Universitat Politècnica
de València

Departamento de Sistemas Informáticos y Computación

Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Trabajo Fin de Máster

Comparison of Ray Tracing GPU Implementations

Autor(a): Luis Carlos Catalá Martínez
Director(a): Francisco José Abad Cerdá

Valencia, Septiembre - 2022

Este Trabajo Fin de Máster se ha depositado en el Departamento de Sistemas Informàticos y Computación de la Universitat Politècnica de València para su defensa.

Trabajo Fin de Máster
Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Título: Comparison of Ray Tracing GPU Implementations

Septiembre - 2022

Autor(a): Luis Carlos Catalá Martínez
Director(a): Francisco José Abad Cerdá
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Resum

El Traçat de Raigs és una tècnica elegant i relativament moderna per dibuixar gràfics en 3D. Tot i que l'algorisme subjacent existeix des de fa dècades, no s'ha pogut fer servir de manera prevalent en indústries com els videojocs o el cinema fins fa pocs anys, gràcies a ser accelerat mitjançant l'ús de GPUs. Actualment hi ha múltiples tecnologies que s'encarreguen de gran part d'aquesta acceleració per nosaltres. Cadascuna té una audiència objectiu específica, ja sigui gràfics en temps real o més qualitat d'imatge. Dues de les llibreries més importants en aquests dominis són Vulkan i Nvidia OptiX, respectivament.

En aquest treball explorarem el desenvolupament d'un traçador de raigs emprant cadascuna d'aquestes i analitzarem el seu rendiment en multitud d'ajustaments gràfics, escenes i configuracions de maquinari. Les mètriques en què ens centrarem són el temps de construcció de lestructura dacceleració, temps de dibuixat dun fotograma i ús de memòria gràfica.

Resumen

El Trazado de Rayos es una técnica elegante y relativamente moderna para dibujar gráficos en 3D. Aunque el algoritmo subyacente existe desde hace décadas, no ha podido usarse de forma prevalente en industrias como los videojuegos o el cine hasta hace pocos años, gracias a ser acelerado mediante el uso de GPUs. Actualmente existen múltiples tecnologías que se encargan de gran parte de dicha aceleración por nosotros. Cada una tiene una audiencia objetivo específica, ya sea gráficos en tiempo real o mayor calidad de imagen. Dos de las librerías más importantes en dichos dominios son Vulkan y Nvidia OptiX, respectivamente.

En este trabajo exploraremos el desarrollo de un trazador de rayos empleando cada una de éstas y analizaremos su rendimiento en multitud de ajustes gráficos, escenas y configuraciones de hardware. Las métricas en que nos centraremos son el tiempo de construcción de la estructura de aceleración, tiempo de dibujado de un fotograma y uso de memoria gráfica.

Abstract

Ray Tracing is an ellegant and relatively modern technique to draw 3D graphics. Even though the underlying algorithm has existed for decades now, it's only been in recent years that it has taken a prevalent role in industries like videogames or cinema, due to it being accelerated through the use of GPUs. Nowadays there are multiple technologies that handle most of this acceleration for us. Each with a specific target audience, be it real time graphics or aiming for a higher image quality. Two of the most prevalent libraries for each domain are Vulkan and Nvidia OptiX, respectively.

In this work we will be exploring the development of a ray tracer with each of these and analyzing their performance across a slew of graphical settings, scenes and hardware configurations. The metrics we will be focusing on will be acceleration structure build time, frame rendering time and graphics memory usage.

Índice general

Chapter 1

Introduction

Ray Tracing is one of the most advanced, elegant and precise ways to produce computer graphics images. Although the original idea has existed since as early as the 16th century, it's first computerized version didn't appear until the year 1968. During the last decade it has become increasingly popular thanks to GPU acceleration, and many libraries have emerged to aid in the development of applications that use this technique.

Each library better suits a different purpose for the final piece of software, be it real time graphics for videogames or other real-time applications, or pursuing a higher image quality for visual effects in movies, animations, complex scientific visualizations, etc.

This work aims to compare the most prevalent technologies used to build Ray Tracing applications for each purpose, Vulkan and OptiX, in order to benchmark their behaviour under different circumstances. We will implement a series of test cases which will vary the ammount of rendered and loaded geometry as well as image quality settings, and record the following data from each one:

* Acceleration Structure Building Time
* Frame Render Time
* Video Memory Consumption

These metrics will not only be analyzed in detail running in a single system, but also compared running across a slew of hardware we could get our hands on. This will allow us to isolate how different components like CPU or GPU affect them, as well as comparing bare metal to virtualized performance.

Chapter 2

State of the Art

In this chapter we will explain the current state of computer graphics its latest advances and how ray tracing fits into it. We will also be looking at the technologies being used in both research and industry.

2.1. Rendering Techniques

The field of computer graphics seeks to generate images with the aid of computers. Nowadays this is a core component of photography, film production and videogames among others. This task is commonly refered to as *rendering*. Throughout the years there have appeared many rendering techniques, of which the main ones being used currently are Rasterization and Ray Tracing.

2.1.1. Rasterization

Rasterization is the process of taking an image described as vector graphics and converting it into a series of pixels which, when displayed together, recreate the image that was represented with vector shapes. This image can then be displayed in a monitor, stored as a bit map and so on.

In figure 2.1, from (*3D Basic Rendering*, 2022) we see the principle of this process. After transforming the geometry (in this case a triangle) to screen space, we check if each pixel in the image overlaps said geometry.

When compared to other techniques for rendering 3D models, such as ray tracing, rasterization is extremely fast. This gives it a prevalent usage in real time 3D engines. We must take into account that the process of rasterization is only responsible for mapping the scene geometry to pixels, and does not compute the color of such pixels. This color is assigned by a Pixel (or Fragment) Shader, which is completely programmable in modern GPUs. This shader may take into account physical processes like light position, or a purely artistic approach. There is no motivation in modifying the rasterization techniques at render time. Therefore, the rest of the process of rasterizing a 3D model into screen space (a 2D plane for displaying said graphics) is often performed by non-programmable hardware with a fixed function within the graphics pipeline. This method allows for high efficiency.

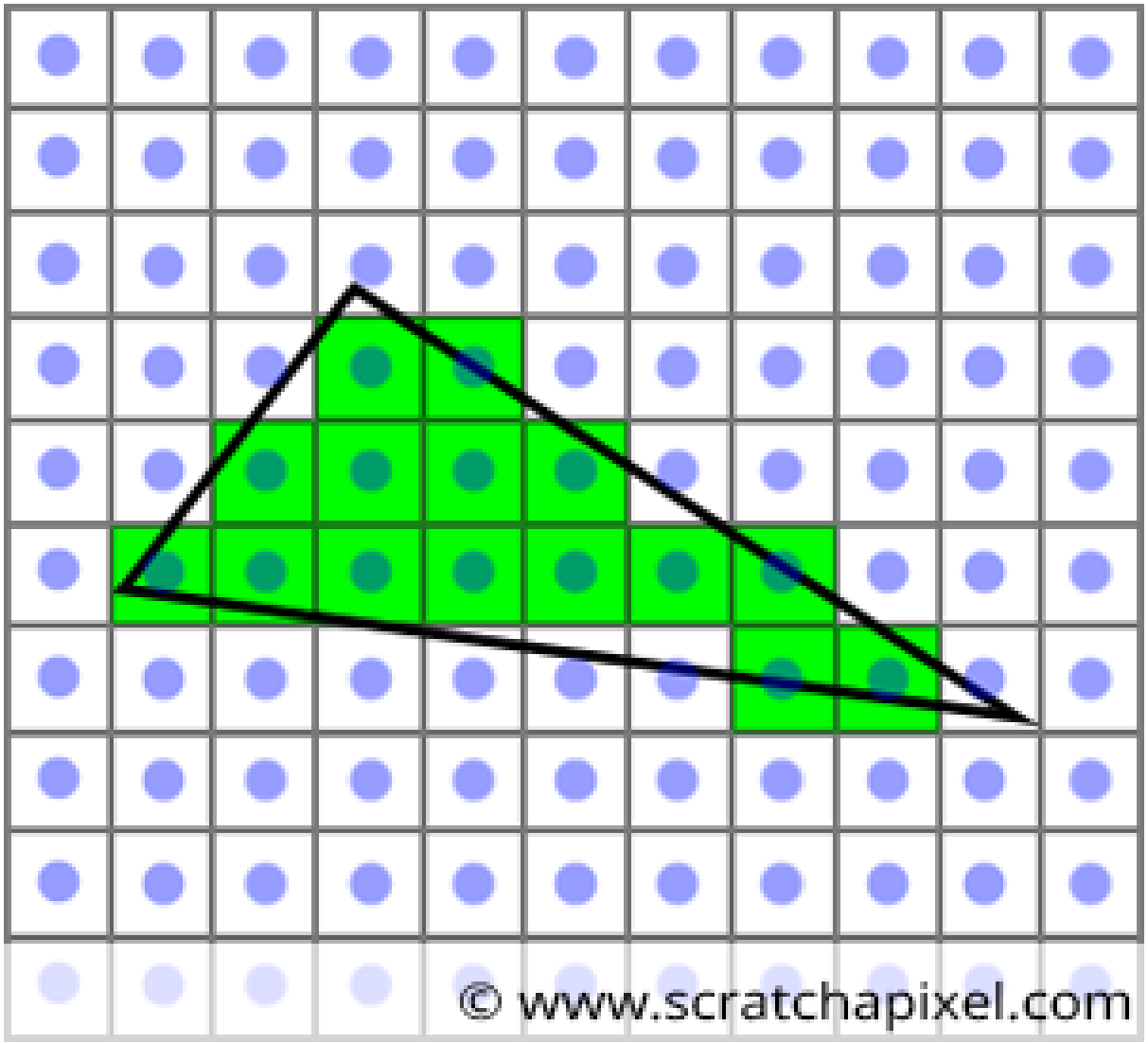


Figura 2.1:

2.1.2. Ray Tracing

The other prevalent technique for drawing 3D graphics, and the main focus of this work, is ray tracing.

This technique models light transport for generating digital images. It does this tracing a path from an imaginary eye through each pixel in a virtual screen and calculating the color of the object visible through it. Each ray is tested for intersection with some subset of the objects in the scene. Once identified the object, it estimates the incoming light at the intersection point and read the material properties of the object to calculate the final color of the pixel.

Although counter intuitive, sending rays *from* the camera towards the scene is many orders of magnitude more efficient than doing it the other way around. This is due to the vast majority of rays coming from a light source not reaching the viewer's eye, thus saving a lot of computation in paths that are never recorded. We take the shortcut of assuming that a given ray intersects the view frame. After a certain number

of reflections or distance traveled by a ray without intersecting anything, that ray ceases to travel and the pixel's value is updated. Figure 2.2 shows an overview of this algorithm extracted from the Wikimedia (*File:Ray trace diagram.svg*, 2022).

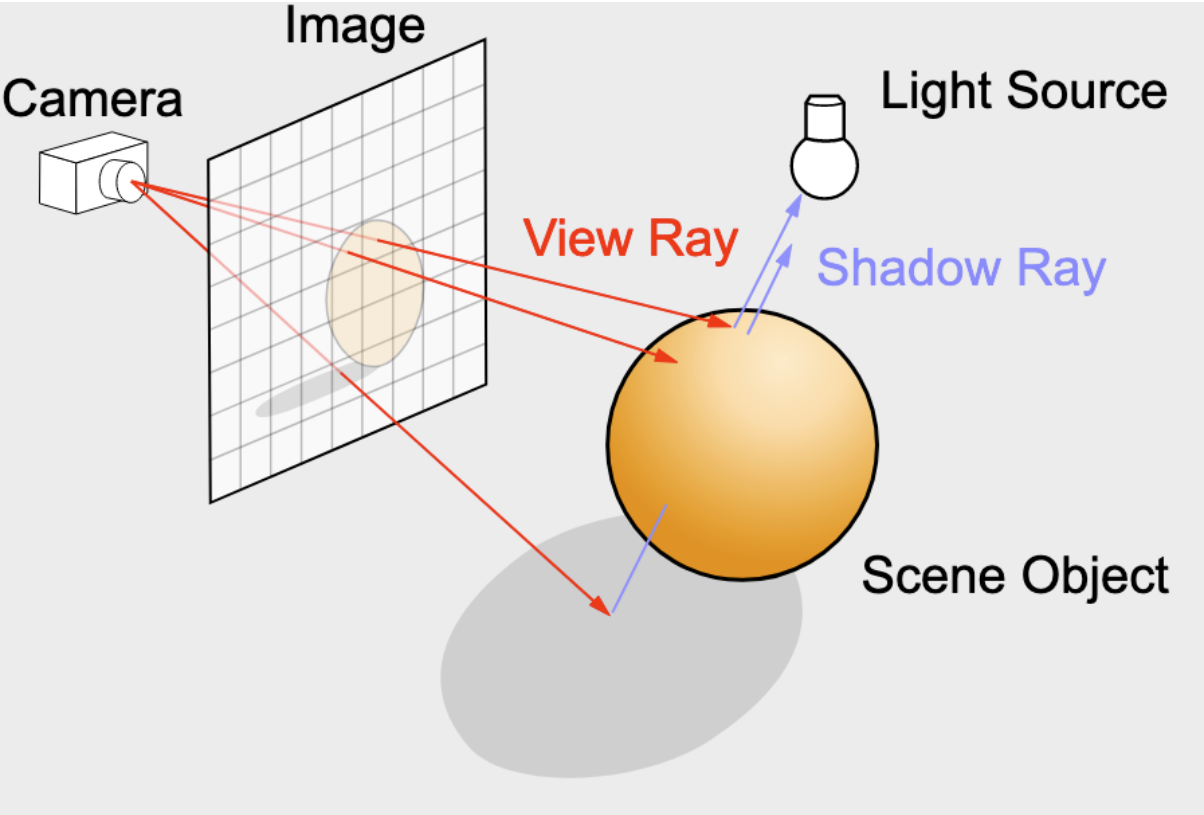


Figura 2.2: Overview of the ray tracing algorithm with spheres as our only geometry and a single light source.

Compared to rasterizing, all the techniques based in ray tracing are generally slower yet provide higher fidelity results than it's counterpart. This made it so that originally it was applied in tasks that could tolerate a relatively long render time, such as film production or still image generation. Applications where rendering speed is critical, like videogames, were less suited for these algorithms. However, since 2018, real time ray tracing has become more feasible thanks to hardware acceleration becoming standard in commercial graphics cards.

2.1.2.1. Mathematical foundation

We will now look at the mathematical definition of ray tracing applied to a rectangular viewport, which is the focus area of this work.

Our inputs are:

* An eye position $E \in \mathbb{R}^3$
* A target position $T \in \mathbb{R}^3$
* A field of view $\theta \in [0, \pi]$. For humans, we can assume it's about 90 degrees ($\approx \frac{\pi}{2} radians$)

* The number of square pixels in the vertical and horizontal directions in the viewport $m, k \in \mathbb{N}$

* The actual number of pixels $i, j \in \mathbb{N}, 1 \leq i \leq k \wedge 1 \leq j \leq m$

* A vertical vector that indicates the up and down direction $\overrightarrow{v} \in \mathbb{R}^3$. Usually $\overrightarrow{v} = [0, 1, 0]$ (the roll component determines the viewport's rotation around it's center C, where the axis is the ET rotation).

We can see an illustration of these components in figure 2.3, extracted from the Ray Tracing Wikipedia article (*Ray Tracing (graphics)*, 2022).

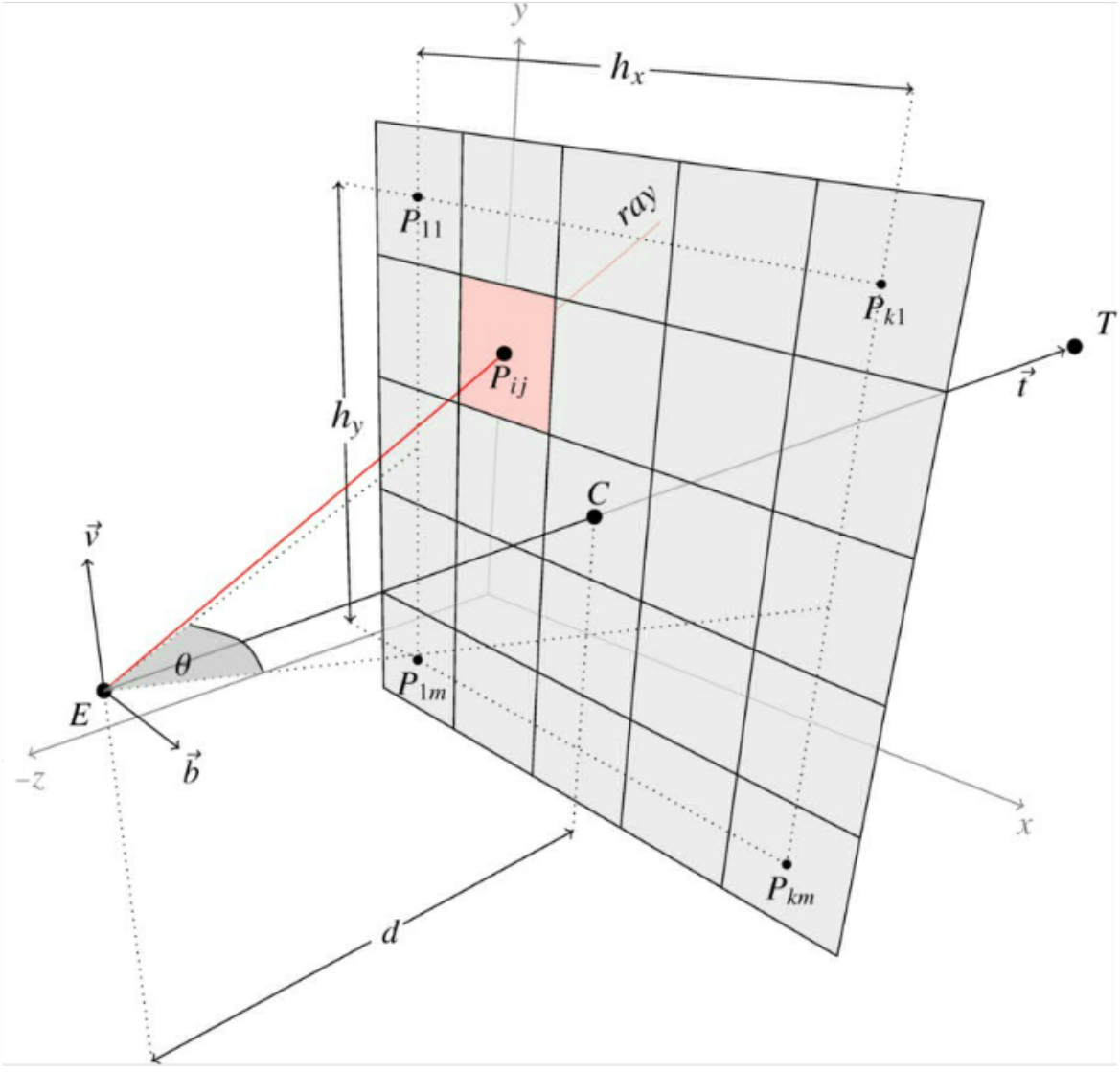


Figura 2.3: Ray tracing schema with all the input components.

With these inputs, our goal is to find the position of the center of each viewport pixel P_{ij}. This will allow us to find the line going from the eye E through that pixel, and describe that ray by the point E and the vector $\overrightarrow{R}_{ij} = P_{ij} - E$ (or its normalization $\overrightarrow{r}_{ij}$).

We start by finding the coordinates of the bottom left viewport pixel P_{1m}, and the subsequent ones by shifting along the directions paralell to the viewport (vectors $\overrightarrow{b}_n$ and $\overrightarrow{v}_n$), multiplied by the pixel size. The equations below include a distance d between the eye E and the viewport. This value will be reduced when normalizing the rays $\overrightarrow{r}_{ij}$, and as such can be interpreted as $d = 1$ and removed from the calculations.

As a pre-calculation we normalize the vectors $\overrightarrow{t}$, $\overrightarrow{b}$ and $\overrightarrow{v}$, shown in figure 2.3, which are paralell to the viewport. This process is explained in equations 2.1, 2.2 and 2.3.

$$\overrightarrow{t} = T - E, \overrightarrow{b} = \overrightarrow{t} \times \overrightarrow{v} \tag{2.1}$$

$$\overrightarrow{t}_n = \frac{\overrightarrow{t}}{\left\|\overrightarrow{t}\right\|}, \overrightarrow{b}_n = \frac{\overrightarrow{b}}{\left\|\overrightarrow{b}\right\|} \tag{2.2}$$

$$\overrightarrow{v}_n = \overrightarrow{t}_n \times \overrightarrow{b}_n \tag{2.3}$$

Note that the viewport center is $C = E + \overrightarrow{t}_n d$.

We then calculate the viewport sizes h_x and h_y, divided by 2 including the inverse aspect ratio $\frac{m-1}{k-1}$. This is done in equation 2.4.

$$g_x = \frac{h_x}{2} = d * tan\frac{\theta}{2}, g_y = \frac{h_y}{2} = g_x\frac{m-1}{k-1} \tag{2.4}$$

Next, we calculate the vectors used for shifting to the next pixel, q_x and q_y, along the directions paralell to the viewport $\overrightarrow{b}$ and $\overrightarrow{v}$, from the bottom left pixel p_{1m}. This is shown in equations 2.5 and 2.6.

$$\overrightarrow{q}_x = \frac{2g_x}{k-1}\overrightarrow{b}_n, \overrightarrow{q}_y = \frac{2g_y}{m-1}\overrightarrow{v}_n \tag{2.5}$$

$$\overrightarrow{p}_{1m} = \overrightarrow{t}_n d - g_x\overrightarrow{b}_n - g_y\overrightarrow{v}_n \tag{2.6}$$

If we consider $P_{ij} = E + \overrightarrow{p}_{ij}$ and the ray $\overrightarrow{R}_{ij} = P_{ij} - E = \overrightarrow{p}_{ij}$, we get the normalized rays in equation 2.7.

$$\overrightarrow{p}_{ij} = \overrightarrow{p}_{1m} + \overrightarrow{q}_x(i-1) + \overrightarrow{q}_y(j-1), \overrightarrow{r}_{ij} = \frac{\overrightarrow{R}_{ij}}{\left\|\overrightarrow{R}_{ij}\right\|} = \frac{\overrightarrow{p}_{ij}}{\|\overrightarrow{p}_{ij}\|} \tag{2.7}$$

2.2. Ray Tracing Technologies

As previously said, though the ray tracing technique has existed for several decades now, it is only in recent years that GPU acceleration has made it usable for real time applications. This acceleration comes from one of several libraries or rendering APIs that offer the programmer a quick and easy way to interact with the graphics card.

It's also worth mentioning that we could technically develop a GPU-accelerated ray tracer with any rendering library from the last two decades, since anything that gives us control over the GPU could be used for parallelizing the required operations and give us a considerable speed increase. An example of this would be the Compute Shaders in OpenGL, which could process an image that we could then draw to a texture in screen made of two triangles. However, we will only be looking at libraries that include functionality exclusively dedicated to ray tracing.

Some of these libraries have legacies that extend to the era of rasterized-only graphics, while others were made from the ground up just for ray tracing. At the same time, each of them was designed with a specific use case in mind, be it real time graphics or achieving a higher image fidelity. We will now be looking at what we considered the most prominent library for each use case,

2.2.1. Vulkan

First announced by the Khronos Group at GDC 2015 as the "next generation OpenGL", Vulkan is intended to be used in high-performance real-time 3D graphics for interactive applications such as videogames. It offers higher performance and lower level control than popular graphics APIs at the time such as OpenGL or DirectX 11.

It is intended to provide a series of advantages over it's predecessor, OpenGL, as well as other APIs. Mainly, Vulkan offers lower overhead, more direct control over the GPU and lower CPU usage. It was originally developed from AMD's Mantle (*Mantle, Wikipedia*, 2022), taking many features and concepts from it. These were later adopted by other APIs, such as DirectX12 and Metal.

Some of it's advantages compared to other APIs at the time were:

* Provides a unified API for desktop and mobile devices, in contrast with OpenGL, which had two different APIs for each (OpenGL and OpenGL ES).

* Vulkan is multiplatform and available in multiple modern operating systems, not being locked to any operating system or device form factor. In this sense it's more similar to OpenGL than to DirectX 12. It currently runs on Android, Linux, BSD Unix, QNX, Nintendo Switch, Raspberry Pi, Stadia, Fuchsia, Tizen, Windows 7, 8, 10 and 11, iOS, tvOS and macOS. It is worth noting that the Apple platforms require the usage of MoltenVK, a library that converts Vulkan code to Metal.

* Lower CPU usage through the use of low level optimizations such as batching operations together. This leaves the CPU free for longer periods of time, allowing it to do more work in the meantime.

* Multi-threading friendly design, in contrast with OpenGL 4 and DirectX 11. These were developed with single-core CPUs in mind and had to be expanded later on to be executable in multiple cores. Thanks to this, Vulkan offers better scalability on multi-core CPUs.

* Pre-compiled shaders. Shaders are programs executed in the GPU, and as such they need to be compiled. The prevalent method for doing this was that each rendering API supported a specific language (OpenGL with GLSL, DirectX with HLSL, etc.) and came with it's own compiler for it. This compiler was executed

at application runtime to translate from a shading language into machine code readable by the GPU. In contrast, the Vulkan driver expects code already compiled into an intermediate language known as SPIR-V (Standard Portable Intermediate Representation). This pre-compilation reduces the application's initialization time and allows for a larger variety of shaders to be used per scene. Now the driver only needs to optimize the intermediate code, and developers have an easier way of obfuscating propietary shaders, since this is no longer stored as souce code.

* Vulkan employs an extension system in which the programmer must explicitly ask for the functionality they require, therefore reducing unnecesary overhead and code size. It is through this system that it provides ray tracing support, by means of the *VK_KHR_ray_tracing* extension.

Due to it's low overhead and customization capabilities Vulkan is widely used in the videogame industry. We see almost every major game engine (Unity, Unreal Engine, CryEngine, Frostbite, Godot, etc.).

2.2.2. OptiX

Released in 2009 as part of Nvidia GameWorks, OptiX is a ray tracing API that offloads computations to the GPU through CUDA. This means it's only available for Nvidia's graphics products. While in previous versions it was considered a high level API, since version 7 it has become much lower level, giving an extensive control to the programmer on how memory and processes are managed. Despite of this, it's still at a higher level than most other rendering APIs, being designed to encapsulate the entire algorithm of which ray tracing is a part, not just the ray tracing itself. This allows for the engine to execute the broader algorithm with greater flexibility and without changes on the application side. Aside from rendering, OptiX is also used in areas where the tracing of rays or GPU acceleration is useful. This includes optical and accoustic design, radiation and electromagnetic research, artificial intelligence queries and collision analysis.

OptiX works using CUDA kernels, instructions supplied by the user, that indicate how a ray should behave in a specific situation to simulate a complete tracing process. A ray might have different behaviours when hitting different surfaces. OptiX allows us to customize these behaviours with kernels, written in CUDA's own flavour of C or PTX code, that are linked together when used by the engine.

The usage of an OptiX ray tracer usually involes the following steps:

1. Define programs for:
    * Ray Generation. Wether rays can be shot in paralell, in a perspective fashion, like a gradient field, etc.
    * Ray Missing. What to do when a ray doesn't interact with any object.
    * (Optional) Exception Program. What to do when a ray cannot be shot for some reason.
    * Bounding Box. This provides a bounding box intersection test for a given object.

    * Intersection Program. How a ray behaves when intersecting with geometry.

2. Define material programs for *anyhit* and *closesthit*. These determine the ray behaviour upon it's first intersection (closest hit) or subsequent intersections (any hit).
3. Define buffers. These are memory regions that allow the host and device (CPU and GPU) codes to communicate.
4. Define scene geometry hierarchy. This generates a tree graph of the scene to be rendered, including objects, groups and selectors among others.

OptiX is also capable of scaling transparently across multiple GPUs. This feature, along with it's higher overhead compared to Vulkan, make it so it's more used for higher image fidelity applications than the Khronos library. If we look at the list of companies that employ it, we find names like Autodesk, Pixar or Redshift.

Chapter 3

Analysis

In this chapter we will explain the problem and objectives we're tackling with this work, as well as our requirements and experiments performed.

3.1. Problem and objectives

As discussed in the State of the Art chapter, there are many ways to accelerate a ray tracing application, and not all of them work the same way. When building a ray tracer, it's important to know the differences between the technologies behind this acceleration in order to pick the one that best suits our requirements.

As explained in the Introduction, we have nailed down the two most relevant options to Vulkan and OptiX. Our objective is to analyze how each of them behave when varying the rendering quality and scene complexity.

For our objectives to be met and our experiments to be performed we will need the following:

* Ray Tracer capable of rendering with the Vulkan API and its ray tracing extension and Ray Tracer capable of rendering with the OptiX library, both capable of drawing images as similar as possible. Both renderers may be the same application with some mechanism to switch APIs, but they could also be separate projects with equivalent features. They will also need to be parametrized, so our automation tools can run our experiments from the command line.
* Set of experiments to run on both renderers. These must test different ammounts of scene geometry and image quality settings, like screen resolution.
* Automation system to run our set of experiments in a consistent and replicable way.
* Result visualization system, to plot the results from our experiments.

3.2. Experiment Design

The first step before starting to record any data or implementing any renderers was to decide what kind of experiments we wanted to run. After reading recent publications on the topic of ray and path tracing, we settled for the same measurements done by

Nvidia at the last GTC, when they introduced some advances in real-time path tracing (Clarberg y cols., 2022). These are frame time, acceleration structure building time, and memory usage.

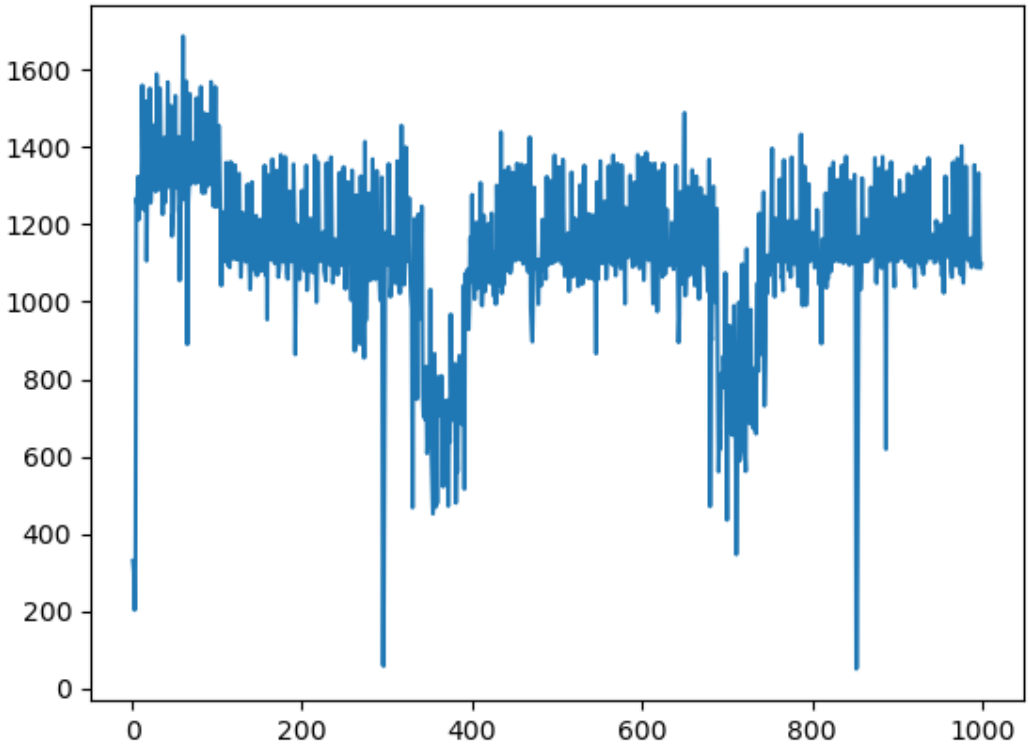


Figura 3.1: Render times over 1000 frames with a frame buffer resolution of 7680x4320, while drawing the model of a viking room with textures. It can be observed how in a very short span of time, the rendering time of the exact same geometry and effects can hugely change.

* **Acceleration Structure Building Time:** as the name suggests, this is a spatial data structure that speeds up the search for triangles, distance fields and other geometry primitives in a given scene. Ray tracing applications use these to achieve better performance. For example, it's much faster to look for ray intersections by traversing a hierarchical structure of Axis Aligned Bounding Boxes than to check a ray against every triangle in a scene. These structures are typically comprised of a single Top Level Acceleration Structure (TLAS) containing multiple Bottom Level Acceleration Structures (BLAS), which encode a single 3D model each with a 3x4 transformation matrix. These structures are usually implemented in hardware. The time it takes for them to be built can be relevant on the general application's startup time for a static scene such as the ones we will be testing, and for runtime performance in case of scenes with dynamic geometry, where this structure will need to be rebuilt for every animation step.

* **Frame Time:** videogames and the media that traditionally covers them have en-

graved in the general public the notion that Frames Per Second (FPS) is the most important performance metric in an interactive graphical application. While this is probably true for the end user experience, the FPS count can be affected by a miriad of things outside of the rendering system, such as GPU-CPU synchronization, physics simulation, etc. As this work aims to compare only the rendering performance of each library, we will only be measuring the time it takes to write the rendered images to a frame buffer. Due to the huge variance of running times inside a modern Operating System, the render times can vary wildly as well (as we can see in 3.1). To mitigate this, we will take the time measurements over a fairly long period of time (1000 frames) and average them together in order to get a more precise idea of the rendering time that's less subject to flukes.

* **Memory Usage:** quite self-explanatory, the last factor we consider of great importance is the video memory consumption of the process. This can be a limiting factor on the hardware requirements of a particular application, since exceeding the user's GPU's memory capabilities will force it to resort to the use of swap memory from the system RAM, significantly impacting the performance due to an abundance of copying between the two.

All these variables will be tested while rendering different ammounts of geometry and scene complexity. In order to simplify the development of all renderers, each scene will be comprised of a single 3D model. The Single Triangle model was built in-house, the Viking Room was obtained from (Overvoorde, s.f.), and the rest were obtained from Morgan McGuire's Computer Graphics Archive (McGuire, 2022). They all can be seen at table 3.2 and in figures 3.2, 3.3 and 3.4.

Model name	Triangles	Vertices	Texture size
Single Triangle	1	3	0
Viking Room	2000	2600	0
BMW	385079	249772	0
Sponza	262267	184330	74.0 MB
Hairball	2880000	1441098	0

Figura 3.2: BMW 3D model used during our experiments

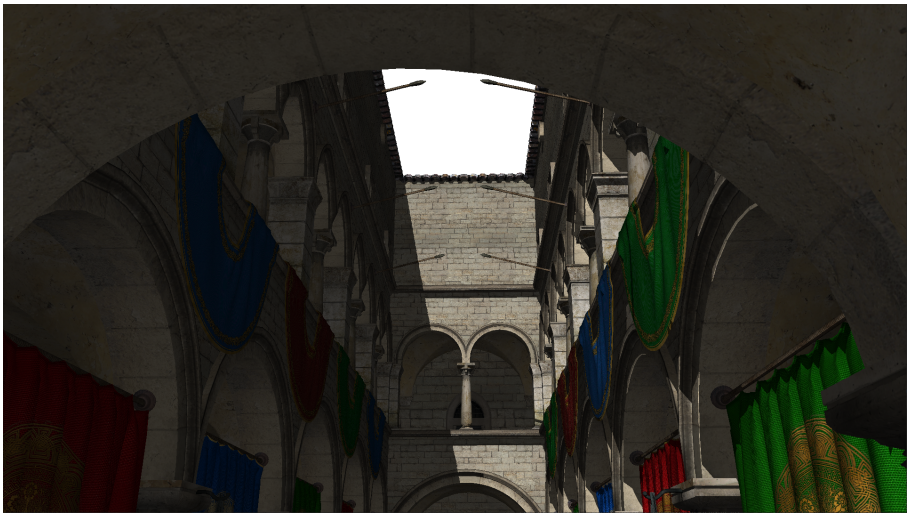


Figura 3.3: Sponza 3D model used during our experiments

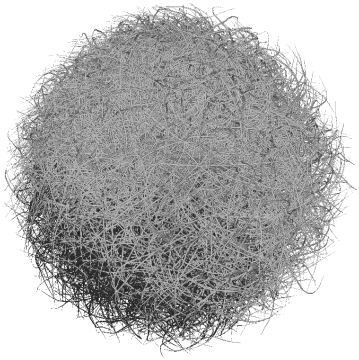


Figura 3.4: Hairball3D model used during our experiments

In order to better gauge how these scene sizes compare, we have plotted them in graph 3.5.

After testing these models we countered some discrepancies with the Acceleration Structure Build Times we expected for each of them. This can be seen in more detail in the Results chapter. We added some extra models for further testing. Their features can be seen in table 3.2 and their triangle counts is graphically compared in plot 3.6

Model name	Triangles	Vertices	Texture size
Human	25422	29708	0
ISCV2	383511	193212	0
Gallery	998831	499314	0

Finally, we considered it will be of use compare how each library performed in different hardware configurations. We gathered all the available hardware compatible

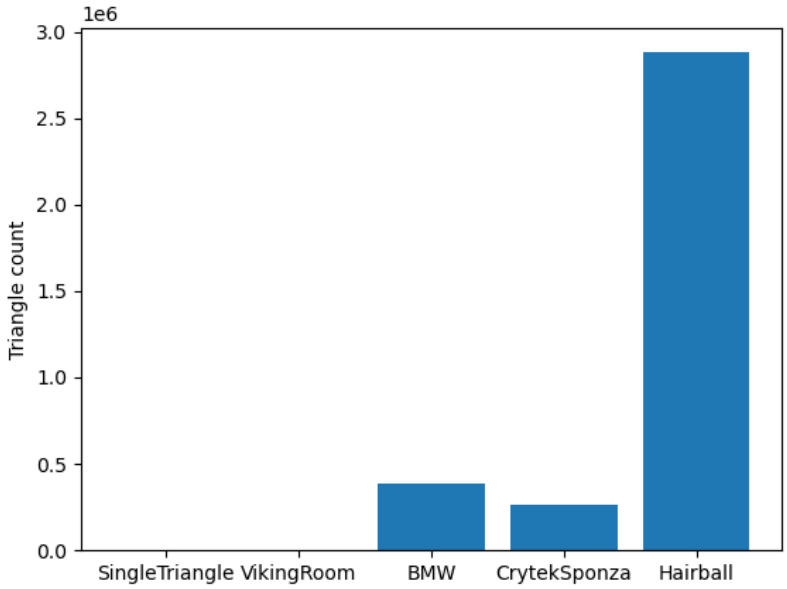


Figura 3.5: Triangle count of each scene used to test both renderers. We can see how there are orders of magnitude of difference every two of them.

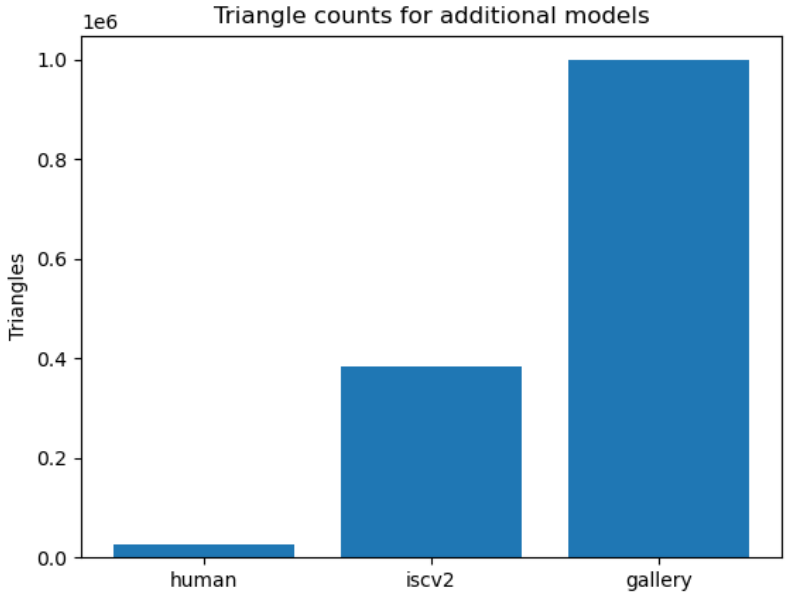


Figura 3.6: Triangle count of the extra scenes used to further test acceleration structure build times.

with GPU accelerated ray tracing, and used it to run the same experiments. This hardware is summarized in table 3.2. This will not only allow us to check how different GPUs handle the same workload, but also how they do it inside a virtualized environment: the Intel Core i9 machine is not running the software in bare metal,

but rather virtualizing Windows under *QEMU*.

CPU	GPU	RAM	OS
Intel Core i7-12700K, 3.60 GHz	Nvidia GeForce RTX 3070	16 GB	Windows 11, 64 bit
AMD Ryzen 5 3600 3.6GHz BOX	Nvidia GeForce RTX 2070 Super	16 GB	Windows 10, 64 bit
Intel Core i5-9600K, 3.7GHz	Nvidia GeForce RTX 3060 Ti	16 GB	Windows 10, 64 bit
Intel Core i9-9900K (12/16 cores), 3.6GHz	Nvidia GeForce RTX 3080	16 GB	Windows 10, 64 bit
Intel Core i7 10th gen	Nvidia GeForce RTX 3080	32GB	Windows 10, 64 bit
AMD Ryzen 7 3800x, 3.9GHz	Nvidia GeForce RTX 2070 Super	64GB	Windows 11, 64 bit
Intel Core i9-12900F	Nvidia GeForce RTX 2060	32GB	Windows 10, 64 bit

Chapter 4

Design

In this chapter we will explain the applications we developed for this work at a high level. We will skip the experiment automation and plot generation parts of the software, since they are either scripts with a couple of loops that run all the configurations we deemed necessary or simply read data from text files and plot them. Instead, we will be focusing on the two ray tracers we built for this work.

4.1. Vulkan Ray Tracer

4.1.1. Rasterized

The rasterized version of the renderer is quite simple in its design, with a single monolithic Application class handling almost everything and a few helper data structures (Vertex, QueueFamilyIndex, SwapchainSupportDetails and UniformBufferObject) for storing and grouping together information. We see an UML class diagram of this application in figure 4.1. This instance of the renderer is much bigger than the following ones due to us not relying on almost any external library and having to handle all the low level operations ourselves, thus resulting in a high ammount of initialization functionality. To all this we added one more class that encapsulates all the performance measuring functionality, which we called FramePerformanceCounter.

4.1.2. Ray Traced

For the ray traced version of the Vulkan renderer we refactored and simplified most of the Application class To achieve this, we left an important part of both the initialization and memory cleanup to the Nvvk library. The UML class diagram in figure 4.2 shows the resulting hierarchy. Finally, we included our own FramePerformanceCounter class for measuring performance.

Figura 4.1: UML class diagram for the rasterized Vulkan renderer.

Figura 4.2: UML class diagram for the raytraced Vulkan renderer.

4.2. OptiX Ray Tracer

For this ray tracer we further encapsulated different functionalities to improve code reusability. This meant increasing the complexity of the class hierarchy. The full diagram can be seen in figure 4.3.

Here we see a central Renderer class thaat serves much as the Application class from the Vulkan renderer. However, we have encapsulated the window functionality in it's own class hierarchy (GLFWWindow, GLFWCameraWindow). Also, we took the camera manipulation system from the 2019 SIGGRAPH OptiX course (Wald, 2022). This system is highly customizable and perfectly functional, though it adds a fair ammount of complexity to the diagram. This last part includes the classes CameraFrame, CameraFrameManip and it's inheritants. To the class GLFWWindow we also added our FramePerformanceCounter.

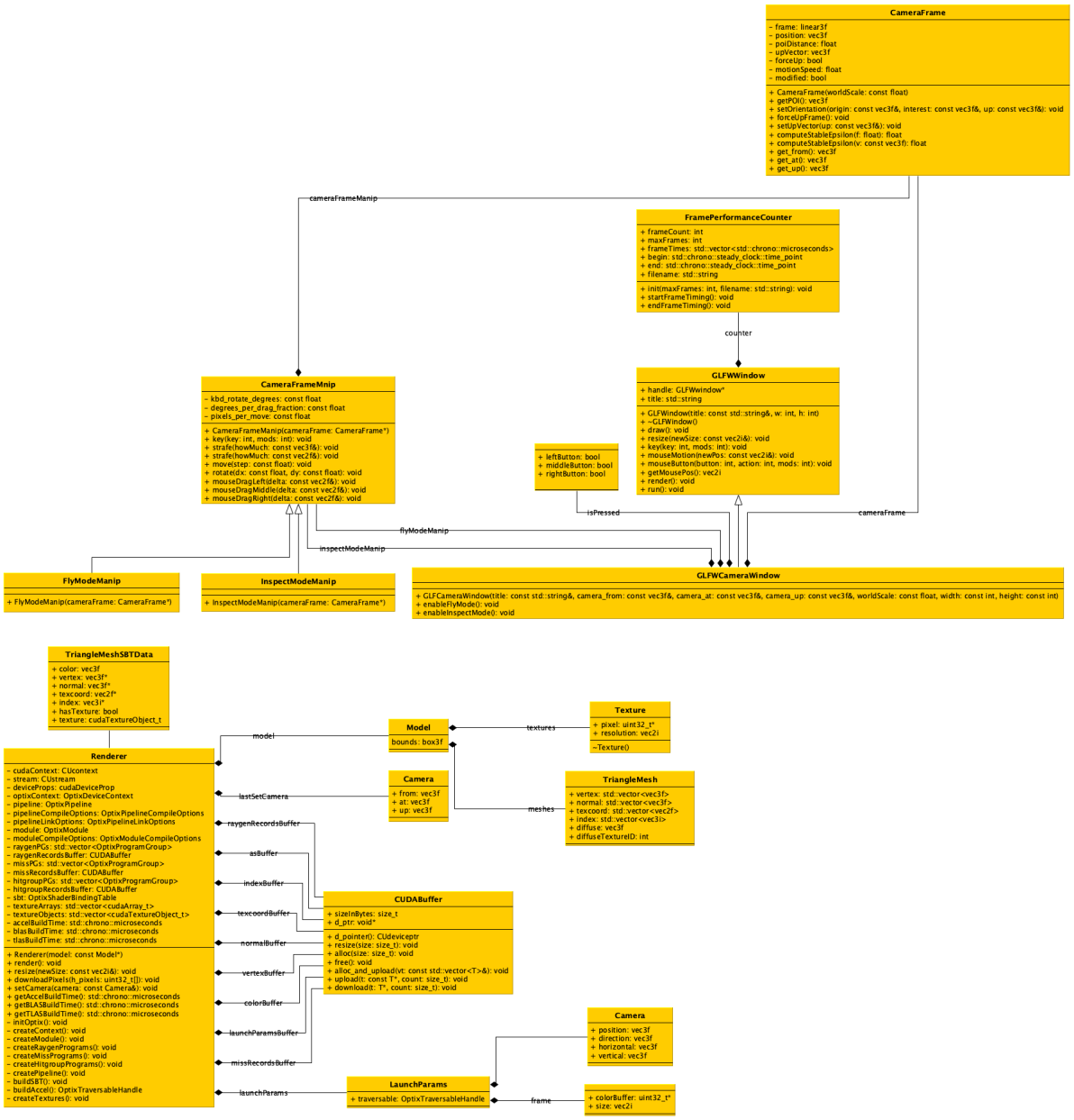


Figura 4.3: UML class diagram for the OptiX ray tracer.

4.3. Flow charts

All the renderers follow the same high level flow. They all start by an initialization process, which we see in detail in figure 4.4. In this they:

1. Read desired window width and height, and model path desired to render, from command line arguments. A couple of fail-safes were placed to have default values for these parameters, in case none were provided. These were added to ease the development process, allowing us to run our software from the development enviroment.
2. Initialize the windowing and input systems are initialized. Despite some low-level differences these all depend on GLFW, so the process is exactly the same for all renderers.
3. Initialize the renderer itself. This process is highly dependent on each API, but it's mostly sequential, checking the hardware we are running it in is supported and filling data structures. This is specified in detail in the Development chapter.
4. Finally, we get to the main loop, which we'll explain in detail next. After exiting this loop, the program ends.

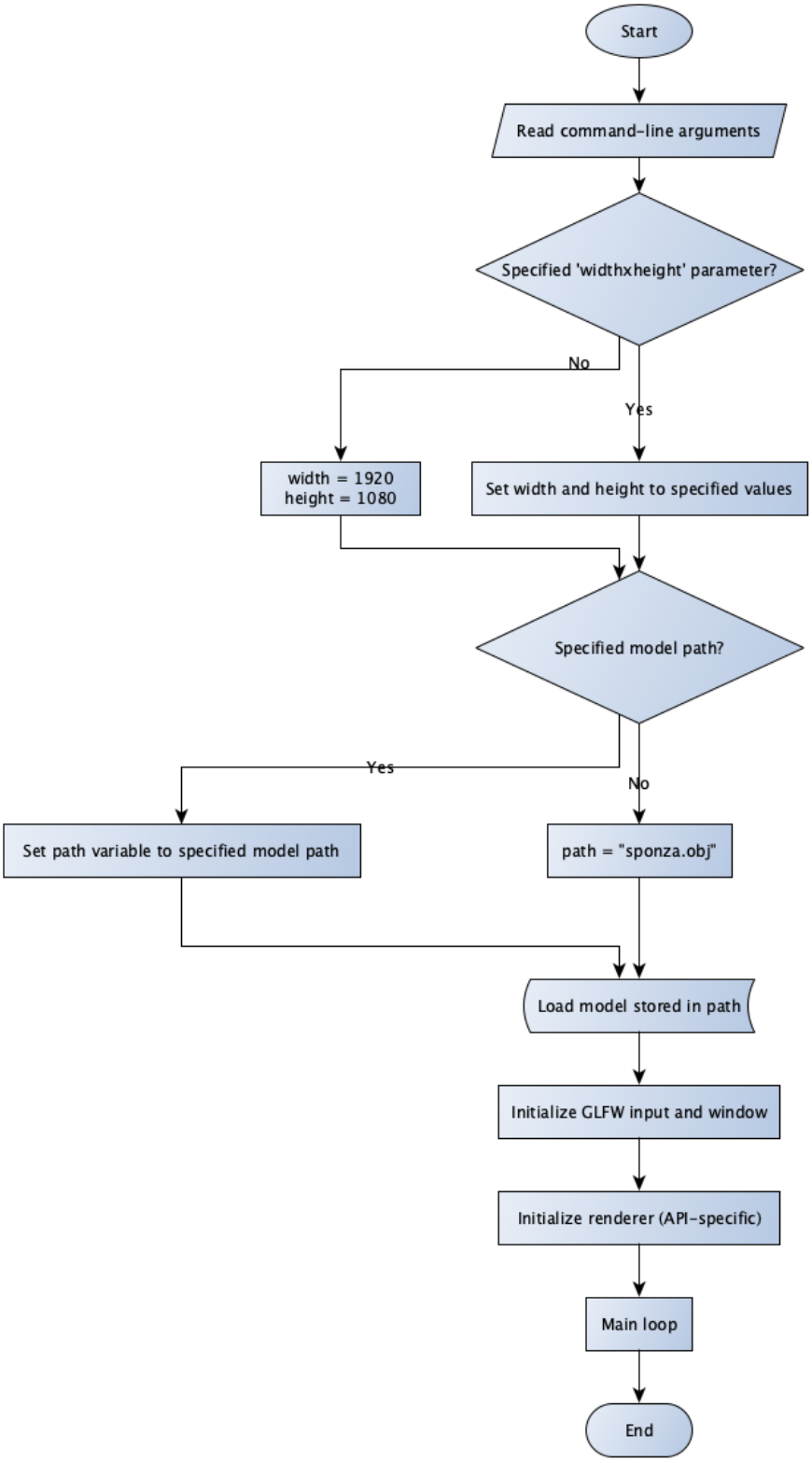


Figura 4.4: UML flow chart.

Chapter 5

Development

In this chapter we will cover the development of all the software employed during the realization of this work, as well as the planning and design of the experiments we desire to make for the benchmarking of both libraries.

5.1. Implementation

In this section we will discuss the process of implementing everything that went into this work. The code written for this comparison can be divided in three categories:

* Two different renderers with as close as function parity as we could, and relying in the same third party libraries for interacting with the operating system and file I/O.
* A series of scripts to automate testing and write performance data to disk.
* A system for generating the plots and graphs in this work from the performance data generated.

In both renderers we used *TinyObjLoader* and *stb_image* for loading 3D models in *.obj* format and *GLFW* for handling the window and user input. The phases of loading models and presenting the rendered image to the screen are out of the scope of this work, so they will be left out of every measure we do. As such, we do not measure the time for loading models and presenting an image to the screen.

5.1.1. OptiX Renderer

First developed by Nvidia around 2009, this ray tracing API allows to offload computations to one or multiple Nvidia GPUs using their own technology CUDA. It has been used in other applications aside from computer graphics where line-of-sight is important, namely optical and accoustical design, radiation and electromagnetic research, artificial intelligence queries and collision analysis. This API is heavily documented in their programming guide, and several beginner-level tutorials have been made for it.

We took the course from *SIGGRAPH 2019/2020* by Ingo Wald in which you learn how to build a simple 3D model viewer capable of simple lighting. Our resulting renderer is very much a typical one consisting of:

* A Ray Generation program on device (GPU) that computes pixel colors. This is called from CUDA, although it requires a "launch parameters"*struct* to be both in device and host (CPU) memory. This structure will encode things like the camera data and acceleration structure (AS). It allows for compaction in the AS, but we will not be diving in it's implementation details here. The program will also need sub-programs for when a ray either misses or hits scene geometry. This CUDA code is compiled and embedded into the C++ host code during compilation.
* A pipeline, which will handle the kind of programs that we want to run. For this we will need to create an OptiX Context and Module with the embedded CUDA code and set up the required program groups that will go into such pipeline (raygen, miss, hit group). Finally, with all this information we can initialize an OptiX Pipeline.
* A Shader Binding Table (SBT), which will handle the exact configuration for the programs we will be running. It's a set of ray generation", "miss.ªnd "hit groupecords to run. Each record contains a header describing the program to run and user-supplied data, like per-mesh CUDA texture objects. All these records are filled during creation of the SBT and need to be uploaded to a CUDA buffer.

* A Frame Buffer in which to store the generated image. After measuring the rendering time, we copied the buffer's contents to a GLFW window using OpenGL. This allowed us to get immediate feedback on what the renderer was doing.

In order to add hard shadows to the scene, we implemented a new ray type with it's own çlosest hit.ªnd .ªny hit"programs (in this case all the work happens in the .ªny hit.ºne, with the others doing nothing for shadows). To keep the development time reasonable we assume all surfaces are opaque, killing the ray upon it's first occlusion. As a result of this second ray type, the Hitgroup Program Group has two entries (one for radiance rays and one for shadow rays) and the SBT has to create two records per mesh. In our case we use the same data in both records, although this is not mandatory.

Finally we ported the resulting code from OptiX 7.3 to 7.5, it's latest version. This required only a couple of minimal changes.

From that starting point we instrumented the ported code in order to measure frame rendering time, acceleration structure building time and memory consumption, as well as parametrizing values for frame buffer width and height and which 3D model to load. We used the *chrono* library from the C++ STL and it's *high_resolution_clock* for measuring time and the function *cudaMemGetInfo()* to consult the memory usage. We built all this functionality in a class *FramePerformanceCounter* that handled data recording and file I/O, making it easy to share this functionality between renderers.

5.1.2. Vulkan Renderer

This phase was the lengthiest of the whole development process. Vulkan, highly explicit, requiring the programmer to recreate the whole rendering pipeline and configure every stage for drawing even the simplest scenes. The following steps require significantly less effort, though it does not decrease as greatly as it does with OptiX.

5.1.2.1. Rasterized

We started by building a traditional rasterized renderer based on the one from the Vulkan Tutorial (Overvoorde, s.f.), which allowed us to get familiar with the API before using it for tracing rays, as well as giving us the opportunity of comparing ray traced graphics to rasterized ones with as similar features as possible. We then instrumented and parametrized it almost as we did with the OptiX renderer. The main difference was querying for memory usage using the *DirectX Graphics Infrastructure* API (White y cols., s.f.) as explained in *There is a way to query GPU memory usage in Vulkan - use DXGI* (Sawichi, s.f.). Even though we could have used the Vulkan extension *VK_EXT_memory_budget*, we decided against it since it was harder to figure out it's usage compared to the more straight forward *DXGI*.

5.1.2.2. Ray Traced

We expanded our rasterized Vulkan renderer following the course by Nvidia on this very topic (Lefrançois, s.f.), which gave us a simple ray tracer capable of rendering 3D models and adding some lights. On a high level, this renderer has similar components to the one we wrote in OptiX, with some differences:

* Acceleration Structure (AS). As with the rasterized renderer, Vulkan is much more explicit than even the relatively low-level OptiX 7, making us manually convert our triangle geometry data into multiple structures that will be consumed by an AS Builder. This builder will then store it in one or multiple Bottom Level Acceleration Structures (BLAS). As with the OptiX raytracer, we indicate (to the AS Builder this time) that all our geometry is opaque, disabling calls to the .ªnyhit"shader. Each BLAS also allows for compaction. Broadly speaking, it works by querying the initial (large) BLAS for only the values we require, creating a new BLAS with a smaller size, copying the data we have to the newly allocated one and destroying the large BLAS. It requires waiting until a whole BLAS is built since only then we know how much memory it actually uses.

  With these BLASes we build a single Top Level Acceleration Structure (TLAS). This is the entry point for the ray tracing scene descriptor, as we will explain later, and stores all the geometry instances. Each instance is represented by a transform matrix, a BLAS ID, an instance ID and a "hit groupïndex. This index represents the shaders that will be invoked uppon hitting the stored object, and are tied to the definition of the Shader Binding Table and the Raytracing Pipeline, as we shall see. Since our renderer only uses one hit group, this index will always be 0. Once all the configuring is done, we build the TLAS. We have decided to optimize for ray tracing performance instead of memory size, though this option doesn't seem to be present in OptiX.

* Ray Tracing Descriptor Set. This component references external resources used by shaders. In the rasterized graphics pipeline we can group the rendering objects by the materials they use, and draw all the objects that use some materials all together. This way, we only need to bind the descriptor set that references those materials while rendering the objects they use them. In Ray Tracing, however, we can't know which objects in the scene will be hit by a ray, and as such any shader can be invvoked at any time. Thus, we need to use a set of Descriptor Sets containing all the resources necessary to draw the scene (like all the textures of all the materials). Finally, since the Acceleration Structure contains only position data, the geometry's vertex and index buffers need to be passed to the shaders so that they can manually look up the rest of the vertex attributes.

* Ray Tracing Pipeline. As mentioned earlier, we need to have every shader available for execution at any time when raytracing, and the shaders to execute are selected on the GPU at runtime. The structure that makes this selection possible is a Shader Binding Table. This is essentially a table of opaque shader handles (probably device addresses) analogous to a C++ v-table. As with everything in Vulkan, we have to build this table ourselves. A high level overview of this process is:

  1. Load and compile shaders into *VkShaderModule*s in the usual way.

  2. Package said *VkShaderModule*s into an array of *VkPipelineShaderStageCreateInfo*.

  3. Create an array of *VkRayTracingShaderGroupCreateInfoKHR*. Each element will eventually become an SBT entry. At this point, each shader group references a single shader by it's index in the array created created in the last step.

4. Compile the two arrays created plus a pipeline layout (as usual) into a ray tracing pipeline. This converts the array of shader indices into an array of shader *handles*. We can query this at will.

5. Allocate a buffer for the SBT and copy the handles to it.

The ray tracing pipeline is more similar to the compute pipeline than the rasterization pipeline: ray traces are dispatched in an abstract 3D space, with it's results manually written using *imageStore*. However, unlike the compute pipeline, we dispatch individual shader invocations, rather than local groups.

The entry point for ray tracing is the ray generation shader, which we call for each pixel. It typically initializes a ray starting at the location of the camera in the direction of the camera lens model at it's corresponding pixel's location. The miss shader and a closest hit shader, which work in the same way as in OptiX.

The *intersection* shader is used to intersect user-defined geometry. This can be useful for intersecting placeholders when using on-demand geometry loading, or procedural geometry without tessellating it beforehand. We will not be using this type of shader in this work, since it requires modifying how the acceleration structures are built. Instead, we will solely use the ray-triangle intersection test provided by the Vulkan extension, which returns 2 floats representing the barycentric coordinates of the hit point inside a given triangle.

Finally, the *any hit* shader is executed in each potential intersection. When we look for the closest hit point to the ray origin, we may find several candidates. The any hit shader is often used to efficiently implement alpha testing so we know if the ray traversal can continue. The default any hit shader is a simple passthrough that returns the intersection to the traversal engine, which determines which intersection is the closest. We will not be using this shader during this work as all our geometry is opaque.

* As mentioned before, the Shader Binding Table (SBT) works as the blueprint of the ray tracing process. It helps selecting the shaders to use as an entry point, in case of rays missing geometry, and which hit shader groups can be executed for each geometry instance. The link between instances and shader groups is created when setting up the geometry, as we provided a hit group ID in the TLAS for each instance. This value is used to calculate the SBT index corresponding to the hit group of said instance.

  From an implementation standpoint, the SBT is just 4 arrays containing the handles of the shader groups used in the ray tracing pipeline. There is one array for each shader group, namely ray generation, miss, hit and callable (not used in this work). Since we only have one shader of each type, for us each .[a]rrayïs just a handle to a group of shaders.

We employed the *nvvk* utilities at the *Nvpro* library (Nviidia, 2022) to facilitate building this renderer. The same instrumentation and parametrization as in the rasterized renderer was used here.

5.1.3. Automation and Plotting

To get reproducible results in a reduced ammount of time, we implemented a series of Python scripts in order to run every renderer with every combination of scenes, renderer settings (like use of shadows or textures) and framebuffer resolutions. Each renderer run will handle savingtheir performance information to disk. We later plotted these results in a single script using *matplotlib* to more easily gauge how each library behaved under the same configuration. The whole suite of experiments ran just under 10 minutes in every machine we tested it on.

Chapter 6

Results

In this chapter we will show the results obtained in this work. They will be divided between the three metrics we chose to monitor (see section Experiment Design in the Development chapter).

6.1. Rasterization Baseline

First of all and as a sanity check, we will compare the performance of a renderer using ray tracing to one drawing the same geometry through rasterization. The model chosen for this will be the Viking Room from the Vulkan Tutorial (Overvoorde, s.f.).

In the first place we will look at how video memory usage changes as the frame sizes get bigger. We can see a comparison of how much memory both renderers employ in graph 6.1. We see an expected increase in memory consumption as the frame buffer resolutions get bigger, as well as a huge difference, of about an order of magnitude, between the two methods for any given resolution. This is to be expected not only because a bigger frame buffer will require more memory to be stored, but also because the ray tracer requires to store an acceleration structure as well as all the possible shaders bound at the same time.

Rasterized renderers have an initialization process quite different than raytraced ones since they do not make use of an acceleration structure. Therefore, we cannot compare it's building time in this context.

To finish this part, we will compare frame times from both renderers in the same fashion as the memory usage one: with the same model (Viking Room), we'll run them for 1000 frames and take the average time it took to render them at different screen resolutions. We can see how they stack up in figure 6.2. This is very similar to the memory usage comparison in that both frame times increase as the frame buffer resolution grows, with the rasterized renderer finishing the job much quicker than it's counterpart. In any case, in this instance the performance difference is even more exaggerated between the two, with rasterized graphics greatly surpassing ray traced ones.

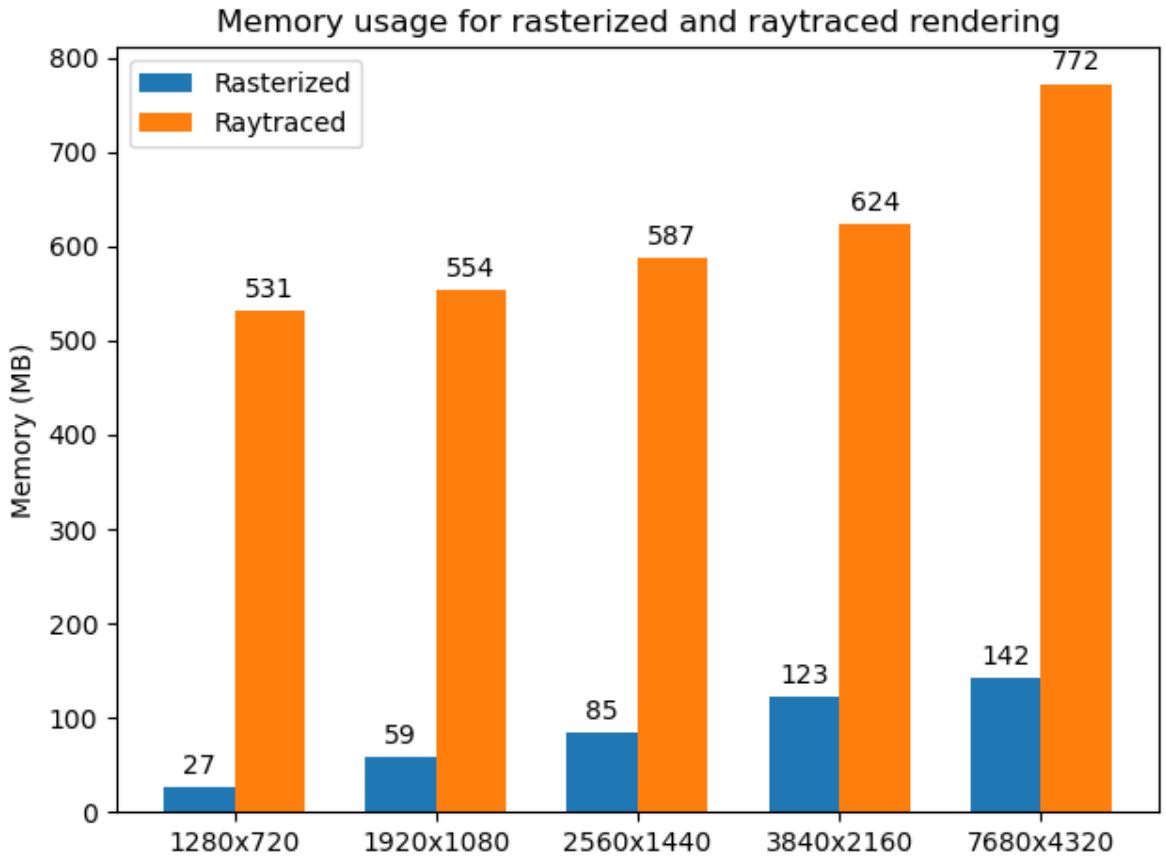


Figura 6.1: GPU memory consumption when rendering the Viking Room 3D model at a range of reasonable resolutions, using ray traced and traditionally rasterized graphics. We can see a significant difference between the two, as well as an expected increase as resolutions get higher.

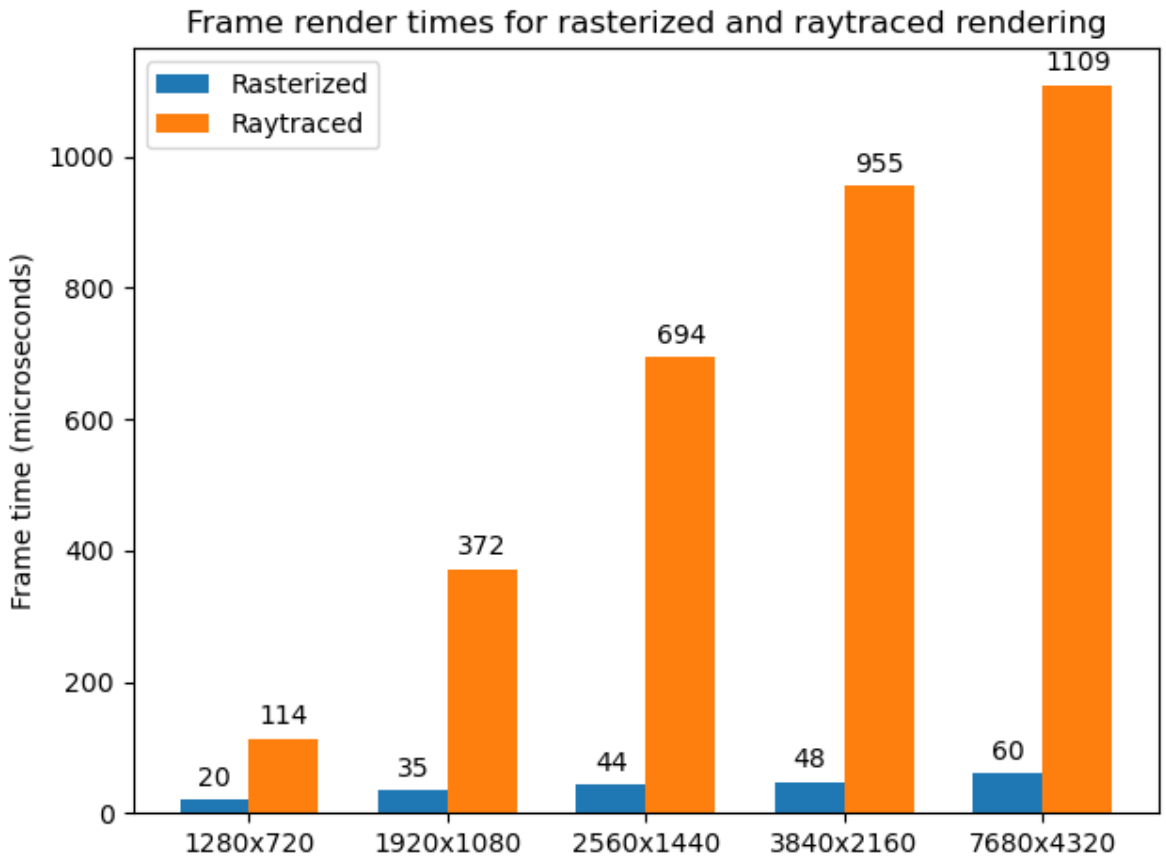
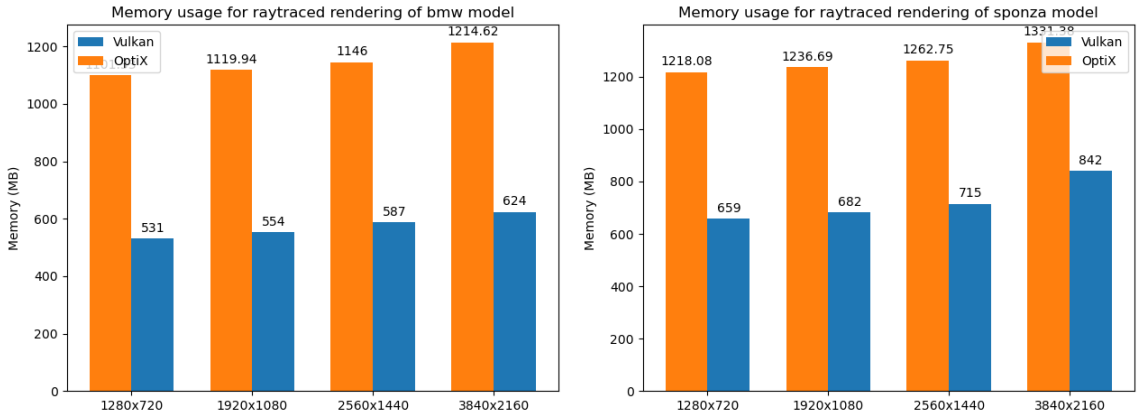
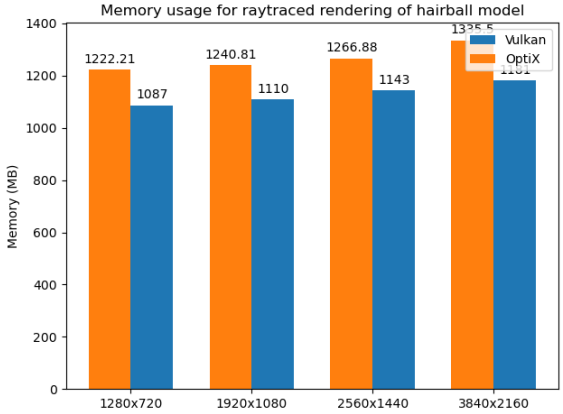


Figura 6.2: Single frame rendering time when drawing the Viking Room 3D model at a range of reasonable resolution, using ray traced and traditionally rasterized graphics. As with the memory consumption graph, we see how the two of them take longer to finish in higher resolutions, while the rasterized renderer greatly outperforms the ray traced one.



(a) BMW

(b) Sponza



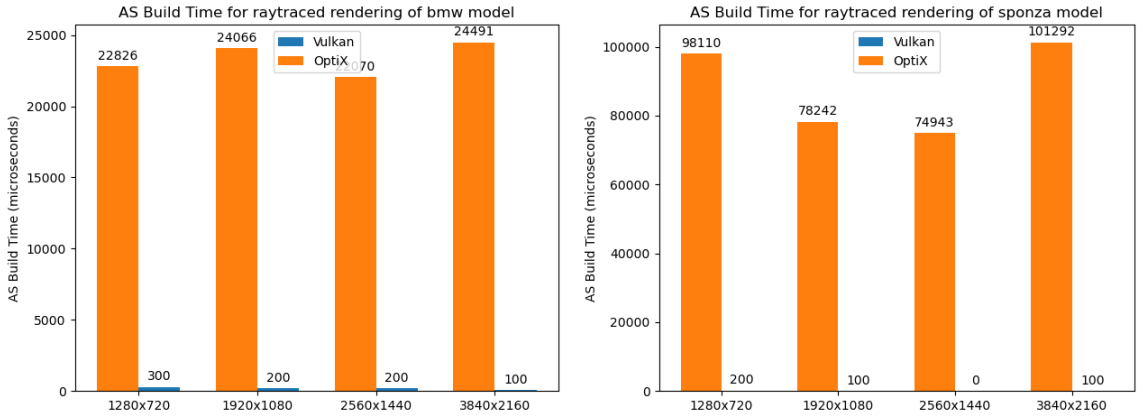
(c) Hairball

Figura 6.3: Memory consumption of OptiX and Vulkan (raytraced) when rendering the 3D models BMW, Sponza and Hairball.

6.2. Memory Usage

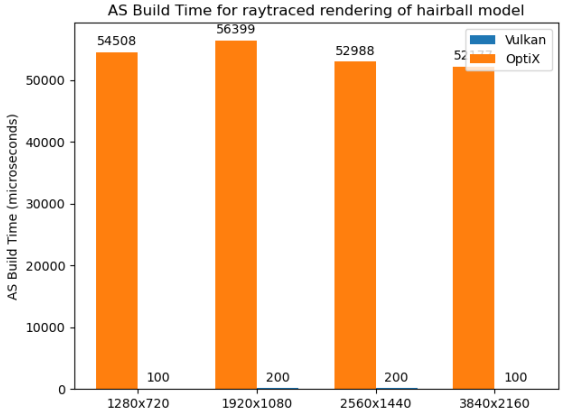
To start the real comparison between Vulkan and OptiX we will look at the memory usage in GPU across both libraries, all models and all resolutions. We can see how much each model consumes in graph 6.3.

This is the first example of a trend we will start seeing throughout this chapter. Vulkan, either despite or thanks to it's explicitness when developing applications, is able to load and render the same ammount of geometry and detail as OptiX while requiring only a fraction of the resources, in this case video memory. This remains true while increasing the frame buffer size, with both libraries increasing the required memory at a similar rate. One last remarkable thing is the Hairball test case. Here we see the memory consumption for both libraries is very similar, which suggests the two technologies tend to use a similar ammount of resources when triangle counts tend to infinity.



(a) BMW

(b) Sponza



(c) Hairball

Figura 6.4: Acceleration Structure Build Time of OptiX and Vulkan (raytraced) when rendering the 3D models BMW, Sponza and Hairball. Sorted by rendering resolution.

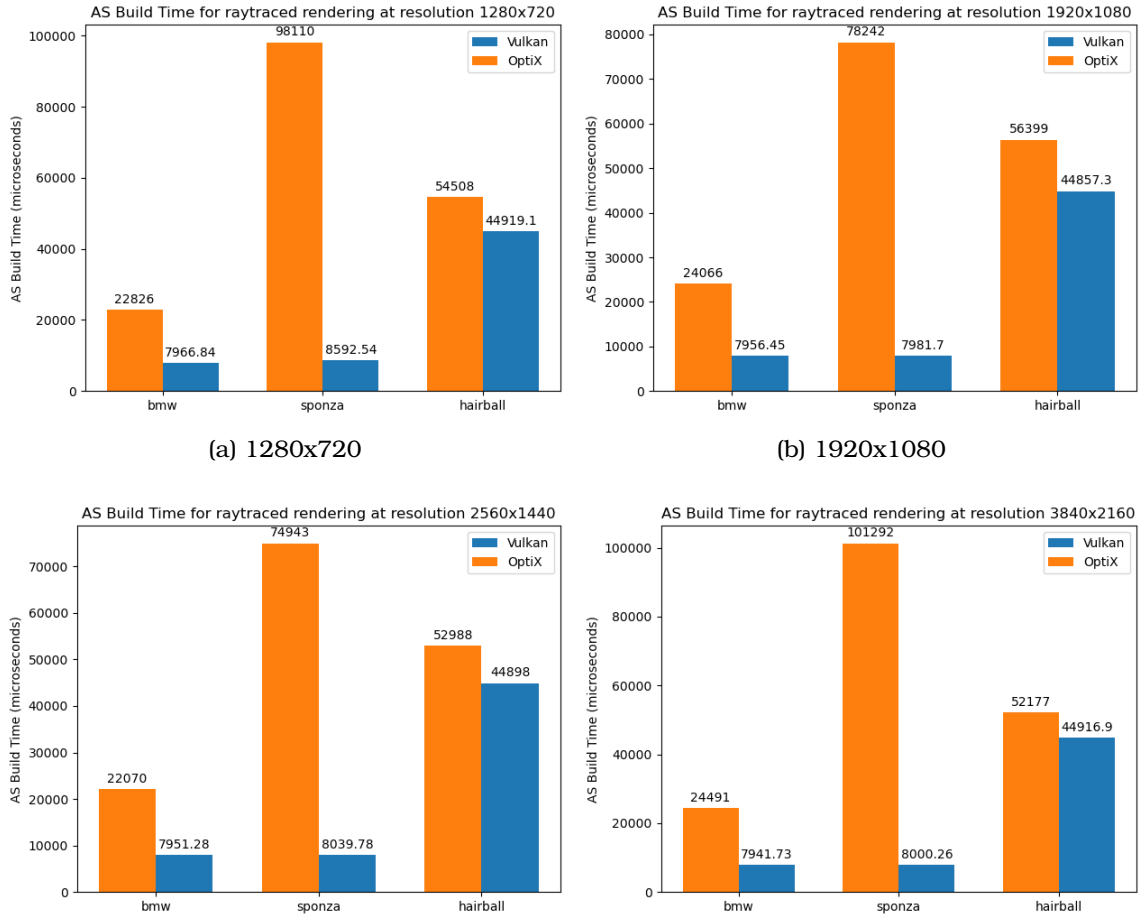
6.3. Acceleration Structure Build Time

Next up we will be looking at the Acceleration Structure Build time across both libraries. Although in our experiments this was only done once per program execution, applications that perform animations or other form of dynamic geometry may need to partly rebuild their AS as frequent as once per frame. Thus, the time it takes to do so (although not all of it) could be counting towards the total time it takes to process a full application cycle (along with rendering, and simulation, etc.).

If we plot the Acceleration Structure Building Times in the same way as we did with the memory consumption, we get the graphs at figure 6.4.

We can plainly see how the rendering resolution does not affect the accel build time. This is to be expected, since a higher pixel count works with the same Acceleration Structure as a lower pixel count. We will now rearrange the plots in figure 6.5 to see how geometry ammount affects acceleration structure build times.

A curious phenomenon is shown here. We initially expected the Hairball model to have the greatest acceleration structure build time. This is true for Vulkan where the



(a) 1280x720

(b) 1920x1080

(c) 2560x1440

(d) 3840x2160

Figura 6.5: Acceleration Structure Build Time of OptiX and Vulkan (raytraced) when rendering the 3D models BMW, Sponza and Hairball. Sorted by model.

triangle count is directly proportional to the build time. In OptiX however, the first spot is granted to Sponza, with about 10 times less triangles than the Hairball. This could be due to this model having a wider variety of shapes formed by it's triangles, which could be harder to optimize, as opposed to the hairball, where everything is more similar and packed together.

When comparing both libraries, even in the most discrepant case (Hairball), we see Vulkan being at least slightly faster, around a 20 % on average across all cases and in most of them simply a lot faster than OptiX when building the AS.

To further investigate why Sponza takes that much longer to build than other models, we first timed each part of the building process separately. We see this in figure 6.6. We observe how the distribution of times for each model is completely different. The Hairball model takes the longest by far to build it's TLAS, while Sponza takes it's much longer time mostly building the Bottom Level part, taking as little as the BMW one in the Top Level stage.

Seeing this, we decided to expand this metric's models testing pool to Human, ISCV2 and Gallery. In figure 6.7 we see how their build times stack up to one another. In

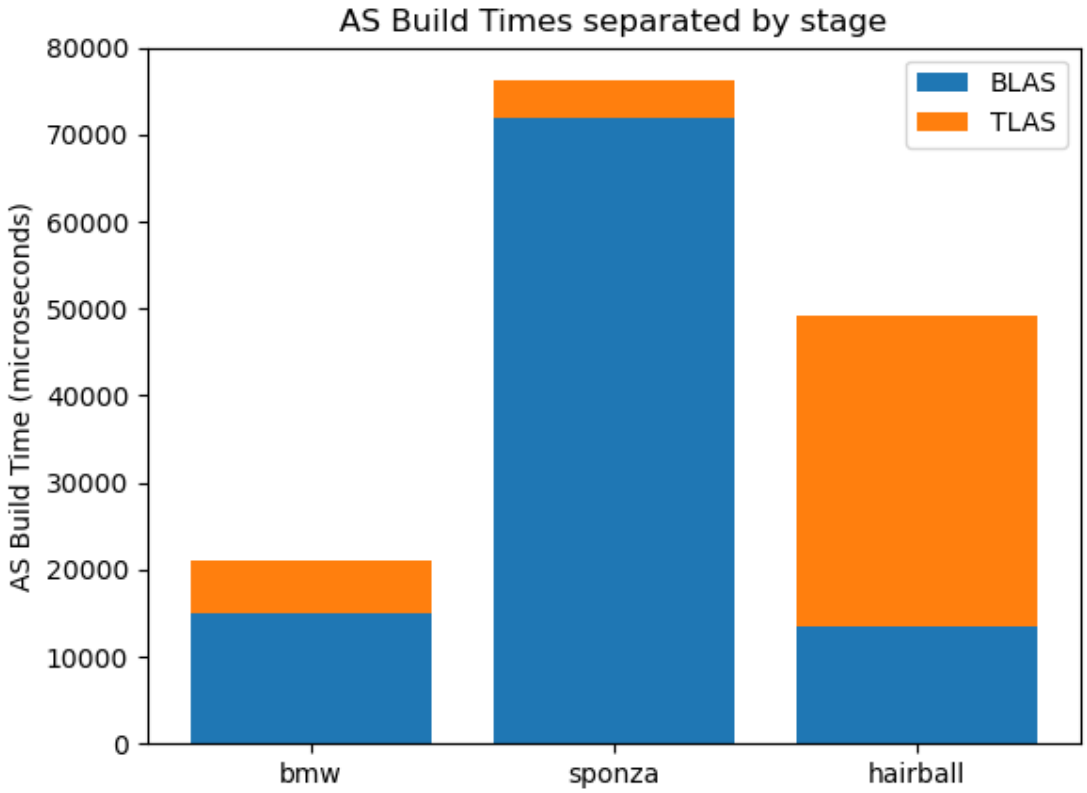


Figura 6.6: Acceleration Structure Build Times for each model, divided between the TLAS and BLAS phases. We see great discrepancies in the distribution of time in each case.

sight of these new data points we appreciate that a longer build time seems to come not only from the triangle count, but rather from the presence of textures, that seem to extend the Top Level Acceleration Structure build time.

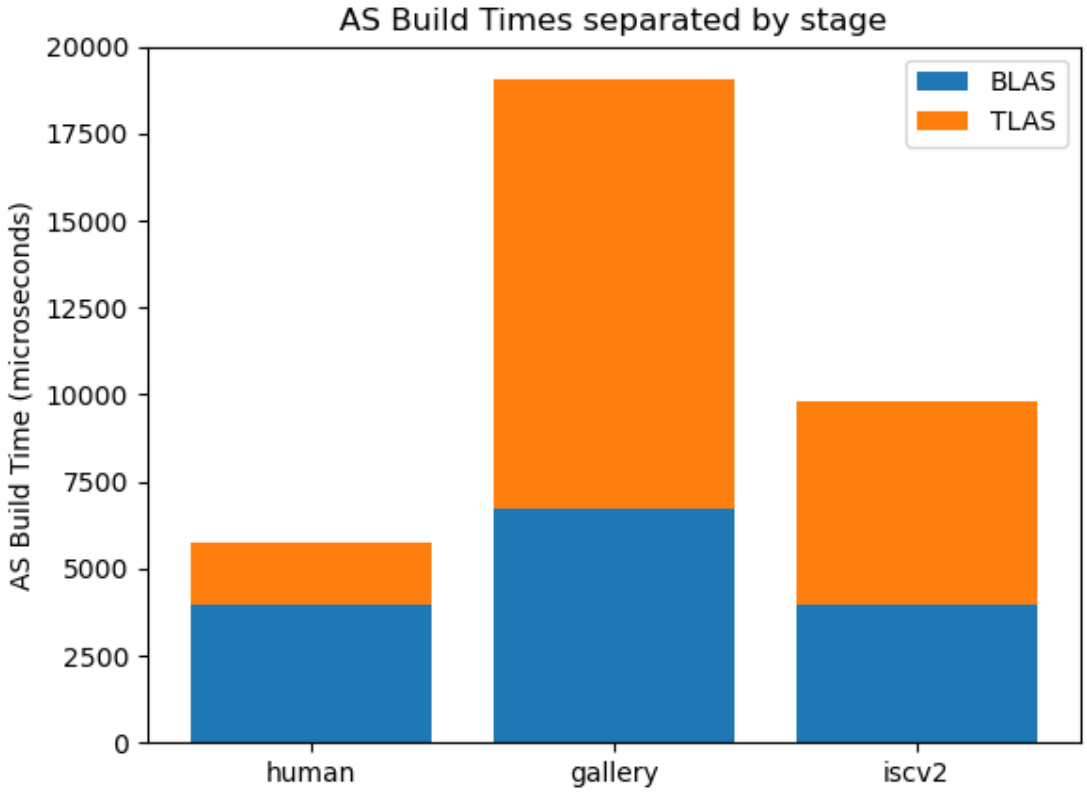


Figura 6.7: Acceleration Structure Build Times divided between the TLAS and BLAS phases for the extra set of models (Human, ISCV2 and Gallery). We see how the presence of textures makes for a longer TLAS build time.

6.4. Frame Time

Finally we will look at how long each technology takes to render a full frame. This could be considered the most important metric to monitor, since rendering a frame is something that will happen as often as possible, and therefore the ammount of time it takes to do so will be more prevalent than updating or initializing geometry structures.

As mentioned before, each frame will be rendered to a frame buffer and we will only count the time it takes to do exactly that, although we will then display them in a window so we can see what the renderer is doing.

In a modern operating system, the time it takes to perform a certain task can vary a lot depending on what other processes are running at the time. To counteract this, we timed the render of 1000 frames for each parameter combination and averaged them. An example of that can be seen in figure 6.8.

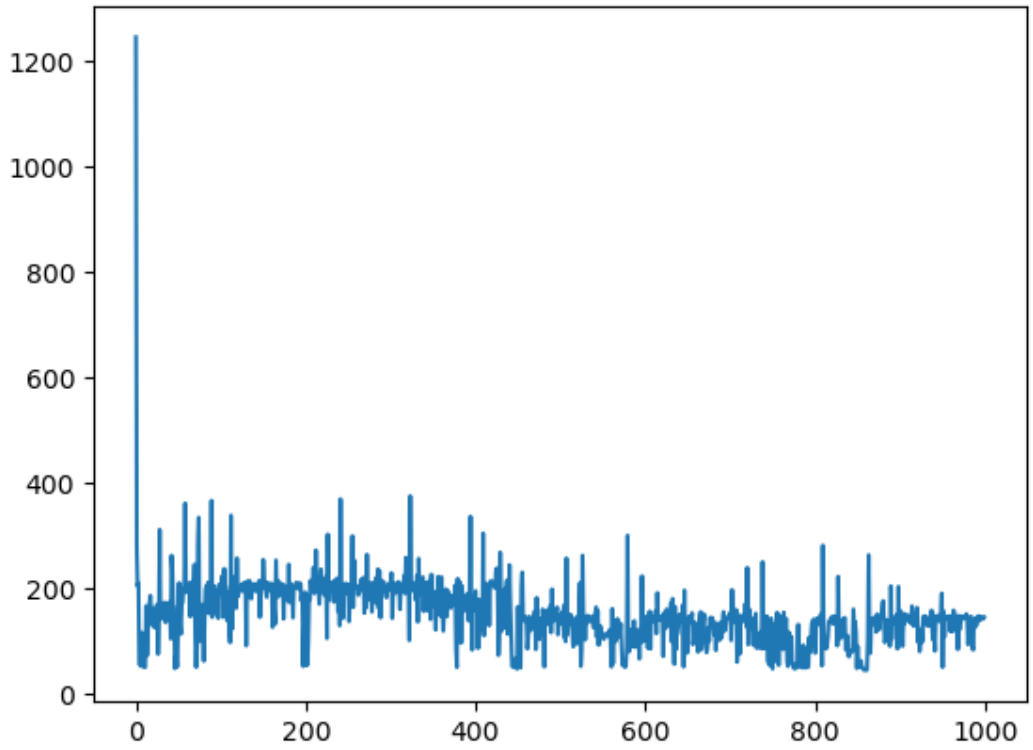


Figura 6.8: Time taken for rendering 1000 frames when drawing the BMW 3D model at 1280x720, using ray tracing. We see how the first frame takes much longer than the rest of them to draw.

We decided to remove the first frame time from this average since, as you can see, it tends to be a statistical outlier, messing with real world.[a]ppreciable results. The same graph from **??** results thus in **??**.

With this preprocessing in place, we will first look at an overview for all models and

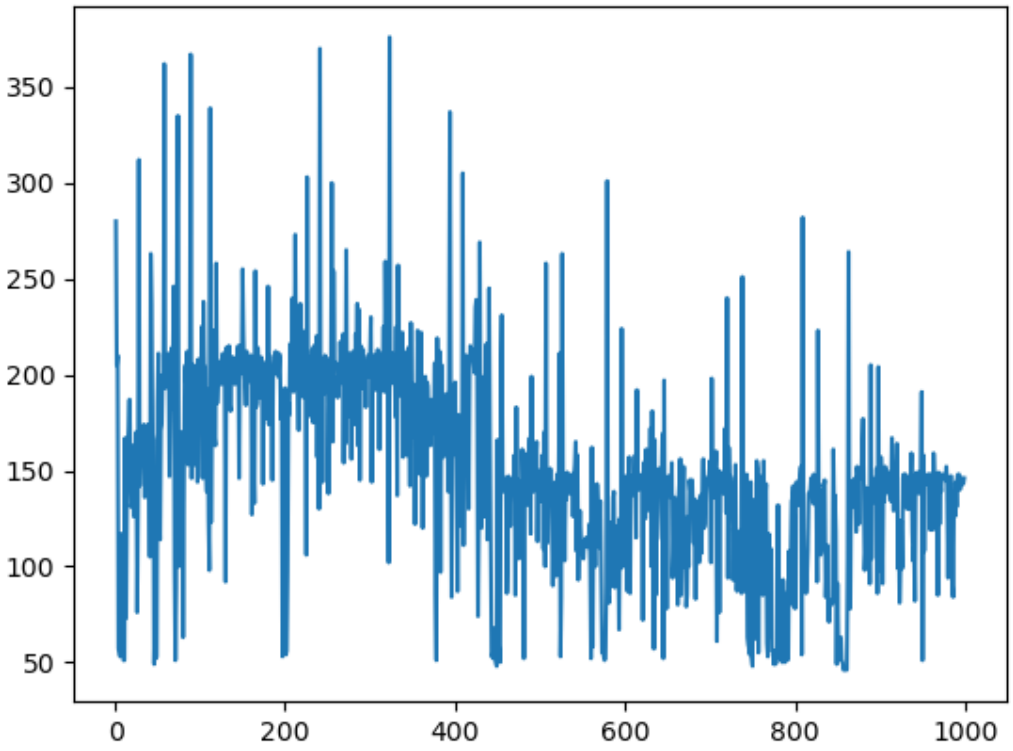
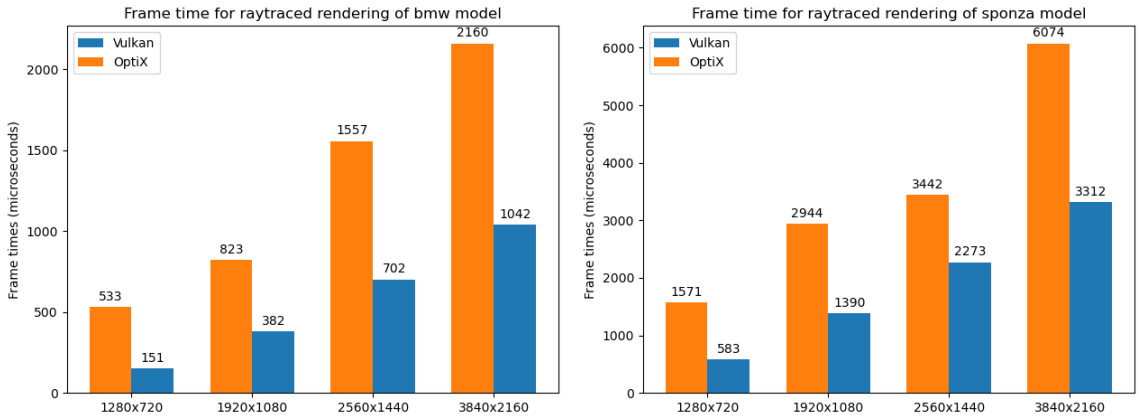


Figura 6.9: Time taken for rendering 1000 frames when drawing the BMW 3D model at 1280x720, using ray tracing. We have removed the first frame time since it took too long to draw in comparison with the rest.

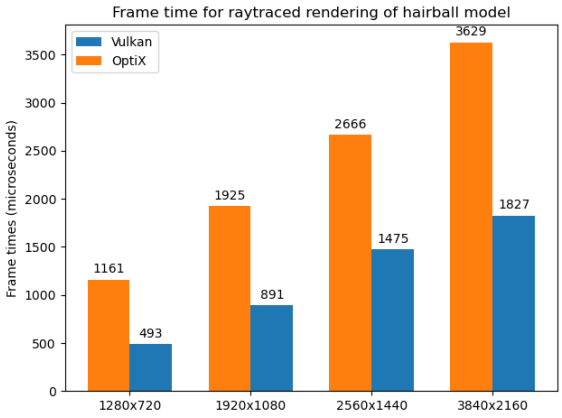
resolutions in figure 6.10. For the most part the graphs show what one might expect: render times get larger as screen resolution increases, with OptiX taking the longest across all cases. There is however a slightly odd yet expected behaviour. As we see, the model taking the longer to draw in both libraries is Sponza. The much bigger Hairball shows times more similar to the BMW model, with about 10 times more geometry. This suggests that complexity in shapes and textures may be more costly to render than a higher triangle count.

This discrepancy between triangle count and processing time reminds to the Acceleration Structure Build Times, where OptiX also struggled the most to optimize the Sponza geometry compared to the others. The main difference is here both libraries have the most trouble to draw this particular scene.



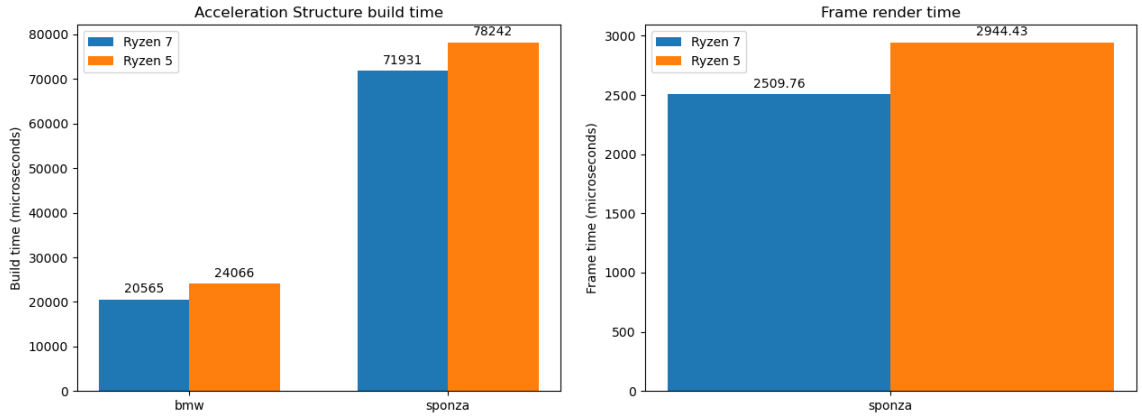
(a) BMW

(b) Sponza



(c) Hairball

Figura 6.10: Frame rendering times for each 3D model (BMW, Hairball and Sponza) across both libraries and 4 reasonable resolutions, from 720p to 4k.



(a) Acceleration Structure Build Time

(b) Frame Times

Figura 6.11: Acceleration Structure Build Time and Frame Render Time comparison in OptiX when using an Nvidia 2070 Super in two systems with different CPUs. We see how a faster CPU means better performance in both cases.

6.5. Hardware comparison

As mentioned in the Development chapter, we ran our software in all the machines we could gather from friends, family and fellow students. This will help us gauge how the same workload is handled by different GPUs, how CPUs affect performance while maintaining the same GPU and even how both libraries behave inside a virtual machine.

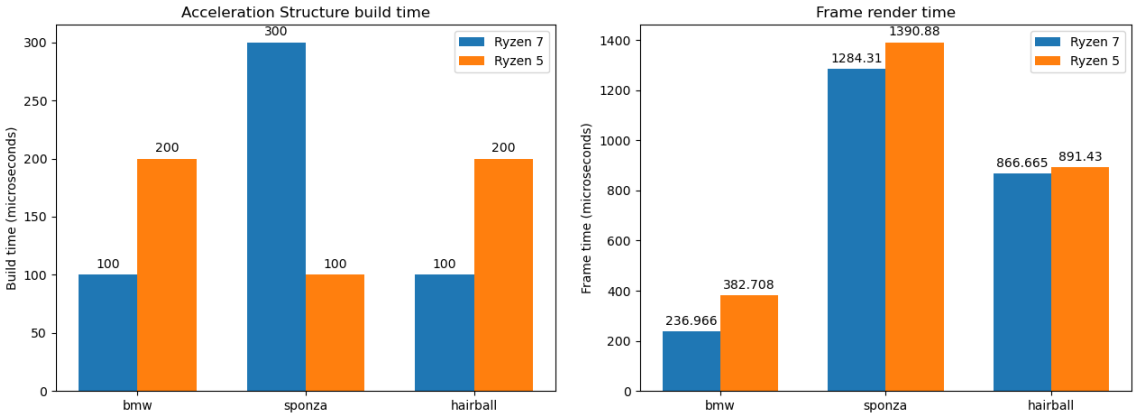
6.5.1. Same GPU, changing CPU

We'll start by comparing how a difference in CPU power affects all the metrics we're monitoring, except for memory usage of course. We have a couple of pairs of systems with the same GPUs and different processors. We will be looking at two systems. One has an AMD Ryzen 5 3700 BOX and the other an Intel Core i9-12900F. Both of them are using an Nvidia 2070 Super.

We will start with OptiX. In graph 6.11 we see how acceleration structure build time and frame times compare to one another when using two machines with a *2070 Super* and AMD Ryzen 5 and 7 processors respectively. Some of the results data went missing when trying to get the software to run on other people's systems, so we will not be comparing all the models' metrics every time. Additionally, we have only compared the render time averages at a 1920x1080 resolution to simplify the graphs.

In both cases we see a marginal increase in performance (less time consumption) in the system with the more powerful CPU. In all cases it's small enough that it could be considered a fluke, being a difference smaller than a 10% every time. However, since it happens accross all tested models and all the averages in render time, we can determine it does have an impact on render and acceleration structure build times.

In figure 6.12 we see the same comparison for Vulkan, only this time using a larger pool of models. Note how in this case the model of CPU doesn't seem as important, with both of them registering shorter times depending on the metric and scene. We



(a) Acceleration Structure Build Time

(b) Frame Times

Figura 6.12: Acceleration Structure Build Time and Frame Render Time comparison in Vulkan when using an Nvidia 2070 Super in two systems with different CPUs. We see more variation in which system does better, in contrast with the previous OptiX test.

assume from this that Vulkan is not as dependent on the CPU as OptiX when it comes to building an acceleration structure or rendering.

6.5.2. Virtualization effects

Virtual machines are a staple of computing, indispensable when dispatching a job to a server or running experiments in isolation. During the development of this work we came across a curious use of these systems, in which people who primarily use Linux run a small virtual machine for specific purposes, such as gaming or having access to some software.

VM's dedicated to gaming or any other intensively graphically intensive work use GPU passthrough to have access to near bare metal performance. We will now analyze to what extent this is true, by running our benchmark in a VM dedicated almost exclusively to gaming by it's user. According to them, it's performance is almost identical to running the games bare metal, with some minimal additional stuttering. As shown in the Development chapter, this machine uses 12 out of the 16 cores available in it's processor.

The comparison will not be exact, since the most similar system we had access to had an Intel Core i7, compared to the i9 running the virtual machine. However, the GPU is the same across both systems, which should be close enough for our purposes. Unfortunately, due to driver issues, we will only be looking at the Vulkan data for this case, since we couldn't get the OptiX ray tracer to run on the Virtual Machine.

We'll start by looking at the Acceleration Structure build time in 6.13. We see how the virtualized system is three times slower than the bare metal one for this task, despite having a more powerful processor. This makes sense to an extent, since this task is relatively CPU-intensive, even though we just saw Vulkan depends less on this component than on the graphics card.

Next up we will look at frame render times across multiple models in figure 6.14. We see how, opposite to the acceleration structure build time, the virtual machine manages to draw frames marginally faster than the bare metal environment. The difference between the two is so small that we could attribute it to a fluke, concluding that the CPU isn't really relevant when rendering a static scene and that the render time will in essence only depend on the GPU's capabilities.

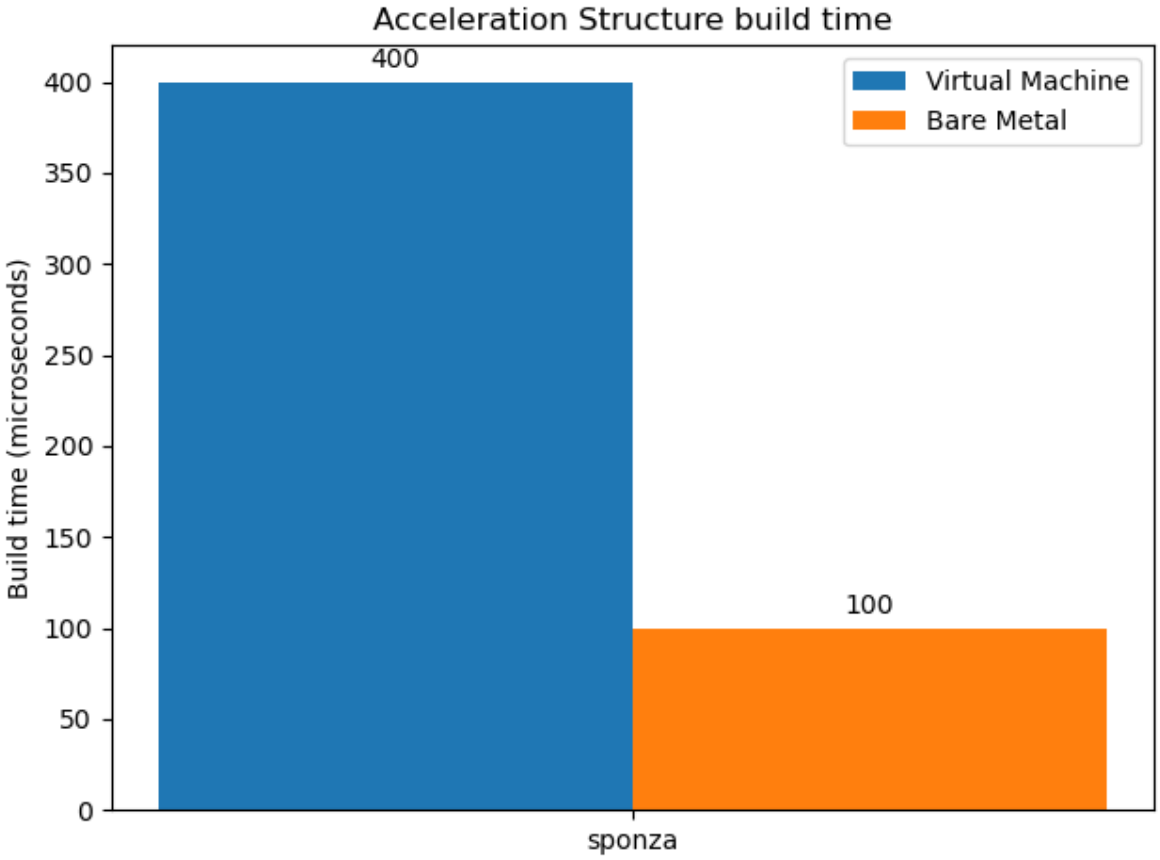


Figura 6.13: Acceleration Structure Build Times for Vulkan in a Virtual Machine and running bare metal. We see how the bare metal solution manages to get a significantly lower time than the virtualized one.

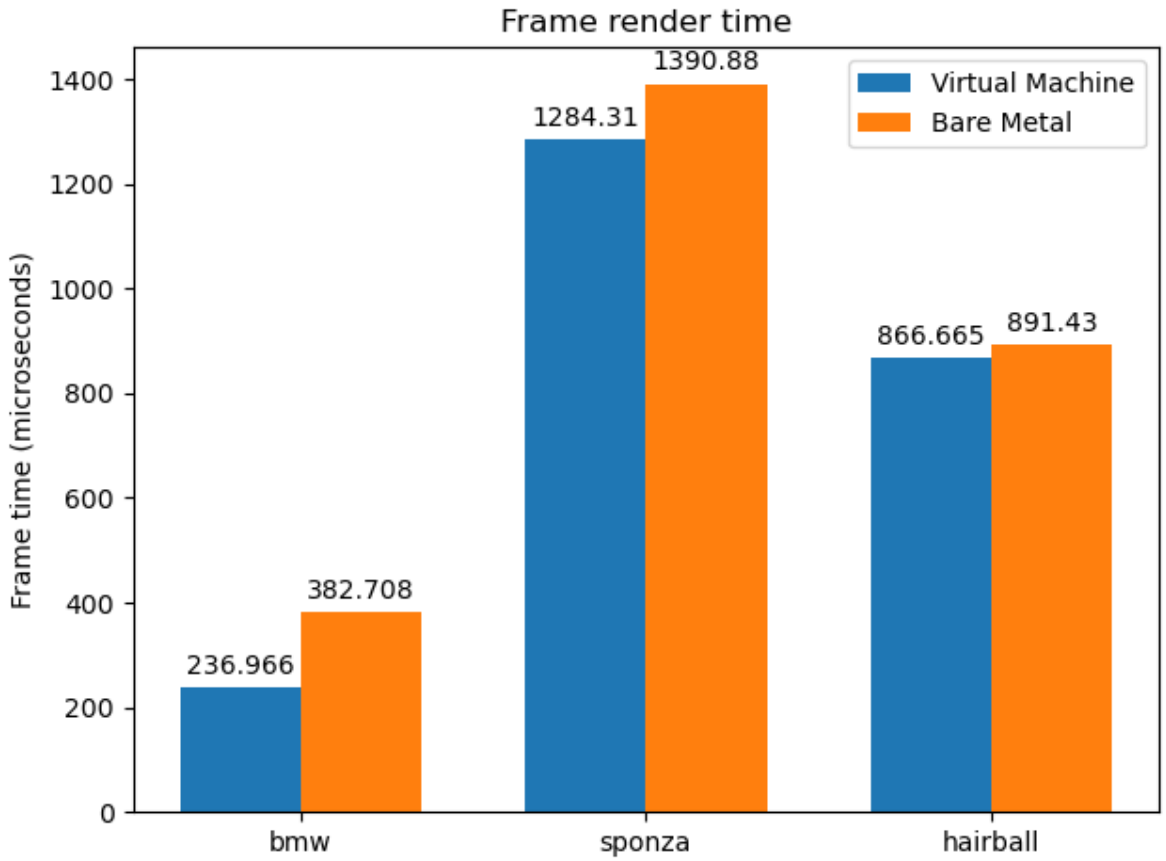


Figura 6.14: Frame render times in Vulkan across two machines with the same GPU (Nvidia 3080). One of them is running the renderer under a virtual machine and the other is doing so bare metal. We see a slight advantage towards the virtualized approach in this side, which could be due to a mere coincidence. This would make the GPU the only important element when drawing a given scene.

Chapter 7

Conclussions

In this chapter we will explain the our personal conclussions from doing this work, as well as possible future works.

My main goal as a student in choosing this particular project was to further my understanding of ray tracing and modern rendering, beyond what casual tutorials in my spare time could give me. I believe this has been achieved, since we had to build not one but two, although basic, functioning ray tracers with technologies that are currently in use in the graphics industry. Not only this, but I was also able to learn other technologies to aid in their development, such as Nvidia Nsight, as well as know what to look for when evaluating rendering applications.

This master's degree has a bigger focus on machine learning than computer graphics and during the school year you have to develop several projects in that area. Combining those and my bachelor's thesis in computer science, which also revolved around ML, I have gathered some experience that was useful in comparing that field to computer graphics. I believe there are two key differences:

* The development time is much longer in this field than in ML. At least for a majority of projects of equivalent scope and level of difficulty in both areas, a project in computer graphics will usually require to build most of the stuff from scratch, and/or work at a much lower level than machine learning. ML, on the other hand, has a much richer ecosystem of libraries, frameworks and pretrained models that is quite easy to take advantage of. This makes the process of experimentation on something brand new much faster, since you usually just need to cobble some stuff together to get most of what you want running.

* The experiment running time is hugely different in both areas. Machine Learning, specially Deep Learning, has reached a point where it requires increasingly larger ammounts of computation to improve it's results, making it's experiments take hours, days or even weeks to run for projects not more ambitious than this. I've seen my fellow students leave experiments running for unfathomable ammounts of time in quite powerful machines, without reaching state of the art results. This pales in comparison to the time it took me to run my benchmark in user-grade systems, taking less than 10 minutes; the only exception to this was the dedicated AS build time experiment, which took a few hours.

This discrepancy between development and running time could very well compensate

itself, making the total ammount of time dedicated to a project in one field or the other pretty much equivalent.

An unrelated note on a completely personal tone: this work has made me realize how behind are the typical development tools in the graphics industry in comparison to others. The most typical development environment for computer graphics, aside from a few exceptions, is Visual Studio running on Windows. Coming from the Unix-based world, these tools seem sluggish, tight and slow in comparison, as well as hard to learn in some cases. And I use Vim.

Finally, being forced to code in Vulkan is something equivalent to using PyTorch for Deep Learning. The low level and explicit approach of these libraries helps you understand in greater detail how some processes work. In the same way that PyTorch makes you understand how a neural network is trained in comparison to running *model.fit* in Keras, configuring every step of the graphics pipeline in Vulkan makes you understand it in greater depth. The same goes for the modern ray tracing pipeline and both libraries I used here, with OptiX having a (relatively) higher level approach, serving as a gentle introduction. In this way it could be considered an equivalent to OpenGL.

7.1. Future work

There are a few areas in which this project could be expanded upon. These are:

* Testing with dynamic geometry. This would allow for a higher number of tests in the Acceleration Structure Build Time part, since the AS needs to be rebuilt when geometry changes.
* Advanced Ray Tracing implementations. Due to the short time frame for this work and having to do everything almost from scratch, the ray tracers we used for testing were fairly basic. There are tones of features that could be added to them, like reflections, different types of materials, etc.
* Porting to Falcor. Halfway through the development of this project (when most of the code was written), Nvidia released Falcor (Kallweit y cols., 2022). I believe using this framework would have made it much easier to implement a software that traces rays, in comparison to our approach of doing everything from scratch. Additionally, it would have made it much easier to run and iterate on a larger number of experiments, as well as testing other rendering APIs such as DirectX 12.

Referencias

3d basic rendering. (2022, September). Descargado de https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/rasterization-stage

Clarberg, P., Kallweit, S., Kolb, C., Kozlowski, P., He, Y., Wu, L., y Liu, E. (2022, March). *Research Advances Toward Real-Time Path Tracing.* Game Developers Conference (GDC).

File:ray trace diagram.svg. (2022, August). Descargado de https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg

Kallweit, S., Clarberg, P., Kolb, C., Davidovič, T., Yao, K.-H., Foley, T., ... Benty, N. (2022, 8). *The Falcor rendering framework.* Descargado de https://github.com/NVIDIAGameWorks/Falcor (https://github.com/NVIDIAGameWorks/Falcor)

Lefrançois, M.-K. (s.f.). *Nvidia vulkan ray tracing tutorial.* https://vulkan-tutorial.com. (Accessed: 2022-08-03)

Mantle, wikipedia. (2022, September). Descargado de https://en.wikipedia.org/wiki/Mantle_(API)

McGuire, M. (2022, August). *Computer graphics archive.* Descargado de https://casual-effects.com/data

Nviidia. (2022, August). *nvpro core.* Descargado de https://github.com/nvpro-samples/nvpro_core

Overvoorde, A. (s.f.). *Vulkan tutorial.* https://vulkan-tutorial.com. (Accessed: 2022-08-01)

Ray tracing (graphics). (2022, August). Descargado de https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

Sawichi, A. (s.f.). *There is a way to query gpu memory usage in vulkan - use dxgi.* https://www.asawicki.info/news_1695_there_is_a_way_to_query_gpu_memory_usage_in_vulkan_-_use_dxgi. (Accessed: 2022-08-03)

Wald, I. (2022, September). *optix7course.* Descargado de https://github.com/ingowald/optix7course

White, S., Batchelor, D., Sharkey, K., Coulter, D., Kelly, J., Jacobs, M., y Satran, M. (s.f.). *Dxgi overview.* https://docs.microsoft.com/en-us/windows/win32/direct3ddxgi/d3d10-graphics-programming-guide-dxgi. (Accessed: 2022-08-03)