Sheffield Hallam University

Faculty of Science, Technology and Arts

Department of Computing Project (Technical Computing) [55-604708] 2019/20

Author:	Diego Llorens Rico
Student ID:	B9020086
Year Submitted:	2020
Supervisor:	Thomas Sampson
Second Marker:	Steve Oldacre
Degree Course:	BSc (Hons) in Computer Science for Games
Title of Project:	Raytracing under the microscope: Performance test of Raytracing Ambient Occlusion in a real time environment

Confidentiality Required?

NO X

YES ☐

Raytracing under the microscope: Performance test of Raytracing Ambient Occlusion in a real time environment

Diego Lloréns Rico

Abstract:

Raytracing is a rendering technique that aims to simulate the physical behaviour of the light. Until now this technique was computationally prohibitive and reserved for offline renderers, but the recent advances and architecture changes in graphic cards suggest we could start including this technology in realtime environments to hugely increase the visual accuracy of the videogames.

The following report details the research and implementation procedures undertaken in the final project of the last year of the BSc in Computer Games. The project consists in the implementation of a Raytraced visual effect into a realtime environment and its performance comparison against its rasterized counterpart. The project is based in a DirectX 12 that integrates a PBR renderer and the rasterized effect, which is compared using the same resources to a Unity scene that implements the same effect using raytracing.

The final motivation for this project is to test whether this advances in raytracing techniques have made them feasible enough for the vast majority of the industry to be adopted in production pipelines in a near future. The results show that although there have been noticeable advances towards this objective, raytracing is is still not ready to be adopted in non AAA production pipelines, as the implementation implications, and performance/visual tradeoff are still too high.

Contents Page

Chapter I

Introduction

1.1 Overview of the document

The following document is divided in 7 chapters that cover every aspect on the research and implementation of the project. Following this introduction an extensive research on the previous work and advances on each topic is conducted. The development reports section delves into the more technical details, the implementation decisions and problems arisen, and set of tools used during the whole development. After that, the results of the investigation is presented by comparing raytracing and rasterized solutions in terms of performance and visual fidelity. Finally, a critical evaluation on the overall outcome and the different conditions the project was involved into, also offering a further improvements section that discusses what could have been done better or improved.

1.2 Introduction

One problem with discussing raytracing is that raytracing is an overloaded term (Boulos et al., 2007).

Raytracing at its simplest is a technique that computes the visibility between 2 points in a Euclidean space; this can be used to know “what we see and how we see it” when talking about rendering a 3D scene. Its strongest point relies on the fact that it can be used to imitate the way humans perceive objects naturally, through rays of light that get into our eyes.

We simulate this behaviour in a 3D environment by casting rays through each pixel of the screen and testing whether these rays intersect with any object of the 3D world; this the first part of the rendering process, the visibility. However, there is still another complex problem to be tackled, and it is about finding out the final colour of the object at the intersection point. Light transport algorithms try to provide an effective solution to this problem by simulating the light-object interaction in a computer-generated image.

Raytracing isn't too slow; computers are too slow (Kajiya., 1986).

Nevertheless, raytracing based algorithms have always been rejected in real-time environments due to its extreme computational cost, which mainly relies on the huge number of rays that need to be casted and the time required to compute the intersection between these rays and geometry. This algorithm, especially when compared to rasterizing, which by far is the most used in real-time industry, it is still too slow to be considered as standard for both visualization and shading solving in the near future.

Technological key players, however, have started to propose a shift of paradigm in the last year; some announcements such as the release of NVIDIA’s RTX GPUs, Sony’s PlayStation 5 hardware accelerated raytracing module, or the latest DirectX 12 API seem to prove that it is possible to make use of these techniques in real-time environments like videogames, which has rocketed equally the excitement and the scepticism around this rendering approach.

1.3 Project Aims & Objectives

The project aims to expose the actual capabilities of a Directx 12 raytracing based project using a NVIDIA RTX graphic card and to study whether its feasible to be integrated as a future standard in the videogames industry, not only in the AAA part. The objectives are to study DirectX 12 API and its DXR module, as the current background in graphics programming limitates to OpenGL and some DirectX 11; to implement a Physically Based renderer and study the theory of lighting and rendering equations, and study, and evaluate a raytracing project in a real time environment.

To achieve that, the project will begin with a research on how the latest advances introduced by NVIDIA and DirectX12 affect the rasterizing pipeline and how a raytracing approximation can be compared to a rasterizing one when trying to solve each one of the 2 main problems that come across when rendering: the visibility and the light transport. This will be followed by a design stage in order to plan the implementation of a hybrid real-time renderer that tries to take the most from both approaches. The expected deliverable will then consist in a single application that will showcase a PBR rendered scene that integrates an advanced visual effect approximation which will be set to be generated through rasterization or raytracing. This deliverable is expected then to serve as a performance and visuals measurement project that showcases the performance of a raytracing against a rasterized solution.

Please refer to Appendix A for more information and the complete project specification, which includes the complete list of objectives and the project plan deadlines and milestones.

Chapter II

Information Review

2.1 Existing Work

Rasterization and Raytracing:

Rasterization rendering, based on the projection of 3D scenes into 2D textures transforming them from model space to screen space to be presented at the screen, has prevailed in 3D real-time industry from its very beginnings, it's really fast and effective, and modern graphics cards have been designed to accelerate rasterization as much as possible. As the computation capabilities improved, researchers developed methods to increase the realism of these representations, however, this usually meant to implement really advanced rendering techniques that complicated the whole process excessively, also, every technique implemented under rasterization rendering is nothing more than an approximation, not a real representation of reality.

We can find an example of this with reflections, currently, with the 3D engines used by modern games, reflections are usually calculated from an environment map. This technique gives a good approximation of the reflections located at "infinity", but cannot reflect closer objects. Closer reflections can be simulated by using dynamic cube maps, but they are expensive to produce, some possible solutions are to render smaller cube maps, resulting in pixeled reflections or reducing the number of times that this is calculated, resulting in latency in reflections. Moreover, interreflections, reflections generated from a 3D object to itself are extremely difficult to reproduce.

Raytracing rendering can be presented as another way to solve the rendering problem, its main advantage over rasterizing is that the conception of the basic algorithm is not only extremely simple, the rendering engine casts a ray that propagates in a straight line until it intersects an element and this initial intersection is used to determines the colour of the pixel as a function of the intersected element's surface, but it can also be extended to simulate many other effects such as lighting or reflections by casting secondary rays from the intersection point. Please note the difference between the *simulation* of the effects and the *approximation* of them used in rasterizing, as the way raytracing operates is exactly the inverse of what happens in the physical world.

Raytracing could seamlessly solve the previous reflections problem by casting a secondary reflection ray and getting the amount of reflected light received at primary ray intersection. This would generate perfectly simulated reflections and interreflections independently of the surface the primary ray hits.

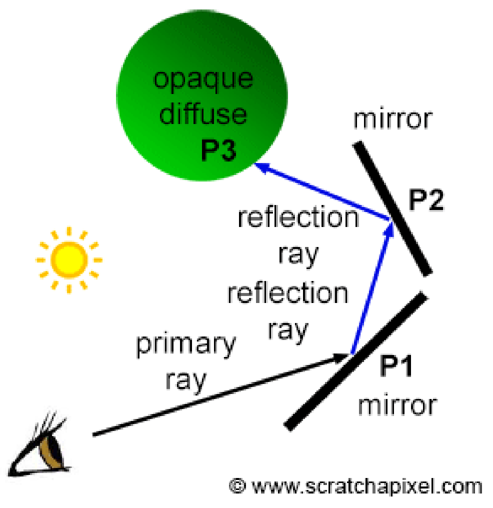


Figure 1: a light ray emitted by the viewer bounces multiple times off of the surface of the two mirrors before reaching the diffuse colour of reflected object. (Scratchapixel,2014)

However, raytracing has been always discarded from real-time environments because of one reason, this technique is computationally unaffordable. Imagine a screen of 800 x 600 rendering a 3D scene using raytracing, this would generate 480.000 primary rays, just for visibility testing (primary rays are used to determine visibility just as Z-buffer used in rasterization), for basic illumination each one of these would generate a shadow ray, a reflexion ray and a refraction ray, which would sum up to 1.440.000 secondary rays, each one being calculated at least 24 times per second.

Raytracing evolution:

Video games have been pushing the boundaries of real-time graphics capabilities over the years in order to pursue the highest realism in visuals.

"As computers have become more capable and less expensive, it became possible to consider more computationally demanding approaches to rendering, which in turn has made physically based approaches viable. This progression is neatly explained by Blinn's law: 'as technology advances, rendering time remains constant.'" (Pharr et al, 2018).

Physically based approaches have become widely used over the past decade, but this concept has undergone a vast research and theorisation since almost the very beginning of computer-generated images. A starting point can be set up in the early 80s, when T. Whitted (1980) presented his paper "An improved Illumination Model for Shaded Display", which introduced the idea of using raytracing techniques to achieve Global Illumination effects. Next remarkable advancement was introduced a year later with Cook and Torrance's reflection model (1981), which defined microfacet reflection models that permitted to accurately render metallic materials.

Three years later, Cook Porter and Carpenter introduced distributed raytracing (1984) which finally generalized Whitted's raytracing algorithm by introducing methods to compute fuzzy phenomena such as motion blur depth of field, penumbras etc.

James T. Kajiya presented shortly afterward "The Rendering Equation" (1986). This paper first presented a rigorous equation which generalized many of the already known rendering algorithms. At the moment, this rendering equation is the heart of all raytracers, there are many approximations to solve it.

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_{\Omega} f_r(x,\omega_i,\omega_o) L_i(x,\omega_i)(\omega_i \cdot n)\, d\omega_i$$

Figure 1: A slightly simplified version of the rendering equation. The reflectance equation, PBR strongly follows this version of the rendering equation

This equation can be read as follows, To find the light towards the viewer from a specific point, $L_o(x, \omega_o)$, we sum the light emitted from such point $L_e(x, \omega_o)$ plus the integral within the unit hemisphere $\int_\Omega \dots d\omega_i$ of the light coming from any given direction $L_i(x, \omega_i)$ multiplied by the chances of such light rays bouncing towards the viewer $f_r(x, \omega_i, \omega_o)$ and also by the irradiance factor over the normal at the point $(\omega_i \cdot n)$. (Basurco, 2017).

As anyone can conclude from the equation, evaluating the integrate for every single point on a surface can lead to an extremely recursive task unaffordable by any machine, Kajiya also presented in his paper a Monte Carlo method to approximate the equation.

Monte Carlo raytracing renders a 3D scene by randomly tracing samples of possible light paths. Repeated sampling of any given pixel will eventually cause the average of the samples to converge on the correct solution of the rendering equation, this is one of the most accepted physically accurate rendering approaches.

The next crucial step forward was presented in Veach's dissertation (1997), that advanced key theoretical foundations of Monte Carlo rendering and developed new algorithms like *multiple importance sampling, bidirectional path tracing and metropolis light transport*, which greatly improved ray tracer efficiency.

From this point, researchers started to pursue real-time raytracing, although many of the approximations that were developed did not stick to the physically based rendering rules, their results led to great progress in raytracing acceleration structures and performant algorithms, see Wald et al. 2001b Interactive distributed raytracing of highly complex models.

Nvidia Turing architecture, DirectX 12 and DXR:

As we have stated before, Raytracing has been relegated during the past years to offline rendering, and most of the research would only help to speed up the convergence of the images. We can define image convergence as the moment where the raytracing process has generated an image where the image error levels have been reduced to negligible levels, a naïve path- tracer usually takes around 5000 primary rays per pixel to reach this point, even more for pathological cases.

Some researchers tried during these years to combine the benefits of raytracing and rasterization techniques, with not so much success. A proper approach was very difficult to be implemented. Most of the times the solution ended up combining both techniques main drawbacks and losing their advantages, as they lost the elegance of raytracing and the performance of rasterization. However, a successful example can be found with Pixar's RenderMan renderer, which was extended with raytracing abilities to perform Reflections, Ambient Occlusion and, Shadows. These techniques were tested for the first time in production during the development of "Cars" movie. (Christensen, Fong, Laur, & Batali, 2006; Christensen & Jarosz, 2016) cover extensively this topic.

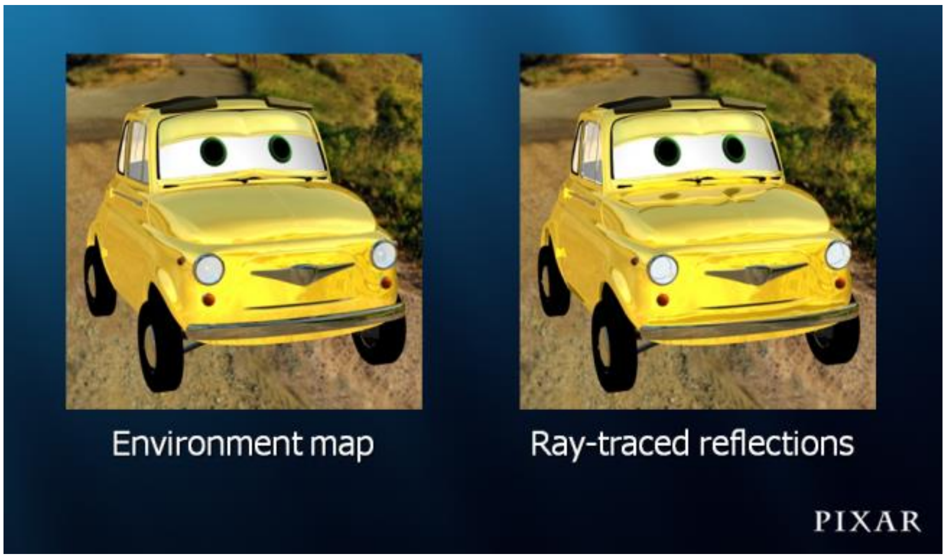


Figure 2: Image from the film Cars, Reflection Test (Left) with environment map, (Right) With environment map and ray-traced reflections. Image from (Christensen et al., 2006)

In 2018 this paradigm turned around after Microsoft presented in March a new API extension for DirectX 12, called DXR (Direct X Raytracing). Essentially, this extension implements a new pipeline model that can be complemented with rasterization and compute pipelines, this functionality does not require any additional hardware support, other than the already required for DirectX 12.

DirectX Raytracing introduces four new concepts to the DirectX API:

1. The acceleration structure is an object that represents a full 3D environment in a format optimal for traversal by the GPU.
2. A new command list method, DispatchRays, which is the starting point for tracing rays into the scene. This is how the game actually submits DXR workloads to the GPU.
3. A new set of HLSL shader types including ray-generation, closest-hit, any-hit, and miss shaders. These specify what the DXR workload actually does computationally. When DispatchRays is called, the ray-generation shader runs. Using the new TraceRay intrinsic function in HLSL, the ray generation shader causes rays to be traced into the scene. Depending on where the ray goes in the scene, one of several hit or miss shaders may be invoked at the point of intersection. This allows a game to assign each object its own set of shaders and textures, resulting in a unique material.

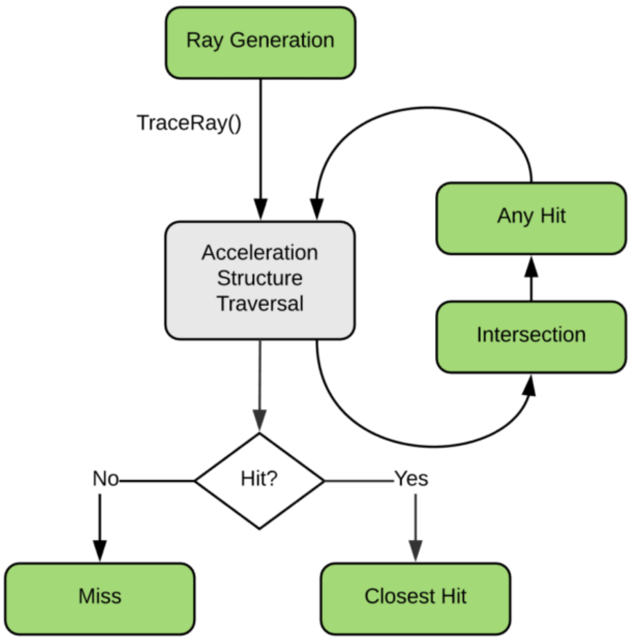


Figure 3: The Raytracing Pipeline. Image from (Stich, 2018)

4. The raytracing pipeline state, similar to Graphics and Compute pipeline state objects, encapsulates the raytracing shaders and other state relevant to raytracing workloads.

More technical information about how the new DXR API works can be found at (Direct 3D, 2018; Stich, 2018)

Although meeting the requirements needed to execute DXR, the computing capabilities of even high-end graphics cards weren't enough to get decent frame rates. This led to some serious doubts about this new API technology and whether it was going to be meaningful in production environments.

This changed later in August 2018, when Nvidia presented their new Turing architecture, whose main feature was the introduction of the *RT Cores.* RT Cores on GeForce RTX GPUs provide dedicated hardware to accelerate 2 of the main time-consuming processes of the raytracing workload, *BVH traversal* and ray-triangle intersection calculations. More interesting features were included to increase the performance of the whole rendering process such as the *Variable Rate Shading,* which allows developers to controls shading rate dynamically or *Deep Learning Super Sampling,* a technology that uses AI to increase even more the anti-alising temporal stability and image clarity and the frame rates achieved using RT Cores. For more information please refer to (Burnes, 2018; *NVIDIA Turing Whitepaper*, 2018)

From that point on, the graphics industry has become a hive of activity, talks and different kinds of research regarding this topic, these are some examples that can be used as a starting point for anyone interested in DirectX and Raytracing:

DirectX: Evolving Microsoft's Graphics Platform (Sandy, Andersson, & Barré-Brisebois, 2018): This talk, led by Microsoft and EA engineers, first DXR API and its usage inside Pica Pica project, which was developed using a Hybrid Rendering Pipeline that combined Rasterizer, Compute and Raytracing effectively inside a real-time game.

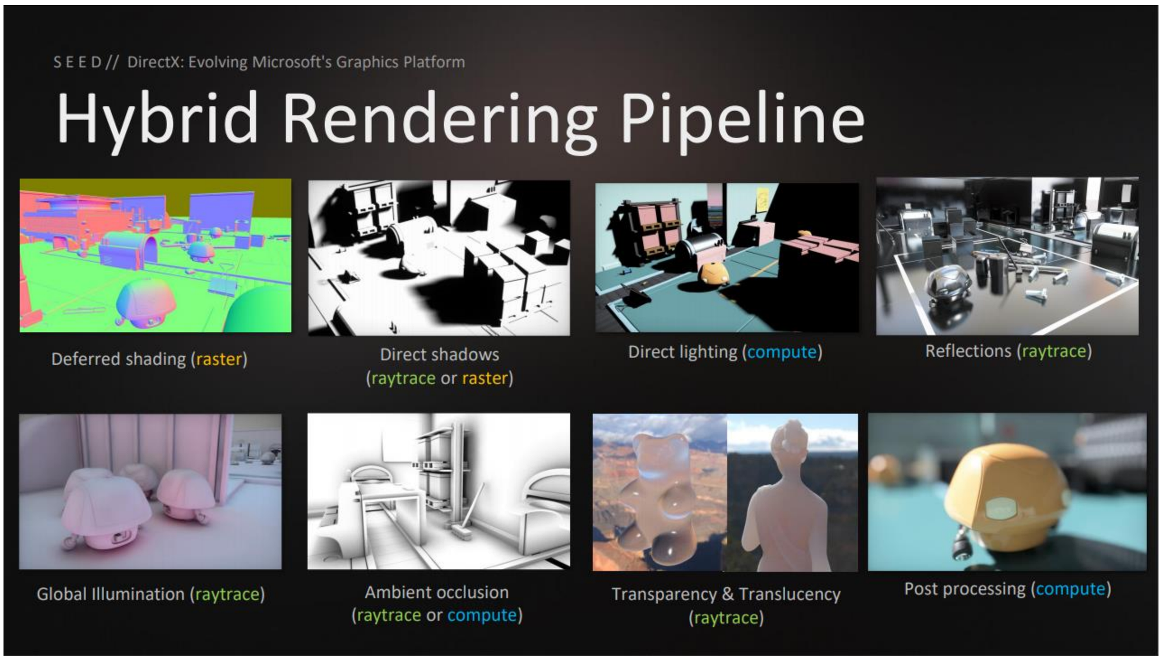

SEED // DirectX: Evolving Microsoft's Graphics Platform
Hybrid Rendering Pipeline
Deferred shading (raster)
Direct shadows (raytrace or raster)
Direct lighting (compute)
Reflections (raytrace)
Global Illumination (raytrace)
Ambient occlusion (raytrace or compute)
Transparency & Translucency (raytrace)
Post processing (compute)

Figure 4: EA's Pica Pica Project Hybrid Pipeline. Slide from (Sandy et al., 2018)

The Ray + Raster Era Begins - an R&D Roadmap for the Game Industry (Benty, Clarberg, & McGuire, 2018): This GDC Talk presented by NVIDIA engineers discuss some of the latest innovations previously presented at this document and proposes a roadmap in which the Game Industry evolves to embrace true hybrid rendering pipelines with raytracing being present at some of its stages.

Raytracing in Games with NVIDIA RTX (Liu & Llamas, 2018) This GDC talk covers a pretty wide range of different topics, first of all it introduces some of the benefits of using the DXR API with Nvidia RTX graphics cards, and present *GameWorks* Raytracing and Denoising modules. GameWorks is an award-winning SDK that provides solutions for visual effects and physics, this library has been used in many AAA games such as *Grand Theft Auto V* or *Batman Arkam Knight.* Finally, the talkers provide some guidelines for integrating DXR into your engine.

Introduction to DirectX Raytracing (Wyman, Hargreaves, Shirley, & Barré-Brisebois, 2018): This SIGGRAPH course serves as an introduction to DXR API suitable for pretty much anybody, as the first half is only focused on raytracing basics, and highly tutorialized code is provided. If a more general approach is needed please refer to Introduction to Real-Time Raytracing (Shirley, Wyman, & McGuire, 2019). The latest covers the same topics but providing more examples using different APIs and programming languages and also provides source code.

Physically Based Rendering:

Physically Based Rendering has become a common topic around the Graphics Programming industry for the last 10 years. Non PBR rendering models may base the way they simulate light more freely, depending on the artistic feel they want to achieve. Physically Based Rendering, on the other hand uses the principles of physics to model the interaction of lighting, the media and matter.

One of its main advantages in a production environment is that the engines do not rely anymore on the artist's understanding on how the light interacts with materials. With PBR, light interacts dynamically with each surface and "automatically adjusts" its final look depending on the lighting conditions and the environment in which the surface is placed.

Before continuing, it would good to clarify some common misconception that some people may have related to this technique. Although this technique is commonly attached to the production of photorealistic images, this does not mean that stylized art cannot be achieved with Physically Based Rendering, the realism we are talking about at this point has to do with the behaviour of light with the matter, and not with the final look of the render. Physically Based Rendering can be defined then as a methodology where every matter interacts with their incoming light. Examples of the behaviour of PBR in stylized games can be found in Zelda Breath of the Wild or Crash N. Sane Trilogy.

Figure : Semi-Stylized PBR Barrel Model (Tar0x, 2018)

To introduce the concept of Physically Based Rendering more formally we need to start describing light from a physics perspective.

Light can be defined as an electromagnetic transverse wave; this means that while the electromagnetic wave "wiggles" sideways while the energy transports it forward. This wiggle can be decoupled as two different fields, electric and magnetic, wiggling at 90 degrees one from the other. Electromagnetic waves can be classified depending on their frequency or their wavelength. Different light waves can be represented in Spectral Power Distribution diagrams.

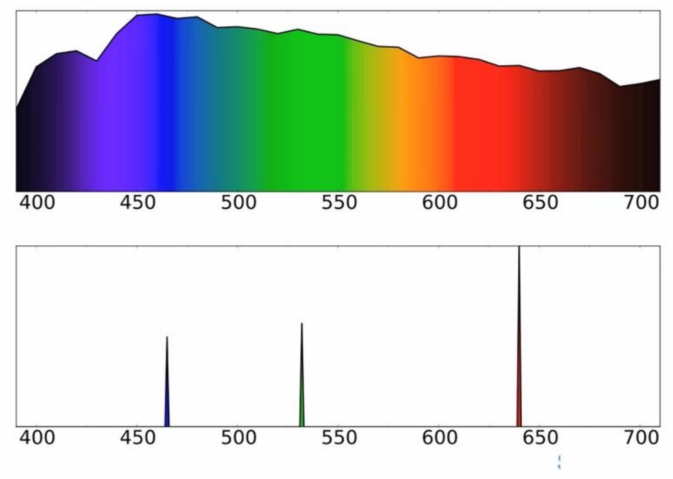


Figure : This image represents 2 SPDs, the one above represents the spectrum of natural white light, and the second one represents the sum of Red, Green and Blue laser. (Hoffman, 2012)

Although these two SPDs could not be more different, human vision perceive them as the same colour, this means that human vision is incredibly lossy, and that the human eye maps the N Dimensional SPD down to a 3-dimensional space.

As we said, we want to know what happens when light interacts with matter. Physically, we can tell that after a light wave hits the atoms, it polarizes them, this polarization absorbs a part of the energy, which is radiated back forming new waves.

Geometric Optics represent a far more simplified model than the presented above, its first assumption is to ignore everything with a size smaller than a wavelength, and treat any optically smooth surface as a mathematically perfect flat surface. However, most surfaces present irregularities that, even being way larger than the light wavelength, are still too small to be directly appreciated by the human eye, we can refer to this as *microgeometry*.

Before continuing it is really important to state other of the main principles of PBR, the *principle of energy conservation*: outgoing light energy should never exceed the incoming light energy (excluding emissive surfaces). At the moment a light ray hits a surface, it gets split in 2 parts, reflection rays, which directly gets reflected by the surface and don't enter the surface, known as *specular lighting*, and refraction rays, which is the remaining light that enters the surface and gets partially or completely absorbed; this is what we know as *diffuse lighting*. The distinction between reflected and refracted light in the principle of energy conservation brings us to another observation.

“Reflected and Refracted light are mutually exclusive. Whatever light energy gets reflected will no longer be absorbed by the material itself. Thus, the energy left to enter the surface as refracted light is directly the resulting energy after we've taken reflection into account.” (Vries, n.d.)

Microgeometry, will cause small variations in reflection rays. The rougher this microgeometry is, the greater the difference between reflected rays, causing blurrier reflections. Because of this microgeometry cannot be directly calculated, as a pixel can cover a lot of different directions, we compute this by statistically approximating the roughness given a certain parameter. This parameter, can be represented as a single floating-point value or as a 2D Texture.

In the case of refracted light rays, everything will depend of what kind of material the surface is made of. We can group materials into three main optical categories, Metals, Dielectrics and Semiconductors, but, as the latest ones are not usually represented, they will be ignored. Metals will absorb the electromagnetic energy refracted from the light so no light will scatter out of the surface. In the case of Dielectric materials, the incoming light interacts with the new medium. Some of the refracted light will eventually come out of the surface after being modified and partially absorbed, light rays re-emerging out of the surface contribute to the surface's observed (diffuse) colour. In PBR, we can make an assumption and discard every light ray that scatters at a certain distance from the incoming ray impact point. However, specific shader techniques known as *subsurface scattering techniques* can take into account this, at a price of performance.

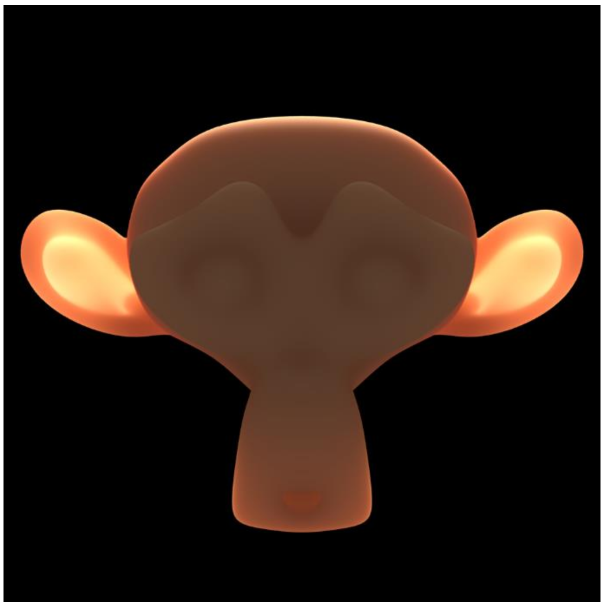


Figure : An example of Subsurface Scattering. Light hits the surface from the back, creating refracted rays that will be modified by this new media, and will eventually emerge at the visible part of the surface, reaching the camera.

Chapter III

Development Reports

3.1 Selection of tools

The following section will delve into the decisions taken for the development of this project.

Graphics API

Although the initial purpose of the project was to work with NVIDIA RTX and DirectX modules, the three main graphic APIs (OpenGL, DirectX and Vulkan) were explored in order to either confirm or replace the API selected in a first decision.

OpenGL: This graphics API was first researched because the project required some experience on using graphics APIs by the developer, and it was believed that it could be useful for the project to settle current knowledge and expand it. However, due to its design, which differs greatly from modern graphic APIs this was discarded in favour of learning at least one of the considered modern APIs.

DirectX: Then a research on DirectX was conducted, Microsoft's library may be considered the industry standard for PC and Xbox platforms, its focus on real time rendering and the current development of a module that introduces a raytracing rendering pipeline makes it very appealing to be the selected API. However, there is a drawback, DirectX12 differs a lot in terms of memory management and resource tracking, having then 2 choices, using DirectX11 with a more friendly API and usage, or moving to DirectX12 and use the Raytracing module, but having more issues on the setup and management of a base framework.

Vulkan: Another option was Vulkan, which, at a first glance had the same issues that DirectX12 in terms of management, those low-level issues were not appealing at allm, as that would require to spend a huge amount of the time on working on the basic framework. This, added to the fact that at the time this research was done on the API its developers hadn't still launched its raytracing module with NVIDIA (Koch, 2020), made this a not to consider option.

Cross-Platform Libraries: Lastly, a cross-platform library that could fit the requirements. BGFX is an API agnostic rendering library maintained by Branimir Karadžić which fully supports many different backgrounds and programming languages. While this could be good for most of general rendering purposes, as it can provide a very useful abstraction layer, but this was not the best option in order to develop a raytracing specific task.

Evaluation

Excepting DirectX, which is only focused to Windows and Xbox at the moment, all the others provide oficial support to multiple platforms, however this is not a requirement as there are no plans to port the tool to any other different system.

On the "oficially" supported languages OpenGL and BGFX do excel, maybe the last one even more due to the fact that not only it provides many bindings on the different languages but also a full graphic agnostic API that integrates many different backends, but it has two key drawbacks, the first one is that the community using this library and the documentation and examples on that are not as extensive as on any of the other libraries. The other one is that although implementing DirectX 12, there is no intention to integrate the raytracing support it is not planned until some of the other graphic APIs apart from DirectX 12 do include a similar functionality on that, so the point of having a graphic agnostic integration would have been dirt by having part of the project written in a "graphic agnostic API" which in reality could only make use of DirectX 12 implementation.

In this article (OpenGL - Will it ever have RTX Raytracing support?, 2019), in which some people discuss the possibility of OpenGL API to include a raytracing module that supports the new features included by Nvidia RTX cards. As the actual only way on doing that in OpenGL is through a binding that does the actual Raytracing in Vulkan, so OpenGL was discarded.

As there where just two graphics APIs left, which after some research looked very similar, the final election was DirectX 12. The possibility of using its DXR modules, natively compatible with the target graphics card took part in the final decision. DirectX 12 is supported in C, C++, C# and in Rust, but as almost all official documentation and project samples are written in C++, the other languages were discarded.

Note: In January 2020 this article that provides an indepth comparison of the graphic APIs was released, which supported the decission on using DirectX 12, (Galvan, 2020)

Engine alternatives

The option of using a commercial engine was taken into consideration, specially if any of these would solve some of the major issues related to the setup and basic usage of the graphics API (Buffer Management, Resource Tracking...). These are the results on each engine.

Unity Engine: This was the first option that was considered due to previous experience in developing successful projects guaranteed that there would not be major inconveniences in constructing the deliverable, however, the engine already provided an in-built solution for enabling and using a variety of Raytracing techniques, and that wouldn't completely fulfil the motivations behind this project, and the same happened with Unreal Engine.

Godot Engine: Godot is a free and open-source engine whose community and popularity has increased during the latest years. This engine was discarded because, due to its open source nature, "will not bother supporting a proprietary API for a single platform if there are open source alternatives (OpenGL, Vulkan)" - Rémi Verschelde, Co-maintainer of Godot Engine.

Other engines were not taken into consideration due to the lack of community or documentation on them.

Evaluation:

Unity and Unreal Engine 4 were the only engines that support DirectX 12 as a graphic backend and they both implement inbuilt raytracing based visual effects.

After having worked using both engines it is believed that the best decision is to use Unity's workflow, which provides vast well documented API and has a more solid community compared to the one in UE4, which is mostly settled on the usage of blueprints and visual scripting.

These arguments, joined to this GDC talk (Getting started with DirectX raytracing in Unity - SIGGRAPH, 2019), which details the procedures to enable and setup a raytracing pipeline in Unity was the decisive factor to choose Unity over Unreal Engine. However the motivation on this project was to first try to integrate all project into one engine independent solution, so first the research would try going that way and only would come back to Unity if anything went wrong on the implementation. So the option of using a commercial engine was initially discarded. Note: although discarding it in a first instance Unity became a key tool on the final outcome of the project, see hereafter reports for more information.

Debugging Tools

During the last 10 years, a variety of debugging tools have been developed to help graphic programmers to detect and correct any possible errors during the development of their projects. These are the different tools being used to detect any inconvenience that may raise.

DirectX debug layer: The DirectX Debug layer can be enabled at the entry point of the application and will provide extensive additional parameter and consistency validation of the different C++ API calls at the cost of the application being substantially slower. This Debug Layer will raise errors and warnings directly pointing to the source of different errors that may raise during the execution of these calls.

RenderDoc: As the DirectX Debug layer can only catch specific errors on API calls, but not detect possible bugs or unwanted results on the final output, this frame-capture based debugger is used to capture and analyze the result and the actual state of every resource sent to the graphics card.

NVIDIA Nsight Graphics: Although being essentially the same tool as RenderDoc, a frame-capture based debugger, Nsight Graphics provides an additional layer currently not supported by RenderDoc, the DXR Debugging module. This includes the possibility of debugging anything related to the Raytracing calls and its state.

Any of these tools was chosen one over the others during development and were used independently (when it comes to the frame debuggers), taking profit of the features and modules they have depending on the specific purposes when debugging the application.

Third-Party Libraries

Some libraries were used in order to facilitate the work on different areas of the final development:

Assimp: The Open Asset Import Library is a library to import and export a wide variety of 3D model formats. The main reason for choosing this library is that, apart from the multiple formats of import that supports, it is written in ISO-compliant C++ and includes a set of mesh post processing tools that are useful for a variety of purposes, such as normals and tangents generation, triangulation or removal of duplicate vertices.

DirectXTex: This library was added to provide support to texture loading and usage in Direct3D. The DirectXTex library was chosen for its simplicity in loading texture files and the direct access their functions provide to DXGI. It also offers some other compression related facilities which will make easier to pack and upload textures to the graphics card.

DX12Lib: Not having any prior experience in the task of constructing a DirectX renderer, and as a result of the previous research on DirectX12 showing results of many differences between DirectX11 and DirectX12. A decision was taken in getting some kind of additional support or abstraction layer that would make not only to produce a better work on the developing of the deliverable, but also on understanding the ins and outs of this approach, more close to a low-level API like Vulkan than to its previous version. In the search of a library that could fit this needs Jeremiah Van Oosten's blog, 3D Game Engine Programming library was found.

This, offered tutorials on how to architecture a rendering library using DirectX12. It also contained specific structures and procedures that, if followed, would help the user to understand and avoid certain pitfalls related to this new approach of the DirectX API. Covering these tutorials took most of the time of the first semester, but also granted a solid knowledge of the implications of constructing a solid renderer using a modern low-level graphic API at the time this abstraction was developed. Some of the classes however, differ from its original counterparts, in order to extend its functionality or adapt their usage to the final needs (see hereafter reports).

3.2 Direct3D 12

Developing an application in Direc3D 12 implies a series of complications if we compare it to its previous version, problems like memory management, state tracking or the deferred nature of the commands pushed to commandLists needed some solid support before advancing to the next step of the deliverable. This report will present and discuss the different decissions regarding the graphics API abstractions and structures undertaken during the development of the deliverable.

Buffer Management

Buffers that need to be uploaded to GPU are firstly stored in a *linear allocator* which creates the resources in an upload heap. Linear allocators may cause internal and external fragmentation if they need to be aligned which can cause a part of the allocated page is wasted. However allocations using this method are performed in constant time. This method is used both in DX12Lib and in MiniEngine, a DirectX12 starter engine maintained by Microsoft. This allocator works as follows:

- A manager creates a page of 2MB (the space should handle all resources that may be uploaded in a single command list). This allocator is in charge of requesting memory, creating and managing created pages, and releasing memory pages when they are not needed anymore.
- Pages contain a method that can allocate a chunk of memory from the page with an specific alignment. Pages manage their data by having a base and an offset pointer. Allocated memory is not freed until all the memory in the page is no longer in use, in which case the page is marked so it can be reused again.

In DirectX 12, command allocators are strictly linked to command lists and cannot be reset or reused until it is not in-flight on the command queue, that is why there will only exist one instance of these buffer managers per command list and allocator.

Descriptors and Heaps

Descriptors in DirectX 12 can be compared with a view object in DirectX 11, they can be defined as *"a piece of data that defines resource parameters. There is no operating system lifetime management, it is just opaque data in GPU memory"* (Coppock, 2019). This means that, in contrast to DirectX 11, their current state is not track or checked by the driver, so its up to the graphics programmer to handle and update the information stored in them (excepting the render targets bound to the swap chain, which are still tracked by the driver to ensure its correct functioning). The management of descriptors in the application is taken as follows:

- The class will allocate CPU visible descriptors that will be first prepared in CPU memory so they can be copied to a *GPU visible descriptor heap* and used in a shader.
- The class will be in charge to define and store updated information for every resource that needs to be uploaded to GPU at a certain moment of the execution, every resource that is no longer needed will be returned to the heap for its later use.
- The implementation uses a free list memory allocation algorithm based on Variable Size Memory Allocations Manager. (DiligentGraphics).

Descriptors at this stage still need to be bound to the graphics card in order to be used at the different stages of the pipeline, this is done by using a GPU-visible descriptor. It is important to note at this point that not every resource can be bound directly to the GPU (see *inline descriptors*), textures for example need to be bound to the GPU through a descriptor table. A descriptor table contains a set of descriptors of one or different types, but won't allocate the memory for them, they will be represented as an index and an offset inside a descriptor heap.

The usage of descriptor tables raises a series of problems from GPU side. An example is that only a descriptor heap can be bound to a command list for each descriptor heap type (*CBV_ SRV_ UAV, SAMPLER, RTV* or *DSV*) at the moment of a Draw call, this causes that every descriptor created for a specific set of calls needs to be bound to a heap that is going to be attached to the command list that contains that set of calls. This, as a chain, raises another problem, descriptors cannot be reused until the command list has completed the execution on the GPU.

DX12Lib solves this problems by architecturing an abstraction of the descriptor heaps which handles these problems and copies every GPU visible descriptors into a single descriptor heap before a draw call. This is achieved by accomplishing the following:

- Allocate GPU visible descriptors needed to match CPU visible created descriptors, since the previously allocated ones cannot be used to bind resources to the GPU rendering pipeline.
- Ensure the currently bound descriptor heap types have a suficient number of descriptors to commit every staged descriptor before a draw call, if this is not accomplished, the currently bound descriptor heap shall be discarded and a new one created.
- Prepare staged descriptors in a cache that is ready to match a given root signature.

Resource Tracking and GPU Synchronization

Barriers and resource transitioning are another of the big issues when developing in DirectX 12, the change of paradigm of controlling this through the application and not from the driver was motivated by the intention of reducing CPU usage and enabling driver multithreading and preprocessing.

In order to perform a resource transition, the API call needs to know the current and the next state of the resource that needs to be transitioned, this can be easily controlled if the renderer is single threaded, but would not if a resource was required by command lists constructed in different threads. An abstraction class is then required to handle this resource and subresource tracking with an scalable to many threads approximation. In a Youtube video, (Merry, 2015) presents a solution for the problem described. This solution works as follows.

When submitting a barrier to the manager, it checks if that resource has been used before, and adds the transition to a list of pending barriers, so next time the same resource transitions to another state, the *after* state from the first transition is taken as *before* state for the second barrier.

Then, when a command list is submitted to the queue for execution, the list of pending barriers (first transition barriers of a resource in a command list) is compared against a map of *global state* of the resources and, if its state is different (maybe the command list has used the same resource in different threads), it injects a third command list, which will be in charge of transitioning every needed resource to its correct state before commiting the command list to the queue and the global state of the resources will be updated. This two level state tracking approximation, with global between commands and local for each command list resource tracking, should solve the issue of manually handling resource states accross different threads.

Command Lists

Finally, in order to provide an as simple as possible solution to the application programming, an abstraction of the command list class has been developed, as presented in DX12Lib in order to make use of the already mentioned managers in the most functional way, reducing as much as possible, the complications around different API calls, all in all, the idea is to provide an interface that abstracts the usage of *ID3D12GraphicsCommandList* and only makes usage of our own calls in order to build the application.

3.3 Physically Based Rendering

The render and the reflectance equation

After the theory behind PBR has been discussed, it is time to present the mathematical model that support this technique. We will start by reviewing the reflectance equation, the “specialized” version of the rendering equation that we presented at the beginning of this chapter.

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_{\Omega} f_r(x,\omega_i,\omega_o) L_i(x,\omega_i)(\omega_i \cdot n)\, d\omega_i$$

Figure 1: Reflectance equation with emission considered

For the moment we are going to discard $L_e(x,\omega_o)$, which represents the radiance emitted by the surface at a point. This formula can be roughly described as the calculation of the irradiance in a point x facing to the viewer position ω_o. The first step is to solve $L_i(x,\omega_i)(\omega_i \cdot n)$, this part of the equation represents the calculations of the radiance of incoming light ω_i at a point x scaled by the $cos\theta$ formed by the normal n of the surface and the incident direction.

The Radiance equation that we need to calculate in our Reflectance model $L_i(x, \omega_i)$, is formally described as the total observed energy over an area A over the solid angle ω of a light of radiant intensity Φ:

$$L = \frac{d^2\Phi}{dAd\omega\ cos\theta}$$

Figure 1: The radiance equation

BRDF Cook-Torrance model

The last part of the reflectance equation that is still not solved is $f_r(x, \omega_i, \omega_o)$, known as the Bidirectional Reflective Distribution Function (BRDF). This function takes into account the material properties to scale the incoming radiance contribution depending on the material properties.

“A BRDF approximates the material's reflective and refractive properties based on the microfacet theory. For a BRDF to be physically plausible it has to respect the law of energy conservation. Technically, Blinn-Phong is considered a BRDF taking the same ωi and ωo as inputs. However, it is not considered physically based as it doesn't adhere to the energy conservation principle. There are several physically based BRDFs that approximate the surface's reaction to light.” (Vries, n.d.)

$$f_{cook-torrance} = \frac{D(h)F(\omega o, h)G(\omega o, \omega i, h)}{4(\omega o \cdot n)(\omega i \cdot n)}$$

Figure 1: Specular BRDF Cook-Torrance model, it is one of the most common BRDF implementations due to its versatility displaying a high range of materials.

The equation presented above represent a general micro-facet specular BRDF, also known as the Cook-Torrance BRDF model (diffuse part is not taken into account for the moment). As we can see, this function is composed of three main pieces.

$D(h)$: The microgeometry normal distribution function will calculate the concentration, relative to surface area of surface points which are oriented in such way that they could reflect light from ωi to ωo.

$G(\omega o, \omega i, h)$: The Geometry function will tell us the percentage of surface points with their normal being the same as the half vector h that are not masked or shadowed as an effect of the micro surface. The product of these two functions will result in the concentration of the surface points that actively participate in the reflectance that goes from direction ωi to ωo.

$F(\omega o, h)$: The Fresnel will tell us how much of the incoming light ωi is going to be reflected from the previous active points.

Please note that Cook-Torrance is not more than a particularly extended BRDF among the industry, but there are many more depending on the ideal feel desired. Some examples of different types of BRDF can be found at (Karis, 2013).

For an in-depth analysis of previously exposed topic please refer to (Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, 2018). Chapters 8 and 9 of this book covers extensively the insights of the Fresnel, Geometry and the microgeometry normal distribution functions. (Hoffman, 2012) also explains the theory of physically based rendering from a physics up to the mathematical model implementation.

Implementation details

When implementing a PBR solution, there is an inquiry that usually pops out at the time of representing the material, and its whether to use a *metalness or a specular workflow*. This issue has most to do on how an artist represents the properties of the surface and stores that information into textures, an extended article on this topic can be found at (PBR Texture Conversion, 2018) . Most production game engines do support both workflows but, as essentially both workflows represent exactly the same, this implementation will just integrate a metalness workflow.

The metalness workflow uses up to 6 different texture maps but some of them can be packed to the different channels. These are albedo, which packs the color of the surface or the reflectivity if the material is metallic, normal, metallic, that specifies if a texel is metallic or not, roughness, which will influence the microfacet orientation, and AO, that will be manually generated using Screen Space Ambient Occlusion.

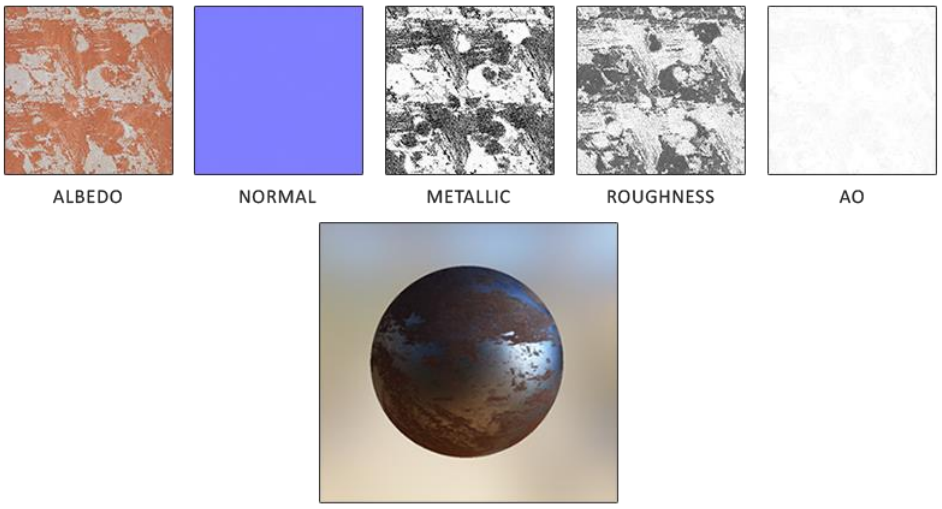


Figure: Example of each PBR texture applied to a sphere (Vries, J. de., n.d.).

Apart from the texture maps, some extra information can be passed to the PBR shader such as lighting information or some other parameters in order to modify the contribution of the different maps to the final image. In the solution, 4 simple point lights are added to better showcase the reaction of the different PBR materials to multiple light sources.

My PBR implementation follows every step described at the beginning of this subsection. There are no emissive materials in the scene, so that part of the equation was discarded. Then, for each light in the scene, the shader calculates the radiance of each light:

```
float distance = length(light.PositionWS.xyz - positionWS);
float attenuation = 1.0 / (distance * distance);
float3 radiance = (light.Color.rgb * light.Color.a) * attenuation;
```

Radiance attenuation is calculated by using the physically accurate Inverse Square Law. All light data is encoded as a float4 representing color RGB and Intensity as A.

After that, we calculate the BRDF, the part of the equation that modifies and scales the incoming radiance depending on the properties of the surface being represented.

$F(\omega o, h)$: The Fresnel.equation, the one that will calculate the amount of light that is going to be reflected is calculated using the *Fresnel-Shlick Approximation.* This is represented as:

```
float3 fresnelSchlick(float cosTheta, float3 F0){

    return F0 + (1.0 - F0) * pow(1.0 – cosTheta, 5.0);
}
```

This approximation expects F0 as the reflectance at normal incidence, being 0.4 for non metallic materials and approximated by using the metallic texture map. cosTheta represents the cosine of the angle conformed by the incident light direction and the normal of the surface.

The rest of the functions in the equation are represented by a raw implementation of the Throwbridge-Reitz GGX Normal distribution function and the Smith's Schlick for the Geometry function. These are the methods used on Unreal Engine 4. It is necessary to quote the research by (Karis, 2013) reflected in his personal blog on other approximations of each function and its cost implications as he was working in this exact shading model for the engine.

The last thing to note from the PBR implementation is a small modification on these last two functions, noted in his paper (Karis, 2013), the roughness is squared because in Disney's BRDF model noted that lighting was slightly more accurate when doing this, and was later adopted by UE4 implementation.

Image Based Lighting

At the beginning of the project the initial plan was to implement a dynamic image based lighting as the only light source to be read from the PBR shader, however, due to complications during development (refer to Raytracing and Critical Evaluation reports) this was replaced by a preloaded from disk convoluted HDR that only takes into account the diffuse part of the illumination and point lights were not taken out.

Diffuse: The diffuse integration is approached as follows, the already convoluted loaded from disk is passed as an additional texture to the PBR shader and, after all point light calculations the following was added:

```
    float3 kS = fresnelSchlickRoughness(max(dot(N, V), 0.0), F0, 
material.roughness);
    float3 kD = 1.0 - kS;
    float3 irradiance = ConvolutedSkyboxTexture.Sample(AnisoSampler, N).rgb;
    float3 diffuse = irradiance * material.albedo;
    float3 ambient = (kD * diffuse);
```

Just like the calculations presented above, first the the fresnel factor that shows the relation between Specular and Diffuse contribution is calculated, then the irradiance stored at the convoluted map is sampled and multiplied by the surface's albedo an the diffuse contribution. Finally, the ambient is added to the already calculated point lighting output.

3.4 Raytracing

Rasterizing and Raytracing Pipelines

Another of the purposes of this project was to develop an hybrid pipeline in which most of the work was done at the rasterization part but a more advanced rendering effect was implemented using Raytracing and the final results were merged into a final output. The considered effects to be implemented were reflections or Ambient Occlusion, as they were, the most appropiate to the scope of this project taking into account the overall amount and target of the tasks.

Although the research was firstly oriented to do reflections and ultimately substitute the reflections part in the PBR shader for the raytraced solution, some problems related to how the engine was built arose and finally opted to implement an Ambient Occlusion, which comparisons with a non-raytraced solution would be far more easier to implement as the engine is oriented to deferred rendering.

Implementation and Problems

The implementation of the Raytracing module was splitted in 3 different parts:

Hello World of DirectX12 Raytracing module: This part of the implementation consisted in being able to render a single triangle in a completely separate solution of Visual Studio. It was successfully carried out by using DXRHelper abstraction classes, developed and documented at (DX12 Raytracing tutorial – Helpers, 2020). These classes aim to provide abstraction to some of the most verbose parts of the Raytracing module API calls while limiting as much as possible the loss of flexibility when using.

This part of the integration was intended as an initial contact to the different procedures of the Raytracing API, and it was not intended to be included into the deliverable because it was just a step by step follow-up on the tutorials presented at (DX12 Raytracing tutorial - Part 1, 2020).

Integration of DXR module into deliverable: This part of the implementation consisted in being able to render the same single triangle but using the deliverable solution as base framework. The implementation would use the same abstraction layer as the previous stage but with minor modifications to the Acceleration Structure to adapt each method to the input vertex data.

The Raytracing pipeline follows a workflow very similar to DirectX 12 rasterized ones, but with some differences. First of all it highly advisable to check that the graphics card supports raytracing at the initialization of the program

```
D3D12_FEATURE_DATA_D3D12_OPTIONS5 options5 = {};
ThrowIfFailed(device->CheckFeatureSupport(D3D12_FEATURE_D3D12_OPTIONS5,
                         &options5, sizeof(options5)));
if (options5.RaytracingTier < D3D12_RAYTRACING_TIER_1_0)
       throw std::runtime_error("Raytracing not supported on device");
```

If there are no problems caused by an incompatibility of the graphics card, the next step was building the acceleration structure. This is no more than a way to store geometry and its relevant information in order to reduce as much as possible the number of intersection tests that need to be run during rendering. In DXR, this structure is divided in a two-level tree, the bottom level which will hold the vertex data of an object, and the top level, that can be seen as the internal node graph collapsed into a single transform matrix, for each BLAS, there will be one or more TLAS references that will instance a reference of the object in the required position.

In the solution the BLAS consists in a single triangle mesh that contains data for Vertex position and color, and the TLAS was just a single Identity matrix.

Secondly, a raytracing pipeline is necessary. Here the API binds the necessary shaders to execute the rendering. 3 shaders are used in this pipeline, RayGen, that initializes a ray descriptor and invokes the ray shooting procedure; ClosestHit, which be called as soon as the ray intersects with its first geometry, and miss, which will be called if the ray gets to its max defined depth without colliding with any geometry. Appart from binding those shaders some other interesting information is defined at the raytracing pipeline, for example the recursion depth, which can issue a new raygen call after hitting a closest hit surface, useful for different effects.

After the pipeline, we need to create a RenderTarget, and its correspondent descriptor heap in order to bind the resources and gather the results of the rendering, since Raytracing can not render directly to screen.

Finally, the Shader Binding Table is created, this was the most different concept when it comes to compare rasterization and raytracing implementations. But, as the raytracing solutions need to have all the information of all the acceleration structure at the same time as some of the rays can hit at points where the camera is not facing at, this is necessary. As fancy as it sounds this table will just store references for the location of shaders and the resources bound to each one. There should always be one RayGen entry, followed by one or more Miss entries, and ended by N hit entries.

This ends with the initialization of the raytracing pipeline. The render loop, on the other side, was surprising due to its simplicity. Just as in rasterization, the loop consists in binding the pipeline resource, declaring a `D3D12_DISPATCH_RAYS_DESC` which binds every raytracing related resource, ensure every resource is in its correct state, and a call to `DispatchRays(D3D12_DISPATCH_RAYS_DESC*);`

Although that this should have been the easiest part as mostly consisted essentially in porting an already working exercise into the deliverable solution, also using the abstraction classes released by DirectX, however, many problems arose (please refer to Critical Evaluation for a further explanation on this). But as most of them were corrected, an error in the generation of the Acceleration Structure appeared and after spending weeks trying to solve it, in order to prevent the impossibility of getting results from the independent renderer, a replanning on the project was discussed and approved with the supervisor.

Development of Raytraced Ambient Occlusion: This would have been the last part of the raytracing research module, but was not implemented due to the aforementioned Acceleration Structure issue.

Raytracing in Unity as solution

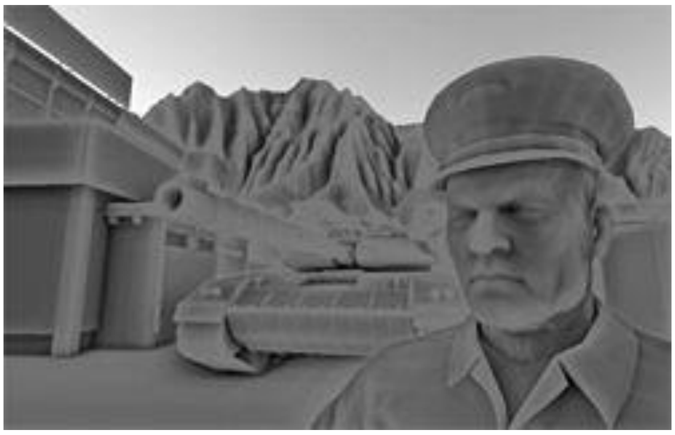
In order to gather the data for the deliverable comparisons between a raytraced and a rasterized solution, the report was needing to gather some raytracing results. In its 2019 version, Unity provides a Rendering Pipeline template which supports DirectX 12 Raytracing modules. So after consulting with the supervisor, we decided that it could be convenient to setup the same scene in a Unity project and enable Raytraced Ambient Occlusion in order to be used both as ground truth reference and comparison with rasterized Ambient Occlusion in the deliverable. Just for the reader information, please note that these features in Unity Engine are in Beta development at the moment of the writing of this report (2020), and are reported to be changed in a near future, so it is highly advisable to not use these assets for production purposes.

DirectX 12 Raytracing module integration in Unity and its workflows were presented in (Getting started with DirectX raytracing in Unity - SIGGRAPH 2019, 2019). Some of the main points of this talk are that first, there is a new class in Unity's API RayTracingAccelerationStructure, which can either be managed automatically or manually. In Unity the AS is normally built once per frame in order to update any dynamic geometry. Unity also includes a new shader type, the RaytracingShader, which handles the HLSL code mostly as in raw DirectX12 but including some limitations, it is still not possible to override the intersection shaders. Finally, point out that although the raytracing module should be pipeline agnostic (when it comes to the multiple rendering pipeline setups that unity offers, HDRP, URP...) it is only officially supported for HDRP. This is also the only pipeline that actually implements in-built features that include raytracing effects. That's the reason why this pipeline was used to generate the Ambient Occlusion maps.

3.5 Ambient Occlusion

Ambient Occlusion is a shading technique that aims to approximate the exposure of each point of a surface to ambient lighting. This technique calculates the "less illuminated" parts of the scene caused by the diffuse reflection of the lighting rays. This effect causes that some of the inner edges and nooks of a surface to be occluded and receive less light than the more exposed ones.

The most used way of generating Ambient Occlusion in offline environments is still raytracing, but due to hardware limitations most of online environments still can't afford that. (Kajalin, 2009) developed a method that calculated an acceptable result and hugely increased the depth sense and the realism of its scenes. This technique estimates the ambient occlusion based on a reconstruction of the depth of each pixel and comparing that depth against a pseudo random sample of the neighbouring pixels. This is what we actually know as Screen Space Ambient Occlusion.



SSAO resulting map of a videogame.

Screen Space Ambient Occlusion, even resulting in a great improvement in terms of visuals and having a more than reasonable cost specially when implementing a deferred renderer (most of the needed resources are already generated) is far from perfect, there are some artifacts caused by the fact that the calculations are done, as the name says, in screen space (refer to Chapter IV). There have been many improvements on this technique. NVIDIA researchers have expanded this topic extensively HBAO and HBAO+ (Bavoil L., Sainz M., 2008) and VXAO (Panteleev, A., 2016), but in order to maintain the scope of the project, only the most essential implementation will be integrated.

My implementation implements a simpler version of the original algorithm that doesn't need any reprojection and does all the calculations in 2D. A prepass render is firstly taken with the objective of storing the resulting position and normals in View Space into textures, this step is automatically done if implementing a deferred renderer. Then, in the SSAO shader we apply the following. The author of the algorithm also simplifies the calculations by treating all neighbouring pixels as points and not as spheres, with that he exposes the following:

$$Occlusion = max(0.0, dot(N, V)) * (1.0 / (1.0 + d))$$

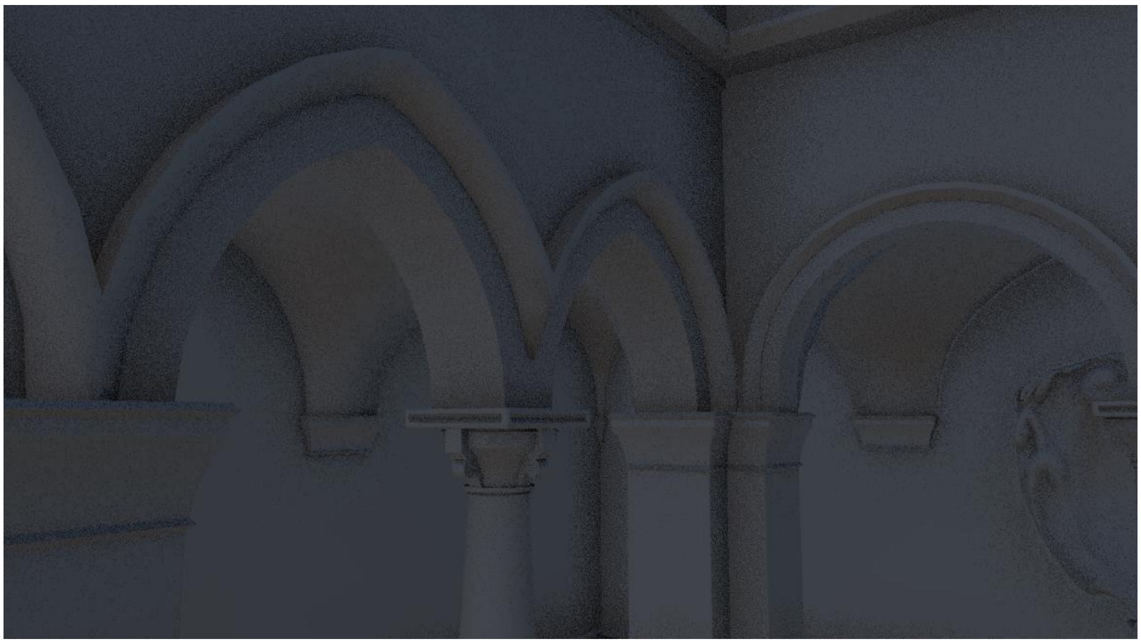
(Mendez, J. M. ,2010)

Being d the distance between occluder and occludee, N the normal of the occludee and V the vector between both of them. The first part of the equation makes the assumption that points directly above the occludee contribute more than the ones that are not, and the second part attenuates the resulting effect linearly.

In order to achieve occlusion 4 samples are tested after being rotated and transformed using a random normal texture. Finally the result is stored in a texture and blurred using a bilinear blurring shader to hide possible artifacts that could appear.

Regarding the raytraced Ambient Occlusion algorithm in Unity, and as the setup and usage was discussed previously, a detailed review on how an RTAO algorithm works and its main advantages compared to the algorithm that was presented on the deliverable is discussed.

Almost any typical RT algorithm is split in 2 main passes, the raytracing and the denoising pass. In RTAO, the raytracing pass consists in reconstructing the world position by using the g-buffer normals and depth maps and for each position to shoot n rays in pseudo random directions (filtered by importance sampling) but always inside a normal oriented hemisphere. If the ray, which length can be variable to increase or decrease the spreading of the effect, does hit any other geometry, it adds occlusion based on the distance from the initial position and the hit point. If there is more than one sample per position this process is repeated and the occlusion is added to a total contribution value for that position. After this process finishes the result is something like this:



Noisy Ambient Occlusion (1 spp).

The remaining pass would be to filter the image. Denoising algorithms and temporal acummulation, are the key for being able to consider raytracing for realtime environments, as they are responsible for cleaning up the above map and generating the final textures which are much closer to ground truth than SSAO algorithms which suffer from darkening screen edges, haloing, not taking into account offscreen geometry and problems on bias.

Chapter IV

Deliverable Comparisons

4.1 Data collection

The following chapter presents the results of the research in terms of performance and visuals, these will be compared to offer a reflection of the resulting work. After this comparison the results on the investigation link together and the feasibility of replacing Ambient Occlusion in videogames and evaluation if a change in this term to a raytraced approach is worth it is discussed.

Performance captures

The changes in the deliverable during development made impossible to compare what would have become the most interesting performance captures on this report, and this is the comparison between a raytraced and a rasterized solution in the same system. As a Unity raytraced solution was implemented, the comparison between both systems makes no sense, however, individual performance captures on each system are presented:

Testing machine	
Model	MSI GE63 Raider RGB 8SG
Processor	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
Memory	DDR4-2666, 16 GB
Graphics	NVIDIA Geforce RTX 2080 8GB GDDR6

Due to the recent events regarding COVID-19 this demo could be not tested in any other machine but the initial testing machine, however, it could have been interesting to check the performance on different target machines.

Original deliverable captures: All captures will be taken as an average of the FPS during 30 seconds of a static camera with full screen mode activated, V-Sync deactivated, and rendering targets to 1920x1080.

Deliverable Reference Image

Test 00: Starting point reference	
Prepass and SSAO	Disabled
Blur	Disabled
Average Elapsed Time per frame	3.01 milliseconds

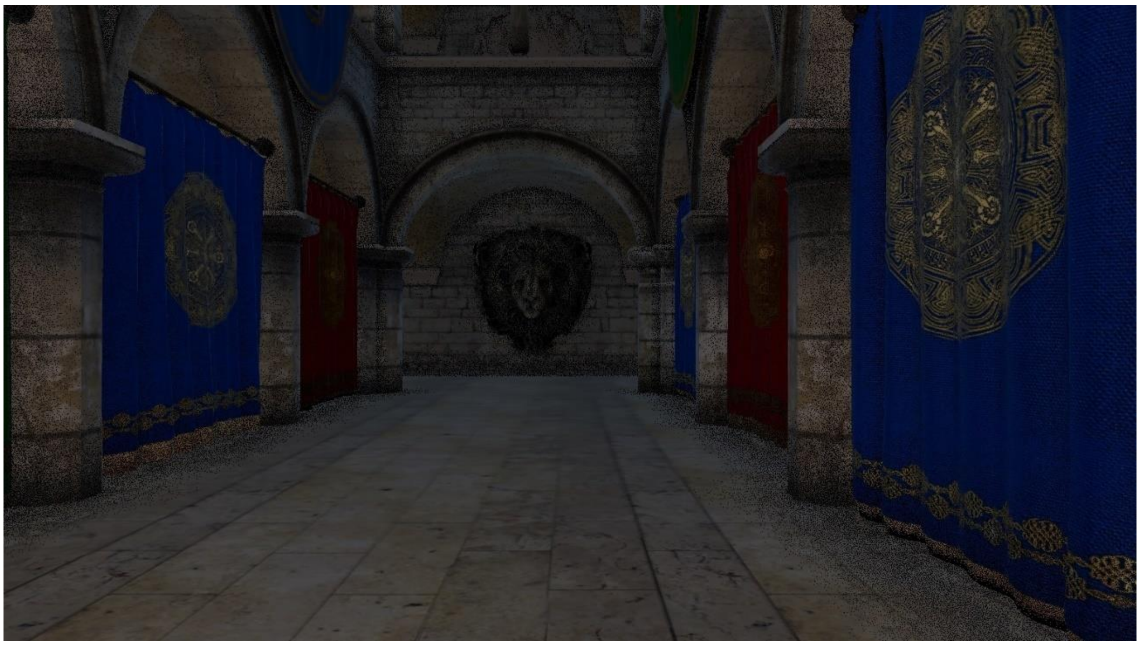
Test 01: Enabled SSAO, No Postprocessing	
Prepass	Enabled
SSAO	4 Passes, No backfaces
Blur	Disabled
Average Elapsed Time per frame	4.6 milliseconds

Test 02: Enabled SSAO, Postprocessing	
Prepass	Enabled
SSAO	4 Passes, No backfaces
Blur	Enabled
Average Elapsed Time per frame	5.1 milliseconds

Test 03: Enabled SSAO, Postprocessing, More Samples	
Prepass	Enabled
SSAO	32 Passes, No backfaces
Blur	Enabled
Average Elapsed Time per frame	5.6 milliseconds

During the recording of these captures there were some more details observed that are worth mentioning. At some points, not only during captures but in development, there was a continuous increase in ms as the deliverable was executing, this was not related to anything in particular as it did not happen every time and the increase was not visually noticeable if not measured, as the deliverable is actually running in the range of 144 to 240 FPS (depending on VSync and enabled effects). This was only observed with Vsync disabled.

Unity Raytracing AO captures: This demo tried to simulate as much as possible the scene conditions that were present in the original deliverable, for that, the same model, materials, lights and camera setup was added. Vsync and rendering to fullscreen was also considered. However, due to limitations in the Unity mesh importer, some of the meshes were not loaded correctly and were discarded from the final output. These are the results:



Unity Reference Image

Test 00: Starting point reference	
RTAO	Disabled
Denoise	Disabled
Average Elapsed Time per frame	10.01 milliseconds CPU 3.4 milliseconds GPU thread

Test 01: Raw RTAO, No Postprocessing	
RTAO	Enabled, 4 Samples
Ray Length	25
Denoise	Disabled
Average Elapsed Time per frame	12.0 milliseconds CPU 5.6 milliseconds GPU

Test 03: RTAO, Denoise enabled	
RTAO	Enabled, 4 Samples
Ray Length	25
Denoise	Enabled, denoiser radius: .5
Average Elapsed Time per frame	12.3 milliseconds CPU 5.8 milliseconds GPU

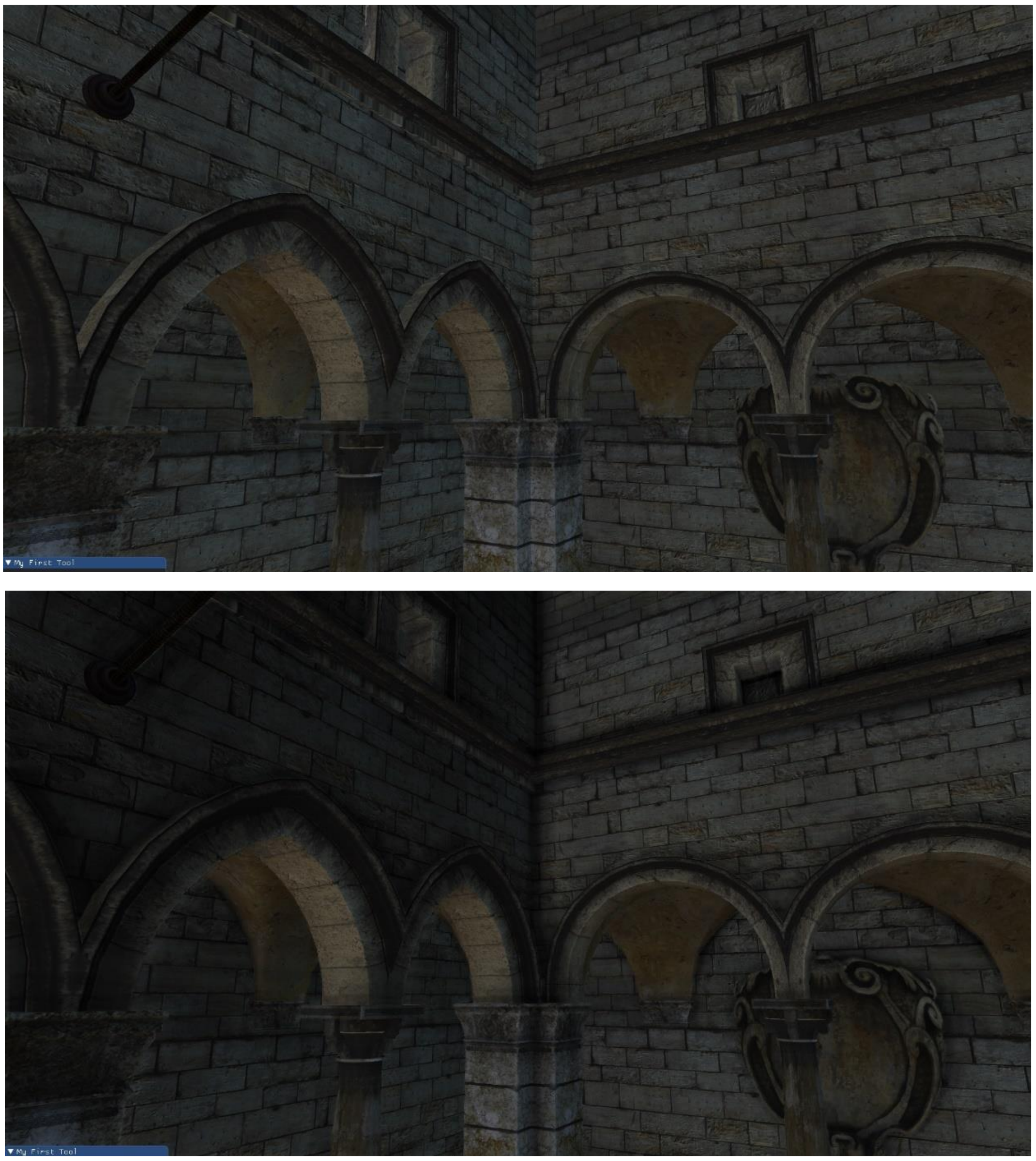
Test 04: RTAO, Denoise enabled, Increased sampling	
RTAO	Enabled, 32 Samples
Ray Length	50
Denoise	Enabled, denoiser radius: 1
Average Elapsed Time per frame	29.3 milliseconds CPU 6.1 milliseconds GPU

We can see that the impact of the calculations highly rely on the CPU which duplicates its calculations time as the samples and the quality of the raytracing pass increases. There were not any issues or remarkable behaviours regarding performance during the development of this report in Unity.

Visuals comparison

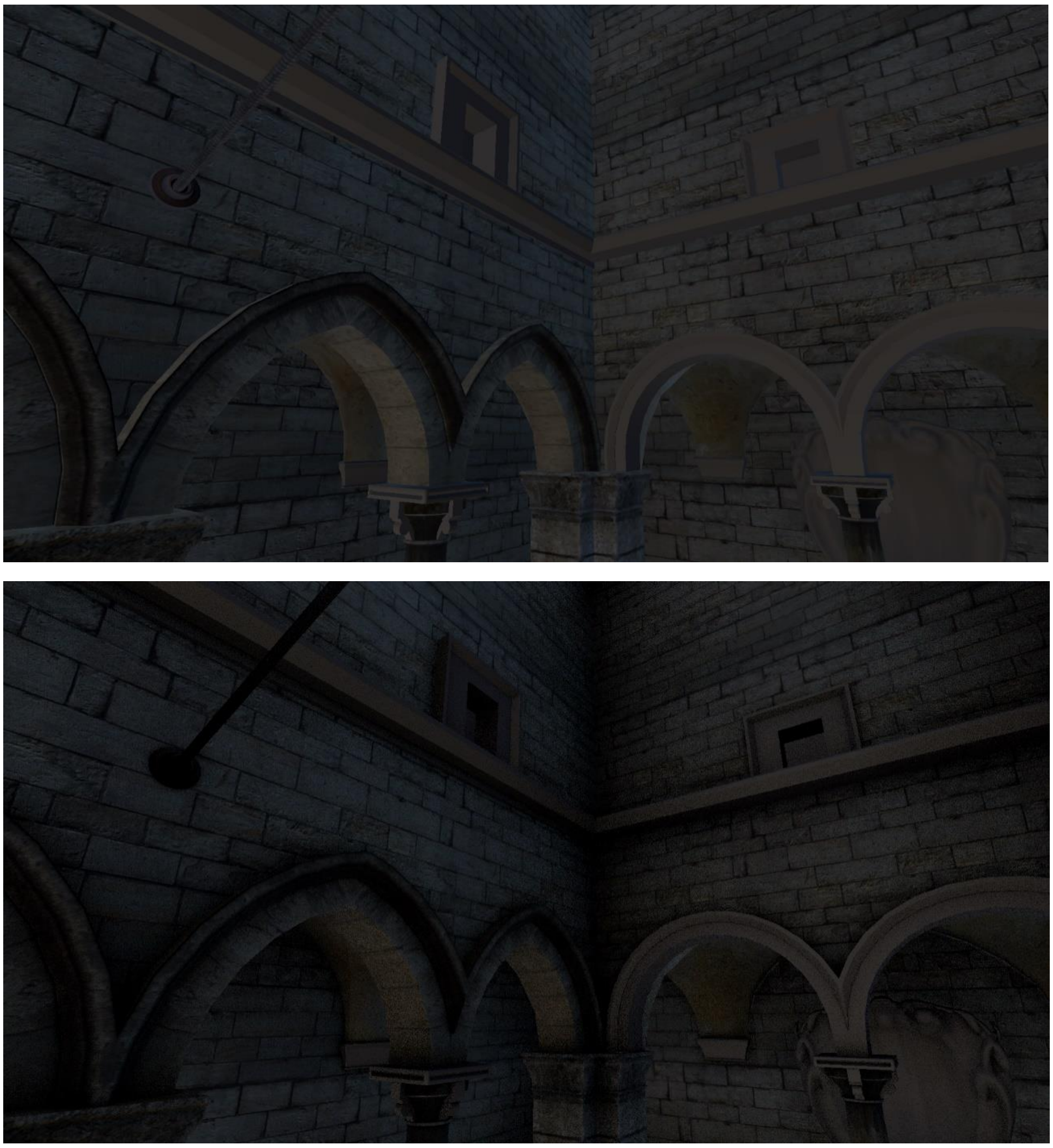
This subsection showcases the visual output of the raytraced version of the Ambient Occlusion map in Unity compared to the SSAO option that was developed in the original deliverable. To achieve that, samples of AO in three different locations inside Unity Scene were taken and tried to be replicated in the deliverable, showing there the divergence between both. The occlusion effect is exagerated in both scenes for better visualization purposes.

Shot 1:



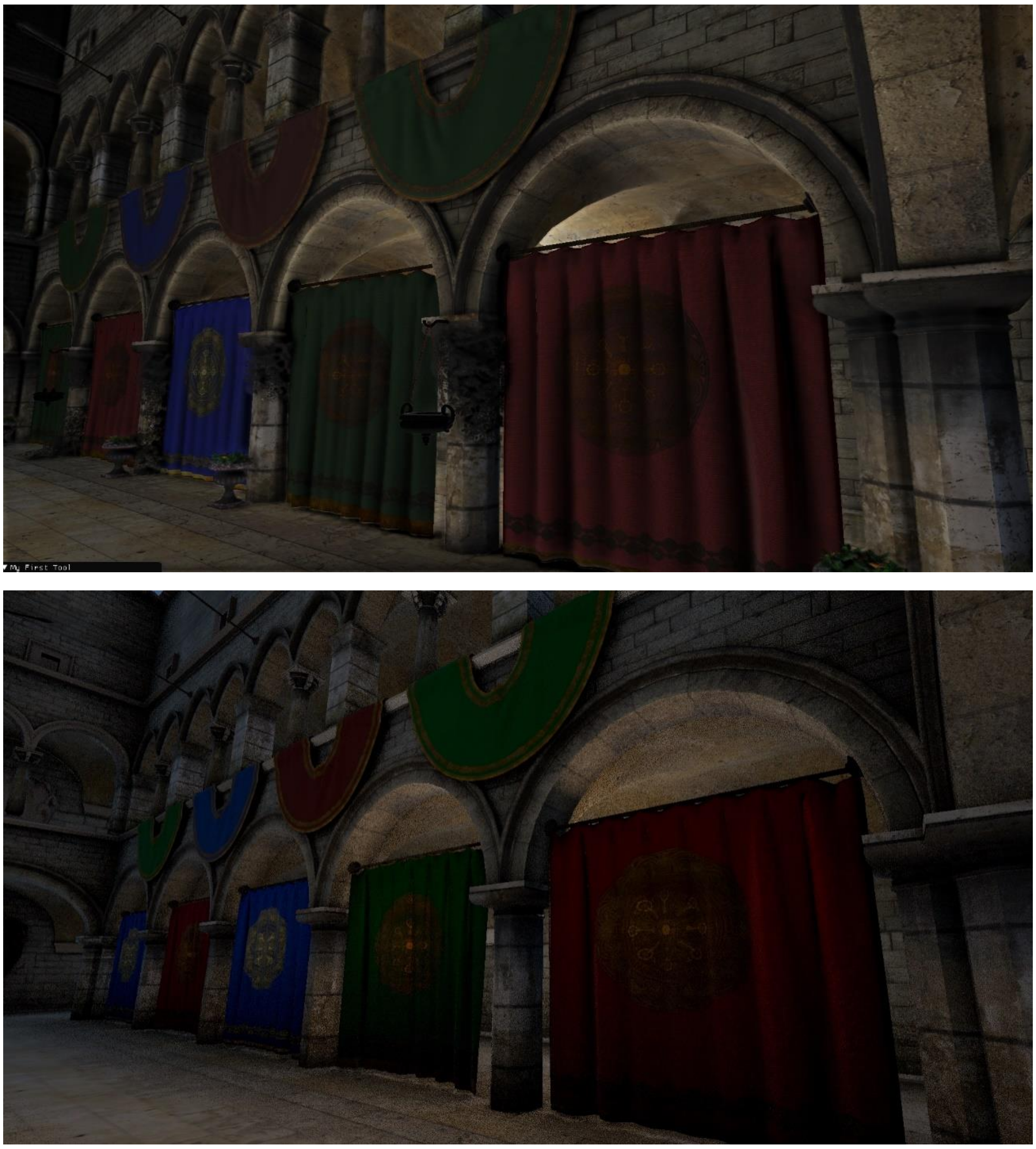
Deliverable no AO to AO Comparison (Test 00 to Test 01 settings)

Here it can be seen that a great sense of depth is gained by using the SSAO approach, nooks and holes are nicely darkened while preserving most of the more exposed points, there is however an artifact causing part of the inner wall to be darkened with no apparent reason, which results in a loss of detail of the image. There is also some visual inconsistence at the very left inner part of the arch which is causing that part to be more illuminated than it should, this is caused as an effect of a point light situated in the scene.


Unity No AO to RTAO Comparison (Test 00 to Test 01 settings).

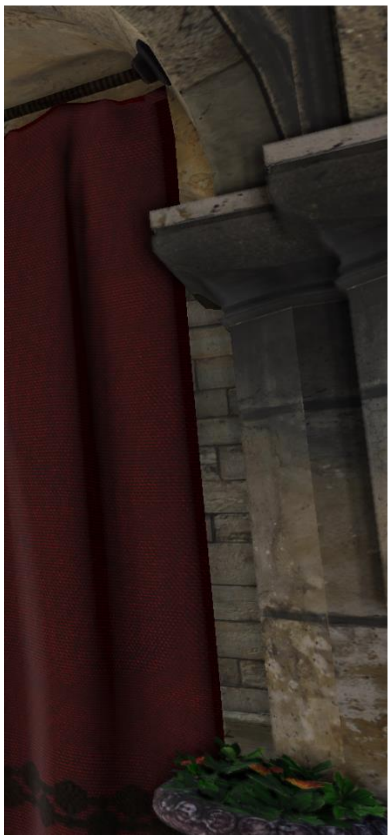
In Unity we also observe that the resulting effect also greatly improves the sense of depth. This image, however, shows noticeable noise apart from some incorrect reprojection issues when placing the camera. This is caused due to Unity system stores and tries to reproject the result of the raytracing map (if the camera changes) of the last N frames in order to decrease the raytracing workload, and offer a better quality of image over the time. However, more uniformly spread darkening areas are placed at the different nooks that the shot has and the AO does not feel strange at any point.

Shot 2:



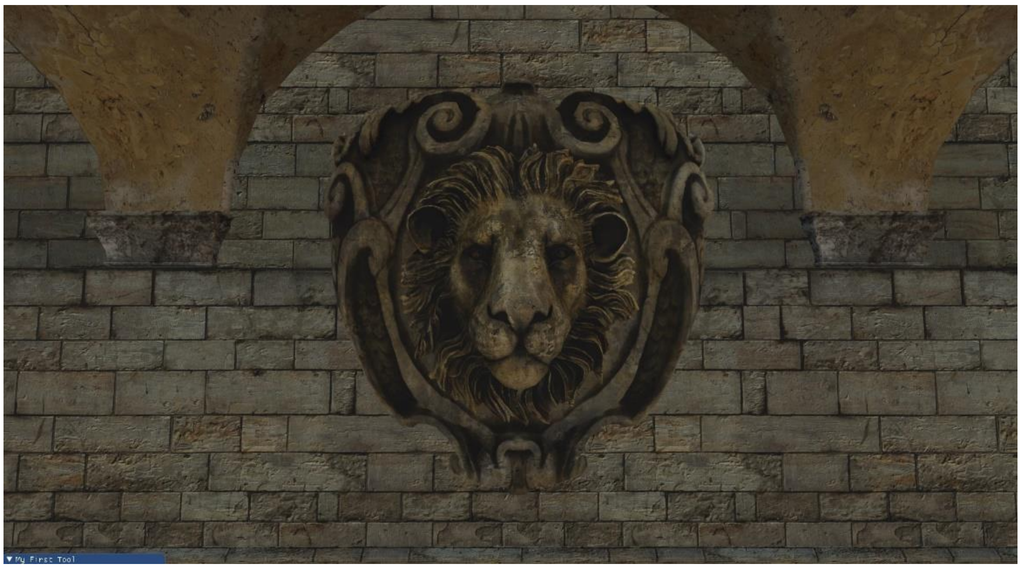
Deliverable AO to Unity RTAO Comparison (same settings as above)

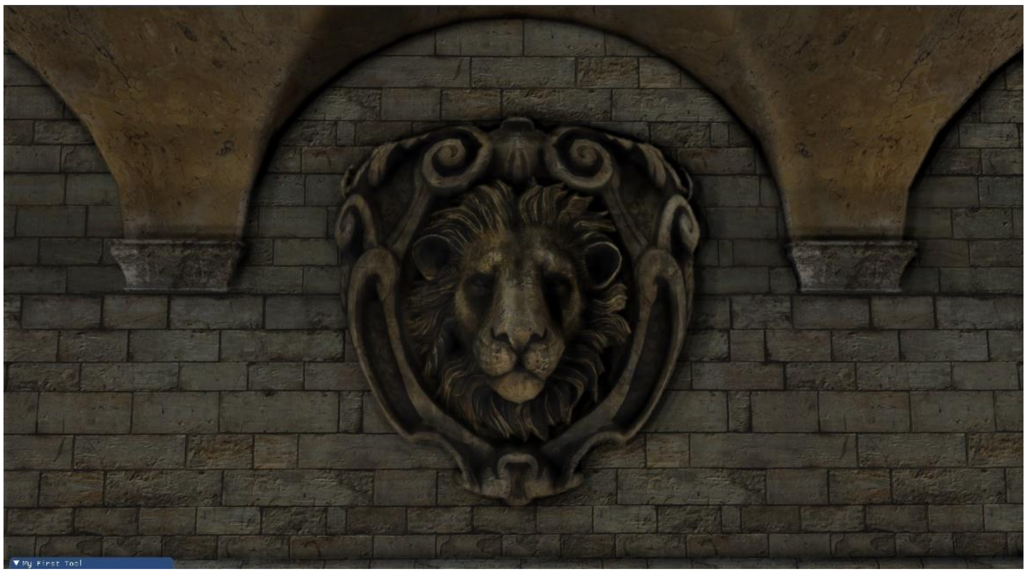
Screen edge artifacts detected when placing the camera on position (See Figure below), there is also a some haloing on one of the round columns at the upper level. At the front wall, there can be seen that a part of the darkening caused by the not visible corner is missing due to the fact that there is no information of that part of the scene. There is also some missing information in the ground, specially noticeable at the left side of the image where the distance is greater. In contrast to that, it can be found that the upper flags shadow perform particularly well against the wall that holds them.

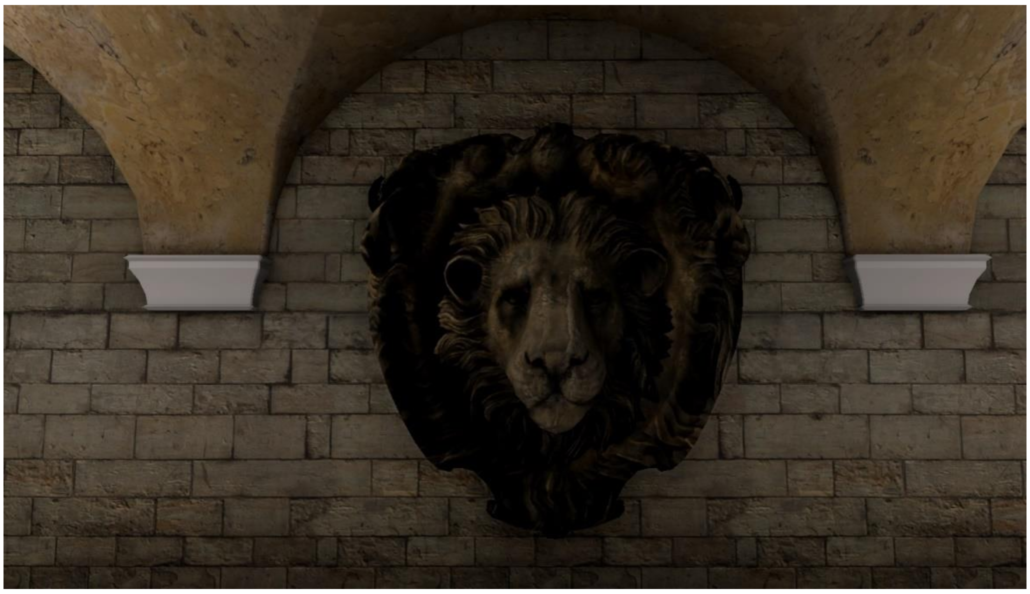


Next to the right border we appreciate a cut in the AO effect caused because the effect is calculated in view space.

Shot 3:







No AO, Deliverable AO and Unity RTAO (Max settings)

Finally, these three shots in which we the stone figure of the lion of Sponza scene. The main difference, apart from some of the textures lost due to Unity loading system limitations, is that the lion is much more darker and less detailed in Unity's version with some of the areas such as the border of the lion contrasting with the wall receiving the point lights, on the other side it is worth mentioning the slight darkening that the part of the wall above the lion and how that is joined to the arch darkening, compared to just the darkening at the corner of the arch that the SSAO approach has. The belief is that in this shot, although not physically correct, the deliverable approach is better integrated into the overall, as the feeling it is received is that apart from not having lost the sense of depth (in this shot in no AO the lion just seems a painting on the wall), the detail on the figure and its shape can also be appreciated.

Summary

After this research, gathering the results, and leaving aside the truism of the fact that the Raytracing approach is physically correct, which causes that this can be appealing in most of the shots. The belief is that, at this point it is still not worth it to even start switching to these hybrid pipelines. These are the arguments that support this statement:

- Raytracing is still not prepared to be fully integrated into the industry: While using Unity with enabled Raytracing, at least a 20% of the development time was dealing with internal errors that caused the engine to break. This, joined with the official statements that say that this new Templated Rendering Pipeline technology is going to be changed raises a lot of doubts on seeing this technology applied to non-huge AAA developments, with their own engines, in a short-term time. It is expected in the future more APIs and commercial to integrate their Raytracing modules and specifically Unity, to develop a solution to these errors that, at the moment, is not usable. There is also the huge change of paradigm on the architecture of the graphic cards, that it is discussed in first chapter, that is going to need some time for the public to be accepted.

- Results on RTAO were not overwhelmingly superior in terms of visual appeal: Taking into account the fact that it is implemented what it is considered to be a very cheap but effective trick to get Ambient Occlusion, the results on RTAO where not that much more visually attractive than the ones produced in the deliverable. Although physically correct and with no more artifacts than the noise and the reprojection issue, it is believed that the videogames industry is mostly an artistic field, and just looking for realistic or physically correct effects such as ambient occlusion is not the way to go. It is believed that there are other effects such as reflections which do make more sense to be RT integrated as there is a much greater difficulty to approximate them from the artist point of view, but AO can be heavily tweaked to create many different atmospheres and auras in a game that are simply not reallist, and definitely cutting the artist ideas after a tirelessly research on the most reallist image seems as comparing an oil canvas to a digital photography, there is a loss. Also keep in mind that the effect here represented is just a basic AO and there is plenty of room for improvements, see Critical Evaluation.

- The performance gap: Although it makes no sense to compare both deliverable and RT in Unity, there is no doubt that there are some serious implications in terms of performance when enabling RTAO in Unity against using another solution. In an industry where the 16 ms rule (up to 35 ms can be acceptable in these circumstances) is taken very strictly, as it is demanded by the vast majority of the public, and spending up to 30ms just rendering and calculating AO on a part of the Sponza scene it is not worth. Also mind that the RT test was run with no extra logic, or assets of any other type that games usually integrate requesting computer resources. All in all, this makes RT still not suitable for almost any production purposes (apart from AAA games, offline rendering and research).

Chapter V

Critical Evaluation

5.1 Introduction

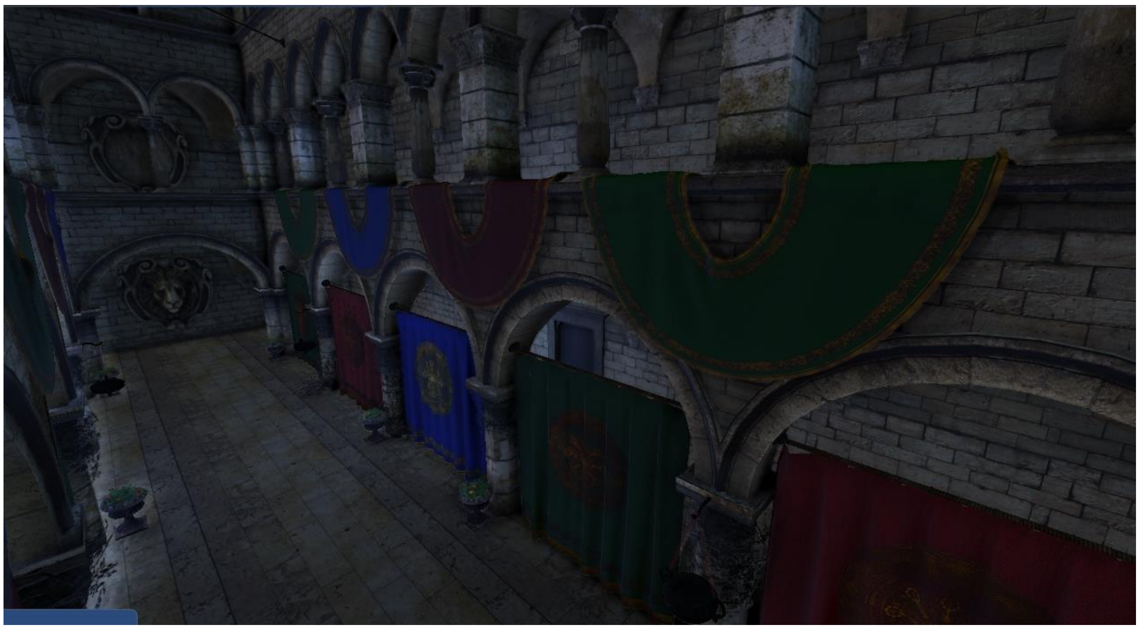
In this chapter, the results of the project development are analysed and discussed, pointing out the learning outcomes of it. The post-mortem analysis is attached and the personal thoughts on this project are also exposed. In the end, some of the improvements that were not taken into account during this development are studied to reflect the results of the overall research.

5.2 About the result of the project

I would not be honest to myself if I said that the project was completely successful as I had not met every milestone I put to myself, but this undoubtedly was not a complete failure. I would like to revisit each of the objectives of the project and analyse its result independently so the overall can be measured objectively.

The first objective was to learn DirectX 12, I wanted to expand my knowledge in this field and learn on how to use this API: (please refer to Chapter 3.1 to expand on the reasons of this selection.). This objective is assured and reflected in (Chapter 3.2, Chapter 3.3). The second target of this project was to successfully implement a fully compliant PBR renderer using the DirectX knowledge I acquired during the first part of the project. This task was also successfully completed, can be tested in the deliverable and its report can be found in (Chapter 3.3). An advanced rasterization technique, Image Based Lighting was also implemented and it is detailed in (Chapter 3.3).

The culmination of this project was the implementation of the DirectX's DXR Raytracing module into the renderer and the introduction of Ambient Occlusion as a replacement of a rasterized SSAO (Chapter 3.4, Chapter 3.5). This would have provided the most accurate data I needed to perform comparisons between a rasterized and a raytraced approach to the physically accurate ambient occlusion representation. But at the moment I am writing this chapter of the dissertation I could only integrate DXR into a separate solution and program a minor test that it is not integrated into the main executable.


Screenshot of the final result of the engine

Analysing the final result, the renderer looks compliant and the results it offers are visually appealing, if the user gets into the code it can also observe that some advanced abstractions were implemented and that this layer is capable to be extended, but some of the aims defined at the project specification were not met.

5.3 Development Reports Post-mortem

This subsection covers the technical issues and main problems that arose during the work on each one of the different blocks of the project.

When developing the framework, A clear idea of one feature that was needed in the project, and it was to decouple as much as possible the inner API calls and management from the programming of the demo. This implied some tough work on the design of the framework, as I did not feel confident to develop a combined solution that fulfiled the needs taking into account that DirectX 12 supposes some big changes compared to any graphics API I have used before, I decided to follow the design guidances and tutorials of already premade solutions. With that, I would learn on the inner functioning of DirectX 12 at the time I would construct a solid decoupled framework in which program the final demo.

Although being extremely thankful to the fact that I could learn most of the usual functioning of this API without major problems, I underestimated the time that learning DirectX 12 by constructing this would take me, not failing the deadline, but overworking to meet it. The framework resulted to be a solid and verbose enough to be understood at every point but with some unnecesary parts that were not used in the final demo nor the previous tests.

If I could restart the time on this project but having this actual experience, I would definitely go after a more naïve solution on the deliverable, saving tons of time that was needed during other parts of the development.

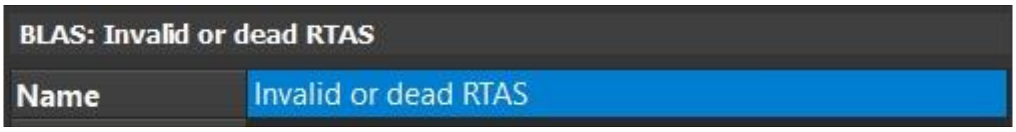
Regarding scene loading, material definition and the implementation of the PBR shader there were no major issues during development. I extended the framework by including some classes that covered the material definition, added the functionality of nodes in a Scene and programmed the workflow on loading the Sponza model and material hierarchy from an OBJ using Assimp. There is something I want to point out on the framework here: although having programmed mesh abstractions before, I found particularly interesting to stick to the one that the tutorials detailed, specially to the part of the vertex buffer information templatization to handle every type of vertex data declared.

Then I moved to the DXR implementation. At the beginning of this block, I still was sticking to the original plan, fulfilling every deadline I set to myself (please refer to Appendix A), and in general, I was very confident on finishing the project. As explained before this block consisted in 3 differentiated parts, isolated integration, integration to the framework and development of the effect. The isolated integration went as expected, there I could learn on some basic concepts of the raytraced pipeline such as how to build an acceleration structure or how shader binding tables store every relationship on the different resources needed on the raytrace pass.

The next part was definitely the turning point on the final result of the project, what I thought that should have been an easy part of the project turned out to be a wall I could not get through. The action plan on this was to use DXRHelpers abstraction classes provided by DirectX that included no more than some helpers which encapsulated the most tedious parts of the API calls, if everything worked out, I could integrate that into the overall framework to provide this additional feature, but it didn’t. I first had some trouble when reprogramming the functionality to be driven by the framework calls as many errors popped out for each method I ported, but these were almost negligible as they were corrected almost at the time as they popped out.

After I checked that the initialization was correctly ported (or that's what I thought) I proceeded to implement the render pass of the raytracing pipeline. Here, I had no major issues, apart from a bug that caused the program to shutdown, I could isolate the bug on `D3D12_DISPATCH_RAYS_DESC` descriptor, more specifically when setting up the hit groups section alignment and starting offsets of the shader binding table layout. After some unsuccessful checks that took around a week, I could solve that with the help of supervisor, the error turned out to be a hidden offset pointer calculation that was missaligned.

After that the program would run without breaking but the resulting output was nothing but the miss shader being called by every pixel. This bug was even difficult to track, as the initial thought was that I imported something wrongly from the external project, and I spent a lot of time reviewing each one of the methods I imported, which resulted to be beneficial, as this decision, and the enabling of `SetEnableGPUBasedValidation` and `SetEnableSynchronizedCommandQueueValidation` made some more Acceleration Structure generation bugs to appear that did not raise before but where making its creation call to not be pushed and executed on the command list. After I ensured that every initialization method was doing what it supposed to do I was still getting this output results, so I decided to use Nsight Graphics, the NVIDIA frame debugger to try to bring some light to this bug. I observed the following on the Acceleration Structure Viewer window:


BLAS: Invalid or dead RTAS

Name	Invalid or dead RTAS

It seemed that the Acceleration Structure was either not correctly generated or uploaded to the GPU. I tried to solve this in every way I could, by myself, asking supervisor, asking in official DirectX 12 discord chats. However, every possible solution or clue that I followed turned out not to be the correct to be followed in order to solve that.

There was a point in which I ran out of time while solving this, so I agreed with supervisor to gather the raytraced results that I needed on the report from any other place. And recalling the Engine subsection on the Selection of Tools, I remembered that Unity had a plug and play DirectX 12 raytracing module which I used to gather the RTAO results. There were no noticeable issues while using the engine apart from the ones already mentioned in other sections.

Looking this with hindsight, I would not have changed the initial plan of splitting the integration of DXR into the aforementioned separate pieces, but there was something at the point of starting to integrate the external solution that I felt to be wrong from the beginning, after some weeks away from that part of the code and briefly reviewing it with some fresh eyes I am not sure that the shader resource heaps are correctly managed but investigating this is something that now I have to leave for further improvements.

The final module is Ambient Occlusion, which was implemented in a hurry due to the time limitations that DXR implementation took me. However, the approach on this effect took no more problems that a headache caused by the HDR to SDR shader trying to convert the positions texture of the g-buffer and outputting strange artifacts to screen while debugging the implementation. There was also a strange reprojection issue that caused the AO to not be generated correctly, but I moved to the actual solution, which does not need any reprojections.

5.4 Personal Thoughts

I would like to start this section by personally dissecting some of the views during the development of the project about the parts that although not directly related to programming or research hereof, still impacted to its overall result.

I believe that the scope that first was defined at the beginning of the project was achievable but also pretty concise, as focused most of the investigation around Raytracing and its implementation. However, during the research I tried to keep the same level of in depth research in every of the other tasks of the deliverable for two basic reasons: I did not want to advance without a complete and in depth knowledge of the basics that were going to support the rest of the work, and, the more I further investigated on these two topics, the more I felt that the project needed to evolve in that way. So, I found that developing more and more the solid fundamentals about DirectX 12 and the investigation about PBR and other techniques made the scope too big to be handled in a single project at this level.

Regarding time management and the accomplishment of the deadlines presented in the action plan. In my judgement there were some delays, originated by the natural evolution I felt the project should follow, but in the end I feel this was not the main key causing the inconclusion of the project. Having extended studying DirectX 12 and PBR topics is a valid trade off that has permitted me to learn in depth about the real implications of these techniques instead of advancing at all costs and getting a result that although accomplishing every objective did not result as instructive as it end up being.

The arise of the Coronavirus COVID-19 however, has supposed a major problem for coursework in the university, as it forced me not only to move back to my country but to hugely extend working hours in two part-time remote jobs in order to help to sustain family’s economy. This issue, which still has an increasing tendency by the time I am writing this was one of the biggest hurdles I have ever faced, but also a great opportunity from which I could learn even though the project wasn’t successful.

From the project I do take a solid understanding of Direct3D 12 API, the implications it has compared to OpenGL and even other versions of DirectX in terms of resource management and structure, and its optimization and abstraction from the hardware layer. I do also have learned about developing a Physically Based Rendering approach to solve the rendering equation, these two resulted in the most visually appealing renderer I have developed so far, and I am proud of that. An overall lesson I learned is that I enjoyed way much more the higher levels of programming, specially the work and the efforts put in the shaders, rather than working the Direct3D API calls abstraction structures or any other graphics work from C++, so I would like to find a way to focus just in that part of the development and expand on that.

5.5 Further Improvements

This section will expand in some of the improvements that should be included if I was given the time with the current state of the deliverable. These include the complete integration of the Raytracing module and its demonstration inside the main executable which I would consider as a formal closure of the project. Apart from that, I consider this renderer as a starting point for future research in the field of shading techniques in DirectX.

Regarding the current state of the project, my first improvement would be to take a closer look into the management of the heaps both from the framework and the raytracing sides of the project, as I feel that the error that stopped the development in that way can be related to that. Whether I found the error there or in any other piece of the code, I would like to keep advancing on the raytracing implementation and completing the demo as I first designed it.

After that I would like to better re integrate both the scene and material management in a more generic way. Due to squeezing deadlines these turned out to become an adhoc solution to load the Sponza model and the PBR shaders but are not scalable or reusable for any other type of material. Expanding on the PBR part I would also like to work on some shaders to work out the complete image based lighting contribution, which at the moment is just constrained to the diffuse part, for that I would need to be able to write on cubemaps from shaders, which is something I have tried but could not work out yet with the current framework.

Finally, I would try to improve the Ambient Occlusion technique and integrate a more modern solution such as HBAO or HBAO+ (Bavoil L., Sainz M.,2008). This improvement has two main reasons, the first one is that I would like to see up to what point I get rid from the current artifacts in my implementations, which mainly are haloing and strange cuts on the edges of the screen. The second reason is that I would like to better compare against the raytracing solution and finally check if, as I fear from the current results on this report, it is still not worth to switch to an RT pipeline at least when it comes to Ambient Occlusion

Chapter VI

Bibliography

Basurco, S. (2017, May 28). Retrieved from https://chuckleplant.github.io/2017/05/28/light-shafts.html

Bavoil L., Sainz M. (2008) Image-Space Horizon-Based Ambient Occlusion. Retrieved from https://developer.download.nvidia.com/presentations/2008/SIGGRAPH/HBAO_SIG08b.pdf

Benty, N., Clarberg, P., & McGuire, M. (2018). The Ray + Raster Era Begins - an R&D Roadmap for the Game Industry. Retrieved from GDC Talk website: https://www.gdcvault.com/play/1024814/The-Ray-Raster-Era-Begins

Boulos, S., Edwards, D., Lacewell, J. D., Kniss, J., Kautz, J., Shirley, P., & Wald, I. (2007). Packet-based whitted and distribution raytracing. Proceedings - Graphics Interface, 177–184. https://doi.org/10.1145/1268517.1268547

Burnes, A. (2018). Raytracing, Your Questions Answered: Types of Raytracing, Performance On GeForce GPUs, and More. Retrieved from https://www.nvidia.com/en-us/geforce/news/geforce-gtx-dxr-raytracing-availablenow/

Christensen, P. H., & Jarosz, W. (2016). The path to path-Traced movies. Foundations and Trends in Computer Graphics and Vision, 10(2), 103–175. https://doi.org/10.1561/0600000073

Christensen, P. H., Fong, J., Laur, D. M., & Batali, D. (2006). Raytracing for the movie “Cars.” RT’06: IEEE Symposium on Interactive Raytracing 2006, Proceedings, 1–6. https://doi.org/10.1109/RT.2006.280208

Cook, R. L., & Torrance, K. E. (1982). A Reflectance Model for Computer Graphics. Retrieved from http://inst.cs.berkeley.edu/~cs294-13/fa09/lectures/cookpaper.pdf

Coppock, M. J. (2019, June 8). Direct3D* 12 Overview Part 4: Heaps and Tables. Retrieved from https://software.intel.com/en-us/blogs/2014/08/07/direct3d-12-overview-part-4-heaps-and-tables

DiligentGraphics. (n.d.) Variable Size Memory Allocations Manager. Retrieved from http://diligentgraphics.com/diligent-engine/architecture/d3d12/variable-size-memory-allocations-manager/

Direct 3D. (2018). Announcing Microsoft DirectX Raytracing! Retrieved from https://devblogs.microsoft.com/directx/announcing-microsoft-directx-raytracing/

DX12 Raytracing tutorial - Helpers. (2020, January 17). Retrieved from https://developer.nvidia.com/rtx/raytracing/dxr/DX12-Raytracing-tutorial/dxr_tutorial_helpers

DX12 Raytracing tutorial - Part 1. (2020, January 17). Retrieved from https://developer.nvidia.com/rtx/raytracing/dxr/DX12-Raytracing-tutorial-Part-1

Galvan, A. (2020). A Comparison of Modern Graphics APIs. Retrieved from https://alain.xyz/blog/comparison-of-modern-graphics-apis

Getting started with DirectX raytracing in Unity - SIGGRAPH 2019 (2019, September 3). Retrieved from https://www.youtube.com/watch?v=DoKOCgNel7E

Hoffman, N. (2012). SIGGRAPH University - Introduction to "Physically Based Shading in Theory and Practice" & Physics and Math of Shading. Acm Siggraph 2012. Retrieved from https://www.youtube.com/embed/j-A0mwsJRmk

Kajalin V. (2009) - Screen Space Ambient Occlusion. Retrieved from ShaderX7, edited by Wolfgang Engel, Chapter 6.1. .

Karis, B. (2013) Real Shading in UE4. Retrieved From https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf

Karis, B. (2013). Specular BRDF Reference. Retrieved from http://graphicrants.blogspot.com/2013/08/specular-brdf-reference.html

Koch, D. (2020, March 17). Raytracing In Vulkan. Retrieved from https://www.khronos.org/blog/raytracing-in-vulkan

Liu, E., & Llamas, I. (2018). GDC Raytracing in Games with NVIDIA RTX (Presented by NVIDIA). Retrieved from https://gdcvault.com/play/1024813/Raytracing-inGames-with

Mendez, J. M. (2010, May 25). A Simple and Practical Approach to SSAO. Retrieved from https://www.gamedev.net/tutorials/programming/graphics/a-simple-and-practical-approach-to-ssao-r2753/

Merry, S. [Microsoft DirectX 12 and Graphics Education] (2015, July 29) DirectX 12: Resource Barriers and State Tracking. Retrieved from https://www.youtube.com/watch?v=nmB2XMasz2o&feature=youtu.be

NVIDIA Turing Whitepaper. (2018). Retrieved from https://www.nvidia.com/content/dam/en-zz/Solutions/design-20 visualization/technologies/turing-architecture/NVIDIA-Turing-ArchitectureWhitepaper.pdf

OpenGL - Will it ever have RTX Raytracing support? (2019, August 28). Retrieved from https://community.khronos.org/t/opengl-will-it-ever-have-rtx-raytracing-support/104489/2

Panteleev, A. (2016, August 4). VXAO: Voxel Ambient Occlusion. Retrieved from https://developer.nvidia.com/vxao-voxel-ambient-occlusion

PBR Texture Conversion. (2018, November 1). Retrieved from https://marmoset.co/posts/pbr-texture-conversion/

Pharr, M., Humphreys, G., & Hanrahan, P. (2018). Physically Based Rendering: From theory to implementation, 3rd edition. Retrieved from http://www.pbr-book.org/3ed2018/contents.html

Sandy, M., Andersson, J., & Barré-Brisebois, C. (2018). DirectX: Evolving Microsoft's Graphics Platform. Retrieved from https://devblogs.microsoft.com/directx/wpcontent/uploads/sites/42/2018/03/GDC_DXR_deck.pdf

Scratchapixel. (2014, August 15). Retrieved from https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview/light-transport-ray-tracing-whitted

Shirley, P., Wyman, C., & McGuire, M. (2019). INTRODUCTION TO REAL-TIME RAYTRACING. Retrieved from http://rtintro.realtimerendering.com/

Stich, M. (2018). Introduction to NVIDIA RTX and DirectX Raytracing. Retrieved from https://devblogs.nvidia.com/introduction-nvidia-rtx-directx-raytracing/

Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, S. H. (2018). Real-Time Rendering, Fourth Edition. 1198 pages.
van Oosten, J. (2019, August 27). Learning DirectX 12 - Framework. Retrieved from https://www.3dgep.com/learning-directx-12-3/

Vries, J. de. (n.d.). LearnOpenGL: Theory of PBR. Retrieved from https://learnopengl.com/PBR/Lighting

Whitted, T. (n.d.). An Improved Illumination Model for Shaded Display. Retrieved from https://artis.imag.fr/Members/David.Roger/whitted.pdf

Wyman, C., Hargreaves, S., Shirley, P., & Barré-Brisebois, C. (2018). Introduction to DirectX Raytracing. Retrieved from https://www.youtube.com/watch?v=Q1cuuepVNoY

Chapter VI

Appendices

Appendix A: Project Specification and Ethics Form

PROJECT SPECIFICATION - Project (Technical Computing) 2019/20

Student:	**Diego Lloréns Rico**
Date:	**18/10/19**
Supervisor:	**Thomas Sampson**
Degree Course:	**BSc (Hons) Computer Science for Games**
Title of Project:	**Ray Tracing under the microscope:** **Performance test of Ray tracing cutting edge techniques in a real time environment**

Elaboration

Stated by NVIDIA as the Holy Grail of Graphics, there is a rendering algorithm that, although being widely extended among offline environments, such as cinema (Pixar) or architecture renders, it has always been rejected in real time environments due to its extreme computational cost (remember that gaming industry is working right now at 33ms per rendered frame), we are talking about Ray Tracing.

The latest technology advances in the last year, the release of NVIDIA's RTX GPUs, Sony's PlayStation 5 announcement, or the latest DirectX 12 API have rocketed equally the excitement and the scepticism around this whole new rendering approach that promises to be the future of gaming.

The aim of this project is to develop a program that successfully showcases an implementation of a ray tracing cutting edge technique and delves into this technology, by implementing it in a classic rasterizing deferred PBR pipeline and optimizing it to match **real-time performance requirements**.

My main objective is to research about ray tracing techniques such as Reflections, Ambient Occlusion, or Shadowing and successfully implement and measure the impact of at least **one of them,** by using ray tracing techniques and discussing in a report on how this approach differs from its classic rasterizing counterpart. Deliverable, however, will focus only on the Ray Tracing implementation and optimization, as its main objective is to successfully implement, optimize and measure a C++ / DirectX 12 implementation of some of these techniques.

Project Aims

The project aims to achieve the following:

- Gain an in-depth understanding of rasterizing and ray tracing pipelines by integrating them into a combined project that gets the most out of both approaches.
- Study and implement advanced optimization graphic techniques (the chosen techniques will vary depending on the implemented ray tracing effect)
- Take advantage of NVIDIA RTX Graphics cards optimizations regarding Ray Tracing and DLSS to test the real value that these technologies can add to modern games
- Learn about DirectX 12 latest Ray Tracing API
- Use a project planning tool and a version control system to organize and plan workflow
- Evaluate the feasibility of different Ray Tracing techniques and its performance impact in a realtime environment
- Provide the final user of a demo where the performance of Realtime Ray tracing techniques can be tested in its computer.

Project deliverable(s)

The primary deliverable will consist in a PC executable DEMO that will be tested under the following conditions:

OS	Windows 10 Home 64 bits (10.0,18362)
Processor	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz (12 CPUs), ~2.2GHz
Memory	16 Gigabytes
DirectX version	DirectX 12
Graphics Card	NVIDIA GeForce RTX 2080 (16Gb)
Monitor Resolution	1920x1080p

As the main purpose of this project is to **MEASURE** the performance of a ray tracing technique in real time, the following environment will be set up:

- A 3D PBR Deferred rendered scene (**3D Sponza scene**).
- A multiple moving camera setup that will swap every *N* second or via user input.
- A checkbox list of implemented graphic effects so the user can enable, disable them.
- Some kind of **runtime performance CPU and GPU report**, so the project can be measured and evaluated objectively.

In order to achieve this, the **program developed** should be capable of the following:

- Load geometries.
- Perform Basic Transform operations with this Geometry.
- Render geometries in a node-based graph scene using a **PBR Deferred Shading** technique.
- Enable/Disable and measure the impact of **at least one** of these Ray Tracing Techniques in Runtime (the technique(s) will be chosen after the research stage of the project, please see ***Action Plan***):
    - Raytraced shadows
    - Ambient Occlusion
    - Reflections
    - Global Illumination
- Trace the impact in milliseconds of every developed process so its performance can be objectively tested and optimized if needed.

(NOTE: The project will be considered successful if at least 1 of these techniques is correctly implemented and optimized to match video game performance requirements, 60FPS at 1080p)

Action plan

Module Deadlines

Task	Deadline
Project Specification & Ethics Form	Friday 25th October 2019

Background Research
 	
1. Research into Game Engine Architecture
  a. Bootstrap, HAL & Class Organization and structure…
  b. Multithread approaches, Command lists
  c. Main loop
  d. Geometry Loading
 	
2. Research into DirectX 12 and integration into the engine
  a. Wrapper vs raw DirectX 12
  b. DirectX12 workflow
  c. Multithreading and DirectX 12
  d. DirectX12 Resources and management
 	
3. Research into Deferred rendering
  a. Differences between forward and deferred rendering
  b. Deferred rendering pros and cons
  c. Deferred rendering requirements
 	
4. Research into PBR
	Friday 29th November 2019
	
	1. Friday 8th November 2019
	2. Friday 8th November 2019
	3. Friday 15th November 2019
	4. Friday 22th November 2019
	5. Friday 29th November 2019

5. Research into Ray tracing Techniques a. Shadows (To be discussed with project supervisor)	
Information Review	Friday 6th December 2019

Project Development
 	
1. Implement Base Engine Features
 	
  a. Main Loop
  b. DirectX 12 Rendering Framework
	Friday 14th February 2020
	
	1. Friday 13th December 2019
	2. Friday 17th January 2020
	3. Friday 14th February 2020

c. Geometry Loading
d. Transform Class
e. Node Visualization
f. Multi Thread

2. Implement Rendering Techniques a. Deferred Rendering b. PBR **3. Implement Ray tracing techniques** a. Shadows (To be discussed with project supervisor)	
Provisional Contents Page	Friday 21st February 2020
Draft Critical Evaluation	Friday 27th March 2020
Draft Report	Friday 27th March 2020
Project Report Hand in through Turnitin	Wednesday 22nd April 2020
Project Report & Electronic Copies of Deliverable	Thursday 23rd April 2020
Demonstration of Work	Agreed before Tuesday 12th May 2020

BCS Code of Conduct

I confirm that I have successfully completed the BCS code of conduct on-line test with a mark of 70% or above. This is a condition of completing the Project (Technical Computing) module.

Signature:

Publication of Work

I confirm that I understand the "Guidance on Publication Procedures" as described on the Bb site for the module.

Signature:

GDPR

I confirm that I will use the "Participant Information Sheet" as a basis for any survey, questionnaire or participant testing materials. This form is available on the Bb site for the module.

Signature:

Ethics

Complete the SHUREC 7 (research ethics checklist for students) form below. If you think that your project may include ethical issues that need resolving (working with vulnerable people, testing procedures, etc.) then discuss this with your supervisor as soon as possible and comment further here.

Both you and your supervisor need to sign the completed SHUREC 7 form.

Please contact the project coordinator if further advice is needed.

RESEARCH ETHICS CHECKLIST FOR STUDENTS (SHUREC 7)

This form is designed to help students and their supervisors to complete an ethical scrutiny of proposed research. The SHU Research Ethics Policy should be consulted before completing the form.

Answering the questions below will help you decide whether your proposed research requires ethical review by a Designated Research Ethics Working Group.

The final responsibility for ensuring that ethical research practices are followed rests with the supervisor for student research.

Note that students and staff are responsible for making suitable arrangements for keeping data secure and, if relevant, for keeping the identity of participants anonymous. They are also responsible for following SHU guidelines about data encryption and research data management.

The form also enables the University and Faculty to keep a record confirming that research conducted has been subjected to ethical scrutiny.

For student projects, the form may be completed by the student and the supervisor and/or module leader (as applicable). In all cases, it should be counter-signed by the supervisor and/or module leader, and kept as a record showing that ethical scrutiny has occurred. Students should retain a copy for inclusion in their research projects, and staff should keep a copy in the student file.

Please note if it may be necessary to conduct a health and safety risk assessment for the proposed research. Further information can be obtained from the Faculty Safety Co-ordinator.

General Details

Name of student	Diego Lloréns Rico
SHU email address	Diego.LlorensRico@student.shu.ac.uk Diego.LlorensRico@student.shu.ac.uk
Course or qualification (student)	BSc (Hons) Computer Science For Games
Name of supervisor	Thomas Sampson
email address	acests6@exchange.shu.ac.uk
Title of proposed research	Ray Tracing under the microscope: Performance test of Ray tracing cutting edge techniques in a real time environment
Proposed start date	Monday 28 th October 2019
Proposed end date	Agreed before Tuesday 12 th May 2020
Brief outline of research to include, rationale & aims (250-500 words).	Stated by NVIDIA as the Holy Grail of Graphics, there is a rendering algorithm that, although being widely extended among offline environments, such as cinema (Pixar) or architecture renders, it has always been rejected in real time environments due to its extreme computational cost (remember that gaming industry is working right now at 16ms per rendered frame), we are talking about Ray Tracing. The latest technology advances in the last year, the release of NVIDIA's RTX GPUs, Sony's PlayStation 5 announcement, or the latest DirectX 12 API have rocketed equally the excitement and the scepticism around this whole new rendering approach that promises to be the future of gaming. The aim of this project is to develop a program that successfully showcases an implementation of a ray tracing cutting edge technique and delves into this technology, by implementing it in a classic rasterizing deferred PBR pipeline and optimizing it to match real-time performance requirements.

	My main objective is to research about ray tracing techniques such as Reflections, Ambient Occlusion, or Shadowing and successfully implement and measure the impact of at least one of them, by using ray tracing techniques and discussing in a report on how this approach differs from its classic rasterizing counterpart. Deliverable, however, will focus only on the Ray Tracing implementation and optimization, as its main objective is to successfully implement, optimize and measure a C++ / DirectX 12 implementation of some of these techniques.

	The project aims to achieve the following: ● Gain an in-depth understanding of rasterizing and ray tracing pipelines by integrating them into a combined project that gets the most out of both approaches. ● Study and implement advanced optimization graphic techniques (the chosen techniques will vary depending on the implemented ray tracing effect) ● Take advantage of NVIDIA RTX Graphics cards optimizations regarding Ray Tracing and DLSS to test the real value that these technologies can add to modern games ● Learn about DirectX 12 latest Ray Tracing API ● Use a project planning tool and a version control system to organize and plan workflow ● Evaluate the feasibility of different Ray Tracing techniques and its performance impact in a real-time environment Provide the final user of a demo where the performance of Realtime Ray tracing techniques can be tested in its computer.
Where data is collected from individuals, outline the nature of data, details of anonymisation, storage and disposal procedures if required (250-500 words).	

1. Health Related Research Involving the NHS or Social Care / Community Care or the Criminal Justice Service or with research participants unable to provide informed consent

Question	**Yes/No**
1. Does the research involve? • Patients recruited because of their past or present use of the NHS or Social Care • Relatives/carers of patients recruited because of their past or present use of the NHS or Social Care • Access to data, organs or other bodily material of past or present NHS patients • Foetal material and IVF involving NHS patients • The recently dead in NHS premises • Prisoners or others within the criminal justice system recruited for health-related research***** • Police, court officials, prisoners or others within the criminal justice system***** • Participants who are unable to provide informed consent due to their incapacity even if the project is not health related	No
2. Is this a research project as opposed to service evaluation or audit? *For NHS definitions please see the following website* http://www.hra.nhs.uk/documents/2013/09/defining-research.pdf	No

If you have answered **YES** to questions **1 & 2** then you **must** seek the appropriate external approvals from the NHS, Social Care or the National Offender Management Service (NOMS) under their independent Research Governance schemes. Further information is provided below.

NHS https://www.myresearchproject.org.uk/Signin.aspx

* All prison projects also need National Offender Management Service (NOMS) Approval and Governor's Approval and may need Ministry of Justice approval. Further guidance at: http://www.hra.nhs.uk/research-community/applying-for-approvals/national-offender-management-service-noms/

NB FRECs provide Independent Scientific Review for NHS or SC research and initial scrutiny for ethics applications as required for university sponsorship of the research. Applicants can use the NHS pro-forma and submit this initially to their FREC.

2. Research with Human Participants

Question	Yes/No
Does the research involve human participants? This includes surveys, questionnaires, observing behaviour etc.	No
Question	**Yes/No**
1. *Note If YES, then please answer questions 2 to 10* *If NO, please go to Section 3*	
2. Will any of the participants be vulnerable? *Note: Vulnerable' people include children and young people, people with learning disabilities, people who may be limited by age or sickness, etc. See definition on website*	
3. Are drugs, placebos or other substances (e.g. food substances, vitamins) to be administered to the study participants or will the study involve invasive, intrusive or potentially harmful procedures of any kind?	
4. Will tissue samples (including blood) be obtained from participants?	
5. Is pain or more than mild discomfort likely to result from the study?	
6. Will the study involve prolonged or repetitive testing?	
7. Is there any reasonable and foreseeable risk of physical or emotional harm to any of the participants? *Note: Harm may be caused by distressing or intrusive interview questions, uncomfortable procedures involving the participant, invasion of privacy, topics relating to highly personal information, topics relating to illegal activity, etc.*	
8. Will anyone be taking part without giving their informed consent?	
9. Is it covert research? *Note: 'Covert research' refers to research that is conducted without the knowledge of participants.*	
10. Will the research output allow identification of any individual who has not given their express consent to be identified?	

If you answered **YES only** to question **1,** the checklist should be saved and any course procedures for submission followed. If you have answered **YES** to any of the other questions you are **required** to submit a SHUREC8A (or 8B) to the FREC. If you answered **YES** to question **8** and participants cannot provide informed consent due to their incapacity you must obtain the appropriate approvals from the NHS research governance system. Your supervisor will advise.

3. Research in Organisations

Question	**Yes/No**
1. Will the research involve working with/within an organisation (e.g. school, business, charity, museum, government department, international agency, etc.)?	No
2. If you answered YES to question 1, do you have granted access to conduct the research? *If YES, students please show evidence to your supervisor. PI should retain safely.*	
3. If you answered NO to question 2, is it because: A. you have not yet asked B. you have asked and not yet received an answer C. you have asked and been refused access. *Note: You will only be able to start the research when you have been granted access.*	

4. Research with Products and Artefacts

Question	**Yes/No**
1. Will the research involve working with copyrighted documents, films, broadcasts, photographs, artworks, designs, products, programmes, databases, networks, processes, existing datasets or secure data?	No
2. If you answered YES to question 1, are the materials you intend to use in the public domain? *Notes: 'In the public domain' does not mean the same thing as 'publicly accessible'.* • *Information which is 'in the public domain' is no longer protected by copyright (i.e. copyright has either expired or been waived) and can be used without permission.* • *Information which is 'publicly accessible' (e.g. TV broadcasts, websites, artworks, newspapers) is available for anyone to consult/view. It is still protected by copyright even if there is no copyright notice. In UK law, copyright protection is automatic and does not require a copyright statement, although it is always good practice to provide one. It is necessary to check the terms and conditions of use to find out exactly how the material may be reused etc.* *If you answered YES to question 1, be aware that you may need to consider other ethics codes.* *For example, when conducting Internet research, consult the code of the Association of*	

Internet Researchers; for educational research, consult the Code of Ethics of the British Educational Research Association.	
3. If you answered NO to question 2, do you have explicit permission to use these materials as data? *If YES, please show evidence to your supervisor.*	
4. If you answered NO to question 3, is it because: A. you have not yet asked permission B. you have asked and not yet received and answer C. you have asked and been refused access. *Note: You will only be able to start the research when you have been granted permission to use the specified material.*	**A/B/C**

Adherence to SHU policy and procedures

Personal statement	
I can confirm that: – YES, I have read the Sheffield Hallam University Research Ethics Policy and Procedures – YES, I agree to abide by its principles.	
Student	
Name: Diego	Date: Llorens Rico
Signature: D.Llorens Rico	
Supervisor or other person giving ethical sign-off I can confirm that completion of this form has not identified the need for ethical approval by the FREC or an NHS, Social Care or other external REC. The research will not commence until any approvals required under Sections 3 & 4 have been received.	
Name: Thomas Sampson	Date: 25/10/2019
Signature: T.Sampson	

Appendix B: Project Installation Requirements

Both the deliverable and the Unity project are self contained and should include every dependency to be tested.

To build the deliverable go to Project_Deliverable folder and click on GenerateProjectFiles, this should run a cmake file that will build the solution used, after that, just compile and execute the deliverable.

In order to run the Unity project some more specifications are required. It is needed to download Unity at its version 2020.1.0b7 and open the project under RTUnity folder. This project will only run with a NVIDIA graphic card of the following:

- NVIDIA Volta (Titan X)
- NVIDIA Turing (2060, 2070, 2080, and their TI variants)
- NVIDIA Pascal (1060, 1070, 1080 and their TI variants)

Also ensure the drivers on the graphics card are up to date.

After opening the project, please head to the HDR Wizard Pipeline (inside Window > Render Pipeline), and check that under the tab HDR + DXR there are no assets that need to be fixed, if so, please press the Fix All button. That should fully enable DXR on the machine.