A real time hybrid pipeline in Vulkan

Florentí Solano, Pau

Curs 2020-2021

Director: JAVI AGENJO

GRAU EN ENGINYERIA INFORMÀTICA

Universitat Pompeu Fabra Barcelona
Escola Superior Politècnica

Treball de Fi de Grau

Agraïments

I would like to thank my supervisor Javier Agenjo for his input and support throughout the project. Also my family and friends for being very supportive from the beginning.

Resum

Les aplicacions gràfiques interactives d'avui en dia, especialment en l'industria dels videojocs, han augmentat la demanda en la versemblança i realisme de les seves imatges. Fins fa ben poc, les aplicacions gràfiques interactives estaven totalment dominades per algorismes de rasterització però amb la potència de les últimes gràfiques i les noves APIs el traçat de rajos s'ha convertit en un nou component en la renderització d'imatges a temps real.

Aquest projecte consisteix en proposar una pipeline gràfica que faci ús tant de la rasterització com del traçat de rajos i sigui interactiva de forma òptima per tal de treure el màxim partit d'ambdós algorismes.

Resumen

Las aplicaciones gráficas interactivas de hoy en día, especialmente en la industria de los videojuegos, han aumentado la demanda en la verosimilitud y realismo de sus imágenes. Hasta hace bien poco, las aplicaciones gráficas interactivas estaban totalmente dominadas por algoritmos de rasterización pero con la potencia de las últimas gráficas y las nuevas APIs el trazado de rayos se ha convertido en un nuevo componente en la renderización de imágenes a tiempo real.

Este proyecto consiste en proponer una pipeline gráfica que use tanto la rasterización como el trazado de rayos y sea interactiva de forma óptima con el objetivo de sacar el máximo partido a ambos algoritmos.

Abstract

Interactive graphic applications nowadays, specially in the video game industry, have increased the demand on realism and plausibility of its images. Until recently, interactive graphic applications were exclusively in the rasterisation domain but with the power of the latest graphics processing units and the newest APIs, ray tracing has become in a new component in the rendering of real time images.

This project proposes a graphic pipeline using both rasterisation and ray tracing algorithms and allows its interaction optimally with the aim of taking the best of both paradigms.

Contents

Agraïments iii

Abstract v

1 Introduction 1
1.1 Introduction 1
1.2 Background 1
1.3 Objectives 2
1.4 Methodology 2
1.5 Thesis Structure 2

2 State of the Art 3
2.1 Rasterised Rendering 3
2.2 G-Buffer 4
2.3 Ray tracing 5
2.3.1 Ambient Occlusion and Global illumination 6
2.4 Shading 7
2.4.1 Microfacets 7
2.4.2 Energy conservation principle 7
2.4.3 Subsurface scattering 8
2.4.4 The render equation 9

3 Why Vulkan? 11
3.1 Initialisation 11
3.2 Executing Commands 12

4 Proposed Solution 13
4.1 The pipeline 13
4.2 Rasterisation rendering 15
4.3 Data organisation 18
4.4 Tracing Rays 19
4.4.1 Acceleration Structures 21
4.4.2 Ray tracing descriptors and shaders 22
4.4.3 Shadows 25
4.4.4 Reflection and Refraction 31

5 Results 38

6 Conclusion 41

7 Future Work 42
7.1 Reducing rays for reflection and refraction 42
7.2 Reducing rays for shadows 42
7.3 Acceleration Structure Instancing 42
7.4 Multi-Threading 42

8 Annex **44**

List of Figures

1	rasterisation pipeline[1]	4
2	Deferred shading using G-Buffers.[2]	5
3	Microfacets at a microscopic level for rough and smooth surfaces.[3]	7
4	Light beam being scattered when hitting a surface.[3]	8
5	Light gets scattered when entering, penetrating a surface and exits at a different point.[4]	9
6	Render equation.	9
7	Primary rays are shot from camera position into the scene for each pixel in the screen. Then, each ray computes intersection calculations with geometry. The return [5]	14
8	Pipeline solution scheme.	15
9	(a) Position (b) Normal Vectors (c) Albedo (d) Material (e) Motion Vectors (f) Emissive texture	17
10	Unpacking vertex data in a ray-tracing pass.	19
11	Primary rays are shot from camera position into the scene for each pixel in the screen. Then, each ray computes intersection calculations with geometry. If a ray intersects any geometry the subsequent rays can be traced to calculate shadows or illumination techniques.[5]	20
12	Acceleration Structure composition[6]	22
13	traceRaysEXT() definition.	23
14	Ray tracing pipeline logic.	24
15	Soft shadow composition[7]	27
16	Shadows traced with 1spp without accumulation.	29
17	Mix function.	29
18	Motion vector calculations.	30
19	Shadows traced with 1spp with spatio-temporal accumulation.	31
20	Shader snippet. Computing the reflective properties to be returned in the ray-generation shader.	32
21	Ray traced perfect specular reflections.	33
22	Shader snippet. Computing the refractive properties to be returned to the ray-generation shader.	34
23	Ray traced perfect specular refraction.	35
24	Returning the payload in the closest-hit shader.	37
25	Launching rays until max recursion value is reached or no more rays are needed.	37
26	System properties used for development and testing.	38
27	Fully ray-traced scene	38
28	Hybrid scene	39
29	Average frame rate statistic of the dynamic scene.	39
30	Hybrid frame profiling.	40
31	Frame with number of reflection and refraction meshes increased.	40
32	Frame with geometry increased.	41

1 Introduction

1.1 Introduction

Computer graphics have always been evolving. It has not reached a static point where a final or perfect solution has been established for all cases. From the beginning, where flashing pixels were displayed on the screen until now where complex geometries can be rendered, the field has constantly been changing to achieve the illusion of reality. But for the most time, this ability to create realistic images was not able for real-time applications. It was mostly focused on the movie industry, which could afford the huge computational cost and rendering times.

Even with the latest hardware, real-time applications cannot achieve a result similar to the movies and pre-computed photo-realistic results. However, improvements and new techniques make it much closer to reality than ever before. The goal of graphically simulating reality has been in the computer games scene for a while now, and the industry is pushing forward in the field due to a huge demand for photo-realistic images.

But due to hardware constraints, this goal has become a daunting task for developers. Such limitations made it impossible to compute accurate physical environments in real-time until recently. With the latest graphical processing units, developers can trace rays in real-time, allowing them to compute how light would behave in reality.

1.2 Background

Due to demands and competition, computer games have grown in complexity and fidelity. Modern systems aim to simulate reality, but games are not only based on their rendering system but also game logic, mechanics, physics, artificial intelligence, and much more. The fact that games are built around so many pieces makes game engines' resources limited since the application is expected to run at an average of 60 frames per second. It is because of that that games use rasterisation as the main rendering method. The main purpose of GPUs is to draw millions of triangles in parallel very fast, and that allows developers to draw images of very complex meshes in very short periods. However, they are not very good at computing how reality works. Current real-time techniques have difficulties dealing with effects caused by light interacting with multiple or different surfaces and materials. In order to achieve fidelity, developers must be able to trace rays to compute how light would behave in the real world and simulate effects such as reflection, refraction, or soft shadows. This algorithm, known as ray tracing, has existed for a long time, but due to its computational cost, it has been kept to only pre-computed images such as the ones seen in movies.

With the release of the new RTX graphics cards line by NVIDIA in September 2018 and subsequent API (Application Programming Interface) versions of DirectX and Vulkan, users were able to run programs that traced rays. This new hardware was specially manufactured to bring real-time ray tracing to a consumer level. Still, the computational cost of ray tracing a whole scene is too elevated to bring it to a game nowadays, and thus it is necessary to combine both worlds to achieve the most believable image.

1.3 Objectives

This end-of-grade project aims to study and propose a graphics pipeline that allows the combination of both rasterisation and ray tracing. The following topics are objectives for the research being proposed

1. To develop and implement a 3D graphical rendering engine that takes advantage of the Vulkan ray tracing extensions.
2. To combine both rasterisation and ray tracing pipeline to build a functional real-time scene.
3. To optimise the number of rays traced.

1.4 Methodology

Once the project have been introduced, let us take a look at how the project was planned.

First and foremost, I wanted to find which weak aspects had a rasterisation pipeline and how ray-tracing would improve such spots. The main topics found here would be shadows, which takes a lot of verbose code and performance for a real-time application, soft shadows, refraction and reflection. These can be implemented physically accurate tracing rays, whereas some hacks are needed for a rasterisation approach with a similar result.

Once those subjects were found, it was time to study how a pipeline would take advantage of both methods and communicate between them. This includes getting familiar with a low-level API such as Vulkan and implementing a rendering engine with the necessary extensions and capabilities, and understanding the process of ray tracing a scene to find ways to reduce such computational cost.

Finally find the spots where rays needed to be traced, reducing the number of rays sampled, thus improving the performance.

1.5 Thesis Structure

The memory of this project has been structured in several sections. First of all, the introduction has exposed the aim and motivations of this project with the background and methodology that has been carried out during the process.

Following you can find the state of the art section to present the latest state of the rasterisation and ray tracing world and a basic introduction to their use.

Since Vulkan is the API used in the project, I thought it important to dedicate a whole section to its functioning and capabilities with a brief explanation of its initialisation and structure of the code.

The proposed solution section is the backbone of the thesis and presents the solution carried out. In it, you will find a detailed explanation of how and why the decisions were made and a detailed structure of the pipeline.

Finally, a results section will scrutinise the results taken by a profiling application.

The future work section will expose ways to improve the pipeline in the future and new possible techniques.

2 State of the Art

The objective of real-time rendering is to obtain images from a computer fast enough to produce the illusion of reality. This illusion makes the user interact or respond depending on the images presented on the screen, believing there is a dynamic process rather than a still image. The rate at which images are presented is commonly known as frames per second (FPS) or Hertz (Hz). In order to maintain this illusion of a dynamic process, a minimum of 30 fps or higher is required. If an application runs at less than this minimum, the user realises the images being produced and loses the feeling of continuity. However, when the rate is high enough, the user focuses more on action and reaction in a fluid way.

A television usually presents each image at a rate of 24 frames per second, but it may display the same frame more than once to avoid flickering on the screen. Then, the display rate is different from the refresh rate. For a non-interactive application, a rate of 24 frames per second may be acceptable since the human eye can perceive motion at this rate. For less frequency, the eye would process the images rather individually than in motion. That may be okay for a movie, but there is more than just the display rate when the focus is an interactive application. Watching a sequence of images at 24 fps may be acceptable for sensing motion, but it may not be enough when response time plays a huge role in the application.

Interactivity is not only the main criterion when building a real-time rendering engine; it has to build three-dimensional images. Otherwise, any fast response application drawing anything on the screen would be acceptable. There are multiple algorithms to represent a 3D scene in an image. The two most important and the ones we will focus in this project are rasterisation and ray tracing.

Real-time state of the art techniques may have a hard time trying to simulate effects caused by light interacting with different surfaces or materials (reflection and refraction) or creating accurate physical behaviours (soft shadows and global illumination). More accurate techniques are used when creating a still image or single frames for a video with no interaction. Due to their computational cost, those techniques are reserved for non-interactive products. One of these techniques would be ray tracing, which works by simulating the path light takes in reverse order until it intersects with the scene, resulting in a more realistic image.

2.1 Rasterised Rendering

Rasterisation is an algorithm that produces 2D images given a 3D scene by using the graphics rendering pipeline or simply known as the pipeline. This algorithm is capable of computing a 2D image given a camera, three-dimensional objects, light sources and more by calculating a series of transformations to the vertices of the polygons and filling the area of the resulting polygons that fall inside the camera cone. In the figure 15, we can see the sequence of operations that the vertices take in the pipeline until they are saved in the output buffer.

Vertex positions that we want to visualize are transformed by 4x4 matrices and are expected to be in normalised device coordinates (NDC). Each x, y and z is between -1.0 and 1.0; coordinates outside this range will not be visible. Transforming coordinates to NDC is accomplished by a series

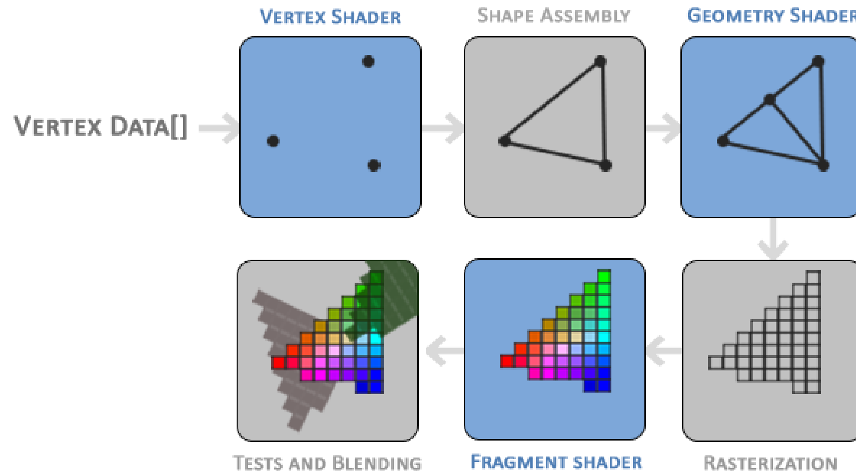

Figure 1: rasterisation pipeline[1]

of transformations, including the camera perspective, which will finally dictate if the coordinate falls inside the visibility range.[2] These transformations are computed in the vertex shader, which carries such computations for each input vertex. The rasteriser pass will then fill the polygon, and finally a visibility test is carried out just in case multiple polygons are superposed. Each pixel stores a depth value, and when another pixel superposes, both depth values are compared, and the nearest one is the final output. Usually, such polygons are triangles since they are the simplest geometry form and the most cost-effective one.[5]

The Rasterisation process is fast and its cost is linear. This method allows us to compute millions of triangles per frame and create images from complex meshes quickly. The efficiency in computing millions of triangles quickly is why rasterisation has become the predominant algorithm for real-time applications. Nevertheless, this is not an accurate way to compute real environments. Some effects like shadows, transparency and reflections need complex, verbose and sometimes high computational cost with a not very realistic result.

2.2 G-Buffer

The industry's requirements pushed forward the need to improve the performance of rendering 3D scenes, and new techniques appeared. One of the most famous, still used nowadays after three decades since its first appearance, is taking advantage of the information stored in texture from a previous geometry pass. Those textures are called geometry buffers or G-Buffers. Essentially those buffers are a 2D array of geometric properties such as position, normal vectors and albedo colour. Then in a second shading pass, this information is read by pixel and used to compute the shading of multiple lights. G-Buffers are widely used in the Deferred Shading technique, which separates the geometry rendering from the light calculations to compute more lights in the scene. Previous to this technique, forward rendering was used. This approach had to draw every model for each light to compute its shading correctly, resulting in a higher demand of resources when multiple lights

were defined in the scene.

Nonetheless, deferred shading had its drawbacks. This technique allows for shading a scene with far more lights. However, it requires storing all necessary geometry information in textures inside GPUs memory, which can have size and bandwidth limitations depending how powerful the GPU is. This can affect the number of textures that can be written to. Mobile applications, for example, cannot take advantage of this approach. Usually, four textures are the established limit, even though more have been used in our project. Multiple materials in a scene also mean more data to be encoded in the G-Buffers, and as just mentioned, it may be a problem due to the memory cost. Also, when using geometry buffers, it is difficult to handle transparency as one pixel can only store one value for position, normal vectors and colours. Sorting objects by distance when rendering may be a solution, but we will take advantage of rays in our project to solve transparencies.

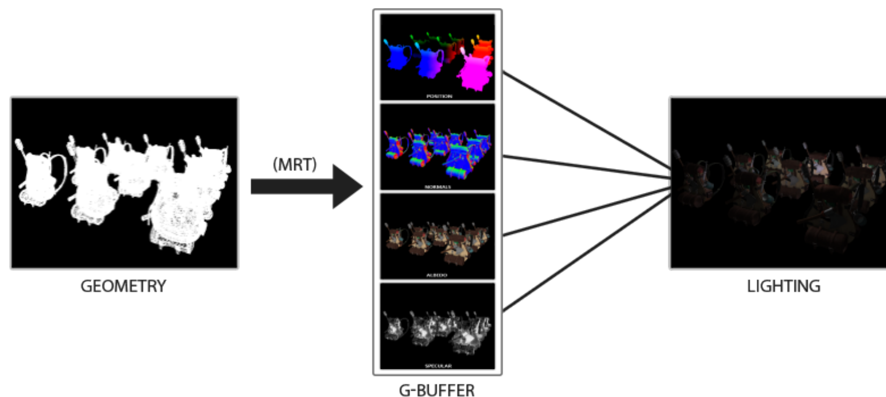

Figure 2: Deferred shading using G-Buffers.[2]

2.3 Ray tracing

On the other hand, we have the ray tracing algorithm with the same objective: render 3D scenes with a result much closer to reality. The algorithm is based on how light behaves in the world to compute how photons react to materials reflecting and refracting, thus delivering a photorealistic result. Photons travel through the air and collide with objects in the scene. Eventually, some of those photons end up in the camera which captures them. In order to only take into account those rays that make to the viewer's eye, we need to trace rays backwards, starting at the camera. Those first rays fired from the viewer's eye are known as primary rays. Those rays eventually will intersect with polygons in the scene, calculate the lighting value at that point and return it to the camera, defining the geometry directly visible. In reality, photons keep on bouncing for all surfaces until they lose all energy. To emulate that effect secondary rays are shoot recursively from the points intersected by the primary and subsequent secondary rays. Again those rays may collide with the scene, compute the lightning value and return to the camera. A mix with the colour at each bounce is computed as the final contribution for that specific pixel.

This process will produce far more photorealistic results than a rasterised pipeline. However, its computational cost makes it impractical for dynamic scenes. That is the main reason why this

algorithm is used in projects such as films. Each frame can be precomputed without worrying too much about the time it takes to render. How rays are cast can determine the cost of computing an image. The number of rays may determine the resolution, and its angle determines the perspective of the camera. Also, the materials where a ray intersects may determine the number of secondary rays or even if no ray is needed to be traced. Refractive or reflective surfaces may scatter rays in multiple directions, increasing the number of rays in the scene. In contrast, an opaque surface may need only the primary ray and the shadow ray to be defined. However, that would leave behind effects such as ambient occlusion and global illumination.

2.3.1 Ambient Occlusion and Global illumination

Ambient occlusion refers to the darkening of parts of the scene that partially have geometry close enough to occlude incoming light. In reality, any illuminated point is returning the effect light coming from all directions around specific position. The hemisphere facing the normal vector in a point is the area through light incises in such a point. Due to the morphology of the surroundings, more or less light can enter through that area. Edges, holes and surfaces can occlude part of the incoming light darkening that pixel. Solving the integral for the visibility in that hemisphere is impossible. Thus an arbitrary number of points are sampled in the area and rays are launched in those directions. The number of rays occluded over the total of rays shot gives the occlusion factor used to darken the colour giving more depth to the scene. For the whole scene, this effect may be included as a post-processed effect which may be computed in real time, but for some complex models this information may be stored in a texture.

Global illumination is based on the same principle that all incoming light from the hemisphere determines the colour of one pixel. Light contribution does not only include the emission from light sources, but from all the surroundings. Photons bounce, losing energy in every collision, but until all energy is depleted, photons still contribute to the colouring of the surfaces they collide. Then all objects contribute to their surroundings even though they do not emit light. This effect is known as global illumination in computer graphics and can be achieved by throwing rays in random directions through the hemisphere, compute the colour of the surrounding elements and add them to the pixel colour.

Both techniques can also be implemented in a rasterised pipeline, but due to its cost, the information of the scene is precomputed and stored in textures. This is also known as baked ambient occlusion and baked global illumination. For static scenes where distant lights affect massive static objects, shadows may be baked too. Static objects, as long as light does not change position, will always cast the same shadow, so there is no need to compute it every frame.

Eventually, new ways to improve ray tracing cost appeared. Scenes are organised and stored in structures called acceleration structures, representing the scene in a kind of tree graph manner. This way, a ray does not have to compute intersections for the whole scene reducing significantly the number of operations required. For real-time applications, every API have its acceleration structure creation process. Those structures are opaque to the developer and are an internal thing of each API.

2.4 Shading

State of the art shading techniques include Physically Based Rendering, which aims to render images to resemble how light behaves when interacting with surfaces and thus represent a more physically accurate look.

2.4.1 Microfacets

PBR techniques are based on a model called the Microfacet model. Such a model describes all surfaces as a group of tiny micro facets at a microscopic level. Those microfacets reflect light when photons hit them, and their positioning will describe how light reflect on an overall scale. The reader should imagine that microfacets are so small that they cannot be represented as a per-pixel basis. Thus a statistical approximation of a group of microfacets gives us the roughness factor of each pixel. Microfacets as a whole dictate if a surface is smooth or rough. If they are aligned in a very similar angle, their reflections will be sharper and well defined. In contrast, if their alignment is somewhat chaotic, representing a very rough surface, rays will scatter in different directions resulting in a much more wide specular reflection.

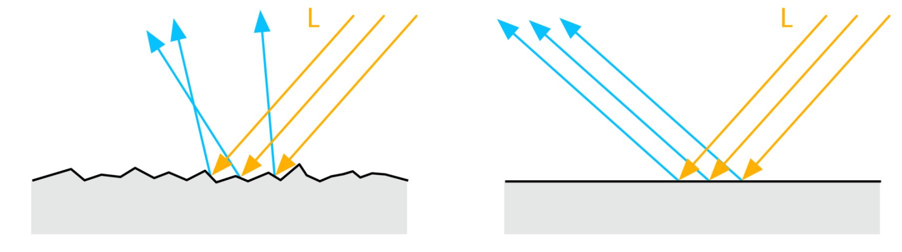

Figure 3: Microfacets at a microscopic level for rough and smooth surfaces.[3]

The roughness value ranges from 0 to 1 representing the average distribution of each microfacet's normal. A 0 distribution value means that the surface is perfectly smooth and all rays bounce in the same direction, whereas a value of 1 means the surface is so irregular that every normal vector will point into different directions.

2.4.2 Energy conservation principle

PBR also follows the energy conservation principle, which dictates that no outgoing light should carry more energy than incoming one, with the exception of emissive surfaces. This is because arrival light will hit a surface, and part of that energy will get absorbed, whereas the other part will bounce off. The part of the ray light being bounced off is the **reflective** part, also known as specular lighting. The part being **refracted** is absorbed by the object and is known as diffuse lighting. Note that they are mutually exclusive. Light being absorbed will not be reflected and vice-versa. This effect can be explained as all surfaces being made of tiny particles, and every material has a different particle scheme, making light react different depending on that. From physics, we know that a ray beam travels indefinitely until it loses all its energy and the way energy is lost is by colliding with particles. When a ray hits a surface, part of the ray hits the top particles, loses

part of its energy, and bounces off, creating what we perceive as the specular part. The rest of the beam enters the surface until it collides with particles deeper in the material. Then the ray keeps bouncing until it loses all energy. Some of those rays bounce until they hit the surface again, and that creates what we perceive as diffuse lightning.

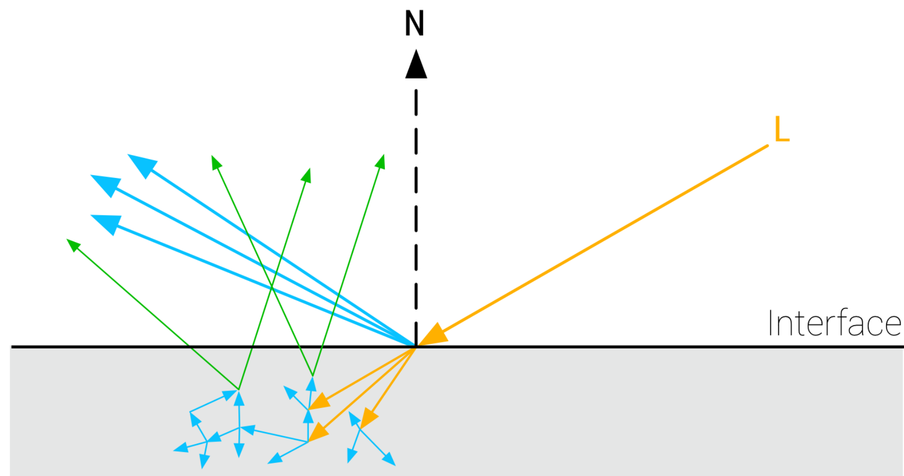

Figure 4: Light beam being scattered when hitting a surface.[3]

We took the mentioned techniques for shading for our project, but we treated light as only hitting the topmost part of the surface, and no subsurface scattering techniques were implemented.

2.4.3 Subsurface scattering

Subsurface scattering techniques try to emulate when light penetrates deep into a surface, gets scattered and re-surfaces into another point. A good example would be when potent direct light hits skin, flesh or any other translucent material.

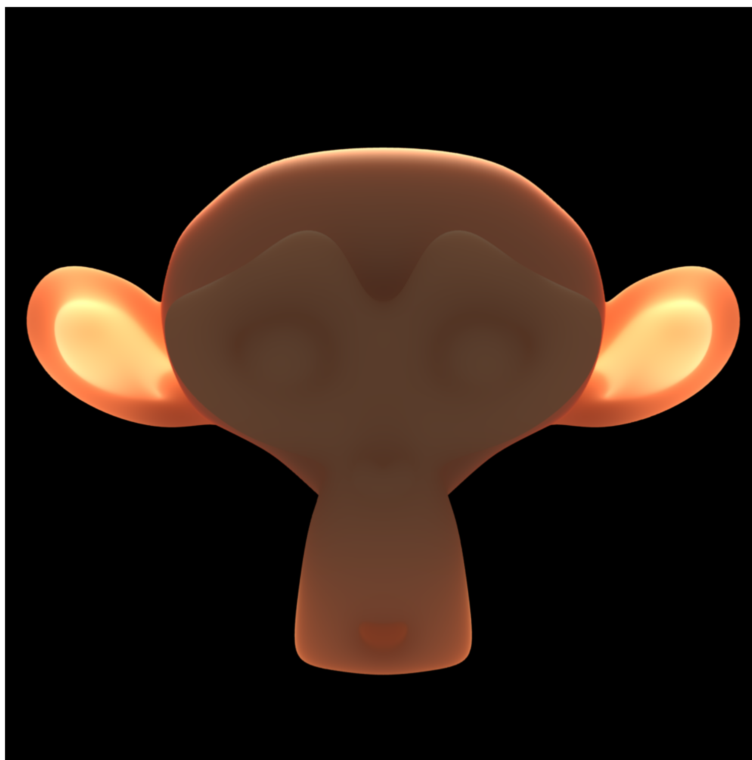

Figure 5: Light gets scattered when entering, penetrating a surface and exits at a different point.[4]

2.4.4 The render equation

All above can be represented mathematically in an equation known as the *render equation* shown in figure 6. This equation is strongly based on the principle of conservation of energy. Simply put, this equation calculates the light out at a given point p in the direction of the viewer w_o. This light is the sum of the emitted light L_e by the object towards the viewer direction surface plus how much light is being reflected in that same direction. The first part of the equation is easy to calculate. We can simply give a glowing value to the material, and we will know how much light it emits but the second part is where things get a little more tricky.

$$L_o(p, w_o) = L_e(p, w_o) + \int_{\Omega} f_r(p, w_i, w_o) L_i(p, w_i) n \cdot w_i dw_i$$

Figure 6: Render equation.

- L_o light directed to the viewer along w_o from point p.
- w_o view direction.
- L_e light emitted by the surface.
- n the normal vector at position p.

- p the position at which light is being calculated.
- Ω hemisphere over position p facing direction n that contemplates all possible w_i.
- $\int_{\Omega}$ integral over hemisphere Ω.
- w_i the negative direction of the incoming light.
- L_i incoming light along direction w_i.
- $f_r(p, w_i, w_o)$ is the proportion of incoming light along w_i reflected at position p towards viewer direction w_o given by the **Bidirectional reflectance distribution function**.

Essentially, what the integral represents is the addition of how much light is coming from all directions of the hemisphere given a specific point. All light contribution in the hemisphere is also known as irradiance. Irradiance is then multiplied by the BRDF function which dictates the amount of light being reflected and by **Lambert's cosine law** that reduces the amount of light reflected depending on how wide the angle is. The wider it gets, the weaker light bounces. Knowing all light incoming to the hemisphere area, we could find the precise result by solving the integral analytically, but there is no available solution.

One approach is to treat the radiance or incoming light for the available lights in the scene and compute only for those. The radiance is treated over an infinitely small point in the hemisphere instead of a solid angle, and our calculations are then done over a single ray per light. The integral is then transformed into a summation for all accountable lights. The **Bidirectional reflective distribution function** takes the normal vector, the light direction, the viewer direction and the roughness factor given by the material. It returns an approximate value of how much light from the incident ray bounces off. For it to be as physically correct, it has to respect the energy conservation principle. Multiple BRDF algorithms respect that principle, but the most used and the one implemented in this project is the **Cook-Torrance BRDF**.

3 Why Vulkan?

One of the first significant decisions in this project was to decide which API to use. There were two clear options among all APIs: DirectX12 and Vulkan (Metal is not included here since the project is done in a Windows environment). They both are low-level API with full ray tracing support. We finally decided to go along with Vulkan mainly due to its cross-platform capability.[8]

Vulkan is a cutting edge programming interface for graphics and compute devices developed by Khronos that aims to provide high-efficency, cross-platform access to modern GPUs while offering potential higher-performance and more balanced CPU and GPU usage due to its low-level capacity compared to its predecessors OpenGL and Direct3D. Vulkan also stands out for being an explicit API, meaning that almost everything is the programmer's responsibility and must be defined more verbosely. On the other hand, it is giving him also much more control in return. Every aspect of the graphics pipeline and its commands must be set up from scratch by the application, including memory management for items like buffers and textures. Its initialisation process may be considered very dull and tedious because of the amount of code necessary to show anything on the screen. It may be somewhat fragile at the beginning, there is an awful lot of work to get Vulkan running, and incorrect usage may result in bugs or driver crashes.

3.1 Initialisation

For time optimisation purposes, the *vk-bootstrap* [9] library was used. This utility library helped us simplifying all Vulkan initialisation process. Vulkan initialisation may become a very tedious and daunting task due to the amount of code involved. Because it is a very explicit API, we have to initialise it to select the desired GPU or multiple GPUs, load extensions and create the `VkInstance` and `VkDevice`. The latter structure is necessary for recording commands. In this process, a library like *vk-bootstrap* saves us a lot of code and work. Unlike OpenGL, Vulkan has no global state, and both `VkInstance` and `VkDevice` must be passed to every API function call.

The `VkInstance` structure is a representation of the API context. It is vital in its creation to enable the *validation layers* if necessary. Validation layers play a huge role when developing in Vulkan. Their purpose is to report any issues with the API calls and thus is a potent tool when debugging our program and reduces significantly the time spent solving errors. `VkInstance` creation also allows us to load extensions that may be necessary for the program.

Once `VkInstance` has been created, we query for its available GPUs in the system. From that list, we can find the properties and capabilities of each one and choose the most suitable for our purposes. Vulkan even allows us to use more than one GPU, but that will not be the case in our project. The `VkPhysicalDevice` handle represents each of the available GPUs. This handle may be helpful in complex applications since it allows us to query its features or available extensions. Our project required a few of those extensions that would allow us to enable tracing rays in our program.

After having chosen our GPU, we are allowed to create a `VkDevice` from it. This structure represents the driver in the GPU and a way to communicate with it, which is why it is needed for most API calls. The creation process for our logical device handle is very similar to the context instance one. We enable necessary extensions and proceed to create the handle. Bear in mind that it is important not to load unnecessary extensions as they can slow the driver and affect

performance. Khronos released the Vulkan Ray Tracing extensions **VK_ray_tracing_pipeline** in November 2020, becoming the first open standard for ray tracing acceleration in GPUs in the industry. Before such date, the extension was in a beta state and extensions from third-party vendors such as **VK_NV_ray_tracing** from NVIDIA were the only ones available. The API is in constant revision, and new extensions and features are being released periodically. These extensions allows the programmer to build and manage acceleration structures, which plays a key role when tracing rays, support for ray tracing shader stages and ray tracing pipelines. This extensions are device specific and thus must be enabled in the logical device creation process.

The process of initialising the context for Vulkan may seem tedious and takes much code to set it up since most of the work has to be done manually, but this has its advantages. In case multiple GPUs are available in a system, we would create a logical device instance for each physical device and use them in parallel for different purposes. We could use the main GPU for graphics purposes, whereas another GPU could be used to perform physical or complex calculations in parallel and finally share their information upgrading the program's performance.

Since this is a graphical project and though instancing a device may be interesting, we need it to perform some rendering. For the application to render an image, a *swapchain* must be created. Luckily for us, *vk-bootstrap* helps us build a swapchain too. The swapchain is created with a given size, and recreation is needed if the window is resized to different dimensions. Essentially a swapchain consists of a list of images that are accessible for being displayed. The ideal list of images for a standard graphical application is between two and three in order to perform a double or triple buffer rendering. Those images are created automatically in the swapchain creation process. Every frame, present commands will have to be recorded and receive the swapchain image that will be presented for it to be inflated with the latest data.

3.2 Executing Commands

In order to render images to the screen, a sequence of commands has to be executed. In Vulkan, commands have to be allocated to a *Command buffer* from a *Command pool* and executed on queues. Usually, the program allocates a `VkCommandBuffer` from a `VkCommandPool` and commands are then recorded into the command buffer. Once all commands have been recorded, we end the recording process by calling `vkEndCommandBuffer`, and the buffer is ready to be submitted to the queue. The recording process prerecords commands and executes them every frame without having to re-record them even though recording commands takes almost no penalty. In the project we combined both approaches depending on the pass. In the rasterisation pass, where we take advantage of push constants, we recorded commands every frame because the data passed through the push constants have to be updated in every frame. The last pass commands are also recorded in each frame since the swapchain images have to be passed for presentation. The rest of the commands have been recorded in advance, and they are only executed when submitted. [10] [11] [12]

4 Proposed Solution

In order to create a dynamic scene that obtains the quality of a ray tracing algorithm and takes advantage of the performance of a rasterised renderer, the concept of hybrid techniques started to surge.[13] There are still many issues to be solved though. To obtain certain effects a huge number of rays and iterations is needed, thus work in finding ways to reduce the number of rays shot is still in progress. Also, it is very difficult for companies to put resources into creating or improving certain techniques that only a few users would be able to run and enjoy. This project aims to create a solution that implements a hybrid pipeline taking advantage of the information from the G-Buffers in the rasterised pass and computing certain effects in a subsequent ray-traced pass.

4.1 The pipeline

In order to mix both algorithms and maintain a certain level of performance is imperative to analyse how the ray tracing pipeline works since tracing rays will take most of our computing cost. Once the ray tracing algorithm is understood, we proceeded to find ways to optimise it. The ray-tracing algorithm can be divided into two main parts: primary rays and secondary rays. The main purposes of primary rays are as following:

- To determine camera properties. The position, FOV and resolution can be defined by primary rays.
- To find the intersection points in the scene, defining the geometry information of the 3D world.
- To compute the lightning and shading at the intersected point and return its value.
- To define the properties of the secondary rays, such as if a secondary ray is needed, its origin and direction.

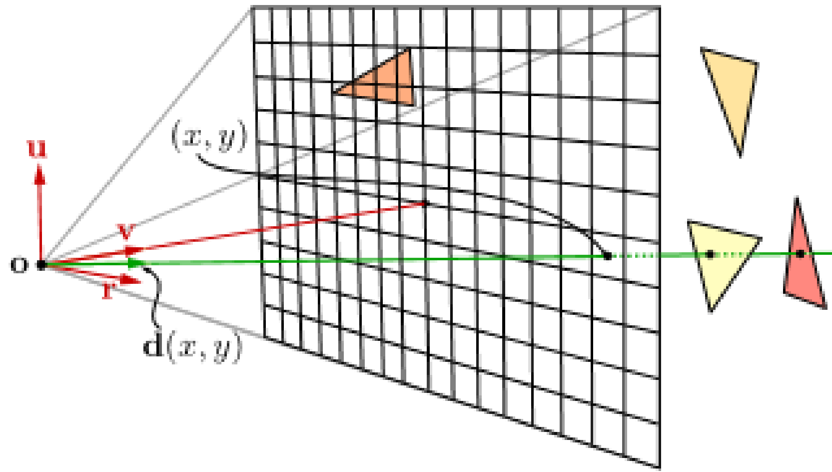

Figure 7: Primary rays are shot from camera position into the scene for each pixel in the screen. Then, each ray computes intersection calculations with geometry. The return [5]

From all four critical points mentioned above, we can easily see that information from the camera, geometry definition and shading can be obtained from a previous rasterisation pass and save such information in the geometry buffers. This way, we can avoid tracing primary rays saving some computational cost. The information for the following secondary rays can be obtained from simple calculations in a subsequent ray tracing pass given the information in the G-Buffers. Thus our pipeline consists of five independent passes. Each one will benefit from the information from the previous pass.

1. **Rasterisation pass:** define geometry and store information in the G-Buffers.
2. **Shadow pass:** trace rays to determine if the position is occluded or not.
3. **Compute pass:** perform a Spatio-temporal accumulation to denoise result of the shadow pass.
4. **Shading pass:** shade the scene and launch secondary rays when needed (refraction and reflection).
5. **Postpo pass:** perform a degamma to the final image.

First, a rasterisation rendering pass will gather the geometry information and material colours from the scene. This step outputs the following information in the G-Buffers: position, normal vectors, albedo colour, motion vectors, material and emissive texture. We will expose more about the geometry buffers in the next section.

Once we stored the information in textures from the geometry pass there will be a ray tracing pass to calculate shadows in the scene. In this pass, rays are traced from the position read in the position G-Buffer and into the lights. We will store a texture with the occlusion information for

each pixel for each light in the scene. This information is then accumulated with a spatio-temporal accumulation algorithm in a compute pass before being ejected to the shading pass.

The fourth pass will have as input all the needed information to calculate the secondary rays and lighting in the scene. With the information from the material texture, we know which pixel belongs to a reflective or refractive surface and then shot rays using the normal and position data only for those pixels.

A final pass is done to perform gamma correction to the output image. The scheme of the whole pipeline is represented in figure 8.

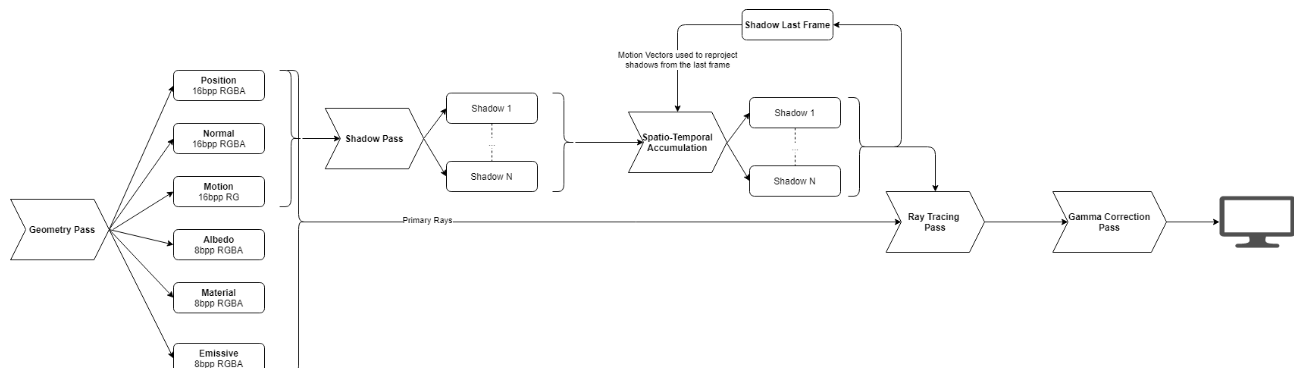

Figure 8: Pipeline solution scheme.

4.2 Rasterisation rendering

Deferred shading plays a massive role in this solution. This algorithm was first presented about three decades ago, and it has been widely used in the industry for its ability to render multiple lights at a low cost. Instead of rendering the scene per each light, as in forward rendering, we divide the pipeline in two: first the geometry pass, where all the geometry will be rasterised and its properties will be stored in the G-Buffers. A final pass then will use the defined geometry information in the G-Buffers to compute the lighting. In this second pass, we only have to calculate the shading for each pixel, taking into account the impact of all lights but avoiding the need to rasterise the whole scene per light again. Our hybrid algorithm is based on this same idea of reusing the information from a previous geometry pass to avoid shooting primary rays and tracing only the needed secondary rays. Due to its low computational cost compared to tracing rays, it was decided to rasterise as much as possible.

For subsequent calculations, we needed the following information to be stored in the rasterisation pass:

- **Position information:** it is vital to store the world position for each pixel in order to compute the lighting and ray-tracing calculations. Each pixel holds the position representation of the scene.
- **Normal vector information:** normal vectors are essential when computing the lighting and direction of secondary rays. Each pixel stores the normal vector, the unitary vector

perpendicular to any surface, of the belonging geometry.

- **Albedo colour:** this texture stores the colour of the geometry. It can be read from texture, if any or from the vertex information of the mesh.
- **Motion vectors information:** these vectors are used to compute the distance and direction the camera has moved from the previous frame. This vector is calculated using the camera position and projection of the previous frame if the camera or scene has changed. It is useful when accumulating information from old frames.
- **Material information:** this texture store the material of each pixel. Materials are not only used for shading but for computing a reflected or refracted ray if necessary.
- **Emissive textures:** will give information about which pixels are emissive and add to the final colour.

This information is essential for the computations in the subsequent passes. Not only it is faster, but it helps us avoid shooting primary rays.

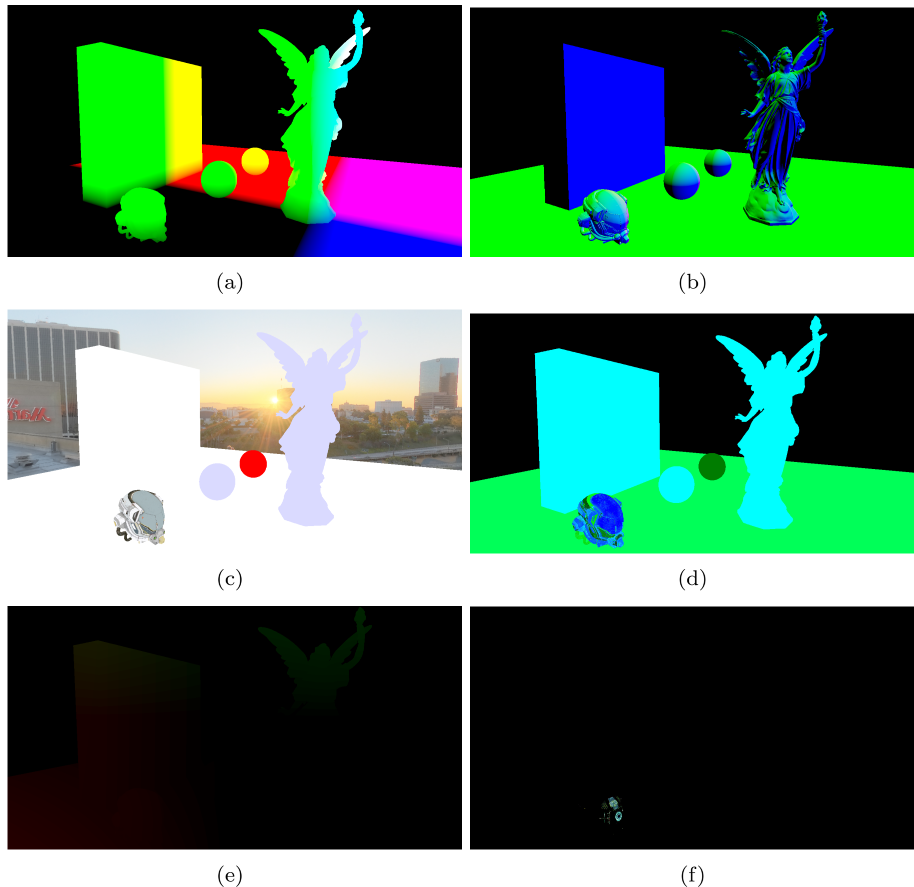

Figure 9: (a) Position (b) Normal Vectors (c) Albedo (d) Material (e) Motion Vectors (f) Emissive texture

In Vulkan, the main way to inflate the GPU with data is through **Descriptor sets**. Descriptor sets are a group of handles that determines the information and location passed to the graphics unit. It is formed by one or multiple descriptors that hold the pointer to the memory data, the size of such buffer or the sampler for an image if it is a texture we are uploading. Vulkan allows only one resource bound per set, but that is very inefficient, and some hardware does not allow it. In fact, some hardware is limited to a four descriptor sets, and thus this is the leading standard.

There is an advantage when grouping bindings per set, and this is that we can decide when to bound each set. For instance, a descriptor set that uploads global variables may be bound only once per set, whereas another descriptor set holding per mesh information may be bound as many times as meshes in the scene per frame.

As with most Vulkan structures, descriptors sets have to be allocated from a descriptor pool. This process will allocate the descriptor set to a specific memory section of the VRAM. After allocating the descriptor, the user has to write it to make it point to the data that he wants to be

uploaded and bind it. Once bound to the pipeline, the descriptor can be used in GPU when calling the rendering commands, but data cannot be modified unless a flag that allows doing so has been added to the pool.

For the first pass, not much data is passed through descriptors. The two only necessary things passed through descriptors are camera matrices and a texture array. The rest of the information, such as model matrix, material properties and textures indices, are passed through push constants. Push constants is a Vulkan way to upload a small chunk of data to VRAM very quickly and with no cost. The main drawback is that push constants data is limited to 128 bytes.

4.3 Data organisation

How data is injected into VRAM may become a complex topic when referring to a hybrid pipeline. The main issue with a hybrid pipeline is that a different data format is needed for a rasterisation and ray tracing pass. Having both algorithms in the same pass ends up with redundant data being passed to the GPU, but it cannot be avoided due to the nature of both approaches.

In the rasterisation pass, we know in advance which mesh is being computed at each moment by the GPU, and we pass only the necessary data for that specific instance. The advantage is that only the necessary data is passed to VRAM, occupying less than what the whole scenery data would, but we have to bind it to GPU for each object. Thus we end up binding data multiple times per frame. The data passed for each object must contain a model matrix to draw the instance correctly and material information. That material information also includes an integer that will describe the type of the surface. Thus in the G-Buffer will be encoded such integer indicating if the pixel corresponds to a surface that has to be ray traced afterwards (reflective or refractive material) or not.

On the other hand, when tracing rays, we have no clue where the ray will collide or if it is even going to collide, and a more proactive approach is mandatory. Also, it may be possible that a secondary ray hits a non-visible point for primary visibility, and such information will not be in the G-Buffers. Then all essential information is uploaded just once beforehand.

Our objects in the engine have a hierarchy node structure that allows to load both .obj and .gltf models [14]. The **Prefab** class will include the topmost node of the hierarchy and a **Mesh** that holds all vertices and indices and its pointer to the buffers in VRAM. The parent node may hold a vector of child nodes and at the same time those nodes can have multiple children and so on. Each node also has a **Primitive** structure containing indices and information useful for building the Bottom Level Acceleration Structure (BLAS). Those indices also give information about which specific chunk of vertex and buffer vector hold by the Mesh corresponds to it. That is not only useful when building the bottom level acceleration structure but also to indicate in the shader where to find in the buffers the geometry data when being hit by a ray.

All data for all possible objects in the scene must be uploaded and once the ray collides unpack certain data to make use of them. This information also includes vertex and indices data, which is necessary because once the ray collides the only information it gives back about the geometry is the primitive ID. Other values such as Instance ID, which points to each of the Top Level Acceleration Structure (TLAS) instances, can be defined by the user when building the acceleration structures.

This custom variable added when building the TLAS instances is vital as will indicate the

position in the array storing important IDs. Those IDs will then be used to unpack the material, matrix, vertices and indices referent to the triangle intersected. How data per triangles is unpacked after a collision is found in a ray-tracing pass is shown in figure 10.

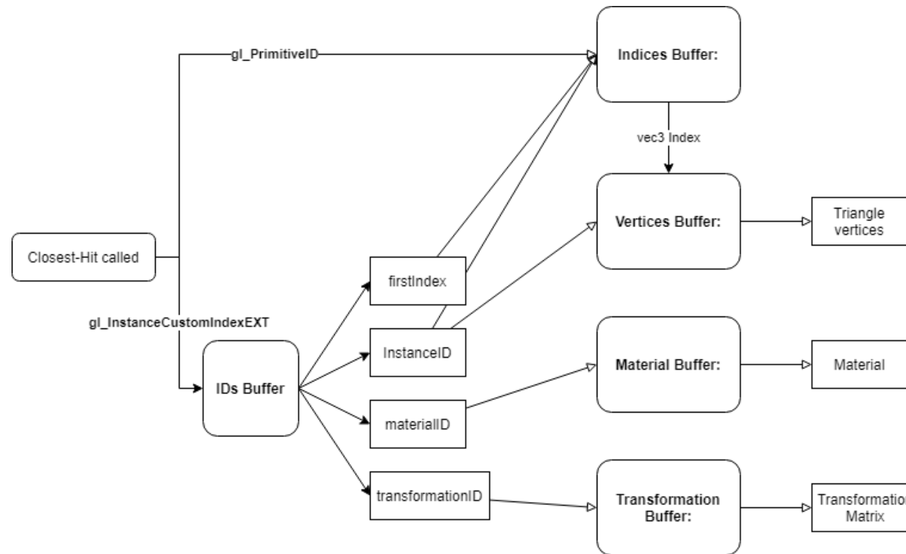

Figure 10: Unpacking vertex data in a ray-tracing pass.

Unpacking the data of the triangle intersected allows us to compute the normal direction and find the corresponding material to proceed to shade such pixel or trace subsequent rays. To successfully unpack the information we also take advantage of the variable `gl_PrimitiveID`. `gl_PrimitiveID` gives us the position or index of the triangle being hit. This allows us to find the position in the indices buffer to find the corresponding indices of the triangle.

4.4 Tracing Rays

Following the rasterisation pass we have two passes that trace rays. The first one will obtain information about shadows in the scene, tracing one ray per pixel per light and storing in a texture. This means we will have as many shadow textures as dynamic lights in the scene. The second pass will be used to trace all necessary secondary rays and compute the shading of the scene, taking into account all shadow information from the previous pass.

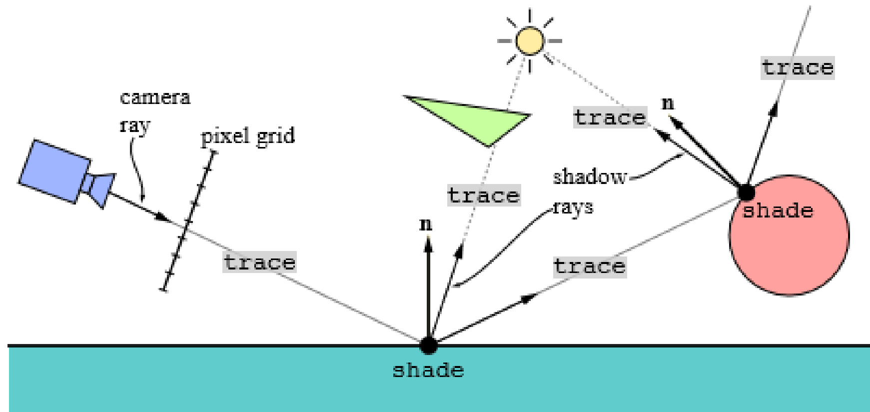

Figure 11: Primary rays are shot from camera position into the scene for each pixel in the screen. Then, each ray computes intersection calculations with geometry. If a ray intersects any geometry the subsequent rays can be traced to calculate shadows or illumination techniques.[5]

A generic pseudo-code of the ray tracing algorithm given a target position p would look as follows:

```
Data: Position p, Normal N, View vector V
Result: Pixel shaded
vec3 position = read(positionGbuffer, x, y);
vec3 normal = read(normalGbuffer, x, y);
vec3 colour = vec3(0);
foreach light in Lights do
    float shadowFactor = 0.0;
    if lightIsVisible then
        shadowFactor = computeShadow();
        if refractive OR reflective then
            vec3 dir = computeScatteredDirection();
            TraceRay(p, dir);
            colour += shadowFactor * (Shade() + payload.colour);
        else
            colour += shadowFactor * Shade();
        end
    end
end
colour += computeIrradiance();
store(output, colour);
return colour
```

Algorithm 1: Ray Tracing pseudo-code.

The algorithm 1 pseudo-code is a simple showcase of what a ray tracing algorithm should do.

There may be many different approaches with a similar result. This one reads the position and normal vector from the G-Buffers and iterates through all lights in the scene to compute the shadow factor and a posterior shading process. This process may involve launching more rays to calculate specific effects.

4.4.1 Acceleration Structures

To use ray tracing in Vulkan it is imperative to build the **Acceleration Structures**. This structures organise geometry data in an optimised hierarchically manner into hardware with the purpose of reducing the overall number of ray intersection tests. In Vulkan this organisation is opaque to the user except for a two-level structure: the top level acceleration structure (TLAS) and the bottom level acceleration structure (BLAS).

BLAS contain basic vertex information and can be built from one or multiple vertex buffer with its own transformation matrix. This matrix allows us to have multiple models in different positions inside the BLAS structure. Take into account that when multiple models with the same geometry are instantiated in a single BLAS, even though its geometry may be duplicated, performance can be improved in a static scene. Note that the lower the number of BLAS the better. The geometry information that a BLAS contains is generally triangles but it can also be a custom geometry. The first one contains information about a set of triangles whereas the latest is bound to an intersection shader that can implement a custom intersection test to a function defined previously.

TLAS on the other hand contain all object instances in the scene. Each instance has its own transformation matrix and points to a specific BLAS containing its geometry representation. This structure is illustrated in figure 12. Building either type of acceleration structure results in an opaque format in memory. BLAS are only referenced in TLAS and the latter is accessed from the shader as a descriptor binding.[12]. For more in depth information about how acceleration structure is defined can be found in the Vulkan specification documentation [15].

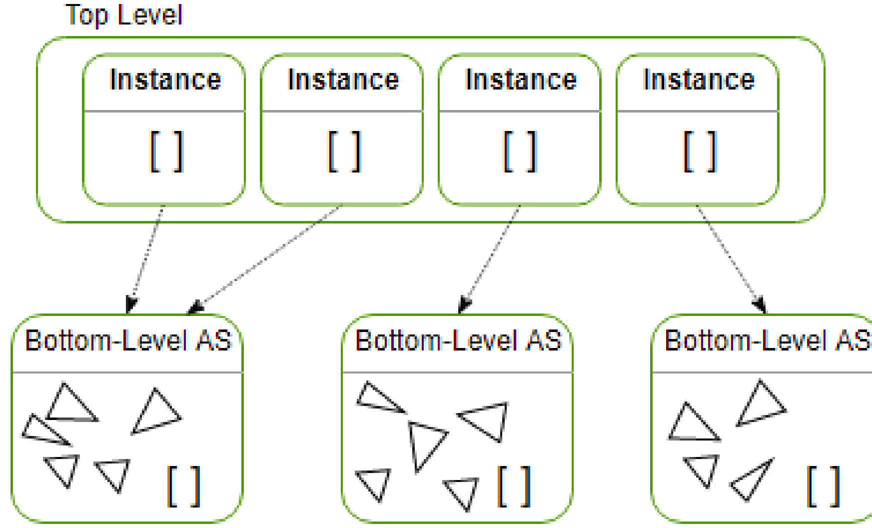

Figure 12: Acceleration Structure composition[6]

4.4.2 Ray tracing descriptors and shaders

In rasterisation pipelines we can know up-front which material is being rendered, thus executing a specific pipeline with a descriptor set defined with the necessary data. This way the user can prepare different pipeline for different materials. In ray tracing things get more complex given that we cannot know in advance which material the ray is intersecting. The descriptor set for the scene must hold all possible information of the scene. Information we found essential in the project would be an array of materials, an array of textures, geometry buffers that hold vertex and index information for the whole scene and an array of matrices. It is imperative to bind the acceleration structure as a descriptor binding if we want to use ray tracing capabilities.

Vulkan has up to five shader types each one representing different stages. In this project we made use of only three of them:

- **Ray generation shader:** essentially it is a compute shader that acts as an entry point to the shaders once traceRayEXT() is called. It has the ability to call traceRayExt() and shoot rays.
- **Closest-hit shader:** this shader is called once a ray shoot from the ray generation shader has intersected with geometry in the scene. Usually the shading process takes places here.
- **Miss shader:** this shader is called once a ray shoot from the ray generation shader has not intersected with any geometry in the scene. Usually background or environment is computed here.

The program has not the ability to predict which shaders will be called at any moment. A **Shader Binding Table** must then be defined[16]. This table is a structure that holds shader function handles and parameters for these functions. During the ray tracing execution different

shaders can be invoked through the table to execution depending on if a geometry was hit or not hit or by any other decision made by the programmer. As an example, if a ray would not intersect any geometry on the scene the table will then call the execution of the indicated miss shader. Multiple miss and hit shaders can be coded and invoked depending on certain parameters such as materials.

Ray tracing work can be called with the command `vkCmdTraceRaysKHR` with a ray tracing pipeline bound and it will initialize a ray tracing dispatch. This generates a series of threads with the width, height and depth specified that allows the ray generation shader to call `traceRaysEXT()`, a function that will instantiate a ray [17]. Such command will initiate traversal work targeting the provided acceleration structure. It is during the traversal process that intersection tests are performed against primitives in the leaf nodes of the acceleration structures. An intersection shader must be used in case intersections have to be tested to custom geometry. In our project only triangles are being used, thus such shader is not needed. Once the traversal is finished either a miss or hit shader is invoked.

```
  traceRayEXT(topLevelAS,  // acceleration structure
        rayFlags,          // rayFlags
        0xFF,              // cullMask
        0,                 // sbtRecordOffset
        0,                 // sbtRecordStride
        0,                 // missIndex
        origin.xyz,        // ray origin
        tMin,              // ray min range
        direction.xyz,     // ray direction
        tMax,              // ray max range
        0                  // payload (location = 0)
);
```

Figure 13: traceRaysEXT() definition.

Information may be necessary across different shaders and Vulkan allows us to communicate between shaders by using a payload. This memory block can be defined as an output or input to a shader and multiple payloads, each for different shaders, can be defined. In our case two payloads, one for the closest-hit and another for the miss shader, were defined. To avoid possible performance and recursive errors when tracing rays and to have more control, the decision to shoot all rays from the ray generation shader was made. Some solutions may allow to shoot from the closest-hit shader as well but it is not the case in our project. Thus, the payload is an essential part since it carries the necessary information to shoot rays recursively.

When a ray misses all geometry in the scene, the miss shader is executed. It computes the background color and returns its value in the payload. When a ray intersects, the closest-hit shader is executed and the shading computations are done. It may be the case that the geometry hit may be reflective or refractive and thus we need to return through the payload more information than just the color. The direction and origin of the new ray as well as the distance is also returned. With that information we can shoot a new ray in the ray generation shader as long as we have not

reached the depth limit. If that is the case, the colour is computed and returned as output. A logic scheme of the ray-tracing pipeline is illustrated in figure 14

In figure 13 the reader will find the definition of the traceRayEXT() and hopefully it will help to further understand its usage. As a first argument the top-level acceleration structure through the ray will traverse must be taken. Following the user can indicate the flags the ray will take. For instance, such flags can be used to indicate if we want to stop traversing once an opaque surface has been hit, if we want to terminate on first hit or if we want to skip the closest-hit shader. `sbtRecordOffset` and `sbtRecordStride` parameters control which shader is to be called from the shader binding table. In this project only one closest-hit shader and no any-hit shader are being used and thus such parameters will stay as 0. Following we find the `missIndex` parameter that indicates which miss shader is to be called if no intersection is found. The `origin, direction, tMin` and `tMax` are the parameters defining the ray. Finally the location of the payload is taken. This location will be used to store a memory that can be read from other shaders and used as a way to communicate between them.

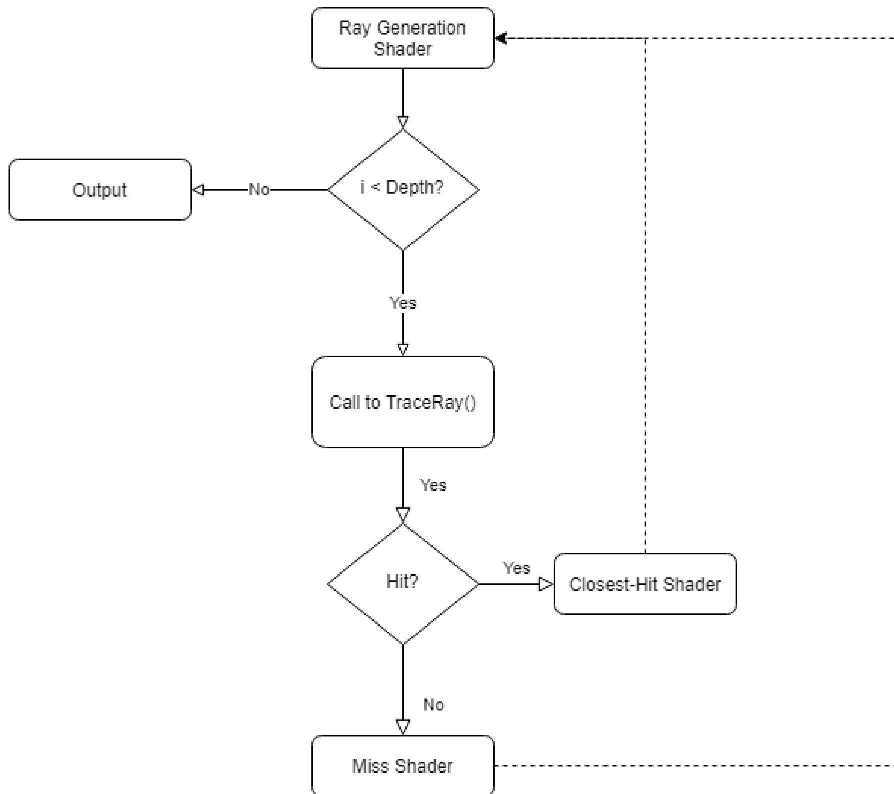

Figure 14: Ray tracing pipeline logic.

4.4.3 Shadows

One of the key points in the project was to ray trace shadows. Shadows are an important topic when rendering a scene since it gives us the depth and realism the image needs to feel natural. When trying to add shadows using rasterisation you can hardly do it in a single pass. Usually it requires to precompute the visibility of the scene from the light position and store it in a shadow map. Once it is time to perform the shading of the scene we can compare the distance of the pixel we are shading with the distance stored in the shadow map and conclude if it is occluded or not. Rasterized shadows carry a number of problems such as not being able to soften the shadows, the textures have to be very large to avoid jigsaw caused by low precision, the scene has to be rendered again for each light and sometimes shadows are not very precise due to limited pixel precision.

Given our usage of rays we decided that shadows in the project would be ray traced to avoid all the verbosity that rasterized shadows need (ray traced shadows can take as little as a few of lines of code) and for the precision advantage and ability to generate soft shadows that ray-tracing proportionate. The shadow process has been divided into two phases. The first pass will store the shadow information in a shadow texture per light and the subsequent pass will perform a spatio-temporal accumulation to generate the effect of sampling multiple rays per pixel in a compute shader.

Shadows could have been traced in the same pass as the secondary rays, avoiding the resources cost of having a texture per light storing shadow information, but we decided to do a previous pass because we wanted to compute a spatio-temporal accumulation in those textures to reduce the amount of noise. Since this could create the ugly effect know as ghosting, where data from previous frames can be clearly seen in the image, we decided not to perform it in the final output, only in the shadow textures.

The first pass has as descriptor bindings the acceleration structure to perform the ray traversals, an array of textures for each light where shadow information of the scene will be stored, an array containing the lights in the scene and three G-Buffers containing the position, normal vectors and motion vectors information. The main idea is to loop for each light in the scene and trace a ray from the position to the light. The ray will then return a boolean value with the occlusion information. If the value is true the pixel is in shadow and the colour stored will be 0, otherwise the pixel is illuminated and the value stored will be 1. This factor is used afterwards in the shading pass. Once pixel colour is calculated the result will be multiplied by the shadow factor read from the shadow texture. If the shadow factor is 0, meaning the pixel is in shadow, the colour will be multiplied by zero resulting in a black colour.

In this project no translucent material is taken into consideration and thus transparent material is treated as if they cannot cast shadow over themselves. To achieve so we first check the material type and if is a refractive surface then we store as if it was not occluded and the amount of shadow rays is decreased since none are shot for those pixels. The algorithm figure 3 shows the basic idea behind the whole shadow process.

```
Result: Write here the result
if isRefractionMaterial then
    for light in lights do
        store(shadowTexture(light), 1);
    end
else
    for light in lights do
        float shadowFactor = 0.0f;
        if isVisible  inRange then
            for samples in shadowSamples do
                traceRay(from p to light);
                if inShadow then
                    do nothing;
                else
                    shadowFactor++;
                end
            end
        else
            s
        end
        hadowFactor = shadowFactor / shadowSamples;
        store(shadowTexture(light), shadowFactor);
    end
end
```

Algorithm 3: Shadow calculation pseudocode

One advantage of using rays to compute shadows and one of the objectives of this project was to implement soft shadows. The common-sense dictates that a point is either visible or not visible by another point, but that approach results in hard shadows, where shadow edges are clearly defined. In reality there is no existence of such infinitely small light point and this is why hard shadows seems so unrealistic to the human eye. Lights tend to have volume, even the furthest shadow-creating light source in our planet, the sun, is not sufficiently far to be considered small enough to produce hard shadows. In a more realistic situation, there are points in the scene that will have a partial visibility of the light creating what is known as penumbra, areas where shadow slowly converges to light.

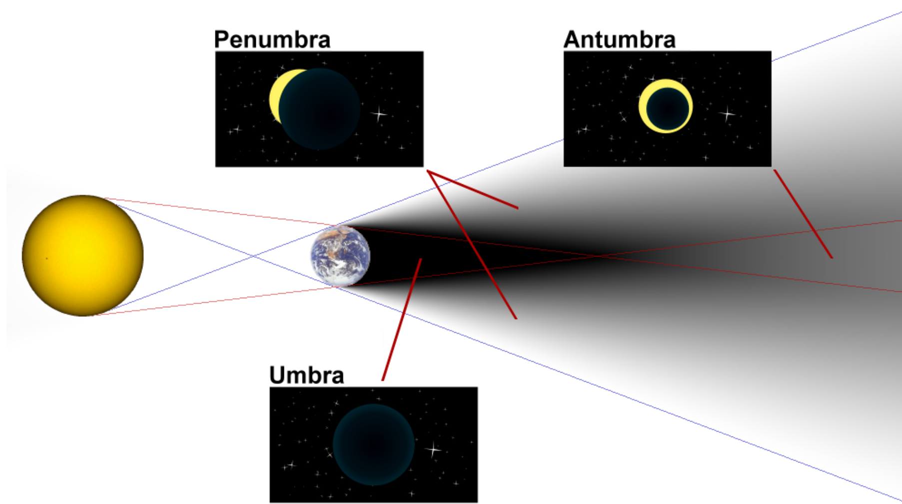

Figure 15: Soft shadow composition[7]

As seen in figure 6 the lightning in a point is given by the total amount of light entering its hemisphere. For optimization purposes, such area is reduced to only the dynamic area lights in the scene but we cannot compute an integral to calculate the amount of light arriving at that point. One solution to that problem would be to simply sample a huge amount of randomised rays in the area light. By increasing the number of rays the result would converge to an approximate solution but given our hardware constraints this is not feasible. Nevertheless we can take advantage of our real-time application. Given we are drawing n frames per second we can take samples from previous frames and compute an approximation result.

To achieve such result we have to sample randomly each frame for each light. The project only included point light spheres. Spheres from the point being shaded are viewed as disks. We proceeded to sample to a disk given the radius of the light and translate that point to L (vector towards the center of the light). The function used to sample the disk is the following:

```
1  vec3 sampleDisk(Light light, vec3 position, vec3 L, uint seed)
2  {
3      float radius = light.radius * sqrt(rnd(seed));
4      float angle = rnd(seed) * 2.0f * PI;
5      vec2 point = vec2(radius * cos(angle), radius * sin(angle));
6      vec3 tangent = normalize(cross(L, vec3(0, 1, 0)));
7      if(L == vec3(0, 1, 0))
8        tangent = vec3(0, 0, 1);
9      vec3 bitangent = normalize(cross(tangent, L));
10     vec3 target = position + L + point.x * tangent + point.y * bitangent;
11     return normalize(target - position);
12 }
```

Once the direction is returned we can shoot a ray. For optimisation purposes we will launch the ray with different flags than the rest of rays shoot in the application. In this case `traceRaysEXT()` will receive the following flags as parameters: `gl_RayFlagsOpaqueEXT, gl_RayFlagsTerminateOnFirstHitEXT and gl_RayFlagsSkipClosestHitShaderEXT`. The first value `gl_RayFlagsOpaqueEXT` is used in the other ray launches and will compute intersections with all geometry labed as opaque, which in our case is all geometry.

The second flag `gl_RayFlagsTerminateOnFirstHitEXT` will stop the execution when the first hit is found. This flag is the most important of all because we only want to find if any object was hit between the origin point and the light but we do not care if that hit is closest one. When computing the hits the program has no way to know if the collision being treated is the closes candidate or not. Vulkan stores the t_max of the closest hit until that moment and the t of the new candidate is compared to the one previously stored and discarded if $t > t_max$ or stored otherwise. As the Vulkan specification [15] points out:

> *'Unless the ray was traced with the TerminateOnFirstHitKHR ray flag, the implementation must track the closest confirmed hit until all geometries have been tested and either confirmed or dropped.*
>
> *After an intersection candidate is confirmed, its t value is compared to tmax to determine which intersection is closer, where t is the parametric distance along the ray at which the intersection occurred. [...] If TerminateOnFirstHitKHR was included in the Ray Flags used to trace the ray, once the first hit is confirmed, the ray trace is terminated.'*

The importance of the flag resides in the fact that may reduce a huge amount of computation since we only want to know if that ray intersected with a geometry, that is all.

Finally the flag `gl_RayFlagsSkipClosestHitShaderEXT` avoids the execution of a closest-hit shader. Once a hit is found we do not want to execute any code in that collision. The `isShadowed` bool is initialised as true and only if the miss shader is executed is then changed to false, meaning no geometry was hit and thus the pixel is not occlueded.

The project by default samples just one sample per pixel (1spp) but that results in a very noisy shadow if not accumulated. This can be seen at figure 19. One solution, supported by the application, would be to increase the number of samples per pixel but at a performance cost. Each ray will return its value and an average of all rays will be the final factor stored. Lets take we shoot ten rays per pixel. The shadow factor if seven out of ten rays are occluded would be 0.3. That factor would then be used to reduce the amount of light in that pixel for that specific light. But due to performance constraints we will work with 1spp with the noisy result. Is because of the noise that a second pass is needed to get rid of it.

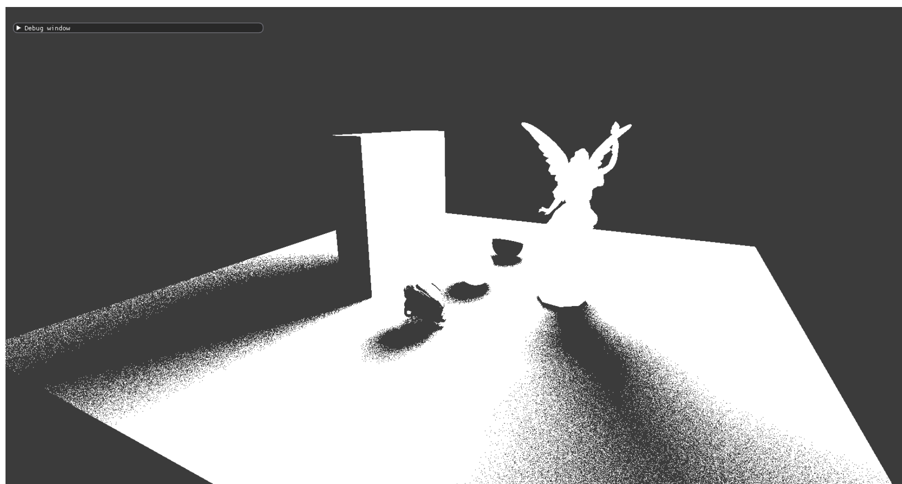

Figure 16: Shadows traced with 1spp without accumulation.

In the second phase we will not need to trace rays, instead a compute shader is used to perform a spatio-temporal accumulation to reduce the noise effect. As the name indicates, space and time will take part in this technique using noisy images from previous frames. This is a Monte Carlo integration and that is why randomised samples every frame are so important.

The more samples available the more approximate to a ideal solution the result will be. But we cannot save into textures indefinite previous samples as it would eat the whole VRAM in a couple of seconds. Only one previous texture storing the accumulation of all previous samples besides the new noisy image per light will be stored. In the shader we use the $mix()$ function, shown in figure 17, which basically does a linear interpolation between to given values and a factor $alpha$ [18].

$$alpha \cdot newShadow + (1 - alpha) \cdot previousShadow$$

Figure 17: Mix function.

Alpha essentially is a factor between 0 and 1 that will interpolate between both previous and actual value. The closer $alpha$ is to one the noisier, but less ghosting appears in the image. Our approach was to approximate an equal value for each frame and our alpha was progressively decreasing with every frame $alpha = 1.0/(1.0 + frameCount)$. When the value is one it will only take the new shadow whereas when the value decreases the newest image will decrease in importance due to all the previous accumulations. This would be a temporal accumulation and the main issue with this approach is that when the scene or the camera changes all previous samples are not longer valid. To try and solve this drawback we added the motion vector information.

Motion vectors are vectors stored in a G-Buffer that indicates the camera movement difference between the last and new frame. To calculate these vectors in the geometry pass we need to bind the camera matrices and the previous camera matrices. This pass will always receive the present matrices and the matrices from the immediately last frame. In the vertex shader we can transform

a vertex position to both new and past normalized device coordinates (NDC). This coordinates are then passed to the rasterisation pass to convert them to 2D coordinates on the screen. In the fragment shader, once we have NDC and the previous NDC coordinates we can compute the motion vector as a result of both positions in screen. This vector represents the reprojection in space of a pixel from the last moment to the newest one.

```
1        // In vertex shader
2        //-----------------
3        mat4 transformationMatrix = projection \cdot view \cdot modelMatrix;
4        mat4 previousTransformation = previousProjection \cdot previouseView \cdot
             modelMatrix;
5        ndc = transformationMatrix \cdot vec4(position, 1);
6        previousNdc = previousTransformation \cdot vec4(position, 1);
7
8        // In fragment shader
9        //--------------------
10       vec3 ndc = inNdc.xyz / inNdc.w;
11       vec3 previousNdc = inPrevNdc.xyz / inPrevNdc.w;
12       vec2 motion = ndc.xy - previousNdc.xy;
```

Figure 18: Motion vector calculations.

With this information we can compute the average between the new shadow factor and the past shadow factor from the reprojected position to slightly reduce the noise when moving the camera. In this case we only have the previous accumulated information and a totally new from a different position and this may not correspond. This is because when moving the camera new areas may appear, such as positions that were behind objects or outside the camera view before movement, and we have no previous information for those areas. Taking such drawback into consideration we cannot interpolate with the same alpha calculation approach, instead we compute the mix with a constant value. We found 0.2 was an acceptable value. If the value were to be very high it would be very noisy since most of the information would fall into the new value, but a very small factor can create the effect called ghosting. This ghosting effect creates image lag, forming what resembles and aura around sharp edges [19].

To differentiate when to temporally accumulate and when to use spatial reconstruction we made use of the number of frames passed to the shader. In CPU, when the application made a change to the scene or the camera moved, the frame counter restarted. This counter is then passed to the shader and indicates the number of frames since the scene was last changed. This is the same value used to compute the *alpha* for the colour interpolation. With only one sample per pixel and the methods exposed above the application achieves the result shown in figure 19.

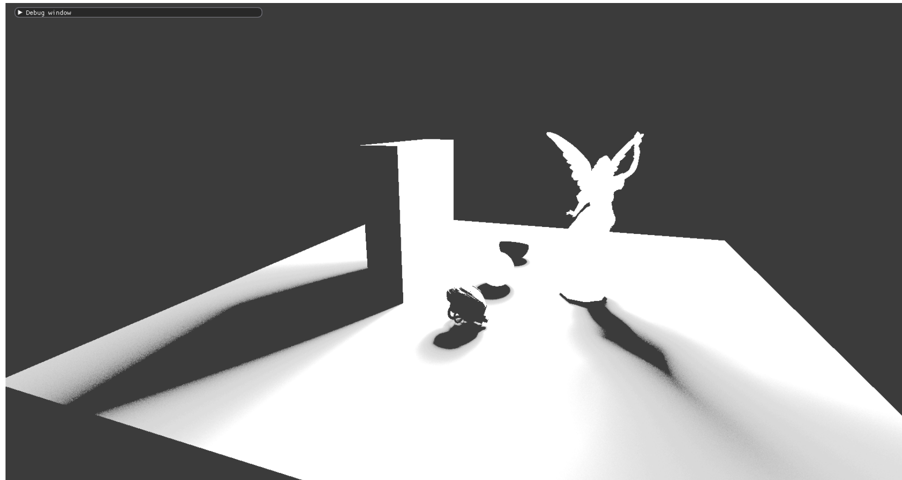

Figure 19: Shadows traced with 1spp with spatio-temporal accumulation.

One important issue we faced during spatio-temporal accumulation was shadows created in the second moment. Until now we explained our approach to reduce noise for the shadows from the geometry directly in the view of the camera, but what about shadows created in the secondary rays phase? The information stored in shadow textures does not include the shadows traced in the pixels reflected in a mirror or behind a glass mesh so they are not accumulated appearing with noise.

4.4.4 Reflection and Refraction

Shadows and soft shadows were not the only subjects to be exploited with the use of ray tracing. Reflections and refractions are another main issue when trying to simulate them in a rasterized pipeline. Usually refracted and specially reflective surfaces, even though a very realistic look can be achieved with shading techniques such as the previously mentioned PBR, are not physically accurate. In this project we wanted to implement refractive and reflective surfaces using rays but we considered only perfect specular surfaces. This only includes perfect specular reflection and perfect specular refraction surfaces. These materials need to be raytraced in a recursive manner in the shading pass. There rays are shot for specific materials with the intention to return the colour contribution in that pixel.

In this pass not only reflective and refractive rays will be computed. It is also a shading pass so light contribution will be the first thing to consider. Given the array of lights and its properties we will loop through them and calculate its contribution to the surface. Depending on the material type we treat them differently. In this project we differentiate three types of surfaces: PBR surfaces, perfect reflective surfaces and perfect refractive surfaces. In the rasterisation pass, each mesh is drawn by a draw call from the rendering system and so we know from which material that mesh made of. The material information, as previously exposed in section 4.3, contains an integer value representing the surface type and thus we can now know of which surface type is the pixel being shaded and act accordingly.

```
// If the shading mode is reflective
else if( shadingMode == 3 )
{
    const vec3 reflected  = reflect(normalize(gl_WorldRayDirectionEXT), N);
    const bool isScattered = dot(reflected, N) > 0;

    Lo += (NdotL > 0.0 && light_intensity > 0.0) ?
        light_intensity * light.color.xyz * attenuation * albedo * metallic :
        irradiance * albedo * metallic;
    direction = vec4(reflected, isScattered ? 1 : 0);
}
```

Figure 20: Shader snippet. Computing the reflective properties to be returned in the ray-generation shader.

A perfect specular reflection would be a mirror like surface, where all microfacets are aligned and the light arriving at all points in the surface is perfectly reflected in the same direction. We can then consider and treat such surface as if only one big microfacet forms the whole goemetry and all light bounce in the same direction. In case the value from the pixel being shaded references a perfect reflective surface we compute the necessary information to shot a bouncing ray. Luckily GLSL language has a predefined function $reflect$ that automatically calculates the bouncing vector given V, representing the incident vector, and the surface normal N. The functions works as follows:

$$Dir = V - 2.0 \cdot dot(N, V) \cdot N.$$

The output is the perfect reflected ray direction. This vector will then be passed to the traceRayEXT() function that traces a ray in the given direction.

Figure 21: Ray traced perfect specular reflections.

In case the pixel material information references a refractive surface we will proceed to compute the direction differently. When the light enters into a different material, which changes the medium the light is travelling through, instead of being bounced off, it is refracted. As we are treating with perfect refraction surfaces light is only refracted in one direction returning resulting in an image well defined. The more the beam scatter the blurrier its result. In a ray tracing application that would return in a noise output but we do not implement such case.

To calculate the refracted vector we followed Snell's law where we find out that the angle of incidence relates to the angle of refraction:

$$\frac{sin\alpha}{sin\beta} = \frac{\eta_2}{\eta_1} = \eta_t$$

and by using Snell's law we can obtain

$$\omega_t = -\omega_i \cdot \eta_t + N(\eta_t cos\alpha - cos\beta)$$

where

$$cos\beta = \sqrt{1 + \eta_t^2(cos^2\alpha - 1)}$$

and from $cos\beta$ is important to highlight that the radicand can be negative, thus no solution is found. When such case happens it means the ray is reflected in ω_r direction. This phenomena is known as **total internal reflection**. When the incident angle is greater than a value called **critical angle**, which is an angle of incidence that yields total reflection, light instead of being refracted is completely reflected. The critical angle is the angle of incidence that yields total reflection and can be defined as

$$\Theta = arcsin(\frac{\eta_2}{\eta_1})$$

where η represents the index of refraction. This only happen when light changes from one medium to another with a lower index of refraction, defined if $\eta_2 \leq \eta_1$. The index of refraction is unique and constant for each material.

```
// If the shading mode is refractive
else if( shadingMode == 4 )
{
    const float ior     = mat.diffuse.w;
    const float NdotV   = dot( N, normalize(gl_WorldRayDirectionEXT));
    const vec3 refrNormal = NdotV > 0.0 ? -N : N;
    const float refrEta = NdotV > 0.0 ? 1 / ior : ior;

    Lo += (light_intensity > 0.0) ?
        light_intensity * light.color.xyz * attenuation * albedo * metallic :
        irradiance * albedo * metallic;

    float radicand = 1 + pow(refrEta, 2.0) * (NdotV * NdotV - 1);
    direction = radicand < 0.0 ?
        vec4(reflect(gl_WorldRayDirectionEXT, N), 1) :
        vec4(refract( normalize(gl_WorldRayDirectionEXT), refrNormal, refrEta ), 1);
}
```

Figure 22: Shader snippet. Computing the refractive properties to be returned to the ray-generation shader.

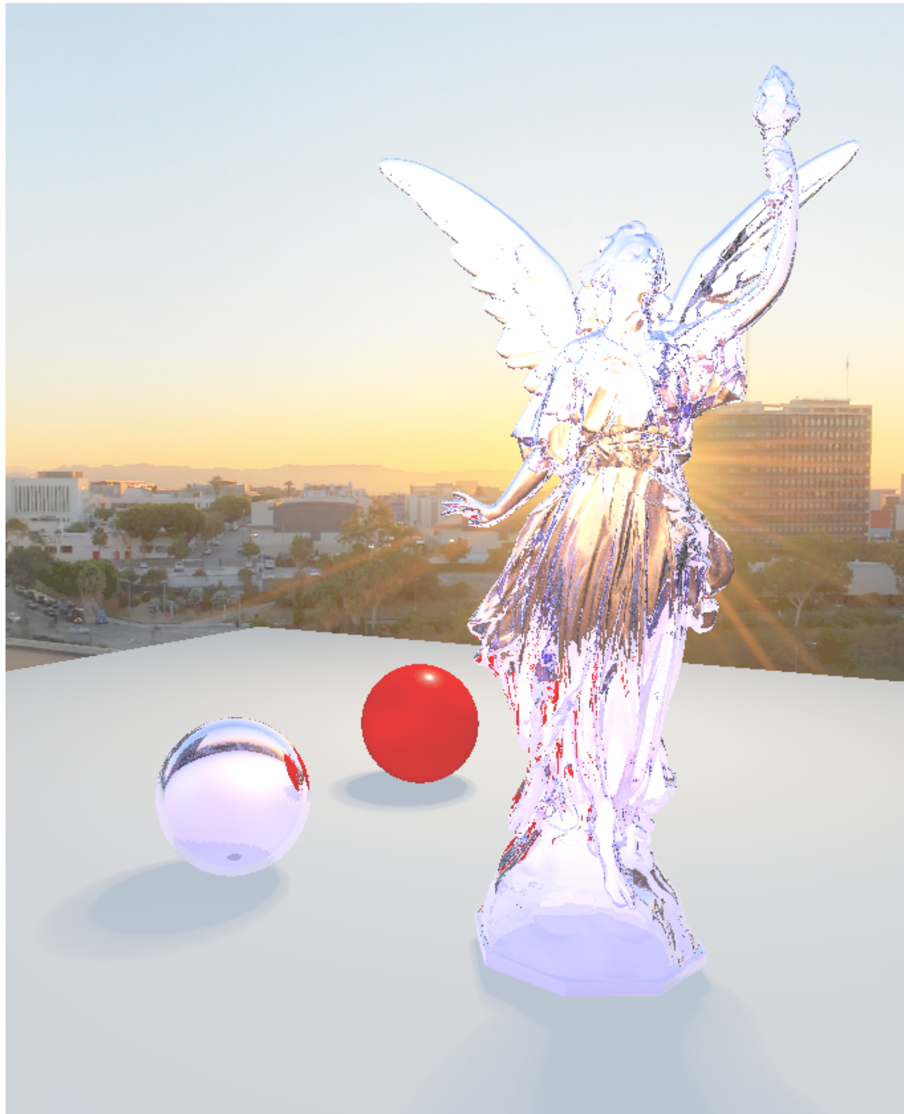

Figure 23: Ray traced perfect specular refraction.

Once we shaded the direct visibility of our scene and computed the directions in the pixels that required secondary rays we proceed to shoot a ray. This `traceRayEXT()` call differs from the previous shadow call in the flags given as parameters. This time the only required flag is `gl_RayFlagsOpaqueEXT` which checks hits to opaque geometry.

In case the ray does not hit anything the miss shader is called and returns the background colour. In case a geometry is intersected the closest-hit shader is called and similar computations to the ray-generation shader are carried. This time we do not have the assistance of G-Buffers for

the data at that point since it may have not been visible in the rasterized pass. Thus we have to benefit from vertex and indices buffers. Luckily the shader can make use of a custom variable `gl_InstanceCustomIndexEXT`, added by the user when building the acceleration structure, which indicates the TLAS instance hit. Knowing which TLAS instance we hit we can retrieve the IDs stored in an array. This IDs helps us unpack the geometry data stored in other buffers. Each position in the array is bound to an instance and stores a four component vector, each component correspond to one of the following indices:

- **instanceID:** this index represents the position inside the indices buffer. This buffer is a 2D array where each position contains the indices for a given mesh.
- **materialID:** this index indicates the position in the materials array that corresponds to the mesh.
- **transformationID:** this index indicates the position in the transformation matrices array that corresponds to the mesh. Transformations are useful in the close-hit pass to convert Normal vectors to the corresponding direction.
- **firstIndex:** this index helps us find the corresponding index inside the indices buffer.

With the help of the indices mention above we can unpack all necessary data. The process will be similar to the ray-generation shader. We loop for each light and compute its contribution to the surface depending on the material. In here rays can still keep on bouncing and new directions must be calculated again. Nevertheless we do not shoot rays, even though it is possible, from the closest-hit ray. Instead we will return all necessary information to the ray-generation shader where it will recursively shoot rays. The way data is compacted in the payload is as follows:

```
1   struct hitPayload{
2       vec4 colourAndDist
3       vec4 direction
4       vec4 origin
5       uint seed
6   }
```

The first vector stores the RGB colour computed in the first three components and the distance the ray has travelled in the last component. The direction vector stores the new direction in the first three components and 0 if no more bounces are needed. Any other number would indicates that more rays have to be shot. The origin vector stores the new origin of the ray. Finally the new seed is stored in the last value for it to be used in the next random generation number. Figure 24 shows how the payload is inflated in the closest-hit shader to be returned to the ray-generation shader.

```
// In to of the shader the payload is defined as follows
layout (location = 0) rayPayloadInEXT hitPayload prd;
//
//... Closest-hit Shader
//
// In the end, the payload prd receives the parameters computed above
prd = hitPayload(vec4(color, gl_HitTEXT), direction, origin, prd.seed);
```

Figure 24: Returning the payload in the closest-hit shader.

In the ray-generation shader we loop for a maximum depth of ten shooting rays when necessary, ending the loop otherwise.

```
// In ray generation shader we iterate until the maximum depth is reached or if no
    more rays are needed.
for(int depth = 0; depth < MAX_RECURSIONS; depth++)
{
    if(shadingMode == 0) // if not reflection or refraction material
        break;

    traceRayExt(...)
    finalColour *= payload.colourAndDist.xyz;
    const float hitDistance = payload.colourAndDist.w;
    const bool isScattered = payload.direction.w > 0;

    if( hitDistance < 0 || !isScattered )
        break;

    origin = payload.origin.xyz;
    direction = payload.direction.xyz;
}
```

Figure 25: Launching rays until max recursion value is reached or no more rays are needed.

5 Results

OS	Windows 10 build version 2004
Compiler	MSVC
CPU	AMD Ryzen 5 2600X Six-Core Processor
GPU	NVIDIA GeForce RTX 2060
Memory	12GB DDR4

Figure 26: System properties used for development and testing.

All development and testing was done using the system with properties shown in figure 26. The application supports multiple pipelines including a fully ray traced one which compared with the hybrid ray-traced pipeline no much difference could be seen except for the jaggies that appear due to the information stored in the G-Buffers which cannot take advantage of the anti-aliasing. Both results can be seen in figures 27 and 28. All application results were taken at a resolution of 1280x720. The software NVIDIA Nsight Systems [20] has been used for profiling purposes. It is important to highlight that the software also takes a toll to the GPU decreasing a little bit the application performance. This can be translated as an average of 30 to 50 frames per second drop compared to a version not being profiled.

Figure 27: Fully ray-traced scene

Figure 28: Hybrid scene

Given the use of G-Buffers at previous passes and using a resolution of 1280x720 pixels we avoid shooting primary rays, which means 921.600 rays are avoided. That huge number reduces significantly the performance cost as we can see using the NVIDIA Nsight tool. For the average scene shown in figure 28 we have an average of 294.86 frames per second which is an average of 3.39 ms per frame.

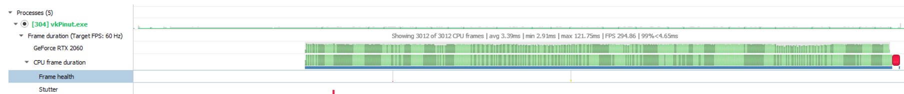

Figure 29: Average frame rate statistic of the dynamic scene.

Out of each frame we can see the ray tracing passes cost. As mentioned in the pipeline explanation our algorithm consists of two ray traced passes, the shadow pass which is the second in the pipeline and the most costly of the ray tracing due to its number of rays cast with a compute time of 0.37ms and the shading one with a compute cost of 0.326ms. That second pass is the most variable since depends on the number of rays that have to be computed in the scene. This means that the number of pixels marked as refractive or reflective material has a direct impact on the generation cost of the image.

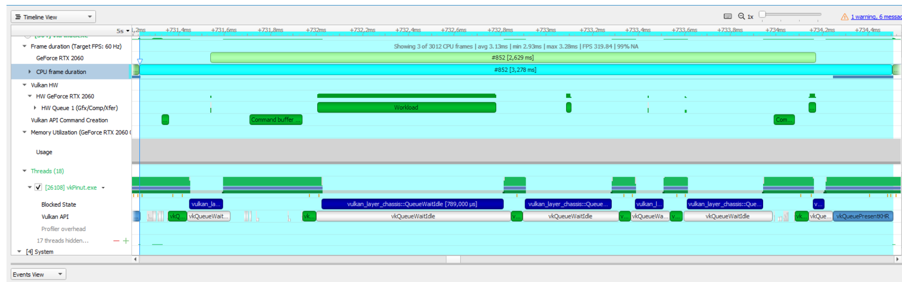

Figure 30: Hybrid frame profiling.

If we increase the number of objects with a refractive material component, which is also the most costly to compute due to the number of rays involved, we can see a considerably drop in performance. For such test two glass lucy model have been included in the scene as well as an extra glass ball. The average frames per second is down to 174 and an average 5.73 ms per frame. In this case the shading pass can increase up to a 2ms but it is compensated with a drop cost on tracing shadows since refractive materials, though not physically correct in reality, we decided would not be affected by their own shadow and the number of rays is decreased 31. The cost is still affected.

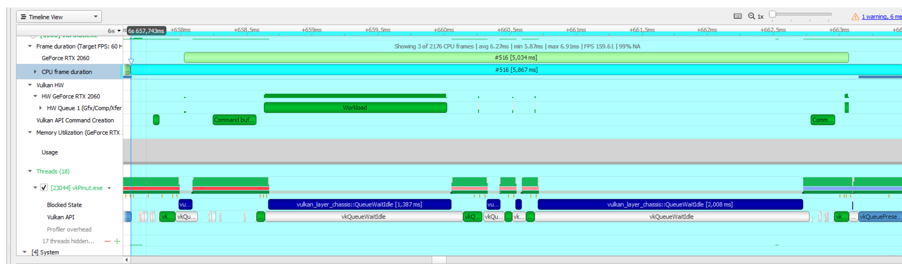

Figure 31: Frame with number of reflection and refraction meshes increased.

Another important subject that can affect performance is the number of triangles in the scene. Though it has more impact to the first pass where geometry is being rendered and the G-Buffers created. It also has its repercussions in the ray tracing passes because of the number of computations that have to be done but it is far less damaging since the number of rays are roughly the same for all scenes given that the number of casts depends more on the resolution of the G-Buffers rather than the geometry of the scene itself. This can be exemplified in figure 32 where the geometry pass is the most costly, taking more time than both ray tracing passes combined.

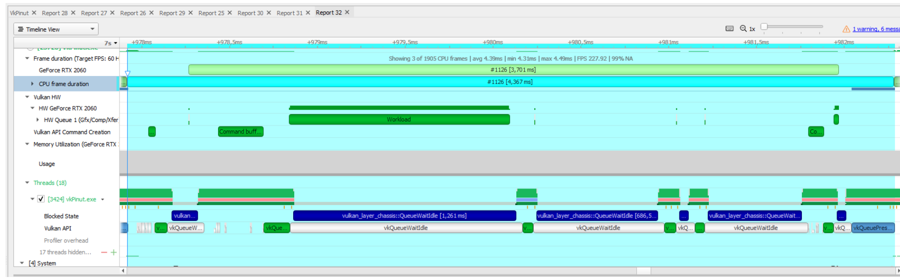

Figure 32: Frame with geometry increased.

6 Conclusion

This project aimed to present an hybrid approach to generating images in real-time through a combination of rasterisation and ray-tracing. The main goals was to reduce the number of necessary rays and the addition of reflective, refractive and soft shadows into the scene, topics only available through the use of rays, while maintaining a minimum of 30 frames per second to create the feeling of interaction.

The first goal was achieved through the implementation and advantage of the data given by the G-Buffers, which avoided the necessity to shoot primary rays. As mentioned in the results sections it allowed to reduce the number of rays which in a pure ray tracing pass would directly depend on the resolution of the image. Compared to the pure ray tracing pass we can see that even though the optimisation is not as much as expected, it still exists. On the other hand we did not expect that creating the G-Buffers in the first pass would take such a toll in performance and more work into optimise such pass is necessary.

For the second goal we could implement all three key points without having a huge drop in performance. In fact, performance for the scene tested was higher than expected, being around 200 frames per second and more. The main issue was not being able to implement a perfect denoising technique. The temporal-accumulation pass achieves great results when the scene is static but it can only blur the noise up to a certain point when the scene is dynamic. Thus, noise can be noticed when moving around the scene or interacting with the objects. This only affects to the soft shadows since perfect specular reflections and refractions were implemented. Had we implemented generic specular reflections and refractions based on the materials properties would result in very noisy images, with spatio-temporal accumulation not being enough to produce a perfectly clean image due to the drawbacks mentioned.

7 Future Work

There is much room for improvement in this project and in this section some of the main paths to achieve better results will be presented for further exploration.

7.1 Reducing rays for reflection and refraction

One way to reduce the time cost when computing the ray tracing passes is to reduce the number of rays shoot. To do so a reduction of the resolution of the material G-Buffer can be considered. That would not only reduce the number of rays shoot, but also the VRAM it takes in the GPU. One thing to consider would be that it may produce a ray to pixels where no reflection or refraction surface is represented and also may create noisy and low resolution reflections.

Another approach would be select small areas of 4 pixels and group them to shoot one ray reducing the number of rays and making sure they represent a reflective or refractive surface. Still in both cases a good denoising algorithm should be implemented to make up for the lost of information since less samples are being taken. Careful profiling should be done to evaluate if the loss in information is worth the performance improvement. It may be a good option for specific scene with no perfect specular reflections and refractions.

7.2 Reducing rays for shadows

In the project we use fully ray traced shadows. This implies that a ray must be shoot for every pixel in the image and an average is then compute to converge to a final soft shadow image. In order to reduce the number of rays we could take advantage of a previous shadow pass where shadows are rasterized and return a shadow map. Such output could benefit the programmer when shooting rays allowing him to trace rays only in the edges of the shadows to smooth them or to approximate it to a soft shadow result. Further exploration in this approach would involve computing the time to create the shadow map plys ray tracing just the edges afterwards and both rasterized and ray traced shadows combined may result in a lower performance output.

7.3 Acceleration Structure Instancing

Using instancing when creating or rebuilding an acceleration structure is important to reduce the amount of data loaded. This would reuse the data of a previous mesh to create a new acceleration structure instance without the necessity to load the data again, avoiding duplicity. Using this we could reuse data from a mesh that appear multiple times in the scene.

7.4 Multi-Threading

One key advantage of using a low-level API as Vulkan is the ability to use multiple threads to parallel some tasks. This is a more advanced approach when learning a new complex API such as Vulkan but a powerful one. Multi-threading could help us to reduce the CPU cost of creating and rebuilding acceleration structures. In this project, to reduce the cost of rebuilding the acceleration structure each frame we considered only when the scene changed, thus when an object was being moved. This would not be an option for an application with permanent changing scenarios or

having animated models. But rebuilding an acceleration structure each frame can end up being a heavy task and multi-threading could take advantage of all cores to reduce the process time cost.

As an example, we could take advantage of one thread to rebuild the acceleration structure of the scene for each frame while recording and submitting commands for previous passes that do not use the acceleration structure. In our case that would be recording and submitting the geometry commands while rebuilding the acceleration structure for the subsequent passes. If we take a look in the result sections we can notice that the AS creation for our simple scene can take up to 0.2ms that could be performed while submitting the geometry pass instead of taking time before.

8 Annex

Geometry vertex shader

```
#version 460

#extension GL_GOOGLE_include_directive : enable

layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec3 inNormal;
layout(location = 2) in vec3 inColor;
layout(location = 3) in vec2 inUV;

layout(location = 0) out vec3 outPosition;
layout(location = 1) out vec3 outNormal;
layout(location = 2) out vec3 outColor;
layout(location = 3) out vec2 outUV;
layout(location = 4) out vec4 ndcPrev;
layout(location = 5) out vec4 ndc;

struct ObjectData{
   mat4 model;
};

// Set 0 - Camera information
layout(set = 0, binding = 0) uniform CameraBuffer
{
   mat4 view;
   mat4 projection;
   mat4 pView;
   mat4 pProj;
} cameraData;

layout(push_constant) uniform constants
{
   mat4 matrix;
   mat4 inv_matrix;
}pushC;

void main()
{
   mat4 transformationMatrix  = cameraData.projection * cameraData.view * pushC.matrix;
   mat4 previousTransformation = cameraData.pProj * cameraData.pView * pushC.matrix;
   gl_Position           = transformationMatrix * vec4(inPosition, 1.0);

   outPosition = vec3(pushC.matrix * vec4(inPosition, 1.0)).xyz;
    outColor   = inColor;
   outNormal   = mat3(transpose(pushC.inv_matrix)) * vec3(inNormal);
    outUV      = inUV;
   ndc      = transformationMatrix * vec4(inPosition, 1.0); // in homogeneous space
```

```
    ndcPrev  = previousTransformation * vec4(inPosition, 1.0);
}
```

Geometry fragment shader

```
#version 460

struct Light{
    vec4 pos;  // w used for max distance
    vec4 color; // w used for intensity
    float radius;
};

layout (location = 0) out vec4 outFragColor;
layout (location = 0) in vec2 inUV;
layout (location = 1) in vec3 inCamPosition;

layout (set = 0, binding = 0) uniform sampler2D positionTexture;
layout (set = 0, binding = 1) uniform sampler2D normalTexture;
layout (set = 0, binding = 2) uniform sampler2D albedoTexture;
layout (set = 0, binding = 3) uniform sampler2D motionTexture;
layout (std140, set = 0, binding = 4) buffer LightBuffer {Light lights[];} lightBuffer;
layout (set = 0, binding = 5) uniform debugInfo {int target;} debug;
layout (set = 0, binding = 6) uniform sampler2D materialTexture;
layout (set = 0, binding = 8) uniform sampler2D emissiveTexture;
layout (set = 0, binding = 9) uniform sampler2D environmentTexture;

const float PI = 3.14159265359;

float DistributionGGX(vec3 N, vec3 H, float a);
float GeometrySchlickGGX(float NdotV, float k);
float GeometrySmith(vec3 N, vec3 V, vec3 L, float k);
vec3 FresnelSchlick(float cosTheta, vec3 F0);

void main()
{
    vec3 position  = texture(positionTexture, inUV).xyz;
    vec3 normal   = texture(normalTexture, inUV).xyz * 2.0 - vec3(1);
    vec3 albedo   = texture(albedoTexture, inUV).xyz;
    vec3 motion   = texture(motionTexture, inUV).xyz * 2.0 - vec3(1);
    vec3 material  = texture(materialTexture, inUV).xyz;
    vec3 emissive  = texture(emissiveTexture, inUV).xyz;
    bool background = texture(positionTexture, inUV).w == 0 && texture(normalTexture,
        inUV).w == 0;
    float metallic   = material.z;
    float roughness = material.y;

    vec3 N        = normalize(normal);
    vec3 V        = normalize(inCamPosition - position.xyz);
    float NdotV   = max(dot(N, V), 0.0);
```

```
45   vec3 F0    = mix(vec3(0.04), pow(albedo, vec3(2.2)), metallic);
46   vec2 envUV    = vec2(0.5 + atan(N.x, N.z) / (2 * PI), 0.5 - asin(N.y) / PI);
47   vec3 irradiance = texture(environmentTexture, envUV).xyz;
48
49   if(debug.target > 0.001)
50   {
51      switch(debug.target){
52         case 1:
53            outFragColor = vec4(position, 1);
54            break;
55         case 2:
56            outFragColor = vec4(normal, 0);
57            break;
58         case 3:
59            outFragColor = vec4(albedo, 1);
60            break;
61         case 4:
62            outFragColor = vec4(motion, 1);
63            break;
64         case 5:
65            outFragColor = vec4(material, 1);
66            break;
67         case 6:
68            outFragColor = vec4(emissive, 1);
69            break;
70      }
71      return;
72   }
73
74   vec3 color = vec3(1), Lo = vec3(0);
75   float attenuation = 1.0, light_intensity = 1.0;
76
77   for(int i = 0; i < lightBuffer.lights.length(); i++)
78   {
79      Light light      = lightBuffer.lights[i];
80      bool isDirectional  = light.pos.w < 0;
81      vec3 L           = isDirectional ? light.pos.xyz : (light.pos.xyz - position.xyz);
82      vec3 H           = normalize(V + normalize(L));
83      float NdotL      = max(dot(N, normalize(L)), 0.0);
84
85      // Calculate the directional light
86      if(isDirectional)
87      {
88         Lo += (NdotL * light.color.xyz);
89      }
90      else  // Calculate point lights
91      {
92         float light_max_distance  = light.pos.w;
93         float light_distance     = length(L);
94         light_intensity       = light.color.w / (light_distance * light_distance);
```

```

         attenuation = light_max_distance - light_distance;
         attenuation /= light_max_distance;
         attenuation = max(attenuation, 0.0);
         attenuation = attenuation * attenuation;

         vec3 radiance = light.color.xyz * light_intensity * attenuation;

         float NDF  = DistributionGGX(N, H, roughness);
         float G  = GeometrySmith(N, V, L, roughness);
         vec3 F     = FresnelSchlick(max(dot(H, V), 0.0), F0);
         vec3 kD = vec3(1.0) - F;
         kD *= 1.0 - metallic;

         vec3 numerator     = NDF * G * F;
         float denominator  = 4.0 * NdotV * max(dot(N, L), 0.0);
         vec3 specular    = numerator / max(denominator, 0.001);

         vec3 kS = F;

         Lo += (kD * pow(albedo, vec3(2.2)) / PI + specular) * radiance * NdotL;
      }
   }

   if(!background){
      // Ambient from IBL
      vec3 F = FresnelSchlick(NdotV, F0);
      vec3 kD = (1.0 - F) * (1.0 - metallic);
      vec3 diffuse = kD * albedo * irradiance;
      vec3 ambient = diffuse;

      color = Lo + ambient;
      color += emissive;
   }
   else{
      color = albedo;
   }
   outFragColor = vec4( color, 1.0f );
}

float DistributionGGX(vec3 N, vec3 H, float roughness)
{
  float a = roughness * roughness;
   float a2 = a * a;
   float NdotH = max(dot(N, H), 0.0);
   float NdotH2 = NdotH * NdotH;

   float denom = (NdotH2 * (a2 - 1.0) + 1.0);
   denom = PI * denom * denom;

```

```
145     return a2 / denom;
146  }
147
148  // Geometry Function
149  float GeometrySchlickGGX(float NdotV, float roughness)
150  {
151    float r = roughness + 1.0;
152    float k = (r * r) / 8.0;
153
154     float nom = NdotV;
155     float denom = NdotV * (1.0 - k) + k;
156
157     return nom/denom;
158  }
159
160  float GeometrySmith(vec3 N, vec3 V, vec3 L, float roughness)
161  {
162     float NdotV = max(dot(N, V), 0.0);
163     float NdotL = max(dot(N, L), 0.0);
164     float ggx1 = GeometrySchlickGGX(NdotV, roughness);
165     float ggx2 = GeometrySchlickGGX(NdotL, roughness);
166
167     return ggx1 * ggx2;
168  }
169
170  // Fresnel Equation
171  vec3 FresnelSchlick(float cosTheta, vec3 F0)
172  {
173     return F0 + (1.0 - F0) * pow(1.0 - cosTheta, 5.0);
174  }
```

Shadow ray-generation shader

```
1   #version 460
2   #extension GL_EXT_ray_tracing : require
3   #extension GL_GOOGLE_include_directive : enable
4   #extension GL_EXT_nonuniform_qualifier : enable
5
6   #include "helpers.glsl"
7
8   layout(binding = 0) uniform accelerationStructureEXT topLevelAS;
9   layout(binding = 1, rgba8) uniform image2D[3] shadowImage;
10  layout(binding = 2) uniform CameraProperties
11  {
12     mat4 viewInverse;
13     mat4 projInverse;
14     float frame;
15  } cam;
16  layout(binding = 3, std140) buffer Lights { Light lights[]; } lightsBuffer;
17  layout(binding = 4) uniform SampleBuffer {int samples;} samplesBuffer;
```

```
18 layout(binding = 5) uniform sampler2D[3] gbuffers;
19 layout(binding = 6) buffer MaterialBuffer { Material mat[]; } materials;
20
21 struct shadowPayload{
22    uint seed;
23    float frame;
24 };
25
26 layout(location = 0) rayPayloadEXT bool shadowed;
27
28 void main()
29 {
30    int frame = int(cam.frame);
31    uint seed = tea(gl_LaunchIDEXT.y * gl_LaunchSizeEXT.x + gl_LaunchIDEXT.x, uint(frame));
32
33    const vec2 pixelCenter = vec2(gl_LaunchIDEXT.xy) + vec2(0.5);
34    const vec2 inUV = pixelCenter / vec2(gl_LaunchSizeEXT.xy);
35
36    vec3 position  = texture(gbuffers[0], inUV).xyz;
37    vec3 normal   = texture(gbuffers[1], inUV).rgb * 2.0 - vec3(1.0);
38    vec2 motion   = texture(gbuffers[2], inUV).xy * 2.0 - vec2(1.0);
39    float matIdx  = texture(gbuffers[0], inUV).w;
40    int mode      = int(materials.mat[int(matIdx)].shadingMetallicRoughness.x);
41    vec3 N = normalize(normal);
42
43    if(mode == 4)
44    {
45       for(int i = 0; i < lightsBuffer.lights.length(); i++)
46       {
47          imageStore(shadowImage[i], ivec2(gl_LaunchIDEXT.xy), vec4(1, 0, 0, 1));
48       }
49       return;
50    }
51
52    for(int i = 0; i < lightsBuffer.lights.length(); i++)
53    {
54       // Init basic light information
55       Light light                  = lightsBuffer.lights[i];
56       const bool isDirectional     = light.pos.w < 0;
57       vec3 L                       = isDirectional ? light.pos.xyz : (light.pos.xyz -
            position);
58       const float light_max_distance = light.pos.w;
59       const float light_distance   = length(L);
60       L                            = normalize(L);
61       const float NdotL            = clamp(dot(N, L), 0.0, 1.0);
62       const float light_intensity  = isDirectional ? 1.0 : (light.color.w /
            (light_distance * light_distance));
63       float shadowFactor           = 0.0;
64       int shadowSamples            = samplesBuffer.samples;
65
```

```
      if( NdotL > 0)
      {
         for(int s = 0; s < shadowSamples; s++)
         {
            if(light_distance < light_max_distance)
            {
               shadowed        = true;
               const vec3 dir     = sampleDisk(light, position, L, seed);
               const uint flags   = gl_RayFlagsOpaqueEXT |
                   gl_RayFlagsTerminateOnFirstHitEXT | gl_RayFlagsSkipClosestHitShaderEXT;
               float tmin = 0.001, tmax = light_distance + 1;

               // Shadow ray cast
               traceRayEXT(topLevelAS, flags, 0xFF, 0, 0, 0,
               position + dir * 1e-1, tmin, dir, tmax, 0);
            }
            else{
               shadowed = false;
            }

            if(!shadowed){
              shadowFactor++;
            }
         }
         shadowFactor /= shadowSamples;
      }

      vec3 color = vec3(shadowFactor);
      imageStore(shadowImage[i], ivec2(gl_LaunchIDEXT.xy), vec4(color, 1));
   }
}
```

Shadow miss shader

```
#version 460
#extension GL_EXT_ray_tracing : enable

layout(location = 1) rayPayloadInEXT bool shadowed;

void main()
{
    shadowed = false;
}
```

Compute shader

```
#version 450

#extension GL_EXT_nonuniform_qualifier : enable
```

```
4  #extension GL_EXT_scalar_block_layout : enable
5
6  const int SIZE = 9;
7  const int NLIGHTS = 3;
8
9  const int WIDTH = 1700;
10 const int HEIGHT = 900;
11
12 layout (local_size_x = 16, local_size_y = 16) in;
13 layout (binding = 0, rgba8) uniform readonly image2D[NLIGHTS] inputImage;
14 layout (binding = 1, rgba8) uniform image2D[NLIGHTS] outputImage;
15 layout (binding = 2) uniform FrameCount {int frame;} frameBuffer;
16 layout (binding = 3) uniform sampler2D motionTexture;
17
18 struct ImageData
19 {
20    float r[SIZE];
21    float g[SIZE];
22    float b[SIZE];
23 } imageData;
24
25 float conv(in float[SIZE] kernel, in float[SIZE] data, in float denom, in float offset)
26 {
27    float res = 0.0;
28    for (int i = 0; i < SIZE; i++)
29    {
30       res += kernel[i] * data[i];
31    }
32    return clamp(res/denom + offset, 0.0, 1.0);
33 }
34
35 void main()
36 {
37    for(int l = 0; l < NLIGHTS; l++)
38    {
39       int frame = frameBuffer.frame;
40       vec3 pixelColor;
41       if(frame > 0)
42       {
43          vec3 center = imageLoad(inputImage[l], ivec2(gl_GlobalInvocationID.xy)).rgb;
44          vec3 old = imageLoad(outputImage[l], ivec2(gl_GlobalInvocationID.xy)).rgb;
45          float a = 1.0 / float(frame + 1.0);
46          pixelColor = mix(old, center, a);
47          imageStore(outputImage[l], ivec2(gl_GlobalInvocationID.xy), vec4(pixelColor,
              1.0));
48       }
49       else
50       {
51          vec2 motionUV = vec2(float(gl_GlobalInvocationID.x) / float(WIDTH),
              float(gl_GlobalInvocationID.y) / float(HEIGHT));
```

```
        vec2 reprojectedUV = texture(motionTexture, motionUV).rg * 2.0 - vec2(1.0);
        vec2 lastUV = vec2(gl_GlobalInvocationID.x + reprojectedUV.x * WIDTH,
            gl_GlobalInvocationID.y + reprojectedUV * HEIGHT);

        vec3 center = imageLoad(inputImage[l], ivec2(gl_GlobalInvocationID.xy)).rgb;
        vec3 minColor = center;
        vec3 maxColor = center;

        for(int y = -1; y <= 1; y++)
        {
           for(int x = -1; x <= 1; x++)
           {
              if(x == 0 && y == 0)
                 continue;

              ivec2 offsetUV = ivec2(gl_GlobalInvocationID.x + x, gl_GlobalInvocationID.y
                  + y);
              vec3 color = imageLoad(inputImage[l], offsetUV).rgb;
              minColor = min(minColor, color);
              maxColor = max(maxColor, color);
           }
        }

        vec3 old = imageLoad(outputImage[l], ivec2(lastUV)).rgb;
        old = max(minColor, old);
        old = min(maxColor, old);

        float a = 0.4f;
        pixelColor = mix(old, center, a);
        imageStore(outputImage[l], ivec2(gl_GlobalInvocationID.xy), vec4(pixelColor,
            1.0));
     }
   }
}
```

Ray-generation shader and shading pass

```
#version 460
#extension GL_EXT_ray_tracing : require
#extension GL_EXT_nonuniform_qualifier : enable
#extension GL_EXT_scalar_block_layout : enable
#extension GL_GOOGLE_include_directive : enable

#include "raycommon.glsl"
#include "helpers.glsl"

layout (set = 0, binding = 0) uniform accelerationStructureEXT topLevelAS;
layout (set = 0, binding = 1, rgba8) uniform image2D image;
layout (set = 0, binding = 2) uniform CameraProperties
{
```

```
14    mat4 viewInverse;
15    mat4 projInverse;
16    vec4 frame;
17 } cam;
18 layout (set = 0, binding = 3) uniform sampler2D[] gbuffers;
19 layout (set = 0, binding = 4) buffer Lights { Light lights[]; } lightsBuffer;
20 layout (set = 0, binding = 9) buffer MaterialBuffer { Material mat[]; } materials;
21 layout (set = 0, binding = 10) uniform sampler2D[] environmentTexture;
22 layout (set = 0, binding = 12, rgba8) uniform readonly image2D[] shadowImage;
23
24 layout(location = 0) rayPayloadEXT hitPayload prd;
25 layout(location = 1) rayPayloadEXT bool isShadowed;
26
27 void main()
28 {
29    uint frame = int(cam.frame.x);
30    prd.seed = tea(gl_LaunchIDEXT.y * gl_LaunchSizeEXT.x + gl_LaunchIDEXT.x, frame);
31
32     const vec2 pixelCenter = vec2(gl_LaunchIDEXT.xy) + vec2(0.5); // gl_LaunchIDEXT
        represents the floating-point pixel coordinates normalized between 0 and 1
33    const vec2 inUV    = pixelCenter/vec2(gl_LaunchSizeEXT.xy); //gl_LaunchSizeExt is the
        image size provided in the traceRayEXT function
34
35    float matIdx       = texture(gbuffers[0], inUV).w;
36    Material mat       = materials.mat[int(matIdx)];
37    int shadingMode    = int(mat.shadingMetallicRoughness.x);
38
39     vec3 position       = texture(gbuffers[0], inUV).xyz;
40    vec3 normal        = texture(gbuffers[1], inUV).xyz * 2.0 - vec3(1);
41    vec3 albedo        = pow(texture(gbuffers[2], inUV).xyz, vec3(2.2));
42    vec2 motion        = texture(gbuffers[3], inUV).xy;
43    vec3 material      = texture(gbuffers[4], inUV).xyz;
44    vec3 emissive      = texture(gbuffers[5], inUV).xyz;
45    bool background    = texture(gbuffers[0], inUV).w == 0 && texture(gbuffers[1], inUV).w
        == 0;
46
47    const float roughness  = material.y;
48    const float metallic  = material.z;
49    const vec3 F0 = mix(vec3(0.04), albedo, metallic);
50
51    // Using the pixel coordinates we can apply the inverse transformation of the view and
        procjection matrices of the camera to obtain
52    // the origin and target of the ray
53    const vec4 camPosition = cam.viewInverse * vec4(0,0,0,1);
54     const vec3 V      = normalize(camPosition.xyz - position);
55     const vec3 N      = normalize(normal);
56    const float NdotV     = clamp(dot(N, V), 0.0, 1.0);
57
58    // Environment
59    vec2 environmentUV = vec2(0.5 + atan(N.x, N.z) / (2 * PI), 0.5 - asin(N.y) / PI);
```

```
60    vec3 irradiance = texture(environmentTexture[1], environmentUV).xyz;
61
62    float tmin         = 0.001;
63    float tmax         = 1000.0;
64
65    vec3 finalColor    = vec3(0);
66    vec3 origin        = vec3(0);
67    vec3 direction       = vec3(0);
68    float attenuation     = 1.0;
69    float shadowFactor    = 0.0;
70
71    // Calculate the light influence for each light
72    vec3 rayColor = vec3(0.0);
73    for(int i = 0; i < lightsBuffer.lights.length(); i++)
74    {
75       Light light             = lightsBuffer.lights[i];
76       const bool isDirectional   = light.pos.w < 0;
77       vec3 L                  = isDirectional ? light.pos.xyz : (light.pos.xyz -
            position.xyz);
78       const float light_max_distance  = light.pos.w;
79       const float light_distance    = length(L);
80       const float light_intensity  = isDirectional ? 1.0f : (light.color.w /
            (light_distance * light_distance));
81       L                    = normalize(L);
82       const float NdotL          = clamp(dot(N, L), 0.0, 1.0);
83       shadowFactor            = imageLoad(shadowImage[i], ivec2(gl_LaunchIDEXT.xy)).x;
84
85       // Check if visible for light
86       if(NdotL > 0.0)
87       {
88          // Calculate attenuation factor
89          // ----------------------------
90          if(light_intensity == 0){
91             attenuation = 0.0;
92          }
93          else{
94             attenuation = light_max_distance - light_distance;
95             attenuation /= light_max_distance;
96             attenuation = max(attenuation, 0.0);
97             attenuation = isDirectional ? 0.3 : (attenuation * attenuation);
98          }
99       }
100      tmax = 1000.0;
101
102      // Calculate illumination
103      //-----------------------
104      // In case material is diffuse, no need to ray trace at the moment
105      if( shadingMode == 0 )
106      {
107         vec3 radiance  = light.color.xyz * light_intensity * attenuation * shadowFactor;
```

```
108        const vec3 H  = normalize(V + L);
109        float D    = DistributionGGX(N, H, roughness);
110        float G    = GeometrySmith(N, V, L, roughness);
111        vec3 F       = FresnelSchlick(max(dot(H, V), 0.0), F0);
112        vec3 kD = vec3(1.0) - F;
113        kD *= 1.0 - metallic;
114
115        vec3 numerator     = D * G * F;
116        float denominator  = 4.0 * NdotV * NdotL;
117        vec3 specular   = numerator / max(denominator, 0.001);
118
119        vec3 kS = F;
120
121        rayColor += (kD * albedo / PI + specular) * radiance * NdotL;
122     }
123     if( shadingMode == 3 )
124     {
125        direction       = reflect(normalize(-V), N);
126        origin          = position;
127        rayColor += (NdotL > 0.0 && light_intensity > 0.0) ?
128                light.color.xyz * light_intensity * attenuation * shadowFactor * albedo *
                    metallic :
129                irradiance * albedo * metallic;
130     }
131     if(shadingMode == 4)
132     {
133        float ior          = mat.diffuse.w;
134        origin             = position;
135        const float cosAlpha  = dot(N, V);
136        const vec3  I      = -V; // incident ray
137        float NdotI        = dot( N, I );
138        vec3 refrNormal    = NdotI > 0.0 ? -N : N;
139        float refrEta      = NdotI > 0.0 ? 1 / ior : ior;
140
141        prd.direction.w    = 1;
142
143        float radicand   = 1 + pow(refrEta, 2.0) * (cosAlpha * cosAlpha - 1);
144        direction     = radicand < 0.0 ? reflect(I, N) : refract(I, refrNormal, refrEta);
145        rayColor += (light_intensity > 0.0) ?
146                light_intensity * light.color.xyz * attenuation * albedo * metallic :
147                irradiance * albedo * metallic;
148     }
149  }
150
151  // RECURSION IF NEEDED FOR BOUNCING
152  // -------------------------------
153  for(int depth = 0; depth < MAX_RECURSION; depth++)
154  {
155     if(shadingMode == 0)
156        break;
```

```
      traceRayEXT(topLevelAS, gl_RayFlagsOpaqueEXT, 0xff, 0, 0, 0, origin.xyz + direction
          * 1e-2, tmin, direction, tmax, 0);
      rayColor *= prd.colorAndDist.xyz;
      const float hitDistance = prd.colorAndDist.w;
      const bool isScattered  = prd.direction.w > 0;

      if( hitDistance < 0 || !isScattered){
         break;
      }
      else{
         origin.xyz    = prd.origin.xyz;
         direction.xyz  = prd.direction.xyz;
      }
   }

   finalColor = rayColor;

   // Ambient from IBL
   vec3 F        = FresnelSchlick(NdotV, F0);
   vec3 kD    = (1.0 - F) * (1.0 - metallic);
   vec3 diffuse  = kD * albedo * irradiance;
   vec3 ambient  = diffuse;

   finalColor += ambient + emissive;

   if(background)
      finalColor = albedo;

   imageStore(image, ivec2(gl_LaunchIDEXT.xy), vec4(finalColor, 1.0));
}
```

Closest-hit shader

```
#version 460
#extension GL_EXT_ray_tracing : require
#extension GL_EXT_nonuniform_qualifier : enable
#extension GL_EXT_scalar_block_layout : enable
#extension GL_GOOGLE_include_directive : enable

#include "raycommon.glsl"
#include "helpers.glsl"

layout (location = 0) rayPayloadInEXT hitPayload prd;
layout (location = 1) rayPayloadEXT bool shadowed;
hitAttributeEXT vec3 attribs;

layout (set = 0, binding = 0) uniform accelerationStructureEXT topLevelAS;
layout (set = 0, binding = 4) buffer Lights { Light lights[]; } lightsBuffer;
layout (set = 0, binding = 5, scalar) buffer Vertices { Vertex v[]; } vertices[];
```

```
17  layout (set = 0, binding = 6) buffer Indices { int i[]; } indices[];
18  layout (set = 0, binding = 7) uniform sampler2D[] textures;
19  layout (set = 0, binding = 8) buffer sceneBuffer { vec4 idx[]; } objIndices;
20  layout (set = 0, binding = 9) buffer MaterialBuffer { Material mat[]; } materials;
21  layout (set = 0, binding = 10) uniform sampler2D[] environmentTexture;
22  layout (set = 0, binding = 11, scalar) buffer Matrices { mat4 m[]; } matrices;
23
24  void main()
25  {
26    // Do all vertices, indices and barycentrics calculations
27    const vec3 barycentricCoords = vec3(1.0f - attribs.x - attribs.y, attribs.x, attribs.y);
28
29    vec4 objIdx = objIndices.idx[gl_InstanceCustomIndexEXT];
30
31    int instanceID      = int(objIdx.x);
32    int materialID      = int(objIdx.y);
33    int transformationID = int(objIdx.z);
34    int firstIndex      = int(objIdx.w);
35
36    ivec3 ind    = ivec3(indices[instanceID].i[3 * gl_PrimitiveID + firstIndex + 0],
37                         indices[instanceID].i[3 * gl_PrimitiveID + firstIndex + 1],
38                         indices[instanceID].i[3 * gl_PrimitiveID + firstIndex + 2]);
39
40    Vertex v0    = vertices[instanceID].v[ind.x];
41    Vertex v1    = vertices[instanceID].v[ind.y];
42    Vertex v2    = vertices[instanceID].v[ind.z];
43
44    const mat4 model    = matrices.m[transformationID];
45
46    // Use above results to calculate normal vector
47    // Calculate worldPos by using ray information
48    const vec3 normal   = v0.normal.xyz * barycentricCoords.x + v1.normal.xyz *
        barycentricCoords.y + v2.normal.xyz * barycentricCoords.z;
49    const vec2 uv       = v0.uv.xy * barycentricCoords.x + v1.uv.xy * barycentricCoords.y +
        v2.uv.xy * barycentricCoords.z;
50    const vec3 N        = normalize(mat3(transpose(inverse(model))) * normal).xyz;
51    const vec3 V        = normalize(-gl_WorldRayDirectionEXT);
52    const float NdotV   = clamp(dot(N, V), 0.0, 1.0);
53    const vec3 worldPos = gl_WorldRayOriginEXT + gl_WorldRayDirectionEXT * gl_HitTEXT;
54
55    // Init values used for lightning
56     vec3 Lo             = vec3(0);
57     float attenuation   = 1.0;
58    float light_intensity = 1.0;
59
60    // Init all material values
61    const Material mat         = materials.mat[materialID];
62    const int shadingMode      = int(mat.shadingMetallicRoughness.x);
63    vec3 albedo                = mat.textures.x > -1 ?
        texture(textures[int(mat.textures.x)], uv).xyz : mat.diffuse.xyz;
```

```
const vec3 emissive        = mat.textures.z > -1 ?
    texture(textures[int(mat.textures.z)], uv).xyz : vec3(0);
const vec3 roughnessMetallic = mat.textures.w > -1 ?
    texture(textures[int(mat.textures.w)], uv).xyz : vec3(0,
    mat.shadingMetallicRoughness.z, mat.shadingMetallicRoughness.y);

albedo                     = pow(albedo, vec3(2.2));
const float roughness      = roughnessMetallic.y;
const float metallic       = roughnessMetallic.z;
vec3 F0                    = mix(vec3(0.04), albedo, metallic);

// Environment
vec2 environmentUV = vec2(0.5 + atan(N.x, N.z) / (2 * PI), 0.5 - asin(N.y) / PI);
vec3 irradiance = texture(environmentTexture[1], environmentUV).xyz;

vec4 direction = vec4(1, 1, 1, 0);
vec4 origin = vec4(worldPos, 0);

for(int i = 0; i < lightsBuffer.lights.length(); i++)
{
  // Init basic light information
    Light light                     = lightsBuffer.lights[i];
    const bool isDirectional     = light.pos.w < 0;
    vec3 L                            = isDirectional ? light.pos.xyz : (light.pos.xyz -
        worldPos);
    const float light_max_distance  = light.pos.w;
    const float light_distance     = length(L);
  L                              = normalize(L);
    const float light_intensity   = isDirectional ? 1.0f : (light.color.w /
        (light_distance * light_distance));
  const vec3 H                   = normalize(V + L);
    const float NdotL                = clamp(dot(N, L), 0.0, 1.0);
  const float NdotH              = clamp(dot(N, H), 0.0, 1.0);
  float shadowFactor             = 1.0;

  // Check if light has impact
  // Calculate attenuation factor
  if(light_intensity == 0){
    attenuation = 0.0;
  }
  else{
    attenuation = light_max_distance - light_distance;
    attenuation /= light_max_distance;
    attenuation = max(attenuation, 0.0);
    attenuation = isDirectional ? 0.3 : attenuation * attenuation;
  }

  vec3 difColor = vec3(0);

  if(shadingMode == 0) // DIFUS
```

```
109     {
110       if(NdotL > 0)
111       {
112         for(int a = 0; a < 1; a++)
113         {
114           // Init as shadowed
115           shadowed          = true;
116           if(light_distance < light_max_distance)
117           {
118             vec3 dir         = sampleDisk(light, worldPos, L, prd.seed);
119             const uint flags = gl_RayFlagsOpaqueEXT | gl_RayFlagsTerminateOnFirstHitEXT |
                    gl_RayFlagsSkipClosestHitShaderEXT;
120             float tmin = 0.001, tmax = light_distance + 1;
121 
122             // Shadow ray cast
123             traceRayEXT(topLevelAS, flags, 0xff, 0, 0, 1,
124               worldPos.xyz + dir * 1e-2, tmin, dir, tmax, 1);
125           }
126 
127           if(!shadowed){
128             shadowFactor++;
129           }
130         }
131       }
132 
133       vec3 radiance = light_intensity * light.color.xyz * attenuation * shadowFactor;
134       vec3 F        = FresnelSchlick(NdotH, F0);
135       float D       = DistributionGGX(N, H, roughness);
136       float G       = GeometrySmith(N, V, L, roughness);
137 
138       vec3 numerator   = D * G * F;
139       float denominator = max(4.0 * clamp(dot(N, V), 0.0, 1.0) * NdotL, 0.000001);
140       vec3 specular    = numerator / denominator;
141 
142       vec3 kS = F;
143       vec3 kD = (vec3(1.0) - kS) * (1.0 - metallic);
144 
145       Lo += (kD * albedo / PI + specular) * radiance * NdotL;
146       direction = vec4(1, 1, 1, 0);
147     }
148     else if(shadingMode == 3) // MIRALL
149     {
150       const vec3 reflected  = reflect(normalize(gl_WorldRayDirectionEXT), N);
151       const bool isScattered = dot(reflected, N) > 0;
152 
153       Lo += (NdotL > 0.0 && light_intensity > 0.0) ?
154             light_intensity * light.color.xyz * attenuation * albedo * metallic :
155             irradiance * albedo * metallic;
156       direction = vec4(reflected, isScattered ? 1 : 0);
157     }
```

```
    else if(shadingMode == 4) // VIDRE
    {
      const float ior     = mat.diffuse.w;
      const float NdotV   = dot( N, normalize(gl_WorldRayDirectionEXT));
      const vec3 refrNormal = NdotV > 0.0 ? -N : N;
      const float refrEta = NdotV > 0.0 ? 1 / ior : ior;

      Lo += (light_intensity > 0.0) ?
            light_intensity * light.color.xyz * attenuation * albedo * metallic :
            irradiance * albedo * metallic;

      float radicand = 1 + pow(refrEta, 2.0) * (NdotV * NdotV - 1);
      direction = radicand < 0.0 ?
                  vec4(reflect(gl_WorldRayDirectionEXT, N), 1) :
                  vec4(refract( normalize(gl_WorldRayDirectionEXT), refrNormal, refrEta ),
                      1);
    }
  }

  // Ambient from IBL
  vec3 F       = FresnelSchlick(NdotV, F0);
  vec3 kD      = (1.0 - F) * (1.0 - metallic);
  vec3 diffuse = kD * albedo * irradiance;
  vec3 ambient = diffuse;

  vec3 color = Lo + ambient + emissive;
  prd = hitPayload(vec4(color, gl_HitTEXT), direction, origin, prd.seed);
}
```

Post vertex shader

```
#version 450
layout (location = 0) in vec3 inPosition;
layout (location = 1) in vec3 inNormal;
layout (location = 2) in vec3 inColor;
layout (location = 3) in vec2 inUV;

layout (location = 0) out vec2 outUV;

out gl_PerVertex
{
  vec4 gl_Position;
};

void main()
{
  outUV = inUV;
  gl_Position = vec4(outUV * 2.0f - 1.0f, 1.0f, 1.0f);
}
```

Post fragment shader

```
#version 450
layout(location = 0) in vec2 outUV;
layout(location = 0) out vec4 fragColor;

layout(set = 0, binding = 0) uniform sampler2D finalTexture;

void main()
{
  vec2 uv    = outUV;
  float gamma = 1. / 2.2;

  vec4 color = texture(finalTexture, uv);
  vec3 rgb = color.xyz;
  rgb = max(rgb, 0.001);
  //rgb = rgb / ( rgb + vec3(1.0));
  rgb = pow(rgb, vec3(gamma));

  fragColor = vec4(rgb, color.w);
}
```

References

[1] J. Agenjo, "Gràfics a temps real." Website available at https://tamats.com/upf/?page$_i d = 793$.

[2] J. de Vries, "Learn opengl website." Website available at https://learnopengl.com/Introduction.

[3] R. Guy and M. Agopian, "Physically based rendering in filament." Website available at https://google.github.io/filament/Filament.htmlabout.

[4] Wikipedia, "Subsurface scattering — Wikipedia, the free encyclopedia." Website available at https://en.wikipedia.org/wiki/Subsurface$_s cattering$.

[5] T. Akenine-Möller, *Real-time Rendering fourth edition.* CRC Press, 2018.

[6] M.-K. Lefrançois, P. Gautron, N. Bickford, and D. Akeley, "Nvidia vulkan ray-tracing tutorial." Website available at https://nvpro-samples.github.io/vk$_r aytracing_t utorial_K HR/$.

[7] Wikipedia, "Penumbra — Wikipedia, the free encyclopedia." Website available at https://es.wikipedia.org/wiki/Penumbra.

[8] T. K. G. Inc, "Vulkan sdk." Website available at https://www.khronos.org/vulkan/.

[9] C. Giessen, "Vk-bootstrap." https://github.com/charles-lunarg/vk-bootstrap.

[10] G. Sellers, *Vulkan Programming Guide.* Addison-Wesley, 2017.

[11] P. Singh, *Learning Vulkan.* Pack Publishing, 2016.

[12] T. K. G. Inc, "Ray tracing in vulkan." Website available at https://www.khronos.org/blog/ray-tracing-in-vulkan.

[13] C. Barré-Brisebois, H. Halén, G. Wihlidal, A. Lauritzen, J. Bekkers, T. Stachowiak, and J. Andersson, "Hybrid rendering for real-time ray tracing," 2019.

[14] T. K. G. Inc, "Gltf tutorial." Website available at https://github.com/KhronosGroup/glTF-Tutorials/blob/master/gltfTutorial/README.md.

[15] "Vulkan 1.2 specification." Website available at https://www.khronos.org/registry/vulkan/specs/1.2/pdf/vkspec.pdf

[16] W. Usher, "The ray-tracing shader binding table three ways." Website available at https://www.willusher.io/graphics/2019/11/20/the-sbt-three-ways.

[17] T. K. G. Inc, "vkcmdtracerayskhr(3) manual page." Website available at https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/vkCmdTraceRaysKHR.html.

[18] T. K. G. Inc, "Mix glsl references." Website available at https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/mix.xhtml.

[19] K. Xu, "Temporal antialiasing in uncharted 4." A lecture from Advances in Real-Time Rendering course, SIGGRAPH 2016 http://advances.realtimerendering.com/s2016/.

[20] NVIDIA, "Nvidia nsight systems." Website available at https://developer.nvidia.com/nsight-systems.