallocation.

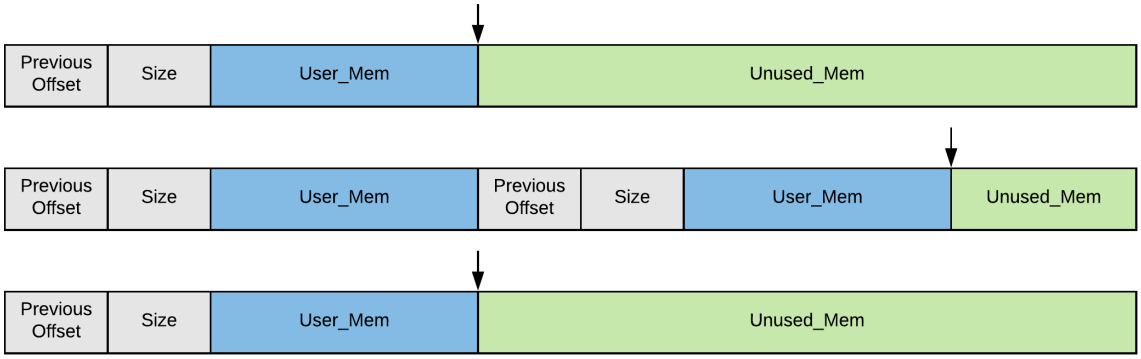


Figure 13: A stack-based allocator's internal state before and after an allocation request, followed by a deallocation of a single block.

Compared to Figure 12 it can be seen that the meta information in front of a user's memory block, from now on called the *allocation header*, has grown. Additionally to the memory block's size it also stores an offset to the allocator's beginning, again using an 32-bit unsigned integer for it. The allocation header now introduces an overhead of 8 byte. But with this initial bit of information the internal pointer can be reset to its location before the last allocation. It is also exactly that behavior which forbids deallocation in any other order than the LIFO one. If a block not coming from the last allocation would be freed, the allocator would create a hole and corrupt the internal pointer, preventing all other currently allocated blocks from being freed. The allocator's implementation features a second mode that runs a LIFO check on every allocation at the cost of an additional 4 bytes overhead. If the mode is activated by using conditional compilation flags, these 4 bytes are used to store an allocation id. On every deallocation the stored id has to match the allocator wide counter or else the LIFO rule was not respected. Common usage examples for this kind of allocation strategy include level loading, where certain values can be rolled back in reverse order after a level has finished.

An extension that can be built upon a stack-based strategy is the one of a *double-ended stack allocator*. The process of allocating memory does not change compared to the underlying stack policy. But the novel feature of this approach is that the allocator can grow from both sides. Memory can be requested from either the front- or the back-end, allowing for better utilization of the allocator's space. While the per-allocation overhead does not grow, one additional pointer has to be stored in the allocator to track the current state of the back-end. Figure 14 visualizes how such an allocator can grow and shrink from both sides and an interested reader can already guess what possible problems can occur and needs additional checks.

When growing from both side, the front pointer advances into the direction of the allocator's end while the back pointer moves to the start. The allocator has to be aware of the possible situation when the two pointer would overlap. This has to be avoided because if one advances the other

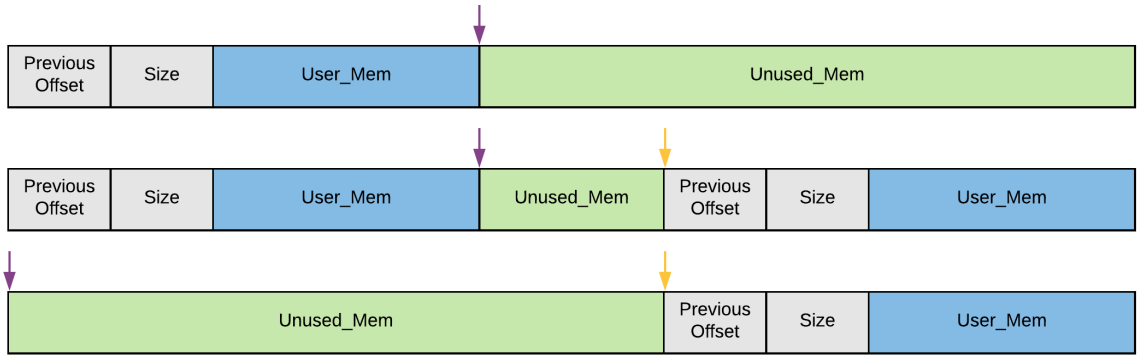


Figure 14: A double-ended stack-based allocator's internal state. Growing from both sides it needs to be ensured that the internal pointers to not overlap.

one, the integrity of issued allocation cannot be ensured. A double-ended stack-allocator is from great benefit when data exists in a compressed and decompressed state. Taking the example of a game world level, the compressed chunk could be loaded by one side, while the decompression allocates memory from the other side.

The last strategy implemented is a *pool allocator*, used for allocation of same-sized elements. Because allocations and deallocations could happen rather frequently when dealing with many small objects, such as particles or other volatile components, they should be fast. A pool allocator allows for allocations and deallocations in a magnitude of O(1). The big-O notation describes the magnitude of O(1) to not scale with the amount of elements stored in a collection. There is one well-known data-structure that satisfy this constraint and that is a linked list. But using a standard list would introduce both, an memory and management overhead because nodes need to allocated and all of them need to be stored somewhere. The technique used in the pool allocator is called an *intrusive linked list* and it comes with no overhead but a single additional pointer. This kind of list lives inside the pooled elements, but only in the ones currently not used. This requires the pooled elements to be at least as large as a `void*` on the current platform. Listing 15 showcases how such an intrusive linked list is created into a pre-allocated block of memory.

```
1 sp::core::FreeList::FreeList(void* memoryBegin, void* memoryEnd, size_t
      chunkSize)
2 : m_nextChunk(nullptr)
3 {
4 const ptrdiff_t memoryBlockLength = pointerUtil::pseudo_cast<char*>(memoryEnd,
      0) - pointerUtil::pseudo_cast<char*>(memoryBegin, 0);
5 const size_t elementCount = memoryBlockLength / chunkSize;
6
7 char* memory = pointerUtil::pseudo_cast<char*>(memoryBegin, 0);
8 m_nextChunk = pointerUtil::pseudo_cast<FreeList*>(memory, 0);
9 memory += chunkSize;
```

```
10
11 FreeList* current = m_nextChunk;
12 for (size_t i = 0; i < elementCount - 1; ++i)
13 {
14 current->m_nextChunk = pointerUtil::pseudo_cast<FreeList*>(memory, 0);
15 current = current->m_nextChunk;
16 memory += chunkSize;
17 }
18 current->m_nextChunk = nullptr;
19 }
```

Code 15: Constructor of the FreeList class in the C++ project. It showcases how the list is created in a pre-allocated memory region.

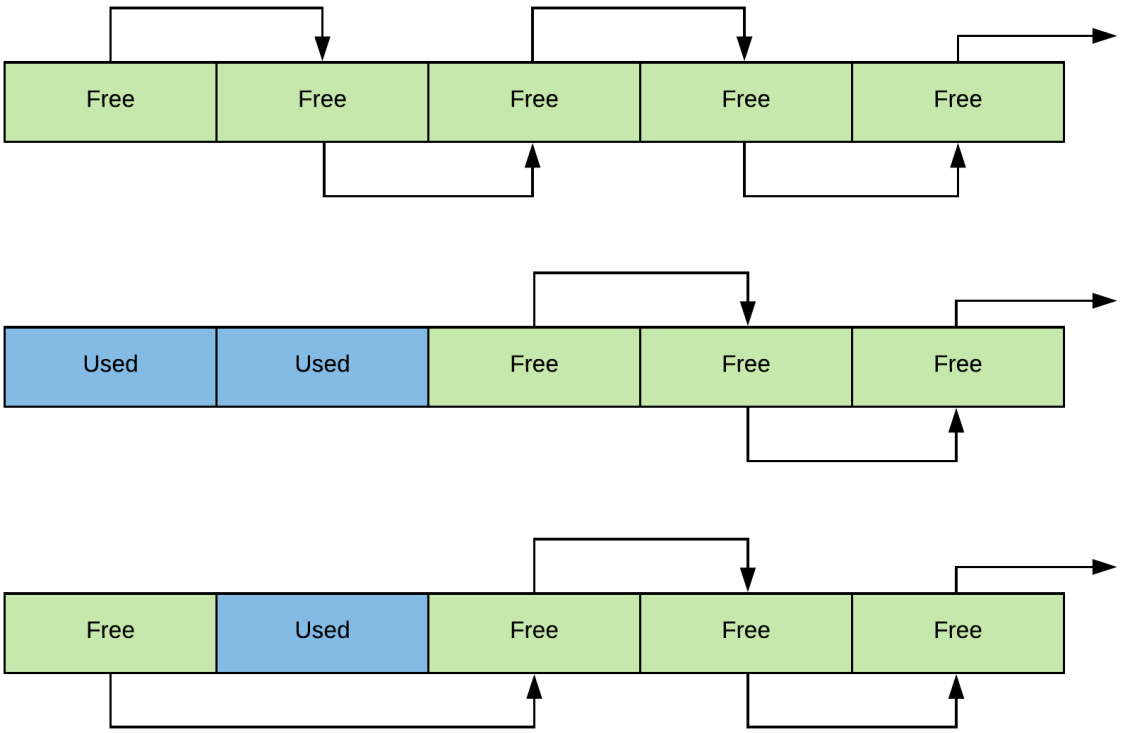


Figure 15: A pool allocator including the pointers of the freelist, which allows for O(1) allocation and deallocation of pooled elements.

With this no overhead data structure in place a user can request an element and deallocate it in O(1). One additional detail is related to how alignment and offset parameters work with the pool allocator. When creating it, the user has to provide the maximum alignment and maximum size of one element. With this information, the allocator computes the minimal chunk size of a single block to fulfill the alignment request. Other than with all previous allocators the offset has to be provided upon construction instead of per allocation. With the minimal chunk size

calculated and the offset defined constantly for every element the allocator can fulfill allocation requests with an alignment requirement up to the defined maximum. This can be done because if an element is aligned to N, it is also aligned to all power of twos smaller than N. To explain this with an example, if address A is aligned to 16 bytes, it is also aligned to 8, 4 and two and can therefore serve as the storage location for objects having such alignment requirements. Figure 15 shows how the freelist works when an allocation is done and how the pointers are bend when an object is returned.

Additionally to the benefits described above a pool allocator can be enhanced and turned into a growing version. In certain situations it can be useful to have the capability to grow as soon as an allocation request fails because of exhausted memory. The C++ project includes an example implementation of a growing pool allocator that is allowed to grow until it reaches a predefined maximum size constraint. The growing behavior is implemented by using *virtual memory api* provided by the core library.

Although custom allocators can boost an applications performance when used in the right situations, they have limited capabilities for discovering erroneous usage and memory bugs. Because finding such bugs can be hard without any helping utilities the implemented memory feature exposes some debugging capabilities to find common bugs. How they work and which bugs they can warn about is described in the following section.

5.3.3 Bounds checker

To harden an engine's memory system it has to prove some mechanism to prevent or at least detect memory bugs. Some of them that can rather easily be detected are memory leaks and stomps. While a memory leaks describes an allocation that was not deallocated and therefore cannot be used anymore, a stomp corrupts or overwrites memory outside of the allocation's range. The author decided to only implement a capability to detect memory stomps in that thesis due to time constraints. It was decided that more time shall be invested into the different allocators and the other subsystems instead.

A technique that is used for detecting memory stomps is called *bounds checking*. One of the simple variants of it is to insert some markers, also called *canaries*, before and after a user's memory block. When the user frees an allocation again, it is checked whether the inserted canaries still hold the marker values or not. If the either one of the front or end canary was corrupted, an assertion is triggered notifying the programmer that a memory stomp has occurred. There are more advanced approaches that check the integrity of all canaries at every allocation and deallocation made, which allows the for faster error detection for the cost of more computational effort.

The value chosen for the canaries often is one that is not commonly seen in the patterns of other allocations. In the thesis implementation each canary occupies 4 bytes in memory, being implemented by a 32-bit unsigned integer, and the stored marked value is `0xCA` in hexadecimal. Such a value is also easily detectable when debugging the application and inspecting the

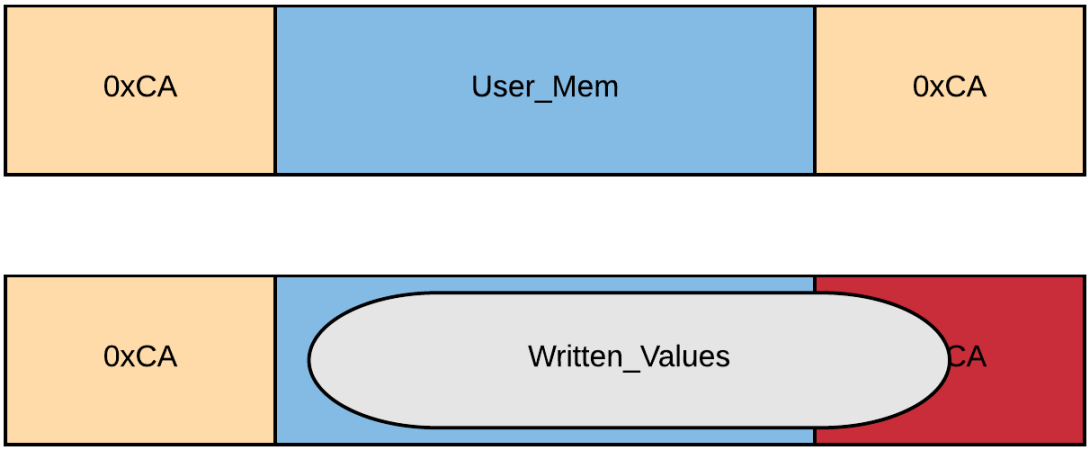


Figure 16: Top: an allocation wrapped by valid canaries. Bottom: a value was written into the memory, overflowing its bounds and therefore invalidating the back canary.

memory. Because using an additional bounds checker that wraps every allocation, the memory system has somehow to ensure that a memory block returned by an allocator is aligned although it also needs to provide extra space for the canaries. That's one of the situation why every allocator can take an additional *offset* parameter. When an offset of X bytes is provided, the allocator ensures that there are exactly X bytes space before the aligned pointer. So if an allocation request with an offset of 4 and an alignment of 16 is issues, the allocator returns a memory block that, if offset by 4 bytes, provides an alignment of 16. Figure 17 showcases how an allocation with an offset parameter can look like. Because it would be rather unergonomic to have to do this for every allocation done in the whole engine, the later described memory realm will show how to combine allocators and bounds checking to automate this step.

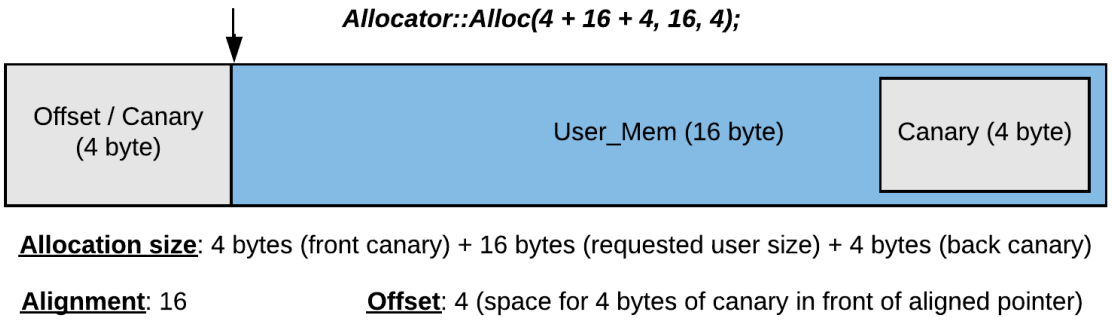


Figure 17: Illustrating the structure of a memory block allocated with an offset parameter. To also have space for the second canary, the allocation size if increased by 4 bytes.

As it can be seen the requested offset bytes are right in front of the aligned pointer that is later handed to the user. Because an additional 4 bytes of space is also needed for the canary at the block's back-end, the allocation size has to be increased by 4 bytes to allow the bounds checker writing to that address. The interface for the bounds checker written in Rust can be seen in Listing 16.

```
1 pub trait BoundsChecker {
2    unsafe fn write_canary(&self, memory: *mut u8);
3    fn validate_front_canary(&self, memory: *const u8);
4    fn validate_back_canary(&self, memory: *const u8);
5    fn get_canary(&self) -> u32;
6    fn get_canary_size(&self) -> u32;
7 }
```

Code 16: Base trait of every bounds checker in the Rust memory system.

The same interface is used in the C++ implementation. It has to be mentioned that there are two methods for both canary locations to allow meaningful assert messages when a corrupted value was detected. The next section is going to show a method for combining both every allocation strategy with every implemented bounds checking variant in a flexible and extensible way.

5.3.4 Memory realm

With allocators and bounds checking facilities in place, the memory system needs a way to expose them to a user in an ergonomically way. Although the different parts can already be used without any further abstraction, that way would be rather unergonomic for a programmer to use. In order to expose a more robust and flexible API, the memory system contains *memory*

realms. This type is an abstraction put on top of allocators and bounds checking to combine them. Both, C++ and Rust, make use of generic code to implement the memory realm. Listing 17 shows a partial implementation lacking some details about special member functions.

```
1 template <typename Allocator, typename BoundChecker>
2 class MemoryRealm : public MemoryRealmBase
3 {
4 public:
5    explicit MemoryRealm(size_t bytes)
6    : m_allocator(bytes)
7    {}
8
9    template<typename MemoryProvider>
10   explicit MemoryRealm(MemoryProvider& memoryProvider)
11   : m_allocator(memoryProvider.start(), memoryProvider.end())
12   {}
13
14   void* Alloc(size_t bytes, size_t alignment) override
15   {
16      const size_t totalMemory = m_boundsChecker.CANARY_SIZE + bytes +
           m_boundsChecker.CANARY_SIZE;
17
18      char* memory = static_cast<char*>(m_allocator.Alloc(totalMemory,
           alignment, m_boundsChecker.CANARY_SIZE));
19
20      m_boundsChecker.WriteCanary(memory);
21      m_boundsChecker.WriteCanary(memory + m_boundsChecker.CANARY_SIZE + bytes);
22
23      return memory + m_boundsChecker.CANARY_SIZE;
24   }
25
26   void Dealloc(void* memory) override
27   {
28      char* allocatorMemory = static_cast<char*>(memory) -
           m_boundsChecker.CANARY_SIZE;
29
30      m_boundsChecker.ValidateFrontCanary(allocatorMemory);
31      const uint32_t allocationSize =
           static_cast<uint32_t>(m_allocator.GetAllocationSize(allocatorMemory));
32      m_boundsChecker.ValidateBackCanary(allocatorMemory +
           m_boundsChecker.CANARY_SIZE + allocationSize);
33
34      m_allocator.Dealloc(allocatorMemory);
35   }
36
37   void Reset(void) override
38   {
39      m_allocator.Reset();
40   }
```

```
41
42     ~MemoryRealm() = default;
43
44 private:
45     Allocator m_allocator;
46     BoundChecker m_boundsChecker;
47 };
```

Code 17: Implementation of the memory realm in C++. Some implementation deatils were omitted.)

The code sample shows that a realm takes two template parameters defining the allocator and bounds checker used. An advantage of this design, also being named *policy-based design*, is that each realm can be rather easily configured to use every allocator and every bounds checker, if they adhere to the same interface. To give an example, one memory realm could use a linear allocator with canary-based bounds checking during development and is exchanged by a realm using the same linear allocator but with a bounds checker that only has no-op methods. The bounds checker used in release implemented each of its methods empty, allowing the called functions to yield a no-op when being compiled for release or retail mode. With this approach a user can define memory arenas in a flexible way, creating the ones fitting a certain use case. This approach could even be enhanced, as Stefan Reinalter also describes in his blog [7], by adding facilities for handling multi-threaded access and memory tracking. These parts can then also be added as template parameters and increase the possibilities a programmer has.

While the C++ implementation uses templates, the Rust approach is based upon the feature of *generics*. Combined with traits the Rust implementation allows for also constraining the passed types and force them to implement a certain interface. Although that functionality would also be implementable in C++, Rust includes it in its core language, while the C++ mechanism would be a custom creation (*type traits* or *concepts*). In Listing 18 & 19 it can be seen Rust's trait bounds can be useful when implementing a generic abstraction where an used type has to provide certain functionality defined by an interface.

```
1 pub struct BasicMemoryRealm<A: Allocator + BasicAllocator, B: BoundsChecker +
      Default> {
2     allocator: A,
3     bounds_checker: B,
4 }
```

Code 18: Rust interface of the basic memory realm allowing only basic allocators.

```
1 pub struct TypedMemoryRealm<A: Allocator + TypedAllocator, B: BoundsChecker +
      Default> {
2 allocator: A,
3 bounds_checker: B,
4 }
```

Code 19: Rust interface of the typed memory realm.

5.3.5 Idiomatic Rust implementation

While the C++ implementation returns raw pointers and indicates error states with a `nullptr`, the Rust project features a safe wrapper around allocations to also showcase an approach using idiomatic Rust principles. To allow the system to be used in safe Rust, the author needed a way to somehow make the allocation API more robust and safe. To reach this goal, the `Box` type of the standard library served as a reference. A `Box` is a way to heap allocate in Rust and it follows the resource aquisition is initialization (RAII) idiom as many other Rust implementations. Because the current version of the language does not feature custom allocators in a stable way, the author implement a custom type calls `AllocatorBox`. It was responsible for managing the allocated memory block and to free it properly when it is dropped. To do so it needed to hold a reference to the allocator the memory block comes from the the block itself. This also ensures, that an allocation can not live longer than the allocator it came from due to Rust's lifetime rules and the borrow checker.

```
1 pub struct AllocatorBox<'a, T: 'a + ?Sized, A: 'a + Allocator + ?Sized> {
2     instance: Unique<T>,
3     allocator: &'a A,
4 }
5
6 // ...
7
8 impl<'a, T: ?Sized, A: Allocator + ?Sized> Drop for AllocatorBox<'a, T, A> {
9 fn drop(&mut self) {
10    unsafe {
11        intrinsics::drop_in_place(self.instance.as_ptr());
12        self.allocator.dealloc_raw(MemoryBlock::new(self.instance.as_ptr() as *mut
              u8));
13    }
14 }
```

Code 20: AllocatorBox abstraction to allow RAII management of allocations.

Listing 20 showcases the members of an `AllocatorBox`. It uses the `Unique` type to indicate that it is the only owner over the value and a reference to the allocator the memory block came from. At the bottom it can be seen how the `Drop` trait is implemented and what happens when an `AllocatorBox` goes out of scope. The owned instance is dropped and then a memory block is created from the owned pointer. That block is handed back to the allocator it came from, properly releasing memory as soon as it is not needed anymore.

5.4 Containers

With the memory system as a quite low-level one, the second part of the implementation section focuses on a slightly more high-level library featuring three different kind of containers. A container serves the purpose of holding several elements, allowing to add and access them. The author decided to implement the following three collections, all of them capable of being used at runtime or for implementing tools.

vector dynamically size array, storing elements in a contiguous memory block

handle map collection that internally stores items contiguous in memory while external handles to them will be kept intact

ringbuffer collection allowing for reading and writing to it with the capability of wrapping around at the end

The following sections will each describe how the collection works and how it was implemented in C++ and Rust.

5.4.1 Vector

A vector is a collection most typically found in the standard library of most programming languages. With `std::vector<T>` and `Vec<T>`, both C++ and Rust, already provide an implementation to be used by a programmer. This collection can also be described as an array that is able to grow on demand. So while it is preferable to use fixed-size arrays due to reasons such as memory waste, book-keeping overhead and performance, there are some situations where the required size of an array cannot be known at compile-time. That are the situations where a vector is the tool of choice. But why is there the need for implementing such a collection if the languages already provide one? The reason is, as often in the world of game engines, performance. Standard library collections fulfill the need of being portable and that's the reason why there are several ways how the performance of them can be improved when certain unknowns are replaced by the knowledge of a machine's capabilities and usage patterns.

The difference between standard library implementations and the one chosen by the author is the way the collection retrieves its memory and how the growth is handled. Vectors commonly use the languages default memory allocation system to allocate space for its elements. While normally this is not a problem, it can become one as soon as the vector's current capacity is exhausted. During this situation, the vector allocates a new space that is bigger than the current one, moves all elements into that space and frees the old one. This can be either done by using allocation capabilities such as `realloc`, provided by the default allocator, or manually. Both of these approaches can lead to the problem that for a short time, the old and the new memory block are allocated at the same time, wasting memory until the old one is freed. The chosen implementation aims to avoid this problem by using the virtual memory system of the OS to allocate and free space.

Spark Vector

Contrary to the standard libraries vectors, the one implemented for Spark only allocates its memory via the virtual memory system of the OS. When a new one is created, it reserves virtual address space up to one GB. Due to that fact the implementation is only useful on 64-bit platforms to the bigger address space available on them. Whenever elements are added, or pushed, to the vector, a check is performed whether it needs to grow or not. If the check results that growing the collection is necessary, physical pages meeting the required capacity are committed. And here lies the different to the standard vector implementation. While normally in this situation memory usage could nearly be doubled, Spark's vector does not need to allocate or reallocate a new block. Because all of the already reserved virtual address space lies contiguous in memory, relocation of the previously owned elements is not needed and the new elements are just appended at the end of the already committed pages.

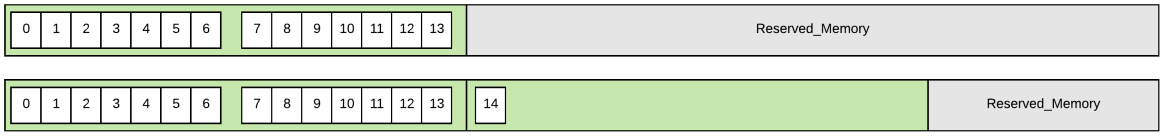


Figure 18: Illustrating the growing process of Spark's vector.

A visual description of the vector growth can be seen in Listing 18. The figure shows the situation where a new element is added to an already exhausted vector. To fit the element into the collection, another range of memory pages is committed. The amount of the newly allocated capacity is determined by fixed growing rules, ranging from twice the current size to more exotic growth patterns. What exactly is done during the growing process of the vector can be seen in Listing 21. This code sample shows how the `grow` method is implemented in Rust. The C++ function is now shown because they logically work in the same way only differing in syntax.

```
1 fn grow(&mut self, bytes: usize) {
2     {
3         let virtual_address_space_exhausted = self.internal_array_begin.as_ptr()
              as *mut u8 == self.virtual_mem_end;
4         debug_assert!(!virtual_address_space_exhausted, "Not␣enough␣address␣space␣
              to␣grow␣further");
5     }
6
7     let page_bytes_to_grow = math_util::round_to_next_multiple(bytes,
          virtual_mem::get_page_size());
8
9     let is_enough_space_for_requested_pages = unsafe {
          self.internal_array_end.offset(page_bytes_to_grow as isize) <=
          self.virtual_mem_end };
```

```
10    let grow_by_bytes = if is_enough_space_for_requested_pages {
11       page_bytes_to_grow
12    }
13    else {
14       let remaining_virtual_address_space = self.virtual_mem_end as usize -
             self.internal_array_end as usize;
15       math_util::round_to_previous_multiple(remaining_virtual_address_space,
             virtual_mem::get_page_size())
16    };
17
18    let ptr = match {
          virtual_mem::commit_physical_memory(self.internal_array_end,
          grow_by_bytes) } {
19       None => ptr::null_mut(),
20       Some(mem) => mem,
21    };
22
23    if ptr.is_null() {
24       debug_assert!(true, "Vector␣run␣out␣of␣memory␣due␣to␣an␣unknow␣error");
25    }
26
27    self.internal_array_end = unsafe { ptr.offset(grow_by_bytes as isize) };
28    self.capacity = self.capacity + (grow_by_bytes / mem::size_of::<T>());
29 }
```

Code 21: Growth function of the Spark vector in Rust.

While growing a vector is normally straight forward, there is a an edge case if the demanded capacity is not covered by available address space. If this situation occurs the vector does not assert or fail, but allocates all available pages until the end. That is the only case where the defined growing rule is ignore to utilize the available memory limits till the maximum. At the end, if committing the physical pages succeeded, the internal state, including current array end and capacity, is updated. It has to be mentioned that due to the virtual memory system, the vector can only grow in a multiple of the OS's page size.

5.4.2 Handle Map

The second collection that was implemented is called a *Handle Map* or a *Dense-Sparse Set*. Other collections often come with the disadvantage that either elements will not be contiguous in memory after deleting some of them or pointers to them are invalidated if the collection is able to reorder them to avoid holes. This collection solves this problem by handing out ids or so called *handles*. They are used to refer to some item in the collection instead of raw pointers. With that technique it is possible to reorder the internal array of items to keep them contiguous while also upholding already existing handles to them. In order to achieve this behavior the handle map contains three different internal arrays.

One of them is the *sparse array*, holding handles. A handle is simply a 64-bit unsigned

integer for the user. Internally it is interpreted as an index into the sparse set (4 bytes) and a *generation* part (4 bytes). The generation is important when requesting the item behind a handle to discover old references. Because handles are reused and array index can be used more than once and to ensure that no invalid or old handle can request an item, the generation of that exact handle is increased with every removing operation to deprecate all previously returned handles.

The second array used is called the *dense set* and holds objects stored by the collection. This array arranges its element to line up in memory and therefore yield better performance when iterating them due to good cache utilization. Figure 19 shows the two array described until now and how they relate to each other.

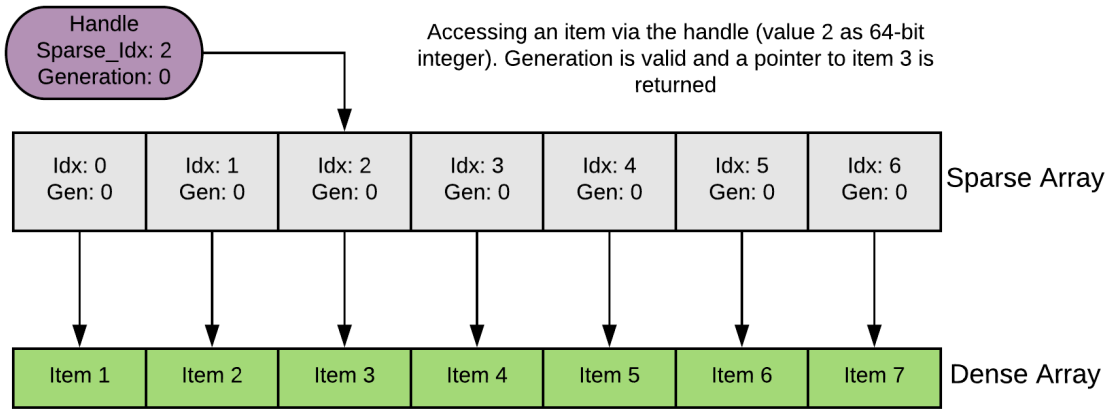


Figure 19: Sparse and dense array of the handle map. Relations defined by the indexes are described by arrows.

Whenever an item is deleted from this array, it is swapped with the last element to fill the hole and decrease the collection's size by one. But without any further processing, all handles to the previous last element would now be invalid which would mitigate all benefits that collection comes with. So whenever internal reordering occurs it is the responsibility of a handle map, to somehow bend the previously returned handles to point to the new location of the item.

For that situation the handle map contains a third array containing the reverse indexes from the dense array into the sparse array. So whenever an object is removed, the index of its handle data is looked up in that array. With that index the data related to the old handle can be accessed and the index into the dense array is overwritten by the new one. So the next time a user accesses the swapped value by its handle, the access into the sparse set will return the overwritten index, yielding the right item. With this process it is ensured that internally all elements are linear in memory while no handle is invalidated by the necessary reordering routine.

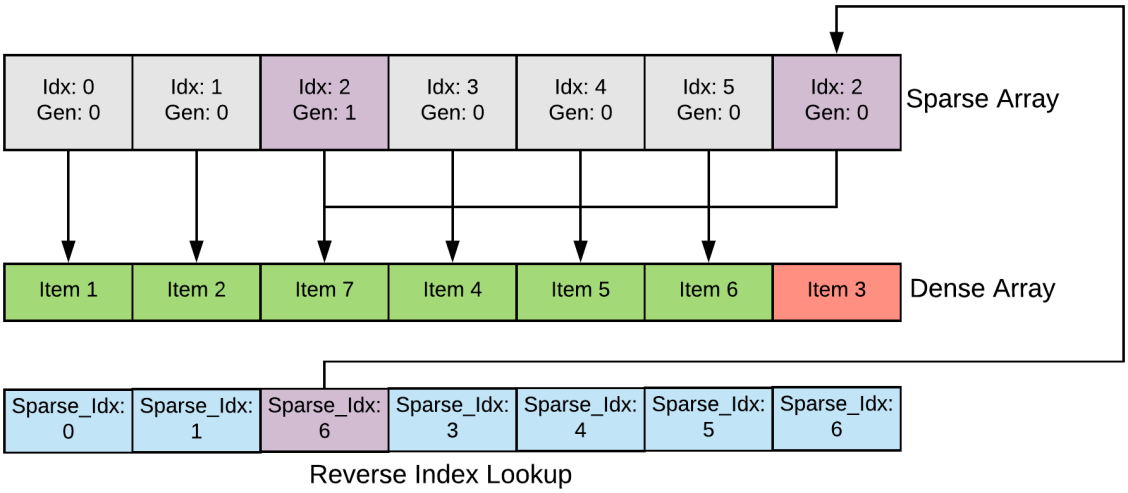


Figure 20: Reordering routine triggered by the deletion of item three. Ensuring old handle integrity of item seven.

As it can be seen in Figure 20, when item three is deleted it is swapped with the last element in the dense array, being item seven in the graphic. As reaction to that situation several parts have to be updated, highlight purple in the above figure. To ensure every old handle to item seven stays valid, the previous handle data stored in the sparse array is looked up with the help of the reverse index lookup table. Then the index into the dense array stored along the received handle is bent to point to the location where item three was stored until now. The index in the reverse lookup array is also updated to signify that from now on the index two of the dense array is referred to by the handle data stored at index 6 in the sparse array. The last action that has to be taken is the increase of the generation of the handle data pointing to the removed item. So, if an old handle containing a sparse array index two and a generation lesser than one is passed to the map, an assertion signals that the value behind that handle is not valid anymore.

```
1 template <typename Item>
2 void HandleMap<Item>::Erase(Handle handle)
3 {
4    const InternalId internalId = pointerUtil::pseudo_cast<InternalId>(handle, 0);
5
6    {
7       const bool userIndexInRange = internalId.sparseArrayIndex < m_itemCount;
8       assert(userIndexInRange && "Index␣of␣handle␣was␣out␣of␣range␣of␣the␣handle␣
            array");
9    }
10
11   HandleData& handleData = m_handles[internalId.sparseArrayIndex];
12
```

```
13    const bool validGeneration = internalId.generation == handleData.generation;
14    if (!validGeneration)
15    {
16       return;
17    }
18
19    const uint32_t lastMeshIndex = m_itemCount - 1;
20    m_items[handleData.denseArrayIndex] = m_items[lastMeshIndex];
21    m_items[lastMeshIndex].~Item();
22
23    const uint32_t indexOfMovedItem = m_meta[lastMeshIndex].denseToSparseIndex;
24    m_handles[indexOfMovedItem].denseArrayIndex = handleData.denseArrayIndex;
25    m_meta[handleData.denseArrayIndex].denseToSparseIndex = indexOfMovedItem;
26
27    m_freeList.ReturnChunk(&m_handles[internalId.sparseArrayIndex]);
28
29    ++handleData.generation;
30    m_itemCount = lastMeshIndex;
31 }
```

Code 22: Deletion of an item in the C++ implementation of the handle map

Additionally to Figure 20, Listing 22 shows how that routine is implement in C++. The Rust implementation is rather similar to the shown function. The major difference is that the Rust implementation returns an `Option` wrapping a handle. That forces the programmer to check whether the returned handle is a valid one or none, whereas the C++ implementation returns an integer that has to be checker for validity due to previously defined rules how invalid state is communicated.

5.4.3 Ringbuffer

The last collection in the contain library is called a *Ringbuffer*. A ring or circular buffer is a data structure that uses an internal fixed-sized array and is able to wrap around if the writing index reaches array length. Such a buffer can either use single bytes as its elements, allowing the user to write arbitrary data, or use typed elements. The implemented variant uses fixed-size objects per ringbuffer. A common use case of that structure are producer-consumer queues where two threads, or routines in general, write and read from the same ringbuffer, using it to communicate and share data. The implemented solution contains the fixed-size array and two additional integers that hold the state of the current write and read index. Figure 21 showcases the internal state of the ringbuffer in the most trivial way. The writ index is pointing to the next slot a value can be written to while the read index is indicating the next value to be read. If any index reaches the last element ad index six and wants to advance further, it wraps around starting at index zero again.

Due to the wrap around that happens as soon an index reaches the end of the array, writing to a ringbuffer can cause the oldest values contained to be overwritten. Some variants do

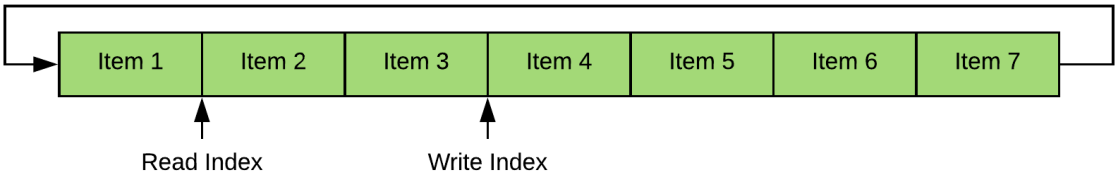


Figure 21: Illustration of the ringbuffer showing the two indexes and the important wrap around when the end is reached.

not allow overwriting old values and yield an error or raise an exception in such a case. Both, the C++ and Rust implementation, allow for overwriting old values when the write index wraps. Another constraint the ringbuffer has to ensure is that the read index is not allowed to overtake the write index or otherwise it would read invalid or old data, running into undefined behavior in certain scenarios. The case of wrapping around is even more complicated when the buffer is used to store bytes and allow a user to insert arbitrary sized elements. It can happen that such an object is split up, having half of it stored at the end of the buffer while the other half is located at the beginning due to wrapping rules. If now the value is read again, the two parts have somehow be copied together to return the memory block previously written. One practice to simplify that case is to implement a technique known as a *magic ringbuffer*. With the magical variant, the buffer is mapped to two adjacent virtual memory regions, allowing it to be read linearly and the only edge case occurs when the read pointer advances into the second virtual memory region. If that happens the read and write index is decreased by the buffer's length to be in the first sector again. Listing 23 shows how the write function is implemented in the Rust ringbuffer and how the case of wrap around is handled.

```
1 pub fn write(&mut self, item: T)
2 {
3     self.empty = false;
4
5     self.items[self.write_idx] = item;
6     self.write_idx = (self.write_idx + 1) % self.capacity;
7
8     if self.write_idx == self.read_idx {
9         self.read_idx += 1;
10    }
11 }
```

Code 23: Write function of the Rust ringbuffer showing how the wrap around case is handled

One big difference between the Rust and C++ implementation is how memory for the internal array is allocated. In the C++ version, which can be seen in Listing 24, the internal array is allocated on the stack because the maximum capacity is passed to the template class as a

non-type template parameter. With that design there are no dynamic memory allocations involved. As it was previously mentioned in section 4.8.1, Rust currently has no concept of const generics, which makes it currently not possible to provide a similar interface in Rust. Due to that fact a vector has to be used that is then initialized with the maximum capacity passed at construction of the ringbuffer. This involves at least one dynamic memory allocation at runtime that could be omitted if it would be possible to use compile-time generic parameters to create arrays in Rust.

```
1 template <typename Item, size_t Capacity>
2 class RingBuffer
3 {
4    public:
5    RingBuffer();
6
7    void Write(const Item& element);
8    void Write(const Item&& element);
9
10   Item* Read(void);
11   Item* Peek(void);
12   void  Reset(void);
13
14   bool IsEmpty(void) const { return m_isEmpty; }
15   bool IsFull(void)  const { return !IsEmpty(); }
16
17   size_t Size(void) const;
18
19   private:
20   bool m_isEmpty;
21   size_t m_writeIndex;
22   size_t m_readIndex;
23   Item m_items[Capacity];
24 };
```

Code 24: Interface of the C++ Ringbuffer. Capacity is a non-type template parameter used to control the maximum element size.

5.5 Entity Component System

The last implemented system is an ECS. Because the performance comparison between two languages is the main goal of this thesis the author reimplemented an already existing Rust ECS in C++. The project that served as reference is called *calx-ecs* and is a rather simple variant of an ECS. This section will describe its API and what parts are implemented differently in C++.

5.5.1 Calx ECS

Calx-ecs was chosen due to its simple design which allows for a quick reimplementation in another language. The project is hosted at GitHub and can be found at https://github.com/rsaarelm/calx-ecs. Although there are more popular ECS implementations in the Rust ecosystem, this one was chosen because a full reimplementation of a more complex one would have exceeded the available amount of time for that part of the thesis. *Calx-ecs* features the following parts that are described by the following paragraphs:

world — responsible for entity creation and component storage

entity — thin wrapper around two integers, an entity is only an id that groups components

component — a type that can be added to an entity, defined by the user by deriving a component interface (class in C++ / trait in Rust)

5.5.2 Entities & Components

As already described in section 3.4.1 an ECS is based upon composition instead of inheritance. While some variants implement an entity as a class containing a list of components, another one defines entities as simple ids that group components. The second approach is also the one that is used in this thesis. Every entity is a unique id stored in a 32-bit unsigned integer. Additionally to the id the `Entity` type also contains an index (also a 32-bit unsigned integer) that is used to describe where the entity is stored in the collection inside the `World` class. This allows the system to store a maximum of around 4 million entities.

Beside the `Entity` the system allows a user to define custom components which can be added and removed to an entity. The original Rust implementation defined a `Component` trait that could be implemented by user-defined types. Each component, then has to be registered with the world and a `ComponentStorage` is created per registered type. This allows for storing components contiguous in memory by using a quite similar approach then the handle map introduced in section 5.4.2. Listing 25 shows the interface of the `ComponentStorage` that is responsible to insert and removed components and keep track of which entity owns them.

```
1
2 // DenseStorage is a contiguous vector of entitites and the components of type C
3 template<typename Comp>
4 struct DenseStorage
5 {
6     std::vector<Comp> data;
7     std::vector<Entity> entities;
8 };
9
10 // Base class to allow storing differently typed storages in a polymorphic
11 // collection
```

```cpp
class Storage
{
   public:
   virtual void remove(const Entity& entity) = 0;
   virtual bool contains(const Entity& entity) = 0;
   virtual ~Storage() = default;
};

// The templated ComponentStorage being specialized for every user-defined
// component derived type
template<typename Comp>
class ComponentStorage : public Storage
{
   public:
   void insert(const Entity& entity, Comp& comp);
   void remove(const Entity& entity) override;
   bool contains(const Entity& entity) override;
   const std::vector<Entity>& entities(void) const;
   Comp* get(const Entity& entity);
   const Comp* get(const Entity& entity) const;

   private:
   DenseStorage<Comp> m_innerStorage;
   std::vector<Index> m_indexIntoInnerData;
};
```

Code 25: ComponentStorage interface of the C++ implementation

Due to the design of the `ComponentStorage` and the decision to use a distinct storage per component type, iterating over all components from one type is cache-friendly and yields good performance due to the linear memory layout. Because all storages and therefore all components are stored in the `World`, it is also the only way of retrieving references to single components or to query for entities owning a certain component.

5.5.3 World

The `World` is the place of the ECS where all entities and components are managed. It is responsible for creating new entities, registering component types and managing how they are stored internally. To allow the system to work with a variable amount of user-defined types, every new component type has to be registered before an instance can be inserted. In the Rust implementation this was done by a macro which implements the `Component` trait for every specified type and adds a property of a typed `ComponentStorage` to the world type. Listing 26 shows how the invocation of the macro looks like in Rust. Because of the macro's internals and how it defines the `ECS` type, the Rust system only allows for one world, containing fixed components, to exist.

```
Ecs! {
    pos: Position,
    vel: Velocity,
}
```

Code 26: Registering custom components with the Rust ECS

Contrary to the macro approach, the reimplementation in C++ did not adhere to this approach. This decision was made due to the very different macro system of the both languages. Rust's macros create an AST the programmer can work with and already includes features to execute part of the macro for any amount of arguments passed. While this can also be done in C++ with variadic macros it is way harder and also not as flexible. With the C++ variadic macro approach it is not possible to support any amount of arguments but only the ones defined by the user. To mirror the behavior of the Rust crate,where any number of component types can be registered, the C++ implementation also wanted to supply that possibility. Listing 27 shows how the registration process works. Whenever a new component is added to an entity, the world checks whether there already exists a corresponding storage and if it does not then a new one is created and stored in an internal hash map. The key for this map is the unique id of the component which has to be defined by the user whenever a new component subtype is created.

```
template <typename C>
void World::addComponent(const Entity& entity, C component)
{
  const uint32_t componentId = C::UID();
  if (m_componentStorages.count(componentId) == 0)
  {
     m_componentStorages.insert({ componentId, new ComponentStorage<C>() });
  }

  ComponentStorage<C>* storage =
      static_cast<ComponentStorage<C>*>(m_componentStorages.at(componentId));
  if (contains(entity))
  {
     storage->insert(entity, component);
  }
}
```

Code 27: Method to add components, implicitly registering them on first use

The unique id of a component used to distinguish between them is requested from a static method the type `C` has to implement. An implementation of that method can look like this: `static uint32_t UID() { return 1u; }`. It is the user's responsibility to ensure that every id returned by `UID` is unique or the system will run into undefined behavior.

5.6 Conclusion

With all the systems implemented in two languages, the author wants to shortly summarize personal experiences and opinions about Rust and the implementations. Coming from the C++ world it was at first difficult to transfer the concepts and architectural choices for the systems to Rust. With the borrow checker complaining about lifetimes and more than one mutable reference to a value, it was challenging to refactor the Rust code to even compile. But after the first few pitfalls and due to the helpful and verbose compile errors of *rustc*, the author adapted a more idiomatic style and the Rust implementations become robust. It was then quite the opposite that some of the features Rust provide were missing when a C++ implementation was reworked. While Rust offers types such as `Option` for describing the state of a return value, C++ sticks with pointers and using `nullptr` to indicate an error. It can be said, in the opinion of the author, that Rust served well as a language for implementing the modules and that the built-in test facilities of cargo were helpful.

The next chapter will include the results of the performance measurements of the implemented systems. It will first describe the benchmark setup and what scenarios where chosen to compare the module to each other. After that the findings will be discussed and differences between the languages will be shown, supported by diagrams of the measurement data.

6 Submodule benchmarks

While the last chapter discussed the implementation details of the different modules, this chapter will present the results of the performance measurements. One of the key concerns for a game engine is performance and that is the reason why it is the computation time of several benchmark scenarios that is compared. The results and findings of the measurement process are then visualized to show where the performance of the implementations diverge and where they are on the same level. Starting with a description of the benchmark process and the environment, the first part will also list the scenarios that were measured.

6.1 Benchmark setup

6.1.1 Time measurement

Retrieving fine-grained and reliable timestamps is important when comparing different implementations against each other. To measure elapsed time the first naive solution would be to use a facility such as the `time()` function from the C standard library. But since `time()` only returns the elapsed seconds since the first January 1970, a more precise method is required. A possible solution can be a high-resolution timer on a CPU. Because this kind of timer uses a register to store the amount of cycles passed since the CPU was started, it is possible to generate more precise timestamps yielding better resolution of differences between them. The option chosen for measuring time on the Windows OS is `QueryPerformanceCounter()` from the *Win32 API*.

6.1.2 Environments

The benchmark process is run on two different hardware setups. The following tables describe the hardware on these machines and group them under a name per environment. The following sections will use these names to refer to the different environments.

Environment name	**Processor / CPU**	**RAM**
Mobile Workstation	*Processor Intel(R) Core(TM) i7-6700 @ 3.40GHz, 4 Cores*	*16 GB*
Razer Blade Stealth	*Processor Intel(R) Core(TM) i7-7500U @ 2.70 GHz, 2 Cores*	*16 GB*

Table 1: Benchmark environment names and hardware capabilities

6.1.3 Benchmark applications

To measure the different scenarios that are listed in the next section, every project includes an application that is able to run them. In order to yield minimal overhead when choosing benchmark scenarios every benchmark executable includes a static lookup table where the function pointers to the scenarios are stored. Each scenario has an id which also serves as an index into the lookup table. When the application is started the id is passed as a command-line parameter. This behavior was chosen to allow the author to automate the benchmark process via command-line scripts. Listing 28 shows how the C++ main function of a benchmark application looks like. At the end it can be seen, that the scenario id passed as a command-line argument is converted into an integer and then used to index into the scenario table. The signature of every benchmark scenario has to follow the one defined as `typedef void(*BenchmarkScenarioFunction)();`. The Rust benchmark main function work similar to the C++ one.

```
1 int main(int argc, char** argv)
2 {
3    if (argc <= 1 || argc > 2)
4    {
5       std::cerr << "Usage: bench_spark++.exe SCENARIO_ID" << std::endl;
6       return -1;
7    }
8
9    scenarios[atoi(argv[1])]();
10 }
```

Code 28: Main function of the C++ benchmark app using a scenario lookup table

6.1.4 Scenarios

To measure the performance of the different systems the following benchmark scenarios were implemented.

Memory management

1. **RawAllocators**: one scenario was implemented per allocator to test raw allocations with the natural alignment of a common struct. Every scenario allocates 1000 elements. Beside the custom allocators the measurement includes the default dynamic heap allocation systems from the two languages.
2. **MemoryRealm_LinearAllocator**: a scenario where 1000 small object allocations are made via a memory realm. The realm includes a linear allocator and uses simple bounds checking.

Container

1. **VecDefault**: push 10000 small objects into a vector with no previous capacity, forcing it to grow at least once.
2. **VecCapacity**: push 10000 small objects into a vector with previously requested capacity
3. **VecIteration**: push 10000 small objects into a vector with previously requested capacity and iterate them afterwards
4. **VecErase**: push 10000 small objects into a vector with previously requested capacity and erase them one by one afterwards
5. **HandleMapInsertion**: insert 10000 small objects into the handle map
6. **HandleMapIteration**: insert 10000 small objects into the handle map and iterate them afterwards
7. **HandleMapRemove**: insert 10000 small objects into the handle map and remove them afterwards one by one
8. **RingbufferWrite**: insert 10000 small objects into the ringbuffer until it is full
9. **RingbufferRead**: insert 10000 small objects into the ringbuffer and consume them all
10. **RingbufferWrite_Wrapping**: insert 15000 objects into the ringbuffer overwriting the oldest 500

Entity Component System

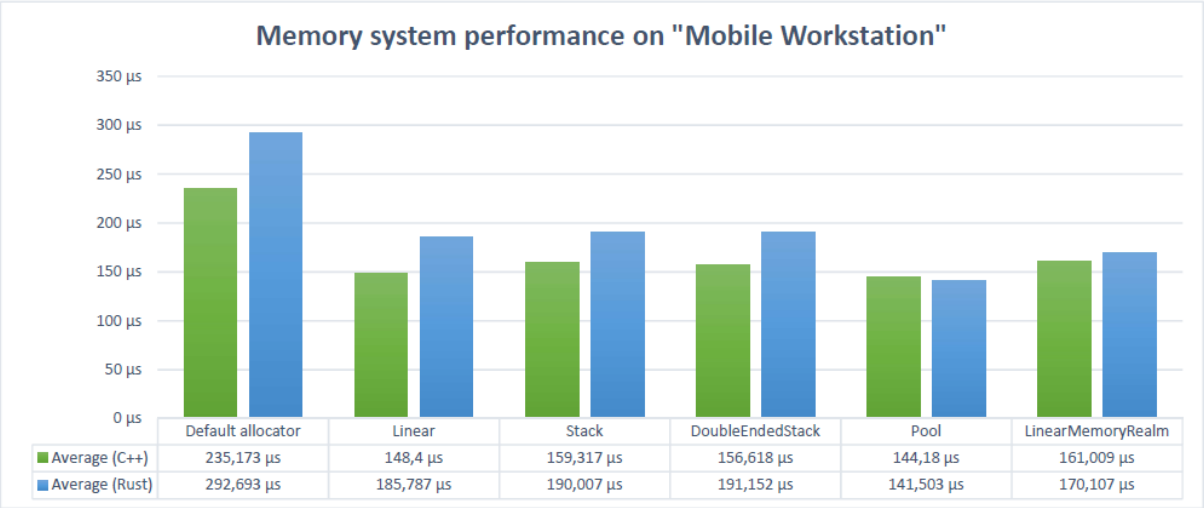
1. **Position**: create 10000 entities and add a position component to each of them
2. **PositionVelocity**: create 10000 entities with one position and one velocity component
3. **Iteration**: iterate over 10000 position components of previously created entities
4. **Remove**: create 10000 entities with position components and remove 5000 components afterwards

6.2 Results

6.2.1 Memory Management

Considering the benchmarks shown in Figure 22a and Figure 22b the C++ version of the memory system performs better than the Rust implementation. The execution times of the Rust allocators are slightly slower than the C++ one, but only differ in a range from 5 to 50 microseconds. The pool allocator on the mobile workstation and the linear memory realm on the razor blade stealth, outperform the C++ system but only by a few microseconds.

A baseline for the general performance of the allocator system is the first data-set in the graph that shows the execution time of the default allocators of the two languages. In C++ that facility was `new` while the Rust variant uses `Box` to dynamically allocate memory.

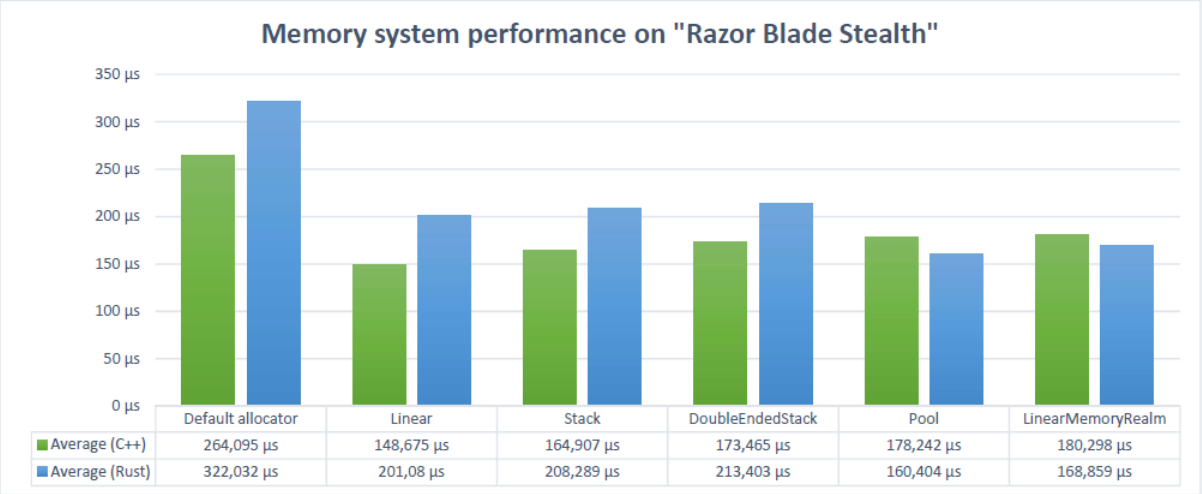


(a) Visual comparison of the different allocators, in Rust and C++

Scenario	Average (C++)	Min (C++)	Max (C++)	Average (Rust)	Min (Rust)	Max (Rust)	Diff	Percentage
Default allocator	235,173 µs	182,38 µs	437,18 µs	292,693 µs	230,46 µs	564,281 µs	57,52 µs	21,79%
Linear	148,4 µs	106,36 µs	308,28 µs	185,787 µs	144,225 µs	423,361 µs	37,387 µs	22,37%
Stack	159,317 µs	109,67 µs	296,26 µs	190,007 µs	143,925 µs	428,469 µs	30,69 µs	17,57%
DoubleEndedStack	156,618 µs	111,17 µs	382,79 µs	191,152 µs	146,028 µs	481,953 µs	34,534 µs	19,86%
Pool	144,18 µs	109,67 µs	301,37 µs	141,503 µs	110,272 µs	366,272 µs	2,677 µs	1,87%
LinearMemoryRealm	161,009 µs	116,28 µs	374,98 µs	170,107 µs	115,981 µs	393,014 µs	9,098 µs	5,50%

(b) Exact measurements of the benchmarks, in microseconds

Figure 22: Memory system benchmarks on the mobile workstation configuration



(a) Visual comparison of the different allocators, in Rust and C++

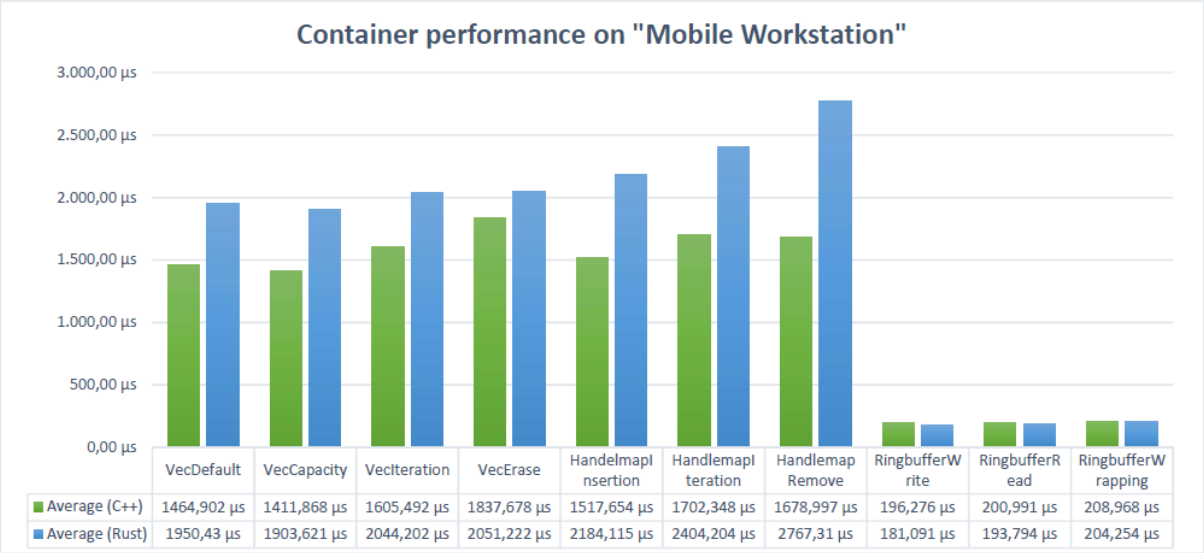
Scenario	Average (C++)	Min (C++)	Max (C++)	Average (Rust)	Min (Rust)	Max (Rust)	Diff	Percentage
Default allocator	264,095 µs	200,28 µs	790,92 µs	322,032 µs	254,237 µs	1191,142 µs	57,937 µs	19,77%
Linear	148,675 µs	122,35 µs	240,83 µs	201,08 µs	169,256 µs	328,992 µs	52,405 µs	29,97%
Stack	164,907 µs	126,94 µs	464,75 µs	208,289 µs	171,372 µs	458,05 µs	43,382 µs	23,25%
DoubleEndedStack	173,465 µs	130,82 µs	351,2 µs	213,403 µs	176,309 µs	306,425 µs	39,938 µs	20,65%
Pool	178,242 µs	136,46 µs	437,95 µs	160,404 µs	134,7 µs	276,805 µs	17,838 µs	10,53%
LinearMemoryRealm	180,298 µs	138,22 µs	371,3 µs	168,859 µs	138,931 µs	244,717 µs	11,439 µs	6,55%

(b) Exact measurements of the benchmarks, in microseconds

Figure 23: Memory system benchmarks on the razor blade stealth configuration

6.2.2 Container

Based on the figures 24a & 24b the C++ version of the container library performs better than the Rust variant. The Rust vector is about 25% to 30% slower than the C++ version, mostly due to the way how the Rust vector writes the data into the internal array. It is first moved into the `push` method and then immediately written into a memory slot of the internal array while the C++ variant uses `placement new` to create a new instance of the object into vector's place. The handlemap implementation yields better results in C++, with an average difference of about 35% to 40% percent. One difference that introduce a small overhead in the Rust handlemap is the way dynamic memory for the internal arrays is allocated. Because the implementation's goal was to use stable features where possible, the allocation library for raw memory could not be used. The idiomatic Rust way would then be to use a `Vec` to allocate the memory needed, but the author decided to use the virtual memory allocator already used for the allocators and the vector. The last container, the ringbuffer, performs slightly better in Rust, but inly by a difference of four and seven microseconds in the performed scenarios.

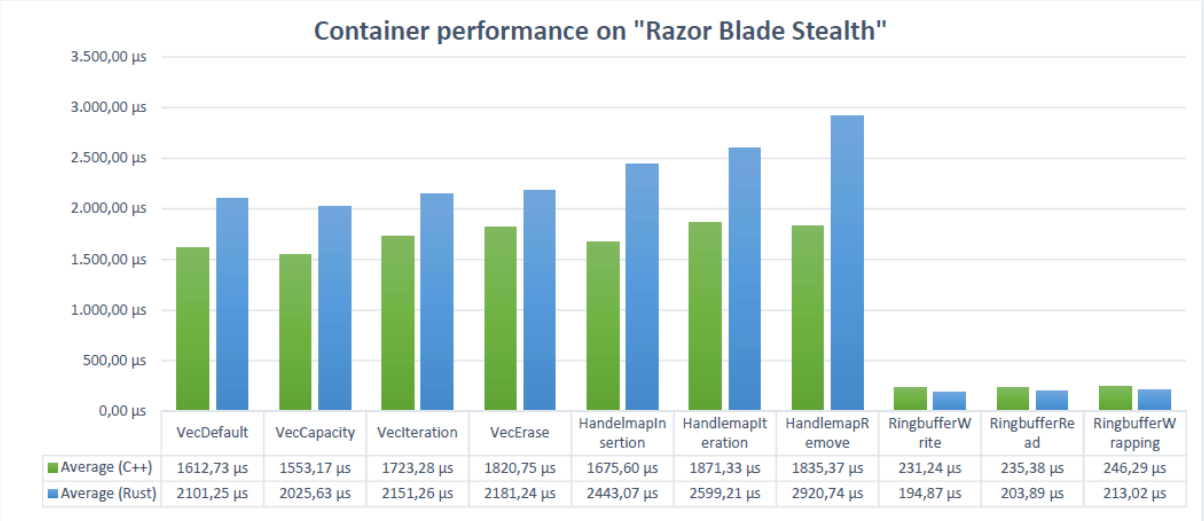


(a) Visual comparison of the different containers, in Rust and C++

Scenario	Average (C++)	Min (C++)	Max (C++)	Average (Rust)	Min (Rust)	Max (Rust)	Diff	Percentage
VecDefault	1464,902 µs	1142,68 µs	2301,59 µs	1950,43 µs	1629,144 µs	3101,143 µs	485,528 µs	28,43%
VecCapacity	1411,868 µs	1114,74 µs	2231,88 µs	1903,621 µs	1586,177 µs	3000,486 µs	491,753 µs	29,66%
VecIteration	1605,492 µs	1231,62 µs	2751,09 µs	2044,202 µs	1659,792 µs	3026,025 µs	438,71 µs	24,04%
VecErase	1837,678 µs	1416,11 µs	3561,46 µs	2051,222 µs	1682,327 µs	3199,396 µs	213,544 µs	10,98%
HandelmapInsertion	1517,654 µs	1179,64 µs	2609,57 µs	2184,115 µs	1761,351 µs	3675,94 µs	666,461 µs	36,01%
HandlemapIteration	1702,348 µs	1314,85 µs	2883 µs	2404,204 µs	1920,599 µs	4053,931 µs	701,856 µs	34,18%
HandlemapRemove	1678,997 µs	1323,26 µs	2707,52 µs	2767,31 µs	2254,721 µs	4993,498 µs	1088,313 µs	48,95%
RingbufferWrite	196,276 µs	159,54 µs	432,07 µs	181,091 µs	131,906 µs	502,685 µs	15,185 µs	8,05%
RingbufferRead	200,991 µs	161,05 µs	439,58 µs	193,794 µs	146,028 µs	682,967 µs	7,197 µs	3,65%
RingbufferWrapping	208,968 µs	169,76 µs	406,23 µs	204,254 µs	150,836 µs	525,22 µs	4,714 µs	2,28%

(b) Exact measurements of the benchmarks, in microseconds

Figure 24: Container benchmarks on the mobile workstation configuration



(a) Visual comparison of the different containers, in Rust and C++

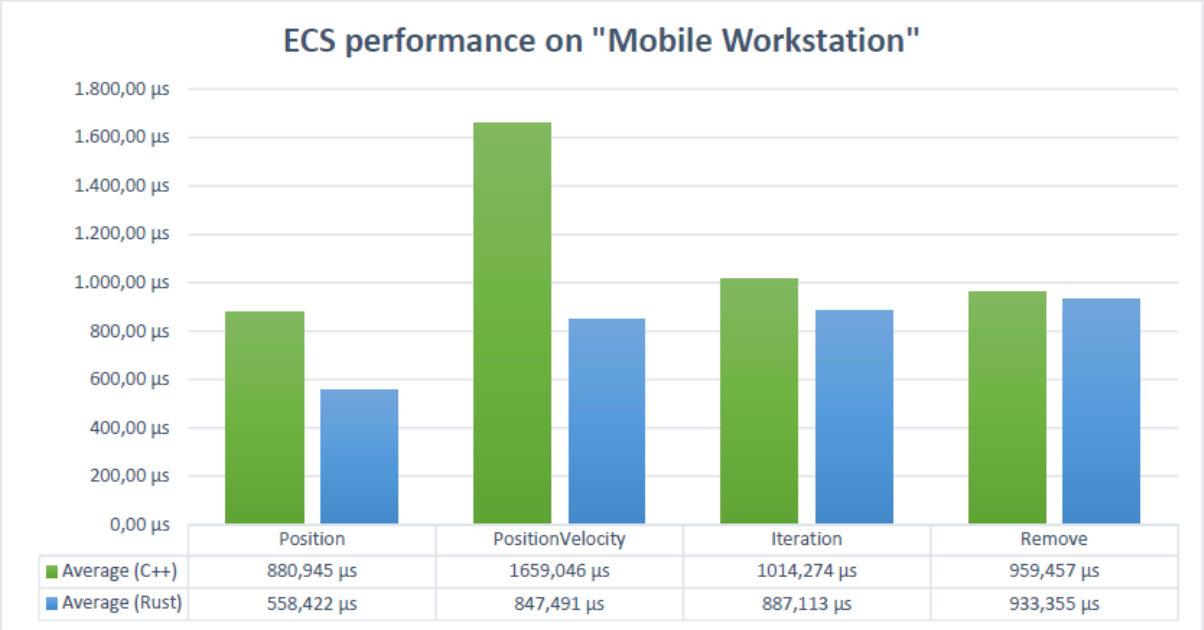
Scenario	Average (C++)	Min (C++)	Max (C++)	Average (Rust)	Min (Rust)	Max (Rust)	Diff	Percentage
VecDefault	1612,73 µs	1319,84 µs	2723,61 µs	2101,25 µs	1893,20 µs	3524,77 µs	488,52	26,31%
VecCapacity	1553,17 µs	1257,08 µs	2630,17 µs	2025,63 µs	1840,66 µs	2819,53 µs	472,46	26,40%
VecIteration	1723,28 µs	1372,38 µs	2807,89 µs	2151,26 µs	1916,83 µs	2804,72 µs	427,98	22,09%
VecErase	1820,75 µs	1499,33 µs	2790,61 µs	2181,24 µs	1984,53 µs	2855,50 µs	360,50	18,02%
HandelmapInsertion	1675,60 µs	1342,41 µs	2272,62 µs	2443,07 µs	2077,98 µs	3435,55 µs	767,48	37,27%
HandlemapIteration	1871,33 µs	1527,53 µs	2572,34 µs	2599,21 µs	2231,01 µs	3857,28 µs	727,88	32,56%
HandlemapRemove	1835,37 µs	1518,01 µs	3235,61 µs	2920,74 µs	2541,31 µs	4383,39 µs	1.085,36	45,64%
RingbufferWrite	231,24 µs	186,18 µs	990,50 µs	194,87 µs	152,33 µs	340,98 µs	36,37	17,07%
RingbufferRead	235,38 µs	191,82 µs	368,13 µs	203,89 µs	168,20 µs	479,21 µs	31,49	14,34%
RingbufferWrapping	246,29 µs	197,11 µs	357,20 µs	213,02 µs	177,01 µs	405,86 µs	33,27	14,49%

(b) Exact measurements of the benchmarks, in microseconds

Figure 25: Container benchmarks on the razor blade stealth configuration

6.2.3 Entity Component System

The results of the ECS benchmark yielded the results that the Rust implementation,the calx-ecs crate, yields better performance as the reimplementation in C++. That difference is mostly due to the way the component storages were implemented in the Rust ECS and how they work in the C++ variant. While in calx-ecs the creator provides a macro to register components with the `Ecs` type, for every type provided, the macro adds a field to the `Ecs` struct. Because the macros system of Rust differs from the preprocessor-based one in C++, the reimplementation used a different approach. Because with C++ macros, the flexibility and amount of types is constrained by the macro expansion rules (variadic macros need a distinct macro for every argument count to serve the same purpose), the C++ ECS uses a hashmap that associates component storages with an unique component id. The lookup into that map to get the right storage for a component introduces an overhead that could be avoided in Rust. Figures **??** and 26b show these results and the second scenario benchmark also visualizes how in C++ the performance impact scales directly with the amount of components added to an entity.

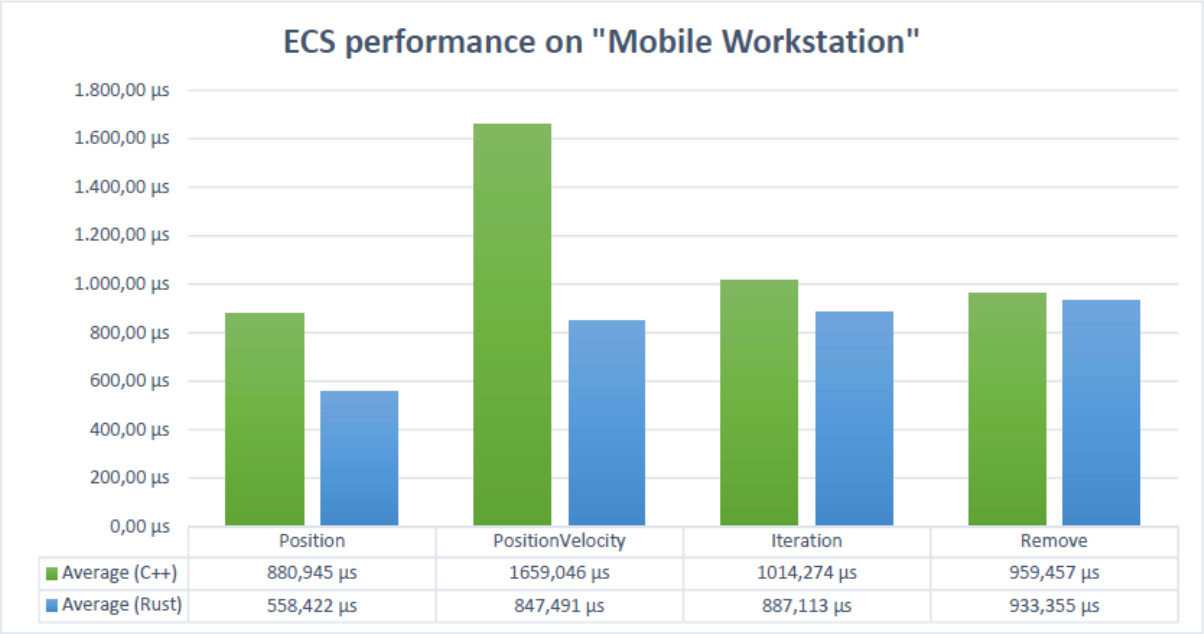


(a) Visual comparison of the ECS implementations, in Rust and C++

Scenario	Average (C++)	Min (C++)	Max (C++)	Average (Rust)	Min (Rust)	Max (Rust)	Diff	Percentage
Position	880,945 µs	690,47 µs	1492,13 µs	558,422 µs	463,023 µs	987,342 µs	322,52 µs	44,81%
PositionVelocity	1659,046 µs	1273,38 µs	2734,27 µs	847,491 µs	684,769 µs	1527,585 µs	811,56 µs	64,76%
Iteration	1014,274 µs	799,54 µs	1685,03 µs	887,113 µs	709,107 µs	1552,524 µs	127,16 µs	13,38%
Remove	959,457 µs	722,62 µs	1704,26 µs	933,355 µs	774,91 µs	1549,219 µs	26,102 µs	2,76%

(b) Exact measurements of the benchmarks, in microseconds

Figure 26: ECS benchmarks on the mobile workstation configuration



(a) Visual comparison of the ECS implementations, in Rust and C++

Scenario	Average (C++)	Min (C++)	Max (C++)	Average (Rust)	Min (Rust)	Max (Rust)	Diff	Percentage
Position	880,945 µs	690,47 µs	1492,13 µs	558,422 µs	463,023 µs	987,342 µs	322,52 µs	44,81%
PositionVelocity	1659,046 µs	1273,38 µs	2734,27 µs	847,491 µs	684,769 µs	1527,585 µs	811,56 µs	64,76%
Iteration	1014,274 µs	799,54 µs	1685,03 µs	887,113 µs	709,107 µs	1552,524 µs	127,16 µs	13,38%
Remove	959,457 µs	722,62 µs	1704,26 µs	933,355 µs	774,91 µs	1549,219 µs	26,102 µs	2,76%

(b) Exact measurements of the benchmarks, in microseconds

Figure 27: ECS benchmarks on the razor blade stealth configuration

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] C. Lattner, "Llvm."

[3] S. Reinalter, "Memory system – part 1," https://blog.molecular-matters.com/2011/07/05/memory-system-part-1/, [Online; accessed 25-04-2018, 22:41]. [Online]. Available: https://blog.molecular-matters.com/2011/07/05/memory-system-part-1/

[4] ——, "Memory system – part 2," https://blog.molecular-matters.com/2011/07/07/memory-system-part-2/, [Online; accessed 25-04-2018, 22:41]. [Online]. Available: https://blog.molecular-matters.com/2011/07/07/memory-system-part-2/

[5] ——, "Memory system – part 3," https://blog.molecular-matters.com/2011/07/08/memory-system-part-3/, [Online; accessed 25-04-2018, 22:41]. [Online]. Available: https://blog.molecular-matters.com/2011/07/08/memory-system-part-3/

[6] ——, "Memory system – part 4," https://blog.molecular-matters.com/2011/07/15/memory-system-part-4/, [Online; accessed 25-04-2018, 22:41]. [Online]. Available: https://blog.molecular-matters.com/2011/07/15/memory-system-part-4/

[7] ——, "Memory system – part 5," https://blog.molecular-matters.com/2011/08/03/memory-system-part-5/, [Online; accessed 25-04-2018, 22:41]. [Online]. Available: https://blog.molecular-matters.com/2011/08/03/memory-system-part-5/

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

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

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

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

[12] 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

RFC request for change

LLVM low-level virtual machine

AST Abstract Syntax Tree

IR intermediate representation

RAII resource aquisition is initialization

IPC inter process communication

LIFO last-in-first-out