SAE Institute Genève

Bachelor of Science
CMN6302.1

Particle Rendering in a Raytracing-based Graphics Engine

Simon Canas
simonf.canas@gmail.com
(#12-134266)

Supervisors:
Elias FARHAN
Nicolas SIORAK

September 15, 2021

Summary

HART[1] technology has been on the rise these past years, and more games start integrating it into their engines to provide better looking gameplay. Since it is pretty young and quickly evolving, research is important to provide resources to anyone who needs to learn how to use it.

This thesis starts with recounting a quick history of ray tracing and where it is used in different industries. It will then show examples of usage in the video games industry.

Afterwards, a solution is proposed to provide more resources and research into a less explored part of HART: particle rendering. An explanation of the implementation will be presented along with the different problems and challenges encountered along the way.

The results proved to be not very reliable due to lack of testing, but were satisfactory when compared to the evaluation criterias and other factors. The solution ends at the state, however, where it cannot be used in a game engine as is.

Keywords: Programming, C++, cplusplus, Vulkan, Graphics, Raytracing, Particles

[1] Hardware Accelerated Ray Tracing

Preface

This work was carried out as part of a game programming bachelor at the SAE Institute in Geneva. It was done with the objectives to demonstrate specific knowledge in ray tracing and graphics programming in general.

New technologies are being created in the realm of graphics rendering and special attention and care needs to be put into experimenting with every facets of it and documenting the findings.

Acknowledgment

This work could not have been possible without the help of these people:

- Elias Farhan, my teacher who helped me keep a steady pace on the project, as well as helping with some of my programming issues.
- Nicolas Siorak, for helping me structuring and refining this thesis into the state it is now.
- Fred Dubouchet, for giving more insight into low-level programming, and helping with graphics-related problems.
- Sascha Willems, for his extensive work on Vulkan and Ray tracing on his Github page.
- Nicolas Schneider, my senior whom I used the code from the private engine he made with his classmates during his last year to help me build the particle system.

As the author, Simon Canas, I testify to have personally done and written this thesis.

I also certify that I have not resorted to plagiarism and have conscientiously and clearly mentioned all borrowing from others.

Contents

Introduction

Problems and Objectives

Video games have always tried to push the graphical capabilities of the hardware as much as possible in order to get the most out of it, graphical- or computational-wise. As developers try to get the most out of the limitations offered by the hardware, the technologies themselves also evolve to give new possibilities and limits to create even more complex and beautiful games than before. Nowadays, most game studios try to push more the graphical capabilities of their games as good graphics is an attractive marketing argument.

Due to this demand, NVIDIA created the RTX GPU series which gave way to HART. This new technology allowed developers to get access to ray tracing more easily and create much more realistic visuals for their games. Due to these advantages, most modern consoles also support HART.

However, HART being still a young field, few resources are available and also become obsolete quite fast as the hardware and APIs supporting it improve. As time goes on, and the available APIs have settled on their implementation, more resources will become available, but further research will be difficult, which is why creating additional data now is important.

One field that isn't very much explored is particle systems in a HART setting. Rendering particles is a very important part of any game engine as games tend to use them extensively for improving visual quality, be it in 2D or 3D games. This research aims at creating and implementing a particle system in a ray tracing-based graphics engine, as well as finding ways to make it run as smoothly as possible in a game engine setting. The question then becomes: How can we render particles in a ray tracing based graphics engine?

The solution should be able to run with satisfying performances, without any slowdown as to not impact the performances of the rest of the game.

Plan

In order to answer this question, this thesis will be divided into 5 parts, each focusing on different aspects of the project.

The first part will explain what ray tracing is and how a graphics engine using this technique works on a surface, and deeper level.

The second part will briefly take a qualitative look over the different engines that

use ray tracing and the work of important personalities that I based my thesis and research on.

The third section will go over the various tests used during the project and the environment they were done in.

The fourth part will cover the implementation and the choices made along its development, along with the problems encountered.

Lastly, this document will analyze the results of the tests, assess whether such an implementation is useful for the industry as a whole and define the use cases in which such a solution is the most efficient.

Chapter I: Ray Tracing in video games

A brief history of ray tracing

Ray tracing as a concept already existed for a very long time, as early as the 16th century by a man named Albrecht Dürer, who is often credited for its invention (Hofmann, 1990).

One of the first attempts of computer-generated raytraced pictures can be to Scott Roth. He created in 1976 a simple flip book animation using a pinhole camera model (Roth, 1982).

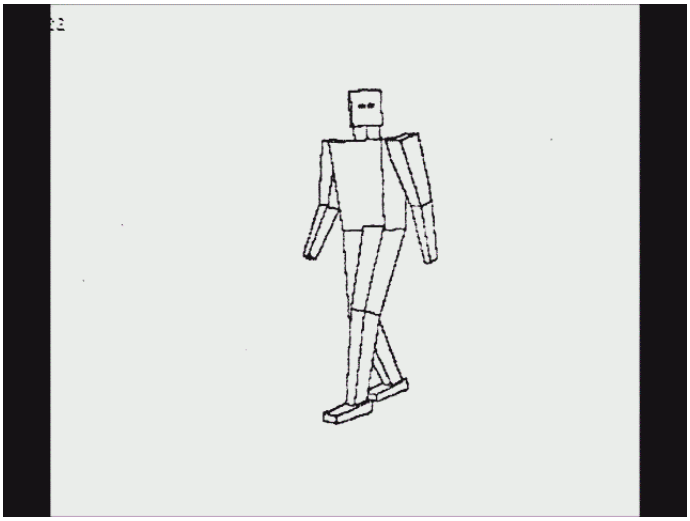

Figure 1: The animation created by Scott Roth. From https://commons.wikimedia.org/wiki/File:Flip_Book_Movie_v2.gif

Ray tracing was at that point, however, still too computationally expensive to be used as an alternative rendering technique. In fact, Goldstein and Nagel of MAGI needed to use a CDC 6000 in order to render a short 30 seconds movie, the world's fastest computer up until 1969 (Goldstein and Nagel, 1971), which was the size of several human-sized closets.

During the 2000s, due to the advancements made in computer hardware, ray tracing became more and more feasible in consumer-sized machines, and interest in the technology started to grow. However, it wasn't until 2013 that the first major full-length ray-traced movie, *Monsters University* (2013), came out (Smith, 2013).

Figure 2: A screenshot of *Monsters University* (2013) by Pixar.

At that point, ray tracing is almost exclusively used in movies and animations, as well as static image rendering since you can just let the computer render the scene and walk away until it finishes. There are very few usages of ray racing in real-time rendering as most consumer machines still couldn't handle the technique very well.

However, in 2018, NVIDIA brings a new technology to the table: their brand new RTX 2080, the first card in the RTX series (Walton, 2018). On top of having better performance than their previous GPUs, the RTX 2080 is the first card to enable the use of HART, allowing drastic improvements in the speed of 3D ray-traced image rendering. Shortly after, Microsoft added HART capabilities to DirectX 12, allowing Windows computers to use the new technology (Team D3D, 2018).

Afterwards, in 2020, Khronos, the creators of graphics API OpenGL, created their own implementation of HART in their most recent API Vulkan, allowing it to be used in a larger number of programs and games (Ridley, 2020).

How ray tracing works

There are 4 major elements that appears when talking about ray tracing:

- The screen, where the final picture will be displayed.
- A scene, which contains all the elements that can be rendered.
- A camera, which will determine from where you see the scene.
- One or more light sources, which will indicate how the light will behave in a scene.

At the start of each frame, the camera will send one, or multiple depending on

the image quality wanted, ray towards each pixel of the screen. Those rays will then bounce around the objects in the scene a set number of times until they bounce back directly to the light source. That last bounce is also called a "Shadow Ray". If the shadow ray encounters an object on the way, the pixel will be considered to be in shadows.

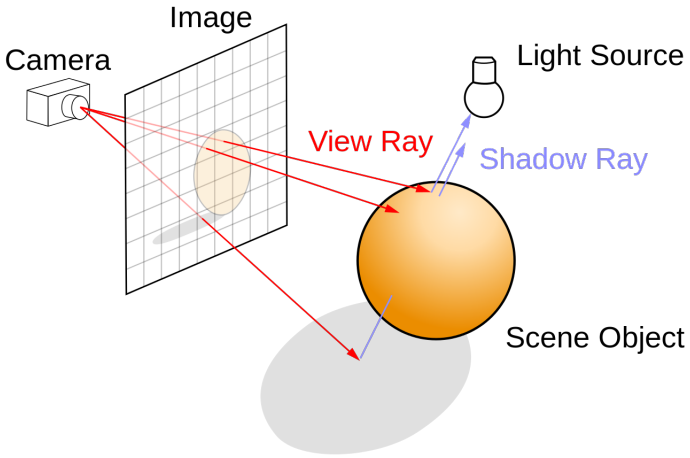

Figure 3: A diagram showing how ray tracing works. From https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg

Using multiple rays for a single pixel can be useful for effects like antialiasing or reducing the noise in the image, an undesirable side-product of ray tracing which can also be countered with denoising algorithms. Of course, using more than one ray per pixel will have a bigger impact on performances, so it must be used responsibly in real time rendering. Another consequence of this technique is that it performs better on lower screen resolutions, since the lower the resolution the less pixels we need to shoot rays from.

This way of rendering mimics how light works in real life, allowing the resulting image to look much more realistic. The biggest difference with real light is that ray tracing treats it only like a particle, bouncing around every time it hits and object, which is only half the truth, but thankfully it only impacts minor effects that can easily be reproduced in other ways.

Despite ray tracing requiring more performance than rasterization, some effects become much cheaper and simpler to render. A good example is shadows: whereas in rasterization we need to render the whole scene twice, everything happens in a single pass in ray tracing and even allows for easy soft shadows.

Where Ray tracing is used

In 3D static image rendering

Rendering a static 3D scene is one of the earliest and most common uses of ray tracing. Even though ray tracing takes a lot of resources to render a single image, this is mostly just a concern when talking about real-time rendering. In fact, when your only concern is to render the most realistic looking picture possible, it doesn't matter if the picture takes a couple of hours to render, as long as the end product looks the way you want.

Since ray tracing, when used well, can produce life-like pictures, it has been used a lot in 3D architectural visualization, since, on top of being a good way to measure realism, it helps companies save money by being able to make prototypes more easily, by saving the cost of installation in order to make previews or by removing imperfections to give a clean image. Some companies, like *Vrender Company* or *The Real Estate Rendering Company* even specialize in this field, making previews for real estate companies.

Figure 4: *The Warm Apartment* by Hendra Darusman. Made in Blender.

The technique keeps on improving, and with the rise of HART, it would be possible to create images of comparable quality in real time.

In cinema and animation

As cited earlier, 3D animated movies weren't fully rendered using ray tracing up until *Monsters University* released in 2013. The reason is simply that before then rendering a single frame would take too much time and, consequently, wasn't financially viable for full-length cinema releases.

Outside of the realm of 3D animated movies, ray tracing is also used a lot for

rendering VFX and CG in live-action movies. Those sort of films needing the highest level of realism in order to not distract the viewer by integrating artificial elements during the post-process phase, ray tracing is a very useful rendering technique and allows those added effects to blend seamlessly with the real life elements (NVIDIA GeForce, 2018).

HART would allow for faster rendering of ray traced movies and enable artists to have a better preview of how a scene looks without having to render the whole scene every time. It would also allow for much higher quality cutscenes in games as Epic Games has shown with a tech demo in 2018 how HART could look in a cinematic environment.

Figure 5: Screenshot of the RTX tech demo shown at the GDC 2018.

In video games

Even though HART has enabled many games to use ray tracing, the technique was already used in games before. Of course, unlike with RTX enabled cards, games needed a hybrid rendering engine in order to ensure stable and comfortable performances. The scene was rendered with rasterization as usual, and the outputs produced were then used to apply some effects using ray tracing, which include shadows, reflections, or baking lightmaps (Morgan and Pranckevičius, 2014).

Thankfully, with HART, games can now more easily use ray tracing, as well as reduce the performance impact associated with ray tracing. It will, however, take time before fully ray traced games become the norm as most consumers don't necessarily have HART compatible cards.

There do, however, exist fully ray traced games such as *Q2VKPT*, a remake of Quake II released in 2019, that show the potential of HART. The project was started

in 2016 and was originally meant to use OpenGL, but later on switched to Vulkan in order to use its HART API (Palumbo, 2019).

Figure 6: Screenshot of *Q2VKPT*.

Summary of findings

Ray tracing is an old technique that has been around for multiple decades in the graphic rendering field, and that has been growing steadily during the late 2010s with the rise of HART.

It was a very computationally expensive technique up until the creation of the RTX series card in 2018 by NVIDIA after which multiple fields were able to benefit from HART in order to render better looking pictures.

Another advantage of HART was speeding up existing pipelines. In fact, being able to render animations using ray tracing faster can boost the production process and allow for more polished work using the time saved with this technology.

Chapter 2: Qualitative study

In order to stay up to date with the latest development in HART as well as learning how to implement it, it is essential to find already existing implementations of this technology to guide the project.

Unreal Engine

Epic Games are one of the main contributors regarding the research towards HART as they have been steadily working on implementing it in their Unreal Engine to push its graphical capabilities even further as by Figure 7.

Figure 7: Screenshot of a real time ray tracing sample made by Archviz using Unreal Engine 4.24. Available at: https://www.youtube.com/watch?v=XBoImPlIA1M

Unreal Engine 5 also unveiled the new Lumen dynamic global illumination and reflections system which replaces older techniques like SSGI and DFAO to create more natural looking lighting and reflections (Kim, 2020). This new version is currently still in early access and their technology will keep on improving further as time goes on, making them a reference on everything relating to real time ray tracing.

Sascha Willems Vulkan samples

Sascha Willems is a programmer well versed into graphics programming and after making multiple contributions to the Vulkan API in 2016 as a member of the Vulkan Advisory Panel, he has become a Khronos individual contributor member in late 2020.

He has since 2015 been keeping a Github page on which he regularly updates and posts new samples regarding the Vulkan API, including HART. His work serves

as a very useful learning resource and has been very useful in guiding and creating this thesis.

He also keeps up a blog on which he posts tutorials and other news relating to Vulkan. Thus, it seemed useful to rely on his samples.

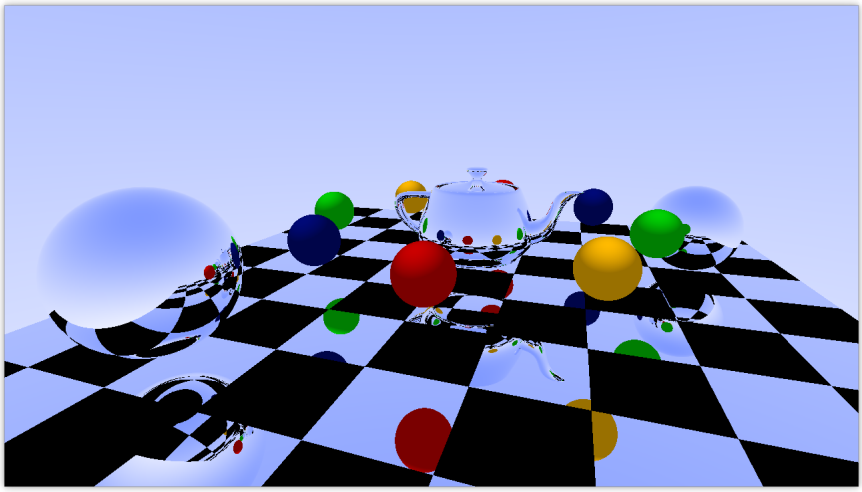

Figure 8: Screenshot of his sample on reflections with HART. Available at https://github.com/SaschaWillems/Vulkan.

Chapter 3: Setting up the test environment

This part focuses on describing and defining the different tests that are going to be done. These tests will serve to check the quality and limits of the solution proposed to the problem.

Focus on performance

Since this project needs to work in real time, the execution time of the implementation is the most critical part to measure in each of the tests. While the aim of this work is to prioritize performances, it is also important to balance quality since the entire purpose of using HART is to get better results than rasterization quality-wise.

To evaluate the performances, those points will need to be measured and tested:

- The average number of frames per second; for video games this is one of the better ways to measure performance. We do not want this number to go any lower than 500 as we would need extra performances to integrate the other elements of the game. This is an arbitrary limit, but there needs to be one.
- The number of particles and emitters should have an impact on the FPS as it will put more load on the GPU, so it will need to be tested.
- Even though the impact should be slim, the tests should be able to measure the impact of the different settings of the particle emitters.

FPS and execution time

Since this project is graphics-based, we will need to measure how many frames are generated by second as well as the general execution time. The computational part on the CPU will also be monitored to determine if any part of the implementation is going to bottleneck the performances.

Number of particles

The goal number of particles would be around 10'000 particles across all emitters, and detect if the program starts to slow down too much with smaller numbers. Being able to render this many particles would be beneficial as it would allow less restrictions on the use of particles.

Ratio between emitters and particles

Another factor to take into account is if having multiple emitters will impact the performances on a noticeable scale. It is important to make sure that having several emitters doesn't slow down the program too much as it would make the implementation non-viable for practical use.

Environment description

Two factors will impact the performances: the engine and computer used.

NekoEngine

This project uses a personal fork of the NekoEngine, a custom 3D engine. It already has a Vulkan engine in rasterization and a ray tracing module will be added during the implementation. Using a custom engine like this reduces unknown overheads and better monitor performances as well as working with a familiar environment.

Hardware

The machine used for the tests is a desktop computer. The most important part is the GPU, which will take the biggest load when rendering all of the particles and tracing the rays. Since RTX or similar is necessary for HART, the GPU used is a GeForce RTX 3070.

The processor is an Intel Core i7-4790K with 4 cores, and may impact the performances when it comes to computing some values for the shaders in the code.

Tests environment

The project will not focus on any particular genre of game as it is generally irrelevant when developing a graphics engine.

The implementation will also be as isolated as possible from the other elements of the engine like the ECS as it would only bottleneck the performances with unnecessary calculations.

As we would only need to know the performance impact from the particle system itself, we will be using an "empty world" where the only existing objects will be one or multiple emitters scattered around it, preventing as much interference as possible.

Overview

The primary sectors to test are simply the different settings of the emitters as well as their numbers. There will be multiple options to tweak in order to measure their impact on the performances.

The number of particles will be very important as well since it means more elements to render on screen. The more particles there are, the bigger the impact and the tests will help determine the upper limit to set and whether this limit is satisfactory or not. The target limit would be around 10'000 particles across all emitters, which would put less restrictions on the use of particles.

The unique and isolated environment will help test performances as we need as little interference as possible. The graphics engine will be at the core of the engine and if this part isn't optimized correctly, it could slow the whole game down.

Chapter 4: Implementation and results

The implementation of the solution devised with the help of research aims to answer modern problems using modern techniques. HART still being a young field, it is also important to use the most recent tools available to integrate it.

Development of the engine

The rendering engine has been developed inside the already existing NekoEngine game engine, using the already made Vulkan engine as a base for implementing the solution. The NekoEngine has been modified to be as minimal as possible in order to have the minimum amount of subsystems that can interfere with the results inside of the rendering loop.

Since the purpose of the implementation is only to measure the performance of the particle system, multiple portions of the solution has been hard coded to facilitate its creation.

Particle system

As cited in the "Acknowledgment" section, the solution borrows parts of the code from Nicolas Schneider's private engine he made during his last year at the SAE Institute with his classmates. This helped rrate the development of the solution, and only the computing part was used as the rendering one had to be adapted to work with ray tracing and the current implementation.

Since the focus is mostly on the rendering part, it seemed better to borrow his work as to be able to work on the ray tracing part as fast as possible, though some changes had to be made due to different engine infrastructures.

Initialization of the ray tracing pipeline

The first thing that had to be done was to modify the existing engine in order to enable ray tracing. The initialization code had to be adjusted to include some required extensions for ray tracing, as well as slightly changing the way some elements are created.

Acceleration layers

The RTX technology uses what's called "acceleration layers" in order to efficiently parallelize the ray tracing work. The basic structure consists of one TLAS[2] and multiple BLASs[3]. The BLAS contains the geometry data of any singular object that has been loaded into memory, while the TLAS contains the instance data for each bottom layer, telling the GPU where each of the objects instances are located in the scene.

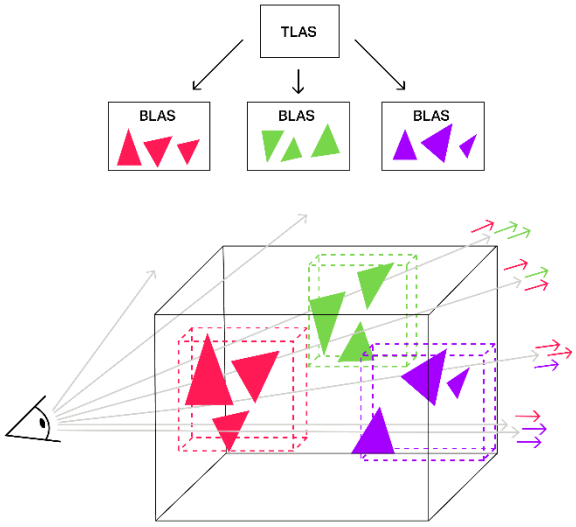

Figure 9: Diagram of how acceleration layers works. From: https://www.ednasia.com/what-is-ray-tracing-and-how-is-it-enabling-real-time-3d-graphics/

As described before, this way of passing data for rendering is quite different from how it is normally done in rasterization, but what this means is that for every model loaded, a BLAS must be created with it, which includes the quad for the particles.

New Shaders

Ray tracing also requires different shaders than in rasterization. In rasterization, you would only need, at minimum, a vertex and a fragment shader to render a basic object, while in ray tracing you would need several:

- A "raygen" shader which will tell the program how the rays should be emitted.
- A "raymiss" shader which will tell the program what to display when the ray doesn't hit anything.

[2]Top Level Acceleration Structure
[3]Bottom Level Acceleration Structure

- A "rayhit" shader which will tell the program what to display when the ray does hit something.

The "raygen" and "raymiss" shaders are pretty straightforward. In the implementation, in order to simplify its creation, the "raygen" shader will tell the program to emit the rays from the camera to the center of each pixel. It will then call $traceRayEXT()$ to emit the ray then call $imageStore()$ right after to store the results, here called "$hitValue$", in the resulting image.

The "raymiss" shader is even simpler, as it only assigns a uniform color to the "$hitValue$". As shadows are used in this implementation, there is also a second "raymiss" shader that disables the "$shadowed$" flag for that ray, preventing it from making the resulting output darker than it should be.

The "rayhit" shader, in the other hand, has much more going on. Since we need the normal of the face instead of the once from the vertex, we first need to interpolate the normal vectors from each of the three vertices of the triangle. In order to do that, The entire array of vertices and indices is directly passed to the shader, which allows us to access any vertex that we might need. The shader gives us access to the "$gl_PrimitiveID$" to determine which face we are working with at the time, allowing us to retrieve the correct vertices and interpolate the normal. The shader then does some simple light calculations, before generating the shadow ray with another call to $traceRayEXT$. If the shadow ray hits something then the shader will just darken the results from "$hitValue$".

The shaders also requires different value to be passed to them:

- The top level acceleration structure, which contains the instance information.
- The image to store the result in.
- The inverse of the view and projection matrices, to calculate the origin point and direction of the rays.
- The position of the light.
- The size of the vertex structure, in order to retrieve the values stored in the vertices array.
- The vertices data, which consist of a single array of 4 dimensional vectors.
- The indices data.

Challenges encountered

Color Bug

Correctly implementing shaders was one of the trickiest part of the implementation as they are very different from how it is done in rasterization. As described earlier, there is a large amount of data to be passed and the lack of debugging tools for ray tracing programs made it difficult to troubleshoot.

Hopefully, this part went without many issues except for one strange problem when the program was first launched:

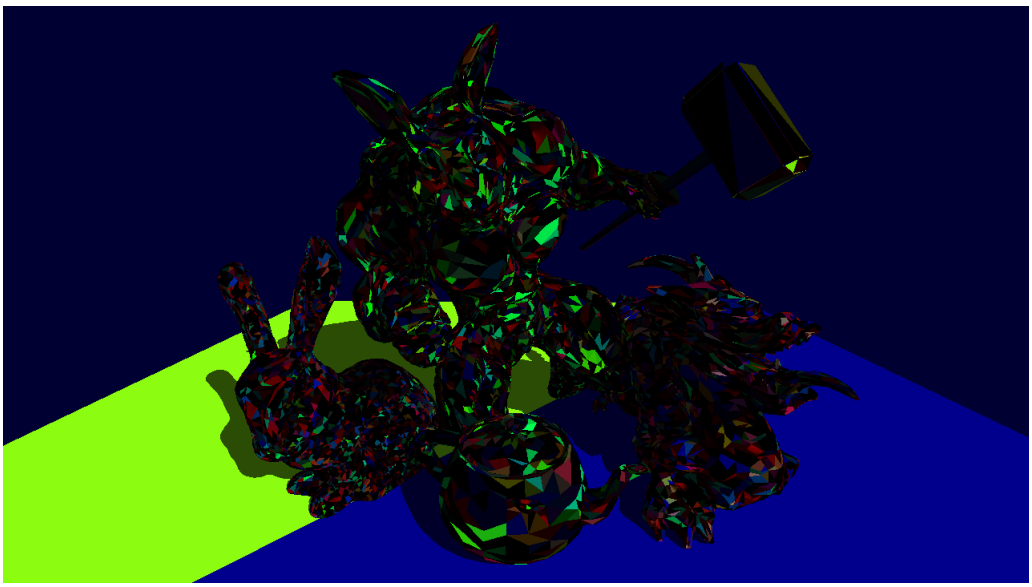

Figure 10: The bug encounter when first launching the program with a test model.

As can be seen on Figure 10, the colors are completely wrong. The model is supposed to be white, so this result was definitely unexpected. As it turns out, this was due to the way the vertices data is extracted in the "rayhit" shader. Since the vertices array is made of 4 dimensional vectors of floats, data can only be extract by chunks of 16 bytes, which also mean the vertex structure size needed to be a multiple of 16 in order to retrieve the data correctly. It turns out the vertex structure was only 56 bytes in size, which made the data align in an unfavorable manner. After adding a new member for padding, the program worked correctly:

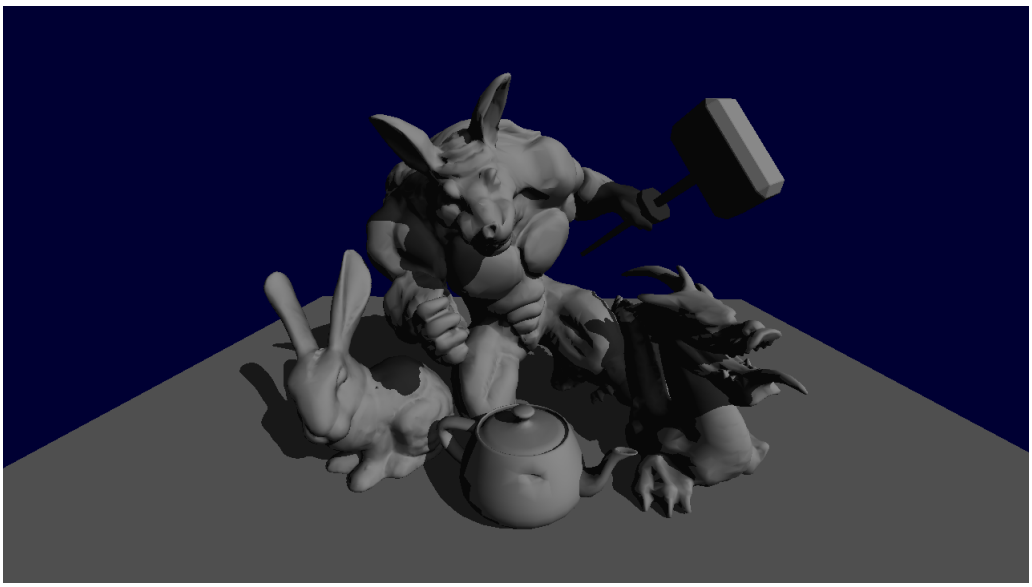

Figure 11: The bug encounter when first launching the program with a test model.

Multiple objects

Initially, one test required multiple emitters to be scattered around the world, however it became clear near the testing phase that this would be impossible with the current implementation.

In fact, as cited earlier, many elements of the solution have been hard coded to save time, but this also makes it very inflexible. As it was first designed to only render one object and its instances, and since each emitter is counted as a single object, it proved too laborious to rewrite the code to account for multiple emitters, which made this test impossible to do.

The most important test, with 10'000 particles, is, however, still possible.

Unstable performances

Quite early during the engine development, something quite surprising happened: the amount of FPS kept fluctuating wildly. The amount of instances didn't seem to impact the performances by a lot, however, as only using the scene with the test model shown in Figure 11 shows this phenomena as well.

The difference can be as small as a couple of FPS or as big as a hundred; it is inconsistent and didn't seem to follow any sort of pattern or react to certain actions like moving the camera. The particle system isn't at fault either since it happened even before it was implemented. This made it quite hard to discern the exact reason of this behavior.

In the end, the reason couldn't be identified and it was decided best to move

on, instead of wasting time upon a problem that didn't seem to find any progress of being resolved.

Results

Before focusing on the various results, it is important to quickly recall the evaluation criteria.

Review of criterias

The initial goal was to be able to render 10'000 particles with an average margin of 500 FPS. Multiple emitters were to used in order to measure their impact on performances. The focus here isn't on quality but on quantity, and the performance hit this solution produces.

Empty world

It was decided during the test section that the world shouldn't contain any other object than the emitters. As explained during the "Multiple Objects" section, the current implementation couldn't support multiple emitters, so only one could be used.

This didn't prove to be much of an issue, however, since using multiple different emitters probably wouldn't have impacted performances in any significant manner.

10'000 particles

Due to other setbacks that were explained earlier, only the 10'000 particle test with a single emitter was able to be performed.

The goal of 500 FPS that was put in place proved to be, quite early in fact, to be too optimistic of a goal to reach as even trying to render the simple model in Figure 11 brought the counter to under 400.

However, despite this initial failure, the program seemed to handling the large amount of particles quite well, since the implementation was sitting between 200 and 300 FPS, due to instability mentioned in the "Unstable performances" section.

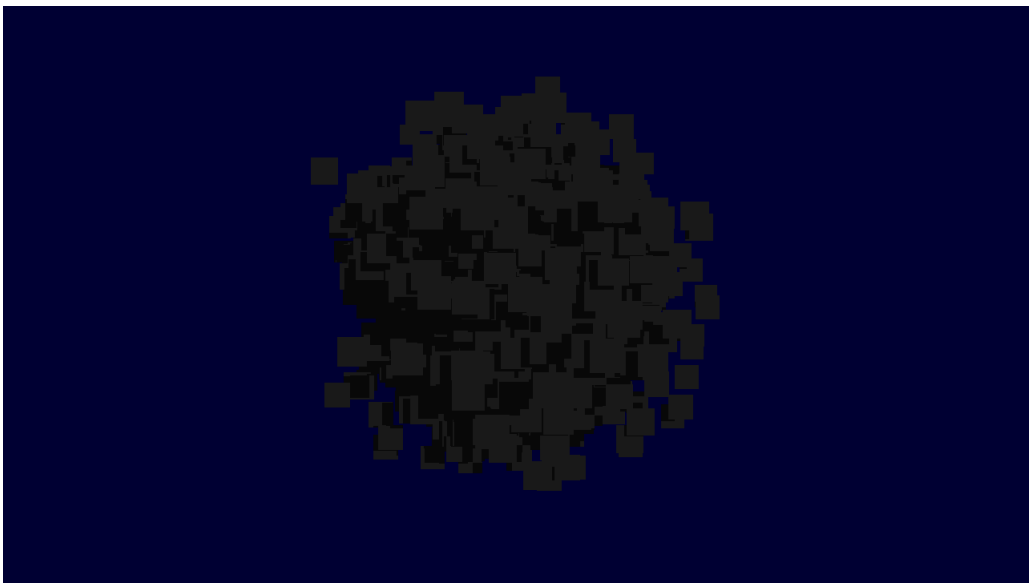

Figure 12: The final result with 10'000 particles.

Review of the implementation

Much of the work done for this thesis was spent on implementing the ray tracing engine. Even though understanding how the Vulkan HART API worked was an imposing initial bump on the road, in the end, a working implementation was able to be produced, despite some setbacks.

The engine was able to handle the expected 10'000 particles, even though other issues probably restrained the performances more than intended.

Chapter 5: Results analysis and conclusion

Initially, the results should have included multiple variations of the tests with multiple settings, however, due to time constraints, only one test with default settings was able to be performed. This however didn't raise any critical issues as the most important test could be performed without problem.

Results analysis

First of, it is important to remind that the initial goal of 10'000 particles was able to be reached with FPS ranging from 200 at the lowest and 300 at the highest. Initially, the results seemed disappointing as it didn't meet the, very optimistic, 500 FPS goal. However, since the drop of performance from rendering one object with one instance and one object with 10'000 instances wasn't very big, the results became acceptable.

The lack of testing makes those results, however, not very reliable and the fluctuation of performance means it is too unreliable to be used as is.

Conclusion

In this thesis, the objective was to create a particle system inside a ray tracing graphics engine using the HART technology. Most of the research was spent looking for resources on how to use HART inside Vulkan's API. The examples found during that process helped guide the project toward the state it is now, and made it possible to produce a working solution.

A big part of the time spent for this thesis was spent on implementing the actual ray tracing graphics engine. This technique being quite different from rasterization in multiple aspects, the time taken to learn and implement the engine took much longer that it would otherwise. New shaders meant that new ones needed to be created from scratch, and the existance of BLASs and TLASs required some rewrites and modification in some part of the base engine.

In the end, the solution was able to reach its 10'000 particles rendered on screen goal, at, however, worse performances than hoped. The results were nonetheless deemed satisfactory due to other factors. Most of the planned tests couldn't be performed, making the results less reliable that they could've been.

Going Further

Even though the solution can be seen as operational, several points need to be changed and tweaked.

First of, the problem of multiple objects is one that should be fixed as soon as possible, as having an engine that can't render more than one object and its instance would be useless in any practical scenario, preventing it to be used in the context of an actual game.

Secondly, the unstable performances problem needs to be quickly tackled as well, since we don't want the performance of the other elements in the game to suffer because of this one singular part.

Finally, modifications needs to be done on the graphical side as well since, as soon on Figure 12, the particles are presently only grey quads moving in the world, which isn't very appealing. New parameters need to be added, most notably the ability to add textures to those quads and allow transparency.

If some wants to look at the code and analyze the solution in order to, hopefully, learn from it or even improve it, please go the public repository available here: https://github.com/CanasSimon/NekoGameEngine/tree/ray_tracing

Bibliography

D3D Team, 2021. Announcing Microsoft DirectX Raytracing!. [online] DirectX Developer Blog. Available at: <https://devblogs.microsoft.com/directx/announcing-microsoft-directx-raytracing [Accessed 28 July 2021].

Goldstein, R. and Nagel, R., 1971. 3-D Visual simulation. SIMULATION, 16(1), pp.25-31.

Hofmann, G., 1990. Who invented ray tracing?. The Visual Computer, 6(3), pp.120-124.

Kim, M., 2020. Unreal Engine 5 Announced With Gorgeous PS5 Demo - IGN. [online] IGN. Available at: <https://www.ign.com/articles/ps5-unreal-engine-5-demo> [Accessed 28 July 2021].

Morgan G. and Pranckevičius A., 2014. Practical Techniques for Ray Tracing in Games. [online]. Available at: <http://cdn.imgtec.com/sdk-presentations/gdc2014_practicalTechniquesForRayTracingInGames.pdf> [Accessed 18 August 2021].

NvidiaGameWorks, 2018. Raytracing and Denoising. [online]. Available at: <https://www.youtube.com/watch?v=7uPLAC5uB8c> [Accessed 18 August 2021].

NVIDIA GeForce, 2018. Ray Tracing Optimizes CG Film Rendering and Game Development. [online]. Available at: <https://www.youtube.com/watch?v=FxRZhnzETis> [Accessed 18 August 2021].

Palumbo, A., 2019. Q2VKPT Is the First Entirely Raytraced Game with Fully Dynamic Real-Time Lighting, Runs 1440P@60FPS with RTX 2080Ti via Vulkan API. [online] Wccftech. Available at: <https://wccftech.com/q2vkpt-first-entirely-raytraced-game/> [Accessed 28 July 2021].

Ridley, J., 2020. Vulkan Ray Tracing support enables even AMD GPUs to run Quake II RTX | PC Gamer. [online] Pcgamer.com. Available at: <https://www.pcgamer.com/vulkan-ray-tracing-driver-support-quake-ii-rtx-benchma [Accessed 20 July 2021].

Roth, S., 1982. Ray casting for modeling solids. Computer Graphics and Image Processing, 18(2), pp.109-144.

Smith, M., 2013. Pixar's Lightspeed Brings New Light to Monsters University. [online] Thisanimatedlife.blogspot.com. Available at: <https://thisanimatedlife.blogspot.com/2013/05/pixars-chris-horne-sheds-new-lig
html> [Accessed 20 July 2021].

Walton, J., 2018. Nvidia GeForce RTX 2080 benchmark, release date, and everything you need to know | PC Gamer. [online] Pcgamer.com. Available at: <https://www.pcgamer.com/rtx-2080-everything-you-need-to-know/> [Accessed 20 July 2021].

List of Figures