Universidad San Jorge

Escuela de Arquitectura y Tecnología

Grado en Ingeniería Informática

Proyecto Final

Tesseract: Realtime Rendering 3D engine

Autor del proyecto: Francisco Núñez Villagómez

Director del proyecto: Eduardo Jiménez Chapresto

Zaragoza, 7 de septiembre de 2022

Dedicatoria y Agradecimiento

Me gustaría agradecer este trabajo a Eduardo Jiménez por su orientación y dedicación no solo durante este proyecto sino desde el inicio de la carrera.

Asimismo, quiero dar las gracias a mis compañeros, en especial a Gabriel Villalonga, Daniel Gracia y Jorge Solano por darme estos años juntos y compartir mi pasión.

También quiero realizar una mención especial a los profesores en general, y en especial, a Jorge Echeverría, por confiar en mí y darlo todo por conseguirme oportunidades para desarrollar mi potencial.

Por último, quiero agradecer a mis familiares más cercanos por apoyarme en mis decisiones y hacer todo lo posible por que pudiese perseguir mis objetivos.

Tabla de contenido

Resumen

El Renderizado en tiempo real es uno de los pilares fundamentales en cuanto al mundo de los videojuegos se refiere, ya que es imprescindible en la creación de imágenes elaboradas. Es un campo muy amplio y en constante expansión, sometido a optimizaciones y desarrollo tanto en el mundo del software como del hardware especializado (GPUs).

Este campo no solo se refiere a la creación de gráficos sugerentes para videojuegos mediante la programación, también busca resolver problemas de modelado del mundo real, como la proyección de sombras, iluminación, creación de materiales realistas (PBR: Renderizado basado en físicas), simulación de fluidos e incluso aplicaciones fuera del ámbito de los videojuegos tales como simulaciones útiles para el campo de la medicina.

Para conseguir dichos objetivos se implementan en constante evolución diferentes algoritmos y técnicas de renderizado, del más bajo nivel al más alto, desde la manera en la que se trata la geometría, hasta las técnicas de muestreado para sombrear un píxel.

En este trabajo se incluyen ejemplos de diferentes técnicas que usadas en conjunción muestran un renderizado 3D en tiempo real donde el lenguaje de programación interactúa con la API de gráficos directamente, desarrollando programas específicos que se ejecutan en la GPU y gestionando los recursos necesarios para mostrar el potencial del pipeline de gráficos y cómo estos pueden ser mejorados mediante la ingeniería de renderizado.

Abstract

Realtime Rendering, due to its importance in image creation and processing, is one of the main pillars within videogame development. It is a wide field which is constantly subject to optimization, expansion, and development in software as well as in specialized hardware (GPUs).

This field is not only focused on fancy graphic generation for videogames through programming, but also solving real world modelling problems such as shadow projection, illumination, Realistic material creation (PBR: Physically based rendering), fluid simulations among others, it is also used to solve complex simulations within the medical field.

In order to achieve said objectives several algorithms which are subject to constant improvement and evolution are implemented, from low to high level. Starting from how geometric data is treated to how a pixel shading is sampled.

In this Project are included several techniques that used in conjunction show a functioning real-time 3D renderer where the programming language interacts with the graphics API, developing specific programs that are executed in the GPU while managing its resources to show the potential of the graphics pipeline, and how graphics can be enhanced through rendering engineering.

1. Introduction

The idea behind this project is to develop a Realtime Rendering 3D Engine from scratch, facing different engineering problems that must be overcome in order to achieve a good understanding of how the computer graphics pipeline works and how it can be enhanced to fit a particular implementation.

This kind of work is commonly performed by a Rendering Engineer or a Game Engine Engineer, both positions are really attractive to me, and I aspire to fill one of them someday. They share a lot in common regarding the knowledge needed to design and implement complex architecture and systems.

The project in question is Tesseract, a Realtime Rendering 3D Engine, in which you can see 3D models being represented with several effects achieved by implementing and adjusting different rendering techniques such as shadow mapping, normal or bump mapping and different shading models.

During the development of Tesseract I have designed systems and architecture, performed QA passes for them, adjusted, debugged and fixed bugs that appeared during the development. I also implemented tools within the engine to do different tasks such as profiling to extract time information, or UI coding to represent values in real time and even a logging system to ensure the low-level systems work correctly.

1.1. The role of the Rendering Engineer

Considering several job descriptions offering this position from companies such as Apple, Rare and Microsoft among others, the Rendering team work closely with the Art and Design teams to help them achieve the artistic vision that is intended. As a Rendering Engineer your work is easily seen, as you work in graphics and graphic related tasks, ranging from working within the renderer to implement new features enhancing existing ones to optimizing code to achieve the desired performance.

A Rendering Engineer work is closely related to the Renderer or Engine that is used to generate the graphics of the project in a closer way than, for instance, a Gameplay Engineer would be related to it, as the latter would not have to modify the engine but work with the given framework whereas the Rendering Engineer will have to modify it sometimes to fit the artistic vision or needs of the project. A strong knowledge of the different graphics APIs is needed to fulfill this role programmatically speaking, as implementations will sometimes vary and this will affect how data is treated and transmitted. A good example is handedness and endianness, the first one being API specific and the second one being Platform specific.

A Renderer is the implementation of an architectural design that transforms data from the virtual world generated in the computer into an image, normally through the GPU by using different techniques depending on the artistic vision or implementation of other systems. There are also software Renderers which perform the graphics calculations in the CPU, in the past, when GPUs weren't so powerful or common, this was the norm. The features of a Rendering system are prone to be optimized and enhanced driven by the needs of the project.

Being able to transmit into the Renderer the features that are needed to fulfill the Art and Design teams' intention is the final objective of a Rendering Engineer. Rendering Engineers also serve as a bridge between the formality of code and the spontaneity of art, as sometimes the technology can't follow the specification of the design. Being able to convey that and find a common spot between Art, Design and implementation is one of the most difficult but important tasks that this role needs to achieve.

1.2. Other important roles in Game Engine Development

It is safe to say that Game Engines are one of the most complex pieces of software a team of engineers can aspire to create. While it is possible to design and implement a Game Engine as a solo developer, it will not reach the level of game engines implemented by a group of engineers focused on it, as usually as a solo developer the final objective is not to produce a game engine but to create a game.

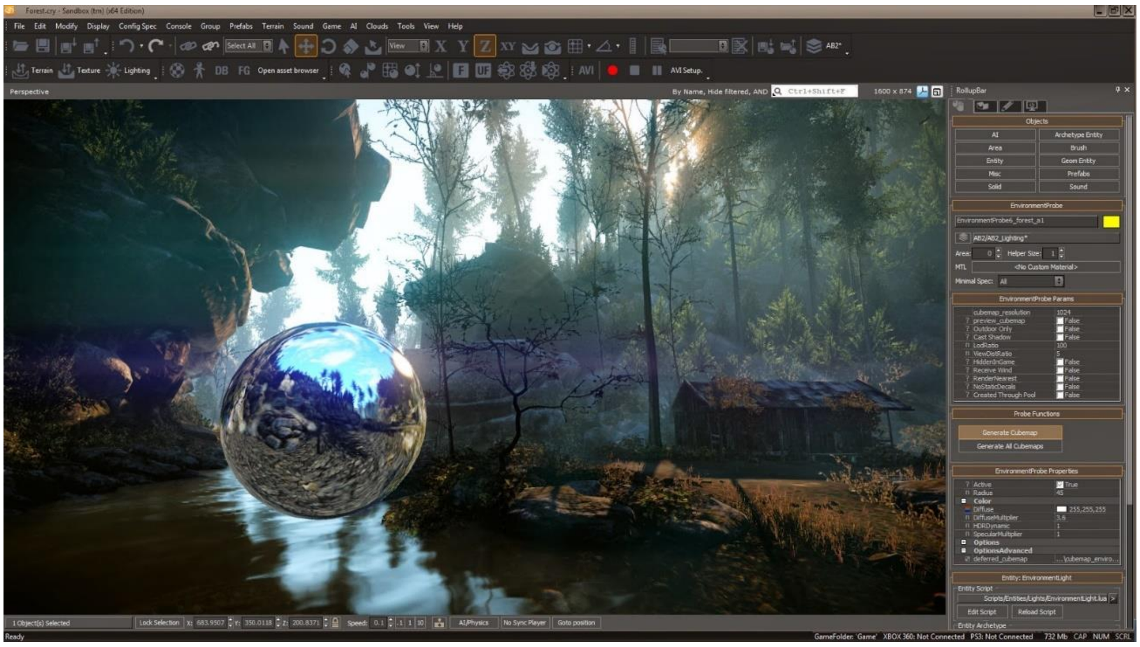


Figure 1: Cryengine Sandbox Editor featuring a complex scene.

Apart from Rendering Engineers, there are many other roles in need of fulfillment in order to create a professional engine. Core Engineers, Physics Engineers, Tools, AI, Network, UI, Animation, even Audio Engineers. Every system that needs to be implemented in the engine will need a professional that is able to work within the system.

Core Engineers implement features at system levels and supports gameplay programmers while in production, they are also fundamental in the optimization stage of any project. Game Engines need to be optimized so gameplay can extract as much power as possible from the different platforms. Physics Engineers job, as the name implies, is to implement and develop features within the physics simulation of the engine.

Tools Engineering helps developing tools built on top of the engine systems that might help speed up the workflow of the team or to implement whole new tools that are needed for the development of games, good examples of this are visual scripting tools, animation sequencers…

UI Engineering focuses on the visuals and user experience part of the engine, artists and other members of the development team need to be able to use the UI properly so it needs to be responsive.

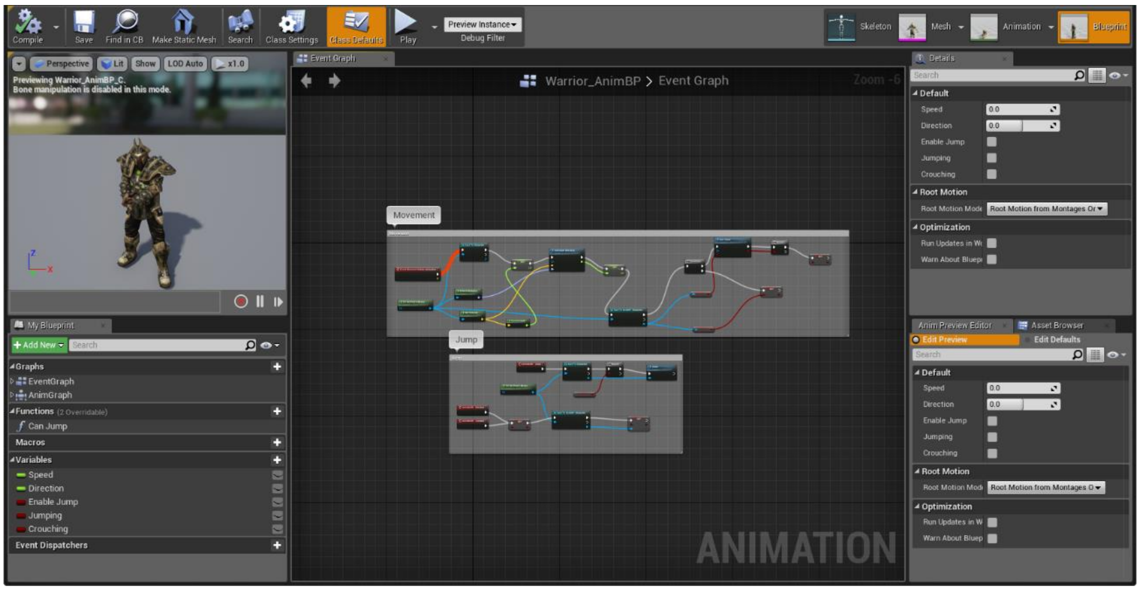


Figure 2: Animation bluepring editor in Unreal Engine 4.

1.3. The Rendering pipeline

The Rendering pipeline is the sequence of steps that the graphics API takes in order to transform data from the virtual world into an image. Some steps from this pipeline are programmable, others can follow a set of directives set by the programmer before that particular stage happens.

The DirectX 11 Graphics or Rendering pipeline programmable stages use the High Level shader language programming language (HLSL), making this pipeline flexible and adaptable to different rendering situations.

The first stage is the Input-Assembler stage, where the vertex shader input is set following certain semantics available in the Graphics API such as POSITION, NORMAL, TEXCOORD...

The programmable steps are the Vertex Shader Stage, the Hull Shader Stage, the Domain Shader Stage, the Geometry Shader Stage, and the Pixel Shader Stage.

The Vertex Shader Stage processes vertices that come from the Input Assembler, it performs per-vertex operations such as skinning, morphing, transformations… There are some techniques that use the vertex shader to perform per-vertex lighting (Gouraud shading), but nowadays it is more common to perform them in the pixel shader to prevent certain artifacts from happening. The pipeline must have a vertex shader to work, even if it is a default one. The Hull and Domain Shader Stages correspond to the Tesselation Stage implemented since DirectX 11, the Tesselation Stage is commonly used to subdivide or join vertices depending on varied factors such as how far from the camera are those vertices, the level of tessellation…

The Geometry Shader Stage, unlike the Vertex, operates with its input set to all the vertices for a full primitive, one vertex for a point, two vertices being a line, three for triangles… It can output triangle strips, line strips and point lists. Some algorithms that can be implemented in the geometry shader are those to generate Fur, or Dynamic particle systems among others.

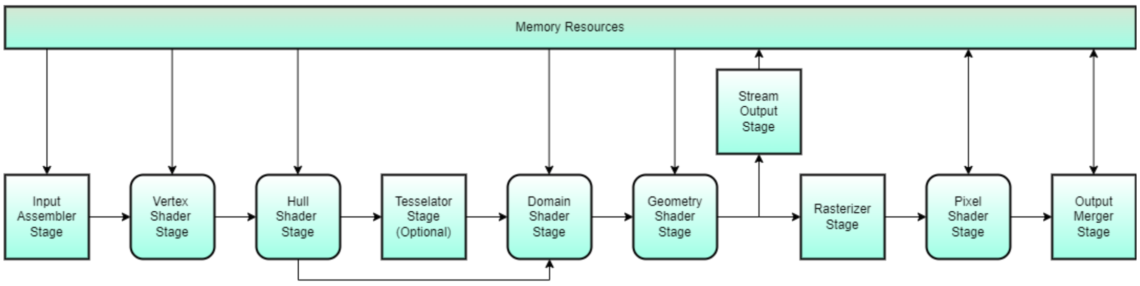


Figure 3: The Rendering Pipeline.

Finally, the Pixel Shader Stage is the most flexible of them all, as it enables plenty of shading, lighting techniques and post-processing. Pixel Shaders can have different types of input, to finally output colored pixels, some of the most important inputs are:

- Textures, which in computer graphics are, usually, information regarding a surface such as albedo, metallic, heightmaps, roughness, normal textures...
- Sampler states, which are entities which allow the GPU to perform different types of filtering to textures, which is the way the pixel color is extracted from the texture into the shader to perform its calculations. Different types of filtering will have different

effects, for example, bilinear filtering is commonly used to smooth out textures when objects are bigger or smaller than they are in memory.

- Constant Buffers, which are structs filled with constant data that are stored in GPU memory and can be accessed from the vertex and pixel shader. A common use for Constant Buffers is to store the different Matrices used to perform transformations such as translation, rotation and scale to the rendered entity, or the camera view or projection.
- Structured Buffers, which are similar to the previous, but it contains an array of homogeneous structures that can be indexable instead of single data. A good candidate for a structured buffer could be a list of lights.
- Output from the Vertex Shader, which are the interpolated per-vertex values such as position, color, texture coordinates, normal...

It combines textures, the interpolated per-vertex values coming from the vertex shader and other data such as constant data generally set as constant buffers containing lighting information or transformation matrices for the entity that is being rendered, set as input for the shader in order to produce per-pixel output.

One caveat of Pixel Shaders is that it will be executed per-pixel, this means that a high computational cost per-pixel can make the difference between being able to use an effect in real time or not. To avoid doing all lighting calculations for each single pixel we can use Deferred Rendering systems, however, due to its implications, Deferred Rendering can't support Translucent objects. For this we can use Forward or Forward-Plus Rendering, which is a mix between Deferred and Forward rendering.

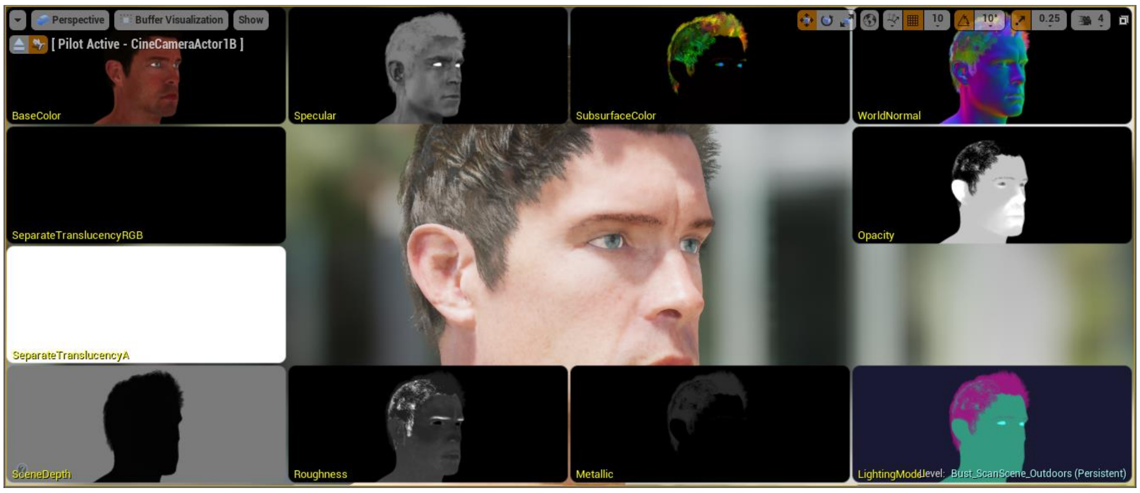


Figure 4: GBuffer combination to produce a high-quality Render image in Unreal Engine 4 Deferred Rendering System.

2. State of the Art

Game engines weren't always so capable as they are nowadays, there were games that didn't even have a game engine associated with. In the early years of game development games were developed by one person or a small team of people and information on the subject was sparse and not so easy to get.

This resulted in really specific and small in-house engines being developed for games that had a slight bigger production, and that was not the general rule. As we will see in following chapters, game engines were popularized during the rise of 3D Computer Graphics during the 1990s.

As game engines rose to being the rule and not an option, different development roles arose as well such as the Tools, Core, Engine and Rendering engineer.

2.1. A brief History of Game Engines and Rendering

Game engines and Realtime Rendering is the primary focus of this project, so we are going to go through some of the most important titles and why they were important to the evolution of this type of engineering work.

Researching the predecessors of the actual engines will make a huge difference on how we understand the medium and why some of the actual engineering decisions are the way they are and how they evolved from the early times so we can make good design and development decisions for Tesseract.

Doing this research, I decided to divide the information into several branches, but following a timeline.

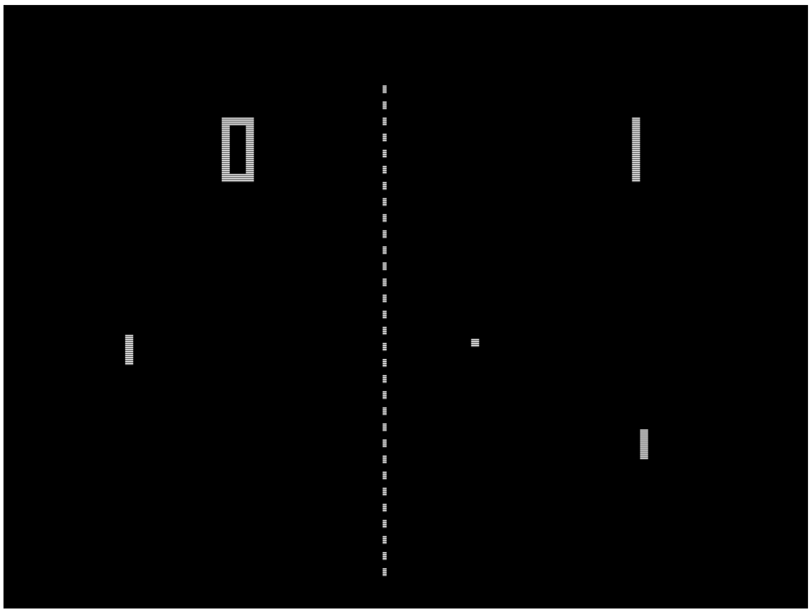


Figure 5: PONG! First game!

2.1.1. Before Game Engines

In the early years of game development, games need to be written from the ground up as single entities, this means that no engine or common framework was shared between the different games that appeared on the market at that time.

A particular example of this is that games for the Atari 2600 needed to be designed from scratch to make optimal use of the display hardware. These days, this is referred to as kernel by developers who still work on retro videogames (retro-developers).

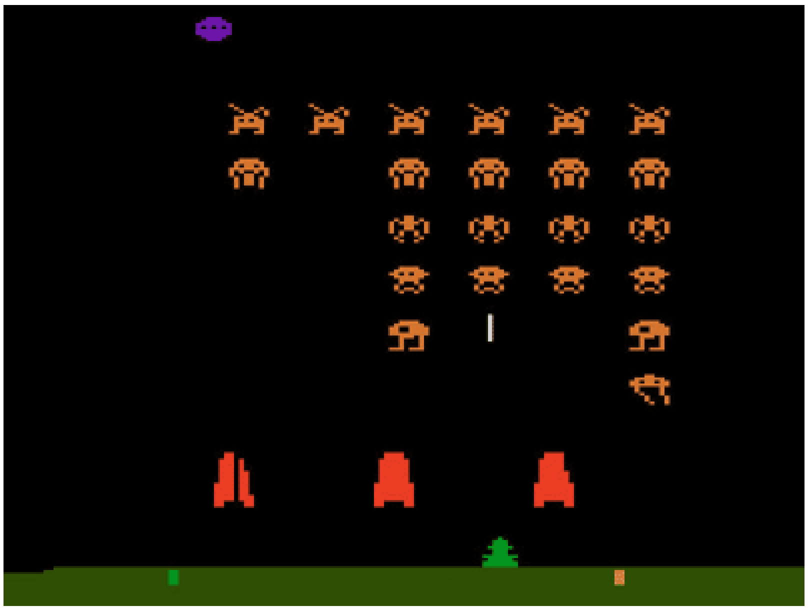


Figure 6: Space Invaders caption, this retro game helped rise Atari 2600 sales.

Engines need a design heavily centered around data and memory, so even if the display was not an issue, these said limitations made it really difficult to develop and sabotaged different attempts to create engines to for this device.

Even for platforms that appeared after this very little was reusable between different games, as the hardware was evolving really fast and there was no point in developing an engine that would be deprecated or unusable for the next console or computer that was out in the market. Also, games at that time had a hard-coded ruleset and very little graphics/levels data.

Later on, Companies started developing small proprietary/in-house game engines to use for developing first-party software.

2.1.1.1. Construction Kits

Figure 7: Wargame Construction Set for C64 front page.

During the 1980s, several 2D creation systems were produced in the independent game development scene. They were called construction kits and some examples of these are the Pinball construction set from 1983, Thunder Force Construction and Adventure Construction Set from 1984, Garry Kitchen's GameMaker from 1985, Wargame Construction Set from 1986, Shoot'Em-Up Construction Kit from 1987 and finally Arcade Game Construction Kit from 1988.

As we can see from the Kit's names, some of the game genres were defined very early on in the videogame history.

The Commodore 64 was one of the most popular home computers and had plenty of games and construction kits associated with. It had BASIC built-in so learning to program using a construction kit was easy using this platform.

This construction sets were from drag and drop onto a table to small level editors (Thunder Force was the first to implement a level editor) and small controls over the game logic for the most earlier versions to even game programming on the latter.

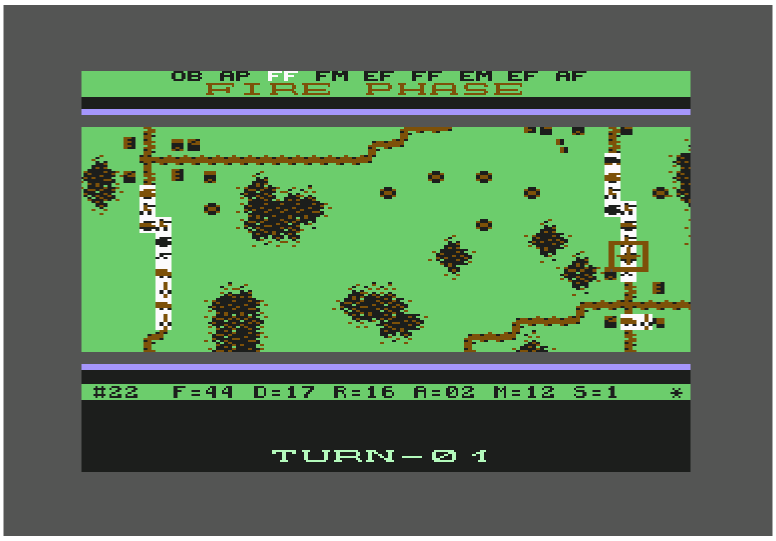

OB AP FF FM EF FF EM EF AF
FIRE PHASE
#22 F=44 D=17 R=16 A=02 M=12 S=1 *
TURN-01

Figure 8: Wargame Construction Set game example.

2.1.2. *In house/Proprietary Game Engines (1990s)*

Many of the first engines that were created were property of the companies they were developed in and mostly didn't come out of those companies because at that time software licensing for engines was not a popular thing.

There were some exceptions such as the Doom and Quake engine from Id Software, that had parts of its core software licensed to other companies to make their own games.

In the following chapters we will see some of the most relevant engines from the first years.

2.1.2.1. *Space Rogue/ Ultima Underworld Engine*

The games developed within this engine were the first ones classified as being 3D due to the varying height and inclined surfaces that were implemented in order to create a 3D effect. It also used an algorithm to implement texture mapping to walls, floors, and ceilings.

As everything was rendered in 3D and used physics, at this time this kind of technology needed really high-end systems, which were not so common at that time, and even there the execution was slow due to the high computational cost.

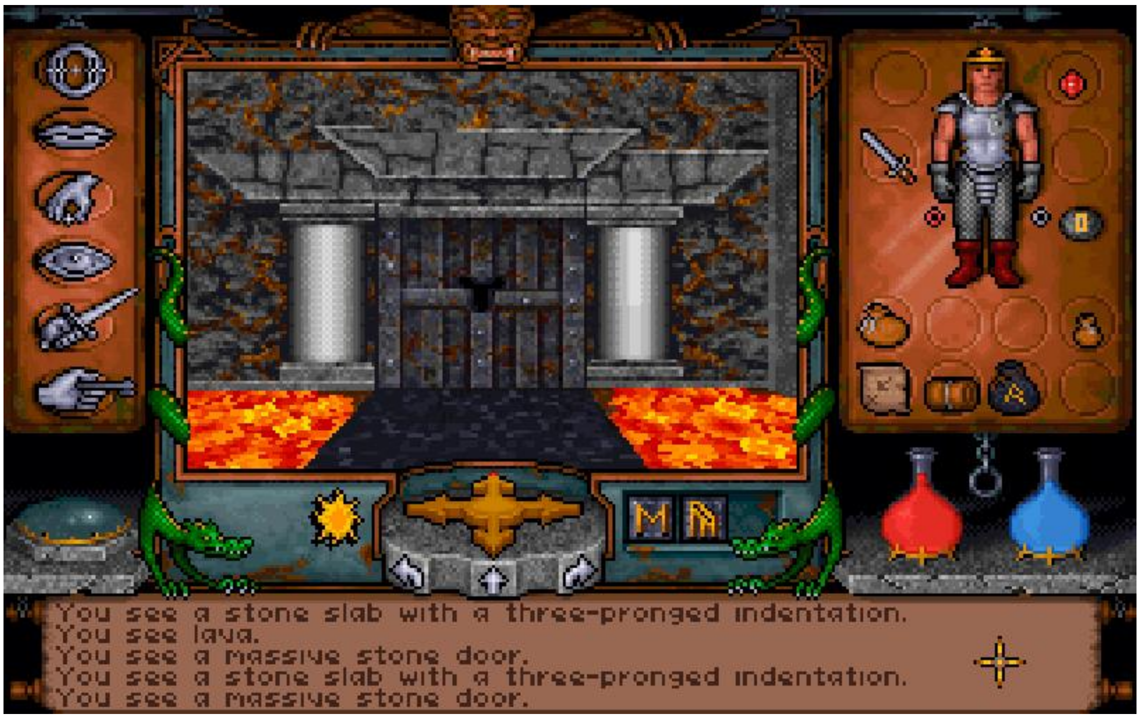

You see a stone slab with a three-pronged indentation.
You see lava.
You see a massive stone door.
You see a stone slab with a three-pronged indentation.
You see a massive stone door.

Figure 9: Ultima Underworld: The Stygian Abyss.

2.1.2.2. *Doom / Id Tech 1 Engine*

The Doom engine release happened in 1993, it was used to develop Chex Quest 1+2, Doom, Doom II, HacX, Heretic, HeXen and Strife.

The Doom Engine was not truly a 3D engine. However, it did trick the player into thinking it was 3D. This effect was achieved by using 2D sprites to represent characters, objects and anything untied with the environments specifically, where Id Software used height differences to difference it from everything else. As it was a 2D engine, that meant that rooms couldn't be added on top of each other, but this allowed for less powerful hardware to render the environment in a quick way.

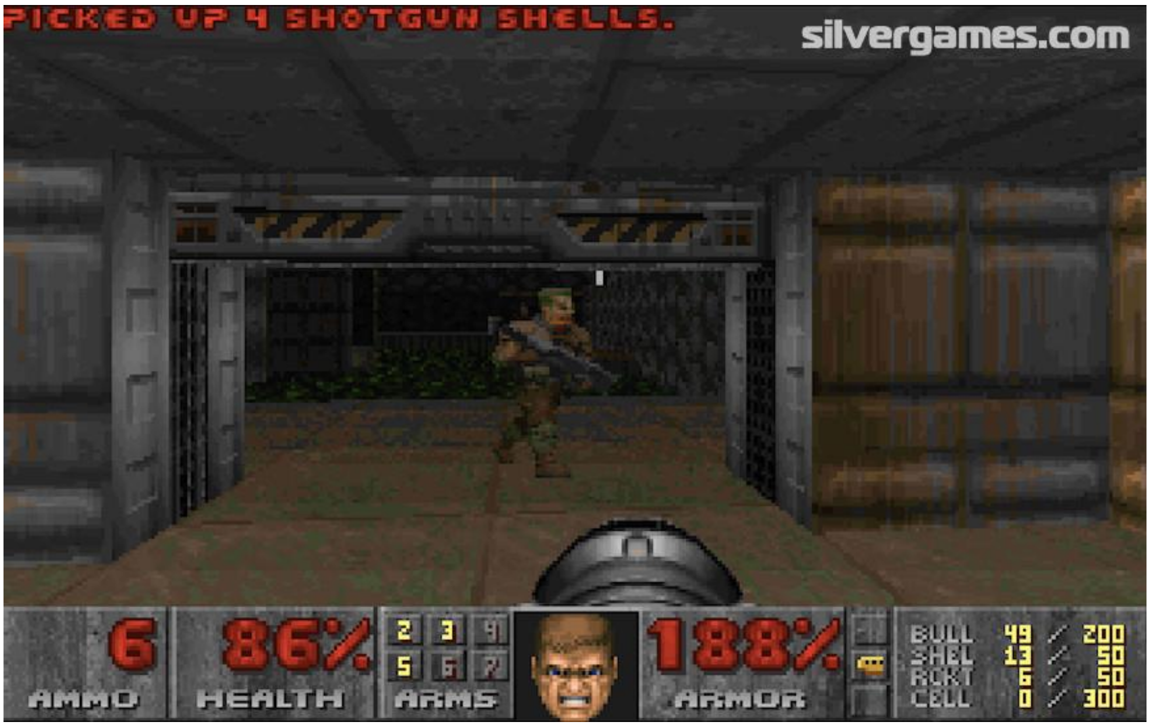


Figure 10: Doom I from Doom Engine.

2.1.2.3. Transitional Engines

I would like to address transitional engines as well, meaning that these engines tried different technologies that helped produce better software in the future. Some of these engines are:

NovaLogic's Voxel Engine, which was used to create some of the props within Blade Runner, Comanche and Command and Conquer among others. It used a combination of the world's volumetric and pixel to render 3D bitmaps rather than vectors, so this created a smoother and more detailed terrain, allowing for an also smoother gameplay.

The Build Engine, which was similar to the Doom Engine, but used a grid system to divide the world so it could create a perfect 3D illusion still using 2D planes with sprites in it. By using a tag system, it allowed for quick transitions from room to room, creating the illusion of falling or moved through a hole when in reality they were teleported. Duke Nukem 3D was developed in the Build Engine.

The first 3D engine was developed by Bethesda in 1995 and its name was XnGine, it was DOS based and had many bugs and stability issues in Windows 95, as it tackled for the first-time clipping, 3D transformations and collisions. Later on, it was able to use high resolution graphics, allowing for huge game worlds, shown in Daggerfall.

2.1.3. The first 3D Game Engines

As we saw on the previous chapters, at the start the engines did not try to feature a real 3D simulation but faking it through diverse algorithms. However, as technology evolved, so did game engines, trying to create frameworks for full 3D, or similar simulations.

2.1.3.1. LucasArts Jedi Engine

The Jedi Engine was one of the most revolutionary engines at the time, it was developed by LucasArts and using it they developed Star Wars: Dark Forces among other titles. The innovative part of this engine was that all the objects were made as 3D models but then they were rendered onto bitmaps up to 32 different angles for each object.

They also implemented a rudimentary LOD (Level Of Detail) system where the objects were rescaled depending on how close or far away from it was the player, allowing for a nice 3D look. It also allowed for jumping, crouching, and looking up and down, making the Jedi Engine one of the most advanced at its time.

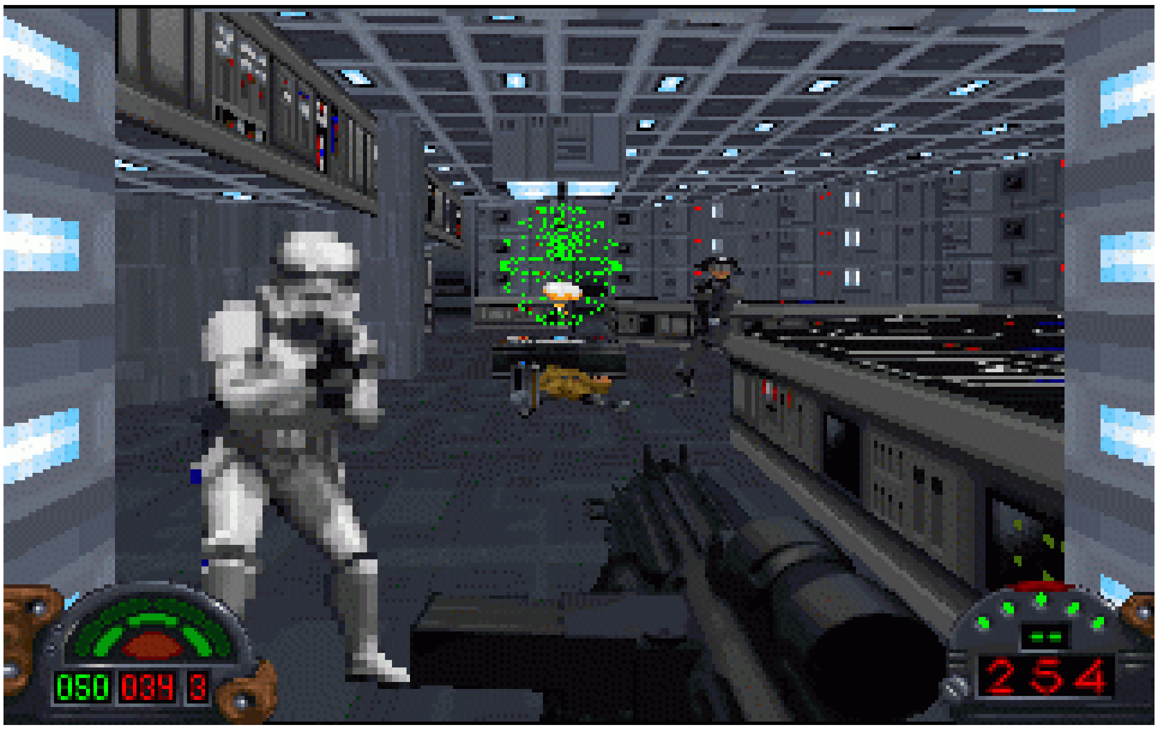


Figure 11: Star Wars: Dark Forces.

2.1.3.2. Quake Engine

Id Software jumped in into the 3D world with this engine, which was their first full 3D engine. Quake was made using this engine.

To prevent being too intensive on processing power, this engine implemented Z-Buffering, which is a technique that only renders the areas the player is in by searching and checking for object boundaries or how these boundaries were called, brushes. This technique sometimes allowed for a save of more than half the polygons from being rendered wastefully.

On the Lighting side, they added 3D light sources on a second pass of the preprocessor, this enabled Quake engine to have a good 3D look and execute smoothly.

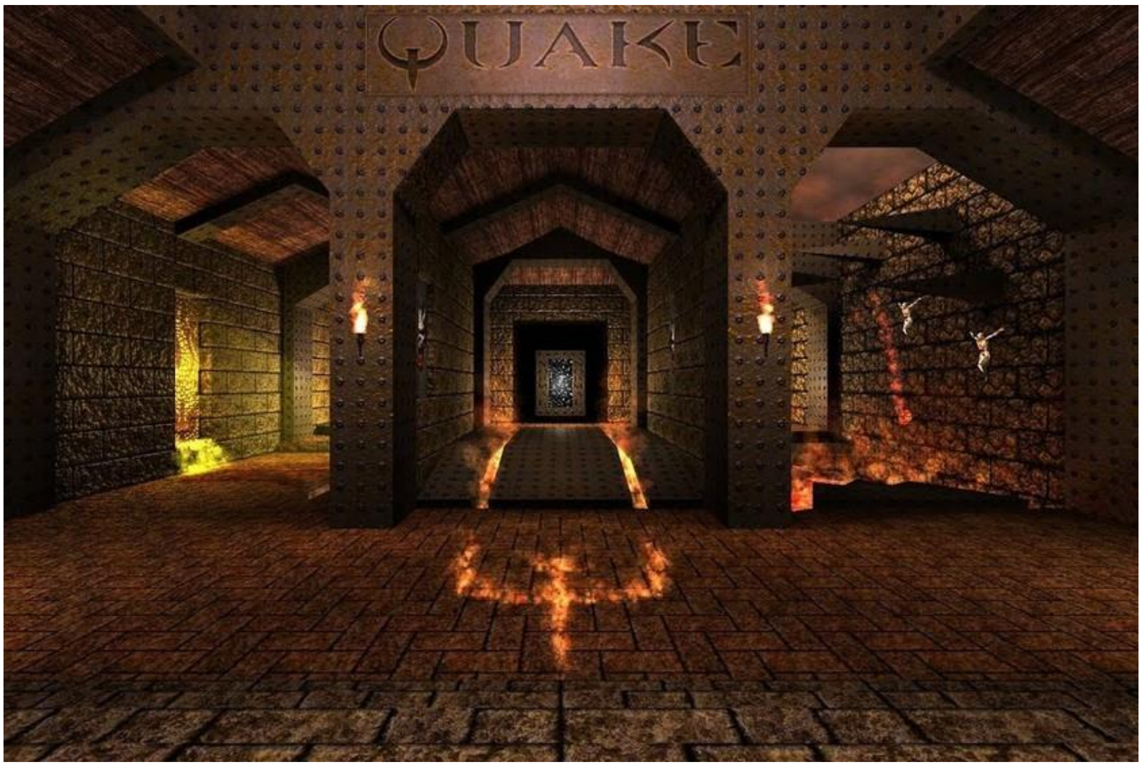


Figure 12: Quake I.

2.1.3.3. Renderware Engine

The Renderware Engine is worth noting due to the high number of titles developed on it: over 200 titles. This engine started losing its popularity due to the rise of hardware accelerated graphics, as it was developed prior to GPUs, and it did not implement a hardware rendering system to present its graphics.

Eventually Epic's Unreal Engine took over Renderware, which was very popular until that moment because of the ability to manipulate in real time art and game processes.

2.1.3.4. Quake Engine evolution

Quake II was an upgrade of the first Quake Engine, and it introduced OpenGL support, lighting effects featuring coloring and DLL loading.

The DLL Loading is performed by writing the game in C and compiling as a DLL which the Engine loads in real time and executes for changes in game code. This system allows for a continuous execution as long as no breaking changes are made into the game code.

This engine was very popular due to its modding capabilities and because Id Software released the source code to the public to allow developers to create RPGs.

Figure 13: Quake II

After Quake II Engine, a tweaked version of the Quake Engine came out, GoldSource or GoldSRC, which featured both support for OpenGL and Direct3D and established PC's dominance over consoles at that time. Even if it shares Quake Engine's core, almost 70% of the code was rewritten in this version of the engine. Games like Half-Life, Team Fortress Classic and Counter Strike were developed in this engine, which helped promote 3D video cards with the API support and games that were developed.

Figure 14: Half-Life I

2.1.4. The Evolution and Advances of 3D Engines

As 3D Graphics were popularized and started being the rule rather than an unknown research field different game engines rose and fell. There was a constant evolution and intention of enhancing what was already done. As we will see in the following chapters, these engines started looking more like what we know today as 3D engines.

2.1.4.1. Unreal

The Unreal Engine was intended as a First-Person Shooter game engine, but it also became the base for many RPG games with Mass Effect among them. It was the principal competitor of Quake II engine from Id Software.

As Quake II was modded constantly by its community and had a custom scripting language, as a response to that, Epic provided a map editor and modification program that was called UnrealEd. At this time, Unreal was capable of doing both software and hardware rendering, collision detection, colored lighting, and a very simple texture filtering implementation. Games like Deus Ex, Star Trek, and X-COM: Enforcer were made within Unreal Engine.

Figure 15: X-COM: Enforcer

2.1.4.2. Quake III

Quake III Engine was more than a refinement to Quake II, as it jumped from skeletal animations into per-vertex animation, this meant this engine was capable of much smoother animations.

Other features were 32-bit color capabilities, shader, shadows, curved surfaces, and network capabilities that surpassed the other engines at the time.

Quake III engine was used on Call of Duty and Medal of Honor: Allied Assault among other titles.

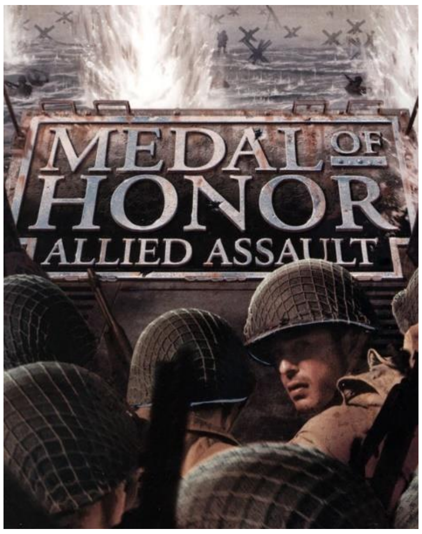


Figure 16: Medal Of Honor: Allied Assault

2.1.4.3. Innovative Engines: Torque, Max-FX, GeoMod

These engines were not a huge hit as Unreal or Quake III but presented innovative techniques such as destructible environments in GeoMod by using Boolean physics to get realistic in-game physics in Red Faction.

Torque was created for the FPS Tribes 2 and had the capability to manipulate LODs on the fly so there were huge polygon rendering savings, along with having a built-in world editor and robustness.

Max-FX was a hardware-only 3D rendering engine, and it introduced the famous The Matrix effect of dazzling bullet-time into the game Max Payne.

2.1.4.4. Unreal 2

This version of the Engine was a heavily modified version which featured integrated physics, 64-bit support, improved special effects, realistic fluid simulations enabling for realistic moving water and was marketed saying it was able to handle 10 times more polygons than the previous Unreal.

Games like Deus Ex: Invisible War and Splinter Cell were made using this engine, among many others.

2.1.4.5. Gamebryo

This engine was used to develop big titles like Fallout 3, Warhammer Online and The Elder Scrolls IV: Oblivion.

It had cross-platform capabilities and it was the only third-party engine that had Nvidia PhysX directly connected with the Wii framework making the engine one of the most flexible at the time.

It was written in C++ and supported many platforms and technologies such as DirectX 9, DirectX 10, 3D Max and Maya integrations, Dynamic collision detection, particle systems, 3D Audio and Multiple-core development among other features.

Around 200 games have been developed in this engine since 2003.

© 2005 Bethesda Softworks LLC, a ZeniMax Media company. All rights reserved.

Figure 17: The Elder Scrolls IV: Oblivion

2.1.4.6. Doom 3

Doom 3 Engine happened as a decision to switch from C to C++ as the core language of the engine, it allowed for realistic shadows due to the surfaces being calculated on real-time. However, it required performant hardware to run due to the unified light and shadow system, where almost every surface would pass through the same rendering pipeline.

Figure 18: DOOM III

2.1.5. 2000s Modern 3D AAA Game Engines

Different companies developed different game engines which explored plenty of fields in real time graphics. As console development was something important to take into consideration, a lot of game developer companies focused on optimizing the code within their renderers and engines to create performant and beautiful games within those consoles, which were and currently are a huge piece of market.

In the following chapters we will see the most notable engines with their characteristics.

2.1.5.1. Source

Valve's Source engine is now known as one of the most common engines, it was originally released in 2004, very relevant games in videogame history like Counter Strike: Source, Garry's Mod, Half Life 2, Left 4 Dead and Portal were developed within this engine.

This engine featured advanced Shader technologies, physics, dynamic lighting and shadows, different effects from which the most relevant were reflective water surfaces, real-time motion blur, lip-sync and facial animations.

Figure 19: Half Life 2

2.1.5.2. RAGE

RAGE was Rockstar's Advanced Game Engine; it was created as a collaboration from said company with RAGE Technology. It quickly became Rockstar's game engine. Grand Theft Auto IV was developed within RAGE.

This engine featured different systems: rendering, physics, audio, network, animation, scripting. All into one package. They also had a Bullet physics engine. It was designed to handle realistic feeling while driving vehicles or walking.

Figure 20: Grand Theft Auto IV

2.1.5.3. Frostbite

DICE created Frostbite from the ground up thinking of multi-core PCs, Xbox 360, and PlayStation 3 Development. Battlefield: Bad Company 1&2 and Battlefield 1943 were developed within this engine.

It featured large destructible environments, building, foliage, and objects destruction, which are one of the key features of the Battlefield games.

2.1.5.4. Anvil/Scimitar

This engine was built from scratch by a group of Tool engineers for the Assassin's Creed games. It featured fluid animation, and for this the engineers implemented a multithreaded system which also allowed for dynamic world loading.

2.1.5.5. Unreal 3

Unreal Engine was intended to be used as a multiplatform development engine, it featured PC, PS3, PS4, Xbox 360 and Xbox One. It featured multi-threaded rendering, 64-bit color, HDR rendering pipeline, Nvidia PhysX and several effects derived from it; particle effects, in-game cinematics, skeletal animation system which supported up to 4 bone influences per vertex and full mesh LOD support among many other programming features.

This engine was released in 2007 and was used to develop big hits like Batman: Arkham Asylum, Bioshock 1&2, Mass Effect 1&2…

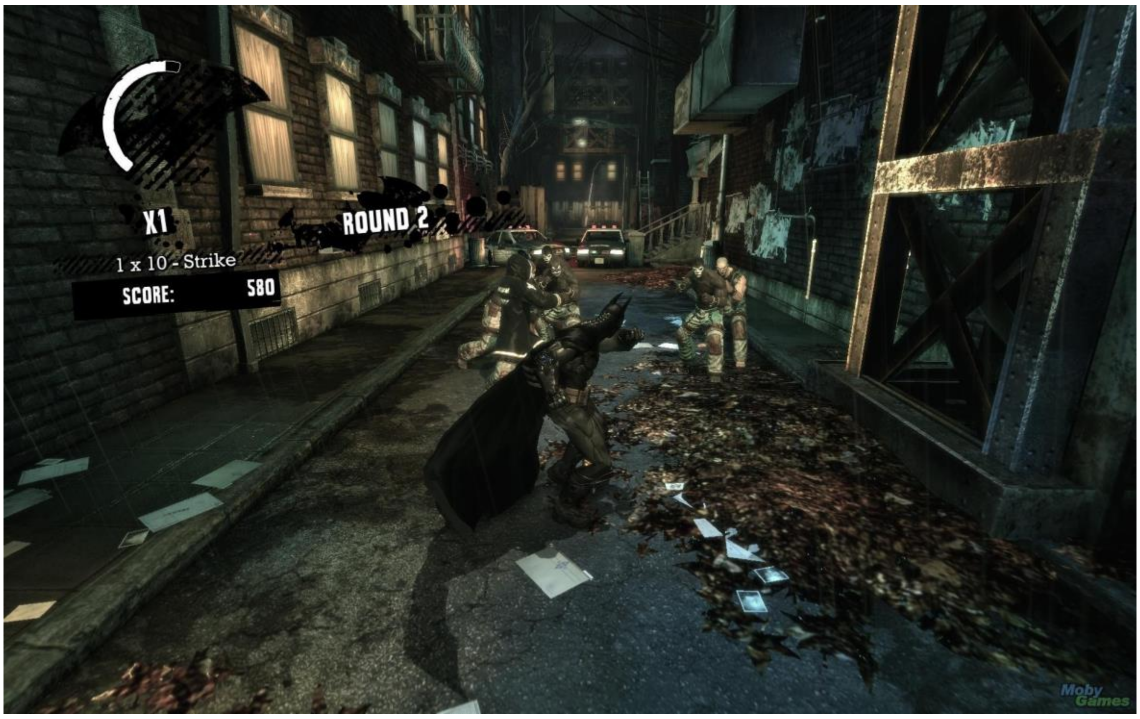


Figure 21: Batman: Arkham Asylum

2.1.5.6. CryEngine, Dunia

CryEngine was the predecessor of Dunia which was developed to feature the capabilities of Nvidia GeForce 3, it was released in 2004 and led to the development of Far Cry, which was very successful.

It featured using Pixel Shaders for realistic water rendering in Far Cry, which immersed the player in the island, surrounded by a lot of vegetation.

Crysis was also developed using this engine, and even nowadays, this very resource consuming DirectX 10 game is still used as a gaming benchmark.

Dunia was used to develop Far Cry 2, and shared some design from CryEngine, but almost all the codebase was rewritten. It was designed to be more forgiving with low-end PCs, allowing for Far Cry 2 to be executed in hardware that was less powerful.

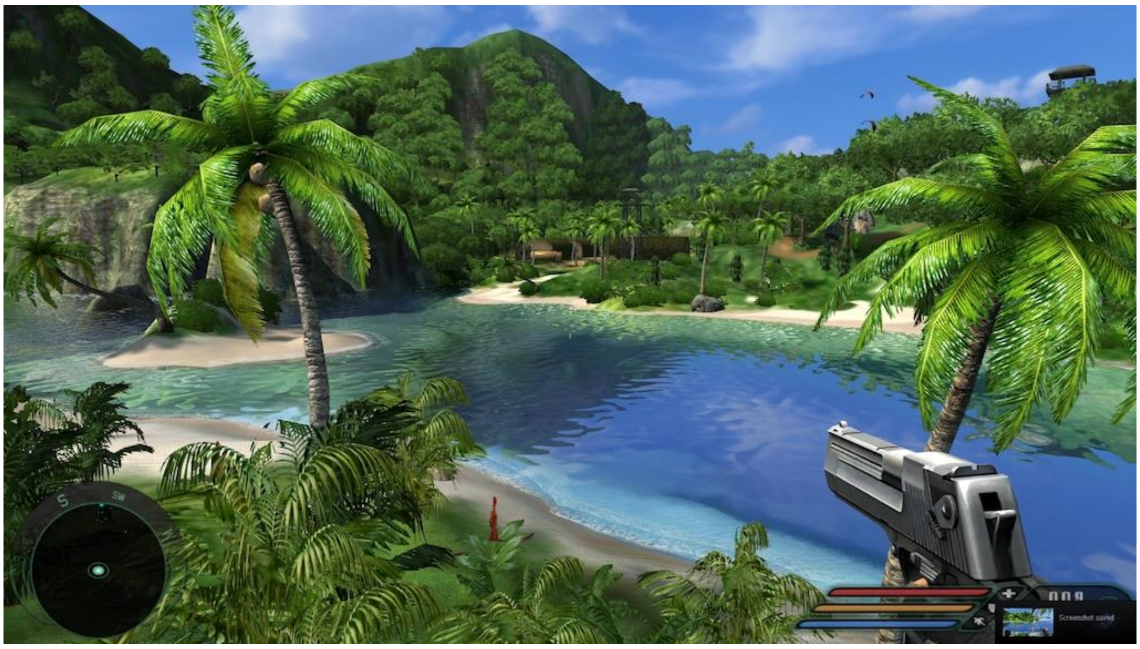


Figure 22: Far Cry 1 visuals.

2.2. Recent History

Game Engines have been moving towards a more generalist development approach, where games featuring completely different genres can be developed from the same engine and built for many different platforms.

This development approach has its own caveats, as not having a genre specific engine can flaw performance sometimes among other things as we will see, having a lot of functionality might not be as good as it looks if we are not using it.

We will review some of the most popular engines at the moment.

2.2.1. Unreal Engine 4 & 5

Unreal Engine has been slowly establishing itself as the king of high-end game engines due to its high amount of functionality and its good graphic quality.

Unreal Engine 4 supports hardware raytracing among other things and Unreal Engine 5 evolved creating its own Global Illumination plus raytracing system called Lumen. It also features Nanite, which is a new geometry system which uses an internal mesh format to render pixel scale detail in high polygonal meshes.

Both versions of the engine allow for a visual scripting system called Blueprint system as well as interoperability with C++ programming, which is the base language where the engine has been built on top.

Figure 23: Final Fantasy VII Remake running on Unreal Engine 4

Unreal Engine has proven to be a good structure for many AAA games. It has been evolving for more than a decade and keeps adding new functionality and polishing what was existing previously. It is fully open source, which means anybody can clone it and modify the engine to implement new features or modify existing ones.

However, some of the caveats of this engine are the lack of in-depth documentation for many systems and its steep learning curve.

2.2.2. Unity

Unity exploded as one of the most popular game engines for independent developers some years ago and featured plenty of big hits such as Hollow Knight, Escape from Tarkov and Cuphead.

It features the capability of developing into both in 2D and 3D pipelines, it started having support for several proprietary scripting languages but moved towards C# along with their own hierarchy system where all unity scripts derive from, called Monobehaviour. This base class provides a framework and hooks into useful events used in game development.

Even though it has been used to create big titles, it is not strong at AAA development due to several factors:

- The source code of the engine is restricted to Unity, so you couldn't fix a bug within the engine and would have to wait Unity to fix it for you.
- The graphics and light pipeline are not as advanced as Unreal's so trying to get similar visuals would take longer and not have the same quality.

Even if not intended for AAA games, Unity still stands as a really good option for indie and mobile development.

It has been used by students as well as experienced developers so there are a lot of resources on the Internet. It is quick and easy to use, the development process allows for fast iterations and allow for deploying in a variety of platforms.

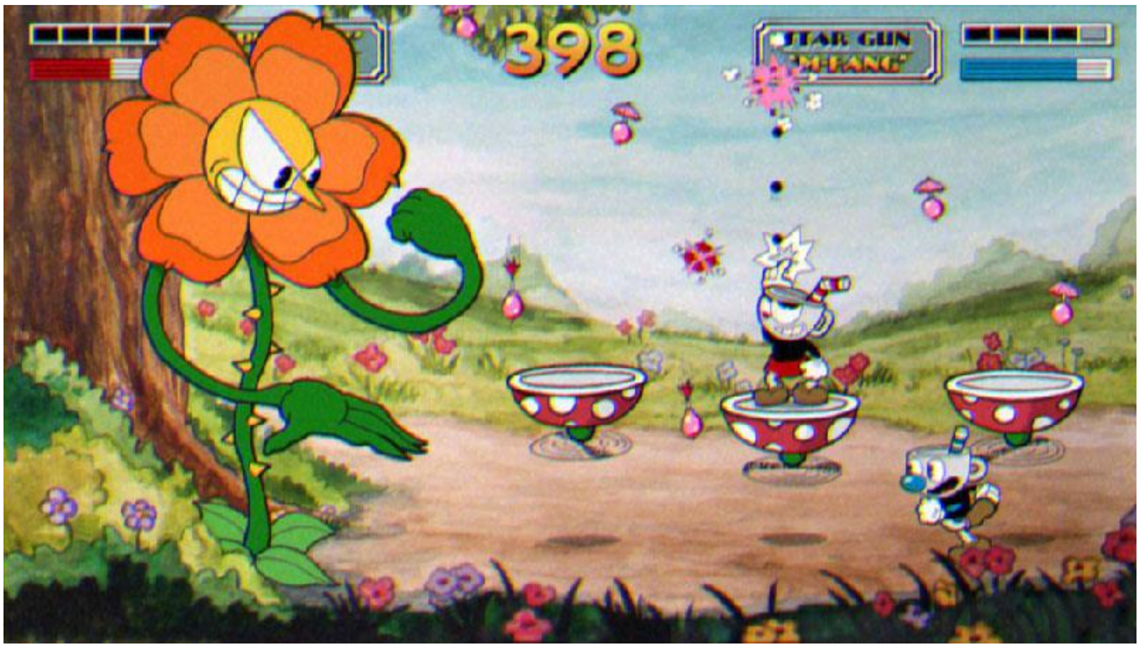


Figure 24: Cuphead

2.2.3. Godot

Godot is a very flexible engine which can handle 2D and 3D games and presents an interesting system where every entity of the game is a node.

This engine can be compared to Game Maker, and chosen over it to develop 2D games, it has its own scripting language inspired by python, GDScript, but also supports C# and C++ development. However, the 3D engine is not ready for complex game development due to the rendering system not being as powerful as the ones from its competitors yet and not having as many features.

Godot is also free and completely open source but does not support built-in console development due to licensing.
iOS and PS4 ports of Deponia were developed in Godot.

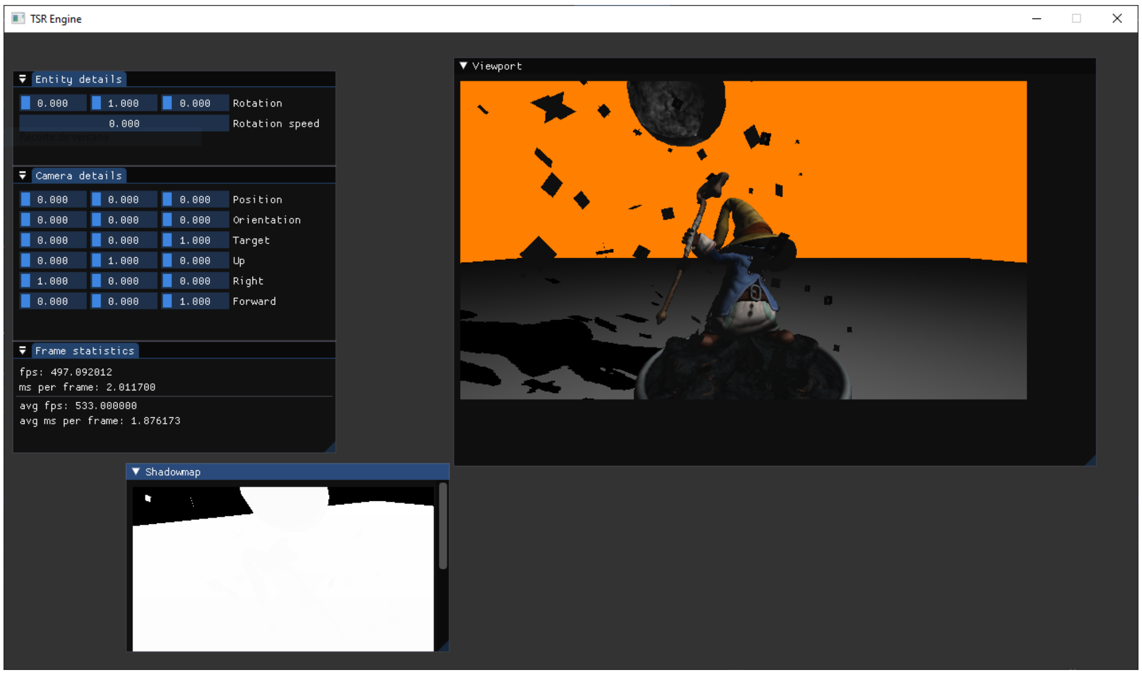
Figure 25: Deponia

2.3. Tesseract

Tesseract is a small engine centered in the 3D Realtime Rendering aspect of it. During the development, as we will see in the Development section of this document I have designed and implemented several rendering features used by actual game engines and some other features that are needed into this kind of software.

Some of these said features are integrating ImGUI as an Immediate mode UI, Third-Party software integrations, textured mesh support, a simple shader system and a simple profiling system and Logging system among some others.

As for the rendering aspect of it some of the most relevant features implemented are Texture mapping, Bump and Normal mapping, shading models support, shadow mapping and a free look camera among others.



```
Entity details
0.000   1.000   0.000   Rotation
        0.000           Rotation speed

Camera details
0.000   0.000   0.000   Position
0.000   0.000   0.000   Orientation
0.000   0.000   1.000   Target
0.000   1.000   0.000   Up
1.000   0.000   0.000   Right
0.000   0.000   1.000   Forward

Frame statistics
fps: 497.092012
ms per frame: 2.011700
avg fps: 533.000000
avg ms per frame: 1.876173
```

Figure 26: Tesseract engine prototype featuring windows with different information contained.

3. Objectives

Tesseract is an exploration of how game renderers and engine architecture work in the practice. I will be focusing on the rendering and effects that can be achieved by graphics programming; this means that I will implement several visual features in the Rendering Engine. My main objective is to design and implement all these features as good as possible.

These objective features can be broken down into several points:

- Integrate the graphics API, in my case DirectX 11, into a set of functions that extract the functionality, so future integration of different graphics APIs can be performed gracefully.
- Integrate a set of utils into the engine such as containers or math libraries, in my case DirectX Math.
- Adjust the graphics pipeline to fit the needs of Tesseract's features. Projecting 3D meshes with their textures, normal mapping, lighting, and shadow mapping among others.
- Continuous integration of profiling and debugging UI tools into the Engine.

During the development time of this project, I implemented different features into the profiling side such as ImGUI integration to create Immediate mode menus and UI to interact and be able to see data changes or choose between different options, and a platform agnostic time delta and frame system. I also integrated Assimp and DirectX Tools to create an asset loader to load into memory 3D meshes along with their textures. EASTL (Electronic Arts Standard Library) was my choice to implement containers into Tesseract.

In the Renderer I implemented Primitive rendering, Complex Mesh Rendering, for the lighting Lambert + Phong model and GGX. Texture mapping and Bump/Normal mapping to generate geometric complexity in texture space and Shadow mapping among others and a free look camera to move around and inspect from different angles and distances.

Different people who looked at it, ranging from not professionals to rendering engineers have manifested it "looked really good".

All this checks all four points that were stated as objectives.

4. Methodology

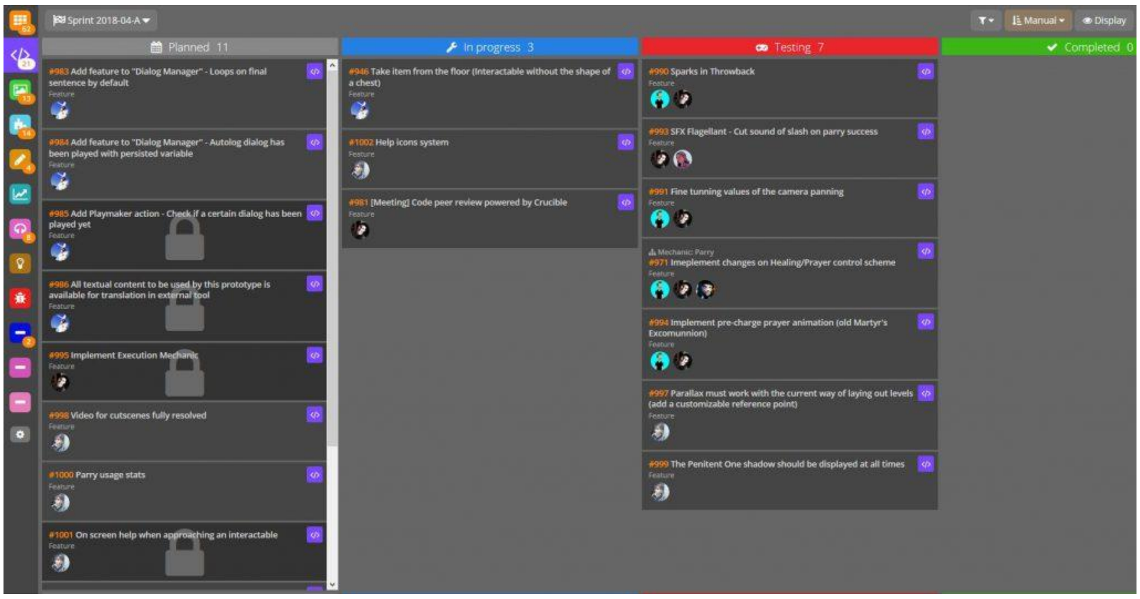
The development of Tesseract was done completely on my own. I learnt what I needed by reading books and querying Engineers and Rendering Engineers with my doubts and questions. From time to time, I revisited the design I had for the project and updated it, as design follows engineering capabilities and queried again people with more experience than me to be able to enhance my systems.

4.1. Tools

As workstations, I have been working on a laptop and on a Desktop PC, each with different hardware.

Several software tools have been used for the completion of this project. Discord has been the application we have used in order to communicate, when necessary, to establish reunions or exchange messages.

To keep track of tasks I have used HackNPlan, which is a tool that allow the user to organize and keep track of work. As a user you can create notes in a board, each note represents a small task or ticket that a person needs to do. In my case, as I am developing this alone, I only created tickets for myself. This tool is helpful to know how much time you spend doing a certain task, how priority is a certain task and if some of the tasks need more work. My revision policy was doing one small sweep each morning and a thorough one each Wednesday to know how the week start was going. This is done similarly in some agile methodologies which involve sprints, such as the scrum methodology.



Planned 11

#983 Add feature to "Dialog Manager" - Loops on final sentence by default
Feature

#984 Add feature to "Dialog Manager" - Autolog dialog has been played with persisted variable
Feature

#985 Add Playmaker action - Check if a certain dialog has been played yet
Feature

#986 All textual content to be used by this prototype is available for translation in external tool
Feature

#995 Implement Execution Mechanic
Feature

#998 Video for cutscenes fully resolved
Feature

#1000 Parry usage stats
Feature

#1001 On screen help when approaching an interactable
Feature

In progress 3

#946 Take item from the floor (Interactable without the shape of a chest)
Feature

#1002 Help icons system
Feature

#981 [Meeting] Code peer review powered by Crucible
Feature

Testing 7

#990 Sparks in Throwback
Feature

#993 SFX Flagellant - Cut sound of slash on parry success
Feature

#991 Fine tunning values of the camera panning
Feature

Mechanic: Parry
#971 Imeplement changes on Healing/Prayer control scheme
Feature

#994 Implement pre-charge prayer animation (old Martyr's Excomunnion)
Feature

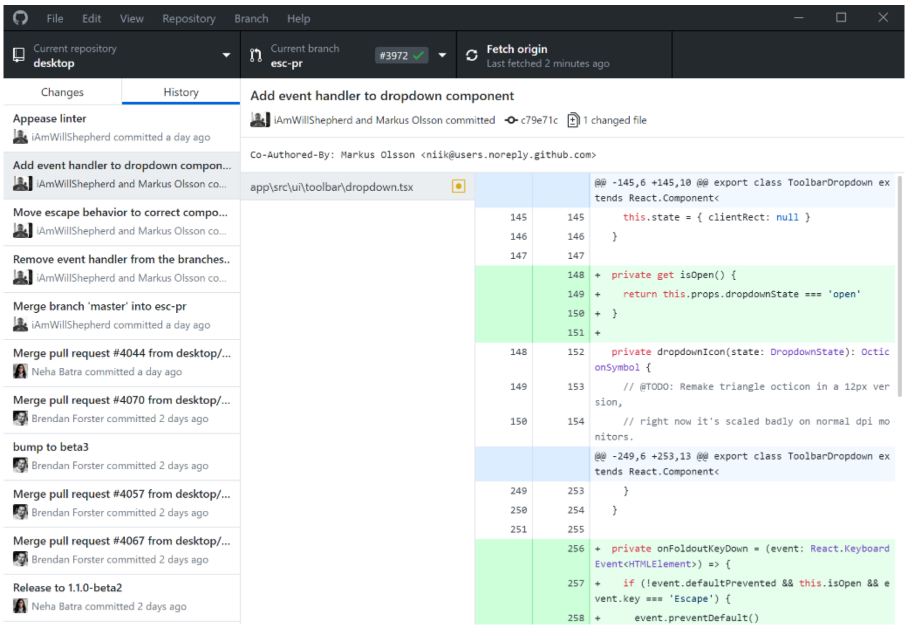
#997 Parallax must work with the current way of laying out levels (add a customizable reference point)
Feature

#999 The Penitent One shadow should be displayed at all times
Feature

Completed 0

Figure 27: HackNPlan board with several tasks.

As the version control system, I mostly used git on Windows Terminal. However, if I needed to keep visual track of commits or needed an interface for some reason, I used tools like SourceTree or Github Desktop.



Appease linter
iAmWillShepherd committed a day ago

Add event handler to dropdown compon...
iAmWillShepherd and Markus Olsson co...

Move escape behavior to correct compo...
iAmWillShepherd and Markus Olsson co...

Remove event handler from the branches..
iAmWillShepherd and Markus Olsson co...

Merge branch 'master' into esc-pr
iAmWillShepherd committed a day ago

Merge pull request #4044 from desktop/...
Neha Batra committed a day ago

Merge pull request #4070 from desktop/...
Brendan Forster committed 2 days ago

bump to beta3
Brendan Forster committed 2 days ago

Merge pull request #4057 from desktop/...
Brendan Forster committed 2 days ago

Merge pull request #4067 from desktop/...
Brendan Forster committed 2 days ago

Release to 1.1.0-beta2
Neha Batra committed 2 days ago

Add event handler to dropdown component
iAmWillShepherd and Markus Olsson committed c79e71c 1 changed file

Co-Authored-By: Markus Olsson <niik@users.noreply.github.com>

app\src\ui\toolbar\dropdown.tsx

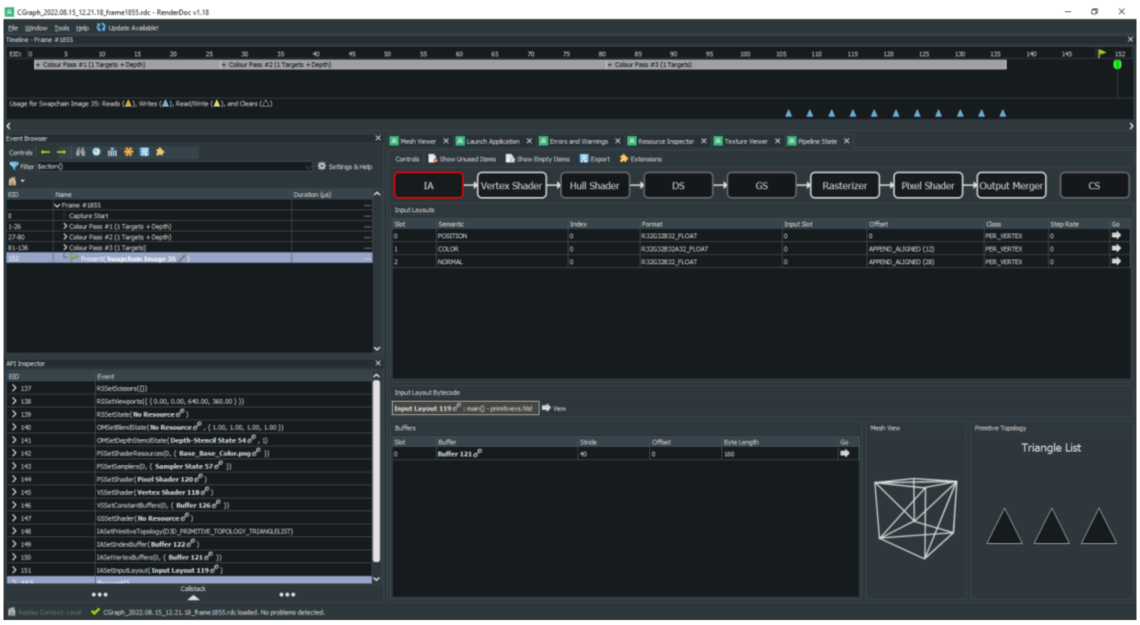
```
@@ -145,6 +145,10 @@ export class ToolbarDropdown extends React.Component<
    this.state = { clientRect: null }
  }

+  private get isOpen() {
+    return this.props.dropdownState === 'open'
+  }
+
  private dropdownIcon(state: DropdownState): OcticonSymbol {
    // @TODO: Remake triangle octicon in a 12px version,
    // right now it's scaled badly on normal dpi monitors.
@@ -249,6 +253,13 @@ export class ToolbarDropdown extends React.Component<
    }
  }

+  private onFoldoutKeyDown = (event: React.KeyboardEvent<HTMLElement>) => {
+    if (!event.defaultPrevented && this.isOpen && event.key === 'Escape') {
+      event.preventDefault()
```

Figure 28: Github Desktop Commit inspection view.

The language I used was C/C++, as it is one of the programming languages, I am most confident with. To debug the CPU processes and develop I used Visual Studio 2019, Visual Studio 2022, and Visual Studio Code, which are IDEs with a lot of integrations and functionality such as memory watches, call stack inspection, dependency management…

One of the most relevant tools I used to develop the Rendering Engine was RenderDoc, which is a graphics debugger. It works simply by attaching to the program and capturing one or a few frames so you can see the program state within the different stages of the rendering pipeline and inspect closely, edit shaders, and relaunch the frame, inspect buffers, framebuffers, textures, configurations…

This makes correct configuration and graphics programming faster and easier, allowing the developer to spot bugs faster, this, to me has been a lifesaver on several occasions where I did not know what the error was because I couldn't inspect directly from Visual Studio shader variable values, RenderDoc solves that issue flawlessly.



Usage for Swapchain Image 35: Reads (▲), Writes (▲), Read/Write (▲), and Clears (△)

Filter $action()

EID	Name	Duration (µs)
	Frame #1855	---
0	Capture Start	---
1-26	Colour Pass #1 (1 Targets + Depth)	---
27-80	Colour Pass #2 (1 Targets + Depth)	---
81-136	Colour Pass #3 (1 Targets)	---
152	Present(**Swapchain Image 35**)	---

Input Layouts

Slot	Semantic	Index	Format	Input Slot	Offset	Class	Step Rate	Go
0	POSITION	0	R32G32B32_FLOAT	0	0	PER_VERTEX	0	➡
1	COLOR	0	R32G32B32A32_FLOAT	0	APPEND_ALIGNED (12)	PER_VERTEX	0	➡
2	NORMAL	0	R32G32B32_FLOAT	0	APPEND_ALIGNED (28)	PER_VERTEX	0	➡

Input Layout 119 : main() - primitivevs.hlsl View

Buffers

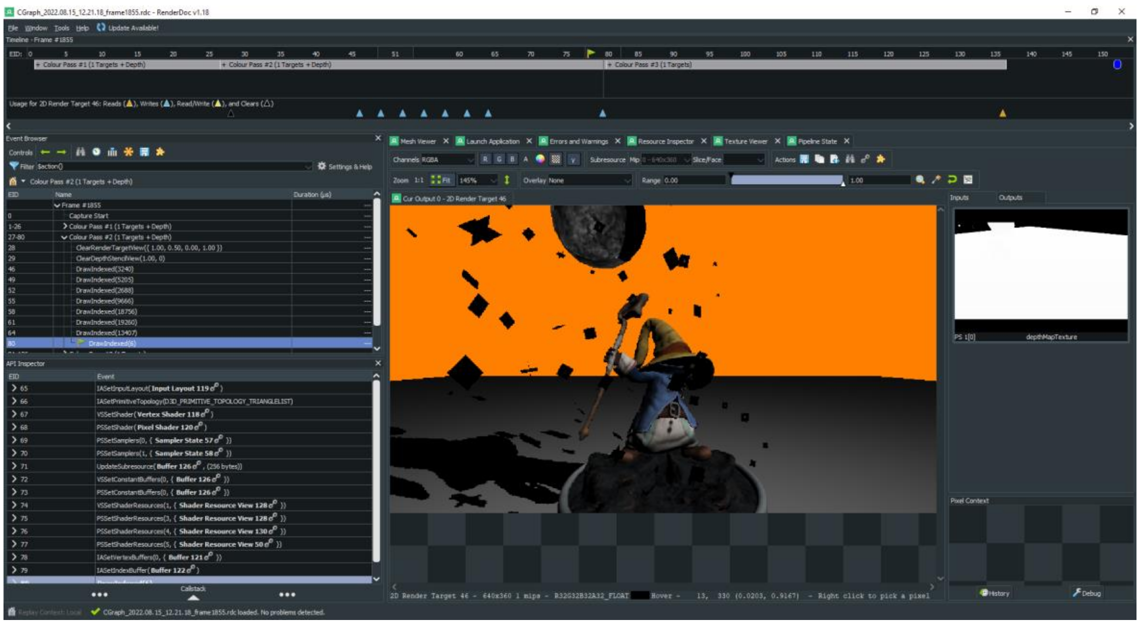
Slot	Buffer	Stride	Offset	Byte Length	Go
0	**Buffer 121**	40	0	160	➡

API Inspector

EID	Event
137	RSSetScissors({})
138	RSSetViewports({ { 0.00, 0.00, 640.00, 360.00 } })
139	RSSetState(**No Resource**)
140	OMSetBlendState(**No Resource** , { 1.00, 1.00, 1.00, 1.00 })
141	OMSetDepthStencilState(**Depth-Stencil State 54** , 1)
142	PSSetShaderResources(0, { **Base_Base_Color.png** })
143	PSSetSamplers(0, { **Sampler State 57** })
144	PSSetShader(**Pixel Shader 120**)
145	VSSetShader(**Vertex Shader 118**)
146	VSSetConstantBuffers(0, { **Buffer 126** })
147	GSSetShader(**No Resource**)
148	IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST)
149	IASetIndexBuffer(**Buffer 122**)
150	IASetVertexBuffers(0, { **Buffer 121** })
151	IASetInputLayout(**Input Layout 119**)

Replay Context: Local ✓ CGraph_2022.08.15_12.21.18_frame1855.rdc loaded. No problems detected.

Figure 29: RenderDoc Pipeline state view.



Usage for 2D Render Target 46: Reads (▲), Writes (▲), Read/Write (▲), and Clears (△)

Filter $action()

Colour Pass #2 (1 Targets + Depth)

EID	Name	Duration (µs)
	Frame #1855	---
0	Capture Start	---
1-26	Colour Pass #1 (1 Targets + Depth)	---
27-80	Colour Pass #2 (1 Targets + Depth)	---
28	ClearRenderTargetView({ 1.00, 0.50, 0.00, 1.00 })	---
29	ClearDepthStencilView(1.00, 0)	---
46	DrawIndexed(3240)	---
49	DrawIndexed(5205)	---
52	DrawIndexed(2688)	---
55	DrawIndexed(9666)	---
58	DrawIndexed(18756)	---
61	DrawIndexed(19260)	---
64	DrawIndexed(13407)	---
80	DrawIndexed(6)	---

API Inspector

EID	Event
65	IASetInputLayout(**Input Layout 119**)
66	IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST)
67	VSSetShader(**Vertex Shader 118**)
68	PSSetShader(**Pixel Shader 120**)
69	PSSetSamplers(0, { **Sampler State 57** })
70	PSSetSamplers(1, { **Sampler State 58** })
71	UpdateSubresource(**Buffer 126** , (256 bytes))
72	VSSetConstantBuffers(0, { **Buffer 126** })
73	PSSetConstantBuffers(0, { **Buffer 126** })
74	VSSetShaderResources(1, { **Shader Resource View 128** })
75	PSSetShaderResources(3, { **Shader Resource View 128** })
76	PSSetShaderResources(4, { **Shader Resource View 130** })
77	PSSetShaderResources(5, { **Shader Resource View 50** })
78	IASetVertexBuffers(0, { **Buffer 121** })
79	IASetIndexBuffer(**Buffer 122**)

2D Render Target 46 - 640x360 1 mips - R32G32B32A32_FLOAT Hover - 13, 330 (0.0203, 0.9167) - Right click to pick a pixel

Replay Context: Local ✓ CGraph_2022.08.15_12.21.18_frame1855.rdc loaded. No problems detected.

Figure 30: RenderDoc frame debugging.

5. Development

Software developing, most of the time, follows an iterative process, where it changes and mutates constantly throughout the development. A game engine as Tesseract makes no exception. Every system on the Engine has been programmed, tested, tweaked, redesigned and reimplemented, sometimes these changes are not so visceral. However, once a system is correctly implemented there is not such need of constantly changing it if it does not need it for the engine to function properly.

The following chapters depict how Tesseract came to be and what decisions had to be made.

5.1. Available Technology, why C/C++ and DirectX

Before starting to design and implementing systems I had to choose a language and a main Graphics API to build Tesseract.

I wanted to do this right, so I researched what were the most used languages in game engine development and why, and it came out that C/C++ was king due to the low-level memory management capabilities it offers the programmer. After that, Rust and C# were featured by some indie developers who coded their engines on those languages.

The choice for the graphics API was clear to me, DirectX 11. It has been the standard for many games in the industry for quite a long time. Even though nowadays there are some systems being implemented in DirectX 12, I chose DirectX 11 for simplicity's sake as both are similar, they only differ on how DirectX 12 is closer to the metal than the 11 version.

I also installed some engines in my workstation so I could see how they did certain things and took notes, for instance, which UI systems I liked most, immediate mode, or event based among other things.

5.2. Game engine design

Following Jason Gregory's knowledge of Game Engines Engineering imparted in his book, Game Engine Architecture, an Engine is the combination of all its systems, that we can differentiate in the following: Low-Level Engine Systems, Graphics, Motion and Sound and Gameplay.

General purpose game engines exist, but they are limited in some ways, as no engine is perfect for every situation. Thankfully, some engines allow developers to modify them to fit their own purposes if they do not want to create a proprietary engine, which is a high time and resource consuming task.

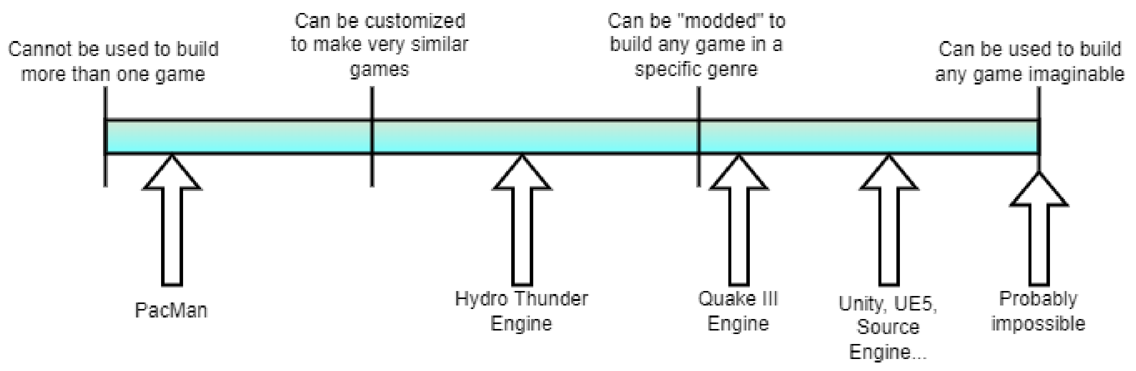

Cannot be used to build more than one game
Can be customized to make very similar games
Can be "modded" to build any game in a specific genre
Can be used to build any game imaginable
PacMan
Hydro Thunder Engine
Quake III Engine
Unity, UE5, Source Engine...
Probably impossible

Figure 31: Game Engine Reusability.

The Low-Level Systems comprise the Support Systems which include all math utils, memory management, containers, basic data types such as strings, vectors, matrices…, configurations, initialization, and shutdown of the different systems. In Tesseract, some of this basic support systems were implemented by me such as some vector operations that the DirectX Math library did not support out of the box, and some others were brought in by external libraries that we will talk about in next chapters such as EASTL.

Some of the different systems that comprise a game engine are:

- The File System and Resource Management systems, which allow to load and unload different files into memory as well as interpreting them into the data types the engine will use.
- The Game Loop and Rendering Loop, or Realtime simulation is the central part of the game, it splits into three main parts: initialization, update and draw.
  - Initializing the game and setting up the environment for the update and draw phases are both done during this phase. Here, we should set up the menu, identify the hardware's default capabilities, and construct major entities.´
  - The update phase's primary goal is to get everything ready for drawing, therefore this is where the physics code, coordinate updates, health point changes, character upgrades, damage given, and other similar activities belong. Additionally, the input will be recorded and processed here.
  - Finally, the draw step, when all of this information is displayed on the screen, once everything has been properly updated and is prepared. All the code necessary to handle and draw the levels, layers, characters, HUD, and other elements should be contained in this function.
- The Animation systems will handle the different types of character animation, skeletons, poses, skinning, blending…
- The Physics engine handles Collisions, rigid and soft bodies, dynamics, simulations…
- The Sound system could be as simple as being able to play a loop sound, clip or even handling sound in a physical mathematical way, as audio can behave as a wave in the 3D world.
- Input System such as recording keyboard key presses and mouse clicking. Input can be read in a variety of different ways: raw, applying post-processing, mapping inputs to predefined sets…
- Debugging and Profiling Tools such as Logging, Tracing, Debug drawing, Menus, Console, Debug cameras…

- The gameplay systems need to be implemented thinking of reusability and interoperability in the Game World Editor within the engine, without forgetting that the Game Loop needs to be optimized enough for the game to be able to play in Realtime at a performant framerate. We usually talk about game objects or entities, depending on how the architecture of the game is designed.
- The Graphics system or Rendering Engine will be in charge of producing high-quality images from the input provided by the engine such as 3D polygons, textures, camera data and lighting data. It follows the previously seen Rendering pipeline, which is usually modified to fit the purposes of the game engine project and is able to be extended to implement features such as advanced lighting, global illumination, overlays, antialiasing, shadow projection… The renderer is a piece of software that is written to take advantage of the features in the GPUs and is prone to receive code optimizations regularly to comply with performance requirements (such as outputting a frame every 1/60th of a second). Renderers also need to get data from other systems such as physics for positions, file system for textures, animation if a polygon needs to be moved somewhere else… so this systems communication needs to be designed carefully to be performant and have healthy data transfers.

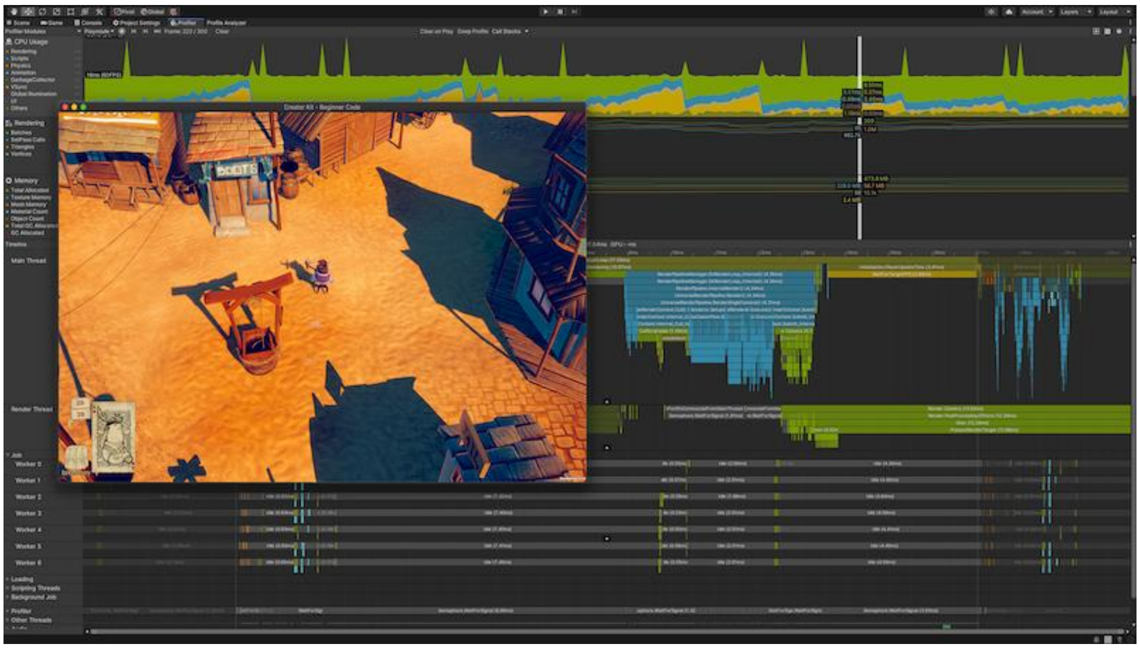


Figure 32: Unity Profiling Toolset.

One of the different ways of achieving that performance is following a data-oriented mindset, which I did while developing Tesseract and we will talk about more in the following chapter. I wanted to focus on how the systems treated and structured data and outputs so the next system in the pipeline could take it from there and do their job.

5.3. Data Oriented Design

As previously said, I wanted to follow a data-oriented mindset during the development of Tesseract.

Data-oriented design is a method to optimize programs that has been used especially in video game development to make the most effective use of the CPU cache. Focusing on the data layout, separating and organizing fields based on when they are needed, and considering data transformations are the main objectives of this approach.

To understand why this approach to programming is more performant, we will need to take a look at the principle of Locality of reference, which in computer science, is the inclination of a processor to repeatedly access the same set of memory locations during a brief period of time. There are two basic types of localities that are the ones of importance for this subject:

- Temporal Locality, which is the reuse of certain data and/or resources across a brief period of time. This means that if a particular memory location is accessed at some point in time, it will be likely that the same location will be again referenced in the future. Then, it is common to try and store the data within that memory location in a faster memory storage such as the CPU L1 Cache to reduce the latency of following references.
- Spatial Locality or Data Locality, which is the usage of data elements that are relatively close between them regarding their storage location. There is a special case for this locality: Sequential locality, which occurs when data elements are laid and accessed in a linear fashion such as traversing one-dimensional array elements. This means that if a memory location is referenced at some point, then it is likely that the nearby memory locations will be referenced shortly after. Some techniques regarding this attempt to guess the size of the area around the reference to prepare the access to the next one faster.

Thanks to these principles, systems that present a strong locality of reference are prone to be optimized with techniques such as caching or prefetching memory.

Caching is a technique that takes advantage of the CPU caches, which are the fastest type of memory available to the computer. These pieces of memory are small and close to the processor cores, featuring a hierarchy of different levels such as L1, L2, L3 and even L4 varying in speed and size. Caching involves laying data in the program in a way that it can be accessed linearly from the CPU cache, and sometimes even considering the size of the CPU cache when programming for a specific platform where the programmer knows the hardware specification, avoiding unnecessary cache misses. Some techniques used by DOP involves having Structures of Arrays (SoA) and Arrays of Structures (AoS) for different data depending on how it will be accessed from the CPU.

A practical Example would be the position of an entity, usually we will want to access the X, Y and Z members of position together, so there would be no point in having a struct that would split the three members, in that case a AoS would be superior to a SoA. However, if we had a component that had several members that were accessed individually a SoA would be faster.

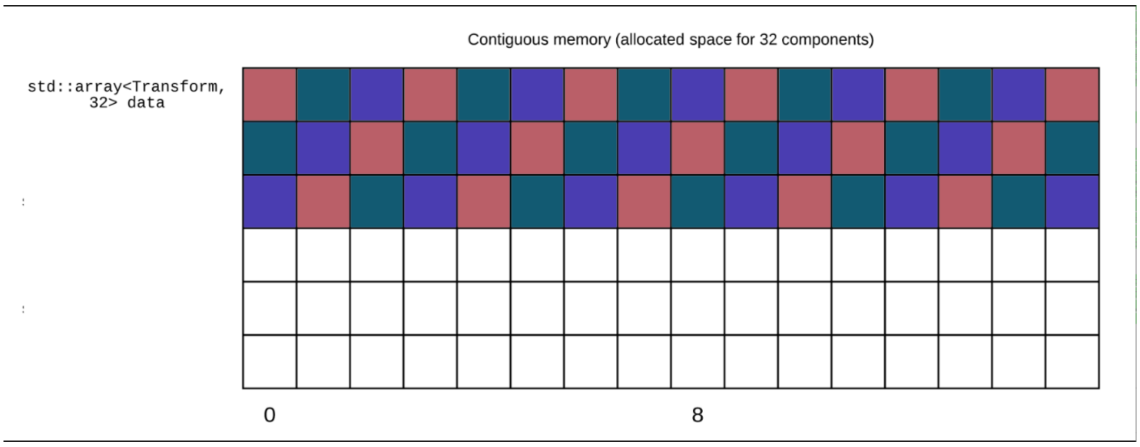

Contiguous memory (allocated space for 32 components)
std::array<Transform, 32> data
0
8

Figure 33: Array of Structs, in contiguous memory. Red:X,Green:Y,Blue:Z.

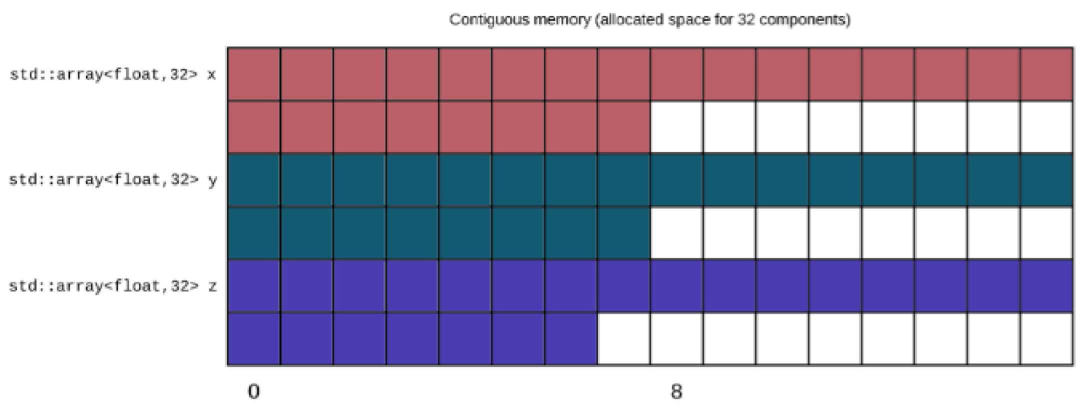


Figure 34: Struct of Arrays, in contiguous memory.

As we can see, these two figures depict the previous example, the first image would be faster to access due to having the data that is being accessed close together. The second image would be faster if every member of the struct was accessed individually.

Figure 35: The Witness, a game that uses a Data Oriented approach.

Prefetching nowadays is done automatically by the CPU by loading the following memory addresses contents into the CPU cache so if the following instructions need them, it will be already loaded and the CPU will not stall or generate a cache miss, which is when a CPU doesn't have the data element needed by the instruction and will need to fetch it from memory and load it into the cache so it can be used. Optimization techniques that structure the programs around the CPU cache tries to avoid this, as fetching and loading data into the cache unnecessarily wastes a lot of processing time.

As we can infer from the information, Data Oriented Programming (DOP) is about how the data is laid and accessed instead of what it could represent. DOP separates data from functionality, as generally the functions are general purpose and can be used with large amounts of data in any point of the process.

As CPUs are friendly towards locality of reference, DOP rules create many benefits such as allowing for a much easier way to implement Parallelization to execute tasks simultaneously, while in Object Oriented Programming (OOP) this can be hard due to the collision of multiple threads trying to access the same data. In DOP, as we group similar data together and write code focusing on the data processing in general, it becomes easier to task multiple threads to process those functions. As we can see in the following Figures, it would be easier for the CPU to parallelize the execution if the data was laid out similar to Figure 9, regardless of the usage of SoA or AoS in those data layouts.

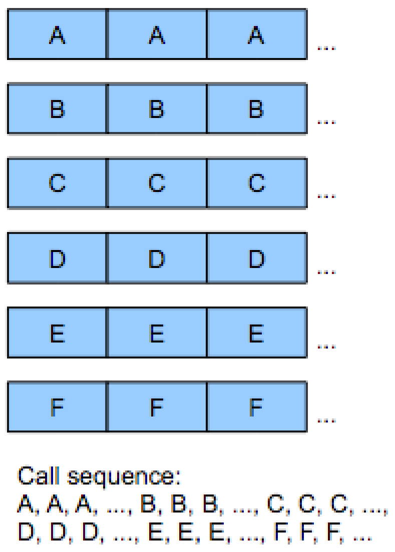

A A A ...
B B B ...
C C C ...
D D D ...
E E E ...
F F F ...
Call sequence:
A, A, A, ..., B, B, B, ..., C, C, C, ...,
D, D, D, ..., E, E, E, ..., F, F, F, ...

Figure 36: Access of data by paralell threads if data is laid out following a DOP approach.

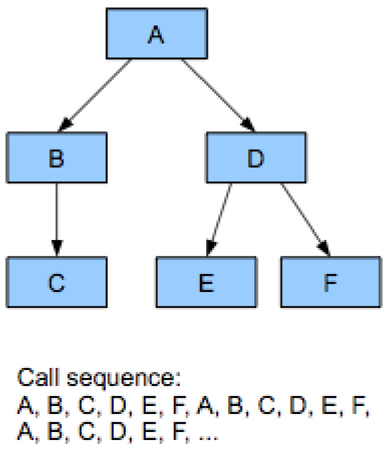


Figure 37: Access of data by paralell threads it data is not laid out following a DOP approach.

On the CPU side of the design, I wanted Tesseract to be as friendly as possible towards memory and the CPU cache so my program could run smoothly. This design paradigm is widely used in video game development and focuses on the data layout and the principle of locality, as we saw previously.

This paradigm became popular during the seventh generation of video game consoles (PS3 and Xbox 360) due to their lack of CPU processing power chosen to redirect more power and budget towards the GPUs.

Data Oriented Design organizes code around data and tries to create good data locality in order to generate as few cache misses as possible.

5.4. Systems & Architecture

A game engine is a very difficult piece of software that needs many different systems depending on the target development idea and an architecture that supports all of them in a performant way.

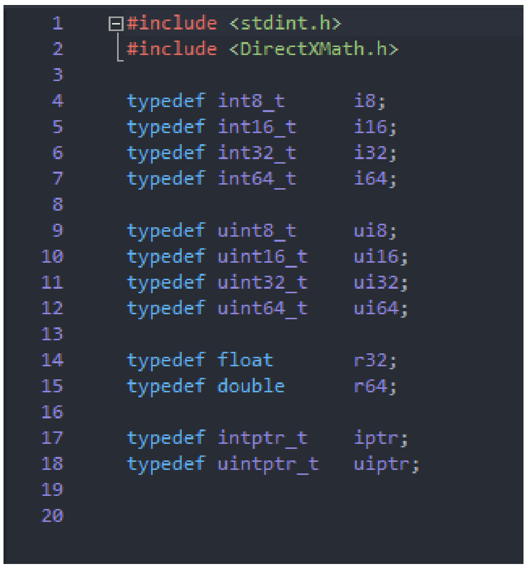
In Tesseract's case, due to the limitation of time and resources, I chose to go for something simple but well done and implement several features that are, to my opinion, the base features every real-time renderer should implement.

5.4.1. Third-Party integrations

As Tesseract needed some base functionality, I needed to integrate some external code, this is something very common in software development, as it is a monolithic enterprise to implement everything you need by yourself, you usually need to use dependencies.

In my case, I added my own type definitions to know the size and type in a way that was more readable for me.

DirectXMath was my choice for vector/matrix math operations, it supports fast mathematical expressions through SIMD operations (Single Instruction Multiple Data) which operates by using the bigger registers on the CPU.



```cpp
#include <stdint.h>
#include <DirectXMath.h>

typedef int8_t      i8;
typedef int16_t     i16;
typedef int32_t     i32;
typedef int64_t     i64;

typedef uint8_t     ui8;
typedef uint16_t    ui16;
typedef uint32_t    ui32;
typedef uint64_t    ui64;

typedef float       r32;
typedef double      r64;

typedef intptr_t    iptr;
typedef uintptr_t   uiptr;
```

Figure 38: Tesseract type definitions

The standard C++ library is okay for development, however I integrated EASTL instead, which is a version of the standard library focused on game/game engine development and done by Electronic Arts. It is very complete and support quite some features which are useful while developing while keeping performance up compared to the standard library.

For file loading I chose Assimp (Open Asset Import Library), which is a very well-known piece of software to load file assets into memory, I used it to load 3D models. However, all the processing of the model data such as extracting textures from the material to load them separately, extracting the vertex information and position and creating my own data layout is done in Tesseract's code.

In order to load textures and work with Mip Maps I integrated DirectXTools, which is a set of functions and separate programs that allow you to create and process data to use within the DirectX Pipeline, in my case DirectX 11.

Finally, I needed a GUI system, and I integrated ImGUI, which we will talk about more in a later chapter, it is an Immediate Mode UI library, open source and highly customizable, supported by AAA game companies such as Activision, Blizzard, King…

5.4.2. Mesh, Textures, Shader

As previously mentioned, the Model information is loaded into memory by Assimp, from there, I process the information to only store in my dedicated memory layout what I need and in the correct order.

A common Model would contain: One or more Meshes, different textures, mathematical information per vertex such as normal, binormal, tangents… Along with other information such as vertex color presence, coordinate system, this last one is usually tricky, as not every software uses the same coordinate system, some uses Left Handed systems and others Right Handed, this forces for further processing to transform the coordinate system from one to another if needed.

This allowed to store vertex information performantly while keeping the different objects within the model differentiated. I allow rendering of 3D models with different meshes within with their own textures and without the need for loading them separately, as Tesseract's code will handle it.

The texture names and path are extracted from the Model information and loaded later into the asset pipeline, where the bigger texture size is chosen and all the subsequent Mips are generated for correct rendering at different distances.

Also, I differentiate from Linear and sRGB information textures, converting them in order to do calculations in a correct way and not miss color information.

Currently the shaders I use within tesseract for entities support bump/normal mapping, simple lighting, and shadow mapping, Linear/sRGB conversion on the fly if needed, Lambert for diffuse and Phong for specular values and a prototype version of GGX as different shading model options.

As we will see in later chapters, to implement shadow mapping several shaders needed to be created in order to generate the information for the shadow reprojection.

5.4.3. UI and Profiling

ImGUI is a very popular UI system which has a low processing performance hit, this is why a lot of games such as League of Legends Teamfight tactics engine use it.

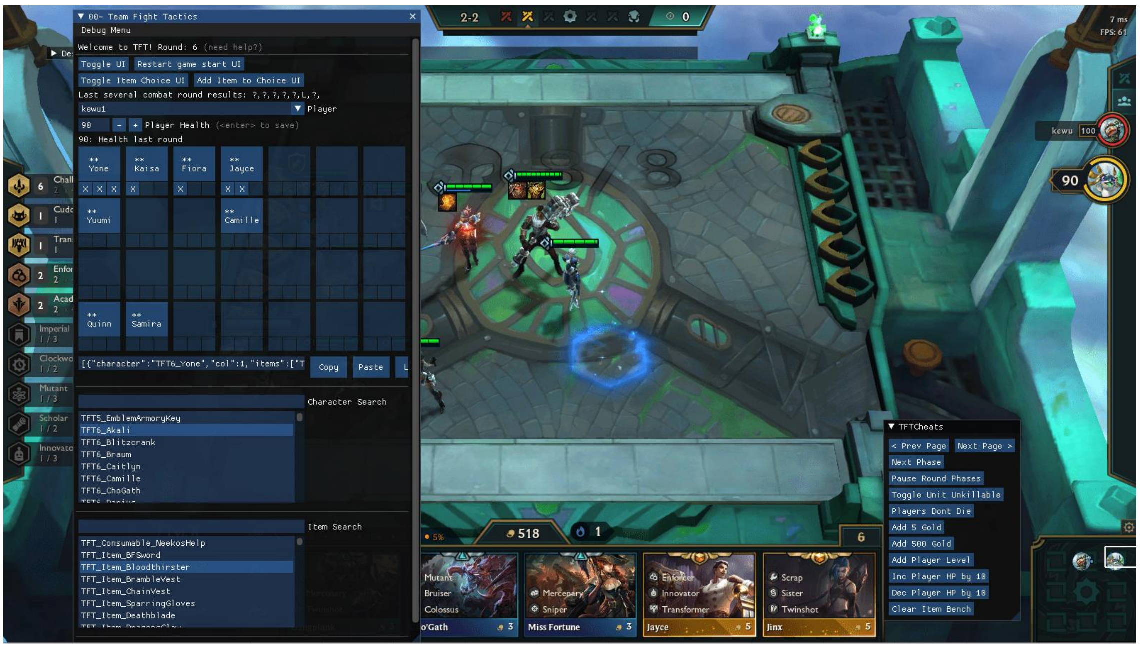


Figure 39: Teamfight Tactics from League of Legends using ImGUI

It allows for quick implementations without the need for creating callbacks or waiting for events. In Tesseract I used it mostly for debugging reasons or to implement quick tests for the vertex transformation or visualization of textures or textures for the shadow mapping to ensure it was loaded and in good state.

Another feature I implemented was a camera information visualizer which allows to see the camera vectors, direction, position, orientation… It proved really useful when implementing the free look camera, as it was tricky.

It is in constant evolution and open source, so any user can modify its source code and recompile it fast. The viewport used in Tesseract is a ImGUI window within the Windows window handle, which I configured and spawned communicating directly with the Windows OS through Microsoft's windows built-in libraries.

Currently the UI system in Tesseract allows for free move and docking, resizing of windows, and even moving the windows outside the main window handle, though this is not recommended as it requires additional processing power and reduces performance significantly.

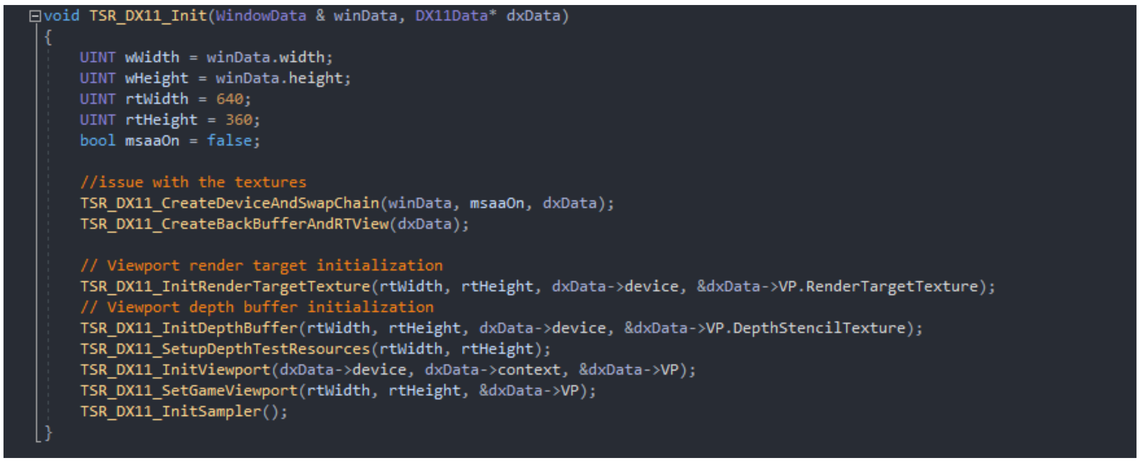
5.4.4. DirectX11 Graphics API

DirectX 11 was the chosen Graphics API, I decided to work directly with this API in order to learn how graphics API's work, as they are all similar, they only differ on implementation mostly, but the required steps are mostly the same, with the exception of Vulkan and DirectX 12 which are closer to the metal and require the programmer to fill many descriptor structs for the correct functionality which translates to better performance if done correctly.

The DirectX framework is developed by Microsoft, so it has a lot of documentation and support throughout the Internet, as it is one of the standards.

The DirectX 11 workflow for initializing it in Tesseract is composed of several steps:

1. Creating the Device and Swap chain, which gets the GPU information and configuration from the descriptor and initializes the swap information for the framebuffers.
2. Creating the Back buffer and Render target views, which are the buffers and handles to show them on screen.
3. Initialization of Render Target, Depth Stencil Textures.
4. Initialization of Depth Stencil Buffer and Depth Test Resources, which composes all the elements that are needed by the shadow mapping.
5. Initializing the main viewport (windows window) and the game viewport (ImGUI window).
6. Initializing the different samplers that are used to extract information of textures.



```
void TSR_DX11_Init(WindowData & winData, DX11Data* dxData)
{
    UINT wWidth = winData.width;
    UINT wHeight = winData.height;
    UINT rtWidth = 640;
    UINT rtHeight = 360;
    bool msaaOn = false;

    //issue with the textures
    TSR_DX11_CreateDeviceAndSwapChain(winData, msaaOn, dxData);
    TSR_DX11_CreateBackBufferAndRTView(dxData);

    // Viewport render target initialization
    TSR_DX11_InitRenderTargetTexture(rtWidth, rtHeight, dxData->device, &dxData->VP.RenderTargetTexture);
    // Viewport depth buffer initialization
    TSR_DX11_InitDepthBuffer(rtWidth, rtHeight, dxData->device, &dxData->VP.DepthStencilTexture);
    TSR_DX11_SetupDepthTestResources(rtWidth, rtHeight);
    TSR_DX11_InitViewport(dxData->device, dxData->context, &dxData->VP);
    TSR_DX11_SetGameViewport(rtWidth, rtHeight, &dxData->VP);
    TSR_DX11_InitSampler();
}
```

Figure 40: Tesseract DirectX 11 Initialization function.

As a side note, all functions that starts with TSR are Tesseract engine specific.

Other functionality I implemented are the use of structured buffers, which require specific configuration to be used within shaders but allow for an array-like access of multiple resources within the same shader.

5.4.5. Transformation Matrices and Camera Projection

In computer graphics, matrices are widely utilized, and matrix transformations are one of the fundamental building blocks of any visualizations. A series of matrix transformations enables the display of 3D objects on 2D screens.

All these transformations are done in an affine space, which is a space that generalizes the geometric properties of Euclidean space. This means that these spaces define how points, lines and planes are allowed to transform. In this case Lines need to stay linear, planes planar and parallel lines parallel.

Also, there is a fourth coordinate included in all calculations called the homogeneous coordinate, or w, which allows for an 4D Homogeneous Space/coordinate system. The w coordinate scales the x, y and z dimensions. When w is equal to one, the coordinates stays the same due to not being scaled, in 3D graphics w is usually set to 1 so it doesn't have effect on the x, y and z values. As we cannot divide by zero, when w is that value, we interpret that as a direction vector.

Transformations are presented in a Transformation Matrix, which is composed of the multiplication in order of three different matrices:

- Translation Matrix, which moves objects with respect to their current locations. In the following figure we can see the translation defined by the T vector.

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 41: Translation Matrix

- Rotation Matrix which rotates objects around the x-, y-, or z-axis. The following matrices define this rotation about each axis if the desired angle of rotation is theta.

X-axis rotation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x & 0 \\ 0 & \sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y-axis rotation

$$\begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Z-axis rotation

$$\begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 & 0 \\ \sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 42: Rotation around predefined axis

- Scale Matrix, which given a vector with three members each one of them representing the scale along a 3D axis, would generate a 4x4 matrix which we will see in the next figure.

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 43: Scale Matrix

These matrices are combined into a single matrix, calling the Translation matrix T, the Rotation Matrix R and the Scale Matrix S. The Transformation matrix or Model Matrix is calculated by multiplying the matrices in this order: TRS.

The Model Matrix is unique per entity in the world. This Matrix allows for transforming the vertices from the 3D model from the Model or Object space into the World Space, as we can see depicted in the following figure.

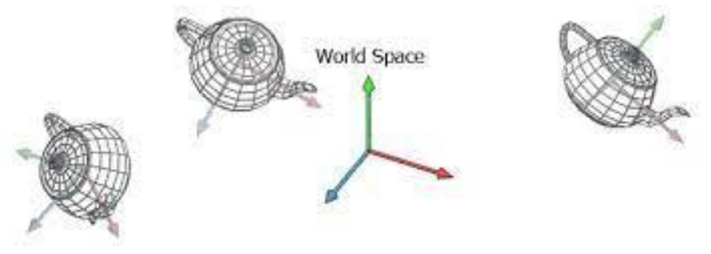
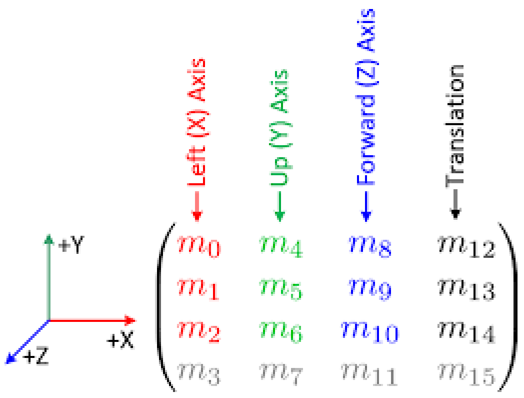


Figure 44: Model Space and World Space.

In order to project onto the 2D display the series of different objects we need to define two more matrices which will be the View and Projection Matrices:

- View Matrix, which converts from world space to camera or view space. This matrix is defined by the camera eye position and the up, forward and right vectors, orthogonal between them. It needs to follow a handed system. In Tesseract's case, as DirectX, I use a Left-Handed system, which defines the Z positive axis pointing away from the camera eye and the Y axis pointing Up.



$$\begin{array}{cccc} \text{Left (X) Axis} & \text{Up (Y) Axis} & \text{Forward (Z) Axis} & \text{Translation} \\ \left(\begin{matrix} m_0 \\ m_1 \\ m_2 \\ m_3 \end{matrix}\right. & \begin{matrix} m_4 \\ m_5 \\ m_6 \\ m_7 \end{matrix} & \begin{matrix} m_8 \\ m_9 \\ m_{10} \\ m_{11} \end{matrix} & \left.\begin{matrix} m_{12} \\ m_{13} \\ m_{14} \\ m_{15} \end{matrix}\right) \end{array}$$

Figure 45: View Matrix

- To construct a projection matrix, we will need to define a view frustum, which is a volume of space that is potentially visible to the camera. It is comprised of six clip planes, being the near and far planes the most important as they correspond to certain camera space Z values. The far clip plane prevents rendering objects that are farther than that Z value. The projection matrix maps the view frustum into the homogeneous clip space.

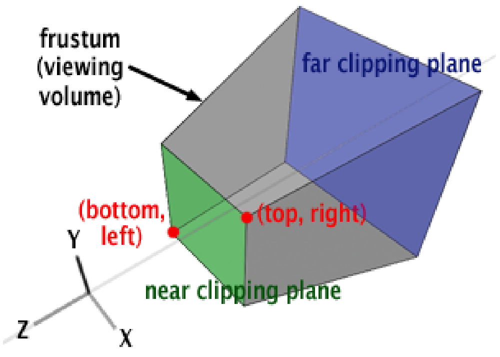


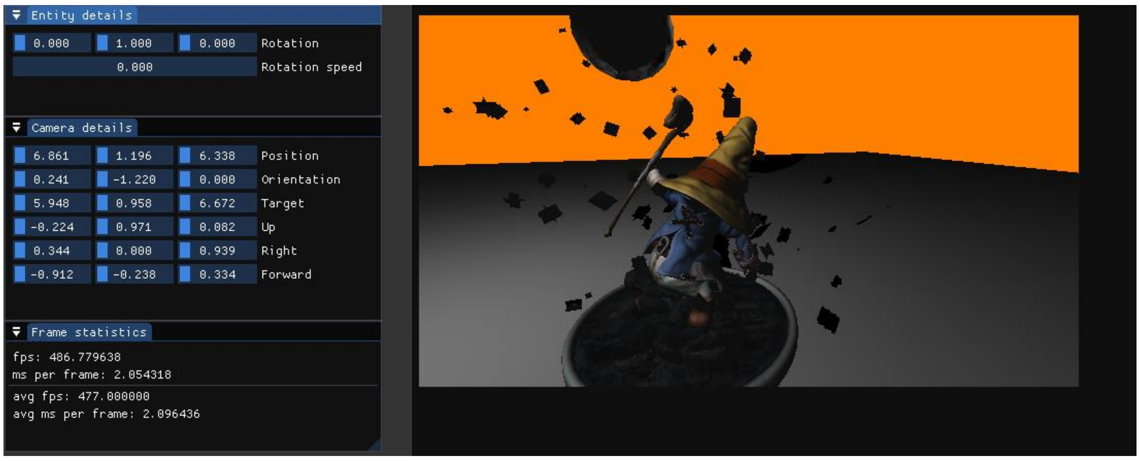
Figure 46: View Frustum.

All these matrices needed to be calculated within Tesseract's systems to render properly the 3D world to the viewport.

5.4.6. Free look camera Tool

A free look camera tool is highly important to debug rendering features, being able to move around the 3D world and look at things from different angles. Moving closer or farther it makes a huge difference both for debugging and implementing features and systems because it allows to spot bugs and things that were not failing but were neither working as expected, although I couldn't see it from a different point of view.

Following the previous investigation on the different coordinate systems and matrices present in computer graphics. A camera that can traverse the 3D world is essential in 3D graphics.



Entity details

0.000	1.000	0.000	Rotation
0.000			Rotation speed

Camera details

6.861	1.196	6.338	Position
0.241	-1.220	0.000	Orientation
5.948	0.958	6.672	Target
-0.224	0.971	0.082	Up
0.344	0.000	0.939	Right
-0.912	-0.238	0.334	Forward

Frame statistics

```
fps: 486.779638
ms per frame: 2.054318
avg fps: 477.000000
avg ms per frame: 2.096436
```

Figure 47: Tesseract's Free Look camera and camera details panel

5.4.7. Texture Mapping

This technique allows for models to be represented with the textures associated to it in the 3D world. It requires several steps to use it correctly:

1. Texture Coordinates needs to be properly generated at some point of the pipeline.
2. Binding the texture coordinates to the shader and also binding the correct texture for each model along with its mipmaps.
3. Configure and bind sampler in the DirectX 11 pipeline to extract color samples from the texture so we can paint our model.

As Tesseract allows for multiple meshes within a model and each mesh can feature a different number of textures the shaders needed to be prepared for this.

Stretching an image over a 3D object's surface is known as texture mapping. A texture is what we call an image utilized in this way, and we may use textures to indicate things like color, roughness, and opacity.

While stretching a 2D texture over a standard shape, like a cube, is simple, doing it with irregular geometry, like a face, is significantly more difficult. Over the years, several texture mapping techniques have been created, but UV mapping is now the most common.

UV Mapping allows us to have the capacity to connect points on the geometry and points on the texture. The texture is divided into a 2D grid using UV mapping, with the points (0,0) at the bottom left and (1,1) at the top right. The exact center of the picture will then be at the position (0.5, 0.5). Similar to this, every point in a geometry has a location in the mesh's 3D local space. Then, UV mapping is the process of tying 2D texture points to 3D geometry coordinates. In simpler words, texture mapping adds a 2D coordinate which represents a point within a texture. Each vertex has a texture coordinate associated that can be used to sample a color from the nearest texels (texture pixels) from that point. There are several techniques to sample from textures, and different ways of storing textures in memory.

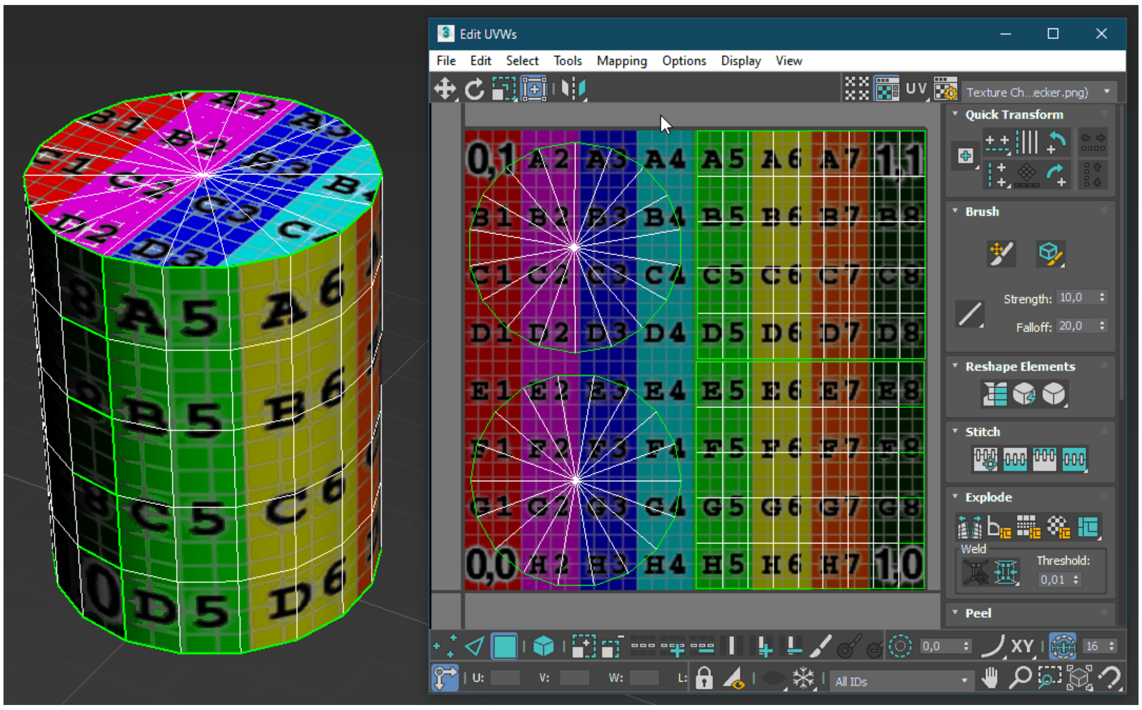


Figure 48: 3D Model and Texture UV Mapping.

In Tesseract, each texture when is loaded generates a mipmap, which is an image that contains a sequence of the same image represented in different resolutions. It is used to reduce rendering artifacts and increase performance, by choosing the correct image resolution to apply as texture to a model based on the depth of the 3D object. Before having mipmaps, an image seen from afar would be sampled from the high resolution texture, which would show bigger jumps between the colors as the sampling would be done from points very far away from each other within the texture, after adding the mipmaps and starting using them, that ceased to happen because the mip or image slice chosen to sample from far distances was a lower resolution one, which means that colors would have been filtered together.

Filtering is referred to the technique the sampler uses to fetch the color from the texture, there are several types:

- Linear filtering samples from an individual mipmap while it interpolates the two closest mipmaps that are relevant to the particular sample.
- Bilinear filtering or blending takes the four texels that are nearest to the pixel center and sample them at the correct mipmap level l. Then, the colors are combined by using a weighted average which is dependent on the distance.
- Trilinear does the same but combining the two closest mipmap levels using a linear interpolation after having done bilinear to each one of the mips.
- Anisotropic improves the quality of distant objects that are viewed at an angle by not using only square but samples the texture in a non-square shape which tries to map the footprint of the pixel.

In Tesseract, textures are sampled using bilinear filtering by default.

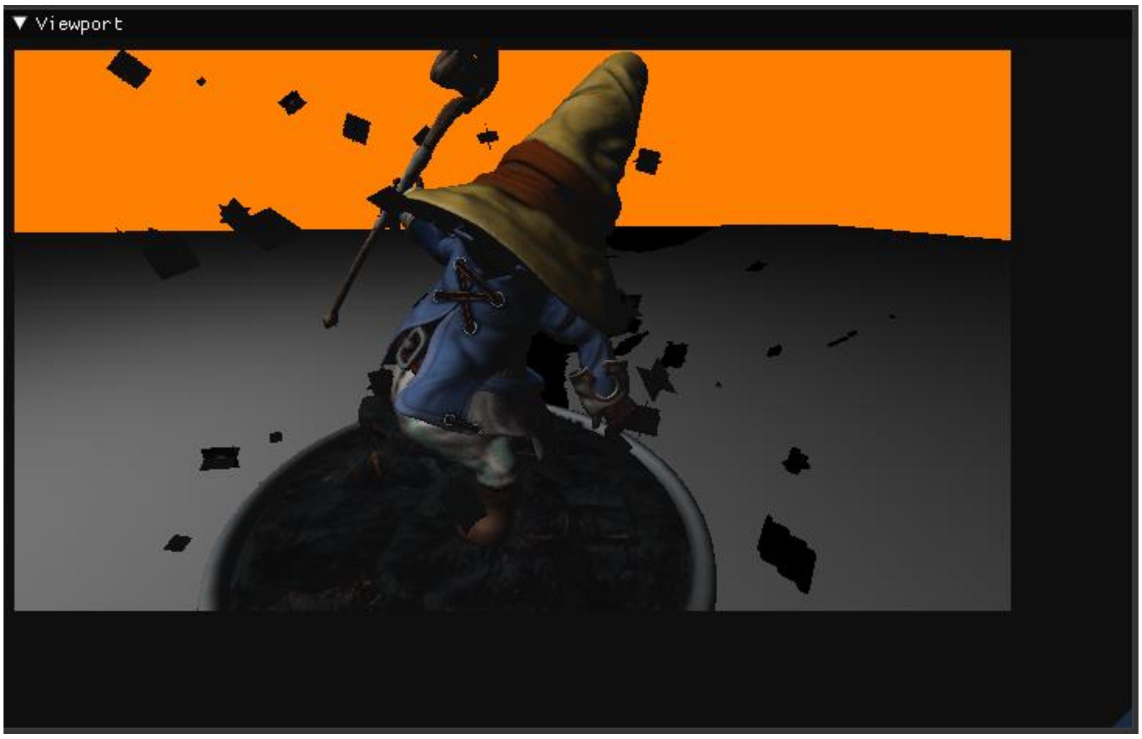


Figure 49: Texture Mapping featuring Models with multiple meshes with different textures each.

5.4.8. *Bump-Normal Mapping*

This kind of rendering effect allow us to create geometric complexity without adding more polygons. To implement it I needed to do some research:

- How normal vectors are represented locally versus world normal information and how to transform them due to the need of the latter in order to implement this effect. This allowed me to expand my knowledge about vector coordinate reprojection.
- Investigate about the different coordinate systems used in rendering and learning in depth about them.
- How to calculate tangents and/or binormals and how they are represented in normal world space.
- Understand how the algorithm works in order to implement it into my renderer.

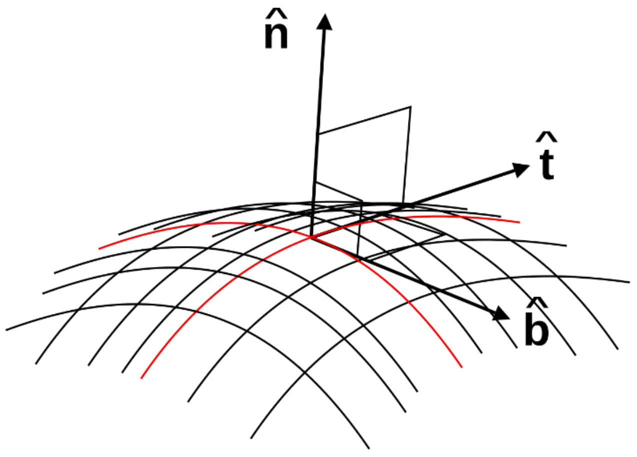


Figure 50: Normal,Tangent and Binormal Vectors over a curved surface point.

After having these three components, we use each one of them to calculate the X,Y and Z values of the Bump Normal intensity, and then normalize it. This will grant us the perturbed normal for the point in the surface.

Normal or Bump mapping is used to simulate bumps and creases on an object's surface. To do this, the object's surface normals are disturbed, and the perturbed normal is used for calculating illumination in our case, with the Phong algorithm. Despite the fact that the underlying object's surface is unaltered, the outcome is an apparent rough surface as opposed to a smooth one. Resulting in extra geometric complexity without adding polygons.

Bump/Normal mapping is seen best from a closer distance, as we will see in the following figures.

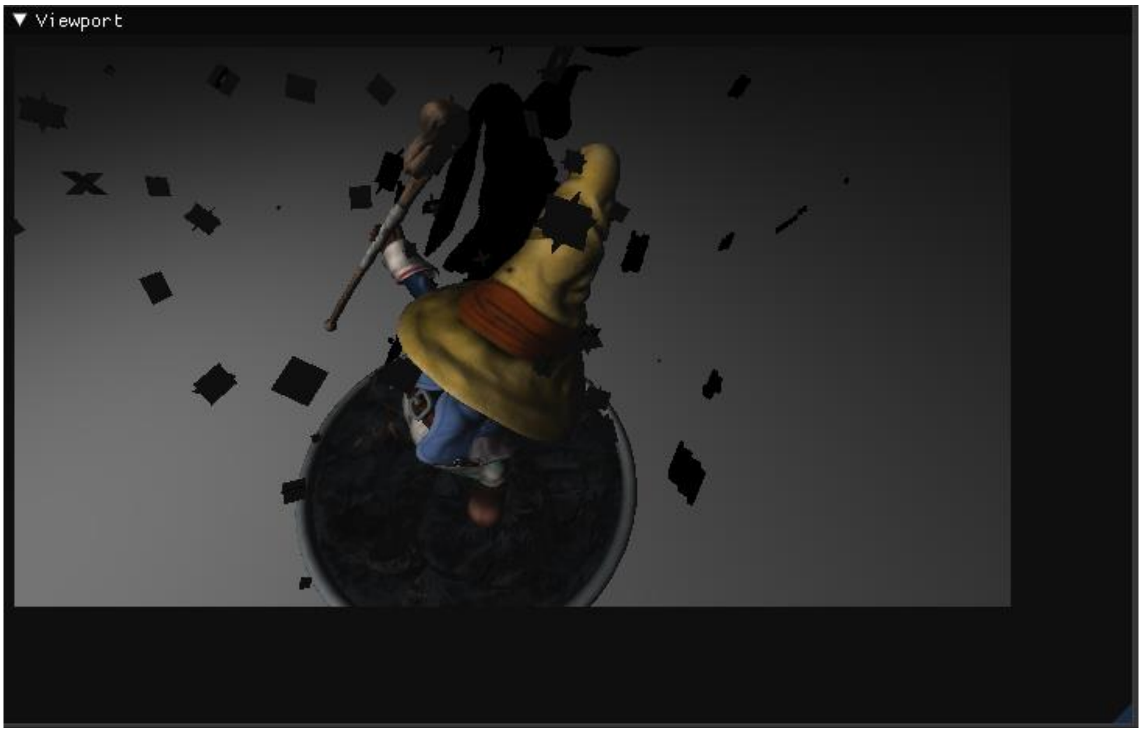


Figure 51: Almost not recognizable Bump/Normal mapping due to being far from the model.

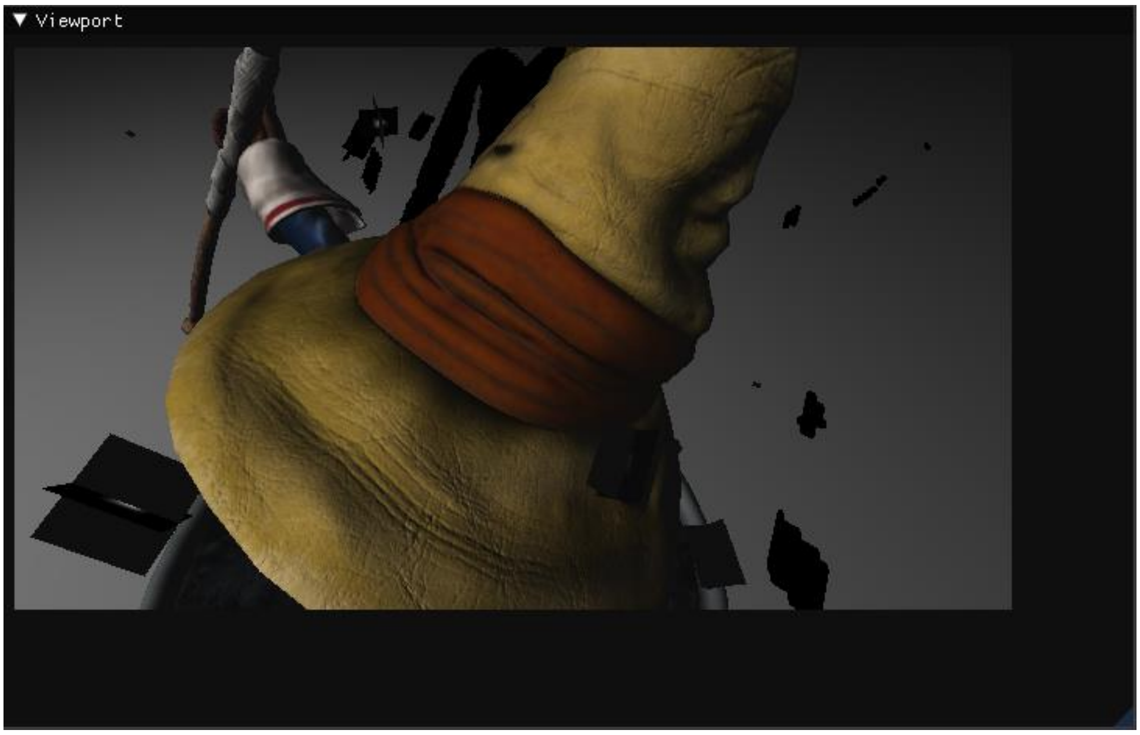


Figure 52: Strong Looking Bump/Normal mapping when we get closer to the model.

5.4.9. Shading Models

Diffuse objects in Tesseract follow Lambert's shading model for diffuse objects and Phong for specular, which relates the Incident ray of light with the normal vector from the point in the surface that is going to be lit or shaded. This is called the Lambert's Cosine Law and it dictates that the amount of light that a surface receives is directly proportional to the angle between the surface normal and the light direction. Commonly known as Light Intensity (LI).

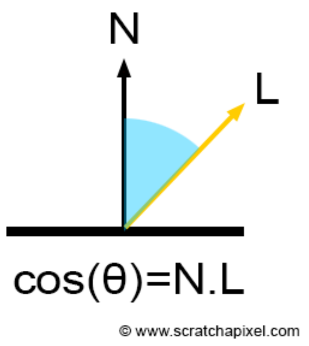


Figure 53: Lambert's Cosine Law representation.

So, to calculate the final illumination of the point for the diffuse in tesseract, as we implemented normal mapping, we take the processed Normal world vector and do a dot product between that and the Light Direction to get the light intensity, which must be in the 0 to 1 range.

In Tesseract, I supported two types of light, Directional and Point Light.

- Directional Lights, which mimics the appearance of light coming from an indefinitely far source. This makes it the best option for replicating the light of the sun since it guarantees that any shadows cast by this light will be parallel.
- Point Lights, which emits light in all directions, just like a realistic light bulb would do. The light is emitted uniformly in all directions from a single point in space. This is done by including an attenuation factor into the light calculations and damping its intensity depending on how far the pixel is from the center of the light.

Also, in Tesseract I use a small value for ambient light to simulate the light rays bounding everywhere, it is not the best way of simulating that, but for my use case it works well enough.

As for GGX (which is the name of a microfacet distribution or Bidirectional Reflectance Distribution Function or BRDF), it is a shading model focused on metallic reflections on rough surfaces. The GGX shader tries to mimic the reflection point and falloff that generates having a glossiness and tail values for the GGX shader allowing to create more realistic metallic objects.

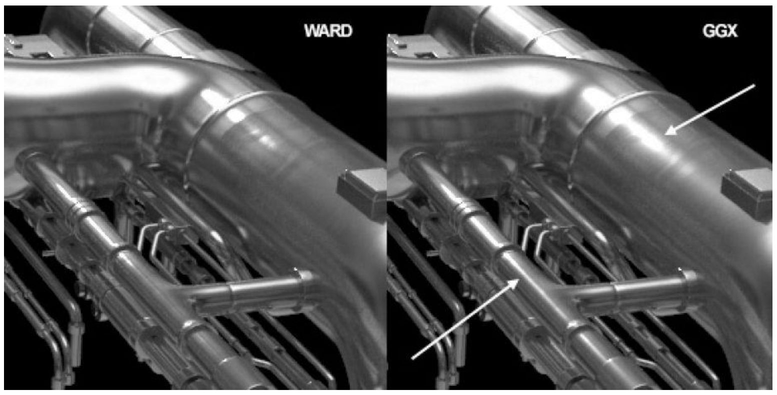


Figure 54: GGX shading model vs Ward shading model applied to the same metallic object.

5.4.10. Shadow Mapping

Shadow mapping is essentially the simplest way to get shadows projected from light sources in renderers that use rasterization, as raytracing calculates shadows differently. However, the thing is that it is not so simple, it involves several concepts that aren't easy to understand unless you stop to think about them and perform several tests on your own to see how it works.

The main algorithm to implement shadow mapping is simple, but it needs previous work done and working perfectly or else it won't project the correct shadow or nothing at all. The problem is that shadow mapping is an unresolved problem and shadow reprojection by shadow mapping is one of the different ways of tackling the issue.

For one directional light source all it takes is to set a depth texture of the scene from the light point of view taken by an orthographic projection, storing it and pass it onto the shader that will calculate the shadow along with some information for it to perform the reprojection and shading of the correct pixels. This is the list of things I needed to do in order to get it working:

1. Initialize a texture for the depth map, which needs a specific format.
2. Initialize a specific Sampler to sample from the depth texture.
3. Render to the depth map texture from the light view. In order to do this, I needed to calculate a few things and set the render target to the depth texture:
    a. The Light position in World space.
    b. The Light position (LightVP) transformed with the matrix that resulted from the multiplication of the World matrix from the object, the Light's view matrix and the Light's orthographic Left Hand projection matrix.
    c. Pass all that information into the depth pixel shader which will calculate the depth value by dividing the LightVP variable z member by the w member and write that onto the texture.
4. After the texture was correctly filled, to ensure this I implemented a texture viewport where I could see if the texture was correctly being calculated and stored, binding it into the shader that was going to calculate the shadow.
5. To calculate the shadow, I implemented an algorithm called a depth test, which calculates the distance to the light as the distance from the point to the plane from where the light is projected orthographically. If this distance is greater than the depth for the light in that point, then that pixel is shadowed.

Tesseract only implements hard shadows for now, but there are many different techniques to implement other types of shadows like soft shadows:

- Percentage Closer Filtering (PCFS) Shadows which filter with a tighter proportion for shadow map sampling. The hardware does four depth comparisons and bilinearly interpolates the shadow value using the fractional portion of the texture coordinate. The shadow result is the percentage of the texel-size area that is in shadow.
- Percentage Closer Soft Shadows (PCSS), this technique builds on top of the regular shadow mapping, it returns a float value for each pixel in the eye view indicating the amount of shadowing at each shaded point. PCSS is based on the observation that as the size of the PCF increases, the shadows are softer. To adjust the algorithm it follows three steps:

- The depths closest to the light source compared to the shadowed point known as the "receiver" are searched for in the shadow map and averaged. The size of the search area is determined by the size of the light and the distance between the receiver and the light source.
- Calculation of the penumbra width which depends on the light size and receiver distances from the light by using parallel planes approximations.
- Finally, we use regular PCF with the number of depth comparisons being proportional to the previous calculation of the penumbra.

There is much room for improvement in shadow mapping, for instance, in DOOM 2016 to render shadows coming from multiple lights, they implemented a shadow map atlas system where distinct depth map is created and stored into one tile of a massive 8k×8k texture atlas for each light that casts a shadow. Not every depth map, however, is computed every frame: DOOM makes extensive use of the preceding frame's output and only regenerates the depth maps that require updating. It makes sense to leave a light's depth map untouched when it is static and just throws shadows on objects rather than needlessly recalculating it. But a new depth map has to be constructed whenever an opponent is moving under the light.

The size of a depth map can change depending on how far the light source is from the camera, and newly created depth maps may or may not remain inside the same atlas tile.The depth map's static section may be cached in DOOM, which then computes just the dynamic meshes projection and composites the results.

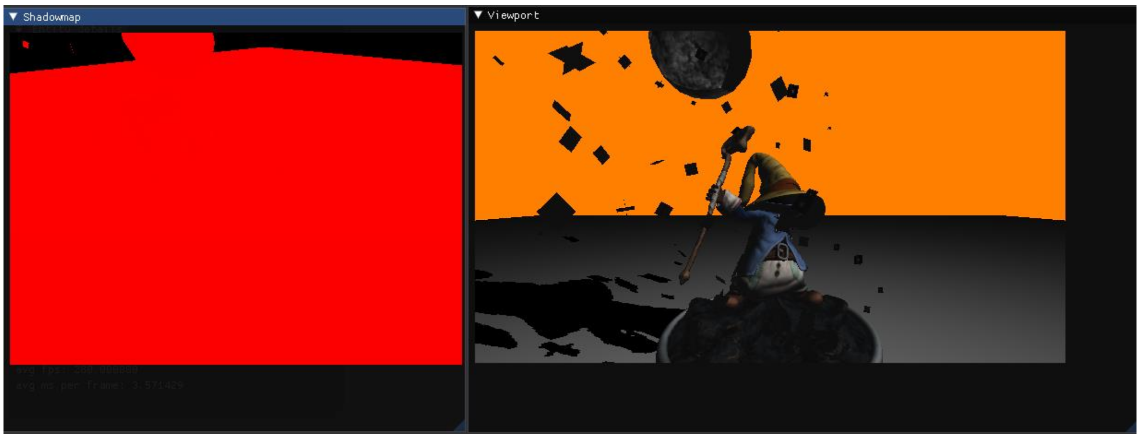
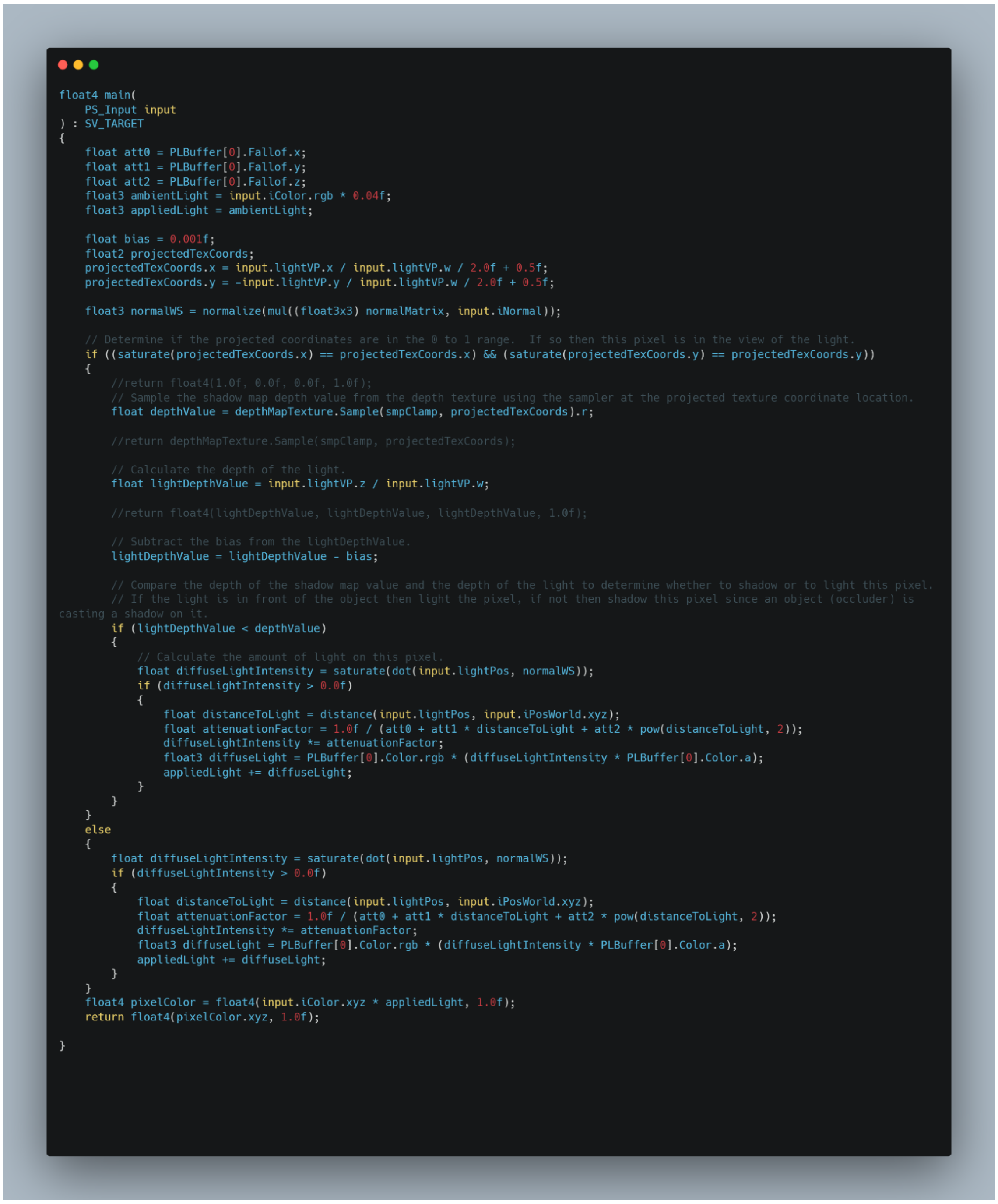


Figure 55: Shadow mapping debugging texture along with shadow mapping reprojection in Tesseract.



```
float4 main(
    PS_Input input
) : SV_TARGET
{
    float att0 = PLBuffer[0].Fallof.x;
    float att1 = PLBuffer[0].Fallof.y;
    float att2 = PLBuffer[0].Fallof.z;
    float3 ambientLight = input.iColor.rgb * 0.04f;
    float3 appliedLight = ambientLight;

    float bias = 0.001f;
    float2 projectedTexCoords;
    projectedTexCoords.x = input.lightVP.x / input.lightVP.w / 2.0f + 0.5f;
    projectedTexCoords.y = -input.lightVP.y / input.lightVP.w / 2.0f + 0.5f;

    float3 normalWS = normalize(mul((float3x3) normalMatrix, input.iNormal));

    // Determine if the projected coordinates are in the 0 to 1 range.  If so then this pixel is in the view of the light.
    if ((saturate(projectedTexCoords.x) == projectedTexCoords.x) && (saturate(projectedTexCoords.y) == projectedTexCoords.y))
    {
        //return float4(1.0f, 0.0f, 0.0f, 1.0f);
        // Sample the shadow map depth value from the depth texture using the sampler at the projected texture coordinate location.
        float depthValue = depthMapTexture.Sample(smpClamp, projectedTexCoords).r;

        //return depthMapTexture.Sample(smpClamp, projectedTexCoords);

        // Calculate the depth of the light.
        float lightDepthValue = input.lightVP.z / input.lightVP.w;

        //return float4(lightDepthValue, lightDepthValue, lightDepthValue, 1.0f);

        // Subtract the bias from the lightDepthValue.
        lightDepthValue = lightDepthValue - bias;

        // Compare the depth of the shadow map value and the depth of the light to determine whether to shadow or to light this pixel.
        // If the light is in front of the object then light the pixel, if not then shadow this pixel since an object (occluder) is
casting a shadow on it.
        if (lightDepthValue < depthValue)
        {
            // Calculate the amount of light on this pixel.
            float diffuseLightIntensity = saturate(dot(input.lightPos, normalWS));
            if (diffuseLightIntensity > 0.0f)
            {
                float distanceToLight = distance(input.lightPos, input.iPosWorld.xyz);
                float attenuationFactor = 1.0f / (att0 + att1 * distanceToLight + att2 * pow(distanceToLight, 2));
                diffuseLightIntensity *= attenuationFactor;
                float3 diffuseLight = PLBuffer[0].Color.rgb * (diffuseLightIntensity * PLBuffer[0].Color.a);
                appliedLight += diffuseLight;
            }
        }
    }
    else
    {
        float diffuseLightIntensity = saturate(dot(input.lightPos, normalWS));
        if (diffuseLightIntensity > 0.0f)
        {
            float distanceToLight = distance(input.lightPos, input.iPosWorld.xyz);
            float attenuationFactor = 1.0f / (att0 + att1 * distanceToLight + att2 * pow(distanceToLight, 2));
            diffuseLightIntensity *= attenuationFactor;
            float3 diffuseLight = PLBuffer[0].Color.rgb * (diffuseLightIntensity * PLBuffer[0].Color.a);
            appliedLight += diffuseLight;
        }
    }
    float4 pixelColor = float4(input.iColor.xyz * appliedLight, 1.0f);
    return float4(pixelColor.xyz, 1.0f);

}
```

Figure 56: Shadow calculation Pixel Shader

6. Economic study

In this chapter we will see the economic impact of the development of this kind of project; we will consider several factors: human, software licenses, salary, hardware, time...

We will start by estimating how much time I invested in the project:

Roughly 200 hours of work from which a 50% would be implementation of the different systems within Tesseract such as the Renderer, UI, profiling..., 15% to iteration over the systems, 15% to testing of these systems and finally 20% to investigation of different techniques.

In the first case, if we were working for a company and were perceiving a salary, by using "Glassdoor", which is a web page that hosts different information regarding jobs such as salary, opinion on various companies, interview processes... and several jobs offers for Rendering Engineers, we would find that the salaries vary between 18k€ and 28k€.

I chose a medium salary of 23k€ for a Junior Rendering Engineer position.

A month salary of 1.916,66€ each month will cost the company around 2.489,74€ each month. The approximate difference, 573€, will be paid by the company to the state for reasons such as social security and other taxes: unemployment (5.5%), common contingencies (23.6%), fogasa (0.2%), formation (0.6%)…, this averages the amount to around 15,56€ per hour which we will use to calculate the costs of development.

Task	Time in hours	Economic cost per hour
Implementation of the different systems within Tesseract	100	1.556 €
Iteration over the systems	30	466,8 €
Testing of the systems	30	466,8 €
Researching different techniques	40	622,4 €
Total	200	3.112 €

We must also consider the cost of each component and material that was used to make this project and amortizing it by estimating the lifespan of each product.

	Cost in €	Project duration	Cost per month	Product lifespan	Total
High-end Desktop PC	1600 €	1,25	22,22€	6 years	27,78€
Laptop	1000 €	1,25	20,83€	4 years	26,04€
Two 2K Monitors	500 €	1,25	13,89€	3 years	17,36€
Misc.	150 €	1,25	2,5€	5 years	3,13€
Total	3250 €				74,31€

No special software license was needed for the development of this project.

Then, the total cost can be calculated as follows:

	Cost in €
Development	3.112 €
Materials used	74,31 €
Total	3.186,31 €

From a business perspective, this project can just be doable by bigger companies which have the funds to hire a bigger team of specialized people who knows its way around the development of this kind of software. Even with those funds available, it is still a dangerous business choice compared to choosing a pre-existing engine such as Unreal Engine, Godot, or Unity. However, it could be a good long run investment if the company does not want to pay extra cuts to the developers of the commercial engine or want to produce a proprietary engine that could license to other companies.

7. Results

The objective of this project was to develop an understanding of the role of a Rendering engineer and game engine developer. As Tesseract has been built from scratch, it has been easy for me to iterate and experiment with different techniques and choose what I think fits best my purposes.

As a Renderer is a very visual piece of software, it has been easy for me to analyze and understand what was working correctly and what not, to pour more time into those fields.

The result of this work is a prototype engine focused on rendering, featuring some of the most important features for real-time/videogame development such as shadows, texturing, proper 3D transformations, Camera…

Finally, Tesseract is a prototype, so it is not at the level of professional engines which have been in development since the early years of game development, however it is a good example of what a single person can accomplish in this period of time, and that small specific engines might be worth investing or investigating in order to expand knowledge.

As a retrospective, Tesseract is capable of rendering 3D Models following certain lighting, shading model, using texture mapping and bump/normal mapping with illumination to provide extra geometric detail and traverse the world to look at them at different angles. Even if this is implemented properly there is much room for improvement, as there are more complex computer techniques that can enhance the look and performance of, and this prototype is still in an early but promising phase.

8. Conclusion and future work

The Tesseract prototype built for this project has been both a research-intensive task and a challenge, as understanding complex techniques and concepts was needed to fulfill the objectives of this project, which were centered around the rendering aspect of it. I am satisfied with the work that has been done until now, even though there is room for improvement and adding new features.

The work I present here successfully satisfies the objectives that were set by taking the first steps into game engine and rendering development, having allowed me to research more into these topics and tackling issues that otherwise I wouldn't have faced.

Tesseract is a prototype, and as such, there is still a lot of fields for improvement and research. There are some things left to do and polish within the implemented systems, as software development is an iterative process. Regarding adding more effects, implementing support for soft shadows and more advanced shading models would be the next step to take. It would be ideal to also start researching and implementing skinning, so an early 3D animation system could start taking form.

Even though the base architecture for Tesseract is set and can be extended further by adding whole new features to it, doing a thorough look into it and polishing the existing systems to do their work better might be needed before start extending it.

9. Bibliography

Data-Oriented vs Object-Oriented Design
https://medium.com/@jonathanmines/data-oriented-vs-object-oriented-design-50ef35a99056

Computer graphics research resources
https://www.scratchapixel.com/

DOOM Eternal – Graphics Study
https://simoncoenen.com/blog/programming/graphics/DoomEternalStudy

Physically Based Rendering, From Theory to Implementation. By Matt Pharr, Wenzel Jakob, and Greg Humphreys.

Fundamentals of Computer Graphics. By Steve Marschner and Peter Shirley.

3D Game Programming with DirectX 11 by Frank D. Luna.

Game Engine Architecture By Jason Gregory.

Foundations of Game Engine Development: Mathematics by Eric Lengyel.

Foundations of Game Engine Development: Rendering by Eric Lengyel.

Real Time Rendering by Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michal Iwanicki and Sébastien Hillaire.

Data-Oriented Design book by Richard Fabian.

RasterTek
http://www.rastertek.com/

Game Engines and Game History
https://www.kinephanos.ca/2014/game-engines-and-game-history/

History and Comparative study of modern game engines
https://www.researchgate.net/publication/259496289_History_and_comparative_study_of_modern_game_engines

Game Engines: Past, Present and Future
https://www.gamedeveloper.com/programming/game-engines-past-present-and-future

History of 3D Rendering
https://pixelperfect-studios.com/history-of-3d-rendering/

Timeline of Computer History
https://www.computerhistory.org/timeline/graphics-games/

Percentage-Closer Soft Shadows by Randima Fernando (NVIDIA Corp.)

10. Annexes

10.1. Project Proposal

0. Note:

The project proposal has been modified due to the topics on basic 3D real-time rendering effects being more attractive towards the development of a small game engine renderer prototype rather than a voxel engine.

1. Project's Title

Tesseract: A 3D Rendering Engine

2. Description and Justification of the Topic

The Project consists in the investigation e implementation of a game engine centered in 3D Rendering. Using a modern graphics API such as DirectX 11 and technologies like ImGUI for an immediate mode UI. The language to develop such a task will be C/C++ following a data transformation philosophy to make an efficient memory management and produce a good performance result.

3. Project's Objective

a. Getting to know the state of the art in concepts such as Videogame Engine Architecture, 3D Rendering techniques and all that is derived from but in relation with the proposed project.
b. Development of a prototype engine focused on the 3D Rendering aspect, using a modern hardware graphics API such as DirectX 11 and ImGUI as an Immediate mode UI Library.
c. Design of a good architecture that allows for adding new features without taking a huge performance hit.
d. Iteration and polishing of different rendering techniques and algorithms used in the engine to obtain a good result with visual quality.

4. Methodology

The methodology will be defined with the tutor in the reunions of the project.

5. Task Planification

The planning of the different tasks will be defined with the tutor in the reunions of the project.

6. Additional Observations

The tutor of the project will be Eduardo Jiménez Chapresto.

10.2. Meetings

Date	26-11-2021	Reunion	Online through Discord.		
Nº	1	Hour	10:00 AM	Duration	20 minutes.
1	Quick review on the final project.				
2	Mention of the most interesting topics regarding the final project: Rendering and effects.				
3	Brief talk about Rendering and effects and writing methodologies.				
4	Greenlight from tutor to start researching and implementing base systems.				

Date	09-12-2021	Reunion	Online through Discord.		
Nº	2	Hour	10:00 AM	Duration	24 minutes.
1	Brief talk about the work.				
2	Extended talk about advancements in Tesseract.				
3	Showed different papers that I researched on Game Engines.				
4	Set what would be nice to have working first, such as Vertex Transformations.				

Date	26-11-2022	Reunion	Online through Discord.		
Nº	3	Hour	10:30 AM	Duration	30 minutes.
1	Brief talk about the project.				
2	Showing of some functionality from the project.				
3	Quick look at code.				
4	Talk about how to improve and continue the research and implementation.				

Date	26-11-2022	Reunion	Online through Discord.		
Nº	4	Hour	11:00 AM	Duration	25 minutes.
1	Quick talk about the project's state.				
2	Comment on the chosen architecture for the project.				
3	Talk about the work that was done and the work that wasn't yet done.				
4	Set of the following steps to follow next.				

Date	08-08-2022	Reunion	Online through Discord.		
Nº	5	Hour	18:00 PM	Duration	25 minutes.
1	Quick chat about how Tesseract was going.				
2	Showing of different images from the project.				
3	Comment on what I had been implementing and readiness for showing.				
4	Comment on some technical related topics such as Shading models, lights, texturing...				
5	Learning and researching Bibliography sharing.				
6	Talk about this document review.				