IT UNIVERSITY OF COPENHAGEN

Master Thesis for
Master of Science in Games

A general-purpose game engine for the original Sony PlayStation

Authored by

Malte Bryndum Pedersen (mbpe@itu.dk)

Supervised by

Paolo Burelli (pabu@itu.dk)

Christian Carvelli (chcha@itu.dk)

Activity Code

KISPECI1SE

June 2024

Acknowledgements

First, a sincere thanks to my supervisors who agreed to do this project with me and allowed me to spend almost half a year on one of my most profound interests. I appreciate the support and the interest you have shown, and the work you have put into making this setup work. Your approach and advice have kept my motivation high.

My work would not have been possible if not for the skilled, approachable and very helpful individuals of the community surrounding the original PlayStation. Their almost 30 years of impressive research and work, done in their free time, has played a fundamental role in this project, and their help and feedback has been crucial in making anything work at all. Their curiosity and helpfulness are inspirational, and being part of this community is a great motivator for continuing my work. They have my deepest respect, and sincerest gratitude.

A great thanks to all the people - friends, colleagues, members of the community and participants at Nordic Game Jam - who showed their interest and curiosity towards my project. It has been the greatest of motivators and kept me going at a great pace throughout most of the project.
For getting started with this project, I must specifically thank Kristian and Christian - the first for sparking the initial idea on an otherwise boring bus-ride, and the second for agreeing to supervise it and showing great interest towards it.

Freddy, Jesper and Noah, a group of three, optimistic and curious fellows, were willing to join up with me and my engine for the Nordic Game Jam. I am not sure if they knew what they signed up for, or whether they regret that decision today. For you, it may just have been a fun time, but for me, it was also a very valuable experiment that would not have been possible without your aid. So, thank you for the great time, and thank you for being my lab rats.

A particular credit and thanks must be given to my partner, for having endured my thesis tunnel vision, my ups and downs and for being a fantastic moral support.

I feel many people have been involved, directly or indirectly, in this project. I have been more open about my project and better at asking for help and feedback than I believe I have ever been. Many have showed interest towards the project. So, a final thanks to all of you not mentioned here, who have helped out, reached out, listened and supported me during this journey.

Abstract

In 1994, Sony released the first iteration of their wildly popular PlayStation gaming console. While still in use by hobbyists today, the PlayStation has never seen an accessible, general-purpose game engine. One argument being that the abstraction would be too slow, as you want full control over the limited hardware. But given the advancement in techniques and development platform, we are not so convinced that this true anymore. In this thesis, we put it to the test, and explore what the technical feasibility of such an engine might be.

The core of the thesis is the development of the engine runtime *Tundra*, that can be used to make games for the original PlayStation. This is built in C++, using the open-source software development kit *psn00bsdk* as its primary library.
Besides input, audio and game loop systems, the engine provides two core systems. The first being a memory efficient *game object system* for managing game logic data. This is provided through *components*, to which logic is applied through user-defined systems that iterate them. The other being the *renderer* that supports drawing of textured and shaded 3D models, 2D sprites and text using console's hardware graphic accelerators.

The final version of Tundra was tested through two experiments: performance benchmarking of different aspects of individual features, and developing a game in a team of 4 people during the 48-hours *Nordic Game Jam*.
The first experiment showed that at 30 frames per second, the engine is capable of individually rendering between 850 - 1,750 triangles per frame, iterating $\sim$7500 - 16500 components per frame and computing $\sim$650 transforms per frame.
The second experiment showed a more holistic picture of the engine's features, as it was successfully used to develop and run a small 2-player game in 3D.

The engine has many missing features and optimization opportunities, and having run all tests in an emulator, as opposed to original hardware, there are some inaccuracies and uncertainties about the result. However, the results *did* hint at such an engine being capable as a tool for making games. Additionally, we argue that despite such a tool may lower the performance limit, it may also lower the barrier for making performant games.

Contents

Figures

Codeblocks

Chapter 1

Introduction

Almost 30 years ago, on December 3rd 1994 *Sony* released their first gaming console that they named *PlayStation* (see Figure 1.1). It was first released in Japan, and over the following year to Europe and North America [45]. It stayed in production until 2006 [6]. Despite competing with consoles like *Nintendo 64* and *Sega Saturn*, it found success, and became the fastest console to reach 100 million shipments, a record that was not broken until Sony released their next-generation console [44]. Contributing to this success were PlayStation exclusive games such as *Crash Bandicoot* [59] and *Spyro the Dragon* [55]. Sony has since released 4 additional generations of consoles, establishing themselves as one of the world's foremost console developers in modern time[1]

While not as prevalent as it once was, the original PlayStation still has the attention of many. This is apparent when looking at various online platforms, such as the *Reddit* forum *r/psx* with over 87,000 subscribers [48], the PlayStation dedicated forum *psxdev.net* with close to 3,000 members [38] and numerous *YouTube* videos related to the topic with up to millions of views.
People interact with the retro console in a variety of ways. Some are collectors of PlayStation games, and still play on the console. Some are interested in creating electronic and software modifications, as well as reverse engineering games. Some are interested in re-enacting the visual style of the PlayStation [18], and some simply seem interested in it for the sake of nostalgia. Despite them being the largest group, there are also some who show interest towards *developing games* for the PlayStation [54].

Despite there being an interest in making games for the PlayStation, the current tools are not very approachable. The only tools that are publicly available are geared towards people familiar with code, and they have no, or lacking, visual interfaces. In other words, there exists no *general-purpose game engine* for the console to the likes of *Godot* [21], *Unity* [50] or *Unreal Engine* [13]. Some suggest that such an engine is infeasible or not of interest, as this complicates utilizing the hardware to its fullest. This was especially true back when the original games were developed. Developers were skeptical of Sony's own libraries and suspecting that they were the performance bottlenecks in their games [3, p.84]. Some developers, such as Naughty Dog when developing Crash Bandicoot, went as far as using *"... as little as it could of Sony's library and the programmers basically hacked everything right to the hardware"* [1]. Back then it may have been true that the development kit provided by Sony was not performing optimally, leading developers to avoid it. However, in the 90s the tools for game development were still in their infancy, and the platforms used for developing the tools were

[1]Today the original console is commonly referred to as PlayStation 1 (PS1) to distinguish it from the newer versions, but this has never been an official title.

either expensive or not up to the job [1]. Since then, significant advancements have been made to many of the dependencies of such game development tools. This includes development in compilers and programming languages, development in software for making tools, new technological ideas and techniques, and, not least, the enormous increase in the computation capabilities of modern computers that the tools are used on. Additionally, today, we have the retrospective knowledge gained from the many games developed for the PlayStation, as well as new knowledge acquired by retro-enthusiasts.

All of these points raise our skepticism towards the argument that such an engine is not feasible or desirable. And one starts to wonder what a general-purpose game engine might be capable of, given modern knowledge, hardware and techniques. Ultimately, this raises the question that we intend to explore in this thesis:

What are the technical capabilities of a general-purpose 3D game-engine for the original PlayStation?

We approach this question by designing and implementing core parts of such as a game engine, and finally measure its technical performance and feasibility.
The scope of this project is limited, and game engines are inherently complex tools. As such, the engine produced in this project is considered only a limited prototype. However, the project also serves as a steppingstone towards a longer term and grander vision of creating a more fully-fledged tool that can reach a wider audience.

Figure 1.1: The original PlayStation released in 1994 with the original controller and memory card [53]

1.0.1 Reading guide

This report takes the reader through the design, implementation, test and discussion of the core parts of a game engine for the original PlayStation 1.

In *Chapter 2* we first establish a background of information that is necessary for the decisions and arguments made later. Given the special circumstances of the sources surrounding the PlayStation, we first give a commentary on these. This is followed by a brief a technical overview of the original PlayStation, along with an overview of some relevant tools.
In *Chapter 3* we go through the design and implementation of the most relevant features of the engine. While not elaborating on the concrete final code, we go into a fairly high level of detail. This is mostly done by comparing different options and arguing for which is best, and by iteratively improving upon certain features and mechanisms. Some features are left out because their role is either negligible or only served to aid the development of the project, such as the testing framework and build system.
In *Chapter 4* we describe the setup of the two experiments: the use of the engine at Nordic Game Jam and benchmarks of the central engine features.
In *Chapter 5* the results of the two experiments are explained and visualized. For the game

jam experiment we comment on the experience and the resulting game. For the benchmarks we argue for why we are seeing the behavior we are and relate the behavior and results to the choices made during design and implementation of the engine.
In *Chapter 6* we first discuss the future work that was not done here, which will influence the answer to the research question. This includes potential optimizations, missing features and rework of certain features. In particular, we elaborate on how the game engine can be extended with visual tooling. Based on this discussion, the results and the implemented engine we discuss the stated research question.
In *Chapter 7* we summarize the work done, the results seen, and experiences made, and finally answer what the capabilities of a general-purpose game engine for the PlayStation are.

Prerequisites

Given the amount of work that must be covered, we explain things on a fairly high level. However, designing an engine requires one to consider the low-level details, so to follow many of the arguments made and explanations given require we assume the reader to have some understanding of certain topics. This includes hardware, C++ programming, computer graphics and more general knowledge about game development from a technical perspective.
In many cases, we name certain concepts or topics that are either considered details or assumed to be known by the reader. These are not elaborated upon, and instead we refer to the sources they are accompanied by.

Chapter 2

Problem setting

2.1 Sources

Most of the research done and information used in this work has been acquired through a few sources. These sources are:

- The documentation of the PlayStation internals, originally written by *Martin 'nocash' Korth*, and further improved upon by contributors to *consoledev.net* [36]
- Personal questioning and discussions on the *PSX.Dev* [37] and *PCSX-Redux* Discord servers [33]
- Posts and discussions on the PlayStation-focused forum *psxdev.net* [38]
- The implementations and documentations of the available custom software development kits (SDK) for the PlayStation (further elaborated in Section 2.3)
- The practical analysis of the console written by *Rodrigo Copetti* [42], which is a derivative work of the other sources

All these sources are community-driven, and in cases very investigative and informal. This means that, while they are generally factually correct, they do at times contain either wrong, speculative or contradicting information, and at times point out places where information is simply unknown. Thus, as a derivative work, some information provided here may suffer from the same symptoms.
The work is in general very practical, and whether the obtained information is correct or not is easily checked in the results produced. However, we may have made certain assumptions that do not hold up that could lead to some wrong invalid conclusions. As such, this work should not be considered a fully factually correct answer to the question that it seeks to answer, but rather an estimate.

Sony publicly released little about the internal details of their console and its SDK was only released to licensed developers, including its documentation. This SDK and its documentation *can* be found online, but it is legally questionably to use, and as a result not used for any of this project.
While the community-driven sources used in this work contain new research, they also, without a doubt, *derive* from the official sources in some part. Which parts of the information find its trace back to this, is hard to say considering how the information has been accumulating and passed around for the past 30 years. Because of this, in combination with the fact that the console has been discontinued since 2006, we work see fit to make use of these sources without noteworthy risk of legal infringement.

A similar legal worry can be had concerning the use of the hardware for this work, as it is required to install modifications in the hardware (so-called *modchips*) to play non-official games on it. In the EU, it is not clear whether this is legal or not, and so this work will not be making use of it, and by extension does not make use of the original hardware. Instead, the work will make use of so-called *emulators*. These attempt to emulate the hardware on other platforms, allowing you, for instance, to run CD images from original games on a modern PC. The legality around these seem more clear, when not used for playing pirated games [5].

2.2 PlayStation's specifications

The PlayStation is an almost 30-year-old console, and many modern techniques and approaches cannot feasibly be used on it. As such, to have a clearer idea of what is, and is not possible, an overview of the most important aspects of the console is described here.

If not otherwise specified, [42] is the source for numbers and information in this section.

2.2.1 CPU

The console's central processing unit is a MIPS-based CW33300 CPU from *LSI Logic* that is binary compatible with the MIPS R3000A family, meaning it uses the MIPS I instruction set architecture (ISA). It is a 32-bit CPU that runs at 33,868,800 Hz consistently, meaning it has no throttling mechanism [37].

Coprocessors

Through the *CoreWare* (CW) product LSI Logic offered business customers the option to extend the core CPU with 4 additional *coprocessors*. Sony opted in for 3 additional processors: the *System Control Coprocessor*, *Geometry Transformation Engine* and *Motion Decoder*.

The System Control Coprocessor is in practice a non-optional coprocessor that, among other things, controls how the cache is implemented. This is further elaborated in the next section.

The Motion Decoder (henceforth referred to as the *MDEC*) is a processor for decompressing images (Macroblocks) that supports one of the console's defining features: playing full-motion video (FMV) in 320x240 resolution at 30 frames per second. In this project, this feature is left unused, and so will not be covered in-depth.

The Geometry Transformation Engine (henceforth referred to as *GTE*) is a coprocessor that primarily accelerates some general-purpose linear algebra operations such as matrix and vector multiplication and cross products, but also some more specialized operations such as mapping z-values of a set of vectors to an ordering table (elaborated later), computing lighting values based on a normal and lighting settings and performing perspective transformation [36]. The latter it can do 3 vectors at a time in 23 cycles.
While it may sound like the GTE's primary purpose is to accelerate the graphics, it is completely detached from the GPU and can be used to accelerate non-graphics related math as well, such as animation or collision.

The CPU has no support for hardware accelerated *floating point* arithmetic. Instead, the PlayStation uses a *fixed-point* format for decimal numbers, which is mostly relevant for the GTE. Specifically, it uses fixed-point numbers with an implicit scaling factor of $\frac{1}{2^{12}}$. Using a scaling factor with a denominator that is a power of two allows for exactly dividing the bits of a number between the fractional and decimal part. In this case, it uses 12 bits for the decimal-part and 4-bits for the whole part for 16-bit numbers and 20 bits for 32-bit numbers. Henceforth, these fixed-point formats will be referred to as 4.12 and 20.12 fixed-point format.

2.2.2 Memory

The console only has 2 MiB of DRAM main memory, but additionally it has 1 MiB RAM dedicated for the graphics subsystem (VRAM) and 512 KiB RAM dedicated for the sound subsystem (Sound RAM). The developer has full control over the entirety of the VRAM, but 64 KiB of the main memory is reserved for a subset of the BIOS (the *kernel*), and 4 KiB of Sound RAM is reserved for the *Sound Processing Unit* (SPU).

VRAM

VRAM is used for storing 2 things: the framebuffer(s) and texture-data. It is *dual-ported*, meaning the CPU/DMA can write to the VRAM while the video-encoder reads from it (i.e. displays something from VRAM), which allows for *double-buffering/page-flipping* [31].

Cache

There is a 4 KiB *instruction cache*, with room for 1024 instructions[1], but there is no regular data-cache. Instead, there is the *scratchpad*, which is 1 KiB of Static RAM that is mapped to a specific set of memory addresses, and from which you can do loads and stores in a single clock-cycle [11]. The developer can then store arbitrary data in this memory for fast.
While there is no data cache, there are still performance implications to how memory is accessed. It takes around 4 cycles to set up RAM access, and the load takes 1 cycle. You can do up to 4 loads on consecutive bytes without incurring the setup cycles.

Direct memory access (DMA)

To alleviate workload of moving data between main memory, VRAM and Sound RAM, the console supports *Direct-Memory-Access (DMA)* via a *DMA controller*. The controller allows subsystems, most notably in this work the GPU and SPU, to read/write main memory directly without involving the CPU, allowing the CPU to do other work in parallel.

2.2.3 Graphics

As mentioned previously, the GTE's responsibility is to accelerate certain math operations, including perspective transformations. As such, the *Graphics Processing Unit (GPU)* only operates in screen space, which is in 2D integer coordinates, and it has one role: rasterizing *primitives* to the framebuffer. The rasterization supports various features, most relevant for this project are the 3 primitive modes, *textures*, *the 2 shading modes* and *transparency*. Other features are *dithering* and *texture brightness adjustment*, but these will not be covered in depth.

Primitives

The GPU internally renders 3 different types of primitives: lines, triangles and axis-aligned quads. Lines are not utilized in this work, so they will not be elaborated upon.
Triangles can be submitted to the GPU either as individual triangles (3 vertices) or rectangles (4 vertices). However, the rectangles are subdivided into triangles by the GPU, so it is purely to save on data needing to be transmitted. This type of primitive supports all features.
It is significantly faster to rasterize axis-aligned quads than triangle-based rectangles as they are actually rasterized as rectangles, and not subdivided. However, these primitives cannot be rotated, nor smooth shaded, but they do support textures and transparency [36, GPU Render Rectangle Commands].

[1] It was not possible to find any clear sources on how this is implemented, so further detail on it is not added.

Textures

Primitives can be submitted with a set of UV-coordinates, so that the colors of its rasterized pixels will be based on an image (texture) located in VRAM instead of a single uniform color. The pixels of a texture are called *texels* and can be stored in VRAM using different *texture modes*: *true color mode* or *palette-color mode*. In *true-color* mode, each texel is a 16-bit distinct color, whereas in palette-color mode, each texel stores an index into a smaller texture that contains a set of colors. This mode has two versions, one with a 16-color palette with 4-bit index-texels and one with a 256-color palette with 8-bit index-texels.
The UV coordinates are 8-bit *texel* coordinates in the range 0 - 255 that are an index into a *texture page*. This is simply a 256x256 region of texels in VRAM. A single texture page may contain multiple textures, but given its size a single texture cannot be bigger than 256x256. Also, given that it is *texels*, its actual width in bytes depends on its texture mode, and may be either 512 bytes (true color), 256 bytes (8-bit palette) or 128 bytes (4-bit palette) wide. Furthermore, the page must be aligned to 128 bytes horizontally, and 256 *lines* vertically, which require the developer to consider how these are laid out.

Color shading

There are 2 different shading modes when rasterizing: flat shading and *Gouraud* shading [19].
With flat shading, a single color is passed with the primitive, and if it is untextured the entire primitive is drawn with that color. If it is textured it is multiplied by the given color instead. With Gouraud shading each vertex in the primitive is given a distinct color, and the color of each drawn pixel is a linear interpolation of each vertex color, depending on the distance to each respective vertex. While the primitive color can be anything, it is typically the color of the primitive including lighting effects. This can be calculated using the GTE, given the surface normal of the whole primitive, or each vertex' individual surface normal. Using the latter, with Gouraud shading, a lit object will appear smoother, and because of this it is henceforth referred to as *smooth shading*.
While Gouraud shading can be visually appealing for surfaces intended to be smooth, it is also more expensive than flat shading. Not only because of the cost of computing the interpolated color for each pixel, but also because you must compute multiple lit colors instead of just one and submit more data to the GPU.

Transparency

The GPU does not support a high range of transparency and blending through the use of an *alpha* value in colors as more modern pipelines. Instead, it supports two modes of transparency: full transparency and semi-transparency.
A color is considered fully transparent when it is entirely black (all colors values set to 0), which will simply discard that pixel. Semi-transparency is not used in this work, so we will not elaborate on that.

Framebuffer and framerate

The console support framebuffers of various widths between 256 and 640, but only heights of specifically 240 or 480 pixels. It supports either 16 or 24 bits-per-pixel (bpp), but the GPU only supports rasterizing in 16-bits, so for the intents of this project that is the color-depth of the framebuffer.
The console was released back when CRT TVs were still the norm, so how this the framebuffer is displayed is beyond the scope of this work. Suffice it to say, that each console was made for either PAL (used in Europe) or NTSC encoding, and that they had a refresh rate of 50 and 60hz respectively[2].

[2]For more information on NTSC and PAL see [20]

No depth buffer

The GPU has no *depth-* or *z-buffer*, and it has no alternative means for tracking or achieving proper order on a *pixel*-level. It does support the possibility of transmitting primitives to the GPU, via a *linked-list* mode using DMA. Utilizing this, it is common to implement a so-called *ordering-table* for maintaining a primitive order based on depth. However, ordering of primitives does not allow for overlapping polygons. These ordering tables are technically part of the SDK, and are further elaborated in Section 3.6.

2.2.4 Audio

Overall, the PlayStation's audio support is rather capable. The audio subsystem has a dedicated *Sound Processing Unit* that can process and play 24 audio channels (or *voices*) simultaneously. Audio is stored in the sound RAM and played back as 16-bit *ADPCM* samples, at a sample rate of 44.1 kHz.
The system supports various audio features such as *ADSR envelopes*, pitch modulation and digital reverb, allowing the developer to create sound variations at runtime.
Also, it supports looping sounds, but this is stored within the audio sample data, meaning a sound will automatically be looping once started if it has the proper data set. While this could be used for music, the audio subsystem has direct access to the CD controller allowing it to stream audio directly from there. This feature is typically used for music or full-motion videos.

2.2.5 CD Drive

Games for the PlayStation are stored and distributed as CDs. This is a definitive feature of the PlayStation because a CD can store between 650 and 700 MiB, which is more than 185 times more than all of its combined memory (3.5 MiB). But as the engine can function with these, and because they are relatively complex to use the CDs and the CD subsystem will not be utilized at all in this project.

2.2.6 Tools

The console itself is shipped with a *BIOS* that is stored within 512 KiB read-only memory (ROM) [36, Kernal (BIOS)]. But it is accessed via an 8-bit data bus and is so slow that some of the BIOS is relocated to the main memory. It provides various low-level functions for interacting with the hardware, all of which are exposed in assembly.
On top of the BIOS, professional developers were provided with a proprietary, closed-source PC-based software development kit (SDK) that was named *Psy-Q* [25, p.53]. This kit included an assembler, linker, debugger and various libraries providing an extra layer of abstraction on top of the BIOS. It allowed the developers to write both C for its ergonomics, and assembly for more performant code.
The SDK did not provide any visual tools like those we know from modern game engines. Instead, developers had to create their own tools, if they so desired, like to what *Probe Entertainment* did for their *Die Hard Trilogy* [12] game [17, 5:50]. The tools they *could* create were very rudimentary compared to today's standards, because of the limited power of the workstations at the time.
It was Sony's intent to provide such a library to abstract the hardware, to ease the development of games, making the console more attractive to game developers. Some developers praised Sony's efforts, such as *Namco* contributing it the reason for why it was possible for them to port *Ridge Racer* [57] in a year. But as earlier mentioned, others preferred having full control of the code running, and credited their performance bottlenecks to Sony's closed-source libraries.

In 1996, Sony released a special PlayStation console named *NET Yaroze* [43]. This console provided regular consumers with tools for developing their own *homebrew* PlayStation games, as well as playing others' games - something the original PlayStation prevented through its anti-piracy mechanisms. However, the development kit had fewer features than the professional kit, including a smaller programming library, and it also cost $750 [43], which is more than double the PlayStation's original launch price of $299 [39, p.8].

2.3 Related work

2.3.1 Emulators

Among people still interacting with the console, the most popular interaction seems to be to replay original games, without having access to the original hardware or the required peripherals, such as a monitor with proper the AV ports. As a result, it also seems like the *emulators* that make this possible are among the most popular tools made by hobbyists.
There are multiple different options in this regard, most notable are *no$psx* (closed source) by Martin Korth [22], *ePSXe* (closed-source) that was already released back in 2000 [41], *DuckStation* [24] (open-source) started by Connor 'Stenzek' McLaughlin in 2019, and *PCSX-Redux* [29] started by Nicolas Noble in 2018, which is a fork of another emulator named *PCSX*.
no$psx and ePSXe are rarely seeing updates with no$psx being last updated in 2022 and ePSXe in 2016. On the other hand, DuckStation and PCSX-Redux are seeing almost daily updates, and they have over 7,300 and 5,800 commits contributed by 97 and 56 people respectively. The two differ in that DuckStation is targeted towards users who seek to play emulated games, whereas PCSX-Redux provides more extensive tooling for developing and reverse-engineering of games, with tools for debugging of code, memory and hardware internals.

2.3.2 Development tools

In regard to the tools for developing games, the options are sparser. Many seem to either build their own small libraries using little more than the BIOS functions or use the original Psy-Q SDKs. There are, however, 2 noteworthy SDKs: *psn00bsdk* started by *Lameguy64* [51] and *PSYQo* [34] developed as part of PCSX-Redux. Both projects are open source, and relatively new, with first commits being made in 2019 for psn00bsdk and in 2022 for PSYQo. This also means that neither of the projects have been heavily used or matured, and at the time of writing, neither is seeing highly active development.

psn00bsdk is written in C and is a re-implementation of the original Psy-Q SDK, meaning it provides very similar APIs, but a custom backend. It provides a somewhat thin layer on top of the underlying hardware, such as APIs for drawing primitive types, using GTE, input and the CD drive. *PSYQo* on the other hand, is a completely custom SDK written in modern C++, but still focused on being lightweight. Beyond providing many of the same PlayStation specific features as psn00bsdk, it provides various programming ergonomics such as fixed-point and vector types, some math utility and support for C++ co-routines. Also, it includes *EA Games' Standard Template Library implementation (EASTL)* [2], providing useful container types (dynamic arrays and associative arrays), a string type and various other commonly used C++ features.
Crucially, both SDKs include some utilities for building projects, in particular a *compiler toolchain* that builds the source code targeted for the PlayStation's *MIPS R3000 chipset*. For setup, psn00bsdk uses *CMake*, and PSYQo is most easily available through the *PSX.Dev* [16] extension for *Visual Studio Code* [27]. This extension installs all the necessary tools, includ-

ing a C++ language server and debugging setups that automatically build and attach to the PCSX Redux emulator.
Both development kits are software-focused, and neither of them provides any *visual* tools dedicated for PlayStation game development. Also, neither of them contains extensive support for asset management, in particular any format for including or rendering 3D models. However, there do exist various tools for converting assets in more common formats such as audio files in *.wav, .mp3* etc. and images in *.png, .jpeg* etc. to formats more suited for the PlayStation. Loading and using these are still up the programmer in both environments.

Chapter 3

Engine implementation

The general-purpose engine that is used to test the research question is named *Tundra* and is custom-made for this experiment. The hardware is ultimately the limiting factor of what is possible on the console, but how the engine is designed and implemented to utilize the hardware has a significant influence. This chapter therefore describes how the engine is designed, and why it is designed like it is.

Popular known modern engines such as Unity, Unreal Engine and Godot are often synonymous with their visual *editor*. However, an engine can be considered in two parts: the *runtime* and a suite of *tools* [15]. The runtime is the code that the actual game utilizes to run, whereas the tools is what the developer interacts with (i.e. the editor). In other words, the tools run on the developer's machine, whereas the runtime run on the user's machine. In our case, the user's machine is the original PlayStation. So, while the runtime is limited by the almost 30-year-old hardware, the tools have, in large part, the same potential as the tools of modern engines. We assume the runtime to be the limiting factor, so to limit the scope of the work the engine we implement here only implements the runtime. Some remarks about future potential inclusion of tools are discussed in Section 6.2.1.

In Appendix E is included a full source code example of how the final version of Tundra is used, and Appendix D includes more concise examples of specific systems and APIs.

3.1 Tools

3.1.1 Programming language

Before even considering the tools to use, a choice was made about the programming language. The choice was between C and C++, because that was what the available SDKs used. The original Psy-Q SDK and all games were made with a combination of C and assembly. C++ existed at the time, and despite its features providing various ergonomics for programmers, there was an aversion towards the language. This was because of a belief that it was not fast enough when you wanted to fully utilize the hardware.
Today the language and the compilers for it have developed and matured, and the ergonomic features have increased significantly. A testament to this is the fact that modern game engines are almost exclusively written in C++.
It is true that some features and abstractions *will* cost CPU cycles and memory, but not compared to implementing something equivalent in C. Examples of this are the specialized type member functions, *constructors* and *destructors*, that are automatically called when an object's lifetime is started and ended (commonly known as *resource acquisition is initialization*

or *RAII*). Calling these functions has additional overhead, but not compared to calling a special function in C that ensures an object is properly cleaned up before deallocating it.
For most C++ features, you do not pay for what you do not use, such as RAII. However, for some features you do pay, such as the *run-time type information (RTTI)* feature. This allows you to safely determine if a pointer or reference points to an object of a specific type, which is, for example, useful for *downcasting* a base type to a derived type. But it embeds additional data for every object, regardless of whether it is used or not. This feature and its overhead can be entirely disabled at compile-time, however.
Additionally, even if certain features have an overhead, such an overhead is only an issue if used in frequently executed code or instantiated objects. So, in such cases, we can simply avoid using them.
Based on these arguments, C++ is chosen as the programming language for the engine. However, it is with the caveat that RTTI is disabled.

3.1.2 Software development kit

There are 3 options of interest for the choice of SDK: psn00bsdk, PSYQo or nothing. Psy-Q is not considered an option, because of the questionable legal use of it.
The engine is itself an abstraction over the hardware and kernel, so to the end-user it should not be visible in the API or workflows whether an underlying SDK is used or not. But by building the engine on top of an SDK, we end up with two levels of abstractions over the hardware and kernel. Extra abstractions may cost extra resources, since the intention of an abstraction is to hide certain details that could otherwise be utilized knowing the details of the use-case (the engine).
In the long-term it may be of interest to forego an underlying SDK and do a custom implementation of the hardware and kernel interaction for the engine, to only get the behavior required, and nothing more. However, at the start of this project we had little knowledge of how the kernel and low-level internals of the PlayStation worked, and thus little idea of the extent of the SDKs abstractions that would have to be custom-made. Considering the tight timeframe of the project, the idea of not using an SDK was abandoned early.

Given PSYQo is an SDK in C++ it may have seemed the obvious choice, given the engine will be implemented in C++. However, there were a couple of things that argued against it.
For one, while psn00bsdk is still lacking certain features, it was considered the more mature and developed of the two. The SDK is 3 years older, describing itself as *"The most powerful open source SDK for the PS1"* [51], and the fact that certain forums only listed psn00bsdk [52].
Additionally, PSYQo seemingly impose a more significant abstraction than psn00bsdk, such as required *application* and *scene* concepts that handle the update-loop. Psn00bsdk on the other hand, only provide features to interact with the bios and hardware, but how and when this is used is entirely up to the program. It was not entirely clear whether PSYQo's more significant abstraction would pose a problem or not, but we saw it as a risk. Ultimately, psn00bsdk was the chosen SDK.

3.2 Platform

As mentioned in Section 2.1, because of legal reasons, this work will not be tested using the original hardware, but instead utilize emulators. In particular, the PCSX-Redux emulator will be used because of its debugging capabilities, and ease of integration with the *Visual Studio Code* editor. While there is a risk of certain mechanisms not working in the emulator like they do on the hardware, we found this to have no implications for the functionality of the

engine. The influence of using the emulator for testing gathering results, is further discussed in Chapter 4.

3.3 Utilities

While it is possible to use C++ with the SDK, the SDK's APIs are themselves written in pure C. This also means the SDK does not provide any implementation of the *Standard Template Library (STL)*. So, given C++ is available, we realized early that there is considerable value to be had by implementing certain programming utilities. In particular, a generic, dynamic list type, a string type, a fixed-point number type and types for vectors and matrices.

3.3.1 List

We implemented a simple dynamic array type, akin to the standard's *vector* type, with the name *List*. The container is implemented as a *template* type, allowing it to store arbitrary elements of arbitrary data types while still providing type-safety. Under the hood, it is nothing but a regular C-style array along with a number denoting its current size. When elements are added to the list, it automatically grows the underlying arrays using geometric growth to ensure asymptotic linear insertion time.

Other container types, such as stacks, queues or associative arrays are more specialized, and less useful in the broader scope. These were not implemented due to time constraints, and we found no need for them during development.

3.3.2 String

The string essentially serves the same purpose as the list, in that it is an extension of C-style strings that allow for dynamic growth by adding together two strings. For any other purpose, there should be no theoretical difference when using this type.
This is entirely an ergonomics feature, as it is not using geometric growth, but only ever allocates space required for the two strings added together. Thus, it is not efficient for complex building of strings, but that is not its intended use-case, and not a use-case expected to be common for the engine.
The string is mostly used for debugging purposes, but also for a few game features, such as rendering text (further elaborated in Section 3.6), where its performance is not of a significant concern.

3.3.3 Fixed-point

As earlier mentioned, the hardware does not have support for *floating point* operations, and instead the GTE and SDK use fixed-point numbers. Neither C nor C++ has any built-in notion of fixed-point numbers, but it is possible to support fixed-point operations by storing numbers in an integer and using some trivial integer operations.

Fixed-point as integer formats

The fixed-point number is stored as an integer (either signed or unsigned) where we implicitly assign a number b of the least-significant bits for the decimal, and the remainder bits for the whole part of the number (see Figure 3.1 for an example). The whole part is calculated as a regular integer, and the decimal part is calculated as

$$i_{-1} * 2^{-1} + i_{-2} * 2^{-2} + \cdots + i_{-b} * 2^{-b}$$

where i_{-1} is the most significant bit in decimal part, and i_{-b} is the least significant bit. A simple way to calculate the whole number is to scale the integer value of all the bits by a factor $\frac{1}{2^b}$.
As earlier mentioned, the console and SDK use $b = 12$.

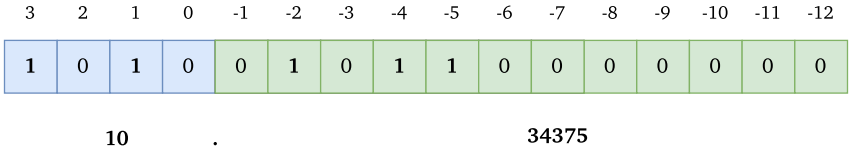


Figure 3.1: Example of the 16-bit fixed-point number where $b = 12$. The number above each bit is the bit-index, with the decimal bits having negative indices.

Addition and subtraction of two fixed-point numbers stored as integers, can simply be done by using the regular integer subtraction and addition.
Multiplication and division are slightly more complicated in that using the result of using the integer operation uses a different implicit scaling factor than the operands. To be exact, its scaling factor is the product of the scaling factors of the operands when multiplied, and the quotient of the scaling factors when divided. This means the result of multiplying and dividing two numbers in format $a.b$ yields a product in $a + a.b + b$ format and a quotient in $a - a.b - b$ format. In practice this means that the result of multiplying 2 numbers of format 20.12 must be bit-shift to the right by 12 to get the result in 20.12 format, and bit-shift to the left by 12 when divided. An important thing to note here is that the result of such a multiplication is therefore in a format that require more bits to represent. For example, multiplication of two 20.12-format numbers results in a number in 40.24 format, i.e. a 64-bit number. Thus, avoid the loss of significant precision, the multiplication must be cast to a type that has a higher number of bits before performing the operation, and then cast back to the original type. Doing this is not for free, as the hardware does not have support for 64-bit operations, being a 32-bit console. Instead, it must simulate such an operation in software.

Template type

Having to represent fixed-point numbers as integers, remember the scaling used, and manually perform this casting and bit-shifting for every multiplication or division operation is both cumbersome and error-prone. Also, it is not something done in modern engines where they use the floating-point format for decimal numbers instead, and as such it is not something many are accustomed to.
To alleviate these problems, we implemented a custom fixed-point type using some of C++'s template features and operation overloading mechanisms. The type is parameterized with an underlying integer type (32-bit vs 16-bit and signed vs unsigned) and a number of bits for the decimal part. It allows you to use regular addition, subtraction, multiplication and division operators as if they were regular integers or fixed-point numbers, and do conversions between formats. Under the hood, it will perform the casting, bit-shifting and necessary scaling conversions to do this.
By enabling software-emulated floating points numbers it is possible to use regular double and float *literals*. You can then use these to initialize fixed-point objects by enforcing constructors supporting the literals to run at compile-time using the `consteval` modifier.
Crucially, because the type and all its functions are parameterized with templates, the compiler has the same information for the operations as if you had done the operations manually. Thus, in theory, there should be no performance overhead in using these types.

3.3.4 Vectors and Matrices

While psn00bsdk does have types for vectors and matrices, they are based in C and thus have two ergonomic issues: they have no math operator overloads, and they only come in the pre-specified types, which are raw signed/unsigned integers in 16- or 32-bit.
There exist many C++ libraries for matrices and vectors such as *Eigen* [4], but they are big and complicated as they are cross-platform, and they typically also come with many more utilities than simple matrices and vectors. They are also often designed to utilize hardware SIMD instructions, such as *AVX* [23], but that is not supported by the PlayStation, so no benefit is to be had there.

At the same time, such classes are somewhat trivial to implement for basic use-cases, so that is the approach we take here. We provide a 2D vector, 3D vector and 3x3 matrix. The type of their underlying elements is templated, meaning they support both integral types and the fixed-point type, without the need for customs implementation for each.
The PlayStation may not support common SIMD instructions, but it does have the GTE. This allows us to accelerate certain operations such as normalizing a vector and multiplying 3x3 matrices. However, instead of overloading the operators, these are exposed as free, global functions. This is primarily to make their use more explicit, as they will override certain registers that, for example, are used by the renderer. But also, those functions only operate on certain element types, and we want to make it clear what is accelerated and what is not.

3.4 Game object system

Any modern game engine has some concept of a *game object system*. It is often exposed in some kind of tool that allows the user to place, compose and customize objects in a 2D or 3D world. In many modern engines it is also means of adding, compositing and customizing *behavior* in the game.
For example, one might drag in a *model* of a tree into the game world, and add a *collider* to it, so you cannot walk through it. For added effect you may add a *particle emitter* that makes leaves fall from the crown. The user may decide that in their game the tree catch on fire if you walk too close with a torch, so they implement a *custom behavior* to do this, which they add to their tree.
It varies how different engines implement this kind of behavior. In some engines it is implemented with two concepts: some central object without any inherent behavior and only meta-data, onto which you can attach *components* that carry more concrete data and behavior, including user-specified behavior. This is the case in *Unity* with their *Game Objects* and *Components*, and *Unreal Engine* with their *Actors* and *Components*.
In other engines, there is no concept of components, but rather you compose game objects together. This is the case for Godot's *Nodes*, and *Entities* within *IO Interactive's Glacier* engine [32].
The runtime implementation of the game object system is meant to map the behavior advertised in the tooling into actual behavior in the game. While it *can* be completely different, it is arguably simpler if they map fairly directly. It is also easier to argue that a runtime implementation that is akin to what is intended to be exposed on the tools side, is easier to extended with tools. So, despite this work not considering the tools side, the runtime implementation is also based on a game-objects approach.

3.4.1 Features

Based on how other game-object systems there are some features that we want our game object system to have:

- *Custom update logic on game objects*: often, we want to simulate some game logic by updating game object data every frame. In this work, we call this game logic *update logic*. It is unrealistic for a general-purpose game engine to implement all thinkable update logic that a developer may need. Thus, it should be possible for developers to exert their own logic onto objects.
- *Grouping objects together*: It is useful to be able to speak of a series of objects as a coherent game object. For instance, when a projectile consisting of some model, some physics, a sound and a particle hits its target, it is convenient to delete everything related to the projectile by just deleting the projectile.
- *Dynamic references to game objects*: Some objects may want dynamic references to other objects. For instance, an enemy object may want to know what game object it is currently targeting.
- *Memory consumption*: There may exist *many* objects in the game at one time, and the hardware's memory is limited. Thus, memory overhead of game objects should be kept minimal.

The remainder of this section describes how these features are approached.

3.4.2 Update logic

A common approach to game-object systems is to utilize an *object-oriented* approach. In this paradigm, every type of object has its own update logic, and the logic operates on a single instance of the type in isolation. Thus, the logic is executed for every instance of an object type. For example, in Unity, every component has a non-static `Update` member function, and this function is run on every component in every frame. The developer can thus extend their own custom components with custom behavior by implementing it in this function[1].
A simple way of implementing this is to have all game-objects inherit a base class with a *virtual* method that can be overridden by the user with their own custom logic. Then all game objects can be stored in a single list as their virtual, common base-type on which the virtual update method is called. See Codeblock 3.1 for an example of this approach.

This object-oriented approach poses two problems: virtual method calls cause an extra memory indirection making such calls more expensive, and it interleaves the update-logic of different game object with each other. The latter is a potential problem given the CPU's 4 KiB *instruction cache* mentioned in Section 2.2.2, from which update-logic instructions may be evicted by other update-logic instructions when interleaved.
Both issues can be avoided if we instead provide a way of iterating all objects of the same type. If the type is known when iterating objects, then a non-virtual member function on the concrete types can be called, avoiding the indirection overhead of the virtual function call. Also, if we are updating objects in groups then update instructions are also grouped together by type, which more efficiently utilizes the instruction cache. In one experiment, we found that iterating components ordered by type rather than being interleaved yielded a speedup of 1.4.
Exposing this *typed* iteration of game objects as an automatic feature that acts like a virtual approach, requires the use of some C++ template features, and some type registration, which is non-trivial or non-automatic. As such, we only implement the support for iterating objects of a specific type in this work, and consider the per-object update implementation a specialization of this that it could potentially be extended with.

[1] There are other functions that one can implement such as *Start* and *Destroy*. Collectively these functions are called *event functions*.

```
// Engine
class GameObject {
    virtual void on_update() { }
};

class GameObjectA : public GameObject {
    void update() override { /* Some custom logic */ }
};

class GameObjectB : public GameObject {
    void update() override { /* Some custom logic */ }
};

List<GameObject*> all_game_objects;

void frame_update() {
    for( GameObject* g : all_game_objects ) { g->update(); }
}
```

Codeblock 3.1: Simplified C++ example of applying custom logic on game objects by using an object-oriented approach and virtual methods.

Leaving out the feature of providing update logic through a member function, the game object behavior is intended to be applied by having *systems* that execute their logic every frame. These systems have the opportunity to iterate all game objects of a given type. There are no requirements on how these systems are implemented by the user, and they do not exist as an explicit concept in the engine. They can simply be functions that are called from the core update loop.
This simpler approach also a few advantages. For one, it allows multiple systems to iterate the same objects, iterate objects of different types, and iterate them as many times as desired. An example of this is given in Codeblock 3.2.
Also, it allows systems, and thereby object logic, to have non-static dependencies that would otherwise have to be copied into every object, consuming excessive memory. This also allows for *dependency injection*, which can make it easier to isolate code and to do unit testing.

Given the approach of iterating components using systems, one may think that this is leaning towards the currently popular approach of *Data-Oriented Design (DOD)*, or the *Entity-Component-System (ECS)* approach in particular [35]. However, while the mechanism is similar to the *systems* in ECS, ECS seeks additional advantages that cannot be obtained on the PlayStation hardware. For one, it seeks to lay components consecutively in memory and keep them small as possible to efficiently utilize the CPU's *data-cache*, which can have significant performance benefits. Also, it seeks to have systems being explicitly dependent on certain types, which allows for ordering of systems into an update graph that can be parallelized. To better utilize both mechanisms, ECS encourages the developer to keep components small and isolated.
The PlayStation cannot leverage these advantages because it is a single-core CPU without any data-cache. But not being able to utilize this relaxes the requirements and provides us with some additional flexibility. Notably, we do not need to worry about keeping components smaller than what makes sense for separation-of-concerns, and we need not worry about thread safety.

```
class GameObject { /* Data */ };

template<typename TGameObject>
List<TGameObject>& get_all() {
    /* Some way of getting all game objects of type TGameObject*/
}

class GameObjectA : public GameObject { /* Data */ };
class GameObjectB : public GameObject { /* Data */ };

void some_system() {
    for( GameObjectA* c : get_all<GameObjectA>() ) { /* Arbitrary code */
    }
}

void some_other_system() {
    for( GameObjectA* c : get_all<GameObjectA>() ) {
        /* manipulation of all objects of type GameObjectA */ }

    for( GameObjectB* c : get_all<GameObjectB>() ) {
        /* manipulation of all objects of type GameObjectB */ }

    for( GameObjectB* c : get_all<GameObjectA>() ) {
        /* More manipulation of  all objects of type GameObjectA */ }
}
```

Codeblock 3.2: Simplified C++ example of applying custom logic to game objects via iteration and systems.

3.4.3 Grouping objects

To group objects together, and for all objects to know they are grouped together, there must be some reference from all objects in a group to all other objects in the same group. A simple approach is for all objects to have a reference to the same *group* object, and have that object reference all objects in the group, as is illustrated in Figure 3.2.

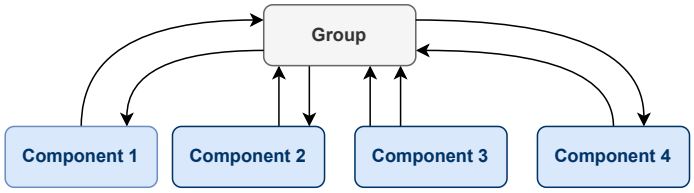


Figure 3.2: Example of grouping components by each component having a reference to the group, and having the group reference each component.

The problem here is that for every object in the group we need two references - one from the group to the object, and one from the object to the group. We can reduce this to a single reference, if the objects in a group are chained using a *circular singly-linked list*; each object references the next object, and the last object references the first. This is illustrated in Figure 3.3.
One downside with this is that finding the previous sibling of a component is now an operation with linear time complexity. This is primarily an issue when deleting a single component, because we must redirect the pointer of the sibling that pointed to it. However, deleting the entire group, which is expected to be a more common use-case, all components are iterated anyway, and no pointers must be fixed.

This solution does not require a central ”group object”, but such an object can still be useful. For example, it can serve as an object to store state that all objects have in common, such

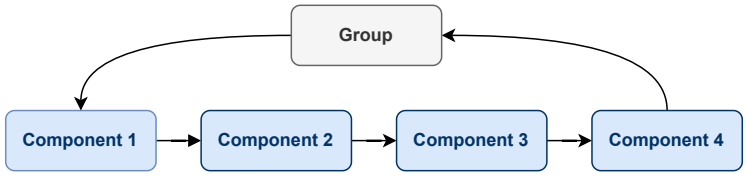


Figure 3.3: Example of grouping components in a chain with the group referencing the first component and last component referencing the group.

as meta-data. To support this, we introduced the group object as the *entity* concept, and the objects within the groups we call *components*. Therefore, the game object system is henceforth synonymous with the *entity system*.
The components do not have an explicit reference to their entity. Instead, the entity is in fact just a specialized component which exists in the same linked list as the components in the entity. Thus, you can iterate this list if you need to find all components of an entity, or if you need to find the entity of a component. For the latter case, we must be able to tell that a component points to an entity. Instead of enabling RTTI globally, we implement a similar behavior specifically for this scenario, by having a virtual method on components that return whether they are actually an entity. The overhead of this virtual method is not considered a problem, as this use-case is expected to be rare.
Despite it being implemented as a concept, the entity was not fully utilized for its intended purpose in this work. Most notably it had no meta-data to speak of.

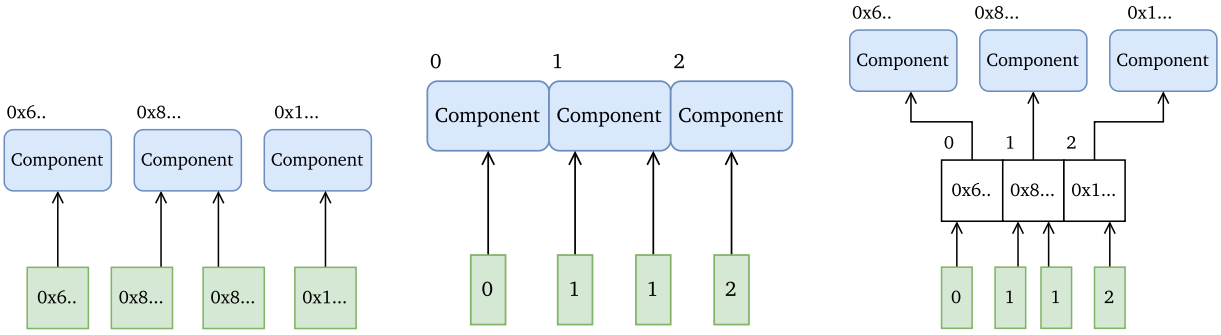
Besides grouping basic components into a coherent entity, you may also find that you want to group *entities* together. The current approach deliberately does not provide a required and built-in parent/child relationship because of the memory overhead it would introduce for components that do not need it. Also, it is easy to provide this on a case-by-case basis with regular references, or by having a specific component child component.

3.4.4 Component references

Because RTTI is disabled there is no simple and type-safe way to get a component of a specific type on an entity. This means that if a component wants to use another component on the same entity it must have an explicit reference to the other component. This is assumed to be a likely use-case, and so it is expected that references to components will, and should, be heavily used. In turn, they are expected to have a significant impact in both memory usage and cycles, and it is our desire to limit both.
One simple approach to this problem is for the references to be regular pointers (see Figure 3.4a). This provides the fastest way of dereferencing, as it stores a direct address to the memory where the component resides.
But a regular pointer is a 32-bit value as it must be able to reference an arbitrary byte in the entire address space[2]. In total, it allows us to reference more than 4 billion distinct values, which is significantly more than we expect to need.
Alternatively, one could use *indices* into a container, such as a contiguous array, where the component is allocated (see Figure 3.4b). The index could be of a smaller size, such as 16 bits with you can still reference about 64,000 distinct components. But it would require additional memory access and operations to dereference the component as you first have to read address of the list, and then the actual entity.
Both approaches have the downside that the reference is static, in that the component cannot be moved in memory without the reference becoming invalid. Moving entities can be useful to optimize how they are stored as components are destroyed and new ones created. Making

[2]Memory is 2 MiB, so you need $\log_2 2097152 = 21$ bits to address every byte on the PlayStation.

components dynamic in memory can be achieved by mapping from some unique component ID into an actual location that is defined by either an index or memory address (see Figure 3.4c). While such a mapping can be implemented in constant average time complexity, it is still an expensive operation. Also, the mapping would mean additional memory overhead as you would now need to store an additional pointer for every component.
Ultimately, we decide to use regular pointers as our component references because of its simplicity and because they are the fastest to dereference[3]. Where and how the components are stored is discussed Section 3.4.5.



(a) Raw pointers to components **(b)** Indices to list of components **(c)** Indices to map of raw pointers

Figure 3.4: Three approaches to component references. Blue rounded boxes are components, green boxes are the component references, white boxes are other data, and arrows indicate a memory reference.

Dangling pointers

A common issue in programming languages that rely on explicit allocation and freeing of memory, such as C and C++, is that of *dangling pointers*. These are pointers that point to invalid memory, because the object that they think is there has been deleted. They occur because code with such pointers may have no mechanism to be notified when the object is deleted. This problem is also common in game object systems that often rely on manual deletion and creation of game objects at some rather arbitrary points in time. For example, an enemy game object may have a reference to its target player game object. If the player dies and is destroyed then the reference the enemy's reference to the player is no longer valid, but it has no way of knowing this.
In Unity, dangling references are handled by a custom, overloaded equality operator on the game object [26]. You can thus check if the referenced object is equal to null, in which case it is either destroyed or was never set.
The problem also exists with our system, and its significance makes it worth alleviating. To solve it, we take an approach similar to Unity's, but also to the C++ standard's concept of *weak pointers* [46]. Essentially, all components store a *reference counter* as part of their meta-data, and every reference to a component increment this counter. When a component is destroyed, it is considered destroyed immediately, but its memory is only deallocated (or *freed*) once all references have been cleared (see Figure 3.5)[4]. This allows references that would otherwise have been dangling pointers to properly signal that the component they referenced is no longer valid.
This implies that components can be in either of three states: alive, destroyed or free. Destroyed mean that the component has been destroyed by some code, but it is not yet freed

[3]Despite them being called *references* they are semantically and syntactically more similar to C++ pointers than C++ references. This misnomer is intended to be rectified in a future revision.

[4]Its destructor is not called with the current implementation, but the component has the option of overriding an *on_destroy* method where it can run destruction logic. Designs are in place to rework this such that the destructor is called when the component is destroyed despite it not being freed, so that it properly supports RAII.

because there are references to it (its reference count is not 0).
The reference can be implemented using C++'s RAII and operator overloading features. This allows for the reference counting to be handled behind the scenes, and to the user, the reference operates the same way as a regular pointer.
This solution implies that a dereference must do a validity check, which costs extra operations. But this is slightly alleviated by the fact that the dereferenced pointer can be cached and re-used within a block of logic that does not change references, and that the memory prefetching mentioned in Section 2.2.2 may make dereferencing of fields within the component faster.
The memory cost of this feature is miniscule, as a reference counter of 8 bits that allows for 255 simultaneous references is considered to be enough. The reference itself does not need to be extended with any extra data.

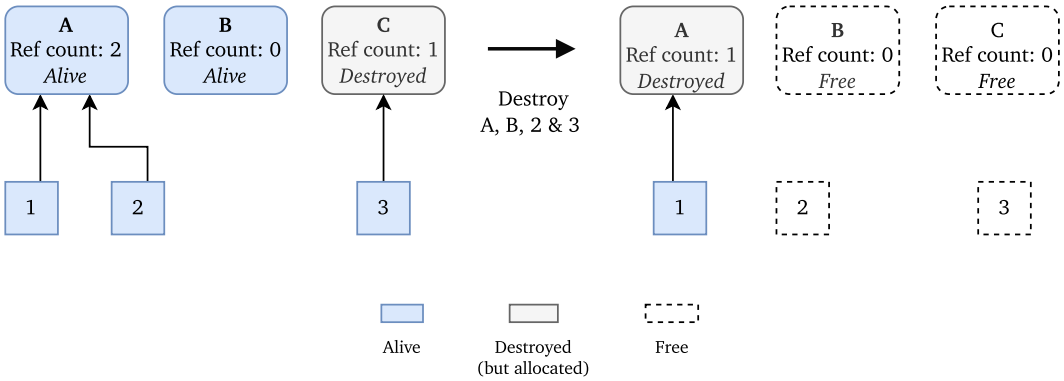


Figure 3.5: Example of how component references keeps components allocated while they have a reference to them. Small numbered boxes are references, big boxes are components.

3.4.5 Storing components

The core problem of the game object system is to store the components so that the desired features of fast iteration, static memory locations and low memory overhead is achieved. At the same the construction and destruction of components should be fairly efficient.

Disregarding any performance or memory overhead, the simplest approach would be to allocate the components individually using the default `new` and `delete` operators that perform individual memory allocations of arbitrary size on the heap.
In the SDK's implementation, dynamic memory is managed as a linked list of interleaving free and allocated blocks of memory. Allocating memory is thus a lookup into this linked list, which has linear time complexity, and allocating many small components can easily make it very slow. Memory allocations also involve a significant memory overhead, as the allocated blocks include 16 bytes of meta-data, beside the allocated memory.
Additionally, performing allocations of various sizes in the same buffer can result in *external memory fragmentation* [40, p. 16.6]. Essentially, this may potentially break up the available memory into many small unusable memory blocks.
Allocating memory on the PlayStation does not, however, entail a *system call* which is commonly given a large part of the blame for making dynamic memory allocation slow. But the fact that this is not here does not offset the other problems.

The base of the approach we go with instead, is to allocate components in a dedicated, pre-allocated and contiguous array of bytes. This array is split into equally sized *entries* with the same size as the type of component that it stores, and an entry may either contain an allocated component or free space. When allocating a component we allocate it at the address

of an entry using C++'s *placement new*[5].
All the data needed to signal the three states that a component entry can be in is a set of flags, which can be stored in a single byte. Thus, this approach has less memory overhead, and completely avoids external fragmentation.
While simple, the approach has three main issues: the array is fixed in size, holes are slow to iterate and finding free entries is slow. We tackle each of these problems in the next sections.

Allocating in blocks

The first problem is that we cannot adjust the size of the array. Doing this would mean we must move the components in memory, breaking the criteria of components being static in memory.
This problem can be solved with a rather simple approach. Instead of allocating a big array upfront that must be big enough for any scenario in the game, we allocate multiple smaller arrays. These arrays, along with some meta-data, we call *registry blocks* and the system tracking the blocks is called the *registry*. Note that there is one distinct registry with a set of blocks per type of component.

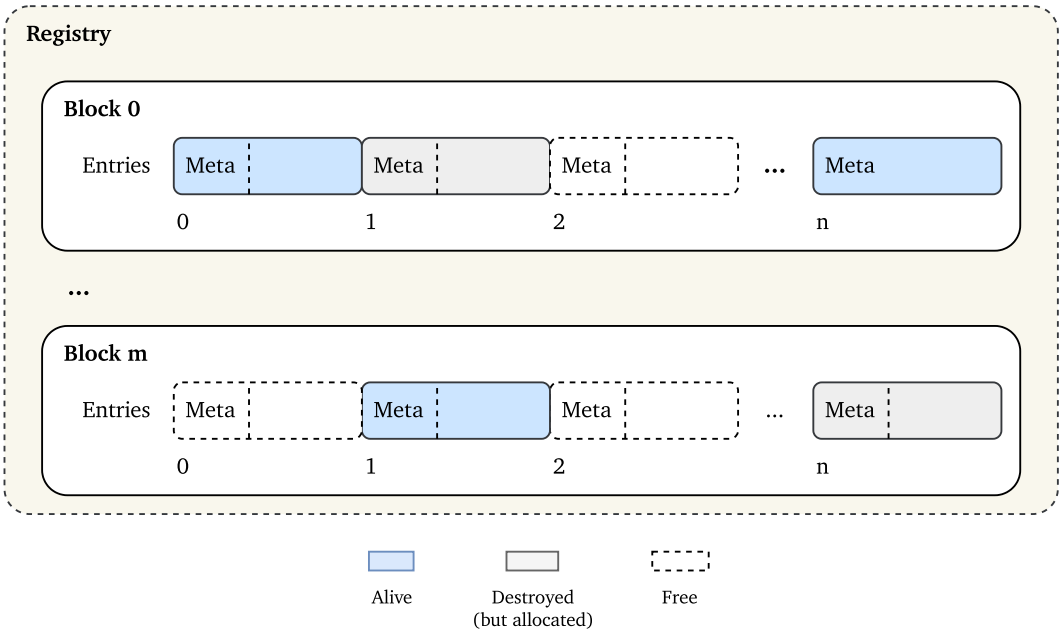


Figure 3.6: Structure of a registry for some component type with m blocks, each with n number of entries that are contiguous in memory. Every component has some metadata (left part of an entry) and some data specific to the component type (right part).

We only allocate one block up front and allocate additional blocks as the current ones become full. The individual blocks are still static and contiguous in memory. This does introduce an additional overhead for construction and destruction, and it does introduce a memory overhead of 24 bytes for the block data. However, this memory overhead is essentially split across all components in the block.
When freeing a component we must update some data in the block, such as the number of free entries in the block. This means that we must locate the block that the component resides in. While this can be done by just looking through the blocks within the registry, it is an operation of linear complexity with respect to the number of blocks in the registry. Instead, extending the component's meta-data with a single byte block index, makes it possible to

[5]Starts the lifetime and calls the constructor of an object with the given type at the given memory address [10, Placement new].

bring this to constant time, and was found to reduce destruction time when 50 blocks were allocated.
The same linear complexity is present when allocating components, because we must search for a block with a free entry. However, we can store an additional list of indices of blocks that have free entries within the registry. This list can be maintained in constant time. When a component is freed in a block, and it is the first freed component in the block, the block is added to the list. When creating a component allocate it in the last block in the list, giving us constant time lookup and constant time removal of a free entry.
So, in general, the extension of using blocks has constant time complexity. This does not mean it is free as the operations and memory access still has a cost.

Jumping holes

The second problem occurs when we want to iterate all allocated components. Doing this requires us to iterate *all* elements in the component array, even if they are free, because we do not know that until we have checked. This can be expensive if, say, only the last component is allocated.
This problem can be solved by doing some additional management for these series of contiguous, free entries or *holes*. On the first entry in a hole (the *head*) we store an index to the last entry in the hole (the *tail*) (see Figure 3.7). This allows us to skip to the next allocated entry when encountering a free entry during iteration. In other words, we can jump over the hole in constant time.

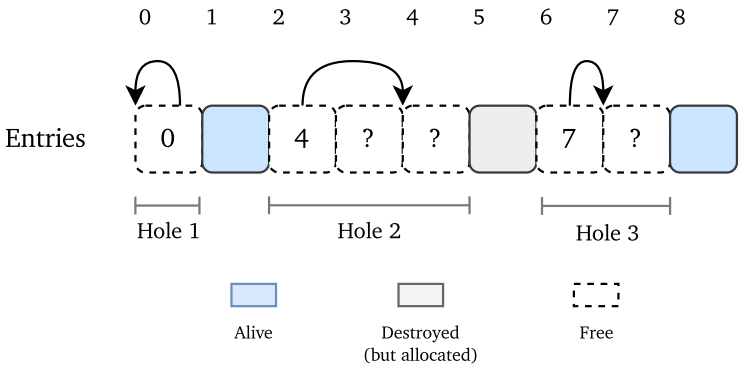


Figure 3.7: Entries in some registry block showing references from hole heads to hole tails allowing us to jump holes in constant time. The number in the entry show the index that the arrow references. Entries are simplified here.

When allocating new components, the hole is simple to adjust, as long as the allocated entry is the head of a hole. All it takes is to make the next entry the new head by pointing it to the tail, or do nothing if the allocated entry was the last in the hole. In both scenarios it is a few simple operations in constant time.
To achieve the same time complexity when freeing components, however, we need some additional data. Consider the case where the component just after the tail of a hole is freed. We can easily check if there is a hole in front of a freed element, but we must update the head's tail-index, and finding the hole's head means we have to look through all entries in the hole.
The trick is to utilize the same index variable on the tail that the head is using to keep an index to the head. This way we can expand the hole in either direction, and because we only ever manipulate either the head or tail of a hole, these two indices are enough to maintain the holes in constant time.
The hole data is only ever required for entries that are free, and those entries do not use most

of their memory. So, instead of extending the component with additional meta-data, we can utilize this empty space to store these additional hole indices. This way, the hole structures will not induce any additional memory overhead.

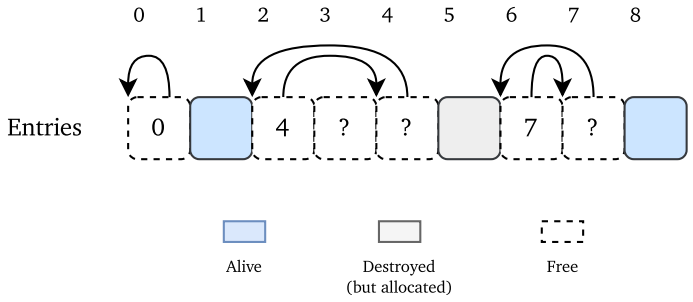


Figure 3.8: Extension of Figure 3.7 with references going from tails to heads, allowing us to jump holes in both directions in constant time.

Finding free entry

The final problem is that within a block, finding a single free entry is potentially linear in time complexity, as the only free entry may be the last entry in the array. But every entry e in a block has a static index i_e, which is its index into the array of entries, so a simple solution is to maintain a *free-list*: a stack or queue of the index of all free entries in the block[6].
When allocating a new component, we pop the topmost index i_e in the stack and allocate our component in the entry e that the index corresponds to. When freeing an entry d we just push its index i_d to the stack.

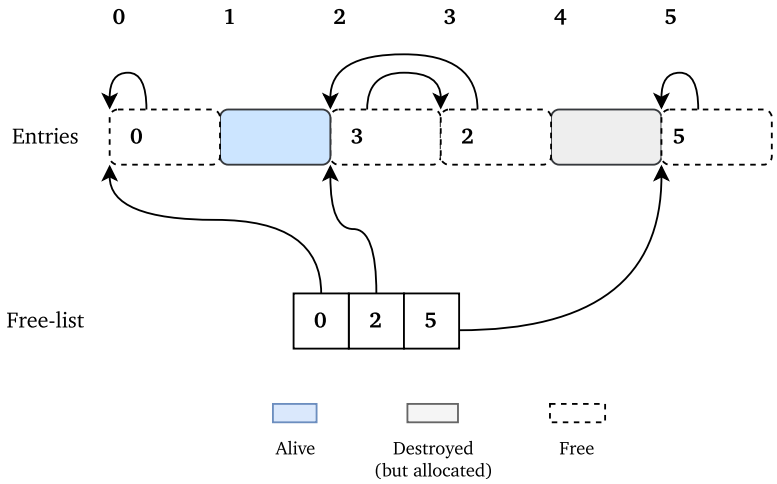


Figure 3.9: Visualization of the free-list within a block with an index of the head entry of every hole in the entry list.

This solution can be improved slightly by utilizing the concept of holes described in previous section. Instead of storing the index of every free entry, we only store the indices of the hole heads (see Figure 3.9). Thus, when allocating we take the top hole index in the stack, and if the hole contained only one entry, it is popped from the stack, but otherwise it stays.
Allocating is thus a constant time operation, and so it is when freeing, if both the previous and next entries are still allocated. But is not constant in the two following cases where we free a component c with index i_c (see Figure 3.10 for visualization):

[6] In code, this stack is implemented as list from which we only add/remove the last element

1. **Entry h after c is the head of a hole**: the hole head becomes the entry we just freed, meaning the index i_h that was stored in the free-list as the hole head is now invalid, as it should now be i_c.
2. **Before c is a hole with head entry g, and entry h after c is the head of hole**: we want to merge the two holes, and thus remove i_h from the free-list. c lies within the new hole and is neither head nor tail, so it plays no part in this.

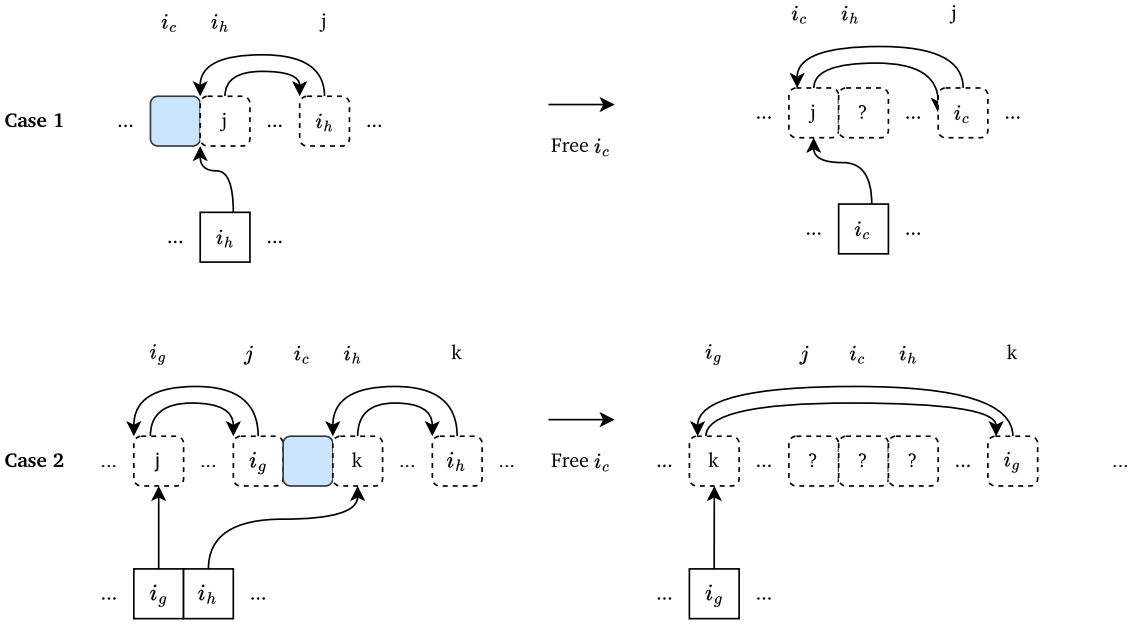


Figure 3.10: Two cases when freeing a component that may require us to locate the index of a hole in the free-list. Square entries are the free-list

Both cases require us to update hole-indices within the free-list, and so we must locate these indices within the list. This is an operation with linear complexity with respect to number of elements in the free-list.
One way of improving this is to store the free-list index s_h of the hole inside the hole's head entry h. This results in a direct two-way reference from the free-list entry to the hole and from the hole to the free-list entry, and it allows us to maintain the free-list in constant time. See Figure 3.11.
In case 1, we locate the hole-index i_h in the free-list using the free-list index s_h stored inside the hole's head entry h. The value at s_h, i.e. the hole-index i_h, is changed to the index of the new head entry i_c.
In case 2, we just want to remove element s_h from the free-list. Instead of shifting every succeeding element s_j to $s_j - 1$, we only move the last element in the free-list s_l to s_h. However, this means we must update the free-list index of the l, because it has now been moved in the free-list. The element s_l contains the index i_l to the hole head l, so we can fetch this entry and update its free-list index to s_h in constant time.

3.4.6 The complete entity system

The final entity system allows the user to construct *entities* to which they can add *components*. These components are either provided by the engine or defined by the user, and internally stored in a *registry block* owned by a *registry* unique to the component type. The components' memory address is static, and can be referenced either by raw pointers or the safe *component reference*. Behavior is intended to be applied to the components by *iterating* all components of a given type.

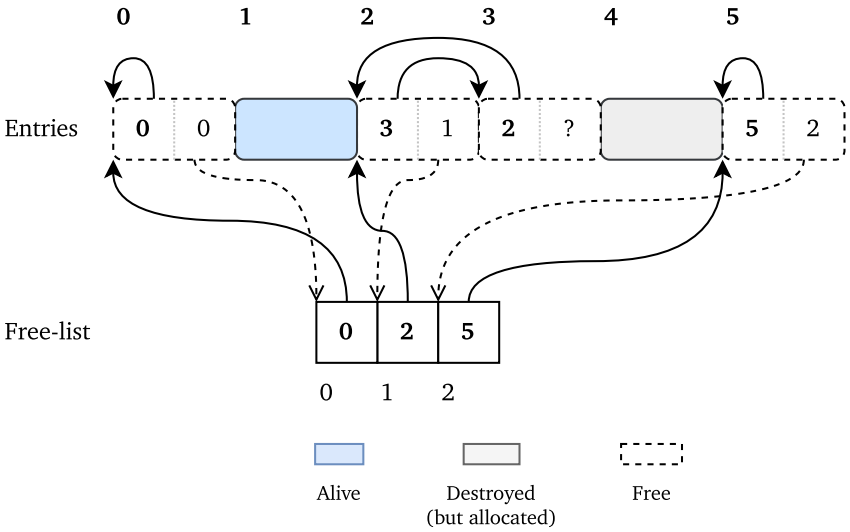


Figure 3.11: Extension of Figure 3.9 with indices of the head entry of every hole instead of every free entry, and with every hole head entry having an index to its location within the free-list. The first number in every entry is index to the head or tail entry, and the second number is the index into the free-list.

Components and entities are constructed in constant time complexity, and the complexity of destruction is linear with respect to the number of siblings on the entity. Iteration is linear with respect to the number of allocated entities plus the number of holes. However, the number of holes is at most half of the number of allocated entities.
The metadata for every component and entity takes up 12 bytes. The registry for every component type takes up 24 bytes, and every block is 24 bytes plus 1 byte for storing the index of the block in the registry, if the block is free. Additionally, every block stores 2 bytes per hole, with at most $b/2$ holes when b is the number of entries in the block. Distributing the block size overhead across components, every component has a total overhead o of

$$o = 12 + \frac{(25 + \frac{b}{2} \cdot 2)}{b} = 12 + \frac{25}{b} + 1$$

With $b = 25$, which is the default, this is 14 bytes overhead per component. This is not including the registry's size overhead nor the additional overhead caused by memory alignment requirements.

An example of the entity system's actual C++ APIs and how it is used, can be found in Appendix D.

3.5 Transform system

It is common for engines to have some concept of a game object *transform* that describes the object's position (often called *translation*), scale and rotation. For example, in Unity all objects have a transform component [49]. Additionally, a transform typically has another transform as its *parent*, which describes what the parameters are relative to. The recursive property of this allows you to build trees of transforms, or so-called *transform hierarchies*.
Regardless of how the transform is stored, it is typically computed to a *transformation matrix*, or a *change of basis matrix*, that can transform points and vectors defined in local space to world space. This matrix is heavily used for graphics and collisions, and it is among PlayStation's GTE primary purposes to combine and apply these. As such, we want Tundra to provide

something similar. [7]

3.5.1 Tundra's Transform Matrix

In Tundra, the convention is to use *column vectors*, and in theory a transformation matrix is thus:

$$T = \begin{bmatrix} x_1 & y_1 & z_1 & t_x \\ x_2 & y_2 & z_2 & t_y \\ x_3 & y_3 & z_3 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

x_i, y_i and z_i are the basis vectors, meaning they represent the scale and rotation, and t_x, t_y and t_z is the translation. As such, for a local-to-world transformation matrix, column vectors are multiplied on the right side:

$$T * v_{local} = v_{world}$$

In practice, the last row of transformation matrices is always $[\,0\;0\;0\;1\,]$. Furthermore, the PlayStation GTE uses different byte sizes for the basis vector (scale-rotation) elements and translation elements, being 2-byte and 4-byte values respectively. So instead of representing it as a single matrix, it represents it as two components: a 3x3 matrix of 2-byte values representing the basis (scale and rotation) and a translation vector of 3 elements. The basis matrix is henceforth called the *scale-rotation matrix*.
Because of GTE's way of storing matrices, Tundra has a specialized *transform matrix* that is different from the general-purpose matrix types described in Section 3.3.4. This matrix class stores the scale-rotation matrix as a 3x3 matrix of fixed-point values in 4.12 format (2 bytes or total 19 bytes), and a vector of 3 fixed-point values in 20.12 format (4 bytes).

3.5.2 Expense of Transforms

Transforms are problematic, because they can be both very expensive to compute, and take up a lot of memory.

The computation performance is an issue for two reasons: 1) computing the local matrix from raw scale, rotation and translation and 2) combining the local matrix with the parent matrix.
Computing the local matrix is mostly expensive because of the rotation. psn00bsdk has a routine for computing a rotation matrix from the axis angles, but it involves two matrix-matrix multiplications, and 3 *sine* and 3 *cosine* computations, each of which contains 3 multiplications and numerous bit-shifts. Despite the GTE, the matrix-matrix multiplication is still expensive, because it has no general-purpose matrix-matrix multiplication operation. But you can utilize the matrix-vector multiplication that it *does* have support for, by multiplying the left-matrix with each column in the right matrix. psn00bsdk also has a routine for this, but it is a 50-instruction assembly routine, with multiple memory loads and 3 matrix-vector multiplications.
Individually, both the translation and scale are just copies and not expensive. But the scale matrix must be combined with the rotation matrix, which, again, is an expensive procedure. GTE's lack of a matrix-matrix operation also makes the combining of transform matrices

[7] See [15, p. 5.3] for more information or elaboration of transformation matrices

expensive. psn00bsdk has a function for performing this multiplication (like generic matrix-matrix multiplication), but again it is also an assembly routine of 70 instructions, including multiple loads and 4 matrix-vector multiplications.

In terms of memory, storing the transform as a rotation of 3 values of 2 bytes, a translation (position) of 3 values of 4 bytes, a scale of 3 values of 4 bytes takes up 30 bytes. It is the same size if the transform is stored directly as the transform matrix. These 30 bytes exclude other data, such as hierarchy meta-data (elaborated in Section 3.5.5).

3.5.3 Transforms as components

It is common for transforms to be a built-in concept of the game-object of systems. In Unity all game objects have a transform component, and Godot has 3D *nodes* that extend its base node with transform information.

For Tundra, we avoid making transforms a required part of components and entities, because of their expensive nature, similar to the approach of Godot. However, we go one step further, in that transforms are components themselves. This means that for an entity to have a transform, you must add it as a component, and you may add multiple transforms if you want.
But it also means that even if a component is spatial in nature, such as a model component, it does not include a transform. Instead, it has a *reference* to a transform that is provided by the creator of the component. This has the advantage that multiple models (or any other components) can use the same transform, if so desired. That is unlike Godot, where each model would have their own transform, with their common transform being that of their common parent. Figure 3.12 illustrates an example of this.

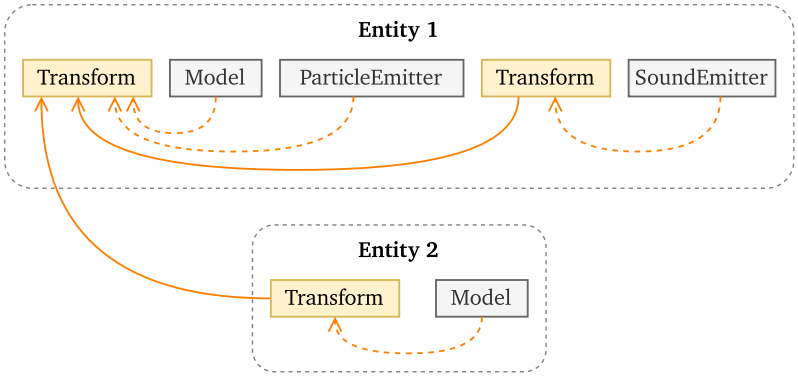
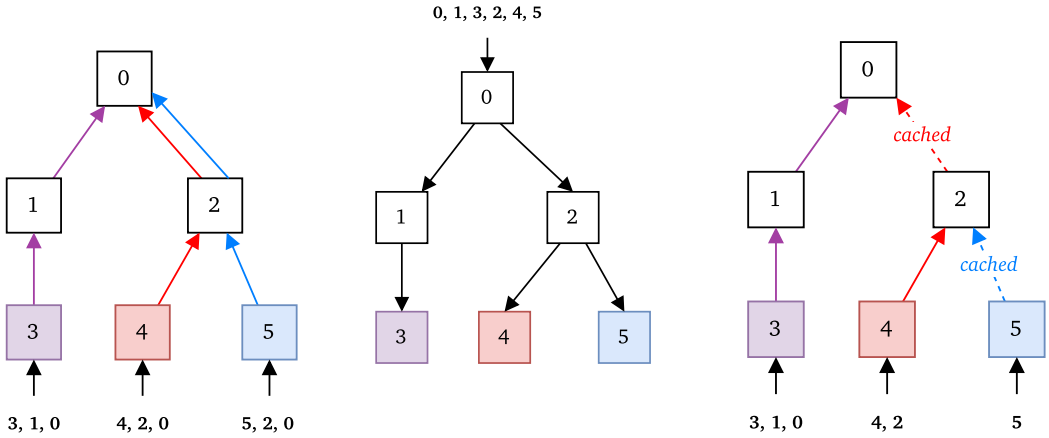


Figure 3.12: Simplified example setup of transforms with two entities. The orange arrows are component references to transforms, with the solid arrows being transform-parent references.

3.5.4 Computation and storage

A simple, but costly approach to computing a transform's matrix, would be to compute the full matrix every time it is requested. This is a recursive operation, as it must use the parent's matrix, and thus also compute that. But if a transform has multiple descendants, its matrix must be computed for every descendant matrix that is computed. See Figure 3.13a. This is redundant work if the matrix never changes.
Alternatively, we may compute all matrices in the hierarchy at the same time. Going from top to bottom, we can temporarily cache a parent's matrix between computing the transform matrices of its children (see Figure 3.13b). But to gain any benefits from this, we must then use all the matrices at the same time.

To avoid these expensive recomputations, and make it more flexible, we sacrifice some memory, and introduce a *cache* of the computed matrix on the transform component: whenever a transform's matrix is computed, it is stored within this cache for later use. Computing the cache for the parent will thus only calculate its matrix once for all children. See Figure 3.13c.
A significant downside is that this introduces an additional 30-bytes into our transforms, which is the size of a transform matrix as noted earlier. While it is still very expensive in terms of memory, its severity is lessened by the choice of making the transform non-mandatory, which likely will reduce the number of total transforms in practice.



(a) From bottom, no cache **(b)** From top, temporary cache **(c)** Bottom up, persistent cache

Figure 3.13: Three approaches to computing the transform matrix for transforms 3, 4 and 5. Each box is a transform, and the box above is the parent. Each solid line is the computation of a matrix on the transform it points to. The outside numbers are the order that transform matrices are computed.

Updating cache

Of course, the cached matrix of a transform can be invalidated if changes are made to the transform or any of its ancestors. One way of handling this is to introduce a boolean *dirty-flag* that signals whether any local changes have been made to the transform. If so, the cached matrix must be recomputed. But it is only local to the changed transform - the ancestors do not require recomputations unless they are also dirty.
This simple approach can be optimized by considering what has actually changed in more detail. If a transform t has a local scale-rotation matrix L, translation t_t and a parent p then its world scale-rotation matrix M_t and translation T_t are computed like so:

$$M_t = M_p \cdot L_t$$
$$T_t = M_p \cdot t_t + T_p$$

According to this, if only the translation of t, t_t, has changed then we do not need to recompute the scale-rotation matrix M_t, as it does not depend on the translation. A similar thing is true for the scale-rotation - we do not need to use and potentially re-compute the world translation of the parent, T_p, if only the parent's scale-rotation matrix, M_p, has changed.
For a transform t and each of its descendants d, we can generally deduce that:

- if translation of t, t_t, has changed, then the world translation of t and its descendants, T_t and T_d, are all dirty

- if the scale or rotation of t, L_t, has changed, then its own world scale-rotation, M_t, and its descendants entire world transform matrices, M_d and T_d, are dirty

To utilize this behavior, we implement *two* dirty-flags instead of one: one denoting if the scale-rotation is dirty and one denoting if the translation is dirty. When changing the translation, scale or rotation on a transform it will set its own and its children's flags dirty appropriately. When computing the transform matrix, we check which flags are set, and only compute the parts required.

Further optimizations

It is possible to add a few additional optimizations by considering the values of the scale and rotation. If either the scale or the rotation is not set, the scale-rotation matrix simply equals the matrix of the property that *is* set. If neither are set, the scale-rotation matrix is just the identity matrix.[8]
One may also cache the local scale-rotation matrix. This way you would not need to recompute it for the world matrix when the parent matrix changes. Also, it would allow you easily get the local transformation matrix for the transform as it would just need to be concatenated with the translation. However, the benefit of this is presumed to be less than caching the world matrix and the latter use-case is not expected to be common. As this is untested, and it will introduce an extra 18 bytes of data in the transform, it is not implemented in Tundra.

Storing scale, rotation and translation

We store the raw local scale, rotation and translation values as individual members. This is to gain the full benefit of the dirty-flags, as changing one of the members will lead to minimal recomputations. This allows you to change the values multiple times and in multiple places between each use of the world transform without incurring a recomputation cost.
But it is also done for simplicity. While it is possible to store, extract and manipulate the values directly in the world matrix or in cached local matrix (as mentioned above), this is more complicated and prone to inaccuracies.

3.5.5 Transform hierarchy

As mentioned earlier, it is common for transforms to be able to form a *hierarchy*, and Tundra should support this.
A very minimal approach to this is for transforms to have a parent reference. This is only a single reference, but given all transforms you can infer the entire hierarchy. However, some earlier described details, such as the ability to set descendants' dirty-flags or destroying the entire hierarchy from the root, require us to traverse *from* a parent to all of its descendants. With the minimal approach, this is very inefficient and impractical.
To make traversal of descendants more efficient we introduce three new references: one to a transform's first child, one to a transform's next sibling and one to its previous sibling. The last and first siblings are also connected, meaning the latter effectively describes a circular doubly linked list of a group of sibling transforms. The reference to a transform's first child is then a reference to the head of this list. Given these references, a transform has access to all its descendants. An example hierarchy is given Figure 3.14, which illustrates these references.

This linked-list approach is similar to that used for component groups, with the difference being that we also have links in opposite directions. Not having these opposite links would

[8] Similar optimizations can be done with the translation, but they have been skipped for now as they are expected to be less impactful.

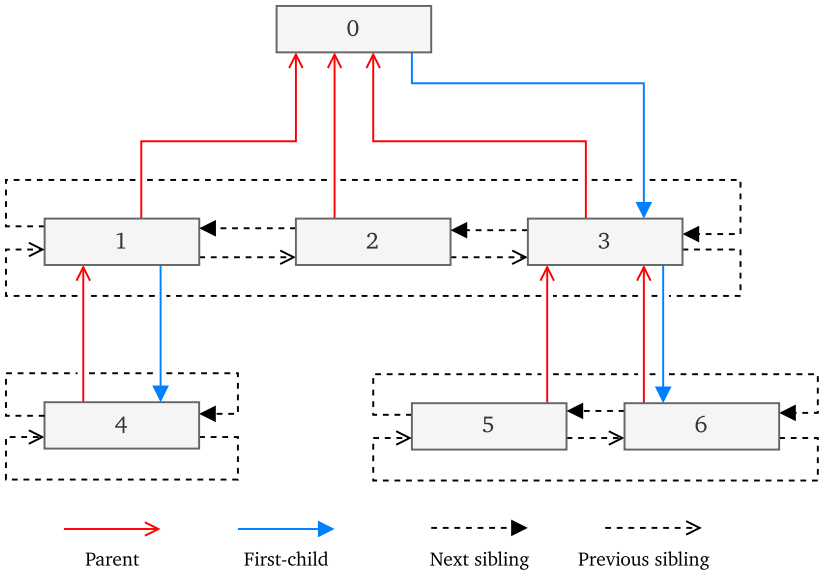


Figure 3.14: A hierarchy of transforms as numbered boxes with references between them being arrows.

mean that adding and removing children would have linear time complexity as we must traverse the entire list to find the neighboring siblings that would need to be updated. We could argue the same for the component links, but the difference with transforms is that we consider the memory overhead less significant for two reasons. The first being that we expect there to be fewer transforms than components, and the second that transforms already take up so much memory that if they cause memory issues the solution is likely to reduce the number of transforms regardless.

The hierarchy data consists of 4 references, each of 4 bytes, and final size of a transform is 92 bytes in total (including component metadata).

3.5.6 Static transforms

A common use-case is that of things that have a transform, but never move, such as terrain, walls and decorations. All of these can be classified as *static objects*. While caching ensures that such objects will not have their world matrices recomputed, they store a lot of unnecessary information. If we are certain they will not be recomputed, there is no need to store the dirty flags or the raw values for scale, rotation and translation. Also, there is no reason for them to hold a reference to a parent. It is conceivable that they could have children, but for simplicity we generalize it, so that all hierarchy information can be removed.

To optimize the static object use-case we differentiate between two transform types: *dynamic* and *static*. The *dynamic* transform is the one we have described up until this point, whereas the *static* is a much simpler variant. It only has one member, which is a pre-computed transformation world-matrix. Its construction can be done from the raw scale, rotation and translation values and even be relative to a parent, but these parameters are not stored along with it. As a result, the component is only 44 bytes (including component metadata).
Systems and components using transforms should not worry about whether they are operating on a static or dynamic variant. All they likely care about is the transforms world-matrix. To allow for this, both transforms inherit an abstract transform base component to which you can hold a component reference, and from which you can get the world matrix[9].

[9]The computation does not use a virtual method, because the computation of transforms is done using a static

3.6 Render system

The purpose of the rendering system is to take the simulated game logic, and display it onto the screen. A low-level library may expose some API to do this, requiring the user to manually decide when and where in their code each object must be rendered. However, being a general-purpose engine, we want Tundra to alleviate this burden from the user. As such the rendering system is an internal part of the engine that automatically runs every frame after the user's game logic. To get objects rendered to the screen, the user must set up either of three renderable components: *models*, *sprites* or *text*. The render system will iterate all of these and *submit* the according primitive to the GPU, which will afterwards *draw* (rasterize) them to the screen.

3.6.1 Ordering table

The psn00bsdk provides a series of types for the different primitives. Besides rasterization information, each primitive also stores a reference to another primitive. This allows for submitting these to the GPU for drawing through the use of a structure called an *ordering table*. The ordering table utilizes the DMA linked-list mode mentioned in Section 2.2.3, and its primary purpose is to provide an efficient way to maintain proper ordering of primitives.
The ordering table is an array of pointers with each element in the list representing a depth value and the length of the list representing the *depth resolution*. By default, each pointer points to the next element in the array. Inserting a primitive into a depth value, will insert the primitive between the array element and what the element pointed to. Figure 3.15 shows an example.
Effectively, the ordering table is a linked list, and the order within the list infers the rendering order. The primitives are submitted to this ordering table by the CPU by specifying a depth-value. To support and accelerate this the *GTE* has operations for mapping 3 z-coordinate values of an arbitrary range to a single value within the range of the array. So, each z-coordinate value does not have to match exactly one depth value.
The ordering table effectively allows us to sort primitives by depth values in constant time. The ordering table array, must, however, exist in main memory, and increasing the depth resolution, increases the array size, so care must be taken to limit the size of it.

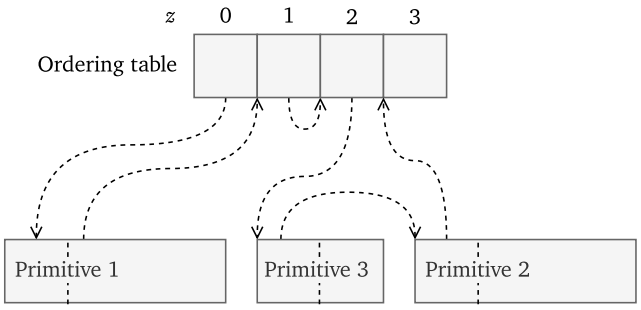


Figure 3.15: Example of ordering table with a depth resolution of 4. It contains 3 primitives - 1 at depth value 0, 2 at depth value 2 and none at depth values 1 and 3. The links can be followed to obtain the rendering order.

3.6.2 Primitive buffer

The simple, but also expensive solution would be to allocate the primitives using individual heap allocations. This grants a lot of flexibility, but it is very expensive, and luckily, not

GTE-reliant functions. To allow GTE to differentiate between the two transform types, a custom RTTI-like feature is added specifically to the transform.

required. If the renderer sequentially constructs a new set of primitives and submits them every frame, then we know there are no gaps between primitives allocated in the same frame. To utilize this, we introduce a *primitive buffer*. This is a contiguous buffer of raw bytes, and allocating a primitive simply appends it to the end of the buffer. It thus allows for allocating primitives of arbitrary sizes in constant time. See Figure 3.16 for the relation between the ordering table and the primitive buffer.

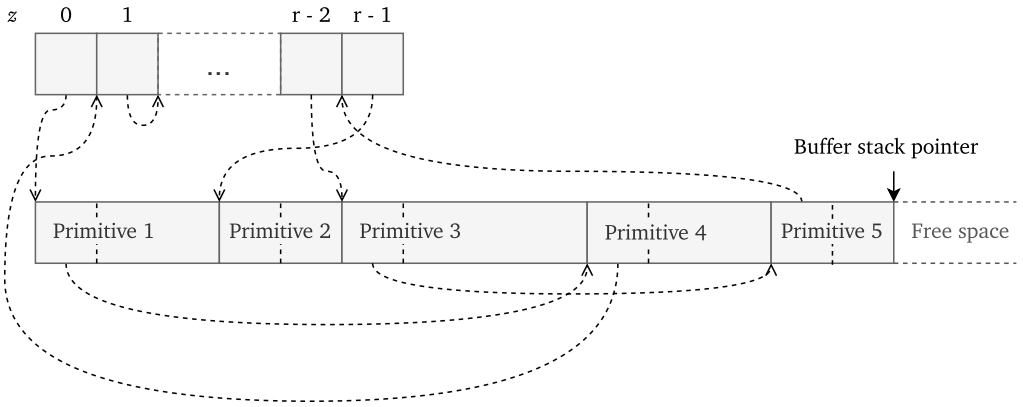


Figure 3.16: Example of ordering table and primitive buffer and how they relate. z is the index into the ordering table, and r is the resolution of the ordering table. Note that primitives are inserted in the front of the existing elements, which is why the pointers point to preceding memory.

Double-buffering

As mentioned in Section 2.2.3, it is possible for the CPU to write to VRAM while it is also being read by the video decoder, which displays some of it to the attached screen. This feature allows us to utilize *double-buffering*. In Tundra, we implement this by supplying 2 sets of ordering tables, primitive buffers and framebuffers that we call a *render context*.
In one frame we submit primitives to the ordering table and primitive buffer of context 1. While doing this we cannot have the GPU draw the primitives of this ordering table and primitive buffer to the framebuffer, because we are changing their state. But this, in turn, means the framebuffer of this context is not being written to, so we can display this without it being disturbed by intermediate rendering results. On the other hand, context 2's ordering table and primitive buffer are currently stable, and its framebuffer is not being displayed. So, we can safely have the GPU draw the primitives submitted in previous frame to the framebuffer. In the subsequent frame, we then flip the buffers that we submit to, draw to, draw from and display. See Figure 3.17 for a visualization of this.

This asynchronous behavior has a great benefit: the drawing by the GPU can happen in the background, while we continue on with the simulation and graphics submission of the next frame. The drawing will only be a bottleneck if it takes longer to draw than it takes to do the simulation and submission. This effectively doubles our performance budget, as both can take $\sim$33 milliseconds to execute and still leave us with a framerate of 30 FPS. Without the double-buffering these 33 milliseconds would have to be split between them.
It does, however, have the downside that the frame we are displaying was simulated 2 frames ago. We can add to this that a button pressed during a frame is not handled until the subsequent frame. As such, there is a 2-3 frame, or $\sim$66-100 millisecond, delay between the player pressing a button, and the rendered game reacting to it[10].

[10]This is not including the hardware delay when pressing the button or the refresh rate of the display.

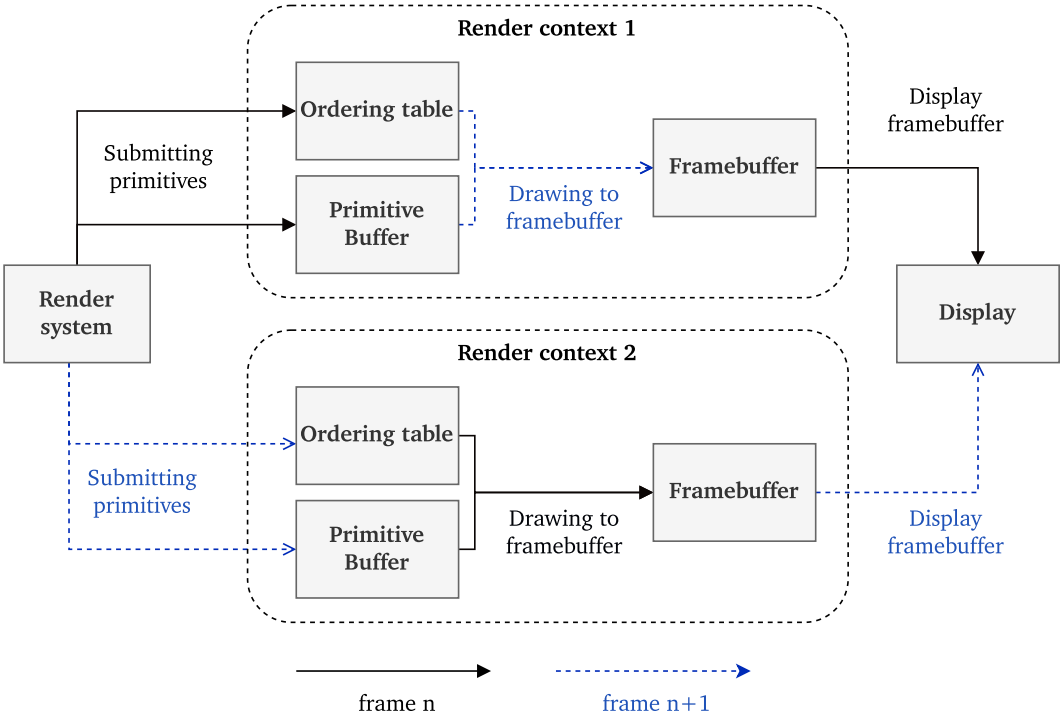


Figure 3.17: Visualization of the 2 render contexts in consecutive frames, and what is being submitted to, drawn to and displayed. The arrows indicate a data direction (data *Framebuffer* is being read and "written" to the *Display*).

3.6.3 Cameras and layers

In game development it can be intuitive to imagine that everything is rendered from the perspective of a *camera* that is placed in the 3D world. So, naturally, we also want to support this in Tundra, and it is made an essential part of the render system.
Most fundamentally, the camera should provide a transformation matrix V, also known as a *view matrix*. This should place an object so its relative to the camera, and thus be applied to every rendered object's *model matrix*[11] M as $W = V * M$. The camera will use a regular transform component to represent its placement, but the matrix must be calculated differently than a regular transformation matrix. We will not go into details here on how to do this, but it is a fairly expensive operation.

Occasionally, you may want multiple cameras in your game. For example, for a split-screen game such like in *Crash Team Racing* [60], or a rearview mirror like in *Gran Turismo* [56]. In Tundra multiple cameras are supported by integrating cameras into the entity system and making them into components. When the render system updates, it iterates and renders each of these cameras in turn.

Besides having multiple camera objects, you typically want your camera to be able to do two things: 1) render to specific subset of the framebuffer/screen and 2) only render a subset of the objects in the world. The latter might be important to ensure different UI in split-screen or reduce the quality of the rearview camera.
In order for a camera to render only a subset of objects, we introduce the concept of *layers*. A layer is a group of rendered graphics with the property that everything in one layer will be entirely in front of or behind another.
A renderable component can then be assigned a single layer by setting its *layer ID*, and, in turn, each camera is assigned a list of layer IDs that it renders. When the render system

[11] See [15, p. 5.3.9] for more details on relation between matrices.

```
procedure RenderFrame:

foreach camera in all cameras:
        C = Compute camera matrix of camera
        L = Indices of layers rendered by cameras

        foreach model in all models:
            if L contains layer index of model:
                O = ordering table layer of camera
                Submit model using C and O

        foreach sprite in all sprites:
            if L contains layer index of sprite:
                O = ordering table layer of camera
                Submit sprite

        foreach text in all texts:
            if L contains layer index of text:
                O = ordering table layer of camera
                Submit text

    Wait for GPU to finish drawing last frame
    Wait for vertical sync
    Flip buffers

    foreach camera in all cameras:
        Order GPU to draw camera's ordering table
```

Codeblock 3.3: Pseudocode of procedure for rendering, which is run at the end of every frame

renders the camera, it iterates all components, but it only submits the components that are assigned to one of its layers.

The layering within a camera is achieved by introducing layers into the ordering tables. This can be done by having multiple arrays for each ordering table with the last element of one array linking to the first element of the next. Each array represents a layer, and this way, all elements in an array are rendered before any elements in the subsequent layer. Because the DMA simply follows the links until it reaches the end (see Section 2.2.3), it does not pay any mind to it being split across arrays.
At the same time, it allows you to use different depth resolutions (array sizes) for different layers. In cases where you know that certain objects will always appear behind others, they can be submitted to a separate layer. But if you also know they will never overlap in that layer, such as a flat floor, or be spaced much further apart, the array can have a lower depth resolution and thus take up less space.
To achieve the layering of cameras, such that all submitted primitives of one camera are laid on top of another, each camera is given its own set of ordering tables. When ordering the GPU to draw, we iterate the cameras and order it to draw each one in turn.

The support for a camera to render to a different, limited part of the framebuffer is not implemented, but the camera system was designed with it in mind.

3.6.4 Render procedure

Based on the earlier described rendering of cameras and layers, a summary of the render system's core procedure is laid out in Codeblock 3.3, and is executed at the end of every frame. The procedures for submitting the individual renderable components - models, sprites and text - are given in the following sections.

3.6.5 Models

A requirement for the engine is to support 3D games, so it is important that it can render 3D objects. There are various ways one can optimize the rendering of certain primitives, such as planes, spheres and boxes. However, most importantly we want to be able to render arbitrary models defined by the user, and such a generalized approach can also render those simple primitives. So, for simplicity, we only expose a single *model* component. This component references a *model asset*, and some data to render an instance of such an asset, including a transform reference and color.

Model asset

The PlayStation has no built-in format for models, but the features we require from such a format are fairly simple, so we introduce our own.
Tundra's model asset includes data for vertices, normals, colors, UVs and whether it is smooth shaded or flat shaded. The vertices, normals and UVs use *indices* to compress data.
Despite the asset includes UVs it does not include the texture itself. Instead, this is included separately and provided to the asset when loaded, which allows multiple models to use the same texture.
A model is subdivided into different *model parts*. Each part is a unique combination of a texture, color tint and smooth-shading flag, and its purpose is mostly to optimize the use of the instruction cache. The vertices and normals are shared by all parts, but each part has its own indices.

For the format to be useful the user must be able to import some model in a different format than our format. To support this, we include a *model compiler* that can take a model in *Wavefront's* commonly used *OBJ*-format, and compile it into our format. We will not go into details with the compiler itself and further details on assets are given in Section 4.3.1.

Model submission

The procedure for submitting models that was referred to in Codeblock 3.3 is given in pseudocode in Codeblock 3.4. Some details are left out, and the last part about allocating and assigning data to the primitive is restructured slightly to improve readability.

In the procedure for submitting models there are a few important things to take note of.
For one, multiple checks are done to cull certain invisible triangles. The most significant of these is the culling of triangles facing the wrong way, as, for example, up to 50% of the triangles of a sphere would be culled with this.
Also, there is a difference in the primitives we must pass to GPU depending on whether a primitive is smooth-shaded and textured - in other words, there are 4 different types. These differ in size, and thus, also the amount of data we have to set. But most significant is that smooth shaded triangles require us to compute 3 colors based on lighting, as opposed to a single color for flat shaded primitives.
Additionally, before iterating any parts or triangles, we do some initial setup for the whole model, which would be very expensive to do for every triangle or part. For one, we compute and set the transformation matrix of the model. Secondly, we compute and use the light directions in *model space* as opposed to *world space*. This way, we can use our normals as-is without transforming them to world space first, which is a significant improvement. The directions can be computed as $L_{model} = M^{-1} \cdot L_{world}$, where M are the directions as a matrix, and M^{-1} is the inverse of the transform matrix. The inverse of a matrix can be very expensive to compute, and GTE has no acceleration for this. We will not go into details how, but this can be simplified to the more performant approach $L_{model} = L_{local}^{T} \cdot R$ where R is the rotation matrix of the transform, and we do not have to do any matrix inversions.

```
procedure SubmitModel(model, camera_matrix, ordering_table_layer):

    B = active primitive buffer

    T = compute transformation matrix
    V = camera_matrix

    R = extract rotation matrix from T
    L = compute model space light-directions using R and light directions
    GTE: set active light directions to L

    GTE: set active transformation matrix to V

    foreach part in model parts:

        c = compute the color based on the asset's color and model color

        foreach triangle in model.triangles:

            GTE: Load triangle's three vertices to
            GTE: Apply transform and projection to vertices

            GTE: Compute wheter triangles are backfacing
            if triangles are backfacing then cull triangle

            z = GTE: Compute average Z-value
            if z is outside ordering_table_layer range then cull triangle

            t = GTE: Get screen triangle
            if t is outside screen then cull triangle

            P = The type of the primitive to allocate
            p = Allocate of type P with ordering_table and B
            Set corners of p to t

            if part is textured:
                Set texture ID of p to triangle's texture ID
                Set UVs of p to triangle's UVs

            if part is smooth shaded:
                n0, n1, n2 = Three normals of part
                l0 = GTE: Compute lit color from c and n0
                l1 = GTE: Compute lit color from c and n1
                l2 = GTE: Compute lit color from c and n2
                Set colors of p to l0, l1 and l2

            else:
                n = Single normal of part
                l = GTE: Compule lit color using c and n
                Set color of p to l
```

Codeblock 3.4: Procedure for submitting a single model for drawing by the GPU.

```
procedure SubmitSprite(sprite, ordering_table_layer):

    w = Width of sprite
    h = Height of sprite
    v0, v1, v2, v3 = Corners of sprite when placed at (0,0) with sprite's
    width and height

    Rotate v0, v1, v3 and v3 around (0,0) by sprite's rotation
    Translation v0, v1, v2 and v3 by sprite's translation

    p = Allocate primitive using active primitive buffer
    Add p to front of ordering_table_layer

    t = Sprite's texture asset

    Set texture ID of p to texture ID of t
    Set UV of p to UVs to t's VRAM position
    Set color of p to sprite's color
```

Codeblock 3.5: Procedure for submitting a single sprite for drawing by the GPU.

3.6.6 Sprites

Even for 3D games, it is useful to be able to draw 2D elements, as most 3D games also have some kind of user-interface. The most versatile and important feature in this regard is the drawing of 2D textured quads, also known as *sprites*. Much like with models, we introduce this in Tundra through a *sprite component*.
Because sprites typically only have 2D movement and scaling, and only rotation around a single axis, their transforms are often simpler. For the sake of scoping we are not introducing a specialized transform, but instead introduce a position, size and rotation directly in the sprite component. Additionally, the sprite has a texture and color tint.

Sprite submission

Submitting sprite components to the GPU is more trivial than the submission of models, as is seen in Codeblock 3.5.
Unlike with models, the sprite submission does not utilize GTE at all, nor does it make any attempt at culling invisible sprites. The latter would only be useful for out-of-screen sprites, and it is unlikely for that to be a significant scenario with UI. Also, the sprites are submitted as regular rectangle primitives and not axis-aligned quads, as they must support rotation.

3.6.7 Text

For rendering of text, we utilize the built-in *debug text* utilities of psn00bsdk. This utility loads a font into VRAM as a 32x64 texture in 16-color palette mode plus the palette taking up a total of 1056 bytes.
The SDK exposes some functions that allow us to render a text given the content as a C-style string, a position, a primitive buffer and an ordering table. But the SDK does not know about Tundra's way of handling primitive buffers and ordering tables. It can still be properly integrated, but it requires some additional work that we will not go into detail with.

As the function parameters give away, the debug font system is very limited. It can only render text in one font of one size in one color. So, the system is not a representation of a proper text rendering system, but it does illustrate something relevant. The text is rendered using *axis-aligned quads*, making them significantly faster to render than regular sprites, and

so it can be used to gauge the potential in rendering such sprites, which is further explored in Chapter 4.

3.7 Other systems

3.7.1 Time system

The psn00bsdk does not provide any utilities for measuring time. So, in order to get such functionality, which is needed to provide a *frame time*[12] and measure performance, we must instead drop to the hardware and BIOS level.
In the hardware, one may set a particular *timer* register to be incremented at different hardware events. For example, it can be incremented every 8th cycle of the system-clock, which is the clock used by the CPU. This results in a time resolution of $8/33,868,800$ seconds, or ~ 0.24 microseconds. This is good enough for our purposes. The problem is that the register is only 16-bit, meaning it can store at most $(2^{16} * 8)/33,868,800 \approx 0.0155$ seconds [36, Timers]. In Tundra, we solve this by registering an *interrupt handler* that is triggered whenever the register is full. In this handler we can then add the register value to a global 64-bit accumulated time stored in microseconds. To both avoid accumulation errors and make the handler performant, we avoid using division by maintaining a remainder value, as seen in Codeblock 3.6.

The user can get the current time through the procedure listed in Codeblock 3.7. Here we add the current register value and remainder cycles to the accumulated microseconds.
The procedure ultimately returns the whole number of microseconds that has passed since the system was started. Any precision lost, because of returning microseconds as opposed to cycles, is only present in the returned value, and it is not a precision loss that accumulates within the system - the system always returns the accurate number of microseconds.
The value is returned as a *duration* type, which internally stores it as microseconds. To retain the precision, storing and comparing time should always be done using this type, instead of converting them to another scale, such as milliseconds.

3.7.2 Audio system

While the PlayStation has many useful audio features as mentioned in Section 2.2.4, Tundra's audio system is limited for scoping reasons. As such, it only supports the playing of a single-

[12] See [15, p. 8.5.1] for further elaboration on frame time.

```
a: accumulated microseconds
r: remainder cycles

m: smallest number of cycles representing a whole number of microseconds
n: number of microseconds represented by m cycles

procedure Tick():
     a = a + whole number of microseconds in filled register
     r = r + remainder number of cycles
     if r >= m:
        a = a + n
        r = r - m
```

Codeblock 3.6: Pseudocode procedure for accumulating cycles as microseconds, which is run whenever the time register is full.

```
t: timer register
a: accumulated microseconds
r: remainder cycles

procedure GetTimeSinceStart():
    non accumulated cycles = (t + r) * 8
    return a + (non accumulated cycles / cycles per microsecond)
```

Codeblock 3.7: Pseudocode procedure for getting the time since start

music asset and up to 23 other sounds, both in mono and neither being spatial. Because the playing audio has no data other than the asset it is using, the audio system does not expose any components through the entity systems. Instead, it just provides some trivial functions for starting and stopping music and sounds.

For each channel we store the *ending time* of the currently playing sound, which is computed when the sound is started. If the sound is looping, the ending time is set to the maximum time value. We need this information, as it is the simplest way to check if a channel is available, or if it is occupied by another sound. When starting a sound, we look for an available channel by iterating through the channels, and if the current time exceeds the channel's ending time it is free to use.
Besides the ending time we also store a *play ID* for each channel, which we increment when a new sound is started. This ID is combined with the index of the channel in a 32-bit value, and this combined value is considered the *sound ID* for the playing sound. This ID is returned to the user when they start a sound, and it allows them to manually stop it from playing. The channel index allows us to efficiently find the channel proper channel, and the play ID ensures that we only stop the channel if it is still playing the sound with the given ID.

The processing of each sound is done by the SPU. So, even though the channels are only silenced and not stopped from processing when a sound has ended, it has no effect on any other systems.
On the CPU side, while playing a sound may require iterating through the full channel list, the list only contains 24 elements. But more importantly it is not expected that the user will play a lot of sounds, as only 24 can be playing simultaneously. So, we generally assume the audio system to have a negligible impact on performance.

3.7.3 Input system

To read controller input, we must pass two 34-byte buffers (one for each controller) to psn00bsdk's API, and at every v-sync the buffers will be filled with the status of the connected controllers. While the PlayStation supports types of controllers such as joysticks and a mouse, we only support the original controller for simplicity. Aside from a single extension to this, Tundra simply wraps each of these buffers in some more convenient types.
The extension allows users to differentiate between the first frame that a button is in a state and the consecutive frames where it *remains* in that state. The first may be useful for a jumping action, whereas the latter may be useful for accelerating a car. But this is a trivial extension as we can just maintain the previous frame's button states for each controller and compare if the state has changed.
In general, the input system has a negligible impact on the performance of the engine, both in terms of cycles and memory.

Chapter 4

Experimental setup

The engine has been tested through two different experiments: performance benchmarks of various features and using it for a short team-based game jam.

4.1 Testing platform

As noted in Section 2.1, the results of this work will not be evaluated on the hardware for legal reasons. Instead, both experiments were done with the use of emulators, despite not these not being completely accurate to the hardware.
The core CPU is probably the most accurate. The PlayStation's CPU clock is running at a constant rate, and the time measurements are based on this. So as long as the instructions of the emulated CPU take the same number of cycles as the original, the timing will be correct. The accuracy issues come from the instructions that access peripherals such as RAM, GTE, GPU, SPU and DMA, as these timings are harder to emulate accurately. In particular, PCSX-Redux is least accurate when it comes to emulating the GTE and GPU rasterization.

4.2 Performance benchmarking of features

Through the benchmarking of features we aim to gauge how performant various uses of the engine's features are in isolation. The features tested here are the entity, transform and the rendering systems.
Other features, like the audio and input system are not tested, as it is expected only a few of these systems' very simple functions run on every frame. The likelihood of these features being performance bottlenecks are thus very low.

The measurements are done in various ways depending on the system, but they all measure how many milliseconds it takes to do a specific thing. The timings are measured using the engine's timing system. As this uses the underlying system-clock for timings, the millisecond result is equivalent to a number of cycles. This also means that there is very little variation in the measurements, and for CPU only setups it is completely deterministic. As such, the result of each benchmark is the result of a single execution.
To get more confident results, all benchmarks are tested in 2 emulators: *PCSX-Redux* and *DuckStation*.

All benchmarks are compiled in *release* mode, which strips away any debug logging and various safety tests in functions. This also enables full compilation optimizations (the `-O3` compiler option), but it does not enable *link-time optimization* as that is not supported by the

compiler toolchain.

4.2.1 Entity system

Three uses of the entity system are tested: creation, deletion and iteration of components.

In all tests, we operate on 1000 components of 3 different types, in various setups. The creation and destruction of the entities that hold the components are not included in the measurement.

The component types differ in the size of their custom data, with their component metadata being the same:

- *Small*: 1 byte
- *Medium*: 16 bytes
- *Large*: 64 byte

We test different sizes despite there being no data cache, because of the memory's burst operation capabilities as mentioned in Section 2.2.4 and their potential impact.
The setups vary in whether there are holes in between the stored entities. As was elaborated in Chapter 3, the component system attempts to reduce performance cost of holes, so testing setups with these serves to illustrate and test the effect of this. Note that the holes themselves do not count towards the 1000 entities.

For all tests the registry block size is set to 25.

Construction

The construction tests add a new component to an already existing entity and measure the time to allocate it. The component has a default constructor, with no logic.
Four different setups are tested to test internal behavior of the system:

- **Empty**: Each component is constructed on its own empty entity with the registry for the component type being entirely empty.
- **Reserved**: Each component is constructed on its own empty entity with the registry being empty, but having pre-allocated all the necessary registry blocks.
- **With holes**: Each component is constructed on its own empty entity within a registry where every other component slot is allocated in every block. Space is pre-allocated, because of the pre-existing components between the holes.
- **With 4 siblings, Reserved**: Each component is constructed on an entity that already has 4 other components of the same type, and room is pre-allocated for all components (5000 in total).

Destruction

The destruction tests destroy a component in four different setups:

- **No holes**: Each component has its own entity and there are no holes or other components between them.
- **With holes**: Each component has its own entity, but between every component there is a hole.
- **Last of 5 siblings**: Each component is the last component on an entity with 4 other components of the same type. The destroyed components are constructed last and thus have no holes between them.

- **5 components together**: Components are grouped together on 200 entities in sets of 4, and destroyed by destroying the entire entity (5 components in total). There are no holes.

Iteration

The iteration tests iterate all components of the given type, and for each component it increments the first byte by 1. The byte is marked `volatile` to ensure the iteration is not optimized away.
Three different setups are tested:

- **No holes**: Components are tightly packed with no holes in between.
- **1-1 holes**: After every component there is a hole of a single entry.
- **5-5 holes**: After every 5 components there is a hole of 5 entries.

4.2.2 Transform system

Four different uses of the transform systems are tested: construction, destruction, updating and computation. The transforms are added to their own entity, so that component siblings will influence the results as little as possible. The construction and destruction of these entities are not included in the tests.
All tests run on 240 different transforms across a multitude of hierarchies of varying types. The hierarchies vary how many layers they have, and how many children each transform has. Following 4 types of hierarchies tested are

- **Small**: root with 1 child (120 hierarchies with 2 transforms).
- **Medium**: root with 3 children, each with 2 children (24 hierarchies with 10 transforms).
- **Large**: root with 3 children, each with 3 children, each with 3 children (6 hierarchies with 40 transforms).
- **Wide**: root with 30 children (8 hierarchies with 30 transforms).

Construction

We test the construction of 240 transforms in each of the different hierarchy setups. For example, we construct 24 *Medium* hierarchies. The construction includes both the construction of the transform and adding of the transform to its appropriate parent.
We test 2 different setups:

- **Top down**: Parents are created before children, and children are added when they are created.
- **Bottom down**: Children are created before their parents, and they are added to their parent immediately after the parent has been created.

Destruction

We test the destruction of 240 transforms in each of the different hierarchy setups. Destroying a transform will implicitly also remove it from its hierarchy, so both are included in the measurements.
We test 2 different setups:

- **Top down**: Parents are destroyed before children.
- **Bottom down**: Children are destroyed before their parents.

Updating root

We test how long it takes to change the translation, rotation and scale of 240 roots of different hierarchies. This is to gauge the cost of requiring a transform to mark itself and its descendants dirty. It does not include construction or computation of transform matrices. There are no variations to this setup.

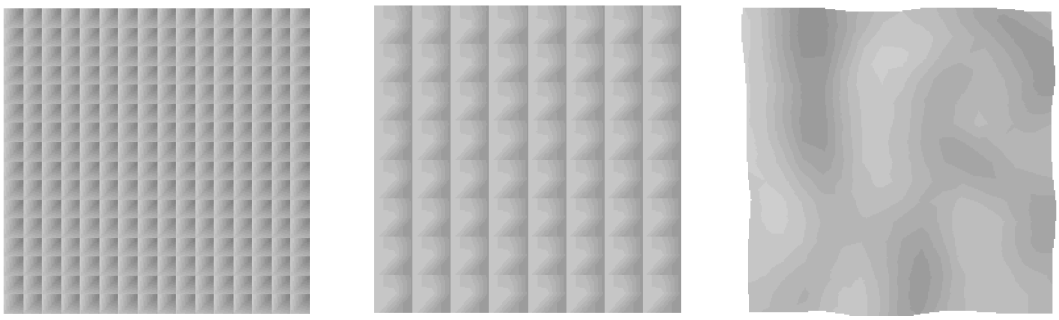
Requesting matrix

We test how long it takes to request the transform matrices of 240 transforms setup in the different hierarchy types. Depending on the setup, requesting may cause the computation of the transform's own matrix and that of its ancestors, or it may use cache. The matrix is computed using the GTE acceleration, but it is not used for anything.
We test 3 different setups:

- **None dirty, top down**: Request parent matrix before children, but with no changes having been made to any of the transforms.
- **All dirty, top down**: Request parent matrix before children, when all transforms have changes to translation, position and rotation.
- **All dirty, bottom down**: Request child matrix before parent, when all transforms have changes to translation, position and rotation.

4.2.3 Model rendering

To test the rendering of models we render a multitude of models that in total contain 512 triangles. We render 100 frames and take the average.
The models are wobbly planes, and together they take up almost the entire screen. They are non-overlapping, and they face towards the camera, so that all triangles are visible. The three plane types differ in how many triangles they contain as shown in Figure 4.1.



(a) 256 planes of 2 triangles **(b)** 64 planes of 8 triangles **(c)** 1 plane of 512 triangles

Figure 4.1: The three variations of planes used for measuring model rendering. Smooth shaded here, so they are visible

Each of these plane types are tested in three different setups:

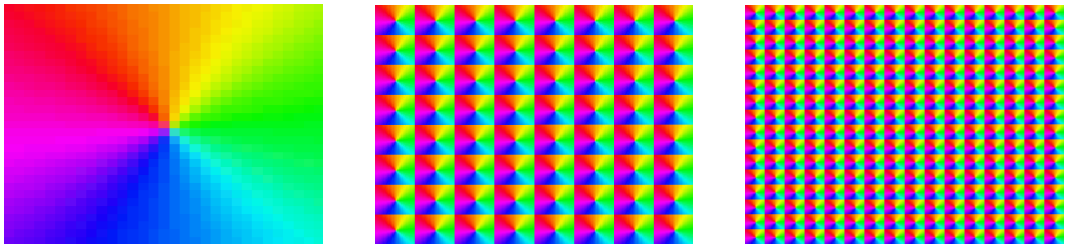
- **Flat**: The planes are flat-shaded and untextured.
- **Textured**: The planes are flat-shaded and textured with a 128x128 texture in 8-bit palette mode (256 colors).
- **Smooth**: The planes are smooth shade and untextured.

We test both the time it takes for the render system on the CPU to *submit* the triangles, and how much time it takes for the GPU to *draw* the triangles.

We do an additional separate test, where the planes are placed so far away from the camera that all triangles are culled. As no triangles are drawn, it does not matter what kind of shading they use or what whether they are textured, so we only test flat-shaded, untextured triangles, and we do not measure the GPU's draw time.

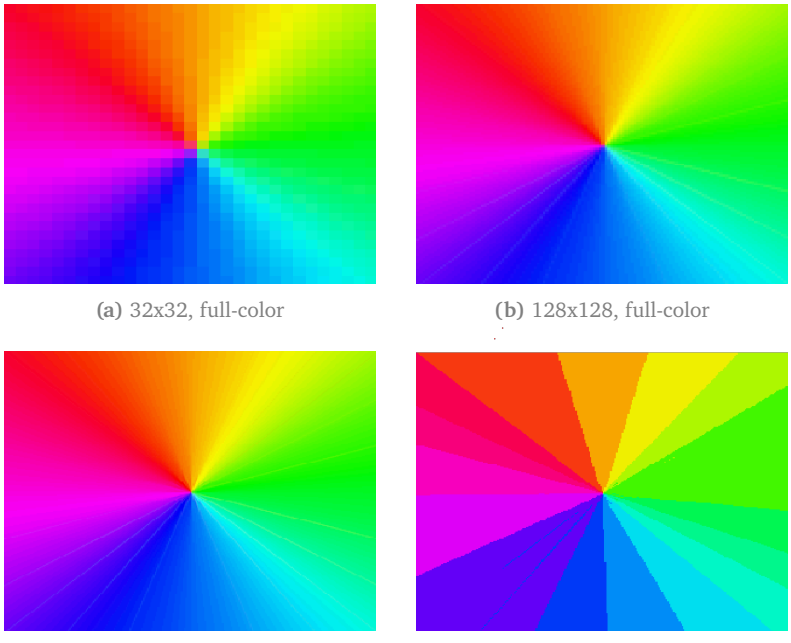
4.2.4 Sprites rendering

We test rendering of textured sprites by rendering a varying number of sprites and texture types that take up the entire screen. The number of sprites tested are 1, 64 and 256 (see Figure 4.2), and the four different textures used are full-color in sizes 32x32, 128x128 and 256x256 4-bit palette in size 256x256 (see Figure 4.3).
Just like with model rendering, we measure both the time it takes to submit the sprites by the render system on CPU, and the time it takes for the GPU to drawn them.



(a) 1 sprite **(b)** 64 sprites **(c)** 256 sprites

Figure 4.2: Screen full of differing number of sprites used for benchmarking sprite rendering using 32x32 full-color texture. Texture source: [30].



(a) 32x32, full-color **(b)** 128x128, full-color

(c) 256x256, full-color **(d)** 256x256, 4-bit palette (16 colors)

Figure 4.3: Single sprite taking up entire screen using the four different textures used for benchmarking. Texture from [30].

4.2.5 Text rendering

Given the limited feature of text rendering, only a single test is run, where we render a single text component with 1200 glyphs on the screen, which is many as can fit on the screen (see

Figure 4.4). The test renders 100 consecutive frames and take the average, and both the submission time and drawing time are measured.

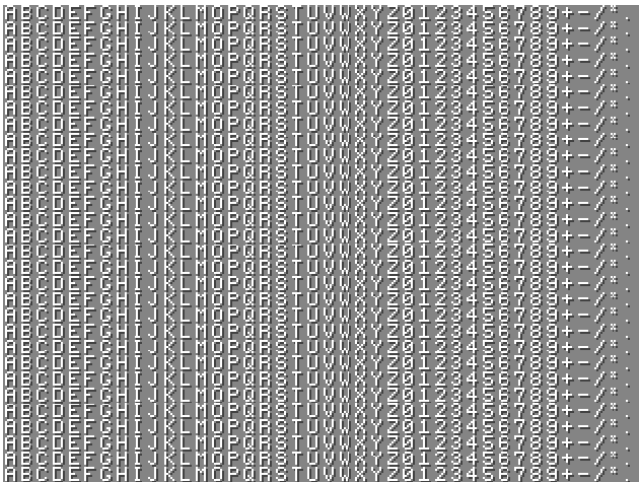


Figure 4.4: The rendered result of the text rendering benchmarks, consisting of a single text component rendering 1200 glyphs.

4.2.6 Comparison and summary

To compare the different features and their cost relative to each other, we derive summary metrics from the other benchmarks. For each system, we compute a single value for each operation normalized to a single object. The value is the average of the measurements by DuckStation and Redux. As a result, have the follow derived summary measurements:

- **Entity system**: Normalized to a single *Medium* component
  - **Constructing with 4 siblings**
  - **Destroying last of 5 siblings**
  - **Iterating in 1-1 interleaved holes**
- **Transform system**: Normalized to a single transform in *Medium* hierarchy
  - **Constructing top-down**
  - **Destroying top-down**
  - **Only one option here**
  - **All dirty, top down**
- **Model rendering**: Normalized to a single model with 8 triangles
  - **Submit flat, untextured model**
  - **Draw flat, untextured model**
  - **Cull only one option here models**
- **Sprite rendering**: Normalized to a single sprite that takes up 1/64th of the screen
  - **Submit with 32x32 texture**
  - **Draw with 32x32 texture**
- **Text rendering**: Normalized to 100 text glyphs

4.3 Nordic Game Jam

To test if the engine is at all feasible to use for a game in practice, we attempted to make a game with it for a game, together with a small team.

The jam was the *Nordic Game Jam 2024* taking place between April 19th and 21st. It is a 48-hour jam hosted in Copenhagen, and more than 500 people participated in the jam, submitting close to 100 games.

The team for the jam was assembled beforehand, and besides me, it included 3 other people. One was a last-year 3D graphics student, one was a last-year computer science student with an interest in engine development, and the last was a graduate of film and media studies. The latter had no experience with game development prior to jam, but helped out producing music, 2D visuals and some ad-hoc work. None of the team had any knowledge or training about the engine before the jam.

The version of the engine used for the jam closely resembles the final engine, except for very few convenience features, performance improvements and bugs that have been fixed since (further elaborated in Chapter 5).

4.3.1 Preparing the workflows

For the jam, it was expected that a significant part of the issues would revolve around the workflow of the engine. Especially, setting up the project, such that everyone could play the game via emulator, and the process of adding new assets. Because of these worries, we spent some time preparing some tooling for this.

To make the project setup easier and faster, we set up a *project template*. This template included almost all the programs and tools the developer needs, such as the SDK, CMake and the various external tools for generating assets, as well various custom easy-to-build build scripts.
Besides the template, the user only had to install the emulator (PCSX-Redux) and Visual Studio Code, along with a single extension for it. PCSX-Redux required that a few settings were as well.
The project template is included as Appendix B.

The engine uses the *VAG* and *TIM* file formats for audio and textures respectively that were also used by the original SDK. Third-party programs to convert from more popular formats, such as *JPEG* and *WAV* files, are included in the template. For 3D models, a custom program was written that converts 3D models from the *OBJ* format to a custom format made format for the engine.
All the file conversion programs, or *asset compilers* as they are named in this work, are command-line interface programs. Even for someone used to using such tools, this is cumbersome, as the argument format differs, the options are not clear, and you have to run them individually for every asset. The latter is especially problematic when the compiler is updated, such as when a bug is fixed in the model-compiler, and you then have to recompile all model assets.
To ease this workflow, we created some CMake tooling, allowing the developers to simply specify the assets in a CMake file. The assets are given in the format shown in Codeblock 4.1.

Where the options are specific to the asset type and may be optional. A full example is given in Codeblock 4.2.

The assets are then compiled to their proper asset type, and the asset is further embedded into the built program, and a byte array symbol is added referring to this array. This is all

```
type name path option1 option2 ...
```

Codeblock 4.1: Format in CMake when including an asset into the game

```
set(ASSETS
    MODEL mdl_player models/player.obj

    TEXTURE tex_sprite textures/sprite.png PALETTE8

    SOUND snd_music sounds/soundtrack.wav LOOP
    SOUND snd_jump sounds/jump.wav
)
```

Codeblock 4.2: Example of how to define assets to be compiled and included in the game.

happens automatically when building, but the developer does, however, have to deserialize the asset to a C++ type specific to the asset type in their game, but this is a single function call per asset.

Chapter 5

Results

In this chapter we illustrate the results of the benchmark and game jam experiments described in Chapter 4. Additionally, we analyze the results and argue for their expected and unexpected patterns.

5.1 Feature benchmarks

The results of each test are shown as a bar graph with both the measurements from both *DuckStation* and *Redux*. Generally, the two are expected to be slightly different, so we include both values in our considerations. The minimum and maximum values are the minimum and maximum in set of values that includes both DuckStation and Redux. Beyond this we only point out cases where there is a significant difference between the two.

5.1.1 Entity system

Constructing components

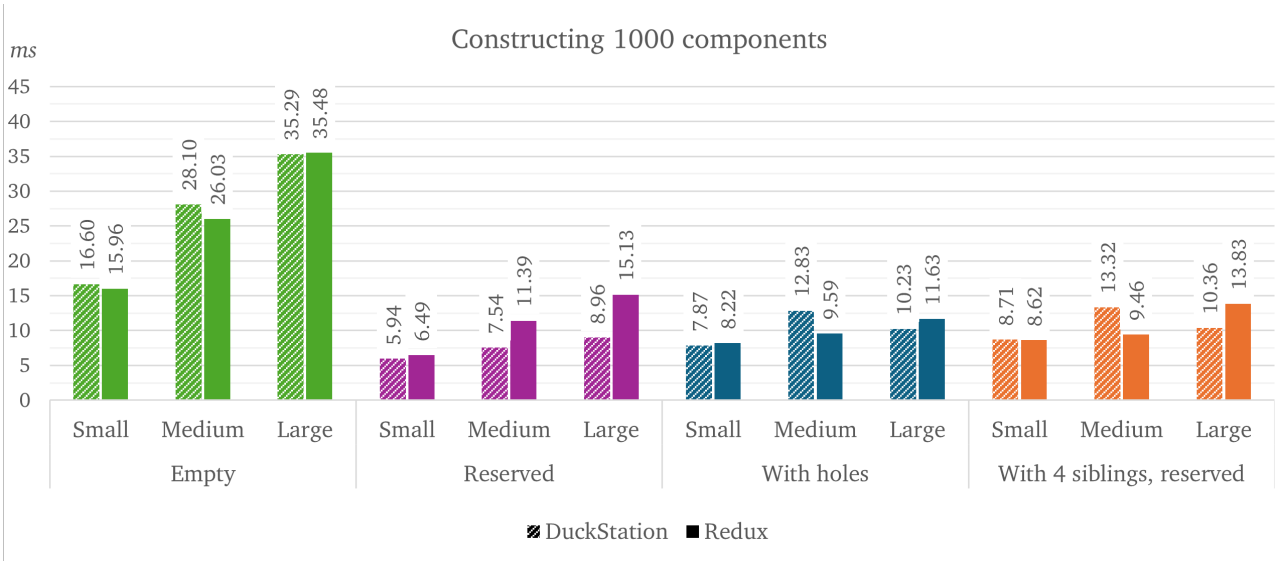


Figure 5.1: Benchmark results of the time it takes to construct 1000 components in registries with varying size of interleaving holes.

Figure 5.1 shows that it takes ~5.9-35.5 milliseconds to construct 1000 components. Constructing small components on individual entities in a pre-allocated and empty registry blocks

are the fastest, and this is expected to be the most performant scenario in which to construct a component.

Whether the registry blocks are pre-allocated is by far the most influencing factor, and it is almost 3 times faster to construct components with pre-allocated blocks for the medium and large components. The reason is believed to be that allocating a new registry block will do 2 - 3 dynamic heap allocations.
If registry blocks are kept allocated this is a one-time cost, and it will afterwards end up in a situation like that of the other results. In other words, this poor performance is likely only for setups and loading. Also, the registry block size can either be tweaked or pre-reserved by the user if they expect this behavior. In both the setups with holes and with 4 siblings, the blocks are pre-allocated, and so the difference lies only in the overhead of fixing holes or appending to sibling list.
In this test there is reason to believe that Redux' results are off. In theory, it should take *at least* as long to add a component with a sibling as a component without any siblings. The same can be said for holes. However, Redux shows that it is actually faster. One reason for this behavior could be the effect of the instruction cache, but that should only affect the first constructed components. DuckStation on the other hand *does* follow the expected behavior, so in this test we disregard Redux.
While significantly less than the effect of not pre-allocating, holes and siblings still have a noticeable effect on constructing entities with the allocating Medium components with siblings taking ∼1.75 times longer. Unlike the first setup however, these scenarios are expected to be more common while the game is running.

Destroying components

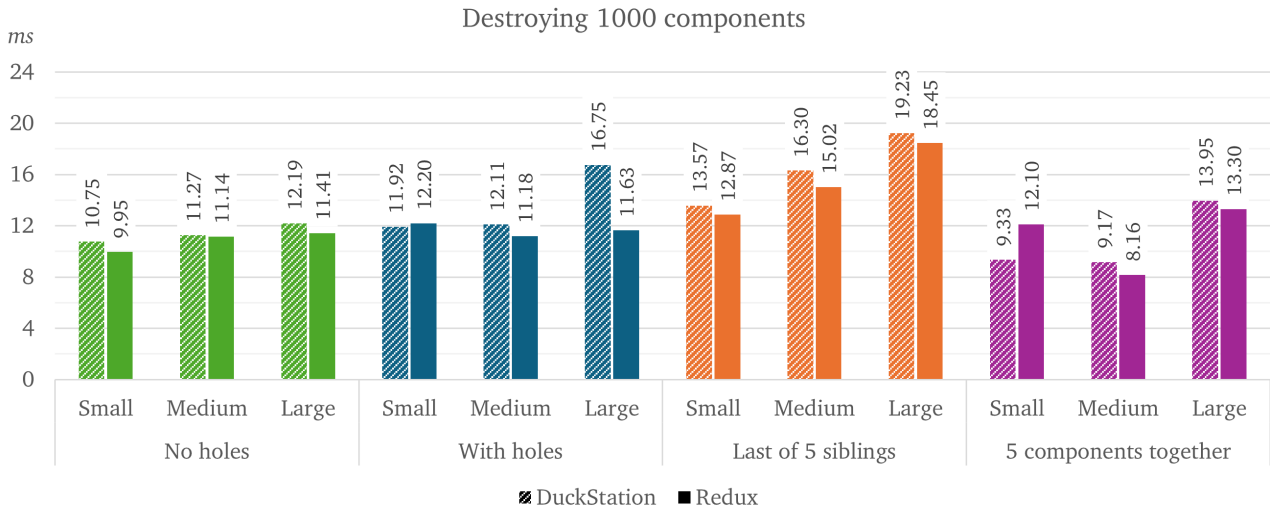


Figure 5.2: Benchmark results of the time it takes to destroy 1000 components in registries with varying size of interleaving holes.

Figure 5.2 shows that destroying 1000 components takes ∼8.7-19.2 milliseconds depending on the scenario, or about 8 microseconds or about 280 cycles per destroyed component at the very least.

Generally, it seems that whether components exist in holes, and thus have to merge holes, only matter slightly. This is apart from DuckStation's measurements for large components, where it takes substantially longer. It is suspected that this may be because memory's burst operations are utilized inefficiently for large components, and that this is somehow modelled

differently than in Redux. But generally, the behavior verifies that hole management only result in a slight, constant overhead.

It is expected that in practice, components will either be destroyed few at a time, such as when you remove a single temporary behavior from a game object, or all together, such as destroying the entire game object.
The first can be a fairly slow operation, as illustrated by the difference between the first and third setup. While the first setup represents the lower bound, the second does not represent the upper bound. Because all components of an entity are iterated when component is destroyed, increasing the number of components should increase the time linearly. It does, however, also mean that it is predictable, and thus can be mitigated by keeping entities smaller.
Whether deleting entire entities is faster or slower than deleting them on individual entities (first setup) is not as clear a cut as the fourth setup shows. It is slower in as many cases as it is faster. However, it is clearly faster than deleting a sibling from a group of 5, which is its primary purpose.

Iterating components

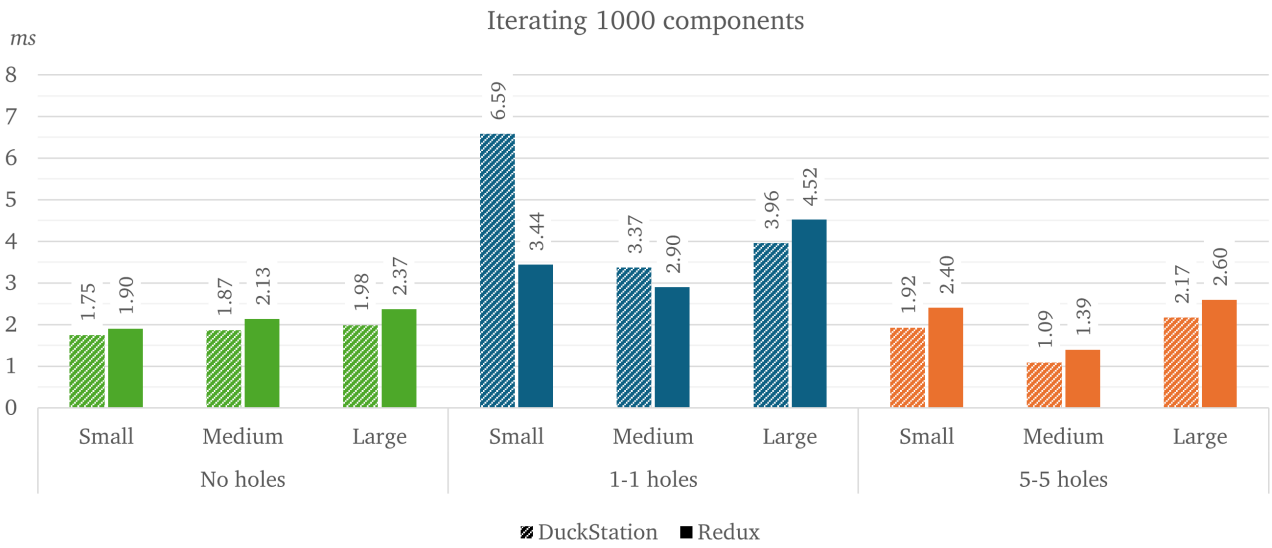


Figure 5.3: Benchmark results of the time it takes to iterate 1000 components from registries with varying size of interleaving holes.

Figure 5.3 shows that iterating 1000 entities takes ∼1.1-4.5 milliseconds.
DuckStation and Redux are fairly close in the results, except for iterating small components with small holes. It may be a result of how memory is accessed, but accessing holes of size 1 should be similar to iterating twice the number of components. No reasonable explanation for this behavior was found, and given behavior of the others seem fairly consistent, we consider this an emulator anomaly.

The single-entry holes take up to twice the time as the setup with no holes, which makes sense. Jumping holes still require checking the first entry in the hole. Since, the holes has only a single entry, jumping has no effect, so we are essentially checking twice as many components.
The setup with 5-5 holes takes significantly less time than the setup with single entry holes. This shows the effect of jumping holes, and that it is the number of holes that matter, not the size.

The size of the component seems to have a slight impact on the performance. It seems to be most effective on medium components, which may be because it is better aligned for bulk memory fetches.

These setups here are considered to cover the best case and worst case when iterating. You cannot introduce more holes in the 1-1 setup, and increasing their size should (in theory) not affect the iteration speed.
Thus, it is a fair estimate that it takes between 1 and 5 microseconds, or around 35 to 169 cycles to iterate an entity. While the system supports more than just iterating, this seems fairly slow compared to what one might imagine of iterating a single, or even multiple, contiguous array.

Effect of entity size

We can see that the size of the components has some effect on the results, but despite some behavior has been credited this size difference, we cannot say there are any clear patterns in how the memory size affect the results. It is likely that it is a small enough detail to not be that is not well-emulated, but this has not been fully investigated. It may also be that it is simply sensitive in these cases where we do many of the exact same, simple operations and that it may be too difficult to fully utilize in a more realistic setup. But one would need the proper hardware to make conclusions about this.

5.1.2 Transform system

Creating transforms

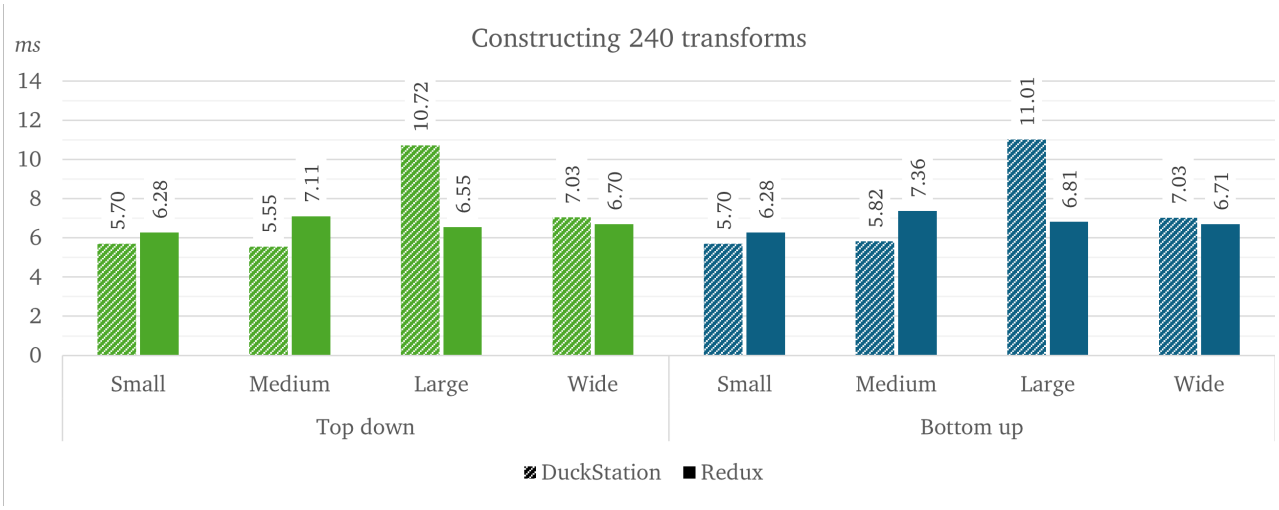


Figure 5.4: Benchmark results of the time it takes to create 240 transforms in and add them to varying types of hierarchies.

As the results show in Figure 5.4, it takes ∼5.5-7.4 milliseconds to create 240 transforms. This is if we disregard DuckStation's results with large entities. As transforms are added to their own entities, and that adding a transform to its parent is an operation of constant time complexity, we do not expect to see this kind of severe variations. As such, the DuckStation's result is disregarded. Doing this, the results show this intended behavior, in general.
We are not seeing a significant difference between constructing from the top to the bottom or vice versa. This is good as it makes it more flexible when creating hierarchies, and require less effort from the user.

One must remember that the tests do include the construction of the component on the entity. As we saw earlier this took at least 6 microseconds per component, and here it takes

22 microseconds per component. This does illustrate that creating transforms that must be added to hierarchies brings with it a significant overhead.

Destroying transforms

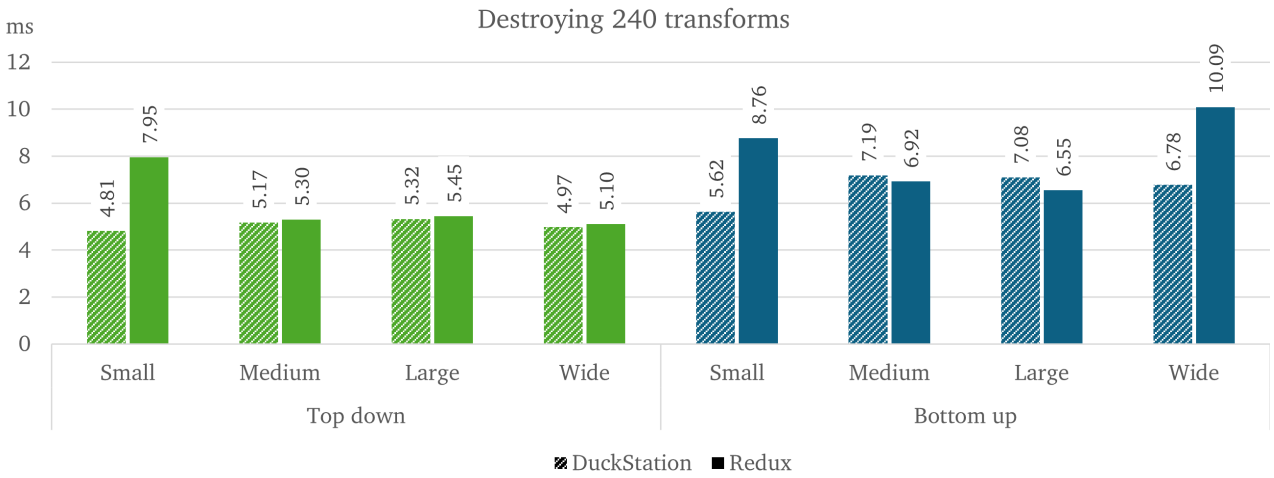


Figure 5.5: Benchmark results of the time it takes to destroy 240 transforms in varying types of hierarchies.

The results in Figure 5.5 show that it takes ~5-7.2 milliseconds to destroy 240 transforms in the various hierarchies. As with creating transforms, this is an operation of constant time complexity, and there should be little difference between the hierarchies. But it also has some suspicious results with Redux taking significantly longer on certain hierarchies, whereas DuckStation does not. Again, these are left unexplained and are considered emulation anomalies and are disregarded.
The results do hint at destroying top-down as opposed to bottom-up is slightly faster. This should in theory make little difference, and it may come down to the destruction of a child requiring an additional function call to remove itself from its parent.

Updating root transform

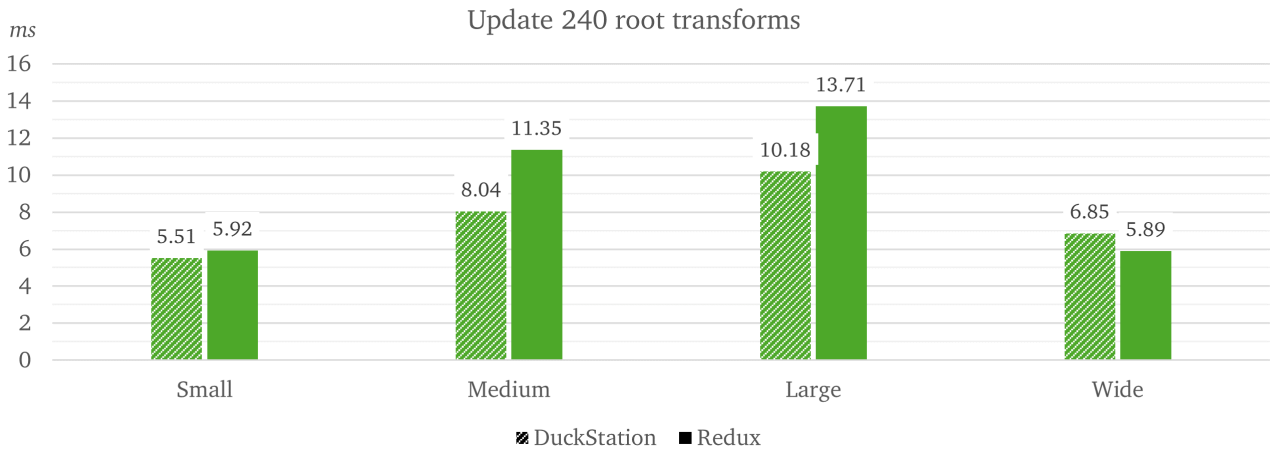


Figure 5.6: Benchmark results of the time it takes to update the scale, rotation and translation of 240 transforms that are roots of different types of hierarchies.

The results in Figure 5.6 show that updating the position, translation and rotation of the 240 roots in the different hierarchies takes between ~5.5-13.7 milliseconds.

In general, this is a very significant overhead for doing some otherwise simple operations - i.e. just setting the positions, translation and rotation with no computations. The cost of this comes down to the fact that a parent that gets its values changed must mark all its descendants dirty. This, naturally, takes longer for bigger hierarchies as the results also reflect.
The problem is made worse by it needing to do this marking of descendants when *every* property changes, meaning it does it three times in this test. There may be potential for optimizing this with a function to set all properties, or better reasoning of what is already marked as dirty.

Request transforms

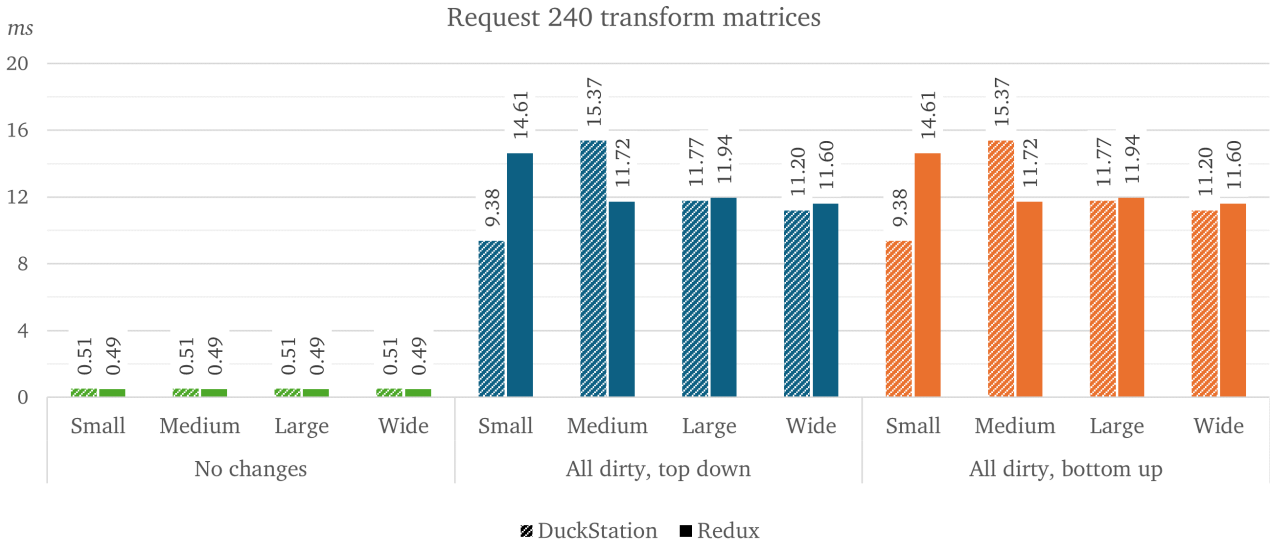


Figure 5.7: Benchmark results of the time it takes to request the transform matrix of 240 transforms.

According to the results in Figure 5.7, the time it takes to request 240 transforms varies significantly between whether the result is cached or not. A completely cached hierarchy takes ∼0.5 milliseconds and a completely dirty hierarchy takes ∼9.4-15.4 milliseconds. This goes to show the significance and importance of doing this caching, and the value gained from sacrificing the extra bytes for the cache.
Whether the matrices are requested from top and not bottom makes no difference in either DuckStation or Redux. That the timings are exactly the same is suspicious, but no faults in the experiments have been found. It may be a result of the actual matrix computation taking most of the time and the difference in the control logic between the different hierarchies is so small that it does not show.

5.1.3 Model rendering

Submitting models

As is shown in Figure 5.8, submitting 512 triangles take between ∼6.7 and ∼23.6 milliseconds.
It is by far mostly influenced by the overhead of the model, as it takes more than twice as long to submit them if they are distributed across 256 models as opposed to one. A multitude of things may contribute to this overhead such as there being more entities to iterate, but the most significant is expected to be the computation of the model's view matrix and the lighting matrix.

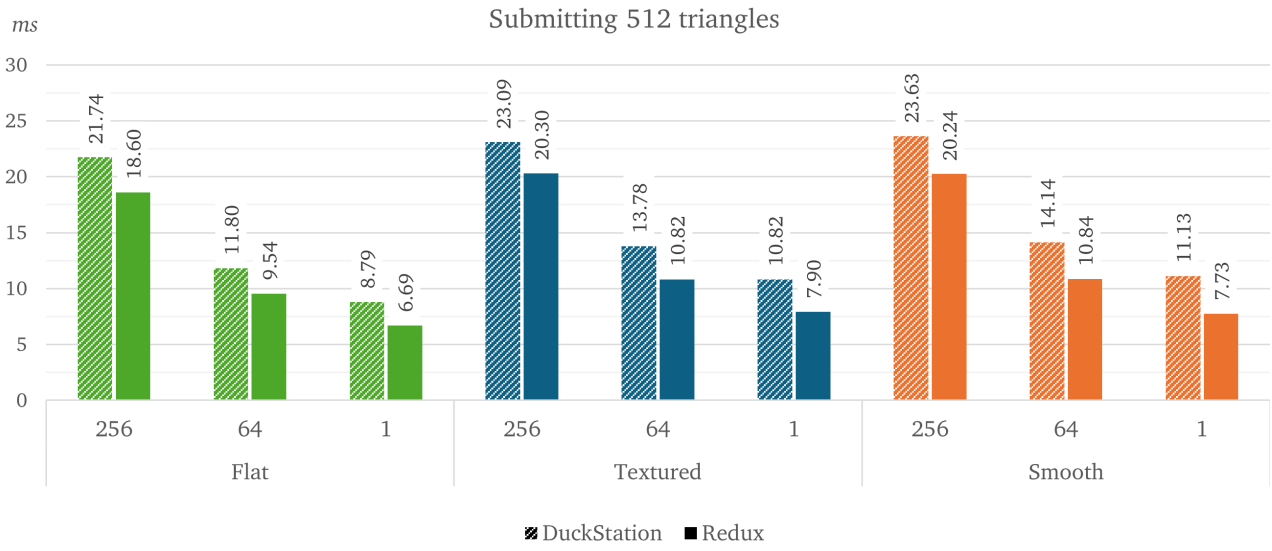


Figure 5.8: Benchmark results of the time it takes for the CPU to submit 512 triangles for the GPU to draw, through a varying number of models.

Whether textures and smooth shading is enabled has very little influence on the submission time. This is sensible as the only difference are a few assignments and small computations.

Drawing models

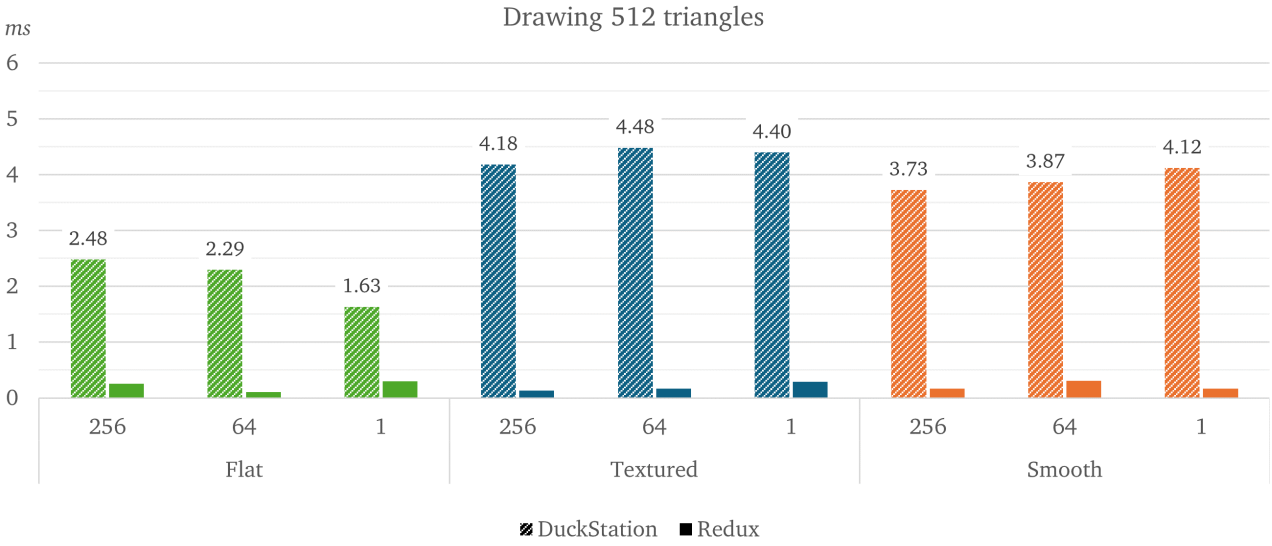


Figure 5.9: Benchmark results of the time it takes for the GPU to draw 512 triangles through a varying number of models.

Figure 5.9 shows how long it takes for the GPU to draw (rasterize) 512 triangles. PCSX-Redux does not emulate the drawing at all, which is clear in these results as it takes very little time. Consequently, we only look at DuckStation.
Compared to the submission, the difference lies in the opposite parameters. That is, it takes about the same time to draw 512 triangles from 1 model as 256 models. This is sensible as the drawing has no notion of the models. Instead, the results show that there is a significant difference in drawing textured or smooth shaded triangles. This is in line with other sources [1]. One must remember that the texturing and smooth shading are likely accumulative,

making them up to 4 times slower drawing non-textured flat shaded. Also note that this is not a bottleneck of the engine, but rather of the GPU.

Culling triangles

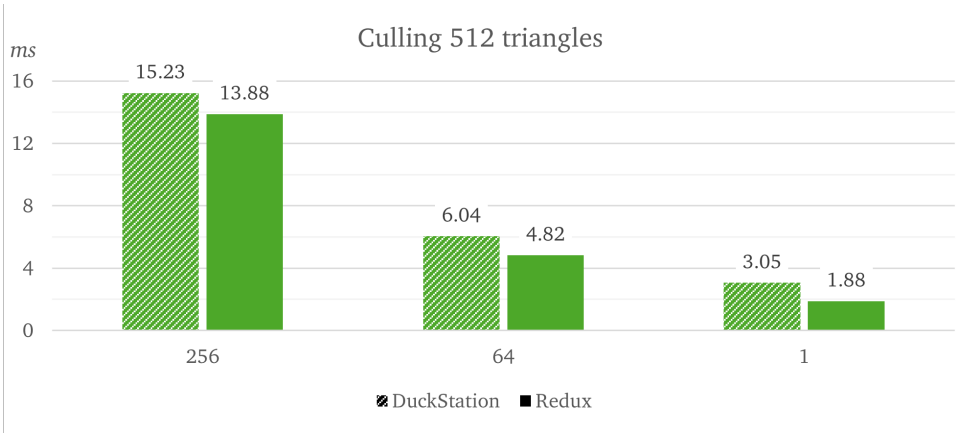


Figure 5.10: Benchmark results of the time it takes to iterate a varying number of models with a total of 512 triangles when all triangles are culled.

Figure 5.10 shows that for the render system to iterate, but discard 512 triangles takes ~1.8-15.2 milliseconds, with the slowest being to render 256 models. This makes sense given that the 256 models still must compute and set the model's view and lighting matrix. It hints at there potentially being performance to be had, if the models can be culled at an even earlier stage in the render pipeline.

5.1.4 Sprite rendering

Submitting sprites

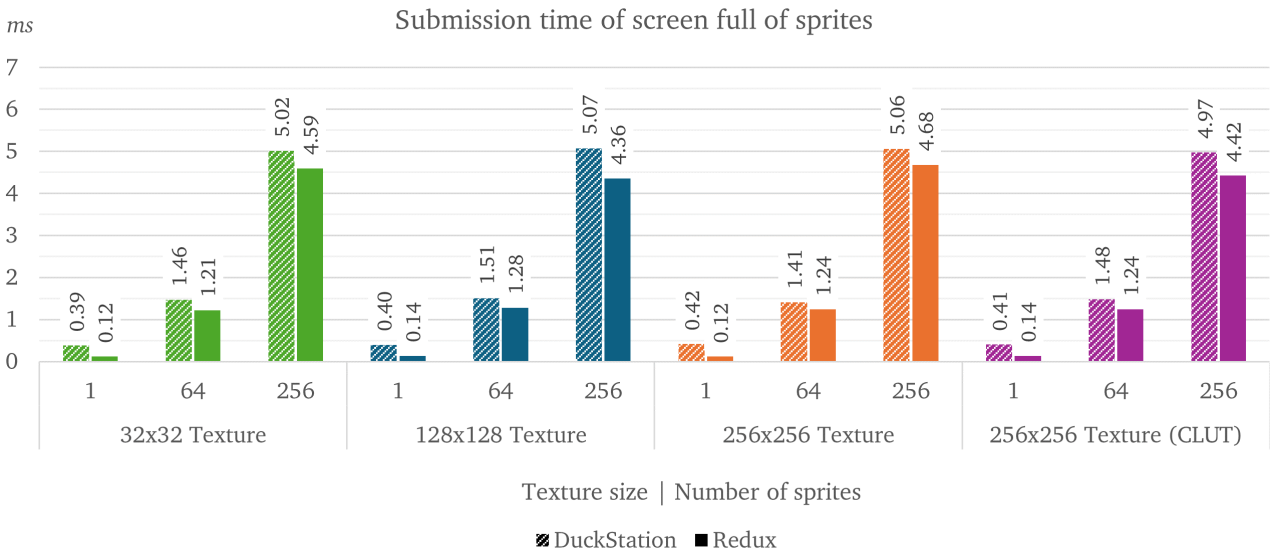


Figure 5.11: Benchmark results of the time it takes for the CPU to submit a varying number of sprites and varying types of textures for the GPU to draw.

Figure 5.11 shows that it takes ~0.1-0.4 milliseconds to submit 1 sprite, ~1.2-1.5 milliseconds to submit 64 sprites and ~4.4-5.1 milliseconds to submit 256 sprites. That is around 400, 23.5 and 20 microseconds per sprite. That the last two are fairly close indicates that

the majority of the time spent on the first is the overhead of the general render pipeline. This will of course be shared among all rendered graphics, which is why the last two are faster per sprite.
The type of texture makes no noticeable difference, as the submission drives no logic based on this.

Drawing sprites

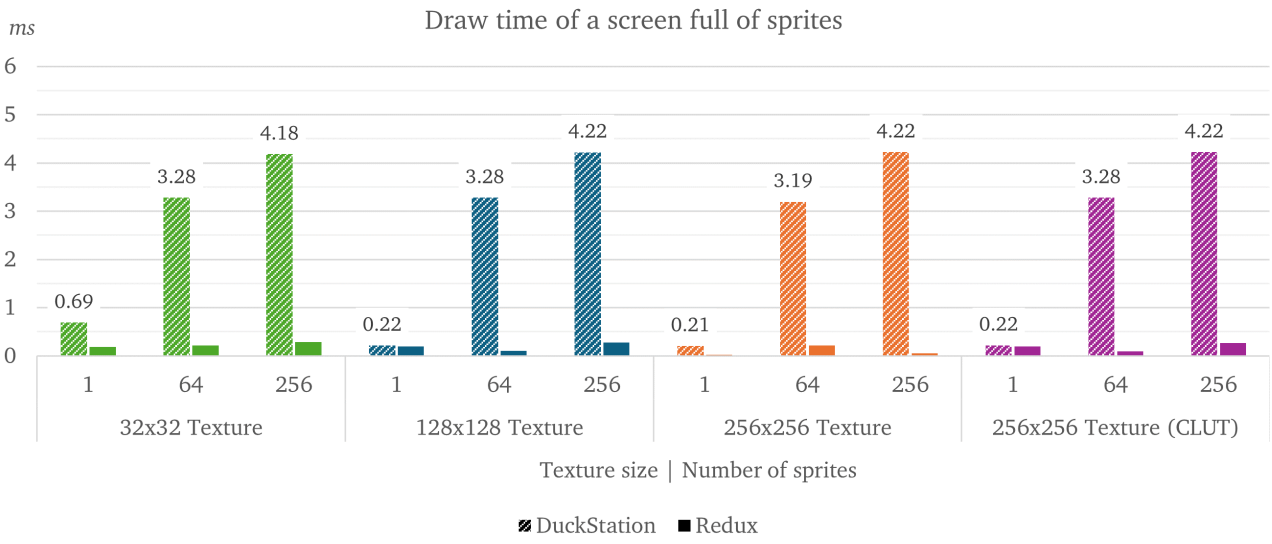


Figure 5.12: Benchmark results of the time it takes for the GPU rasterize a screen full of a varying number of sprites and varying types of textures.

Figure 5.11 shows that it takes ~0.1-0.7 milliseconds to draw 1 sprite, ~3.2-3.3 milliseconds to draw 64 sprites and ~4.2 milliseconds to draw 256 sprites. Like with drawing of models, PCSX-Redux' drawing time is not emulated property, so we discard it.
Even for the drawing, the type and size of texture seem to make no difference. This is unexpected, and whether this is a trait of the emulator, or if this mimics the hardware properly is unknown.

5.1.5 Text rendering

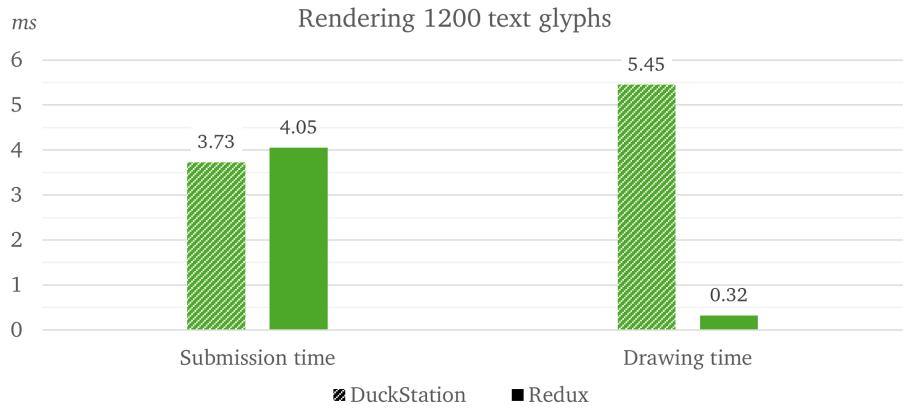


Figure 5.13: Benchmark results of rendering 1200 text glyphs as 1 text component, including submission time (CPU) and drawing time (GPU).

Figure 5.13 shows that it takes ∼3.7-4.1 milliseconds to submit and ∼5.5 milliseconds to draw 1200 text glyphs. Like with drawing of models and sprites, PCSX-Redux' drawing time is not emulated property, so we discard it.
While text is conceptually the same as *sprites*, this experiment shows the potential in axis-aligned quads and grouping of sprites, as it is significantly faster. This is both in regard to submitting, but also drawing.

5.1.6 Comparison

Figure 5.14 shows a comparison and summary of a subset of the different setups of all the system features tested in this section, as was explained in Section 4.2.6.

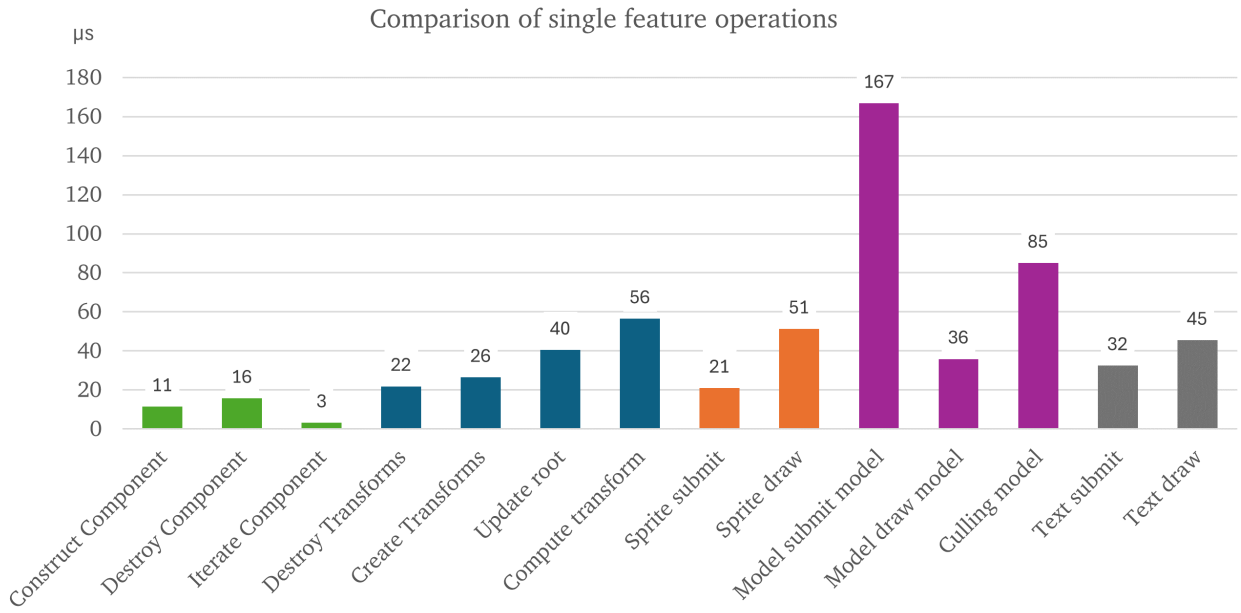


Figure 5.14: Comparison of performing operations of various features on single objects based on the results in the more extensive benchmarking results.

The primary thing to note here is the significant difference between component and transform related operations, compared to submitting models. Even culling of models it is expensive. This is even optimistic scenario, as the model in question contains only 8 triangles. Comparatively, a cube contains 12 triangles, though only 6 will be visible at a time.
It is less expensive to submit sprites and text. Additionally, these primitives are expected to be primarily used for UI, and exist in fewer numbers. So, it is expected that they only take up a marginal proportion of the frame time.
The cost of drawing models is less than that of submitting models, but that is expected to increase as the triangle count of the model is increased. But considering this, and the fact that the submission of models must share resources with the other operations, it does hint at the GPU not becoming the bottleneck for a game.
Transforms are expected to be directly related to the number of models rendered; one model likely related to one transform. Given their cost, they will further limit the rendering capabilities. Other components, however, are not necessarily linked to a model, and so it is for there to be many more components than models. Additionally, more than one iteration is expected to occur during simulation, and so the relative small cost of this operation is expected to accumulate. The tests here, of course, do not take into account the logic that is to be executed in the iteration of components.

5.2 Nordic Game Jam

At Nordic Game Jam, our team succeeded in creating a game for the original PlayStation in 48 hours with the Tundra game engine. However, we missed the first afternoon and evening due to external circumstances, and we ran into a multitude of issues with the engine. During the showcase people showed great interest in the project - not because of the quality of the game, but because of curiosity towards the PlayStation.

5.2.1 Resulting game

The resulting game is named *Color Cannon Coop*, and is a 3D game where two players collaborate to take down an enemy cannon by reflecting its projectile. The projectiles take the shape and color as the logo on one of the 4 buttons on the right-hand side of a PlayStation controller, and to reflect a projectile you must press the matching button when the projectile reaches you. However, this will only reflect the projectile to your team-mate, and it will change shape and color of the projectile. This may come in handy because in order to damage the enemy cannon, you must reflect the projectile that matches the shield of the cannon. When the cannon is hit, the shield rotates to a new shape.
The game's *itch.io* page can be found at [7], and a screenshot of the game made in the jam can be seen in Figure 5.15.

Figure 5.15: Screenshot of the game, Color Cannon Coop, made at Nordic Game Jam.

5.2.2 Issues

During the jam, multiple bugs were encountered:

- Texture brightness on models was off, so we had to pre-brighten the texture, loosing significant detail in the process.
- Depth was not mapped properly, preventing us from inserting a ground plane.

- Random errors in the entity system causing occasional, unpredictable crashes
- Having more than one active text component caused crash
- Failed to load models with textures larger than 256x256 pixels.

These bugs had two consequences: 1) some bugs were preventing us from using certain features, so time was spent trying to fix them, which meant less time for developing the game, and 2) we succeeded in fixing very few of them, so some features could not be used.

After the jam, time was spent fixing these, and during that multiple other bugs were discovered:

- Model UV coordinates were off, so some models would have texture-bleeding between triangles.
- Debug assertions were not removed from release builds, impacting performance.
- Dead but not freed components were included in iteration.
- Transforms were not updated properly changed when updated.
- Other significant performance issues.

5.2.3 Updated game

Because of the issues, we ended up having a game that included the core mechanics but lacked some things to bring it all together into a proper game. Given the issues were with the engine, and that we had gotten a late start to the jam, we decided to spend some time after the jam to update the game. We spent about 4 hours per person (not equally distributed). Before updating the game, we fixed the mentioned issues with the engines, which is not included in the estimated time.
A screenshot of the final game can be seen in Figure 5.16.

Final feature list

With the updates, the game uses every feature provided by the engine. Some of the most notable are as follows:

- Various different entities with multiple components, using the iteration API to apply logic to them
- Multiple hand-made 3D models with textures with varying resolutions, with a total of about 700 triangles
- Multiple sound effects and a 24-second-long soundtrack
- Ambient lighting and 3 directional lights, two of which change color to match the color of the current shield
- Menus with game logos and text, along with in-game health sprites and text showing player and boss health
- Dynamic transforms for players, projectile and cannon, including use of hierarchical setups for the cannon, as well as static transforms for the ground geometry.
- 2D player movement limited to a playable area, controlled with individual controllers and using most buttons, including button combinations

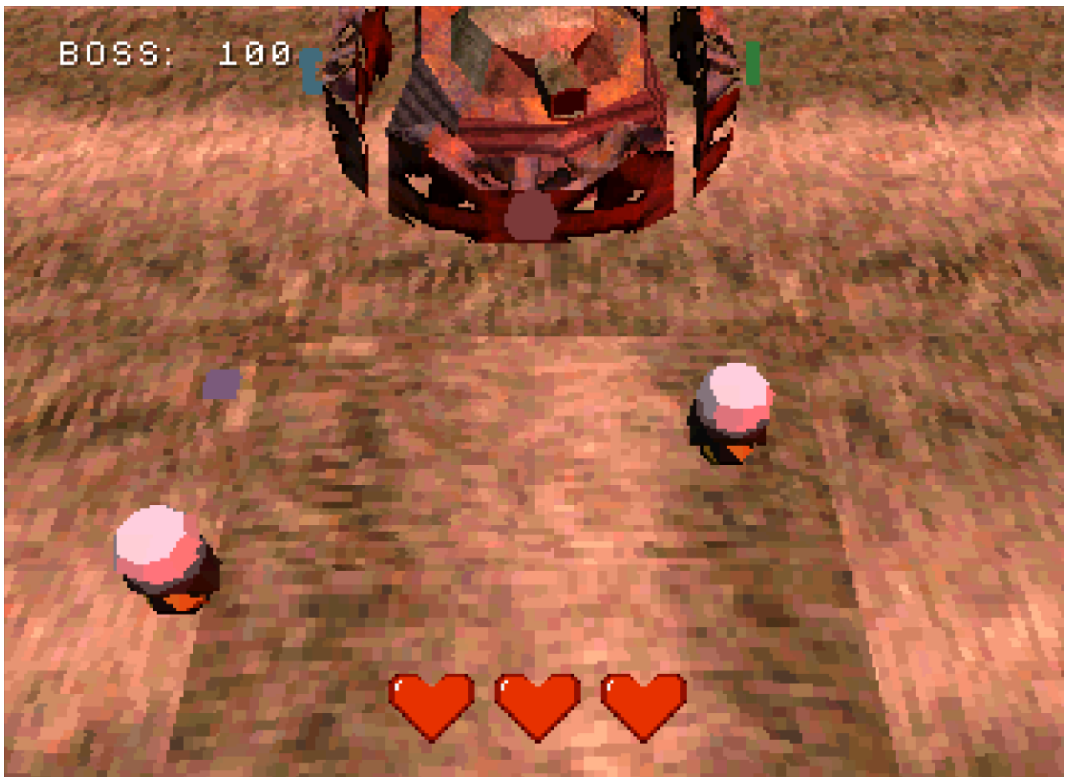


Figure 5.16: Screenshot of the game, Color Cannon Coop, with revisions made after Nordic Game Jam.

Chapter 6

Discussion

The experiments have been completed, and the results have been processed and analyzed. Before discussing what these results mean for the research question, we comment on some things that have an influence on the answer. This includes the quality of the experiments, potential optimizations and effects of missing features and changes.

6.1 Quality of the experiments

While multiple experiments have been done in this work, some words must be said about their quality, how viable it is to extrapolate the conclusions made from them to the original hardware.

6.1.1 Emulator instead of hardware

In general, the experiments are very unlikely to be completely accurate, and there is a chance for some things to be significantly off. But there are some arguments to be made for the results being a rough estimate of the same results achieved on the hardware.
For one, the timings seen in the benchmarks definitely do not resemble those of modern hardware. Looking at it with modern expectations, the capabilities are very poor. But at the same time, the emulators do perform well enough to run many of the original games, so it is not underperforming either. Also, while some hardware details may differ, the console still seem to act according to the same performance principles, so things that improve performance in the emulator will likely also have an effect on hardware.
There is also a chance that the functionality of the engine may differ on the hardware. For instance, if emulators ignore or do not handle scenarios that cause errors on the hardware. However, the emulators *have* been tested on games that were made for the original hardware. In particular, DuckStation has a long compatibility list of games with no apparent issues [8]. So, it is assumed, by extension, that the engine is also compatible with the hardware to the same extent.
In general, while the emulator is not an exact simulation of the hardware by any means, it seems unlikely that the results achieved from our experiments paint a picture that is completely different from the hardware.

6.1.2 Experimental coverage

The benchmarks attempt to cover the most direct use cases and paint a general picture of the engine's capabilities and performance. But there are many systems, many ways to use

them and many states they can be in. To gain full coverage of this would result in a many-dimensional parameter matrix that would not have been feasible to test in this work.
This means there may be combinations of states and uses that will cause some systems to perform very differently from what has been seen here. For example, no formal testing has been done for the size of the blocks in component registries, but anecdotal tests show that they may have a noticeable effect. It is, however, expected that the most common use-cases of the engine have been covered by the experiments, and that these unknown setups would rarely happen, only cause lag-spikes or be unintended behavior that can be fixed.
However, there are some obvious things that one might test that we did not because of the scope of the project. One is to gauge how much memory is used by the engine's features, as well as the engine's code. We have made many comments on optimizing memory, but only done high-level anecdotal tests on it. With only 2 MiB of main RAM, many games were limited by memory, which only underlines the importance of this.
The other is to gauge the performance of a concrete game in higher detail. This could for example be the Nordic Game Jam game, but preferably it would be a game that has higher demands of the hardware (see next section). Details investigated here could be some general metrics such as simulation time, render submission time, render drawing time and memory consumption used by various systems.

6.1.3 Game is not pushing the limits

The game developed for Nordic Game Jam, was developed in a very short process with a team with no experience with the engine. Furthermore, the engine had severe bugs and missing features. In order to get a real sense of how well a game based on the engine would perform, one would need more time, more experience, and a more mature tool. This could push the results in either direction, as it could either mean more ideas and features, but also better possibilities for optimization. But regardless, the experience at Nordic Game Jam does show that a game of at least this caliber is doable with the engine.

6.2 Future work

General optimizations

Much of the engine's fundamental design and implementation was argued for because of their influence on performance. But besides these fundamental and feature defining optimizations, limited work has been put into optimizing the lower-level implementations. This includes things like avoiding certain unnecessary computations, implementing key functions in assembly, such as computing transform matrices and model submissions, or just creating specialized functions for certain expensive operations, such as constructing entities and strings.
Some optimizations of this kind have been done, such as making the marking of descendant transforms dirty not using function-recursion. This particular example improved resulted in a speedup of 5 when updating the root transform in big hierarchies. It goes to show that putting some effort into this can achieve some very substantial performance improvements.

Link-time optimization

The compiler provided by psn00bsdk does not support *link-time optimization*. The lack of this prevents any optimization across translation units. For example, this prevents inlining of functions, such as the member functions of the custom string type. Certain key function definitions were moved to header files, to allow for this kind of optimization. For example, we moved the trivial function that gets a component's status flag, which doubled the speed

of iterating components. Originally, we paid little heed to this, as we expected to make use of *unity builds*[1] that mimic the behavior of link-time optimizations. But this was never implemented, and so link-time optimization can potentially bring substantial performance improvements.

Scratchpad

In Section 2.2.2 we mentioned the *scratchpad* which is 1 KiB of fast memory with load and store instructions taking a single clock-cycle. We make no use of this in Tundra, but it could be used to store frequently accessed items, such as the camera's transform matrix that is repeatedly used for every rendered model. Another example is to store pre-computed vertices for models, so that indices referencing the same vertex would not require doing multiple computations.

Axis-aligned world quads

As was shown by the performance difference between text and regular sprites, the use of *axis-aligned quads* bring significant improvements. But this is only comparing against sprites. Even in modern games, a common technique, called *billboarding*, is used to place 2D sprites in the 3D world that always face the camera. These sprites mimic 3D objects, but at a fraction of the cost, and is commonly used for environment such as trees. As the render system is particularly hard-pressed on rendering 3D models, this could bring some much-needed fidelity at a lower cost.

Use of CD drive

The CD drive of the PlayStation was one of its defining features, and so not making any use of it leaves a few opportunities on the table.
For one, it is possible to stream audio directly from the CD to the SPU without the use of main RAM or the CPU. This allows you to have minutes long music, dialog or FMV audio at 44.1 kHz (quality of CD albums) by using only the SPU and a few kilobytes of sound RAM for streaming. This leaves up room for more and higher quality sound effects in the sound RAM.
Because there is no support for CD, the engine currently requires that you load the entire game into RAM up-front, meaning it must fit into 2 megabytes of RAM. This includes both the source code, but also the assets. With CD support, one step above this would be to split the game into multiple levels. When starting a level, it is then loaded into RAM, allowing the level to take up to almost 2 megabytes of RAM, depending on the memory overhead of the engine APIs. Many original games, like *Twisted Metal* [61], did this [47, 24:00].
More advanced games, like Crash Bandicoot, used techniques where only parts of a level were loaded at any time, and the game would dynamically unload and load new parts as the player moved through it [1]. This had two benefits, one of them being that it allowed you to expand levels to take up as much memory as could fit on the CD, and the other being that you could limit the number of objects that had to be processed at a time.
It is imaginable that such a system could be implemented in this engine. However, it would require the engine to spend processing resources to perform this unloading and loading of assets and game objects. In particular, one would need to fit textures into VRAM at runtime, which can be an expensive problem to solve.

Pre-computing assets

In some original games they would pre-compute, or *bake*, certain assets before using them in the game, so runtime resources would not have to be spent on it. In Crash Bandicoot

[1] Building all the code as a single combined translation unit (see [28])

they pre-computed both sorting and culling of occluded triangles offline, with little runtime cost [1]. Pre-computing may also come in handy for producing acceleration structures for collision with static geometry, or even for lighting, which is a common use in modern games. Remember, the PlayStation only has native support for directional lights and not *point lights*. This can be computed in software with some acceleration from GTE, but it is costly. However, the tooling could support static point lights, such as a torch on a wall, and perform the entire lighting calculation outside the game, and embed the results as vertex colors.
Andy Gavin from Naughty Dog, speaks of the pre-computation process being painful and slow, and that *"levels often took several hours to process on our 5-8 machine farm"* [1]. However, a key-point with a modern engine for the PlayStation is that the tooling runs on significantly better hardware, and so these processes would be much easier and faster. Perhaps, it will even open up for other pre-computations that were avoided because of the lack of performant development hardware.

Collisions and physics

Most, if not all, original games had some kind of collision system, and many also showed various physics-like capabilities such as the platforming movement in Crash Bandicoot and Spyro the Dragon, or the driving in Gran Turismo. Even when developing the Nordic Game Jam game, we had to introduce collision logic for the projectiles.
That Tundra does not provide anything to support this is a problem. Not only from the user experience, but also because both collision and physics can take a significant toll on performance. While this is true, we may assume, given the low fidelity of the graphics, that the fidelity of physics would not have to be as high as in modern games. It may be possible to implement some simple collision and physics that the developer can apply to core objects, such as the player in a platforming game.

Model animation

Many games also included animation of both playable and non-player characters. In particular, in fighting games like *Tekken 3* [58] the movement of characters is central to the gameplay. All common general-purpose game engines also support such a feature, and it is central to a general-purpose game engine. While clearly not infeasible on the hardware, as seen from the many games that implement it, animation can be a performance-heavy system, and so it will further limit what is capable by other systems.

Less focus on the game object system

From the benchmarks it was found that the number of entities is likely going to be limited because of the cost of the behavior linked to certain components. In the benchmarks it was the rendering, but other behaviors such simulating collisions, physics and animation will likely add to this. While the entity system has an impact, the results does hint at the impact not being as important as first anticipated. In this light, too much focus was put on it, and the technical capabilities of the engine may have been better, if we had put more focus on the render system. Moving forward, we believe time is better spent seeking for performance opportunities in the render system, as opposed to the entity system.

6.2.1 Tools for the runtime

This work has focused on developing and experimenting with the *runtime* of a general-purpose engine and has not included *tools*. This was done for scoping reasons, and the original and long-term potential for the engine was indeed to include tools. So, a few comments must be made on how we might extend the runtime with tools and what their influence might be.

The tools of an engine can be designed and built in many ways. Some may include multiple programs that do different things, whereas others, from the user's perspective, consist of a single monolithic application. The latter is the approach of popular engines like Unity, Unreal Engine and Godot, and it is this kind of tooling that is envisioned for Tundra.

Game object hierarchies

It is very common for game objects to support some way of structuring them in a hierarchy, with objects being able to have a parent and children. The current system only allows components to be grouped together using entities. This can be seen as either having *siblings* or *children*, and thus no "hierarchy" exists. However, the core entity system in itself is very minimal, but that also makes it flexible. For example, a hierarchy can be modelled by having some *hierarchical* component with a reference to a child component, and potentially also a reference to a parent component. The group of this component would then be its siblings, which in turn could be entities, or just regular components. See Figure 6.1 for an example.

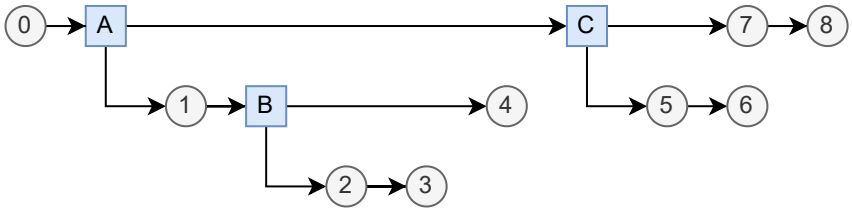


Figure 6.1: Visualization of a component hierarchy using *hierarchical* components. Each row is one component group or set of *siblings*. Squares are hierarchical components, and circles are other components. Arrows represent the references, but references from last sibling to first sibling (e.g. from 8 to 0) are left out for brevity

Very often, the logical hierarchy in the tooling, is reflected in the transform hierarchy, meaning that a child in the tool hierarchy is also a child of the transform. However, Tundra's game system does not tie transforms together with specific component or entity.
One way to achieve this on the tooling side could be for hierarchical components to come in two variants: one that is spatial and one that is non-spatial. The spatial variant would implicitly add a transform component to its list of children. When adding a new hierarchical component H in the hierarchy, the default transform parent of H's transform would be the transform of the first ancestor in the hierarchy that is a hierarchical component. The same would hold for separate transform components manually added. See for example Figure 6.2.

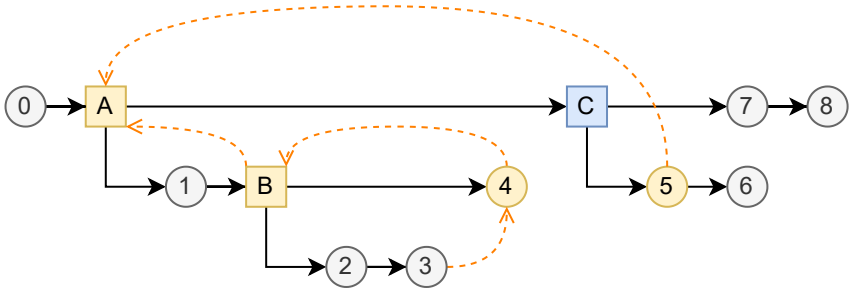


Figure 6.2: Example component hierarchy with transforms. Yellow squares represent spatial hierarchical components that has an implicit transform component (not shown) and yellow circles represents individual transforms. Yellow arrows represent transform parent references.

Game object serialization and reflection

An important aspect of the tooling is that game object setups produced by the user must somehow be serialized and deserialized at runtime. Deserializing in this case is about con-

verting game object data from some format, for instance *JSON*, to component objects. This requires that you have some sort of dynamic way to construct component instances at runtime. In C++, this is difficult, because of its notorious lack of built-in *reflection* features. For example, given the string name `"A"` of some unknown class `A`, you cannot construct an instance of this class without having some hard-coded mapping from this string to a function that constructs an instance of `A`.
The same lack of reflection is when you need to obtain information about the available component types within the tooling. There is no way to get meta-data about available types in a program and even given a concrete type there is very little information that can be gathered about that type.
One, not so useful, approach is to hard-code this information. For instance, the developer can define the name of the component, what members it has, a mapping between a name and some abstract construction function, and so on. This is, for one, very tedious work to have to do for every component defined by the developer, but it is also very error prone.
Another approach is to use external reflection libraries, of which there exists multiple [9, Metaprogramming]. But those libraries are very general-purpose, and we fear it would potentially bring in too much overhead - remember, that the deserialization has to be available at runtime.
It is also possible to write a C++ parser and code generator that can find all the available components and generate the reflection features needed based on that. But C++ is a complicated language, so this might be a very significant undertaking. It can be simplified, like what Unreal Engine does, where they require certain macros to be present on properties and actors that must support reflection [14].
What the best way of approaching this cannot be concluded without more in-depth research on the topic.

References

When using the tools it is likely that you want to be able to reference objects. For instance, either game objects referencing other game objects or assets. But currently, these references operate by pointers to memory addresses, and those addresses will not persist when going from tools to runtime. So, serializing a reference on the tools side and deserializing it at runtime, will yield invalid pointers.
This problem could be alleviated by using a different referencing scheme in the tooling than the runtime. For example, one could give all components a *globally unique identifier (GUID)*. When deserializing at runtime, these could then automatically be resolved to actual memory addresses.

Rendering and graphics

Importantly, the renderer is probably not going to see much change with the extension of tools, except for the earlier mentioned optimizations. It is completely agnostic to the gameplay logic, and only relies on the feature of iterating components. As long as iteration is supported, it can stay mostly the same. This is positive, as the rendering is what takes up most of the frame-time. Knowing that it will change little gives us more confidence that tooling will not degrade the runtime performance beyond usefulness.

Focus on tools

In this work, we have solely focused on the runtime of the engine, and paid little heed to the tools, except for these brief comments. Despite this, the value of a general-purpose game engine that the work seeks to explore, lies heavily in the accessibility provided by its tools. Without the tools the runtime is just another SDK. While the runtime does provide certain useful abstractions that the psn00sdk and *Psyqo* do not, such as entity system, the *Psyqo* SDK

is still very sophisticated. And it may be sophisticated enough for it to still be a better option for a team of programmers to use this SDK.
Given this, we believe that moving forward, a general-purpose game engine should heavily focus on the tools, as that is what seems to be unavailable. To some degree, this should even be done to the detriment of performance, as the performance of games that are not made has little importance.

6.3 Using Tundra for making games

As mentioned earlier, PlayStation games are either developed for PAL or NTSC, meaning they run at 50 or 60 frames-per-second (FPS) at most. Often games, such as Crash Bandicoot and Spyro the Dragon, targeted half the framerate, and in turn had double the performance budget. We assume here that we want to achieve the same, meaning 25 or 30 FPS. To cater to the worst case, we specifically assume 30 FPS. This leaves us with $\frac{1}{30}$th of a second, or $\sim$33 milliseconds, to compute a frame, including gameplay simulation and rendering graphics.

The feature benchmark showed that games will be hard-pressed by the graphics capabilities of the engine. But it is not the rasterization of polygons on the GPU that seems to be the issue but rather the *submission* of polygons to the GPU. Submitting a single model of 512 triangles took around 10 milliseconds, so you can submit around 3 of such models within the frame-rate limit. But distributing these triangles over more models would increase the submission time.
To put that into perspective, the player model used for the Nordic Game Jam game was 152 triangles (of which there were 2) and the cannon was 216. In some original games, the characters were of much higher detail. For instance, in Spyro the Dragon the main character was 413 polygons and in Crash Bandicoot it was 732 [42]. Besides the central character, these games also featured rich environments that would eat up even more of this budget.

Graphics are submitted by the CPU, and it is not all that the CPU has to do every frame. Thus, along with the submission of graphics, simulation of the game must also take place, such as computing transforms and game logic. But transforms that need updates and components that need simulation likely correspond to visible game objects. So, while doing such things on a big number of components takes several milliseconds as the benchmarks showed, it is unlikely that you will ever have that many components, because you may not be able to render them. Additionally, features like animation and more advanced collisions and physics, that both Crash Bandicoot and Spyro the Dragon boasts, will of course take up even more of the simulation time.

All that being said, the Nordic Game Jam game shows that it is possible to make something that resembles a game with the engine. And that particular game was running at a constant 50 frames per second, so there is room for higher fidelity and more advanced simulation. Still, it is likely unfeasible to achieve the same fidelity as some advanced original games. But such games also use more advanced features, such as dynamically loading assets, and pre-baking occlusion and lighting [1]. Tundra, in its current state, has none of those things, and it also lacks various optimizations as stated earlier.

In its current state, the general-purpose aspect of the engine is not particularly defining, as it has limited features, and even more limited tools. As more of both will be introduced, ease-of-use should increase, but the flexibility may drop. By extension, it would mean that implementing some very specific features would either not be possible or not as performant without dropping to a lower level of abstraction.
However, being a general-purpose engine it may also have the advantage that more resources could be spent on making the various features more performant than what individual games

might have time for. So, while it may lower the performance potential across games, or at least make it harder to achieve the same potential, it may also ease the access to mediocre or even high performance. Whether this seems good or bad depends on what the developer wants to achieve, but it seems like the proper purpose of a general-purpose game engine.

Overall, with the current state of the engine, it is difficult to say what kind of games are achievable. But it does hint at it being feasible to make a game with such a generalized tool.

Chapter 7

Conclusion

In this work we have set out to explore the technical capabilities of a general-purpose PlayStation engine by developing a prototype of the runtime part of such an engine. Based on the *psn00bsdk*, such a prototype was successfully developed under the name *Tundra*. It sports features such as a game object system, a renderer with support for textured models, smooth lighting, sprites and texts, an input system with support for two simultaneous controllers, an audio system with support for music and sound effects, and an asset compiler that automatically converts a list of raw assets to their runtime counterpart.

Benchmarks that tested features in isolation demonstrated the engine's performance. Notably, they showed that the engine iterates 1000 components within 1.1 - 4.5 milliseconds, computes 240 transforms in 9.4 - 15.4 milliseconds and submits 512 triangles for rendering in 6.7 - 23.6 milliseconds. From these results we found that projects will be mostly pressed by the rendering limitations on the CPU, as the simulation aspects seem to account for a comparatively little proportion of the frame time. The significance of this imbalance was discovered late in development, and so was not reflected in the focus of our work. In future work, we recommend that the rendering pipeline should be given more focus.
In general, we do not expect it to be able to support the same fidelity contained in some advanced original games like Crash Bandicoot. However, it may bring down the bar for better performance, and it does not discard the engine as a likely driver for a game.
At the Nordic Game Jam the prototype was used to develop a simple game by a team of 4 people, 3 of which had no prior experience with the engine. Due to severe bugs in the engine, the game was not properly finished during the jam. However, these issues were afterwards fixed, and the game was wrapped up by the team, with the final product being a collaborative 2-player game in 3D with timing a central game mechanic. It made use of all features of the engine, demonstrating that Tundra is functionally proficient and performant as a tool for making a game of at least that caliber.

While showing capabilities, the runtime prototype is not fully satisfying the vision that we imagine in a general-purpose game engine. As such there are still uncertainties in whether the results will successfully scale to a product that does satisfy this vision. This includes missing features expected of such an engine including animation, collisions and proper visual tooling, but also many optimizations utilized by original games and modern engines such as baking of lighting and scene hierarchies as well dynamically loading and unloading of game data. Whether these features and optimizations will ultimately sum to a better or worse performance is impossible to gauge without further implementation and testing.
Additionally, the results of the experiments involve some uncertainties. For one, because of the significant number of possible setups and uses. But particularly because the experiments

were done using emulators, rather than the original hardware. While such emulators attempt to match the functionality and performance, and do hit within the proper ballpark, there are some expected inaccuracies. But regardless of this, and their use in gauging the technical potential of a more fully-fledged general-purpose engine, we believe there is still value to be had from this work.

For one, we have showed concrete implementations of features based on the constraints of the target console that belong in a general-purpose game engine. We have evaluated and compared these systems, and given insight into the kind of performance bottlenecks that one may be facing and given hints at what such an engine may be capable of.

General-purpose game engines are complex pieces of software with many interconnecting systems and limiting factors. But at the same time, it is a tool for creative minds, for which limitations can be a mere catalyst. So, nothing can be said for certain until it properly exists and is put into the hands of creative minds with the outcome being a proper game experiences. Bringing such an engine into existence involves many unknowns, and wrong, uneducated decisions at a fundamental level can have severe consequences. Thus, a great aid for such a project is to shed light on these unknown. At this, we believe this work brings significant value, and we believe it may act as an inspiration and guide for priorities and choices for a longer-term project.

Bibliography

[1] Andy Gavin. “Making Crash Bandicoot - part 3”. In: (). URL: %7Bhttps://all-things-andy-gavin.com/2011/02/04/making-crash-bandicoot-part-3/%7D.

[2] Electronic Arts. *EA Standard Template Libary*. Source Code. Version 3.21.12. June 9, 2023. URL: https://github.com/electronicarts/EASTL.

[3] Reiji Asakura. *Revolutionaries at Sony the making of the Sony PlayStation and the visionaries who conquered the world of video games*. McGraw-Hill, 2000.

[4] Gaël Guennebaud Benoît Jacob. *Eigen*. URL: https://eigen.tuxfamily.org.

[5] Boštjan Berčič. “Software Emulation in the Light of EU Legislation”. British & Irish Law, Education and Technology Association. In: (Apr. 2005). URL: https://www.bileta.org.uk/wp-content/uploads/Software-Emulation-in-the-Light-of-EU-Legislation.pdf.

[6] Brendan Sinclair. “Sony stops making original PS”. GameSpot. In: (Mar. 23, 2006). URL: https://www.gamespot.com/articles/sony-stops-making-original-ps/1100-6146549/.

[7] *Color Cannon Coop*. Itch.io Page. URL: https://jesperpapiorgmailcom.itch.io/colorcannoncoop (visited on 05/23/2024).

[8] Connor ’Stenzek’ McLaughlin, ACStriker, ElizabethNoir. *DuckStation Compatibility*. URL: https://docs.google.com/spreadsheets/d/e/2PACX-1vRE0jjiK_aldpICoy5kVQlpk2f81Vo6P4p9vfg4d7YoTOoDlH4PQHoXjTD2F7SdN8SSBLoEAItaIqQo/pubhtml (visited on 05/23/2024).

[9] cppreference.com. *A list of open-source C++ libraries*. Apr. 14, 2024. URL: https://en.cppreference.com/w/cpp/links/libs (visited on 03/06/2024).

[10] cppreference.com. *new expression*. Apr. 23, 2024. URL: https://en.cppreference.com/w/cpp/language/new (visited on 02/06/2024).

[11] Jakub Czekański. *ps1-tests*. Source Code. Mar. 5, 2023. URL: https://github.com/JaCzekanski/ps1-tests (visited on 06/01/2024).

[12] Probe Entertainment. *Die Hard Trilogy*. Fox Interactive, Sept. 18, 1996.

[13] Epic Games. *Unreal Engine*. Version 5.4. URL: https://www.unrealengine.com.

[14] Epic Games. *Programming Quick Start*. Unreal Engine documentation. URL: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/CPPProgrammingQuickStart/ (visited on 05/25/2024).

[15] Jason Gregory. *Game Engine Architectur (Third Edition)*. Taylor & Francis, 2019.

[16] Grumpycoders. *PSX.Dev*. Visual Studio Code extension. Version 0.3.4. Open Source (multiple contributors). Feb. 21, 2023. URL: https://marketplace.visualstudio.com/items?itemName=Grumpycoders.psx-dev.

[17] Dennis Gustafsson. *Die Hard Trilogy, Making of - Part 2 of 7*. URL: https://www.youtube.com/watch?v=9NCvdlSbAlI (visited on 06/01/2024).

[18] hacktic. *How to create a PS1 style horror game in Unity*. YouTube. Jan. 25, 2023. URL: https://www.youtube.com/watch?v=yIkDdE-utjA (visited on 05/25/2024).

[19] Henri Gouraud. "Computer Display of Curved Surfaces". In: (June 1971).

[20] Nick Jasuja. *NTSC vs. PAL*. URL: https://www.diffen.com/difference/NTSC_vs_PAL (visited on 05/29/2024).

[21] Juan Linietsky, Ariel Manzur. *Godot*. Version 4.2.2. Open Source (multiple contributors). URL: https://godotengine.org/.

[22] Martin Korth. *No$psx*. Computer Software. Version v2.2. Dec. 16, 2022. URL: http://www.problemkaputt.de/psx.htm (visited on 06/01/2024).

[23] Chris Lomont. *Introduction to Intel® Advanced Vector Extensions*. May 23, 2011. URL: https://hpc.llnl.gov/sites/default/files/intelAVXintro.pdf (visited on 05/27/2024).

[24] Connor 'Stenzek' McLaughlin. *DuckStation*. Version 0.1-6292-g0bc42c38. Open Source (multiple contributors). URL: https://www.duckstation.org/ (visited on 06/01/2024).

[25] Imagine Media. "Inside the PlayStation". In: *NEXT Generation* 6 (June 1995). URL: https://archive.org/details/nextgen-issue-006 (visited on 03/12/2017).

[26] Lucas Meijer. *Custom == operator, should we keep it?* Unity Blog. May 16, 2016. URL: https://blog.unity.com/engine-platform/custom-operator-should-we-keep-it (visited on 05/20/2024).

[27] Microsoft. *Visual Studio Code*. Version 1.89.1. May 7, 2024.

[28] Austin Morlan. *Working with JumboUnity Builds (Single Translation Unit)*. Aug. 5, 2022. URL: https://austinmorlan.com/posts/unity_jumbo_build/ (visited on 05/31/2024).

[29] Nicolas Noble. *PCSX-Redux*. Version 18563.20240329.1.x64. Open Source (multiple contributors). Mar. 29, 2024. URL: https://pcsx-redux.consoledev.net/ (visited on 06/01/2024).

[30] OpenClipart-Vectors. *Rainbow Colors Gradient Ray*. Image. URL: https://pixabay.com/vectors/rainbow-colors-gradient-ray-153296/.

[31] Oracle. *Double Buffering and Page Flipping*. The Java Tutorials. URL: https://docs.oracle.com/javase/tutorial/extra/fullscreen/doublebuf.html (visited on 05/05/2024).

[32] Maurizio De Pascale. *Assets and Entities in the Glacier 2 Engine*. 4C Conference Presentation. Feb. 9, 2018. URL: https://www.youtube.com/watch?v=qw7BhS1lMkI.

[33] *PCSX-Redux*. Discord Forum. URL: https://discord.gg/KG5uCqw.

[34] PCSX-Redux. *PSYQo*. Source Code. Version Commit fb451991e39dbe34e45c36c245967f7ee5ab0df6. Open Source (multiple contributors). Jan. 16, 2024. URL: https://github.com/pcsx-redux/nugget/tree/main/psyqo.

[35] Frederik Mads Pil. "Developing Flexible and Scalable Games using Data-Oriented Design in Comparison to Object-Oriented Design". MA thesis. IT University of Copenhagen, May 2024.

[36] *PlayStation Specifications*. consoledev.net. URL: https://psx-spx.consoledev.net/ (visited on 05/02/2024).

[37] *PSX.Dev*. Discord Forum. URL: https://discord.gg/QByKPpH.

[38] *PSXDEV Network*. URL: https://www.psxdev.net/ (visited on 01/05/2024).

[39] Imagine Publishing. *The PlayStation Book*. Imagine Publishing, 2015.

[40] Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. *Operating Systems - Three Easy Pieces (Version 0.80)*. Arpaci-Dusseau Books, Inc., May 2014. URL: http://www.ostep.org.

[41] retrogames. *retrogames news October 2000*. Oct. 2000. URL: http://www.retrogames.com/102000.html (visited on 06/01/2024).

[42] Rodrigo Copetti. *PlayStation Architecture - A practical analysis*. Apr. 3, 2019. URL: https://www.copetti.org/writings/consoles/playstation/ (visited on 05/02/2024).

[43] SONY. *NET YAROZE*. URL: https://web.archive.org/web/20160311153757/http://www.absolute-playstation.com/api_faqs/faq13.htm (visited on 06/01/2024).

[44] Sony Computer Entertainment. "PlayStation 2 breaks record as the fastest computer entertainment platform to reach cumulative shipment of 100 million units". In: (Nov. 30, 2005). URL: https://web.archive.org/web/20090823155448/http://www.scei.co.jp/corporate/release/pdf/051130e.pdf.

[45] Sony Computer Entertainment Inc. *Corporate Information*. scei.co.jp (WaybackMachine). 2007. URL: https://web.archive.org/web/20071217051703/http://www.scei.co.jp/corporate/data/bizdatajpn_e.html (visited on 05/05/2024).

[46] *std weak_ptr*. cppreference.com. URL: https://en.cppreference.com/w/cpp/memory/weak_ptr (visited on 05/20/2024).

[47] Ars Technica. *How Crash Bandicoot Hacked The Original Playstation | War Stories | Ars Technica*. Video interview with Andy Gavin. Feb. 27, 2020. URL: https://www.youtube.com/watch?v=izxXGuVL21o (visited on 05/31/2024).

[48] *The Original PlayStation: Are you ready?* Reddit, r/psx. URL: https://www.reddit.com/r/psx/ (visited on 05/03/2024).

[49] Unity. *Transform*. Unity Documentation. URL: https://docs.unity3d.com/ScriptReference/Transform.html (visited on 05/26/2024).

[50] Unity Technologies. *Unity*. Version 2022.3.24. URL: https://unity.com/.

[51] John "Lameguy" Wilbert Villamor. *psn00bsdk*. Source Code. Version v0.24. Open Source (multiple contributors). June 3, 2023. URL: https://github.com/Lameguy64/PSn00bSDK.

[52] *What do I need to start developing on PSX?* URL: https://psx.arthus.net/starting.html (visited on 05/04/2024).

[53] WikimediaImages. *PlayStation console*. Image. Apr. 4, 2017. URL: https://pixabay.com/photos/video-game-console-video-game-play-2202613/.

[54] Wituz. *How to make PlayStation 1 games - Part 1: Hello World*. YouTube. Aug. 10, 2016. URL: https://www.youtube.com/watch?v=ITXleeBpic8 (visited on 05/25/2024).

Ludography

[55] Insomniac Games. *Spyro the Dragon*. Sony Computer Entertainment, Sept. 9, 1998.

[56] Japan Studio. *Gran Turismo*. Sony Computer Entertainment, Dec. 23, 1997.

[57] Namco. *Ridge Racer*. Namco, Oct. 30, 1993.

[58] Namco. *Tekken 3*. Namco, Mar. 20, 1997.

[59] Naughty Dog. *Crash Bandicoot*. Sony Computer Entertainment and Universal Interactive Studios, Sept. 9, 1996.

[60] Naughty Dog. *Crash Team Racing*. Sony Computer Entertainment, Oct. 19, 1999.

[61] Sony Interactive Studios America, SingleTrac. *Twisted Metal*. Sony Computer Entertainment, Nov. 10, 1995.

Appendix A: Tundra Source Code

This appendix exists in a separate file: `appendix-a-source-code.zip`

The full source code for the final version of the engine.

Overview

All folders are split into three parts: *base*, *developer* and *playstation*. *developer* contain the things that are only meant to be run on the developer's machine (the model compiler), *playstation* contain the things that rely on PlayStation hardware (e.g. the render system) and *base* are things that can be used on both (e.g. entity system, asset formats, utilities).
The project consists of the following folders:

- **`external`**: The psn00bsdk and external tools used for the build pipeline
- **`include`**: The public headers of the engine. These are the code API of the full engine, and they are the headers included in the user's project.
- **`src`**: The internal source and header files for the engine. This code is not included in the users project.
- **`tests`**: Tests for the project including 156 unit tests, a couple of manual tests, all the benchmarks used in the project and test assets.
- **`tools`**: CMake build scripts for building and running the project.

Building

The project can be build by first generating and building the *developer* part:

```
> tools\build\developer\generate
> tools\build\developer\build debug
> tools\build\developer\build release
```

and then building the engine itself:

```
> tools\build\playstation\generate
> tools\build\playstation\build debug
> tools\build\playstation\build release
```

This will put the build output in the `build` folder, and the `build/deploy` folder contains all files that must be included in the user's project.

Appendix B: Tundra Project template

This appendix exists in a separate file: `appendix-b-project-template.zip`

A file structure that can be used as a template for creating a new Tundra project. This template was used for the Nordic Game Jam experiment.

Appendix C: Revised Nordic Game Jam game

This appendix exists in a separate file: `appendix-b-nordic-game-jam.zip`

The source code for the revised version of Nordic Game jam and the final game built as a CD image (see folder `game` in the appendix folder). The image is tested in *DuckStation* [24] and *PCSX Redux* [29].

Appendix D: Tundra code examples

Some C++ code examples of the various features in the engine.

Fixed-point type

```
td::Fixed32<12> from_int = 1;
td::Fixed32<12> from_float = td::to_fixed(12.34);

td::Fixed16<12> with_16_bits = 1;
td::UFixed16<12> unsigned_16_bits = 12;

// Operations
td::Fixed32<12> a = 1;
td::Fixed32<12> b = 2;
td::Fixed32<12> result;

result = a + b;
result = a - b;
result = a * b;
result = a / b;
result += 1;
result += td::to_fixed(12.34);

// Printing fixed-point
TD_DEBUG_LOG("Printed fixed-point: %s", result);
```

Vectors and matrices

```
td::Vec3<td::Fixed32<12>> v1{1, 2, 3};
td::Vec3<td::Fixed32<12>> v2{6, 5, 4};

v1 += v2;
v1 = v2 * 10;

td::Vec2<td::uint32> uv;
td::Vec2<td::int32> iv;


td::Mat3x3<td::Fixed32<12>> m {
    1, 2, 3,
    4, 5, 6,
    7, 8, 9
};

d::Vec3<td::Fixed32<12>> column = {1, 2, 3};
m.set_column(0, column);
```

Entity system

```
// Custom component type
class MyComponent : td::Component<MyComponent> {
    // Implemention details
};

void entity_system_example() {

    // Create entity and add component
    td::Entity* entity = td::Entity::create();
    MyComponent* my_component = entity->add_component<MyComponent>(
        arg1, arg2);

    // Iterate all components of type MyComponent
    for( MyComponent* my_component : MyComponent::get_all() ) {
        // Apply behavior
    }

    // Example of component references and reference count
    {
        ComponentRef<MyComponent> ref_1 = my_component;
        ComponentRef<MyComponent> ref_2 = my_component;

        my_component->destroy();

        // my_component is destroyed, but not freed because both
        // ref_1 and ref_2 have a reference count on it

        if( ref_1 != nullptr ) {
            // This is false as my_component was destroyed, and
            // ref_1 has now released its reference count because of the
            // boolean operator use (it is overloaded)
        }

        // my_component is still not freed because ref_2 has reference
    }

    // ref_2 is destroyed and no longer holds a reference count on
    // my_component, so my_component is now freed
}
```

Transforms

```
td::Entity* e = Entity::create();

// Dynamic transforms
td::DynamicTransform* parent = e->add_component<td::DynamicTransform>();
parent->set_scale({td::to_fixed(0.25), td::to_fixed(1), td::to_fixed(0.5)
    });
parent->set_rotation({td::to_fixed(0.25), td::to_fixed(0.5), td::to_fixed
    (0.75)});
parent->set_translation({-2, 1, 3});

td::DynamicTransform* child = e->add_component<td::DynamicTransform>();
parent->add_child(child);
```

```
// Static transform
td::StaticTransform* s = e->add_component<td::StaticTransform>(
    td::gte::compute_world_matrix(
        {3, 2, td::to_fixed(0.5)},
        {td::to_fixed(0.25), td::to_fixed(0.5), td::to_fixed(0.75)},
        {-2, 1, 3}
    )
);

// Requesting matrices
TransformMatrix world_matrix = td::gte::compute_world_matrix(transform);

Vec3<Fixed16<12>> v { 1, 2, 3 };
Vec3<Fixed32<12>> expected { -6, td::to_fixed(2.5), td::to_fixed(0) };
Vec3<Fixed32<12>> result = gte::apply_transform_matrix(world_matrix, v);
```

Model asset and component

```
// Tundra initialize method
extern void initialize(td::EngineSystems& engine_systems) {
    const td::uint32 LAYER_WORLD = 1;

    // Load assets
    td::ModelAsset* model_asset = engine_systems.asset_load.load_model(
        asset_data::mdl_raw_asset);

    td::Entity* entity = td::Entity::create();
    td::DynamicTransform* transform = entity->add_component<td::
    DynamicTransform>();

    // This component is rendered by the renderer
    td::Model* model = entity->add_component<td::Model>(
        *assets::models::model_asset,
        LAYER_WORLD,
        transform
    );
}
```

Sprite asset and component

```
// Tundra initialize method
extern void initialize(td::EngineSystems& engine_systems) {
    const td::uint32 LAYER_FOREGROUND = 1;

    // Load assets
    td::TextureAsset* texture = engine_systems.asset_load.load_texture(
        (td::byte*)assets::tex_some_texture);

    td::Entity* e = td::Entity::create();
    td::Sprite* sprite = e->add_component<td::Sprite>(LAYER_FOREGROUND);
    sprite->texture = texture;
    sprite->color = td::Vec3<td::uint8>{100, 230, 140};
    sprite->position = td::Vec2<Fixed32<12>>{ 1, 2 };
    sprite->size = td::Vec2<Fixed32<12>>{ 10, 10 };
}
```

Text

```
// Tundra initialize method
extern void initialize(td::EngineSystems& engine_systems) {
    const td::uint32 LAYER_FOREGROUND = 1;

    td::Entity* e = td::Entity::create();
    td::Text* text = e->add_component<td::Text>(LAYER_FOREGROUND);
    text->text = "Hello, world!";
    position = td::Vec2<Fixed32<12>>{140, 200};
}
```

Time

```
// Tundra update method
extern void update(td::EngineSystems& engine_systems, const td::FrameTime&
     frame_time) {

    td::Duration start = engine_systems.time.get_duration_since_start();

    // Do stuff...

    td::Duration end = engine_systems.time.get_duration_since_start();
    td::Duration length = end - start;

    TD_DEBUG_LOG("Time passed: %d ms", length.to_milliseconds());
}
```

Sound

```
namespace assets {
    extern "C" td::byte snd_jump[];
    extern "C" td::byte snd_music[];
}

// Tundra initialize method
extern void initialize(td::EngineSystems& engine_systems) {
    const td::SoundAsset* jump_sound = engine_systems.asset_load.
    load_sound((
        assets::snd_jump);

    const td::SoundAsset* music = engine_systems.asset_load.load_sound((
        assets::snd_jump);

    td::uint32 play_id = engine_systems.sound_player.play_sound(jump_sound
    );
    engine_systems.sound_player.stop_sound(play_id);

    engine_systems.sound_player.play_music(music);
    engine_systems.sound_player.stop_music(music);
}
```

Input

```
// Tundra update method
extern void update(td::EngineSystems& engine_systems, const td::FrameTime&
     frame_time) {
    if( engine_systems.input.controller_1.is_pressed_this_frame(td::Button
    ::Square) ) {
        // ...
    }
    if( engine_systems.input.controller_2.is_pressed_this_frame(td::Button
    ::L1)) {
        // ...
    }
}
```

Appendix E: Tundra Source Code Example

Full source-code example of how Tundra is used. The example defines it own `Box` component that uses a loaded white cube model. It adds a single camera that views from an angle, as well as a directional light that lights the box from the upper right. The rendered output can be seen in Figure E.1.
This source code is nearly identical to the source code found in the *template project* (see Appendix B)

```
#include <tundra/startup.hpp>
#include <tundra/core/fixed.hpp>

#include <tundra/assets/model/model-asset.hpp>
#include <tundra/engine/entity-system/entity.hpp>
#include <tundra/engine/entity-system/component-ref.hpp>
#include <tundra/engine/dynamic-transform.hpp>
#include <tundra/rendering/model.hpp>
#include <tundra/rendering/camera.hpp>
#include <tundra/rendering/render-system.hpp>

// Custom box component
class Box : public td::Component<Box> {
public:

    Box(td::DynamicTransform* transform) : transform(transform) { }

    static Box* create() {
        td::Entity* entity = td::Entity::create();
        td::DynamicTransform* transform = entity->add_component<td::
    DynamicTransform>();
        entity->add_component<td::Model>(*assets::models::white_box,
    LAYER_WORLD, transform);
        return entity->add_component<Box>(transform);
    }

    const td::ComponentRef<td::DynamicTransform> transform;

};


// Settings
const td::EngineSettings ENGINE_SETTINGS {
    30000 // Size of primitive buffer
};

const td::uint32 MAIN_LAYER_DEPTH_RESOLUTION = 2048;
```

```
const td::uint32 LAYER_WORLD = 1;

// Assets
namespace asset_data {
    extern "C" const td::uint8 mdl_white_box[];
}

namespace assets::models {
    const td::ModelAsset* white_box;
}

// Globals
td::Camera* camera;

// Called once, at startup, before update loop
extern void initialize(td::EngineSystems& engine_systems) {

    // Setup lighting
    engine_systems.render.set_light_direction(0, { td::to_fixed(-0.577),
    td::to_fixed(-0.577), td::to_fixed(0.577) });
    engine_systems.render.set_light_color(0, { 255, 255, 255 });

    // Setup layer settings
    td::List<td::CameraLayerSettings> layer_settings;
    layer_settings.add({LAYER_WORLD, MAIN_LAYER_DEPTH_RESOLUTION});

    // Create camera
    td::Entity* camera_entity = td::Entity::create();
    td::DynamicTransform* camera_transform = camera_entity->add_component<
    td::DynamicTransform>();
    camera_transform->set_translation({td::to_fixed(0), td::to_fixed(0.5),
     td::to_fixed(-0.5)});
    td::Camera* camera = camera_entity->add_component<td::Camera>(
    camera_transform, layer_settings);
    camera->look_at({0,0,0});

    // Load assets
    assets::models::white_box = engine_systems.asset_load.load_model(
    asset_data::mdl_white_box);

    // Create box
    Box::create()->transform->set_scale({ 2, 2, 2 });
}

// Update loop that is called every frame
extern void update(td::EngineSystems& engine_systems, const td::FrameTime&
     frame_time) {
    for( Box* box : Box::get_all() ) {
        box->transform->add_rotation({0, td::to_fixed(0.005), 0});
    }
}
```

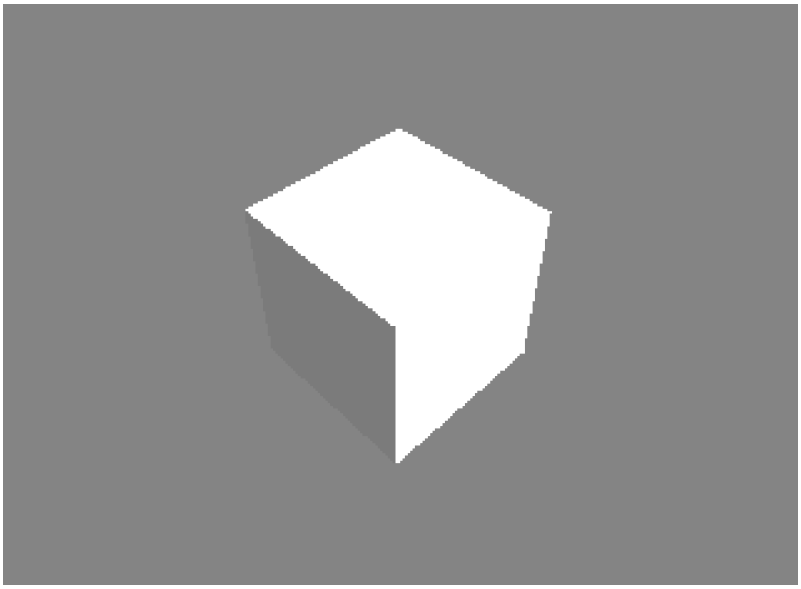


Figure E.1: Rendered output of the Tundra source code example. A single rotating white box lit by a single directional light and viewed from upper right corner.