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

«Ací va el resum del TFM. Extensió màxima 2 pàgines.»

Resumen

«Aquí va el resumen del TFM. Extensión máxima 2 páginas.»

Abstract

«Abstract of the Master Project. Maximum length: 2 pages.»

Índice general

1. Introduction **1**

2. Development **3**
- 2.1. Experiment Design 3
- 2.2. Implementation 7
  - 2.2.1. OptiX Renderer 7
  - 2.2.2. Vulkan Renderer 8
    - 2.2.2.1. Rasterized 8
    - 2.2.2.2. Ray Traced 9
  - 2.2.3. Automation and Plotting 11

3. Results **13**
- 3.1. Rasterization Baseline 13
- 3.2. Memory Usage 14
- 3.3. Acceleration Structure Build Time 17
- 3.4. Frame Time 21
- 3.5. Hardware comparison 22

4. Conclusiones **25**
- Referencias 26

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

Chapter 2

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.

2.1. 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.

* **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 engraved 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 ti-

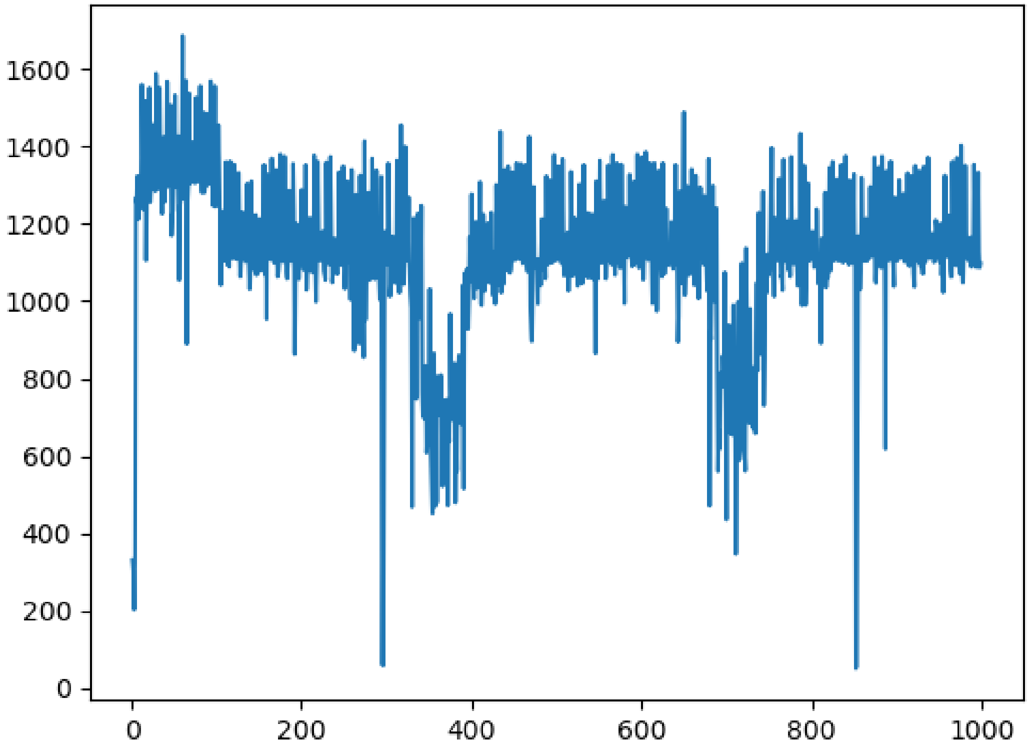


Figura 2.1: Render times over 1000 frames with a frame buffer resolution of 7680x4320, while drawing the model of a viking room with textures. It can be observed how in a very short span of time, the rendering time of the exact same geometry and effects can hugely change.

mes inside a modern Operating System, the render times can vary wildly as well (as we can see in 2.1). 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 execeding 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 2.1

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

In order to better gauge how these scene sizes compare, we have plotted them in graph 2.2.

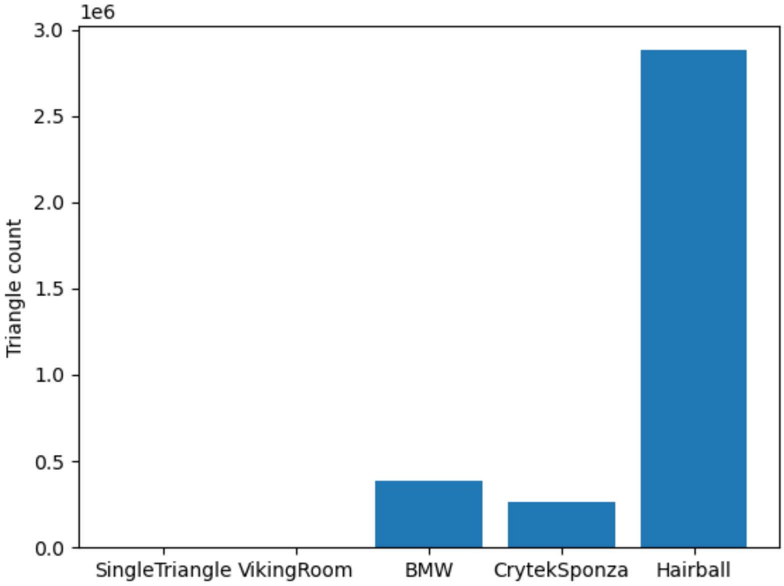


Figura 2.2: Triangle count of each scene used to test both renderers. We can see how there are orders of magnitude of difference every two of them.

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 2.1 and their triangle counts is graphically compared in plot 2.3

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 with GPU accelerated ray tracing, and used it to run the same experiments. This hardware is summarized in table 2.1. 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*.

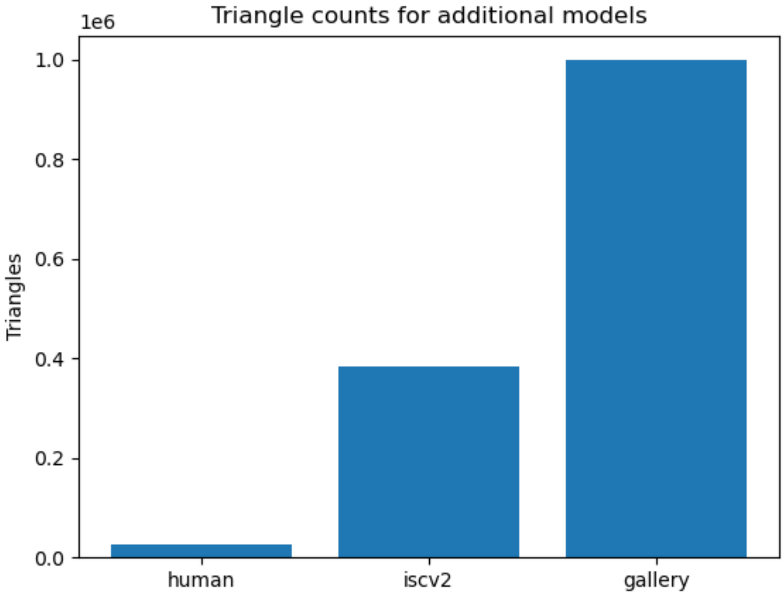


Figura 2.3: Triangle count of the extra scenes used to further test acceleration structure build times.

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

2.2. 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.

2.2.1. 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.^{a}nd "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.

2.2.2. Vulkan Renderer

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, offering higher performance and lower level control than popular graphics APIs at the time such as OpenGL or DirectX 11.

It 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, while usually focusing on traditionally rasterized graphics.

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.

2.2.2.1. 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*.

2.2.2.2. 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.

2.2.3. 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 3

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).

3.1. 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 3.1. 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 3.2. 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.

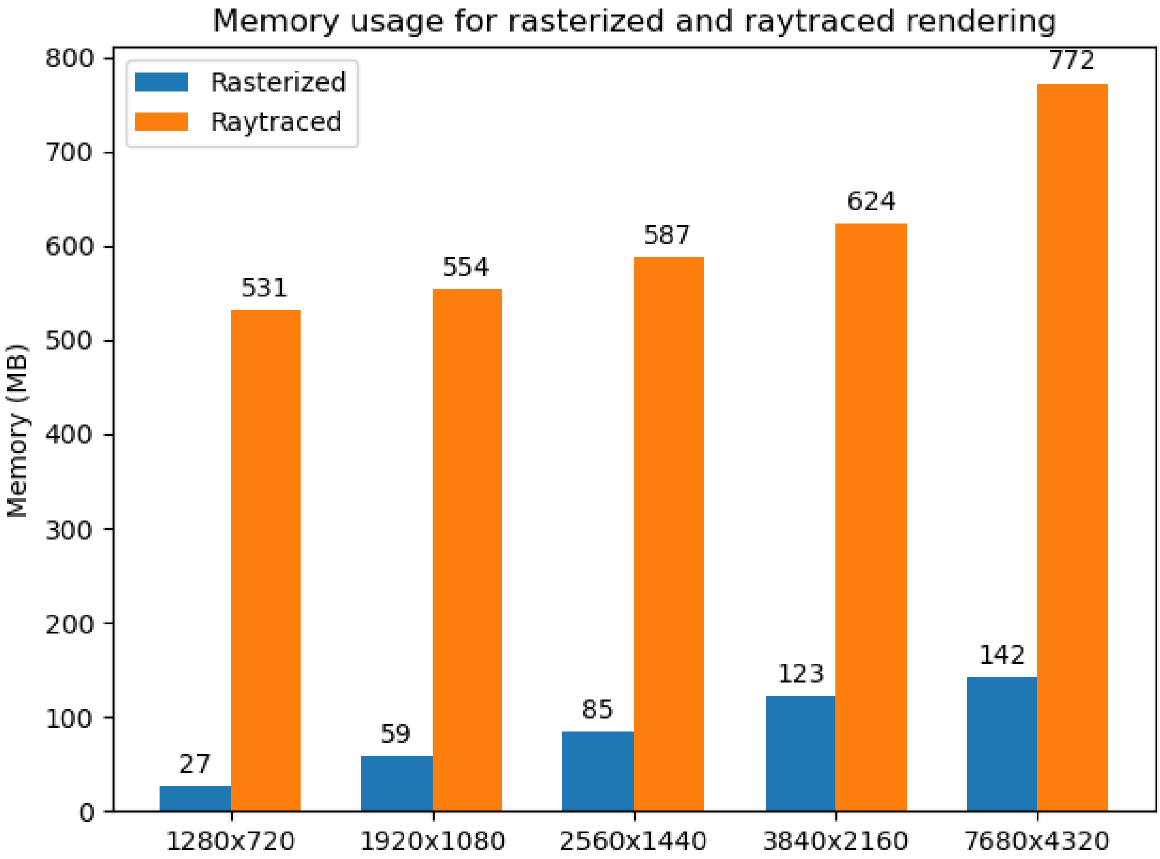


Figura 3.1: GPU memory consumption when rendering the Viking Room 3D model at a range of reasonable resolutions, using ray traced and traditionally rasterized graphics. We can see a significant difference between the two, as well as an expected increase as resolutions get higher.

3.2. 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 3.3.

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.

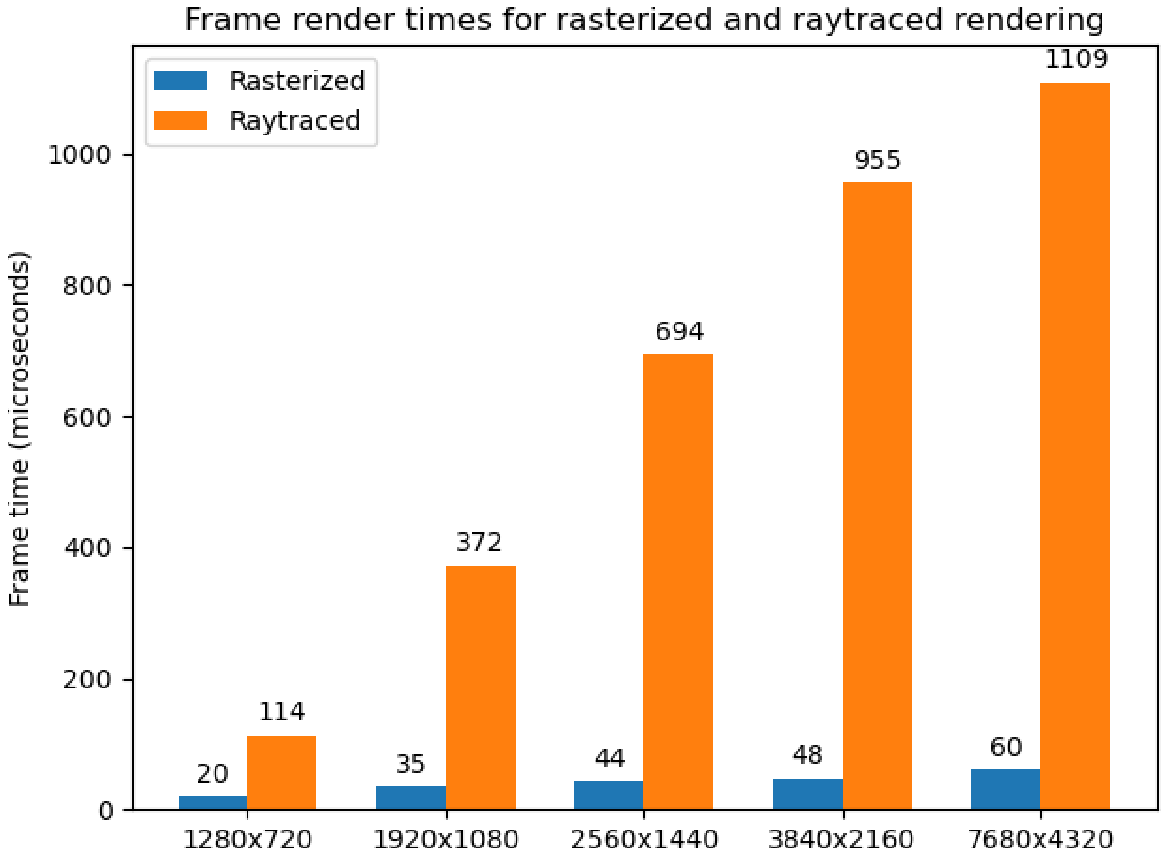
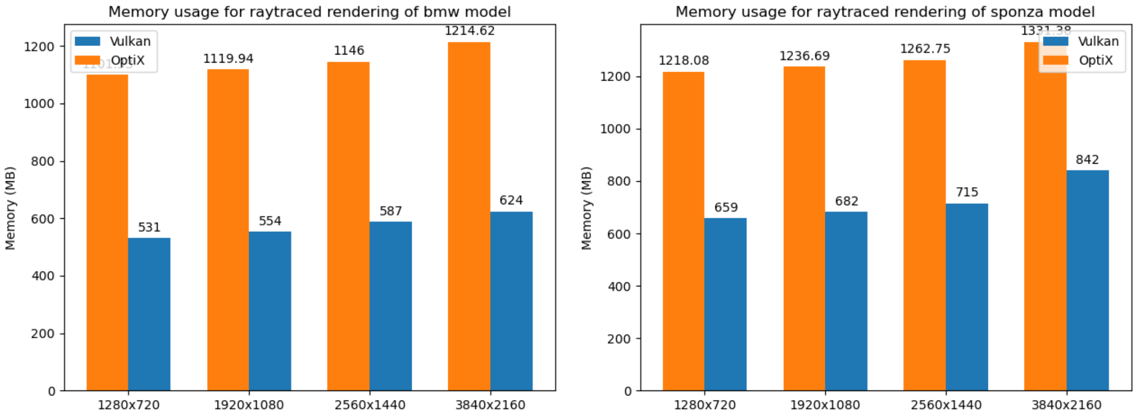
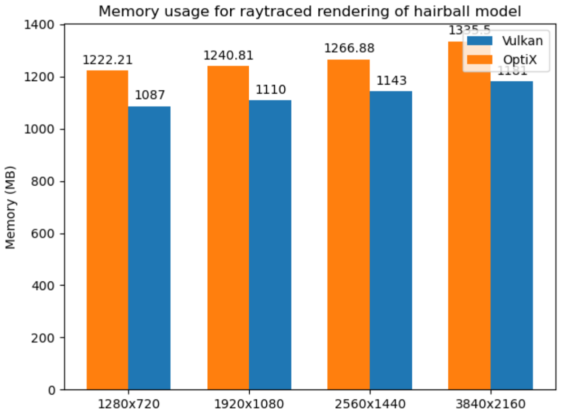


Figura 3.2: Single frame rendering time when drawing the Viking Room 3D model at a range of reasonable resolution, using ray traced and traditionally rasterized graphics. As with the memory consumption graph, we see how the two of them take longer to finish in higher resolutions, while the rasterized renderer greatly outperforms the ray traced one.



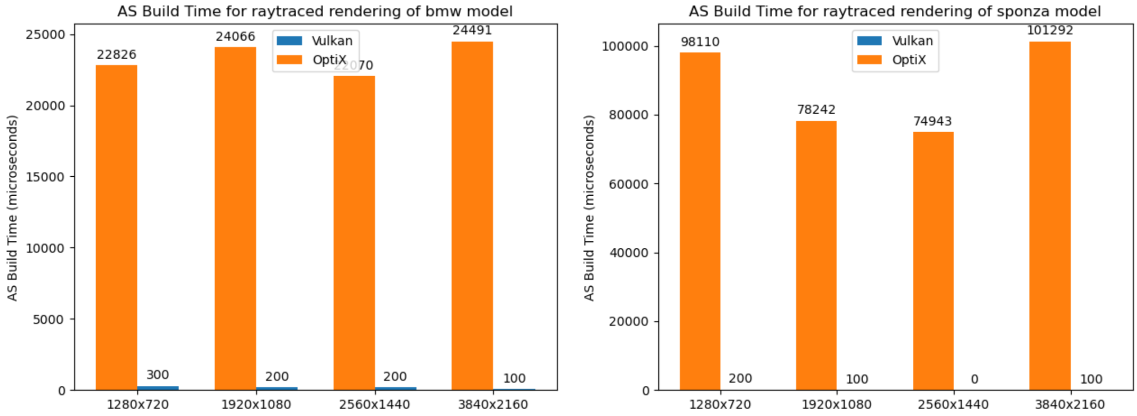
(a) BMW

(b) Sponza



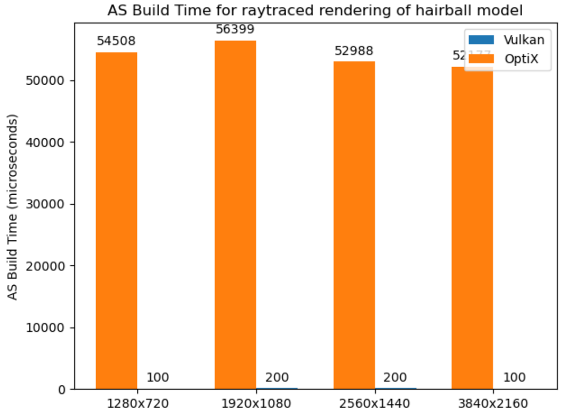
(c) Hairball

Figura 3.3: Memory consumption of OptiX and Vulkan (raytraced) when rendering the 3D models BMW, Sponza and Hairball.



(a) BMW

(b) Sponza



(c) Hairball

Figura 3.4: Acceleration Structure Build Time of OptiX and Vulkan (raytraced) when rendering the 3D models BMW, Sponza and Hairball. Sorted by rendering resolution.

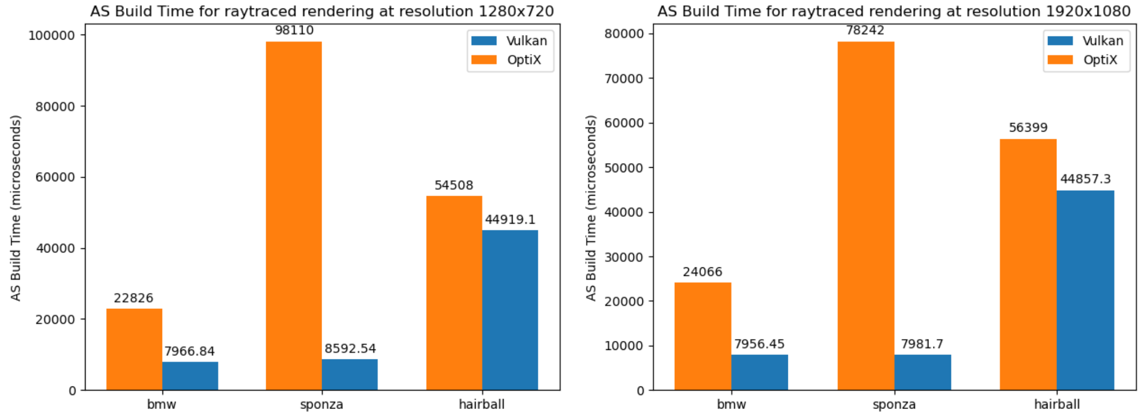
3.3. Acceleration Structure Build Time

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.).

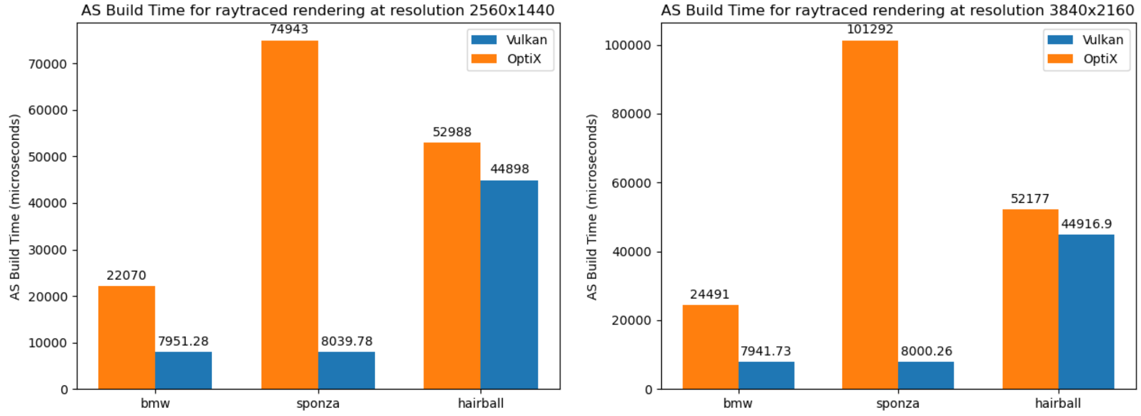
If we plot the Acceleration Structure Building Times in the same way as we did with the memory consumption, we get the graphs at figure 3.4.

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 3.5 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 3.5: 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 3.6. 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 3.7 we see how their build times stack up to one another. In

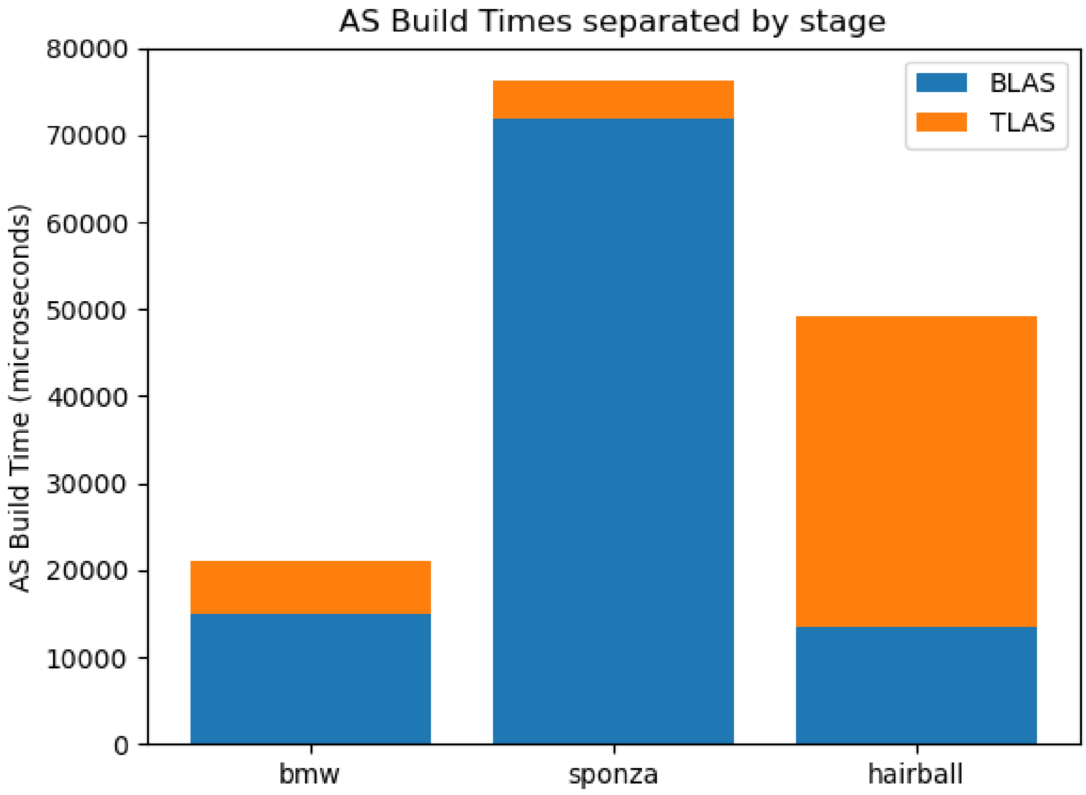


Figura 3.6: Acceleration Structure Build Times for each model, divided between the TLAS and BLAS phases. We see great discrepancies in the distribution of time in each case.

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.

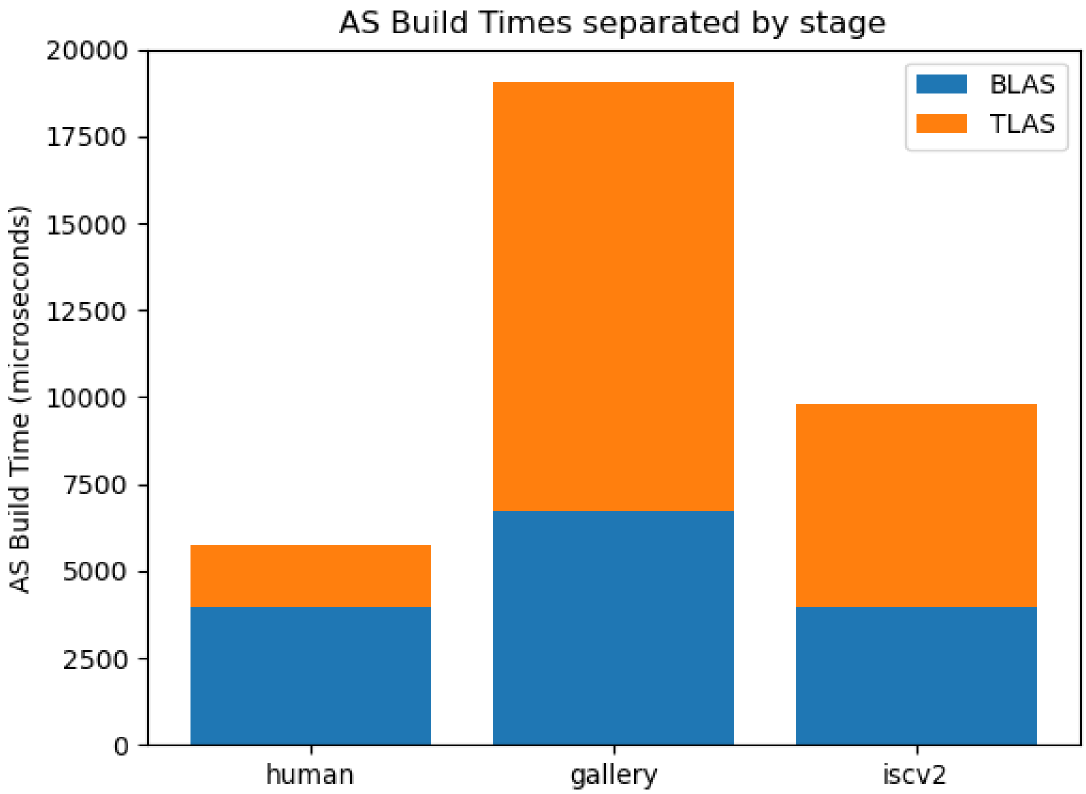


Figura 3.7: Acceleration Structure Build Times divided between the TLAS and BLAS phases for the extra set of models (Human, ISCV2 and Gallery). We see how the presence of textures makes for a longer TLAS build time.

3.4. 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 3.8.

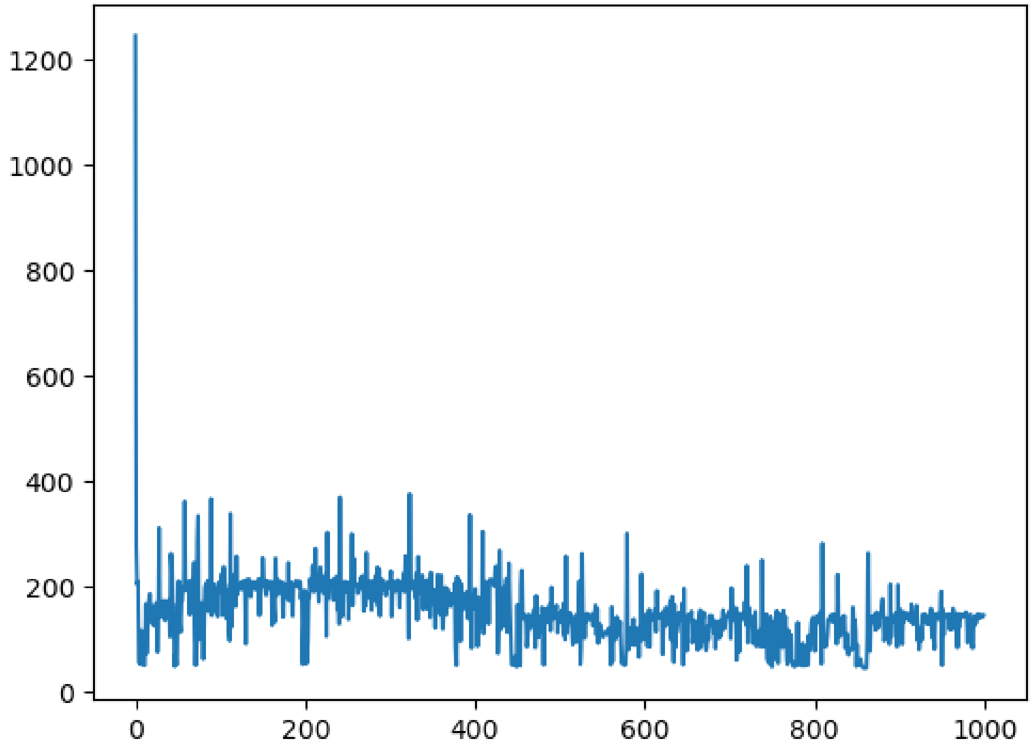


Figura 3.8: Time taken for rendering 1000 frames when drawing the BMW 3D model at 1280x720, using ray tracing. We see how the first frame takes much longer than the rest of them to draw.

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.ªppreciable results. The same graph from **??** results thus in **??**.

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

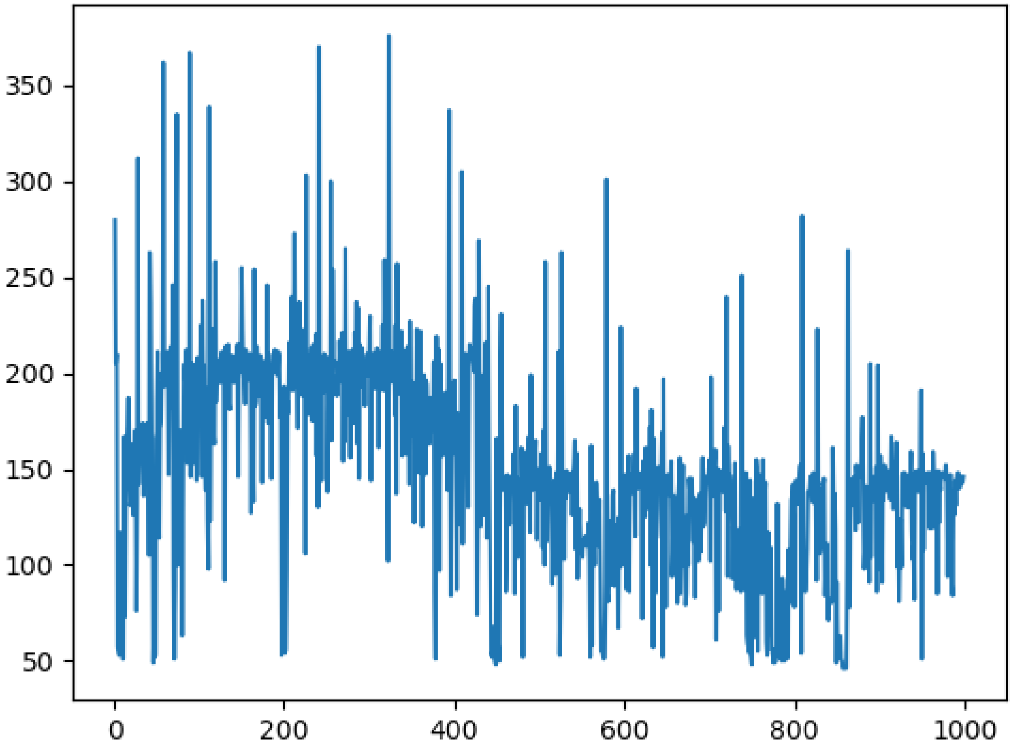


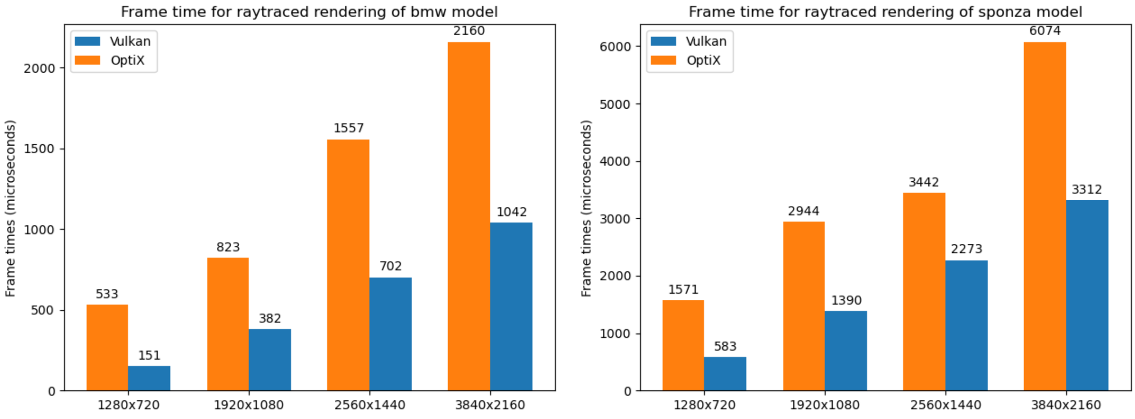
Figura 3.9: Time taken for rendering 1000 frames when drawing the BMW 3D model at 1280x720, using ray tracing. We have removed the first frame time since it took too long to draw in comparison with the rest.

resolutions in figure 3.10. 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.

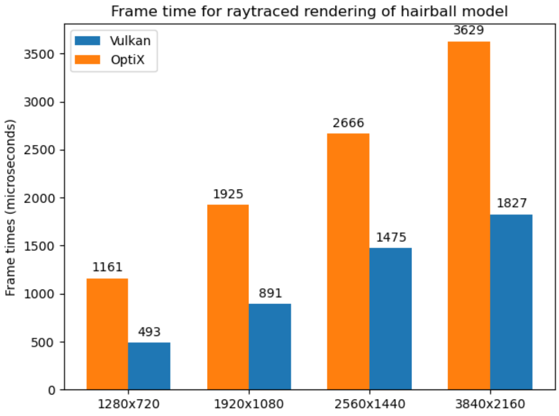
3.5. Hardware comparison

As mentioned in the Development chapter, we will now run 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



(a) BMW

(b) Sponza



(c) Hairball

Figura 3.10: Frame rendering times for each 3D model (BMW, Hairball and Sponza) across both libraries and 4 reasonable resolutions, from 720p to 4k.

inside a virtual machine.

Chapter 4

Conclusiones

En este capítulo se incluyen las conclusiones personales del estudiante sobre el trabajo realizado así como posibles trabajos futuros.

Referencias

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).

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

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)

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)

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)