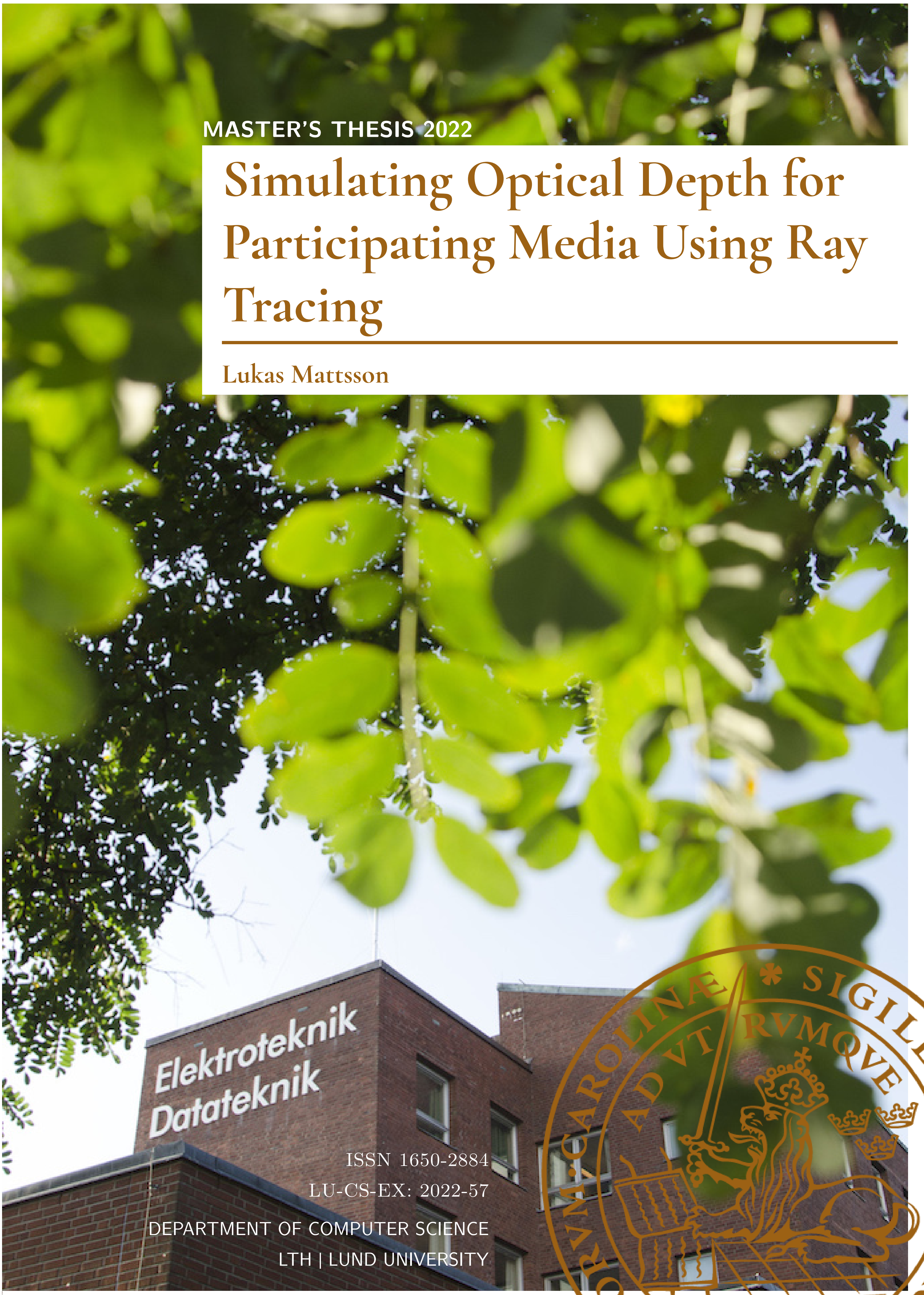


MASTER'S THESIS 2022

# Simulating Optical Depth for Participating Media Using Ray Tracing

Lukas Mattsson

ISSN 1650-2884
LU-CS-EX: 2022-57

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-57

# Simulating Optical Depth for Participating Media Using Ray Tracing

Simulering av optiskt djup för deltagande media med strålspårning

**Lukas Mattsson**

# Simulating Optical Depth for Participating Media Using Ray Tracing

**(A $\LaTeX$ class)**

Lukas Mattsson
lu7511ma-s@student.lu.se

October 20, 2022

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Michael Doggett, `michael.doggett@cs.lth.se`
Calle Lejdfors, `calle@tencent.com`

Examiner: Flavius Gruian, `flavius.gruian@cs.lth.se`

## Abstract

Recent developments in graphics hardware have made it possible to use ray tracing for rendering real-time applications via hardware acceleration. In this thesis we explore how ray tracing can be used to improve the appearance of participating media, such as smoke, while still rendering quickly enough to be used in real-time applications, like video games. In this thesis we focus on a particular aspect of participating media, optical depth. The participating media is represented as a collection of particles which are ray traced through in order to get the distances through participating media that light has to travel through to reach each particle. These distances are then used to simulate optical depth. The final renderer contains parameters which can be adjusted to improve the appearance of the participating media at the cost of performance and vice versa. An example of one such parameter is the number of particles which make up the representation of the participating media. To judge the usefulness of the technique, the renderer's parameters are varied, and image and performance measurements are analyzed. We find that the images produced by the renderer is of good quality and performs well enough to use in real-time applications with average frame times of between 1.70ms and 3.78ms and optical depth calculations specifically taking between around 0.08ms and 1.48ms on average on an Nvidia RTX 2080 Ti GPU.

**Keywords**: ray tracing, real-time, computer graphics, participating media, optical depth

# Acknowledgements

I would like to thank my supervisors Michael Doggett and Calle Lejdfors for all of their help during this project.

I would also like to thank Sharkmob for providing me with the six point lighting textures used for the smoke.

Finally, I would like to thank my friends and family for their support during this project.

# Contents

# Chapter 1
# Introduction

In order to create a highly immersive modern video game it is essential for it to contain visually appealing natural environments that can be rendered in real time at a high frame rate. An important aspect of these natural environments is participating media[1], such as smoke, clouds and fog. Unfortunately, the real world physics of participating media is complex which means that approximating it on a GPU is no easy task. For example, proper distance estimation in participating media is very difficult to accurately perform in a rasterization pipeline, which is the most commonly used pipeline when rendering real time applications. Computing the proper light intensity when a three dimensional object is inside the participating media is also a challenge. This is because a rasterization pipeline does not provide information on where objects are relative to each other when rendering. Ray tracing has for a long time been exclusively used in offline rendering, but recent generations of GPUs have been released with hardware support for ray tracing. This has made it feasible to use ray tracing techniques in real time applications, such as video games. The idea with this project is to use ray tracing to solve the aforementioned problems for rendering participating media.

## 1.1 Project Scope

For this thesis we implement a ray tracer that calculates the distance that light travels through participating media to reach a given point within a medium. The focus in this thesis is on smoke, and while this technique can be applied to any participating media there are some notable differences in how different media behave and how they are normally rendered, which is why it is more suitable to focus on one medium. In participating media in the real world, light gets scattered in all directions as it travels through the media, which is difficult to properly estimate quickly in real time as it can scatter not only once but multiple times at each point. Therefore, to limit the scope of this thesis we only simulate the dampening of light intensity caused by absorption and light being scattered away. The light that is scattered back in which causes an increase in the light intensity is not simulated.

The target is to produce a ray tracer that can render high quality images of smoke while performing well enough to be put in a video game. This means that it should be able to render in addition to everything else in a scene at around 60 frames per second. This leaves only a small number of milliseconds available for smoke rendering techniques. Additionally, the smoke will be represented as a particle system which is what it is commonly represented as in the game industry.

## 1.2 Research Questions

The theory behind this thesis is that ray tracing techniques on modern GPUs should be able to make the task of calculating the distance light has traveled through participating media much faster compared to rasterized rendering. Due to this potential performance increase, it should be possible to make the approximations of participating media more realistic than what is typically used in modern video games while still being able to render in real time. Based on this, the thesis will attempt to answer the following questions:

- To what extent can the visual quality of participating media be improved by simulating optical depth for real time applications using ray tracing hardware?
- What will the render times of this be on modern GPUs?

## 1.3 Contributions

This thesis contributes knowledge in what can be achieved by using ray tracing in real time and what level of performance it can reach using modern graphics hardware. It is also contributes with a new approach to rendering participating media in real time to the field of computer graphics. This knowledge may help the video game industry to render more realistic smoke at good performance in games.

# Chapter 2
# Background

In this chapter we go over the background knowledge required to understand this thesis. We begin by going over some basic concepts used in this thesis on the subject of computer graphics, such as scenes, coordinate systems, textures and physically based rendering. Afterwards, the concept of ray tracing is explained as well as how participating media can be simulated. Thereafter, we go over some technical aspects such as how GPUs work and how the DirectX Raytracing API is used in order to understand the implementation of the renderer described in chapter 4. We conclude the chapter by bringing up related works to this thesis, covering topics related to ray tracing or participating media to show what has and has not been done previously regarding this topic.

## 2.1 Computer Graphics

In order to render an image, there needs to be a virtual scene. A virtual scene consists of 3D objects, light and a camera. These three things provide the information needed to know where objects are and what colors they have, which can be used to calculate what color each pixel in the resulting image should have.

### 2.1.1 Virtual Scenes

All 3D objects are built from triangles, with a higher number of triangles being able to provide more detail to the object. The triangles are created from vertices and indices. Vertices are points with coordinates in 3D space and indices are the numbered indices of the vertices, which are used to determine which vertices make up a triangle. For example, vertices with coordinates (0, 1 ,0), (1, 0, 0) and (0, 0, 1) and indices 0, 1 and 2 in the list of vertices could make up a triangle. Vertices can, in addition to coordinates, also contain other attributes such as the normal vector or the color of that point. The vertex attributes of arbitrary points on a triangle are calculated by interpolating the attributes of the nearest vertices. Light can be represented in many ways, but a simple way to do it is by using a point light as a light source. A point light can be defined as simply as a position and a color. A camera normally has a position and a direction, which is what determines what in the scene will be visible in the image and from what angle.[5] Figure 2.1 shows an example of a virtual scene with a camera, a light source and two 3D objects. The red points represent the vertices which are connected by the black lines to form triangles. These triangles then together form the cube and the ground it is placed on.

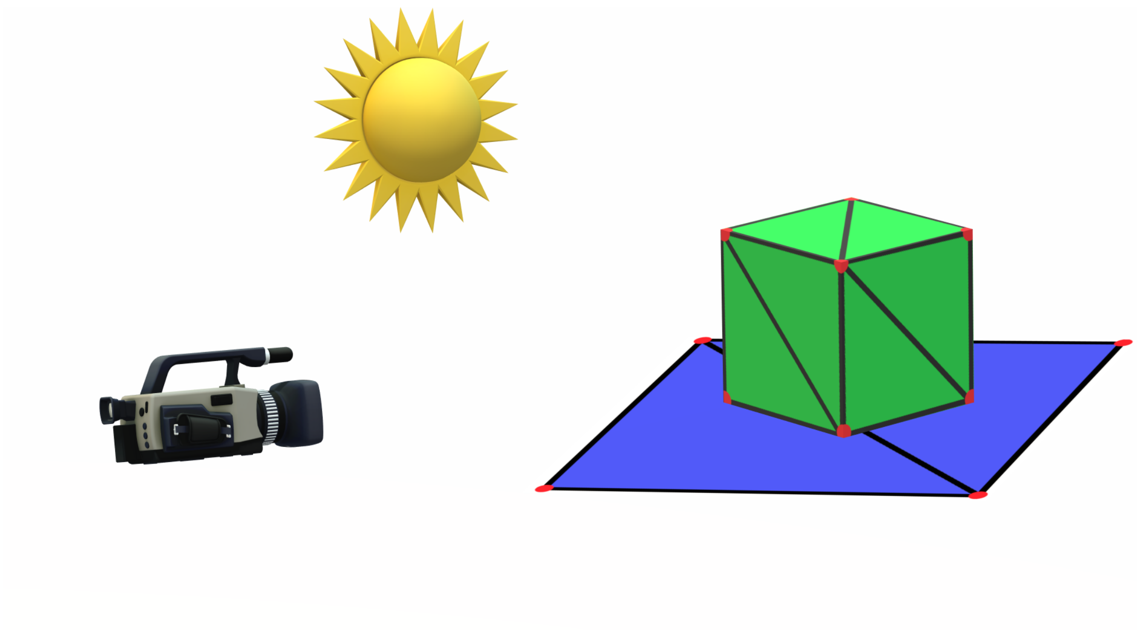


**Figure 2.1:** An example of a virtual scene

### 2.1.2 Coordinate systems

Coordinate systems are an important concept in the world of computer graphics. This is because all coordinates have to be defined in some coordinate system in order to have any meaning. Some common ones are:

- **Tangent Space**, or **Shading Space**, is a system that is unique for every point on an object. The origin is the point on the object and the axes are made up of the normal, tangent and binormal vectors of that point.
- **Object Space** is the system centered around the object, with the origin usually somewhere around the object's centre. Each object has its own object space.
- **World Space** is the system which all other systems are based on. There is only one world space and it is the one that is used when describing everything in the scene in relation to each other.
- **Camera Space** is the system which follows the viewing camera. The origin is the camera's position and its z-axis is the viewing direction.[5]

### 2.1.3 Textures

Textures can be used for many things, but in general it is a function that maps values from one domain to another. A common use case for textures is to give color to objects. This can be done by assigning texture coordinates to vertices. These texture coordinates can then map to pixels on an image, for example, and the colors can then be used for the object's surface. Another example of using textures is for bump mapping, which can be done via tangent space normal mapping.[5] This means storing a normal vector which is in tangent space somewhere, such as in a file, and looking it up with the use of texture coordinates and then replacing the existing normal vector with the one from the texture. This gives the appearance of having added extra details to the object without having to increase the number of triangles.

### 2.1.4 Physically Based Rendering

Rendering can be done in many different styles but a common one is physically based rendering, which uses models from real world physics to base lighting calculations on. Kajiya[6] proposed the rendering equation which describes the behavior of light in the context of computer graphics and is shown in equation 2.1.

$$I(x,x') = g(x,x')\left[\epsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right] \tag{2.1}$$

Where:

- $I(x,x')$ represents the intensity of light from point $x$ to point $x'$
- $g(x,x')$ represents how occluded the points are relative to each other on a scale from 0 to 1.
- $\epsilon(x,x')$ represents the intensity of emitted light from $x'$ to $x$
- $\rho(x,x',x'')$ represents the intensity of light scattered at point $x'$ from point $x''$ in the direction of $x$ and is dependant on the type of material involved
- $S$ is the union of all surfaces[6]

This equation is infinitely recursive since $I(x,x')$ is defined by its own integral, which means that it cannot be used directly when rendering. Instead, approximations are used, such as the Phong model which simplifies the rendering equation. The Phong model[11] essentially divides the light intensity $I$ into three terms, namely: ambient, diffuse and specular light, where materials have unique ambient, diffuse and specular coefficients and light sources emit light in ambient, diffuse and specular colors. The Phong model is what is used to calculate the colors in the final renderer made for this thesis.

## 2.2 Ray Tracing

Ray tracing is a rendering technique which is based on following rays of light from light sources to objects to the camera to determine their color and visibility. It is commonly used when rendering photorealistic images due to how easy this technique makes it to simulate more complex visual effects, such as reflections and refractions. The disadvantage with ray tracing is that it involves many calculations which makes it slow.[5]

### 2.2.1 Ray Intersections

In order to use ray tracing to generate an image, ray intersection calculations are needed. These determine which rays hit which objects and in what order the hits occur. A ray $r$ is normally represented in parametric form, which is shown in equation 2.2.

$$r(t) = o + td \tag{2.2}$$

Where $o$ is the ray's origin, $t$ is the length variable and $d$ is the direction vector. The legal range of $t$ is $(0, \infty)$.

Objects can come in many different shapes, but one example is a sphere which can be represented in implicit form as shown in equation2.3.

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - r^2 = 0 \tag{2.3}$$

Where $x_0$, $y_0$ and $z_0$ are the coordinates for the sphere's centre and $x$, $y$ and $z$ are variables. Substituting $x$, $y$ and $z$ for the ray's equation gives equation 2.4.

$$(o_x + td_x - x_0)^2 + (o_y + td_y - y_0)^2 + (o_z + td_z - z_0)^2 - r^2 = 0 \tag{2.4}$$

For a given ray and sphere only $t$ is unknown and if the equation has at least one valid solution then that means that the ray intersects the sphere. The same thing can be done for testing intersections with a plane, which can be represented in implicit form and is shown in equation 2.5.

$$ax + by + cz + d = 0 \tag{2.5}$$

Where $a$, $b$, $c$ and $d$ are the parameters of the plane and $x$, $y$ and $z$ are variables.[5] Expanding on this reasoning it is possible to determine how to test for intersections with triangles, which is the most common form of geometry in computer graphics. After determining that the ray intersects the plane of the triangle, the intersection point must be tested to see if it is inside the triangle or not. This can be done by using the triangle's edge functions, which are shown in equation 2.6.

$$e_i(x, y) = -(p_y^{i2} - p_y^{i1})(x - p_x^{i1}) + (p_x^{i2} - p_x^{i1})(y - p_y^{i1})e_i(x, y) = a_i x + b_i y + c_i \tag{2.6}$$

Where $i$ refers to which edge [0..2] of the triangle and $p^{i1}$ and $p^{i2}$ are the start and end points of edge $i$. The normal vector of edge $i$ is $(a_i, b_i)$. Whether the point is inside an edge or not is determined by the following tiebreaker rule:

1. if $e(x, y) > 0$ return true
2. if $e(x, y) < 0$ return false
3. if $a > 0$ return true
4. if $a < 0$ return false

5. if $b > 0$ return true

6. return false

If this rule returns true for all edges of the triangle then the intersection point is inside the triangle.[2]

### 2.2.2 Whitted Ray Tracing

An issue with simply following rays of light from light sources is that they can go in any direction and can bounce in any direction, including towards areas which are not visible to the camera. Doing the calculations for all these rays is a waste of computation time and it is impossible to tell in advance which rays will contribute to the image or not. A solution for this is whitted ray tracing, or backwards ray tracing, which instead reverses the process and this technique is what is used in our ray tracer. This is done by starting at the camera and shooting rays from there in all directions that are visible to the camera. These can then hit objects in the scene, where new rays can be traced towards light sources or they can go in other directions in order to do reflections or refractions, for example. Rays can continue to be traced recursively like this in order to provide a more accurate final image at the cost of more computational power.[7] Figure 2.2 shows an illustration of how whitted ray tracing can be used to render an image.

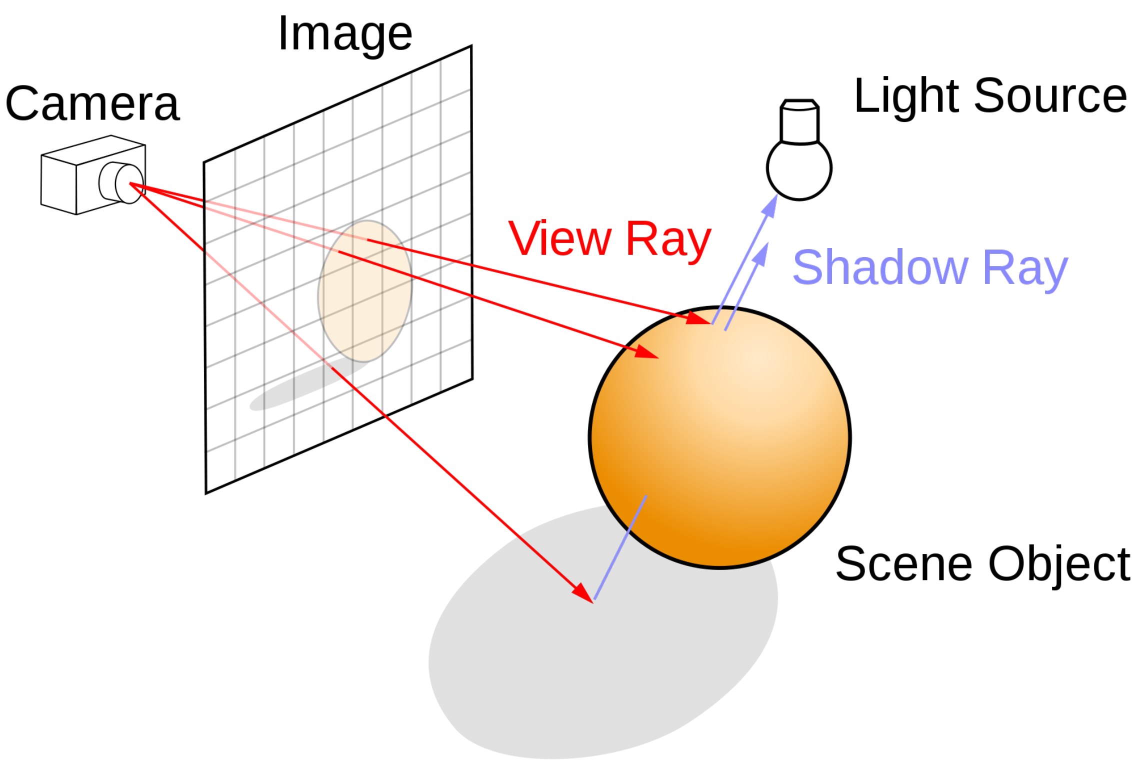


**Figure 2.2:** An illustration of whitted ray tracing

## 2.3 Participating Media

Participating media is a term used to describe volumes that are represented by particles in computer graphics.[1] These are usually substances such as smoke, clouds and fog, for example. For an in-depth description of participating media, the reader is referred to the fourth edition of *Real-Time Rendering* by Akenine-Möller et al.[1] Participating media is difficult to accurately simulate in computer graphics due to the way it interacts with light. As light travels through participating media it gets dampened as it gets absorbed and also scattered around in all directions. Additionally, light that has scattered away from other paths could potentially also have scattered into the current light path. Therefore, when trying to calculate the light at a particular point in participating media one must find out how much of the direct light from the light source is left at that point and also how much light there is that has scattered around one or more times before reaching the point of interest. It is common to describe the total outgoing light as a sum of the direct light and the in-scattered light[1] and this is described by

$$L_{out}(c, -v) = L_d(c, -v) + L_m(c, -v) \tag{2.7}$$

Where $L_{out}$ describes the outgoing radiance, $L_d$ is the reduced direct light and $L_m$ is the media radiance which consists of the in-scattered light that reaches the camera at point $c$ along the negative view direction $-v$. The $L_m$ term is computed by integrating all light contributions along the way from the entry point of the medium $p$ to the camera at point $c$

$$L_m(c, -v) = \int_{t=0}^{\|p-c\|} T(c, c - vt)\sigma_s L_s(c - vt, v)dt \tag{2.8}$$

Where $T$ is the transmittance, $\sigma_s$ is the scattering coefficient of the medium and $L_s$ is the scattered light. The $L_d$ term is dependant on the incoming light $L_{in}$ and the transmittance $T$[1]

$$L_d(c, -v) = T(c, p)L_{in}(p, v) \tag{2.9}$$

In this thesis the focus is on simulating the $L_d$ term from this model and we disregard the $L_m$ term. According to the Beer-Lambert law, the transmittance T from point $x_a$ to point $x_b$ is related to the optical depth $\tau$ as

$$T(x_a, x_b) = e^{-\tau(x_a, x_b)} \tag{2.10}$$

Where the optical depth itself is defined as

$$\tau(x_a, x_b) = \int_{x=x_a}^{x_b} \sigma_t(x)\|dx\| \tag{2.11}$$

Where $\sigma_t$ is the extinction coefficient.[1] In our simulation, we set the extinction coefficient to be constant throughout the participating media, since it is a property of the medium and not part of what is being simulated in this thesis. So we simplify this definition by defining the optical depth $\tau$ from point $x_a$ to point $x_b$ to be as

$$\tau(x_a, x_b) = l(x_a, x_b)\sigma_t \tag{2.12}$$

Where $l(x_a, x_b)$ is the distance from point $x_a$ to point $x_b$, which we refer to as the optical path length. These definitions are what is being referred to when the terms optical depth and optical path length are used going forward in this report.

## 2.4 Technical Aspects

As the previous sections cover the theoretical aspects of the topic of this thesis, this section covers the technical aspects and describes how these concepts are applied to computer hardware and software, which we use to implement our renderer. We begin by briefly describing how GPUs work and what they can do. Then we cover the DirectX Raytracing API and how it works and is used, since this is the API used in this project.

### 2.4.1 GPUs

GPUs are massively parallel devices using single instruction, multiple data (SIMD) processing. They can run thousands of computations at the same time and have little focus on single-thread performance, by allocating more space on the chips for more cores rather than cache, for example.[5] This is important due to the massive amount of pixels on modern screens. A full HD display is 1920 pixels wide and 1080 pixels in height which is more than 2 million pixels in total. This means that, in real time applications, GPUs have to calculate the color of all these pixels in milliseconds.

Until recently, ray tracing was exclusively used in offline rendering because of how computationally slow it is. But recent generations of GPUs have specialized hardware for ray tracing which enables it to run in real time.[5]

### 2.4.2 DirectX Raytracing

DirectX is a collection of APIs related to graphics and video developed by Microsoft and one of these is DirectX Raytracing (DXR). The DXR API is what is we used to implement our renderer and it consists of two main parts, namely: the CPU host code and the GPU device code. The CPU host code is written in C++ and is used to manage memory, load assets, create scenes, build data structures and give commands to the GPU. The GPU device code, which is commonly known as shader code, is written in a C-like language called High Level Shading Language (HLSL). Shader code is designed to be written by the programmers like serial code, when it is in fact parallel code. It is in the shaders that the programmer writes the code to determine pixel colors, for example. There are a number of different types of shaders and the pipelines for the ones we used in this thesis are shown in a diagram in figure 2.3 and described in further detail below.[13] Our implementation of these is described in chapter 4.

#### Ray Generation Shader

The ray generation shader is executed when the GPU is given the DispatchRays() command. The number of threads launched in the GPU is indicated by a parameter in the DispatchRays() command. In general, this shader is where the programmer define the rays and a payload, call TraceRay() and then write the output colors of the pixels. TraceRay() tells the GPU to search for intersections with the given ray and then calls other shaders depending on the results of the search. The payload is used by the shaders called by TraceRay() to write their output in. A simple payload can, for example, be a struct which contains 4 floats for the red, green, blue and alpha components of the resulting color from a shader for a pixel.[13]

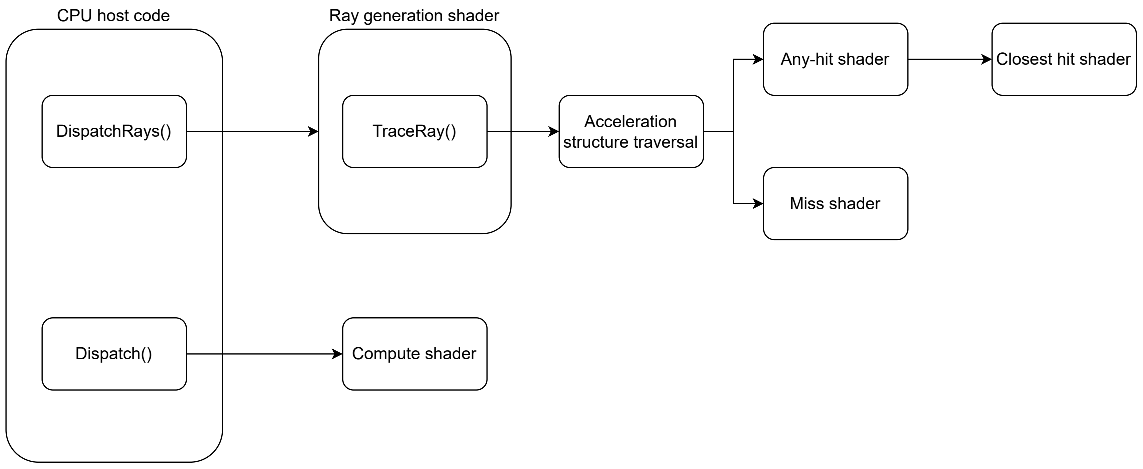


**Figure 2.3:** A diagram of the rendering pipelines

## Miss Shader

The miss shader is executed if the given ray for a TraceRay() call does not find any intersections with any geometry. The miss shader is allowed to modify the payload, which can be used to set a background color, for example. The miss shader may also define new rays and call TraceRay() again, but does not have access to any intersection data since there is no intersection.[13]

## Any-hit Shader

The any-hit shader is executed if an intersection with the given ray and a triangle is found in a TraceRay() call. The any-hit shader is allowed to modify the payload and has access to the intersection data of the hit triangle. The intersection data is user defined and can for example contain the barycentric coordinates of the intersection point on the triangle. The any-hit shader may not trace any new rays, however. A common usage for any-hit shaders is for transparent objects, where the colors of the triangles behind the closest one are also required. An important note is that hits are not necessarily going to be found in order by distance from the ray's origin. Rendering transparency properly needs to be done in order so this must be taken into account when writing an any-hit shader. In the any-hit shader the programmer can choose to either ignore a hit or accept it. Ignoring the hit means that the GPU will continue searching for more intersections, while still retaining any changes made to the payload. Accepting the hit means that the GPU will stop searching for more intersections, even if there are more that have not been handled, and then invoke the closest hit shader. It is not obligatory for the programmer to write their own any-hit shader, since there will otherwise be a default one which just accepts the closest hit.[13]

## Closest Hit Shader

The closest hit shader is executed once per ray when the any-hit shader has accepted a hit. The closest hit shader is also allowed to modify the payload and has access to the intersection data of the hit triangle but it is allowed to trace new rays, in contrast to the any-hit shader. Actual shading of triangles is an example of how this shader can be used, by calculating the colors of the hit triangle and updating the payload with this information. Since it is also possible to trace new rays from this shader it is simple to render shadows or reflections, for example, by tracing a ray in the direction of the light source or in the reflection direction and searching for intersections. It is not obligatory for the programmer to write their own closest hit shader either.[13]

### Compute Shader

The compute shader is a bit different from the other shaders, as it is not exclusively used for ray tracing nor is it a part of the same pipeline as the others. The compute shader is invoked on its own by a Dispatch() command from the CPU, which specifies the number of threads that should be launched. As per its namesake, compute shaders are only used for computing things, usually computations that can be parallelized. The idea is to use them when a program needs to do many calculations, that are not directly related to the rendering, each frame that better suit the GPU's architecture than the CPU's, and so the work is offloaded to the GPU. An example of this would be if a program needs to sort a large number of arrays each frame.[8]

### Acceleration Structures

Testing for intersections with every ray and every triangle each frame gets very computationally expensive and cannot be done when rendering in real time. Therefore, triangles are put into acceleration structures in DXR, which are a bounding volume hierarchy (BVH). These are tree-like structures that can be thought of like putting triangles inside boxes which are inside bigger boxes. This way, ray intersections can be tested first with the biggest box in the top of the tree and if the ray does intersect it then intersections with the smaller boxes within are tested, and so on until the triangles are reached and intersections with them are tested. If a ray does not intersect with a box, then all the computations for testing for intersections with all potentially hundreds or thousands of triangles within are skipped.[13]

In DXR there are bottom level acceleration structures (BLAS) and top level acceleration structures (TLAS). BLAS contain the actual triangles themselves and are slow to build. TLAS contain object instances which point to BLAS and have their own transform matrix and they are fast to build. As an example, if a scene was to have two identical triangles next to each other, one would need to create one triangle, put it in a BLAS, create two object instances which point to the BLAS and have slightly differing transform matrices and then put these in a TLAS. In order to animate the triangles, such as moving them, new instances need to be made which contain the new positions in the transform matrices and then rebuild the TLAS every frame.[13]

### Buffers

In a renderer there is lots of data that needs to accessed in some way by both the CPU and the GPU and this is handled by using buffers. Buffers are parts of memory where data can transfer in and out at different rates at the same time, like when streaming a video it can be played back at the same time the application is downloading it. The three types of buffers that are included in DirectX and are used in this project are Constant Buffer Views (CBVs), Shader Resource Views (SRVs) and Unordered Access Views (UAVs). CBVs are generally used for placing user defined structs into which have constant values throughout a frame, such as the camera's position. SRVs are used for resources needed specifically for the shader such as vertex and index data. UAVs are for resources that are written to, and potentially also read from, by the GPU and an example of this is the output texture where a ray tracer can write the calculated colors of all of the pixels to.[10]

## 2.5 Related Works

There are a lot of previous works about rendering participating media and ray tracing. One such work is *Volumetric Particle Shadows* by Green. In that paper Green presents a half-angle shadowing technique which adds volumetric shadowing to smoke. However, the method proposed in that paper uses rasterization instead of ray tracing to render the frames. Additionally, while the smoke does render in real time, it is still not performant enough to put into something like a video game where the smoke would need to be rendered in addition to everything else in the scene.[4] In this thesis we implement a renderer based on ray tracing instead which is performant enough, shown in chapter 5.

Another work is *Real-time Smoke Rendering Using Compensated Ray Marching* by Zhou et al. In that article the authors present a way to render smoke using ray marching. They achieve good images by calculating the light absorption, single scattering and multiple scattering. However, similarily to the previously mentioned paper, their method of rendering smoke does run in real time but still does not render fast enough to be able to use it in a video game.[14] We ignore the light scattering in our thesis, reducing the accuracy of the participating media comparatively, but with the benefit of having very low render times, shown in chapter 5.

Additionally, Shirley describes a simple approach to rendering participating media using ray tracing in his book *Ray Tracing: The Next Week*. That method focuses on the scattering of light inside participating media and not on the light absorption. That approach is also only used with offline rendering, which makes it unsuitable to use in real time applications.[12] In this thesis the target use of the renderer is real time applications and we do focus on the light absorption.

Brüll and Grosch present a method for rendering transparency using ray tracing in their article *Multi-Layer Alpha Tracing*. That method uses real time ray tracing with DXR to render transparent objects and smoke. Transparency is an important feature of participating media, making the article relevant to this thesis, however it does not calculate any absorption or scattering of light,[3] which we do in this thesis.

# Chapter 3
# Approach

In this chapter we will go over the theory behind the implementation and how it works. This involves describing the ray tracing algorithm for simulating optical depth and what is important to consider when using it. In order to highlight the details that simulating optical depth adds to smoke, we have created different render modes. We will also go over what these different render modes are and how the results will be measured in this chapter.

## 3.1 Algorithm

As previously mentioned, the smoke is represented as a particle system, where each particle is represented as a sphere. In this particle system, particles overlap to varying degrees to form continuous clouds of smoke. The fact that some points in space are covered by more than one particle is not taken into consideration as the particle system is only used to determine if a point is inside smoke or not. The value being calculated by the algorithm is the distance in some length unit that light travels through participating media in order to reach a given particle, the optical path length. For each particle, a ray will be traced from its centre in the direction of the light source. A counter will be used while marching along the ray, which will keep track of when the ray enters and exits participating media. The idea is that if the counter is incremented by one each time the ray enters a particle and decremented by one each time the ray leaves a particle, the counter will be zero when the ray is outside participating media. Figure 3.1 shows an example of a ray being traced from the centre of a particle towards the light source. The optical path length to be calculated is then the length of the ray minus the lengths $X$ and $Y$ in this case.

A challenge with this algorithm is choosing the start value for the counter. Since the rays start at each particle's centre, most of the time it would work to set the start value to one. The problem is that since particles can overlap, any number of particles could theoretically overlap a particle's centre point. Therefore, one would need to know how many particles cover the origin of the ray before starting to calculate the distance. The solution to this problem that we use is to first go through all of the entry and exit points of the particles that the given ray intersects and then sum them up, with plus one for entry and minus one for exit. It is then possible to use the sum multiplied by minus one as the start value for the counter. This can be described in mathematical terms as

$$counter_{start} = -\sum_{n=1}^{N} H(n) = \begin{cases} 1, & \text{if hit } n \text{ is an entry point} \\ -1, & \text{if hit } n \text{ is an exit point} \end{cases} \tag{3.1}$$

Where $N$ is the total number of hits and $H(n)$ is the value of hit number $n$. For example if two particles were to cover a ray's origin, the sum would then be minus two, since there would be two more exit points than entry points in total and therefore by setting the start value to two the counter would be zero when leaving participating media.

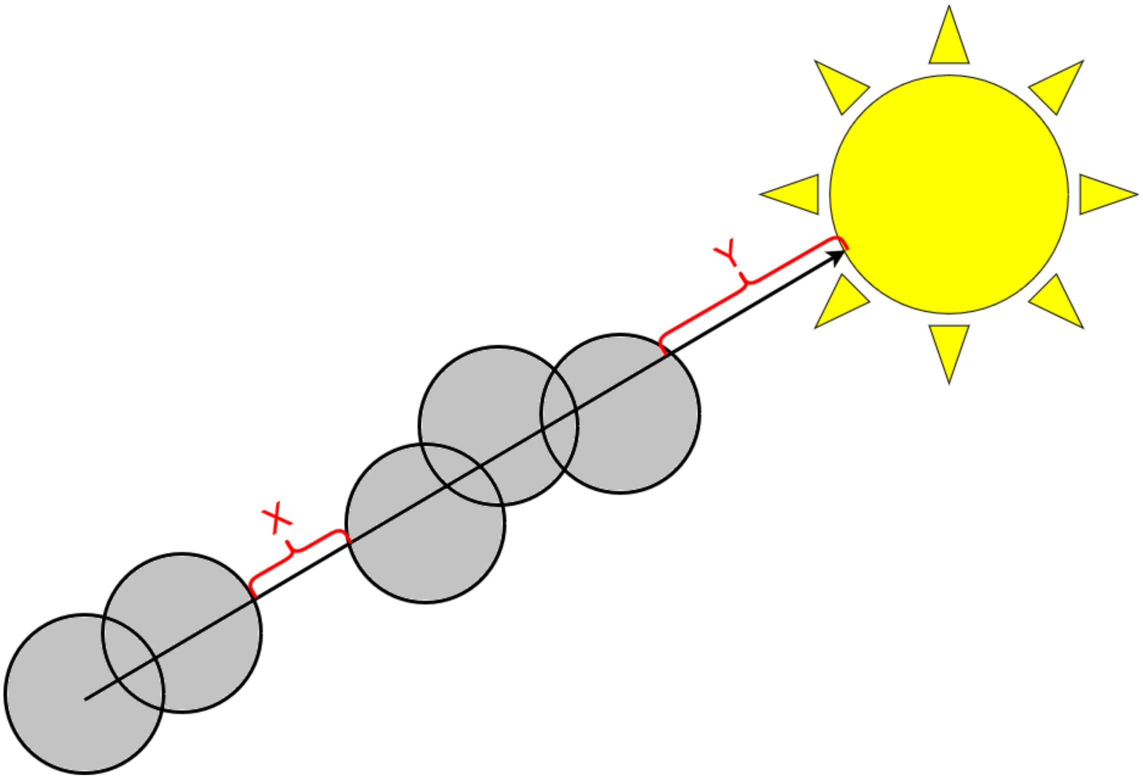


**Figure 3.1:** An example of tracing a ray from a particle to a light source

## 3.2 Render Modes

Below we will go over the different render modes where different aspects of the optical depth calculations are implemented. These include what we call Basic Mode, Simple Shadow Mode and Full Optical Depth Mode. The purpose of these render modes is to render images with and without simulated optical depth in order to show the details it adds.

### 3.2.1 Basic Mode

In Basic Mode, optical depth is not included in lighting calculations and no shadows are rendered. The smoke is shaded with light from the light source and it uses the same textures as the other modes. The purpose of this mode is to demonstrate what exactly the images lack in terms of detail when optical depth is not being taken into consideration when rendering smoke. Furthermore, it will also show what details of the smoke are not due to calculating the optical depth and are just artistic additions, which is an important distinction to make. For example, we use textures when drawing the particle system to make it look more like smoke, but these textures do not have anything to do with optical depth.

### 3.2.2 Simple Shadow Mode

In Simple Shadow Mode, optical depth is included in lighting calculations for the smoke and shadows are rendered. The smoke is shaded with light from the light source and it also uses the same textures as the other modes, but the amount of light received by a particle is affected by its optical depth. However, the shadows are rendered using a very simple algorithm, where a point will receive no direct light if anything is in between it and the light source and otherwise it receives the full light intensity from the light source. The purpose of this mode is firstly to demonstrate what the smoke looks like when optical depth is calculated and used when shading, which can be compared with Basic Mode in order to see the difference that it makes. Secondly, this mode also shows what some very simple shadows can look like when few parameters are factored into the calculations.

### 3.2.3 Full Optical Depth Mode

In Full Optical Depth Mode, optical depth is included in all lighting calculations, including when rendering shadows. The smoke is shaded with light from the light source and also uses the same textures as the other modes. The amount of light that a particle receives is affected by its optical depth, similarly to Simple Shadow Mode. However, the shadows rendered in this mode are more nuanced than the ones in Simple Shadow Mode. We came up with a shadow algorithm that will, for a given point, check if anything is in between it and the light source, and if there is then the closest occluding object's optical depth will be looked up and used to determine how much of the incoming light should be reduced for the point. The purpose of this mode is to demonstrate the full effect of using optical depth when rendering smoke and its shadows. This mode can be compared with Basic Mode to see the impact optical depth has on smoke and with Simple Shadow Mode to see the impact optical depth has on shadows.

## 3.3 Evaluation Strategy

Below we will go over the evaluation strategy used for this work. The evaluation strategy includes comparing images rendered in the different modes and performance measurements which shows render times.

### 3.3.1 Image Comparisons

A number of images are rendered using the different modes and from different angles. These images are then to be compared against each other to see the effect that factoring in optical depth in lighting calculations have on smoke and shadows from smoke. The comparisons are made by the human eye, which means that they will be somewhat subjective. This is due to there being significant limitations with doing more objective comparisons such as RSME measurements for this project and these limitations are discussed in detail in section 6.1 of this report. There are a number of parameters involved in the Full Optical Depth Mode rendering which can be adjusted in order to increase performance at the cost of image quality or vice versa. These parameters are presented in chapter 4 and some image comparisons are also made for different values of these parameters.

### 3.3.2 Performance Measurements

The performance of the different renders are measured in terms of render time and are then compared to each other. The performance when using different values of the parameters in the Full Optical Depth Mode are compared in order to see how well they scale. The render times are broken down into the different components of the render loop in order to show what parts are affected the most by the different changes in values of the parameters as well as simply providing a general view of what components have the biggest impact on the overall frame time. Unfortunately, there are also significant limitations to doing performance comparisons with other renders of smoke made by others which are also discussed further in section 6.1 of this report.

# Chapter 4

# Implementation

In this chapter we go over how the renderer was implemented. Figure 4.1 shows a diagram of the renderer that was implemented for this project. The diagram shows the application setup to the left, the main render loop in the middle and the shaders invoked by the render loop to the right. The project's application code is written in C++ and is using the DirectX 12 API and the shader code is written in HLSL.[9]

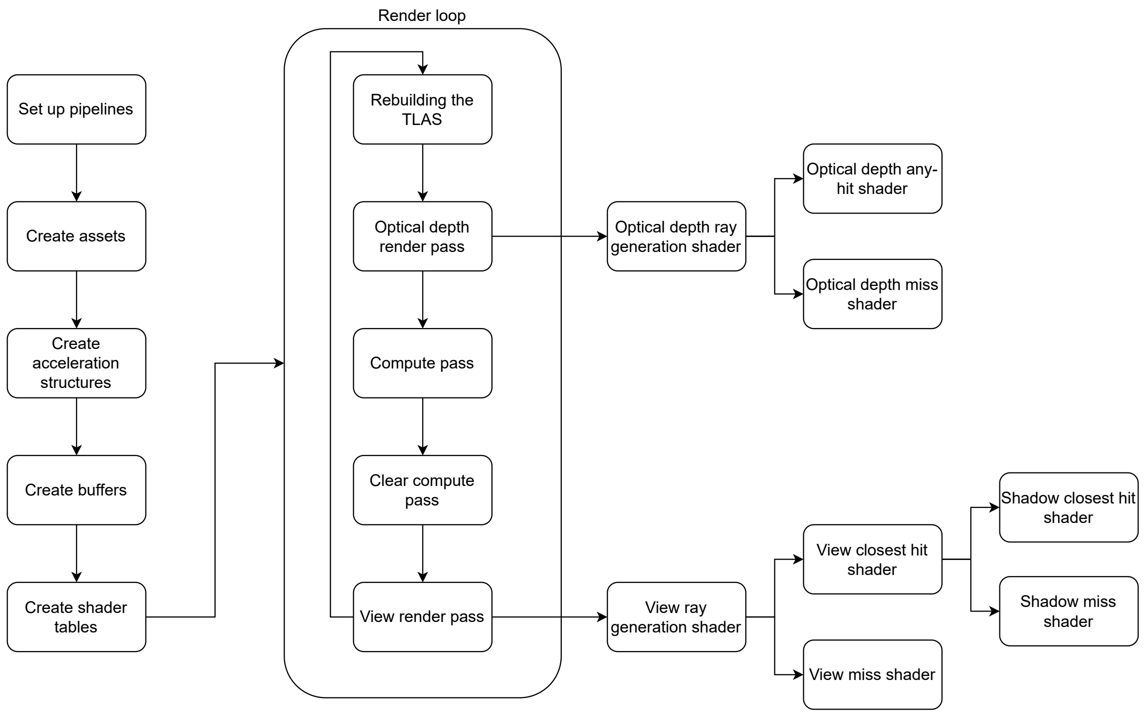


**Figure 4.1:** A diagram of the renderer

## 4.1 The Scene

The scene consists of two tubes of smoke, where one tube is slightly above and perpendicular to the other tube. It also contains a red quad, which is a square made up of two triangles, and it acts as a ground plane and can show shadows from the smoke. For the sake of simplicity, we consider the light source to be far enough away to be thought of as practically infinitely far away and it rotates around the z-axis, making the lighting behave similarly to lighting from the sun. The smoke is made up of a large number of particles, ranging in the hundreds, where each particle is made up of a quad and an icosahedron which are used at different points during rendering. Since the particles are made up of triangles we need to approximate the spheres which is why we use icosahedrons.

### 4.1.1 Particle System

The particle system is built by randomly sampling positions from a cuboid that covers a whole tube and discarding the samples, based on Pythagoras theorem, where the $x$ and $y$ coordinates squared and then added are larger than the tube radius squared. Every other sample that is not discarded is moved a little in the positive y-axis direction and the rest of the samples that were not discarded are moved a little in the negative y-axis direction and then rotated 90 degrees around the y-axis. Each particle can also be assigned motion vectors, movement speeds, rotation speeds and scale speeds for making scenes where the smoke behaves more naturally.

## 4.2 The Core Renderer

In this section we go over the stages of the application setup, which are shown in the first five boxes in figure 4.1. These stages are setting up the pipelines, creating the assets, defining the acceleration structures, allocating the buffers and creating the shader tables.

### 4.2.1 Pipelines

The first part of the render process was setting up the pipelines and for this implementation we chose to use one ray tracing pipeline and two compute pipelines. For the ray tracing pipeline, this included defining the shaders, inserting the shaders' bytecodes and defining the hit groups. The hit groups are used to organize the different hit shaders and defining which closest hit and any-hit shaders are paired together. Additionally, it was also here that the shader and pipeline configurations were set, such as the maximum payload size and the maximum trace recursion depth and the root signatures, which basically define where data can be accessed from. Compute shader pipelines are much simpler and only require a root signature and the shader bytecode.[9]

### 4.2.2 Assets

For this scene there are only three different geometries used, two quads and an icosahedron which is used as an approximation for a sphere. One of the quads and the icosahedron are used to represent the particles and the other quad is used as a ground plane. Furthermore, two textures are also used which together contain six point lighting information for 128 frames and are used to shade each particle.

#### Six point lighting textures

Six point lighting textures are made from offline renders of, in this case, smoke that has been lit up from six different directions: up, down, left, right, front and back. Six images are rendered in black and white with the light coming from a different direction in each image. The grayscale values from these images are then stored in red, green and blue channels from directions right, top and front respectively in one texture, which is referred to as the positive texture, and left, bottom and back in the other, which is referred to as the negative texture. Using a quad which always faces the camera to draw the particle and knowing how much this quad has rotated from its starting position, gives which color channels should be used from the textures with the incoming light when shading the quad. How this is implemented in this renderer is described in section 4.3.5. The textures essentially contain precomputed lighting influence values from all directions which are used to make the geometry look like the smoke while being able to take into account the direction of the incoming light and the viewing direction. Figure 4.2 show the positive and negative textures of the six point lighting for a single frame.

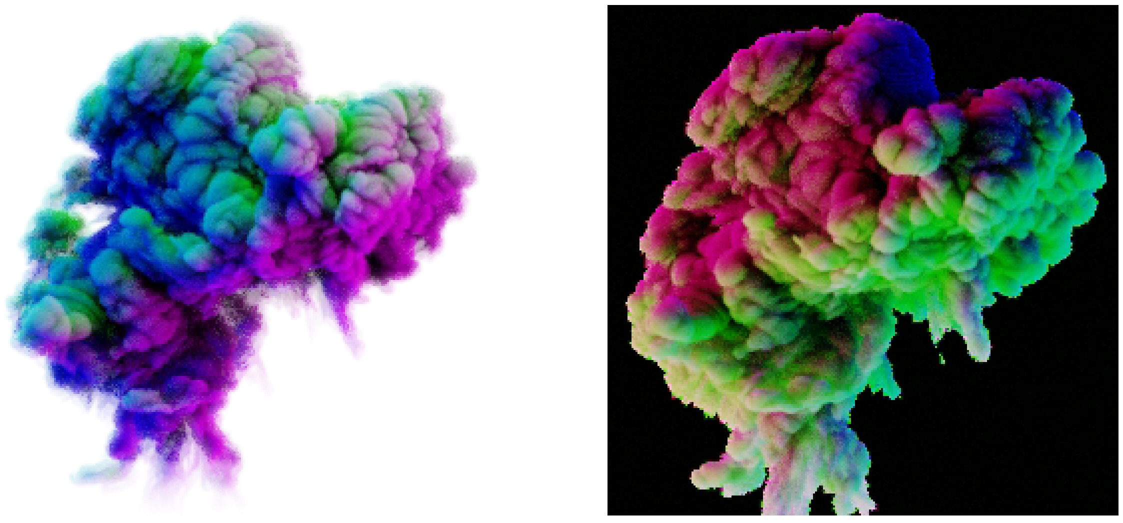

**(a)** The positive texture **(b)** The negative texture

**Figure 4.2:** The two images show a single frame of the six point lighting texture used

### 4.2.3 Acceleration Structures

For this scene we are using three BLAS, one for the isocahedron, one for the particle quad and one for the ground plane. For these three geometries we use the D3D12 ray tracing geometry flag *NO_DUPLICATE_ANYHIT_INVOCATION*. If this flag is not used it would be possible for the any-hit shader to be invoked more than once for the same intersection which is not desired when calculating the distance through smoke in this application. For the acceleration structure builds we set the D3D12 ray tracing acceleration structure build flag *PREFER_FAST_TRACE* to specify that we want high quality acceleration structures that are as efficient as possible. These acceleration structures come at the cost of build time and are used because the tracing is what takes the most amount of time in the rendering which is made clear in chapter 5. We also create one TLAS with many instances, where each particle has one instance for its quad and one for its isocahedron and then there is one instance for the ground plane. The particle isocahedrons are assigned 1 as their instance mask, the particle quads are assigned 2 and the ground plane is assigned 3. Instance masks are used in the ray tracer when certain geometries are not of interest for some rays in order to ignore those intersections. The specific numbers chosen are not important, only that these three are unique. All instances are also assigned unique IDs which can be looked up in the hit shaders and used to handle different instances in different ways. After everything is defined we send the command to the GPU to build the acceleration structures.

### 4.2.4 Buffers

The renderer creates a number of different types of buffers that are needed in the rendering process. Figure 4.3 shows a diagram of the buffers used in the renderer. The diagram shows the eleven total buffers used and which of the three types of buffers they are. The diagram shows the names of the buffers and then the type of data contained within in parentheses. The first of the three CBVs is for storing camera information, such as the camera position and the up, left and forward vectors expressed in world space, and light information, such as the light direction and color. We call this data *RayGenData* and it is used in the drawing part of the render. The second CBV is for storing all of the particles' positions and their IDs and is used in the optical depth part of the render and we call this data *ParticlePositionData*. Finally, the third CBV is used for storing other diverse information that is constant throughout a frame, such as which frame of the lighting textures should be used and how much the particles have rotated from their starting position. These values in particular are used in the drawing part of the render and we call this data *ValuesData*.

Furthermore, we also use four SRVs and the first of these is the vertex buffer. The vertex buffer store vertex information as structs which contain the vertex position, normal, texture coordinate and a color. The second SRV is the index buffer which simply stores the index data as an array of unsigned shorts. Then, the third and fourth buffers are used for the two lighting textures which are two dimensional and contain the respective red, green, blue and alpha components of each texture.

Finally, we also use three UAVs and the first of these is the output texture. This is a two dimensional read and write texture that contains a red, blue, green and alpha component and it is used to output the final calculated colors for each pixel to. Then, the second UAV is a one dimensional read and write buffer that is used to store the calculated optical path lengths for each particle. This is stored as an array of floats with the size being the same as the total number of particles. The last buffer is also a one dimensional read and write buffer containing a long array of floats, but contains information on the hits found from the rays that originate from the particles' centres. The hit information contained is the $t$ value of the ray at the hit point and a one if the ray hit a front face or a minus one if the ray hit a back face. The data here has to be handled manually where every other entry in the array is a $t$ value and the rest are hit values making the array size twice as large as the total number of hits stored. This buffer is used to calculate the optical path length which gets stored into the previously mentioned UAV.

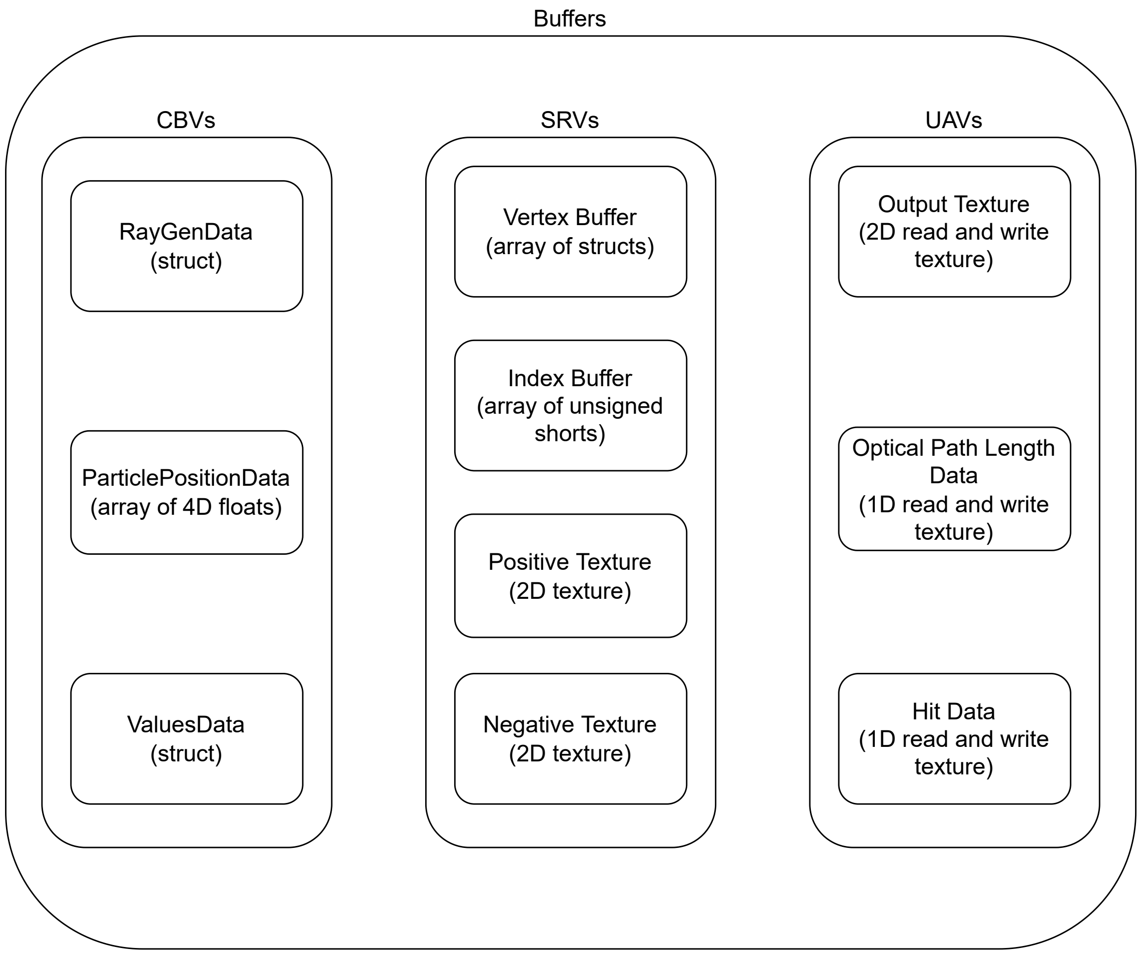

Buffers
CBVs
SRVs
UAVs
RayGenData (struct)
ParticlePositionData (array of 4D floats)
ValuesData (struct)
Vertex Buffer (array of structs)
Index Buffer (array of unsigned shorts)
Positive Texture (2D texture)
Negative Texture (2D texture)
Output Texture (2D read and write texture)
Optical Path Length Data (1D read and write texture)
Hit Data (1D read and write texture)

**Figure 4.3:** Diagram of the buffers used

### 4.2.5 Shader Tables

In this renderer there are two shader tables set up that point to the various shaders used. The first of these shader tables is used for the drawing render pass and contains references to a ray generation shader, two miss shaders and two closest hit shaders. The extra miss shader and closest hit shader are used for the shadow rays. The other shader table is used for the optical depth render pass and contains references to a ray generation shader, a miss shader and an any-hit shader.

## 4.3 The Render Loop

The render loop refers to the while-loop executed after the application setup shown in figure 4.1 and it contains the code that is executed every time a new frame is being rendered. This section will cover the most important parts contained within the render loop which are rebuilding the TLAS, the optical depth render pass, the compute pass, the clear compute pass and the view render pass. In addition to the major components of the render loop described below, some other values are also updated each frame such as the camera's position and direction, the frame statistics and the current texture frame.

### 4.3.1 Rebuilding The TLAS

The TLAS needs to be rebuilt every frame in order for objects to be able to move, rotate and scale. In order to properly draw the particle quads they need to be billboarded, which means that they always need to face the camera regardless of the angle it is viewing from. This means that every frame the particle quads' rotations need to be recalculated based on the camera's position and direction vectors. This is done by creating a matrix from the direction vectors of the camera, inverting it and then multiplying it with the translation matrix containing the current particle's position. Using this transform matrix and the x-, y- and z-axis vectors we can also get the rotation in radians to put into the *ValuesData* CBV. The particle spheres only need to be moved to their specified position and do not need to be rotated. The ground plane quad does not need to be moved or rotated in the TLAS as its positioning and rotation is already done at the vertex level and so it only uses the identity matrix as its transform.

### 4.3.2 Optical Depth Render Pass

The first step in calculating the optical path lengths is to trace rays from each particle towards the light source and then store the ray hits in a UAV. The optical depth render pass consists of a ray generation shader, an any-hit shader and a miss shader. The payload used in this render pass is a struct that contains an integer. The triangle intersection attributes specified are the built in default type which contain the barycentric coordinates of the triangle at the intersection point, but these are not used.

The ray generation shader begins by calling DispatchRaysIndex() to get its thread ID and then uses this as the index to look up its particle's position and ID in the *ParticlePositionData* CBV. Afterwards, the ray to be traced is defined with the particle position as the ray's origin and the light direction from the *RayGenData* CBV as the direction for the ray. The ray's TMin value is set to the smallest possible value of $10^{-7}$ and the ray's TMax value is set to the largest possible value of $10^{38}$ to minimize the potential amount of intersections missed. Next, the ray payload is defined and the integer in the payload is initialized to zero and it is used to count the number of hits. Finally, the TraceRay() call is performed to start the search for ray-triangle intersections. The call receives the indices of the hit group and miss shader to be used, the ray, the payload and the acceleration structure as parameters. Additionally, an instance inclusion mask is used which is set to ~2, meaning negated 2. This means that this ray will ignore intersections with the particle quads since they have the instance mask value 2 and this is done because they are irrelevant for this part of the render as they are only used when drawing the image. The ray flag used is the *RAY_FLAG_SKIP_CLOSEST_HIT_SHADER* which means that no closest hit shader will be invoked by the trace of this ray. This is done because all the necessary work can be done in the any-hit shader and therefore there is no need for a closest hit shader in this render pass.

The any-hit shader begins by calculating the starting index for the current particle to place its hits at in the hit data UAV, since the data here has to be managed manually, as previously mentioned. Afterwards, the instance ID is checked to see if the current hit is with something that is not another particle, such as the opaque ground plane. If this is true, it means that the particle is occluded from the light source and there is no need to continue the search for intersections. A minus one is then stored in the position of the current particle's final hit's ray $t$ value as an indicator for later on that the particle is occluded and then the AcceptHitAndEndSearch() function is called to end the search for more intersections. However, if the particle is not occluded, then another check is made to see if the total number of hits so far, which is found in the payload, has reached the maximum number of allowed hits, and if so, AcceptHitAndEndSearch() is also called. The reason why we have a limit to the number of allowed hits is explained in section 4.4.3. Finally, if the hit is on another particle and more hits can be stored, then the ray's $t$ value and a one or minus one is stored in the UAV. Whether a one or minus one is stored depends on if the ray is entering a particle or leaving one and this is defined as the ray hitting a front face or a back face respectively of a triangle. In HLSL there is a built in function named HitKind() which returns either *HIT_KIND_TRIANGLE_FRONT_FACE* or *HIT_KIND_TRIANGLE_BACK_FACE* which is used in this any-hit shader to determine which face is hit. The shader finishes by incrementing the counter in the payload and then calls IgnoreHit() in order to continue the search for more intersections. The miss shader used for these rays does nothing at all. It is only there because the API requires that there is a miss shader.

### 4.3.3 Compute Pass

Using the ray hits that were found in the previous pass, we employ a compute shader to calculate the optical path length for each particle as described in section 3.1. The compute pass begins, for each thread, by looking up its particle ID and using this to calculate where in the UAV it should retrieve the hits from, just as is done in the optical depth render pass.

Afterwards, the last element in the space of the UAV that is allocated for the current particle is checked to see if a hit has been stored at this location. If it has, then it means that the trace of this particle's ray either hit more than the maximum number of allowed hits or the particle is occluded by, for example the ground plane, indicated by the minus one mentioned in section 4.3.2. If this is true then the particle is considered to be fully occluded and a very high value is placed into the UAV that contains the optical path lengths at the particle's index and then the thread returns.

However, if the current particle's ray only intersected other particles and not too many of them, the sorting phase of the compute pass begins. In the sorting phase, all of the hits are sorted according to the stored ray $t$ value from smallest to largest, using bubble sort. The hits need to be in order because it is important to know exactly when the ray leaves smoke and when it enters it, as a ray can potentially intersect two or more separate clouds of smoke and the distances between them must not be included. We define one cloud of smoke as a collection of particles that overlap to form one continuous volume of smoke.

After the sorting is complete the calculation phase begins. The first part of the calculation phase is to go through all of the hits and sum up all of the hit values, that is the ones and minus ones. This gives the starting value of the counter to be used, as is described in section 3.1. Then, in the second part, the optical path length is calculated, which is done by iterating through the sorted hits in a loop. Before the loop starts, three variables are initialized and the first of these is a variable that keeps track of the $t$ value of the latest entry point into smoke which is initialized to zero. The second variable is a boolean which keeps track of when the entry point is accurate and is initialized to true and it is used to let the program know that the entry point needs to be updated after the ray has left smoke. The third variable is a float which keeps track of the total distance and is initialized to zero. In the loop, the hits are iterated through and the empty elements in the list are skipped. Since there is a fixed amount of memory allocated per particle for the hits, there are empty elements in the list if fewer hits than the maximum limit have been found. Then, if the boolean is set to false, the entry point gets set to the current $t$ value because this would mean that the ray left smoke after the previous hit and the boolean is then flipped to show that the latest entry point is now accurate. Next, the hit value is added to the counter and if it becomes zero the difference between the $t$ values of the current hit and the entry point is added to the total and the boolean is set to false because this would mean that the ray leaves smoke after this hit. When all hits have been iterated through, the total distance is stored in the appropriate UAV at the current particle's index.

### 4.3.4 Clear Compute Pass

After the compute pass has finished, we execute another compute shader to clear the UAV containing the hit data to ensure that the old hits are not left in the buffer in the next frame. The reason that this is a separate render pass is to better distribute the resources of the GPU for this task, so that each thread can clear one element in the array. The shader itself is extremely simple, where each thread uses its own thread ID as the index for the buffer and sets the value contained at that location to zero. This task could also have been performed in the compute pass after having calculated the optical path length, but this would mean that each thread in that pass would need to clear all of its particle's hits. Depending on the maximum limit for amount of hits to be stored, this could be many hits for each thread in the compute pass to clear. The performance impact of this is most likely minimal since clearing elements in the array is a simple operation, but our reasoning is that dividing up the tasks as much as possible gives a more clear picture of the performance of each of the different tasks when measuring.

### 4.3.5 View Render Pass

Once the optical path lengths are calculated and the buffer is cleared, the final step in the render loop is to draw the final image by calculating the colors of each pixel on the screen and outputting these colors to the output texture. One thing of note that we would like to point out is that we use ray tracing in order to draw each image, which is not usually done in the video game industry. In modern video games forward or forward+ rendering is commonly used instead when drawing the images. The reasons that we use ray tracing for this part are that this was easier for us to implement and that the main focus of this thesis is on what is done in the previous passes and not so much on how it is drawn. The view render pass consists of a ray generation shader, two closest hit shaders and two miss shaders. There are two payload types used in this pass, where the first of which is the view ray payload. The view ray payload is a struct that contains a 4D float used for a color, a 3D float used for the position of a hit triangle, a unsigned integer used for the instance ID of a hit triangle and a boolean which is used as a condition variable for a loop. The other payload type is the shadow ray payload and it is a struct which contains a float used to store an optical path length of a particle. The triangle intersection attributes specified are the built in default type which contain the barycentric coordinates of the intersection point of a hit triangle.

The ray generation shader begins by calling DispatchRaysIndex() to determine which pixel the current thread should calculate the color for. Then, the direction of the ray to be traced is calculated using the position of the pixel and the values provided in the *RayGenData* buffer. The origin of the ray is set to the position of the camera, also found in the *RayGenData* buffer and the ray's TMin value is set to $10^{-3}$ and the TMax value is set to $10^{38}$. Having a TMin value that is too large means that particles can be missed and having a very small TMin value was found to be detrimental to performance and this is why we chose the value $10^{-3}$. The TMax value is set to the largest possible value in order to not miss any particles. The ray tracing is done iteratively in a loop in this ray generation shader. Before the main ray loop starts, some values are defined, namely the final pixel color with the starting values of zeroes and the flag which determines if the main ray loop should end with the starting value false. A counter is also used in order to limit the maximum number of iterations of the main ray loop. In each iteration of the main ray loop a view ray payload is defined and then TraceRay() is called. The TraceRay() call receives the acceleration structure, the previously defined ray and the payload as parameters and the ray flag used is *RAY_FLAG_NONE*. The instance inclusion mask used is set to ~1, meaning negated 1, which means that the ray will ignore intersections with the particle isocahedrons since they have the instance inclusion mask value 1 and this is done because they are not supposed to be actually drawn on screen. The closest hit shader used is the view closest hit shader and similarly the miss shader used is the view miss shader. After the TraceRay() call returns, the boolean is checked and if it is set to true it means that the ray did not hit a particle and if false it did hit a particle. If the ray did hit a particle the final color value will be updated by first looking up the optical path length of the hit particle in the UAV using the instance ID found in the payload as the index. Then the color is calculated according to equation 4.1.

$$c_{tot} = c_{tot} + (1 - a_{prev}) * c_{new} * a_p \tag{4.1}$$

Equation 4.1 is based on the algorithm to calculate transparency found in *Multi-Layer Alpha Tracing* by Brüll and Grosch [3]. We modify the algorithm slightly since we update the total color after each hit, whereas Brüll and Grosch calculate it once after having gathered all the hits, but the result is the same. $c_{tot}$ represents the total color value, $a_{prev}$ the previous alpha value, $a_p$ the alpha value found in the payload and $c_{new}$ is defined in equation 4.2.

$$c_{new} = c_p * e^{-l*\sigma_t} + c_{amb} \tag{4.2}$$

Where $c_{new}$ represents the new particle color to be added to the total, $c_p$ the color found in the payload, $l$ the looked up optical path length, $\sigma_t$ the extinction coefficient and $c_{amb}$ the ambient light color found in the *RayGenData* CBV. We set the extinction coefficient to be constant throughout the smoke and is user defined so it can be adjusted to change how much light is absorbed and scattered away in the smoke. After the total color is updated the previous alpha is set to the alpha found in the payload. If the ray hit something other than a particle the color will be updated mostly the same way, except that the optical depth factor is disregarded since the ray would then not have passed through any smoke. Afterwards, the ray's origin is set to the hit position found in the payload in preparation for the next iteration of the main ray loop where the color of the object behind the one found in this iteration will be calculated. This is assuming that the object that was hit in this iteration was a particle, otherwise the loop would end. The loop counter is also incremented at this point, except in case the ray hit a particle quad at a point which is not covered by the textures, indicated by the color in the payload being zero in all four channels. Not doing this can mean getting dark or fully black parts on screen if many of these quad corners happen to align perfectly with the view rays when they are not supposed to be drawn at all. After the main ray loop is finished, the total color is written to the output texture.

The view closest hit shader begins by looking up the indices of the hit triangle to then look up the corresponding vertex attributes. If the ray has hit the ground plane, then a new ray is defined with its origin at the hit point and it is directed towards the light source. The ray TMin and TMax are set to $10^{-3}$ and $10^{38}$ respectively. A shadow ray payload is also defined with the float initialized to zero and then TraceRay() is called using this new ray and payload. The acceleration structure, shadow closest hit shader and shadow miss shader are also input as parameters and the ray flag used is *RAY_FLAG_FORCE_OPAQUE* which means that any-hit shaders will not be invoked and we do this because we are only interested in the closest hit in this case. The instance inclusion mask used is ~2, meaning negated 2, which means that the ray will ignore intersections with the particle quads which are not relevant for these rays. The shadow closest hit shader will look up the optical path length in the UAV using the instance ID as the index and then store it into the shadow ray payload before returning. The shadow miss shader will store a zero in the shadow ray payload. Returning to the view closest hit shader the color of the ground plane will then be calculated with the optical path length returned from the shadow ray trace. We base the color calculation on the Phong model described in subsection 2.1.4 and our implementation of it calculates the color according to equation 4.3.

$$c = c_{ground} * e^{-l*\sigma_t} \tag{4.3}$$

Where $c$ represents the color of the ground plane with shadowing taken into account, $l$ the looked up optical path length, $\sigma_t$ the extinction coefficient of the smoke and $c_{ground}$ is calculated according to equation 4.4.

$$c_{ground} = c_{base} * c_{diff} * max((L \cdot N), 0) \tag{4.4}$$

Where $c_{ground}$ represents the color of the ground plane with the lighting taken into account, $c_{base}$ the predefined base color of the ground plane, $c_{diff}$ the diffuse light from the light source found in the *RayGenData* CBV, $L$ the light direction also found in the same CBV and $N$ the normal of the ground plane found in the vertex attributes. With the color calculated it, along with the position of the hit in world space and the instance ID of the hit, is stored in the payload. The boolean flag in the payload is set to true to end the main ray loop because the ground plane is not transparent and therefore there is no need to know what is behind it when drawing.

However, if the ray hits a particle quad instead of the ground plane, then things are done differently. Firstly, the texture coordinates of the intersection point are looked up using the vertex attributes and the barycentric coordinates and the current texture frame is also looked up from the *ValuesData* CBV. Then a check is made to make sure that the intersection point on the triangle is covered by the textures, because if not this point should not be drawn and so zeroes are stored in the color in the payload and then the shader returns. If the intersection point is covered by the textures, the next step is to calculate the lighting. In order to do this some matrices must be constructed first namely, a tangent space matrix and rotation matrices around the x-, y- and z-axis. Quads do not really have any tangents and binormals, but since these quads always face the camera it is possible to construct a tangent space matrix based on the camera's direction vectors. The rotation matrices are created using the rotation angles of the quads found in the *ValuesData* CBV. These rotation matrices are multiplied with the tangent space matrix in order to make it take the current rotation of the particles into account. Afterwards, the light direction is looked up and converted to tangent space using the previously mentioned matrix. Using the light direction in tangent space it can be determined which of the two textures at the current angle should be used for each axis. The following code block describes how it is done as well as how the texture values are combined to get the final light map value.

```
float hMap = (lightDirectionTS.x > 0.0f) ? posTexture.r : negTexture.r;
float vMap = (lightDirectionTS.y > 0.0f) ? posTexture.g : negTexture.g;
float dMap = (lightDirectionTS.z > 0.0f) ? posTexture.b : negTexture.b;

float lightMap = hMap * lightDirectionTS.x * lightDirectionTS.x +
                 vMap * lightDirectionTS.y * lightDirectionTS.y +
                 dMap * lightDirectionTS.z * lightDirectionTS.z;
```

The final color value to be stored in the payload is calculated by multiplying this light map value with the light diffuse color from the *RayGenData* CBV and the alpha channel of the positive texture. The alpha value for the payload can be manually defined to choose how transparent the smoke should be. The intersection position and the instance ID are also stored in the payload before the shader returns.

Finally, the view miss shader is a very simple shader. It sets the color in the payload to a blue color and sets the boolean flag to true in order to end the main ray loop in the ray generation shader. After this is done the miss shader returns.

## 4.4 Parameters

In this section we go over the main parameters of the program. These are parameters that are set at compile time and can be changed to either increase performance or image quality. These are the number of particles in the particle system, the amount of particles whose optical depths get updated each frame and the cutoff point of the optical depth render pass. The effects of changing the values of these parameters is presented in detail in chapter 5.

### 4.4.1 Number of Particles

The number of particles in the particle system can be set to any number. Closely related to the choice of number of particles is their size and having a large amount of particles means that they can be smaller and having a small amount of particles means they will need to be bigger. More particles means that the smoke will be able to have a more realistic shape and be more nuanced. However, it also means that the performance will be worse since the ray intersection tests will need to traverse a bigger BVH since there are more particles in it.

### 4.4.2 Particles per Group

In this program it is possible to only update the optical depth for a certain fraction of the particles each frame. For example, all particles can be divided into four groups where each group's particles get their optical depths updated one at a time over four frames. In a typical video game we expect this value to be set as part of graphics quality settings or controlled dynamically by the rendering engine itself to balance performance versus accuracy. The idea with this parameter is to distribute the calculations over time which would decrease individual frame time and since a frame is rendered so quickly from a human perspective the effects of this would not be so noticeable. It is shown in chapter 5, that this parameter produces the intended effect.

### 4.4.3 Optical Depth Cutoff Point

Due to limitations in the implementation it is not possible to store varying numbers of hits in the hit information UAV so a hard maximum limit has to be placed. This limit is what we refer to as the optical depth cutoff point and setting it to one hundred, for example, would mean that only one hundred hits per particle can be stored in the hit information UAV. If the number of hits found in the optical depth render pass for a given particle is larger than the optical depth cutoff point then it is considered fully obscured from the light and it is given a very high value in the optical path length UAV. The idea with this parameter is that after light has travelled through enough smoke there will be so little of it left that it would not be noticeable to just consider that point to be obscured from the light source. Also, decreasing this limit has a positive impact on performance. It is shown that this parameter produces the intended effect, for the most part, and this is discussed in chapter 5.

# Chapter 5

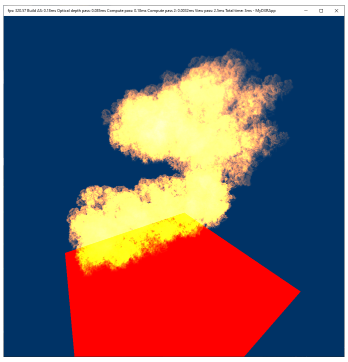
# Results

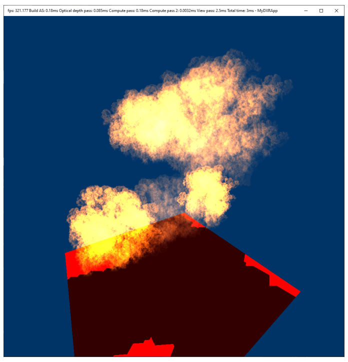
In this chapter we present the results produced by the implementation. This includes produced images and their average frame times for different values of number of particles, amount of particles per group and the optical depth cutoff point. We also present the produced images using the different render modes. The images produced were made using an Nvidia RTX 2080 Ti GPU. All images presented are rendered at 1024x1024 resolution.

## 5.1 Render Modes

In figures 5.1 and 5.2 the three different render modes can be compared. Figure 5.1 shows the scene with the light coming from above and figure 5.2 shows the scene with the light coming from the right from a different viewing angle. We use these different angles to show different effects of the optical depth simulation. The images are rendered using 750 particles, the particles are not divided into any groups and the optical depth cutoff point is set to 60. The purpose of this comparison is to make it clear what details of the image are because of simulating the optical depth specifically.

In figures 5.1a and 5.1b there are some clear differences, such as the presence of shadows in figure 5.1b, but also that the light is overall dimmer due to the self-shadowing from the optical depth. Looking at the middle of the bottom tube in figure 5.1b another important feature of the optical depth is made clear and this is the shadowing that is caused by the upper tube. The differences in figures 5.1b and 5.1c is in the shadows on the ground plane. In simple shadow mode, a point on the ground plane is either fully occluded from the light or not at all, where as in full optical depth mode the shadowing is based on the optical depth of the closest particle in the direction of the light. This is made clear by comparing figures 5.1b and 5.1c where the shadows vary in strength depending on how much smoke is in between a given point and the light source in full optical depth mode. The shadows go from being very faint under the edges of the tubes to being quite strong in the middle of the plane where both tubes are occluding in figure 5.1c.


**(a)** Basic Mode


**(b)** Simple Shadow Mode

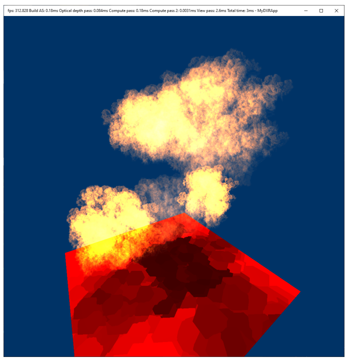
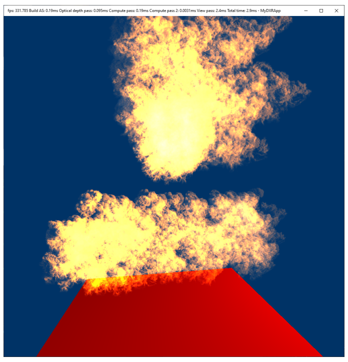
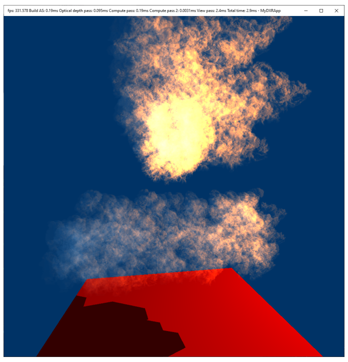

**(c)** Full Optical Depth Mode

**Figure 5.1:** The three images show the smoke scene in the three different render modes with the light coming from above

In figures 5.2a and 5.2b there are also some clear differences. The biggest difference is that in the bottom tube in figure 5.2b the light fades as it travels through the tube from the right to the left which is due to the optical depth. Additionally, there are also the shadows at the left part of the ground plane which differ between the figures. Again, the differences between figures 5.2b and 5.2c is in the shadows. The shadows are fainter in figure 5.2c, which is more appropriate given that the smoke is not thick enough to block all light traveling through from that angle.


**(a)** Basic Mode


**(b)** Simple Shadow Mode

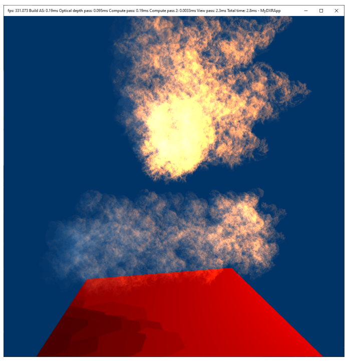

**(c)** Full Optical Depth Mode

**Figure 5.2:** The three images show the smoke scene in the three different render modes with the light coming from the right

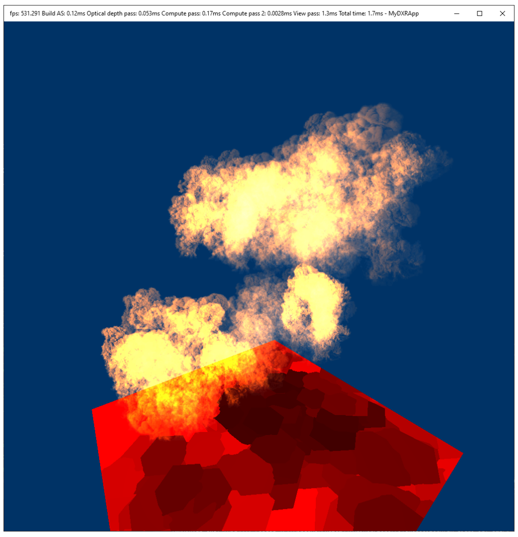
## 5.2 Image Comparisons

In this section we compare images using different values for the different parameters. From here on out all images referred to are rendered in full optical depth mode. The first parameter presented is the number of particles in the system, followed by the amount of particles per group that get updated optical depth values each frame and finally the optical depth cutoff point.

### 5.2.1 Number of Particles

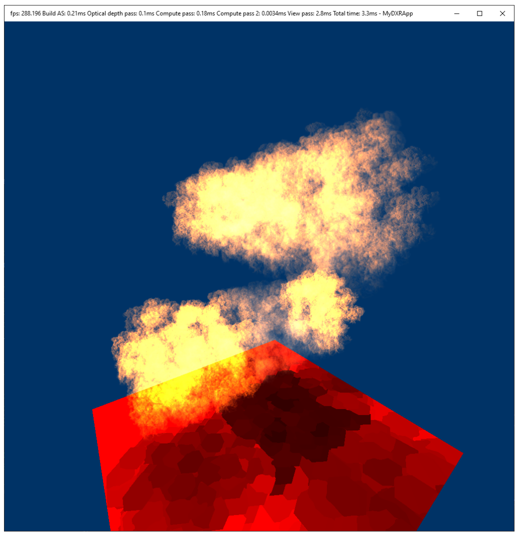
In figures 5.3 and 5.4 two different values for the number of particles in the system can be compared from two different views and these values are 250 and 1500 particles. The two different views are used to show changes in different effects of the optical depth simulation. The values for the number of particles were chosen to provide a somewhat balanced trade-off between performance and image quality. In order to keep the differences in the images as focused on the optical depth as possible, some values related to the appearance have been changed between images. The size of the particles is adjusted to compensate for the total number of particles in order to keep the total volume of the smoke roughly the same. The opacity of the particles is also adjusted to compensate for the increased or decreased volume of each particle, as a larger particle represents a larger portion of the total smoke and should therefore be more opaque. All of these images are rendered with all particles in one group and the optical depth cutoff point set to a maximum of 60 hits.

The two images in figure 5.3 show the smoke with the light coming from above, both with a different total number of particles in the system. The main difference between the images is that as the number of particles increase, the shadows become smoother and more nuanced due to the optical depth being calculated at more points in the smoke when there are more particles. Besides this, the lighting in the smoke is also a bit smoother and more precise as the number of particles increase because, again, the optical depth is calculated at more points. In addition to these images, four more are shown in figure A.1 in the appendix. These four extra images are captured from renders with 500, 750, 1000 and 1250 total particles in the system.



```
fps: 531.291 Build AS: 0.12ms Optical depth pass: 0.053ms Compute pass: 0.17ms Compute pass 2: 0.0028ms View pass: 1.3ms Total time: 1.7ms - MyDXRApp
```

**(a)** 250 particles

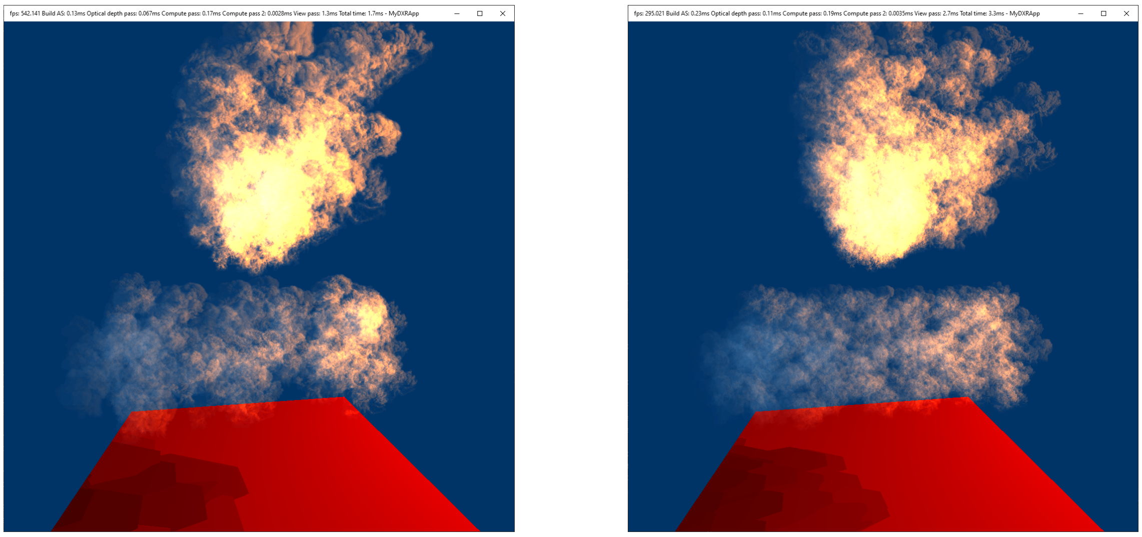


```
fps: 288.196 Build AS: 0.21ms Optical depth pass: 0.1ms Compute pass: 0.18ms Compute pass 2: 0.0034ms View pass: 2.8ms Total time: 3.3ms - MyDXRApp
```

**(b)** 1500 particles

**Figure 5.3:** The two images show the smoke scene with different numbers of particles in the system with the light coming from above

The two images in figure 5.4 show the smoke with the light coming from the right, both with a different total number of particles in the system. In these images, the difference in the appearance of the smoke when using different amounts of total particles is a bit more clear when compared to figure 5.3. In the bottom tube in the images in figure 5.4 it is more apparent that the light fades more gradually throughout the tube with an increasing total number of particles in the system. In the left part of the ground plane it can also be seen that the shadows look better with more particles, like previously described in the last paragraph. In addition to these images, four more are shown in figure A.2 in the appendix. These four extra images are captured from renders with 500, 750, 1000 and 1250 total particles in the system.



```
fps: 542.141 Build AS: 0.13ms Optical depth pass: 0.067ms Compute pass: 0.17ms Compute pass 2: 0.0028ms View pass: 1.3ms Total time: 1.7ms - MyDXRApp
```

```
fps: 295.021 Build AS: 0.23ms Optical depth pass: 0.11ms Compute pass: 0.19ms Compute pass 2: 0.0035ms View pass: 2.7ms Total time: 3.3ms - MyDXRApp
```

**(a)** 250 particles **(b)** 1500 particles

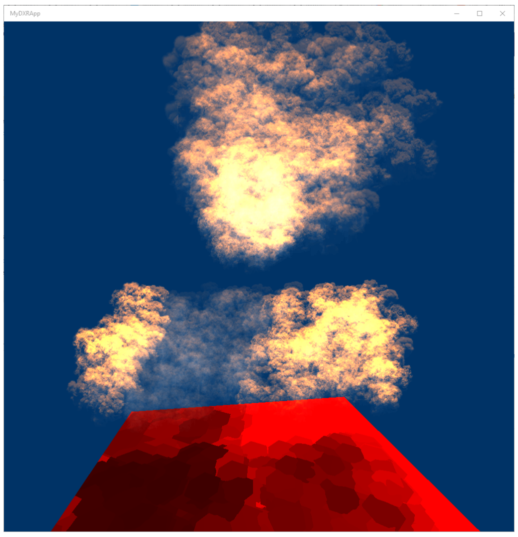
**Figure 5.4:** The two images show the smoke scene with different numbers of particles in the system with the light coming from the right

### 5.2.2 Particles per Group

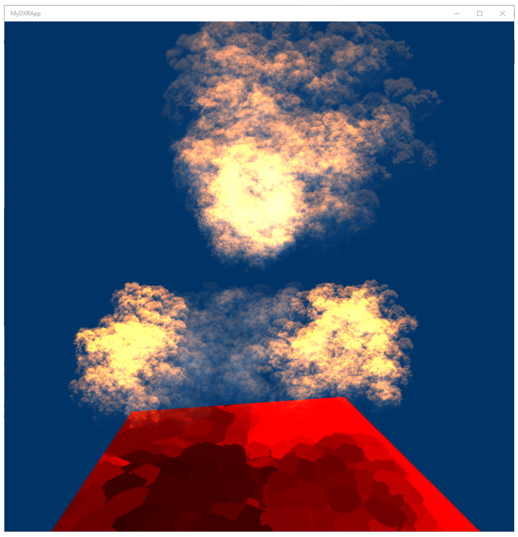
In figure 5.5 two images are presented which are captured at the thirty-second frame of two different real-time renders. The reason that it was the thirty-second frame specifically that was chosen was so that all particles would be able to have their optical depth calculated at least once for all of the renders. Both renders are of the smoke scene with 1024 particles and an optical depth cutoff point of 60 and the light source also rotates around the z-axis clockwise. The difference between the renders is that they are configured to only update the optical depth of a fraction of the total amount of particles each frame, where which fraction that is chosen is what differentiates them. In subfigure 5.5a it is all particles every frame and a thirty-second of the particles in subfigure 5.5b. Additionally, images were also captured from renders where the optical depth of half, a fourth, an eighth and a sixteenth of all particles are updated each frame, and these images, along with the two previously mentioned images, are shown in figure A.3 in the appendix. It is difficult to compare still images when varying which fraction of particles should be updated every frame since the intended effect is better observed in real-time so videos of each render is available in the Git repository for this project

[1] [2] [3] [4] [5] [6]. In these videos the framerate has been capped at 60 frames per second.

The two images in figure 5.5 do not show considerable differences in the appearance of the smoke. The most notable difference in the smoke is that the shadow from the upper tube on the lower tube lags behind more and more with an increasing number of particle groups. The edges of the same shadow also become more and more blurry as the number of particle groups increases, since it takes longer for each particle to get updated lighting from the new position of the light source. The shadow on the ground plane in subfigure 5.5b also shows how the shadows cannot quite keep up with where they are supposed to be, which is shown in subfigure 5.5a, and this is due to the former render having many particle groups.



**(a)** 1 particle group with 1024 particles



**(b)** 32 particle groups with 32 particles each

**Figure 5.5:** The two images show the smoke scene with different amounts of particles per render group both captured at the 32nd frames

### 5.2.3 Optical Depth Cutoff Point

For the optical depth cutoff point parameter, six different values were used to render six different images. These values range from 20 to 120 and increase in steps of 20. The values were, again, chosen to provide a somewhat balanced trade-off between performance and image quality. In figure 5.6 the three most significant images are presented, which have the

[1] `https://gitlab.com/LukasStudent123/mydxrapp/-/blob/main/videos/view2_1024particles_1particlegroup_60opticaldepth.mp4`

[2] `https://gitlab.com/LukasStudent123/mydxrapp/-/blob/main/videos/view2_1024particles_2particlegroups_60opticaldepth.mp4`

[3] `https://gitlab.com/LukasStudent123/mydxrapp/-/blob/main/videos/view2_1024particles_4particlegroups_60opticaldepth.mp4`

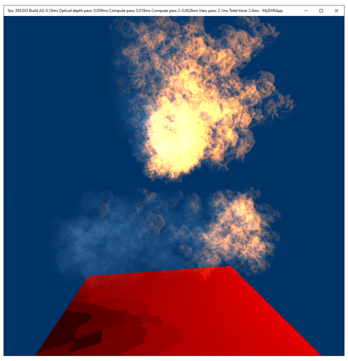
[4] `https://gitlab.com/LukasStudent123/mydxrapp/-/blob/main/videos/view2_1024particles_8particlegroups_60opticaldepth.mp4`

[5] `https://gitlab.com/LukasStudent123/mydxrapp/-/blob/main/videos/view2_1024particles_16particlegroups_60opticaldepth.mp4`

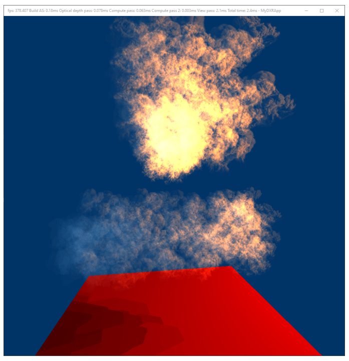
[6] `https://gitlab.com/LukasStudent123/mydxrapp/-/blob/main/videos/view2_1024particles_32particlegroups_60opticaldepth.mp4`

values 20, 40 and 80 set as the optical depth cutoff point in their renders. All six images are presented together in figure A.4 in the appendix. The values for the other parameters are 750 particles and 1 particle group. In these renders, the light is set to come from the right towards the smoke tubes.

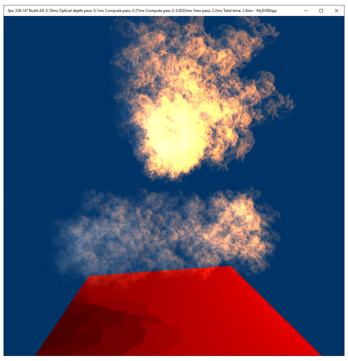
The three images in figure 5.6 show some significant differences between each other. In subfigure 5.6a a low value of 20 is set as the optical depth cutoff point which shows by how short of a distance the light is allowed to travel through the bottom tube. The light does not completely reach through even the upper tube either. In subfigure 5.6b a slightly higher value of 40 is set as the optical depth cutoff point and this makes a remarkable difference, as the light can now travel much farther through the bottom tube and also completely through the upper tube. In subfigure 5.6c the value 80 is set as the optical depth cutoff point and now the difference is not as remarkable. A bit more light is visible towards the left end of the bottom tube, but not much else differs between this and the previous image. In subfigure A.4c the value 60 is set as the optical depth cutoff point and this image contains barely any differences at all compared to subfigure 5.6c. The only noticeable change is that there is a little more light visible at the very leftmost edge of the bottom tube in the latter image. In subfigures A.4e and A.4f the optical depth cutoff points are set at 100 and 120 respectively. In these images there are no noticeable differences compared to subfigure 5.6c.


fps: 395.053 Build AS: 0.18ms Optical depth pass: 0.059ms Compute pass: 0.018ms Compute pass 2: 0.0028ms View pass: 2.1ms Total time: 2.4ms - MyDXRApp

**(a)** 20 optical depth cutoff point


fps: 378.407 Build AS: 0.18ms Optical depth pass: 0.078ms Compute pass: 0.063ms Compute pass 2: 0.003ms View pass: 2.1ms Total time: 2.4ms - MyDXRApp

**(b)** 40 optical depth cutoff point


fps: 336.147 Build AS: 0.18ms Optical depth pass: 0.1ms Compute pass: 0.37ms Compute pass 2: 0.0033ms View pass: 2.2ms Total time: 2.8ms - MyDXRApp

**(c)** 80 optical depth cutoff point

**Figure 5.6:** The three images show the smoke scene with increasing values for the optical depth cutoff point parameter

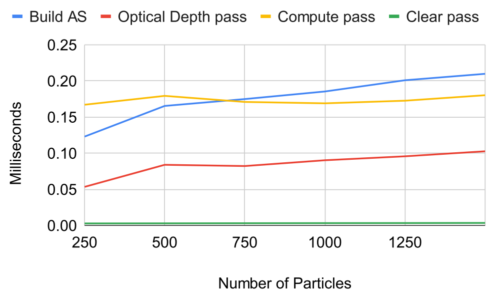
## 5.3 Render Time Comparisons

In this section we compare the frame times and all major components of the frame times for different values for the program parameters. First, we go over the frame time differences when changing the number of total particles in the scene, followed by the differences when changing how many particles' optical depths get updated each frame and then the differences when changing the optical depth cutoff point value. Graphs are presented which show the render times of the different components of the render loop as well as the total frame time. The total frame time here is measured on its own from the start of a frame to the end of a frame and is not measured as a sum of the other components. The render times are measured by adding timestamp queries to the command list of the GPU before and after each pass, where the difference between these is used as the render time. The total frame time is measured by the difference between the values of a timestamp query at the very top of the command list and another at the very bottom of the command list. This means that the total frame time also includes the time taken by things like resource barriers. Also, we would, again, like to point out that the view render pass is ray traced and because of this it is most likely slower than an equivalent render pass used in a video game that uses forward or forward+ rendering due to ray tracing hardware being slower. The graphs are split into two per data set because of the difference in scale between some of the components.

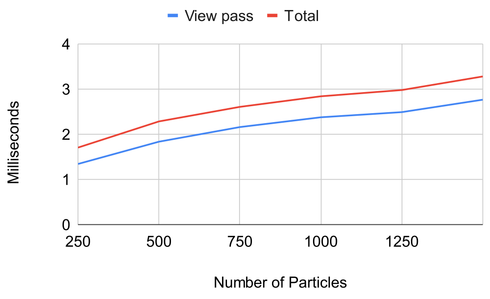
### 5.3.1 Number of Particles

In figure 5.7 the average render times of the renders which produced the images in figure A.1 is presented. In subfigure 5.7a the render times of building the acceleration structures, the optical depth pass, compute pass and clear compute pass are presented. In subfigure 5.7b the average render times of the view pass and the average total frame times are presented. The x-axes show the different number of total particles starting with 250 particles and increasing in steps of 250 until 1500 particles. The y-axes show average render times over 5000 frames in milliseconds. The average render times stabilize after around 5000 frames for these renders which is why this number was chosen.

In figure B.1 in the appendix a table is presented which shows the numerical data which was used to create the graphs in figure 5.7 with columns for the different components of the render loop and rows for different total amounts of particles used in each render. Figure B.1 in the appendix shows two similar graphs which are instead based on the render times of the renders which produced figure A.2, though these graphs are nearly identical to the ones in figure 5.7. Similarly, there is also an equivalent table for the numerical data which produced the graphs in figure B.1 and this data is shown in table B.2 in the appendix. Looking at the graphs in figure 5.7, it is clear that the frame times increase somewhat linearly with an increasing number of particles in the system. The main contributors to this increase appears to be mainly the view pass, building the acceleration structures and the optical depth pass.



**(a)** Render times for building acceleration structures, optical depth pass, compute pass and clear compute pass



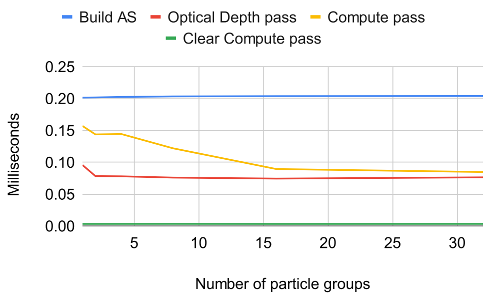
**(b)** Render times for the view pass and the total frame times

**Figure 5.7:** The two graphs shows the average render times for the render components depending on the number of particles in the system in view 1

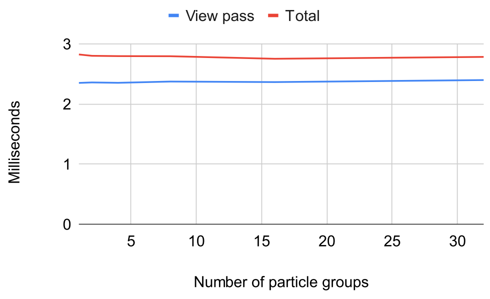
## 5.3.2 Particles per Group

In figure 5.8 the average render times of the renders which produced the images in figure A.3 is presented. In subfigure 5.8a the render times of building the acceleration structures, the optical depth pass, compute pass and clear compute pass are presented. In subfigure 5.8b the average render times of the view pass and the average total frame times are presented. The x-axes show the different number of particle groups starting with 1 particle group and goes up until 32 particle groups. The measured values are 1, 2, 4, 8, 16 and 32 particle groups. The y-axes show average render times over 5000 frames in milliseconds.

In figure B.3 in the appendix a table is presented which shows the numerical data which was used to create the graphs in figure 5.8 with columns for the different components of the render loop and rows for different total amounts of particles used in each render. Looking at subfigure 5.8a, the average render times of the compute pass seem to decrease with higher numbers of particle groups, where the render times of dividing the particles into more groups appears to decrease less and less with higher numbers of particle groups. For the rest of the components of the render loop, as well as the view pass seen in subfigure 5.8b, there is not any significant impact on performance with respect to the number of particle groups, and so the overall average frame time does not change much.



**(a)** Render times for building acceleration structures, optical depth pass, compute pass and clear compute pass depending on the number of particle groups



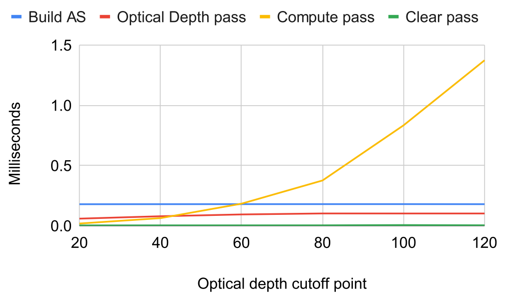
**(b)** Render times for the view pass and the total frame time depending on the number of particle groups

**Figure 5.8:** The two graphs shows the average render times for the render components depending on the number of particle groups in the system

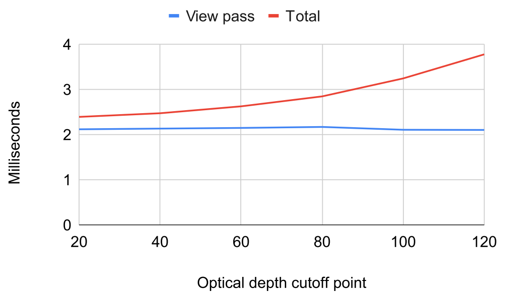
## 5.3.3 Optical Depth Cutoff Point

In figure 5.9 the average render times of the renders which produced the images in figures 5.6 and A.4 are presented. In subfigure 5.9a the render times of building the acceleration structures, the optical depth pass, compute pass and clear compute pass are presented. In subfigure 5.9b the average render times of the view pass and the average total frame times are presented. The x-axes show the different optical depth cutoff points starting at 20 and increasing in steps of 20 until 120. The y-axes show average render times over 5000 frames in milliseconds.

In figure B.4 in the appendix a table is presented which shows the numerical data which was used to create the graphs in figure 5.9 with columns for the different components of the render loop and rows for different total amounts of particles used in each render. Looking at the graph in subfigure 5.9b, it is made clear that the frame time increases exponentially with an increasing value for the optical depth cutoff point. This is clearly due to the compute pass specifically, which is shown to also increase exponentially with higher cutoff points in subfigure 5.9a. This render pass is where the hits are sorted and then traversed through. Another thing of note is that the render time of the optical depth pass is shown to stop increasing for cutoff points higher than 80 when looking at the graph in subfigure 5.9a, however this is more easily observed in table B.4 in the appendix. This is likely due to there not being any rays traced during this pass which exceed 80 hits in total, which would mean that the limit does not affect the execution of this render pass and therefore the render time remains the same. This is further supported by there not being any noticable differences between the images in subfigures A.4d, A.4e and A.4f.



**(a)** Render times for building acceleration structures, optical depth pass, compute pass and clear compute pass depending on the optical depth cutoff point



**(b)** Render times for the view pass and the total frame time depending on the optical depth cutoff point

**Figure 5.9:** The two graphs shows the average render times for the render components depending on the optical depth cutoff point

# Chapter 6
# Conclusion

In conclusion, in this thesis we have explored using ray tracing to render more realistic participating media by simulating optical depth for use in real-time applications. The results from section 5.1 show that our simulated optical depth can convincingly add important details to the smoke that improves its appearance. These would be details like that the light gets dimmer and dimmer as it travels through the smoke and the shadows of varying strength which are caused by it occluding the light source. Additionally, it is also shown that it is possible to render this with very low frame times on a modern GPU.

In section 5.2 we show the flexibility of our renderer by adjusting the different parameters to create different images. Regarding the number of particles, we find that, while differences are noticeable when comparing the resulting images side by side, the optical depth effect is still quite convincing even when using the lowest measured value of 250 particles in the system. Although, that being said, the effect does look better the more particles that are in the system. Regarding the number of particle groups, we find that, again, there are some minor differences between the resulting images and videos but these are not distracting enough to ruin the effect, even with the largest measured number of groups of 32. However, this is highly dependant on the total frame rate. In the videos the frame rate was capped at 60 frames per second, but some video games may run at only 30 frames per second for example, which would mean that the time between each optical depth update for each particle would be doubled. Regarding the optical depth cutoff point, we find that there are considerable differences between the resulting images which are rendered using different cutoff points, though these differences are smaller and smaller with increasing cutoff points. However, it is important to note that the optical depth cutoff point is highly dependant on how many and how big the particles are since it determines how many ray intersections can be stored, and does not limit the maximum distance. That being said, we find that a cutoff point of at least 40 produces a good image for this scene in particular.

In section 5.3 we show the performance of our renderer when adjusting the different parameters. We find that increasing the number of particles in the system will slightly increase the frame time linearly and that this is mainly due to the view pass, building the acceleration structures and the optical depth pass. Combining this with the fact that the renderer produces images of fairly good quality even for a smaller number of particles, we believe that it can handle a variety of values for the total amount of particles and still perform well. Updating only a fraction of the total particles' optical depth values each frame was added as a way to be able to improve performance and we find that doing this does indeed help. However, this only impacts the compute pass and this impact also lessens with more and more particle groups. Additionally, the render time of the compute pass is not a very significant portion of the total frame time, because of the view pass, so the usefulness of this technique is limited in this particular renderer when considering the overall performance. Regarding the optical depth cutoff point, we find that this parameter does not scale well in terms of performance since high values will make the compute pass very slow. Although, by dividing up the particles into groups it should be possible to lessen the performance hit for higher optical depth cutoff points.

In section 1.2 we bring up the research questions for this thesis

- To what extent can the visual quality of participating media be improved by simulating optical depth for real time applications using ray tracing hardware?
- What will the render times of this be on modern GPUs?

In the figures in sections 5.1 and 5.2 we show the important details that simulating optical depth adds for participating media which improves its appearance and we show how accurate these details can be for real time applications when changing the values of the parameters. In section 5.3 we show that the average total frame time lies between 1.70ms and 3.78ms for our implementation depending on the values of the parameters. The execution time of the optical depth calculations specifically are also shown to be between around 0.08ms and 1.48ms on average. We define the execution time of the optical depth calculations as the render times of the optical depth render pass and the compute pass combined.

## 6.1 Limitations

During the work in this thesis we encountered some limitations which changed the way we approached this project. We originally wanted to do more image comparisons that would be between our rendered smoke and some smoke found in a modern video game in order to be able to claim definitively whether or not this technique would be able to provide an improvement. However, doing this in a fair way would mean that we would have to be able to either very accurately replicate a scene from a game or somehow extract information on how exactly the smoke is rendered in the game so we could use their rendering technique in our scene. Both of these things can be quite difficult and were deemed to be beyond the scope of this project. Additionally, smoke in games usually have a lot of work from artists put in to them to make it look good. So, When doing an image comparison with our smoke and smoke from a game, we believe that it would most likely be difficult to separate what is done by artists and what is done by the rendering technique for the game's smoke.

Furthermore, we would also have liked to compare the performance of rendering our smoke against rendering some modern video game's smoke, but this would mean similar challenges to doing image comparisons as mentioned previously. It is, of course, possible to measure the performance of a scene in a video game that has smoke in it, but this would only give the performance of everything combined, whereas we are only interested in the smoke. Therefore, we would, again, need to either very accurately replicate the scene in the game or extract the information on how smoke is rendered in that game to use in our scene in order to properly compare the performance.

Finally, there is the limitation that we had to use an optical depth cutoff point because of there not being an easy way to use variable length arrays in buffers. Because of this, we had to set a fixed length for the array at compile time, which leaves very little flexibility. As is shown in section 5.3 there is no performance penalty to having too high of a cutoff point for the optical depth pass, but this is not the case for the compute pass in this implementation. At the same time, having too low of a cutoff point negatively affects the appearance of the smoke as is shown in section 5.2. This is why we consider it to be a limitation to have to use a fixed length array in the hit buffer.

## 6.2 Future Work

For future work, there are some areas that we believe could be improved upon. Firstly, is the optical depth cutoff point, as previously discussed. In order to get rid of the problems brought up in the previous section, some solution to implementing variable length arrays for use in buffers needs to be created. However, we do still believe that there is some merit to limiting the amount of hits, as was shown in chapter 5, where increases in the cutoff point become less and less noticeable but come at a significant cost to performance.

Furthermore, another area that could be improved is the hit sorting that occurs in the compute pass in order to get the hits in order before iterating through them to calculate the distance. In this thesis we used one simple algorithm to sort the hits, but there are many other sorting algorithms that could be tested to find the most optimal one for this use case. Additionally, there are many libraries based on Nvidia's CUDA API which can be used for sorting. Sorting on GPUs is an entire research subject on its own, so there are a lot of potential improvements to be found in this area.

Finally, to further expand on the topic of rendering participating media, there is also single- and multiple scattering involved in the real world physics models, which has been outside the scope of this thesis. For future improvements to the overall appearance of participating media, scattering is an area which could be delved into. There would, of course, need to be some major simplifications and approximations done to the real world physics models to render in real-time as they are quite computationally expensive.

# References

[1] Tomas Akenine-Mller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd., USA, 4th edition, 2018.

[2] Tomas Akenine-Möller. Mobile graphics hardware, 2012. `https://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/gh20121127.pdf`.

[3] Felix Brüll and Thorsten Grosch. Multi-Layer Alpha Tracing. In Jens Krüger, Matthias Niessner, and Jörg Stückler, editors, *Vision, Modeling, and Visualization*. The Eurographics Association, 2020.

[4] Simon Green. Volumetric particle shadows, December 2008. `https://developer.download.nvidia.com/assets/cuda/files/smokeParticles.pdf`.

[5] Greg Humphreys, Wenzel Jakob, and Matt Pharr. *Physically Based Rendering: From Theory To Implementation*. Morgan Kaufmann, 3 edition, 2016.

[6] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, aug 1986.

[7] Morgan McGuire. *The Graphics Codex*. Casual Effects, 2.17 edition, 2021.

[8] Microsoft. Compute shader overview. `https://docs.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-compute-shader`.

[9] Microsoft. Directx-specs | engineering specs for directx features. `https://microsoft.github.io/DirectX-Specs/`.

[10] Microsoft. Resource binding overview. `https://docs.microsoft.com/en-us/windows/win32/direct3d12/resource-binding-flow-of-control`.

[11] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, jun 1975.

[12] Peter Shirley. Ray tracing: The next week, December 2020. `https://raytracing.github.io/books/RayTracingTheNextWeek.html`.

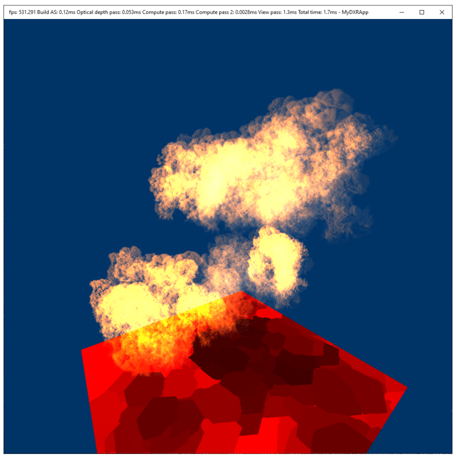
[13] Chris Wyman, Shawn Hargreaves, Peter Shirley, and Colin Barré-Brisebois. Introduction to directx raytracing. In *ACM SIGGRAPH 2018 Courses*, August 2018.

[14] Kun Zhou, Zhong Ren, Stephen Lin, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Real-time smoke rendering using compensated ray marching. *ACM Trans. Graph.*, 27(3):1–12, aug 2008.

# Appendices

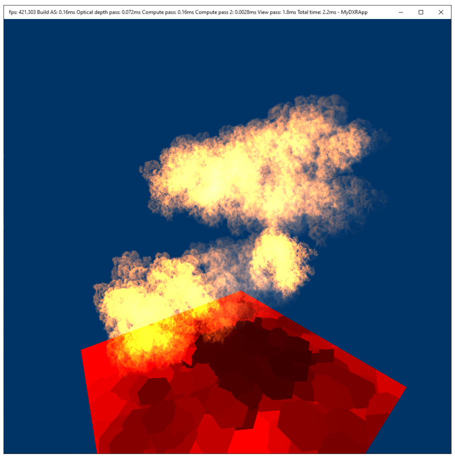
# Appendix A
# Images

Appendix A contains all of the images that were rendered using varying values for the different parameters which are brought up in section 5.2. This includes images that are left out from the aforementioned section.



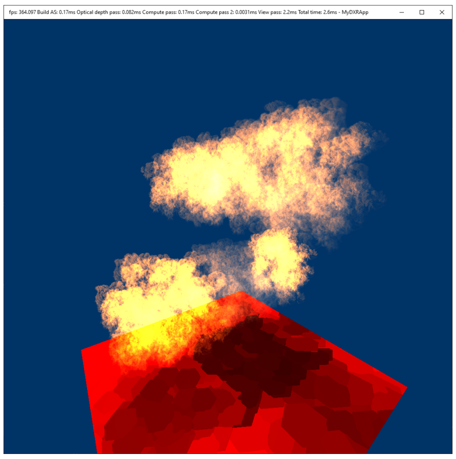
```
fps: 531.291 Build AS: 0.12ms Optical depth pass: 0.053ms Compute pass: 0.17ms Compute pass 2: 0.0028ms View pass: 1.3ms Total time: 1.7ms - MyDXRApp
```

**(a)** 250 particles



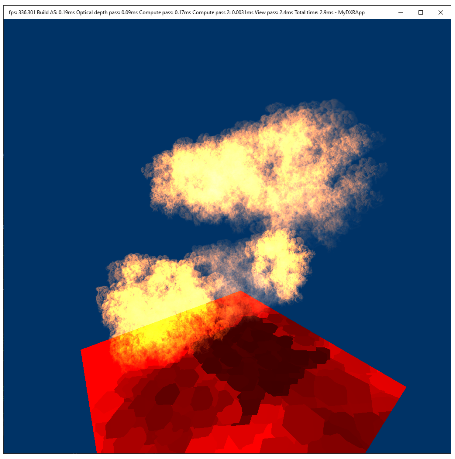
```
fps: 421.303 Build AS: 0.16ms Optical depth pass: 0.072ms Compute pass: 0.16ms Compute pass 2: 0.0028ms View pass: 1.8ms Total time: 2.2ms - MyDXRApp
```

**(b)** 500 particles



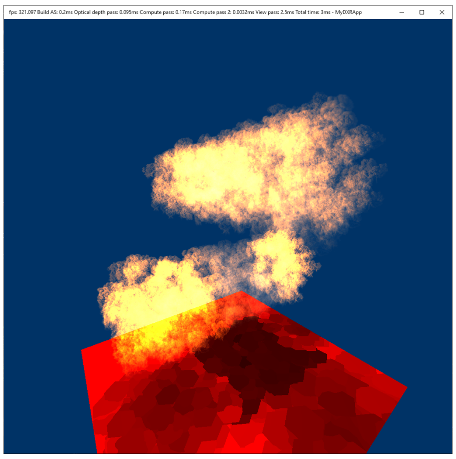
```
fps: 364.097 Build AS: 0.17ms Optical depth pass: 0.082ms Compute pass: 0.17ms Compute pass 2: 0.0031ms View pass: 2.2ms Total time: 2.6ms - MyDXRApp
```

**(c)** 750 particles



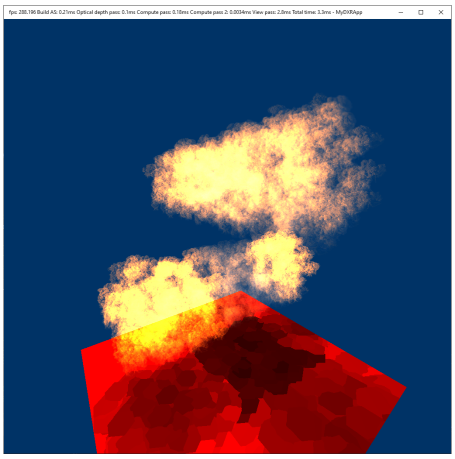
```
fps: 336.301 Build AS: 0.19ms Optical depth pass: 0.09ms Compute pass: 0.17ms Compute pass 2: 0.0031ms View pass: 2.4ms Total time: 2.9ms - MyDXRApp
```

**(d)** 1000 particles



```
fps: 321.097 Build AS: 0.2ms Optical depth pass: 0.095ms Compute pass: 0.17ms Compute pass 2: 0.0032ms View pass: 2.5ms Total time: 3ms - MyDXRApp
```

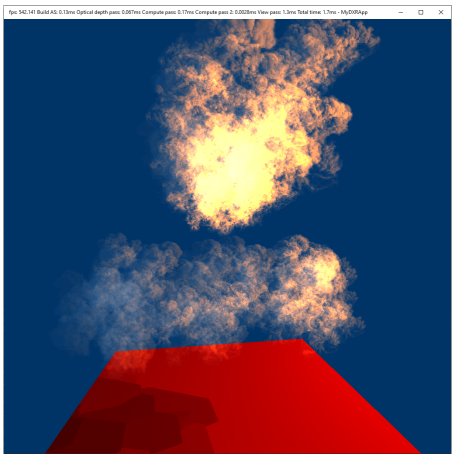
**(e)** 1250 particles



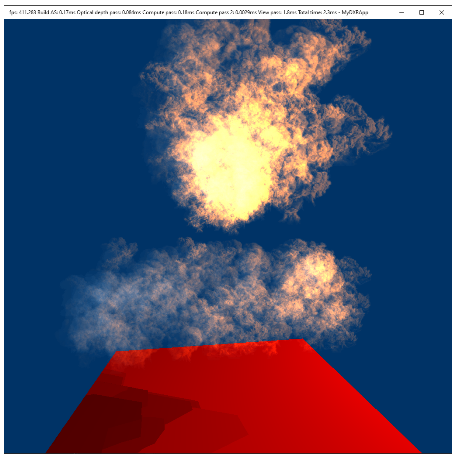
```
fps: 288.196 Build AS: 0.21ms Optical depth pass: 0.1ms Compute pass: 0.18ms Compute pass 2: 0.0034ms View pass: 2.8ms Total time: 3.3ms - MyDXRApp
```

**(f)** 1500 particles

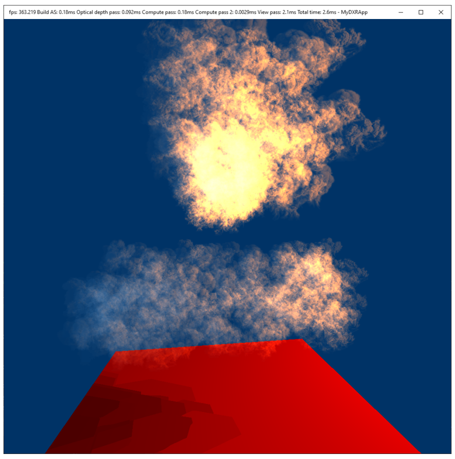
**Figure A.1:** The six images show the smoke scene with varying numbers of particles in the system with the light coming from above


fps: 542.141 Build AS: 0.13ms Optical depth pass: 0.067ms Compute pass: 0.17ms Compute pass 2: 0.0028ms View pass: 1.3ms Total time: 1.7ms - MyDXRApp

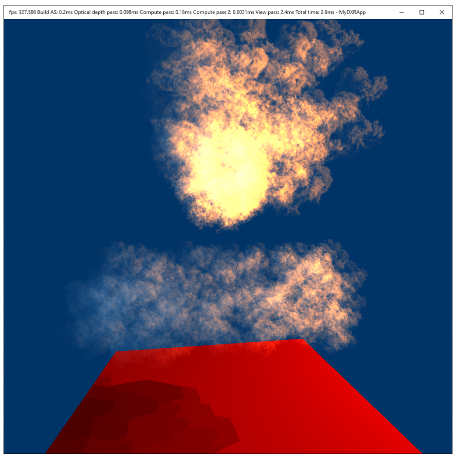
**(a)** 250 particles


fps: 411.283 Build AS: 0.17ms Optical depth pass: 0.084ms Compute pass: 0.18ms Compute pass 2: 0.0029ms View pass: 1.8ms Total time: 2.3ms - MyDXRApp

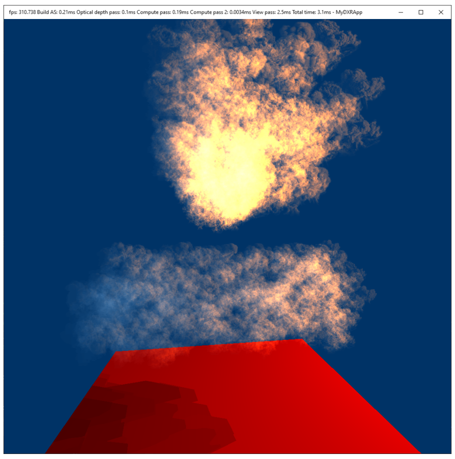
**(b)** 500 particles


fps: 363.219 Build AS: 0.18ms Optical depth pass: 0.092ms Compute pass: 0.18ms Compute pass 2: 0.0029ms View pass: 2.1ms Total time: 2.6ms - MyDXRApp

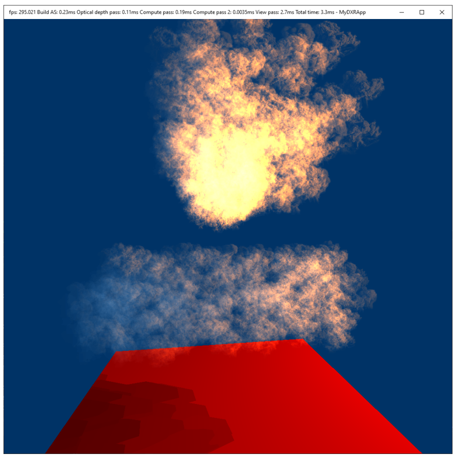
**(c)** 750 particles


fps: 327.586 Build AS: 0.2ms Optical depth pass: 0.098ms Compute pass: 0.18ms Compute pass 2: 0.0031ms View pass: 2.4ms Total time: 2.9ms - MyDXRApp

**(d)** 1000 particles

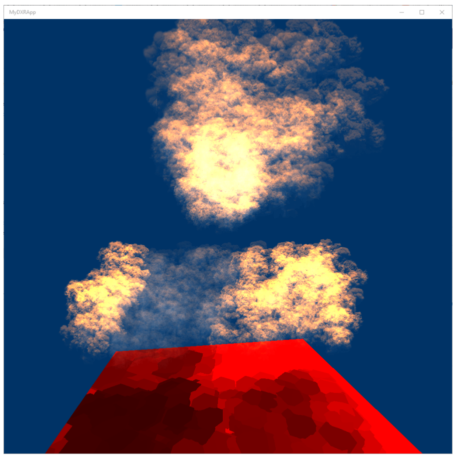

fps: 310.738 Build AS: 0.21ms Optical depth pass: 0.1ms Compute pass: 0.19ms Compute pass 2: 0.0034ms View pass: 2.5ms Total time: 3.1ms - MyDXRApp

**(e)** 1250 particles

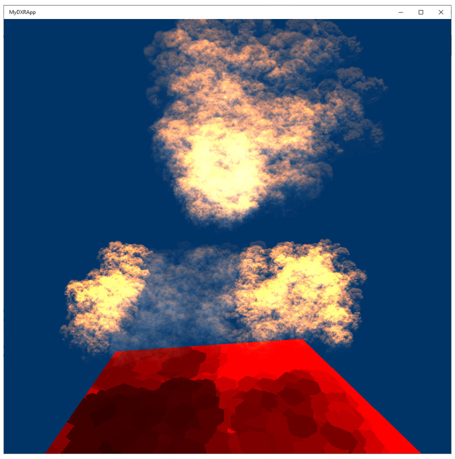

fps: 295.021 Build AS: 0.23ms Optical depth pass: 0.11ms Compute pass: 0.19ms Compute pass 2: 0.0035ms View pass: 2.7ms Total time: 3.3ms - MyDXRApp

**(f)** 1500 particles

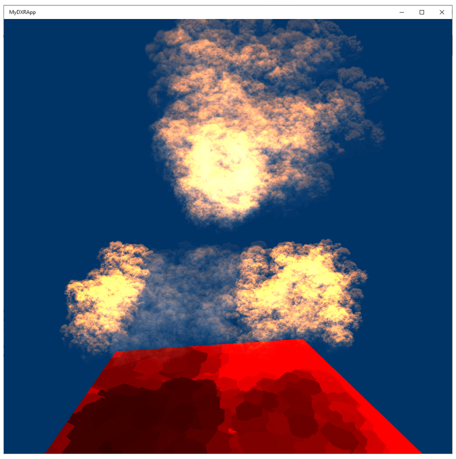
**Figure A.2:** The six images show the smoke scene with varying numbers of particles in the system with the light coming from the right



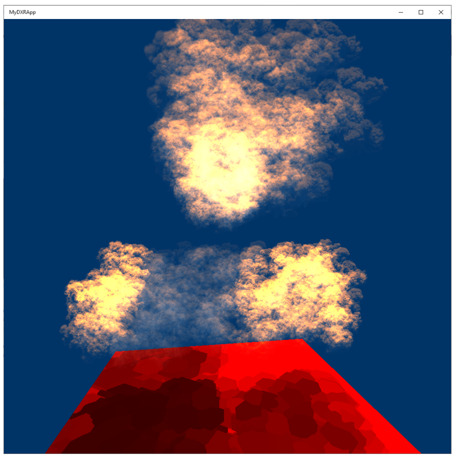
**(a)** 1 particle group with 1024 particles



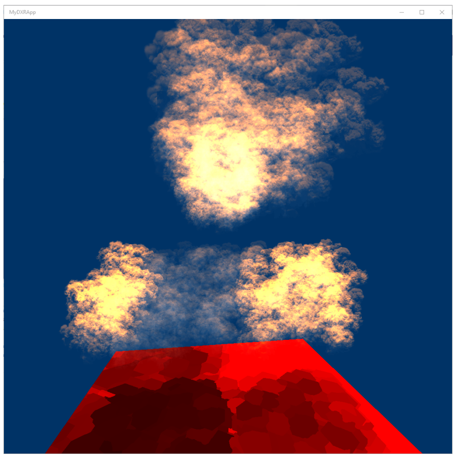
**(b)** 2 particle groups with 512 particles each



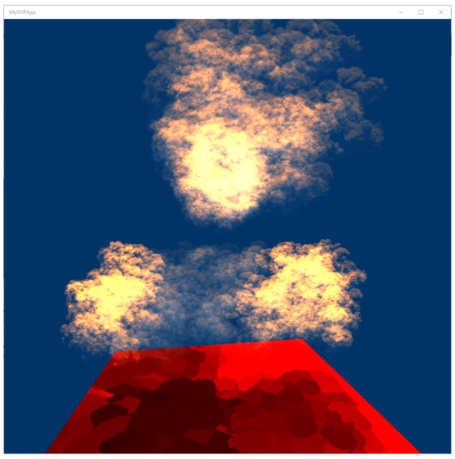
**(c)** 4 particle groups with 256 particles each



**(d)** 8 particle groups with 128 particle each

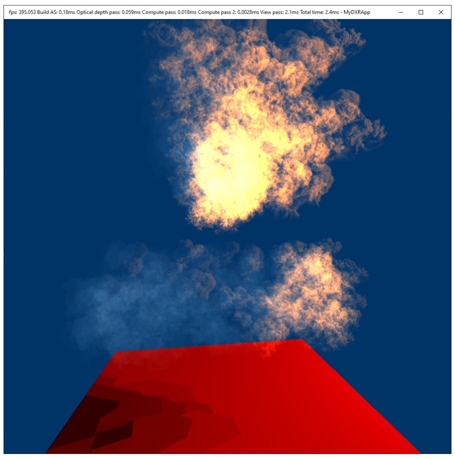


**(e)** 16 particle groups with 64 particles each

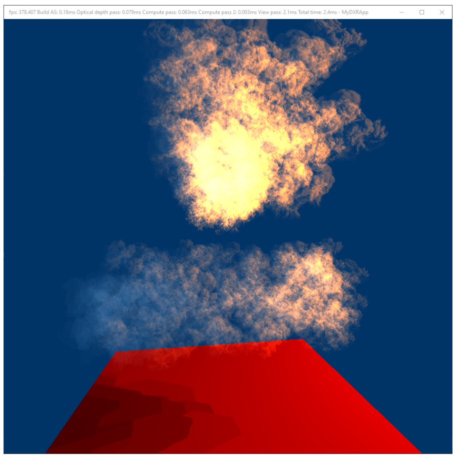


**(f)** 32 particle groups with 32 particles each

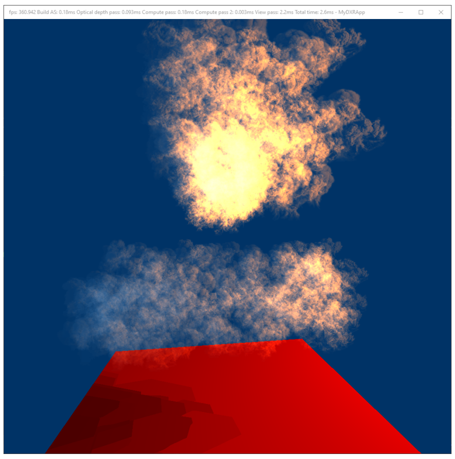
**Figure A.3:** The six images show the smoke scene with varying amounts of particles per render group all captured at the 32nd frames


fps: 395.053 Build AS: 0.18ms Optical depth pass: 0.059ms Compute pass: 0.018ms Compute pass 2: 0.0028ms View pass: 2.1ms Total time: 2.4ms - MyDXRApp

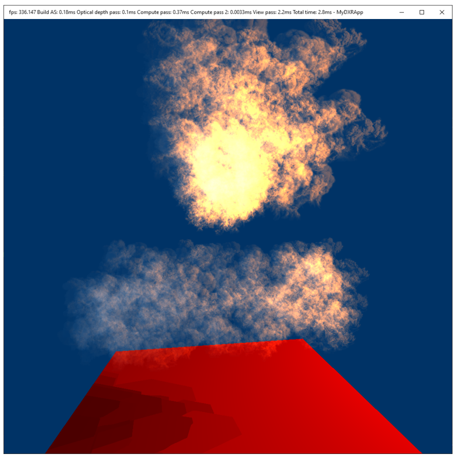
**(a)** 20 optical depth cutoff point


fps: 378.407 Build AS: 0.18ms Optical depth pass: 0.078ms Compute pass: 0.063ms Compute pass 2: 0.003ms View pass: 2.1ms Total time: 2.4ms - MyDXRApp

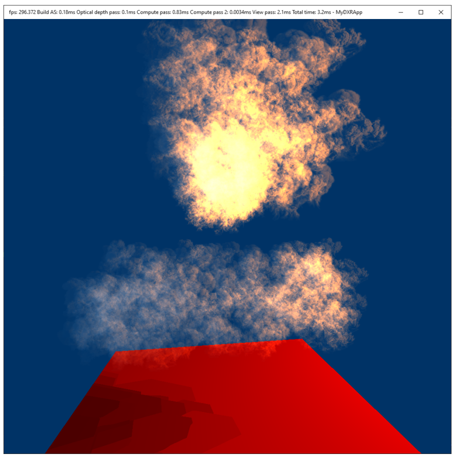
**(b)** 40 optical depth cutoff point


fps: 360.942 Build AS: 0.18ms Optical depth pass: 0.093ms Compute pass: 0.18ms Compute pass 2: 0.003ms View pass: 2.2ms Total time: 2.6ms - MyDXRApp

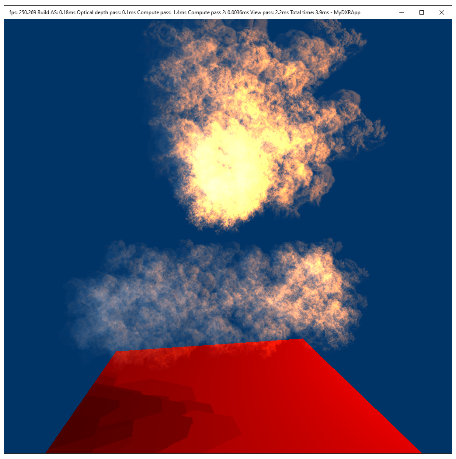
**(c)** 60 optical depth cutoff point


fps: 336.147 Build AS: 0.18ms Optical depth pass: 0.1ms Compute pass: 0.37ms Compute pass 2: 0.0033ms View pass: 2.2ms Total time: 2.8ms - MyDXRApp

**(d)** 80 optical depth cutoff point


fps: 296.372 Build AS: 0.18ms Optical depth pass: 0.1ms Compute pass: 0.83ms Compute pass 2: 0.0034ms View pass: 2.1ms Total time: 3.2ms - MyDXRApp

**(e)** 100 optical depth cutoff point


fps: 250.269 Build AS: 0.18ms Optical depth pass: 0.1ms Compute pass: 1.4ms Compute pass 2: 0.0036ms View pass: 2.2ms Total time: 3.9ms - MyDXRApp

**(f)** 120 optical depth cutoff point

**Figure A.4:** The six images show the smoke scene with increasing values for the optical depth cutoff point parameter

# Appendix B
# Tables & Graphs

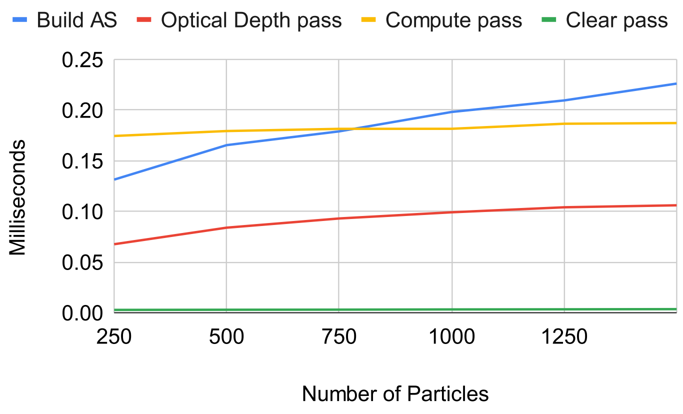
Appendix B contains tables with the data used to generate the graphs found in section 5.3. Additionally, there are also two graphs containing the average render times for the render components depending on the number of particles in view 2.

| | Build AS | Optical Depth Pass | Compute Pass | Clear Pass | View Pass | Total |
|---|---|---|---|---|---|---|
| 250 | 0.1229 | 0.0533 | 0.1670 | 0.0028 | 1.3416 | 1.7041 |
| 500 | 0.1653 | 0.0839 | 0.1793 | 0.0029 | 1.8347 | 2.2836 |
| 750 | 0.1748 | 0.0822 | 0.1709 | 0.0030 | 2.1584 | 2.6057 |
| 1000 | 0.1853 | 0.0902 | 0.1690 | 0.0032 | 2.3750 | 2.8396 |
| 1250 | 0.2009 | 0.0956 | 0.1726 | 0.0033 | 2.4891 | 2.9780 |
| 1500 | 0.2098 | 0.1026 | 0.1802 | 0.0034 | 2.7643 | 3.2789 |

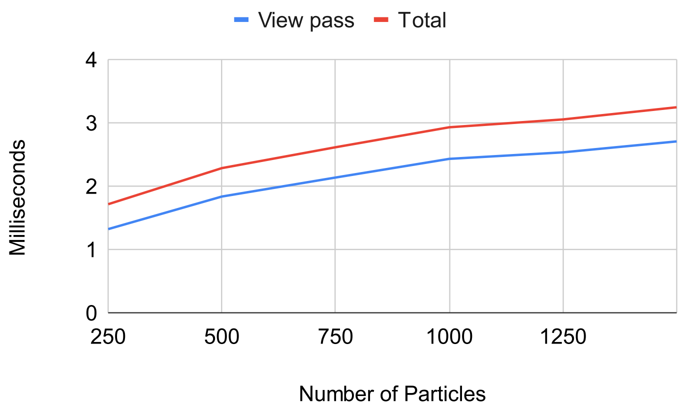
**Table B.1:** The table shows the average render times for the major components as well as the average total frame time over 5000 frames for varying numbers of total particles in milliseconds in view 1

| | Build AS | Optical Depth Pass | Compute Pass | Clear Pass | View Pass | Total |
|---|---|---|---|---|---|---|
| 250 | 0.1313 | 0.0676 | 0.1744 | 0.0027 | 1.3204 | 1.7136 |
| 500 | 0.1653 | 0.0839 | 0.1793 | 0.0029 | 1.8347 | 2.2836 |
| 750 | 0.1789 | 0.0930 | 0.1814 | 0.0030 | 2.1348 | 2.6131 |
| 1000 | 0.1981 | 0.0991 | 0.1815 | 0.0032 | 2.4296 | 2.9285 |
| 1250 | 0.2095 | 0.1040 | 0.1865 | 0.0033 | 2.5324 | 3.0523 |
| 1500 | 0.2261 | 0.1060 | 0.1871 | 0.0035 | 2.7051 | 3.2443 |

**Table B.2:** The table shows the average render times for the major components as well as the average total frame time over 5000 frames for varying numbers of total particles in milliseconds in view 2



**(a)** Render times for building acceleration structures, optical depth pass, compute pass and clear compute pass



**(b)** Render times for the view pass and the total frame time

**Figure B.1:** The two graphs shows the average render times for the render components depending on the number of particles in the system in view 2

| | Build AS | Optical Depth Pass | Compute Pass | Clear Pass | View Pass | Total |
|---|---|---|---|---|---|---|
| 1 | 0.2013 | 0.0955 | 0.1568 | 0.0032 | 2.3500 | 2.8231 |
| 2 | 0.2016 | 0.0782 | 0.1436 | 0.0032 | 2.3578 | 2.8007 |
| 4 | 0.2024 | 0.0778 | 0.1442 | 0.0032 | 2.3517 | 2.7958 |
| 8 | 0.2032 | 0.0758 | 0.1218 | 0.0032 | 2.3726 | 2.7939 |
| 16 | 0.2036 | 0.0743 | 0.0893 | 0.0032 | 2.3637 | 2.7515 |
| 32 | 0.2038 | 0.0762 | 0.0846 | 0.0032 | 2.3975 | 2.7826 |

**Table B.3:** The table shows the average render times for the major components as well as the average total frame time over 5000 frames for different amounts of particle render groups in milliseconds.

|  | Build AS | Optical Depth Pass | Compute Pass | Clear Pass | View Pass | Total |
|---|---|---|---|---|---|---|
| 20 | 0.1784 | 0.0588 | 0.0181 | 0.0027 | 2.1160 | 2.3905 |
| 40 | 0.1789 | 0.0786 | 0.0629 | 0.0030 | 2.1313 | 2.4713 |
| 60 | 0.1790 | 0.0936 | 0.1829 | 0.0030 | 2.1462 | 2.6242 |
| 80 | 0.1792 | 0.1017 | 0.3755 | 0.0032 | 2.1680 | 2.8445 |
| 100 | 0.1793 | 0.1015 | 0.8334 | 0.0054 | 2.1054 | 3.2425 |
| 120 | 0.1788 | 0.1015 | 1.3735 | 0.0035 | 2.1013 | 3.7751 |

**Table B.4:** The table shows the average render times for the major components as well as the average total frame time over 5000 frames for different values as the optical depth cutoff point in milliseconds.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-10-14

**EXAMENSARBETE** Simulating Optical Depth For Participating Media Using Ray Tracing
**STUDENT** Lukas Mattsson
**HANDLEDARE** Michael Doggett (LTH), Calle Lejdfors (Tencent)
**EXAMINATOR** Flavius Gruian (LTH)

# Strålspårning för att rendera realistisk rök i realtid

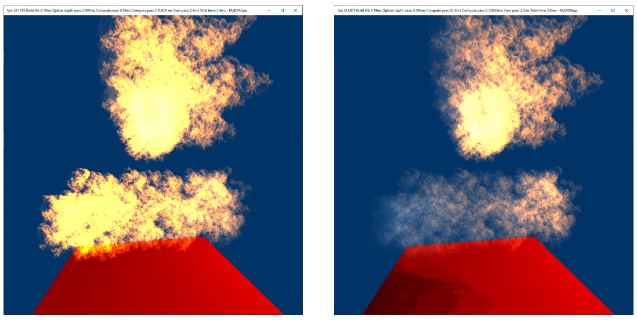
POPULÄRVETENSKAPLIG SAMMANFATTNING **Lukas Mattsson**

Strålspårning är en metod som har använts för att rendera väldigt högkvalitativa bilder, men med lång exekveringstid. Med modern hårdvara är det dock numera möjligt att utföra strålspårning i realtid och i detta arbete utforskar vi hur denna metod kan användas för att rendera mer realistisk rök för användning i datorspel.

För att skapa datorspel med bra inlevelse krävs snygga virtuella miljöer som renderas i realtid med många bilder per sekund. En viktig beståndsdel av många virtuella miljöer är ämnen som rök, dimma och moln och dessa är vanligtvis väldigt svåra att rendera realistiskt på grund av komplexiteten av deras interaktion med ljus i verkligheten. Ljus skingras och absorberas medan det färdas igenom dessa ämnen och begreppet optiskt djup beskriver dämpningen av ljusstyrkan orsakat av detta. Strålspårning, eller *ray tracing* på engelska, är byggt på idén att följa ljusstrålar som kommer ifrån någon ljuskälla för att avgöra vad som ska visas på skärmen. Detta forskningsområde har ökat enormt i populäritet i samband med att moderna grafikprocessorer nu kan utföra strålspårning i realtid via hårdvaruacceleration.

I detta examensarbete utforskar vi användningen av strålspårning för att förbättra utseendet av rök för realtidsapplikationer så som datorspel. Detta görs genom att simulera det optiska djupet i röken. För att beräkna det optiska djupet för en punkt i röken behöver vi veta sträckan som ljuset från ljuskällan har färdats genom röken för att nå denna punkt. Genom att spåra en stråle från punkten mot ljuskällan kan vi ta reda på denna sträcka genom att se när strålen lämnar röken.

För att kunna bedöma resultatet jämför vi de


Figur 1: De två bilderna visar rök med en ljuskälla till höger. I den vänstra bilden är det optiskta djupet inte beräknat och i den högra är det beräknat

renderade bilderna med avseende på utseendet och dessutom mäter vi exekveringstiderna av programmet. Vi renderar nästan identiska bilder där den enda skillnaden är ifall det optiska djupet är beräknat eller inte och använder dessa bilder för att visa förbättringen av utseendet som ges vid medräkningen av det optiska djupet. Vi mäter även exekveringstiderna av de olika komponenterna av renderingsprocessen för att visa den totala prestandan av programmet. Vi uppmäter exekveringstider på mellan 1,70ms och 3,78ms per bild.