Hasselt University

A thesis submitted in partial fulfillment of the requirements for the bachelor's degree in computer science

# Shared Rendering Computation in Cloud Gaming

*Author*:

Jente Vandersanden

*Promoter*:

Dr. Jeroen Put

*Mentor*:

Mr. Hendrik Lievens

Academic Year 2020-2021

# Acknowledgements

First of all, I would like to thank my promoter Dr. Jeroen Put and mentor Mr. Hendrik Lievens for their great guidance and enthusiasm for the topic. Their expertise, tips and examples really inspired me and made it possible to achieve the desired results. It was a great privilege to study under their supervision.

Secondly, I would like to express my special thanks to Prof. Philippe Bekaert for allowing me to write my thesis under his department.

I am also very grateful to all other professors and (assistant) teachers whom I have met during my educational career. Thank you for sharing your expertise, wisdom and inspiring me to keep working on my knowledge and skills. Furthermore, thank you for encouraging me to push my limits and step out of my comfort zone. I have learnt my most important lessons through encountering real challenges.

Another thank you goes out to my best friends, who always took their time to show their interest, go for a walk or entertain me with their banter.

Of course I cannot forget to thank my family, who gave me the chance and trust to pursue this goal. I am very grateful for all the opportunities you have given me in life.

And last but definitely not least, my wonderful girlfriend. Thank you for always supporting me and believing in me no matter what. I am really lucky to have you.

– *Jente*

# Summary


## Introduction

The gaming industry has been continuously evolving throughout history [1]. One of the more recent innovations in gaming infrastructure is the concept of *cloud gaming* [15]. Cloud gaming removes all the workload of rendering and managing a video game away from the client. Instead, it is assigned to a server, which then streams the results to the client. Similarly, but not equally, a *shared rendering computation* infrastructure consists of a server that enhances the client. The difference here is that both server and client perform some form of rendering/managing tasks. This is usually done in such a way that the server gets assigned the heavier tasks, whereas the client performs relatively lightweight tasks. To eliminate redundant work, the server should ideally perform tasks that would otherwise need to be executed by all clients. In this way, the client can save its resources for other tasks that it needs to perform. Even better, a client that does not have the necessary resources at its disposal to render and manage a game individually, is now able to run the game in question.

### Objective

This thesis aims to develop an efficient technique to organize a *Shared Rendering Computation Cloud Gaming Service*. More specifically, the computationally intensive task that is moved to the server is the computation of **diffuse global illumination** in a virtual scene. Global illumination [11] computation is an expensive task to perform in real-time, it requires the rendering application to perform some form of ray tracing, which quickly becomes a heavy task for complex scenes. The main idea is to move this task to the server application, which stores the result in a diffuse global illumination map (a texture), and sends this result over the network to the client application. The client application still has the comparatively simple task of rendering the scene's geometry, and applying the received texture to it. This delivers the final result of a globally illuminated scene at client side, while the client only had to perform the inexpensive task of rendering geometry and mapping a texture to it. In the context of single player games, this means that the client application is spared a lot of work. A more interesting scenario is a multiplayer game. In this case, the diffuse global illumination, that each client has in common, only needs to be calculated once. Since a *diffuse surface* reflects the light into every direction, this means that the perceived light is the same, independent of the viewpoint. This implies that regardless what the viewpoint is, the resulting luminance value for a certain point in the scene is the same, assuming that only the diffuse component of the global illumination is considered. Therefore, a lot of redundant computation can be prevented by centralizing the diffuse global illumination computation on a server, and distributing the result to each client.

Finally, because the time frame of this thesis is fairly limited and the diffuse illumination has a very specific advantage as mentioned above, the implementation made in this thesis will only consider the diffuse component of the global illumination. The specular component, however, is also very important for a realistic result. Although this thesis will not focus on that aspect, adding specular illumination to the implementation can be considered as future work, and should follow the same workflow. Despite this, the diffuse component on its own provides a representative result from which some considerable benefits may be deduced.

### Problem areas

The two main disciplines that define the area of this research are Networking and Computer Vision. Challenges will be faced in both of these fields. A system that runs a video game is continuously performing rendering calculations. In context of this thesis, the server will need to perform the heaviest calculations. Therefore, it would be desirable to develop an efficient graphics pipeline so that the server's rendering operations are as performant as possible. Additionally, the server needs to update the diffuse global illumination whenever a modification that affects the lighting is made to the scene. Every time an update is made, the new diffuse global illumination map needs to be sent to the client. All of this together needs to happen at real-time frame rates, to ensure a good experience. Although this is a problem that is definitely not trivial to solve, this thesis will aim to achieve a working, representative implementation. Any further optimizations can be considered as future work.

## Implementation

The implementation of this thesis consists of two main high-level components: the client and server. This section will first shortly discuss the high-level architecture of the application, after which more details are provided on each component.

### High-level architecture

Given the research problem, a very simplified set of requirements for the resulting application would be:

1. The server application needs to calculate the diffuse global illumination in the scene, and render it into a texture.
2. The client application, that receives a texture into which the diffuse global illumination is 'baked', needs to render the scene's geometry and map this texture to that geometry in order to obtain the resulting diffuse illuminated scene.
3. The application needs to have a certain data flow between the server and client (to transfer the resulting textures).
4. The application needs an interactive method to influence the scene's lighting. (To demonstrate that the application can handle dynamic changes in illumination).
5. The service needs to be available on most platforms with internet and browser access.

In fulfillment of requirement 5, the choice was made to implement the client application in a web browser. Besides providing multi-platform support, it has other benefits, e.g. making the application 'plug-and-play'. No additional software besides the web browser would need to be installed to play the game.

The application's streaming mechanism that is used to stream the diffuse global illumination texture and other arbitrary data between server and client was implemented using the *WebRTC* [34] specification. More specifically, WebRTC's datachannels were used to set up a constant bidirectional arbitrary data stream. Note that, since WebRTC is used to set up peer-to-peer connections, the server is considered as a special peer, to distinguish the clients from the server. However, due to the peers not having information on each other's IP address, the peer-to-peer connection cannot be set up directly. To solve this, an intermediate server (the signaling server) is used to perform a 'handshake process' between the client peer and server peer. For more detailed information about WebRTC, the reader is requested to consult section 2.2.

### Server (*virtual light field rendering*)

The server application will take the responsibility of computing and providing the diffuse global illumination map of the rendered scene. Therefore, a render engine that renders a scene using the theory of *virtual light fields* [32] was implemented. A very brief explanation of the term 'virtual light field' would be: "*The complete set of all light rays in a virtual volume*". However, this is merely a very limited definition of what a virtual light field really is. Since the goal of this summary is to briefly summarize the thesis, the reader is requested to consult section 2.1 for a broader introduction to this topic.

The algorithm (render loop) performed by this engine goes as follows: The scene is surrounded by a virtual, imaginary sphere. A uniformly distributed point $P$ on this sphere is taken. The camera is placed into point $P$, and an orthogonal projection is made towards the origin of the scene, which overlaps with the center of the sphere. The area that is covered by this orthogonal projection is called a *parallel subfield* (PSF). In the image that results from this orthogonal projection, for each pixel a ray is sent out (which results in $n$ x $m$ rays, with $n$ x $m$ being the resolution of the image). To capture color and depth information for more than one depth layer, multisampled framebuffers were used. A visual representation of a *parallel subfield* is presented in figure 1. The next steps in this render loop are performing a technique called *stencil routed a-buffer*, similar to the one proposed in [25]. These steps are executed per PSF ray and are summarized in figure 2. Furthermore, this process is repeated for $k$ uniformly distributed points $P_i$ on the sphere, with $i \in [0; k[$ , and it can possibly be repeated for multiple lighting bounces. The first lighting bounce represents the scene's direct lighting.

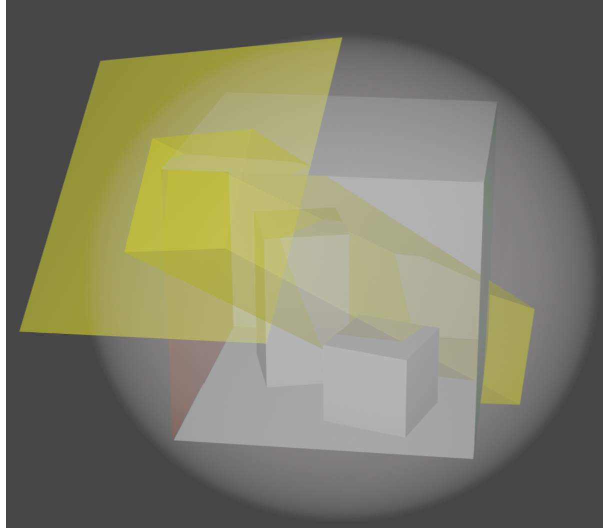


**Figure 1:** Visual representation of a *parallel subfield*. The scene is surrounded by an imaginary sphere, here marked in transparent white. A point $P$ is taken on the sphere, from which an orthogonal projection is made towards the scene's origin. This is marked by a yellow light beam (not all rays were drawn). Note that a PSF covers the whole scene. That is, for each PSF, every point in the scene intersects with exactly one ray of that PSF.

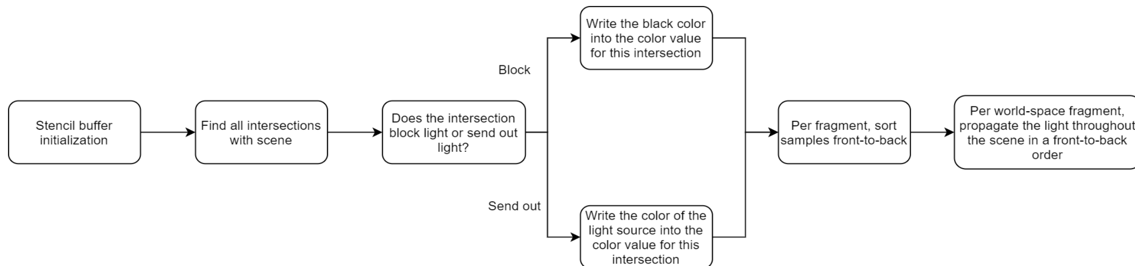



**Figure 2:** Simplified schematic representation of the *stencil routed a-buffer* algorithm, which is executed for each ray in a PSF.

Next, to obtain the final diffuse global illumination, the intermediate results from each viewpoint and lighting bounce need to be combined, since they all contribute to the final result. This is done by performing additive color blending. Additionally, to convert the radiance that was incoming during the current lighting bounce into outgoing radiance for the next lighting bounce (according to the *Bidirectional Reflectance Distribution Function* [6]), an additional step performs multiplicative color blending on the luminance value of each point in the scene and the diffuse color of that point in question. For the purpose of this thesis, this will provide sufficient approximation to the application of a true Lambertian diffuse BRDF while allowing for an easy implementation.

Finally, since the color values have been merged multiple times using additive blending, the resulting image will have a much higher *dynamic range*[1] than the $[0;1]$ range in which *RGBA* tuples (color values) are encoded. This would result in the whole scene being completely white, which would be unacceptable. To solve this, a tone mapping algorithm is used. A tone mapping algorithm, as its name implies, maps the color values of a higher dynamic range back into the $[0;1]$ range so they can be displayed on non-HDR displays. In the implementation of this thesis, the *Reinhard tone mapping* [30] algorithm was used to fulfill this task.

### Client (and client-server communication)

The client side application will be enhanced by the diffuse global illumination map received from the server. In addition to this, it has the relatively lightweight task of rendering the scene geometry. It will map the diffuse global illumination texture to this geometry to retrieve the final illuminated scene. It is worth noting that the server did all of the heavy work, i.e., the global illumination calculations. Additionally, the user of the client application has the option to move cube that is assigned to them by providing keyboard input. This input will trigger an event that sends a movement command to the server, which will then update the position of the cube in question. The diffuse global illumination in the updated scene will be recalculated, and the new texture is sent to the client, so they are able to see the illumination updates.

The client-server communication is managed by a very simple custom protocol. This protocol governs the data that is exchanged between the server and client. There are currently only 3 situations in which it is necessary for the server and client to communicate:

1. Sending the diffuse global illumination texture (server to client)
2. Sending the (new) current positions of the movable cubes (server to client)
3. Sending movement commands to move a cube (client to server)

In situation 1, every time a new texture image becomes available at server side, the server sends it into the datachannel (established between server and client) in the form of a binary byte stream, which is recognized by the client. The end of an incoming image is marked by the following message: `'PNG_TRANSFER_COMPLETE'`.

According to situation 2, the server periodically sends a message to each client in the following format: `'x1!y1/x2!y2'`. `(x1;y1)` and `(x2;y2)` represent the coordinates of cube 1 and 2 respectively (only 2 movable cubes were provided in this implementation).

Finally, according to situation 3, each client gets assigned a movable cube, based on the order in which they joined the session. The commands that they can send to the server are in the following very simple format: `'up'`, `'down'`, `'left'`, `'right'`. The server distinguishes between these commands and updates the position of the cubes according to which client was the sender of the command.

## Results and conclusions

To demonstrate the final resulting implementation, a small experiment was set up in which two clients connect to the server. Each of them gets assigned a movable cube, which they can move through keyboard inputs. The diffuse global illumination in the scene gets updated by the server accordingly. I was able to test two virtual light field configurations:

1. 384 PSF-viewpoints, 2 lighting bounces
2. 1536 PSF-viewpoints, 2 lighting bounces

More lighting bounces / PSF directions could be desirable, depending on the scene. Unfortunately, due to hardware-related constraints I was not able to test with higher configurations.

Throughout the thesis, various parameters have been experimented with. One of these parameters is the texture format. Both the *JPEG* [26] and *PNG* [33] formats have been tried out. There was no great difference in performance between these two, however, there still seemed to be a great bottleneck in the

[1] `https://web.archive.org/web/20150426032033/http://www.electropedia.org/iev/iev.nsf/display?openform&ievref=723-03-11`

encoding/decoding process. Therefore, an alternative method was employed to overcome this bottleneck. This method consists of sending the raw texture data over the network, so no encoding/decoding needs to happen. This managed to improve the performance significantly, however, the final implementation had resort back to the PNG format due to texture mapping problems with raw textures in the front-end framework. This issue could likely be resolved in the future.

Furthermore, it should be mentioned that due to hardware memory limitations, (512 x 512) is the maximum texture resolution that could be used for the diffuse global illumination map. It is also worth noting that the texture resolution has a rather great impact on performance, since it determines the amount of video memory that will be utilized. To obtain a somewhat interactive experience with the hardware setup used for this thesis, I had to resort to (256 x 256) textures, although some static screenshots have been taken at the (512 x 512) resolution.

More importantly in terms of performance, when evaluating the render loop, it was found that the main bottlenecks here are located in the *stencil routing* and *emit radiance* render passes. The *emit radiance* render pass is the render pass that renders to the samples of a multisampled texture. This lower performance could be explained by the fact that this render pass needs to render to 8 samples per fragment, instead of one. Secondly, the *stencil routing* pass is the render pass that sorts the samples in a front-to-back manner and propagates the light throughout the scene. The main overhead here seems to be located in the sorting operation. Two sorting algorithms, namely *bitonic sorting* and a *compare and swap* algorithm have been experimented with. The compare and swap algorithm performed significantly better. The amount of per-fragment sorting operations that need to be completed have a major impact on the performance. Furthermore, alternative methods could also be considered in the future, such as sorting polygons front-to-back instead of samples. This would not result in a 100% accurate solution in every situation, but might be sufficient for this purpose. Because the current implementation does not yet take in account the large amount of redundant work that is being done for dynamic virtual light fields (that is, only the illumination of the dynamic objects should be recomputed), high level optimizations are still achievable as well. Another interesting topic to look into regarding the stencil routing pass is *order-independent transparency* [9] [31].

Evaluating the final implementation (the one used for the experiment), the conclusion is that unfortunately, the resulting application is not yet able to operate at a real-time performance. The abovementioned absence of optimizations is most likely to blame for this. **The employed rendering technique has a comparatively large memory complexity, although this should not be a major issue in a server environment. Furthermore, because no further optimizations for dynamic virtual light fields have been added yet, the render loop currently has a time complexity $O(l \times k)$, with $l$ the amount of lighting bounces and $k$ the amount of PSF-viewpoints**. Nevertheless, it should be emphasized that the aforementioned optimizations could be (a part) of the solution to achieve a real-time performance.

Finally, it would be useful to compare the technique presented in this thesis to other methods. It was found that if the virtual light field method were to be adopted, it would likely not be used to compute direct lighting. Although it does have the capability to do so, alternative techniques (e.g. *shadow mapping* [35]) are more suitable for this task. For indirect lighting purposes, however, this technique stands out much more due to the fact that indirect lighting (e.g. color bleeding, diffuse global illumination) consists of low frequency content. For low frequency content, textures of lower resolutions often suffice. This is advantageous to the virtual light field technique because its memory consumption is scaling up exponentially with higher resolution textures, therefore reducing the overall performance of the application.

# Contents


**1 Introduction** **10**
1.1 Problem areas and challenges . . . 11
1.2 Global illumination . . . 11
1.3 Implications . . . 12

**2 Literature study** **13**
2.1 Virtual Light Fields . . . 13
2.1.1 Introduction . . . 13
2.1.2 *Parallel Subfield* (PSF) generation algorithm . . . 13
2.1.3 Rendering the actual objects into the light field . . . 15
2.1.4 Light field propagation . . . 16
2.1.5 Diffuse objects . . . 17
2.2 WebRTC . . . 18
2.2.1 What is WebRTC . . . 18
2.2.2 Datachannels . . . 18
2.2.3 Connection establishment - Prerequisites . . . 18
2.2.4 Connection establishment - Signaling Server . . . 19
2.2.5 Connection establishment - *Interactive Connectivity Establishment* (ICE) . . . 19
2.2.6 Connection establishment - *Session Traversal Utilities for NAT* (STUN) . . . 20
2.2.7 Traversal Using Relays around NAT (TURN) . . . 21
2.3 Related work . . . 21

**3 Implementation** **22**
3.1 High-level Architecture . . . 22
3.1.1 Architectural design choices . . . 22
3.1.2 Architectural layout . . . 23
3.1.3 WebSockets . . . 24
3.1.4 Limitations . . . 25
3.2 Server (virtual light field rendering) . . . 25
3.2.1 General approach . . . 25
3.2.2 Technologies and Tools . . . 25
3.2.3 Producing a global illumination map . . . 27
3.2.4 *Emit radiance* render pass . . . 30
3.2.5 *Stencil routing* render pass . . . 32
3.2.6 *Geometry texture* render pass . . . 35
3.2.7 *Blend light* render passes . . . 35
3.2.8 Converting irradiance into radiance . . . 36
3.2.9 *BRDF* render pass . . . 36
3.2.10 *Tone mapping* and presentation render pass . . . 36
3.2.11 Argumentation . . . 38
3.2.12 Virtual light fields vs. traditional ray tracing . . . 38
3.3 Client (and client-server communication) . . . 38
3.3.1 General approach . . . 38
3.3.2 Technologies and Tools . . . 39
3.3.3 Communication protocol . . . 39

**4 Results and Conclusions** **41**
4.1 Setup . . . 41
4.2 Texture formats . . . 41
4.3 Texture sizes . . . 42
4.4 Streaming . . . 42
4.5 Render passes . . . 43
4.6 Overall result and learning process . . . 44
4.7 Other methods of global illumination computation . . . 45
4.8 Future work . . . 46
4.8.1 Coherence of data . . . 46
4.8.2 PSF interpolation . . . 46
4.8.3 Sparse resources and resident textures . . . 46
4.8.4 Simplification of geometry . . . 46
4.8.5 Synchronization of lighting . . . 47
4.8.6 Specular reflection . . . 47

**A Nederlandstalige samenvatting** **48**
A.1 Introductie . . . 48
A.1.1 Objectief . . . 48
A.1.2 Probleemgebieden . . . 49
A.2 Implementatie . . . 49
A.2.1 High-level architectuur . . . 49
A.2.2 Server (rendering met virtuele lichtvelden) . . . 49
A.2.3 Client (en client-server communicatie) . . . 51
A.3 Resultaten en conclusies . . . 51

# Chapter 1

# Introduction

The gaming industry is a continuously evolving sector which is mainly driven by software and hardware improvements. As of today, the majority of consumers in this industry (gamers) are playing their games on their own home entertainment systems (game consoles), computers or handheld devices. Throughout history [1], gaming evolved from arcade machines, to mainframe computers, to the first generation of gaming consoles, personal computers and handheld devices. This evolution was mainly possible because of the improvements in hardware (leading to the production of new platforms). In the modern days of gaming, *cloud gaming* [15] is an upcoming phenomenon. It steps away from the concept of the consumer owning the gaming software (the game) and the gaming hardware (console, gaming pc, ...), but instead serves as a provider to the consumer. The *cloud gaming service* provides a library of games, which can be purchased/rented *on demand* by the user. Additionally, it offers the necessary hardware for all these games to run on. The game is installed and run on the provider's infrastructure, and streamed to the end-user over the network. Although it has not been widely adopted yet, with increasing quality of internet connections, and more connectivity worldwide, this can be considered a logical next step in the evolution of gaming.

This thesis aims to develop an efficient technique to organize a *Shared Rendering Computation Cloud Gaming Service*. Presently, relatively high-end gaming is made accessible on many different end systems, including mobile devices. However, there is always a gap between hardware capabilities of these end systems. Several video games are unable to run on lower-end platforms, simply because the computational resources of that system do not satisfy the needs of the heavy graphical application. In an ideal scenario, it would be wonderful if the user of a system would be able to play multiplayer games together with users on different end systems, with possibly different hardware capabilities. Furthermore, more centralization could improve the experience in a multiplayer gaming environment, and the principle of *Cloud gaming* can offer that. The concept of *cloud gaming* is built on top of a server-client architecture. Therefore, it is capable of moving the computational intensive tasks to the server, so the client can run a program (in this case a video game), even if it does not contain the computational resources that are necessary for the program in question. In this way, a client could request a video game on demand, remotely, without needing to install any additional software, or upgrading their hardware. This is how *cloud gaming* mainly operates today. The infrastructure runs a separate game instance for each client, which results in a great amount.

However, in a multiplayer gaming session, many aspects of the game could be common for all the clients. Because of this, it is in principle unnecessary to recalculate them for each client separately. Instead, let us consider the concept that the client might still perform some computations. What if *cloud gaming* was employed in such a way that it only performs the heavy operations for all the clients in a multiplayer gaming session, so that it lowers the workload for each of these clients? That is exactly what this thesis is trying to prove conceptually. The practical implementation of this thesis is not trying to remotely stream an instance of a program running on the server. Instead, the question is: **How can the *cloud gaming* client-server architecture be organized efficiently so that the server can perform the computationally intensive task of calculating the diffuse global illumination in the scene and share the results with each client application?**

## 1.1 Problem areas and challenges

There are two main disciplines defining the area of this problem: Networking and Computer Vision. To begin with, a video game is in principle a collection of rendered 3D (or 2D) scene projection images, rapidly (according to the frame rate) shown on a monitor screen. This implies that the rendering component covers a huge part of what a video game consists of. In line with this thesis, it would be desirable if the server, having access to (multiple) high-end computational resources, could perform the rendering calculations in an efficient way. Secondly, in *cloud gaming's* client-server architecture, the rendered data computed at server side will need to be communicated to the client side over the network.

This makes up the 2 main areas in which several challenges will be faced. For example, since a video game is in principle an interactive, dynamic graphics application, a real-time frame rate is necessary to provide a good experience for the player. Furthermore, if this idea is expanded to online multiplayer games, it is important that dynamic changes in the virtual scene are propagated to each player in a synchronized way (ideally each player should have the exact same global representation of the game scene at each instance of time). This will form a challenge in the way that each player's modification to the scene need to be communicated to the server, which will then need to recompute the diffuse global illumination of the scene accordingly. Finally, the server needs to redistribute the global illumination map to each player so they are able to see the updates that were made. Additionally, in order to preserve a good experience, this should all happen at a real-time frame rate performance. Each of the steps above involve a certain latency, which means that trade offs will need to be made while selecting a certain implementation. It should be noted that, because of the format of this research subject, the implementation consists of different high-level components, each of which can be sophisticated to its limit (if that limit can be found). Since the time frame of this thesis is fairly limited, certain implementation choices need to be made. **The objective of this thesis is to find a representative working setup for the *shared rendering cloud gaming* architecture. This will be done in the form of a proof-of-concept implementation, consisting of a server that computes the diffuse global illumination of a scene, and a client that uses the results of this process to render the same scene.** Although in the end the implementation might not have the most optimal setup possible, improvements on each of the components that define the architecture can definitely have potential for future work.

Another reason why the solution to this problem is not trivial or at least why it has not been widely adopted yet, is that it requires the software (the game itself) to be written in a different way. Traditional games are made using game engines, which is in principle an abstract framework consisting of a lot of libraries (physics, linear algebra, I/O, ...). These game engines facilitate the work of game developers by providing all the abstract tools necessary to create any game, covering up the details (which are often a lot harder to implement). The traditional game engines are made to produce games which will be rendered by the user's end system. However, the approach that this thesis uses, requires the game scene to be partially rendered on the client side (simple render operations, like rendering the geometry), but also on server side (mainly the heavier rendering operations, such as calculating the diffuse global illumination). Both of these combined will be able to produce the final resulting scene. How this division of tasks between the server and client is balanced could vary based on the implementation (and thus per game), however it would be optimal if the server performed all the operations that each client has in common, and the client only executes specific operations which are only relevant for itself. This thesis presents a proof-of-concept implementation, but there is surely room for adaptations. One could even take it further and build an engine on top of this concept which steps away from the traditional game engines and provides tools to make the creation of (multiplayer) server enhanced games easier.

## 1.2 Global illumination

In the research question the term *global illumination* [11] was mentioned, thus it might be useful to denote what global illumination actually means. Global illumination is a group of algorithms which not only consider light that directly comes from light sources, but also light reflected by other surfaces in the scene. This implies that the color of an object is not fully decided by only the position of the camera, the position of the light source and the viewing angle. All the other lighting effects in the scene come into play as well. Theoretically reflections, refractions and shadows are all examples of this. However, in practice, mainly the simulation of reflection between *diffuse* objects (1) or caustics[1](2) are called global illumination. In

[1] https://en.wikipedia.org/wiki/Caustic_(optics)

this thesis, global illumination is defined as (1). Further, **instead of a full global illumination map, only the *diffuse* component of the global illumination will be considered**. This is important to take into account, since additional calculations are necessary to produce a full global illumination radiance value map. However, the addition of the algorithm to calculate the *specular* component can be considered as future work. The workflow presented in this thesis should be reusable for that purpose too, only the implementation details will obviously differ. This thesis opted to start the implementation with the *diffuse* component because a *diffuse* object reflects incoming radiance uniformly in every direction, which will simplify a part of the *radiance propagation* (subsection 2.1.5), more details to be read there. Due to the very limited time frame of this thesis, this simplification was necessary.

## 1.3 Implications

To stimulate interest in this subject, this section will discuss some of the possible implications that a wide adoption of the concepts proposed in this thesis could have. First of all, let us look at a few game genres that could benefit from this architecture. Massively multiplayer online role-playing games (MMORPGs) often consist of tremendously large worlds in which players are scattered all around. Highly populated regions in this virtual world cause a great overlap in calculations being performing on each player's end system. Diffuse global illumination, which is what this thesis focuses on, is one example. In theory, the diffuse global illumination for the world only needs to be calculated once, and can then be distributed to each player, saving their end systems a lot of work.

Additionally, one could consider competitive multiplayer games (e.g. a *First Person Shooter*) in which a centralized global illumination calculation could make a difference in the perception of fairness in the game. For example, imagine a first person shooter in which player $A$ and $B$ are looking at a player $C$ from far away, through a mirror. $C$ initially is in a dark room, so the player is not visible to $A$ and $B$, but after someone turns on the light in the room, the mirror reflects the character of player $C$ towards $A$ and $B$. However, since player $A$ has a better hardware setup (and therefore higher graphical settings) than player $B$, player $A$ can see player $C$, while player $B$ cannot. This is only a conceptual example, but there are more possible situations like this that could result in an unfair experience, which is not desirable and can be very frustrating in a competitive environment. A centralized and synchronized illumination calculation could resolve this issue.

Finally, this architecture might open doors to new genres of games, which have not been invented yet. The games in question would benefit significantly from the centralization of the calculation of global illumination or possibly other aspects that can be centralized at server side. Although I have not been able to come up with a new game genre myself, as mentioned in the beginning of the introduction, the gaming industry constantly evolves, driven by software and hardware improvements. It is often new innovations that lead to creative products exploiting these. The proof-of-concept infrastructure presented in this thesis could be a software improvement that will possibly give new ideas to creators. This would be a great bonus, but just looking at the current games available, there are a variety of genres that might benefit from this infrastructure, as the two paragraphs above briefly showed.

Hopefully this section has made clear what the research question is about, why this is an interesting problem to look into, and why it is not trivial to solve. The next section covers the literature study, which explains several concepts that are crucial to understand if one wants to understand the implementation in detail.

# Chapter 2

# Literature study

## 2.1 Virtual Light Fields

Before talking about the implementation itself, it would be really useful to introduce the principle of *virtual light fields* [32]. Many algorithms and ideas used throughout the thesis have their origin within this concept, so it is necessary for the reader to have a good understanding about this topic to be able to follow along in later sections. In this section, *virtual light fields* will be introduced using a real-life example. In addition to that, the algorithms used to construct a *virtual light field* are also covered.

### 2.1.1 Introduction

To make *virtual light fields* easier to understand, let us first consider physical light fields. Imagine a 2D grid of 20m x 20m, divided into cells of each 1m x 1m (this will give us 400 cells). Also suppose that there is a castle in front of the grid. A simplified version of this experimental setup is illustrated in figure 2.1. Now imagine that a camera is placed in the center of each of these cells, with the viewing direction perpendicular to the grid. Each of these cameras capture an image of the castle in front of the grid. Then this would result in 400 pictures, each with a slightly different view on the castle. Now consider starting in cell (1, 1) and moving to cell (1, 2). The image that is visible from cell (1, 1) is different compared to the image in cell (1, 2), because it has another position relative to the castle. However, these 2 images combined contain enough information to represent the perspectives from both positions. Nevertheless, it should be noted that there is a small gap in between the cameras of each cell, which causes some loss of information (a person could stand in between the cameras and have a different view on the castle than the either one of the cameras have). Despite this, if the space between these cameras is kept minimal (it can never be 0) and interpolation is used between the images produced by each camera, a good approximation of the viewing spectrum of all cells combined for this viewing angle can be obtained. If this would be expanded to all possible angles (again there will be gaps in between different angles, because one can only take a finite amount of samples), a reliable viewing frustrum containing the information for each cell position, and each possible viewing direction can be produced. This can be thought of as having the information to know what the castle looks like from each possible viewing position and viewing angle.

### 2.1.2 *Parallel Subfield* (PSF) generation algorithm

A *virtual light field* is very similar to the real life example mentioned before. Instead of the real world environment, consider a virtual 3D scene, and instead of real cameras, consider virtual camera viewpoints and viewing angles. Another subtle but important difference is that real-life cameras use a perspective projection, while a virtual light field is constructed by a collection of orthogonal projections (parallel light bundles). The *global illumination* information of the scene can then be retrieved by considering each possible viewing position and viewing angle. For virtual approximations, it can be assumed that light moves in a straight line. The virtual light field can be organized by using a sphere and constructing so named *parallel subfields* (PSFs). A parallel subfield is a collection of parallel rays in a certain direction going through the sphere. If all possible parallel subfields are combined, one could say that approximately

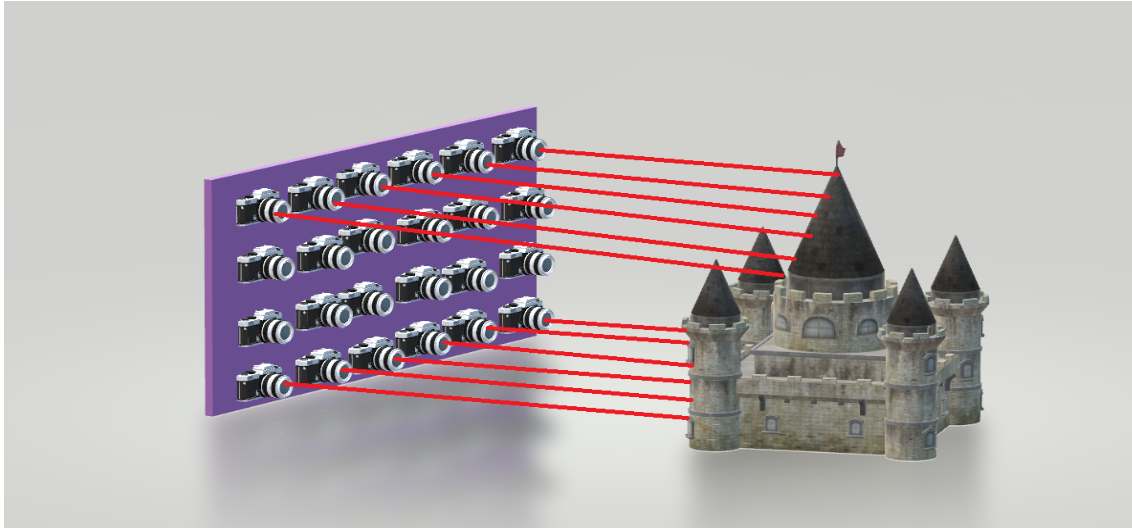


**Figure 2.1:** Model representation of a subfield of a physical light field.

all possible viewing directions and positions for the scene surrounded by the sphere are covered. The algorithm to construct these subfields goes as follows:

1. Take a sphere with origin $O$.
2. Take a uniformly distributed point $P$ on the sphere, which forms vector $\vec{p}$.
3. Construct plane $k$ orthogonal to vector $\vec{p}$, containing $O$. We call $d$ the disc intersection of the sphere with $k$.
4. Divide $d$ into a deterministic pixel grid. Each point on the grid will represent a ray in the subfield.
5. Now shoot rays through these grid points parallel to vector $\vec{p}$.
6. Repeat this procedure for different $P$s, to retrieve the illumination information for the whole scene.

To further explain this, consider a sphere $[-1, 1]^3$ with origin $O$. This sphere contains the whole scene of which the diffuse global illumination needs to be calculated. Assume that a PSF is constructed, according to the algorithm described above. This includes that a point $P$ was chosen on the sphere with origin $O$, and a grid orthogonal to $\vec{p}$ was constructed. The objective is to construct a *virtual light field*. This means that for each PSF, along each ray, throughout all depth layers the luminance information needs to be saved somewhere. To represent the stored luminance information for the *virtual light field*, an abstract data structure $VLF$ is introduced. $VLF$ can be accessed by a 4-dimensional vector $(i, j, u, v)$. In this vector, $(u, v)$ represents the direction of $\vec{p}$, which is nothing more than the *'viewing direction'* of the current PSF. Furthermore, $(i, j)$ represents the ray index in the 2-dimensional deterministic pixel grid that was constructed in step 4. of the algorithm described above. In this way, one can access any arbitrary ray within the *virtual light field*. The 2D pixel grid serves as the base for illuminating the whole scene from one direction with a parallel light bundle. Considering all PSFs, and therefore all parallel light bundles with different directions, this results in a good approximation of the diffuse global illumination in the scene. This is because the illumination was evaluated from a high amount of uniformly distributed directions.

Of course, the amount of rays that can be stored in this data structure is limited and decided by the amount of cells in the 2D-deterministic pixel grid. The more cells this pixel grid contains, the higher the amount of rays that are evaluated. This can thought of as a higher resolution, representing a more dense *virtual light field*. However, with a higher resolution comes a higher memory cost, since more rays will need to be stored in the $VLF$ data structure. This illustrates that global illumination computation through the use of *virtual light fields* is a memory intensive task. The abstract PSF-generation algorithm is visualized in figure 2.2 for 1 PSF.

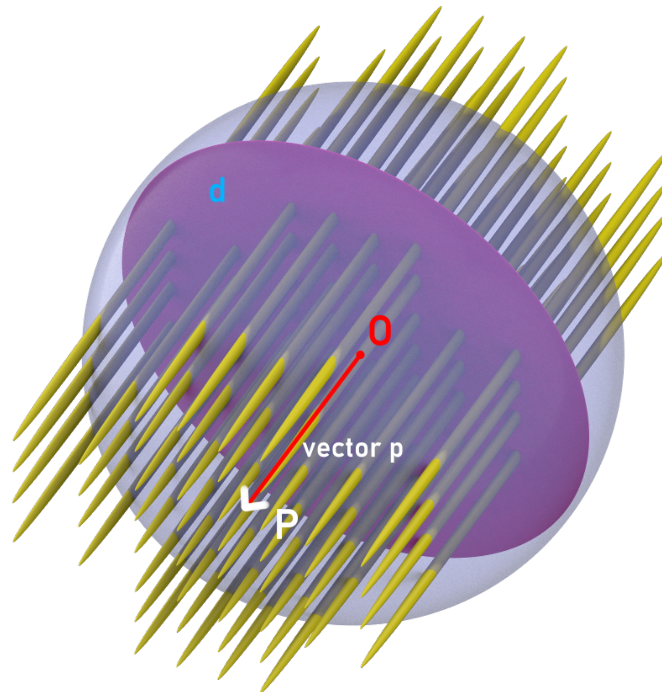



**Figure 2.2:** Visual representation of a possible PSF generation algorithm. Note that only one PSF is presented here. A point $P$ was taken on the sphere with origin $O$. After that, a plane containing $O$ is constructed perpendicular to $\vec{p}$. The disc intersection of this plane with the sphere is called $d$. Then $d$ is subdivided into a pixel grid, through which the (yellow) rays are **orthogonally** projected. These rays together form the PSF. Note that not all rays were drawn in this example.

Hopefully this abstract explanation of the algorithm is making clear how a virtual light field can be constructed, which is in principle nothing more than a collection of luminance information samples for every possible viewpoint in the scene. These information samples were represented by an abstract data structure ($VLF$) in this section, but can in practice be represented as a 2D image (a texture). Compare this to the sphere in figure 2.2, each ray through the sphere represents a pixel in the final texture that we want to obtain. The final texture in question is the diffuse global illumination map. This implies that each pixel will also have to get an initial color value, which later gets influenced by each PSF that casts light on the pixel in question.

### 2.1.3 Rendering the actual objects into the light field

The next step is to use the parallel subfields to render actual objects into the scene. The data structure described in the previous section represents each possible ray out of each possible PSF. Something that needs to be pointed out is that the virtual light field approach does not render the objects themselves. Instead, the **objects illuminate the rays**, and the virtual light field approach **renders the rays**. Think of it as a view independent ray tracer, each possible view position and viewing direction is ray traced. Another way to see this concept is through the process of painting. The scene is 'painted' by illuminated rays from different points of view. Light rays coming from emitters (light sources) will be illuminated and result in a color at their first intersection, whereas others will not be illuminated (those laying behind occluders for example). Therefore, each time a ray goes through a light emitter (along the direction of irradiation) it will leave a 'paint mark' on the scene at its next intersection with an object. Of course, this is a very simplified way to think about the subject, but might help to understand it better. A visual example of the illumination of the scene along these rays is presented in figure 2.3.

To come to the insight of why this can work so efficiently, I refer back to the sphere defined in the section above. To render an object into the scene, one needs to find a sequence of $f$ intersections $0 < f_1 < f_2 < ... < f_n$ for each ray that intersects with this object (since all these intersections together will define our object). Consider the sphere with the PSF parallel to the $z$-axis, finding these $f$ intersections is in principle nothing more than the rasterization of this object, although in this case one

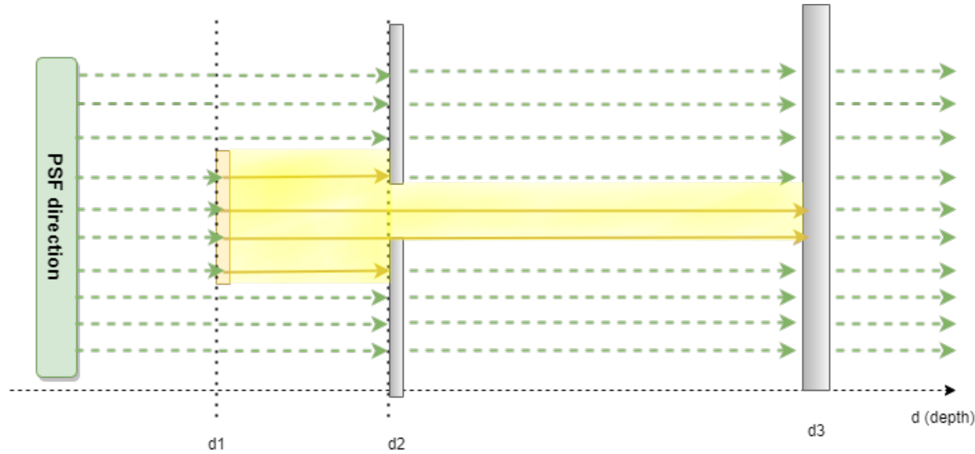



**Figure 2.3:** Emission of radiance throughout different depth layers ($d1$, $d2$, $d3$) of the scene. An *emitter* is represented by a yellow rectangle, while an *occluder* is represented by a gray rectangle. Rays are orthogonally projected in the PSF direction. Rays containing no energy are represented by dotted green arrows, while rays containing energy are represented by full line yellow arrows. Rays gain energy whenever they intersect with an emitter, and lose their energy when they intersect with an occluder.

wants to keep track of all $z$-depths rather than only the one closest to the viewpoint. Rasterization is an operation that can be performed very efficiently by the GPU rasterization hardware. What does need to be taken into account is that the camera through which the scene is rendered iterates over all the PSF-viewpoints. Therefore, the scene is being transformed into the viewing space of the current PSF under consideration. The total cost of the rendering operation of the object is thus approximately the cost of the rasterization operation plus the cost of the *model/view/projection* matrix transformations of this object. It was mentioned before that one wants to keep track of all $f$ intersections. Each of these intersections could contain additional information, like e.g. an object ID, $f$ value of the intersection, direction bit (used to determine the direction of light out of the object), radiance, a diffuse reflection radiance accumulator. The $(i, j, u, v)$ index can be seen as an address that leads to the sequence of intersections belonging to the ray that $(i, j, u, v)$ represents. The first step is thus to render all the objects into the virtual light field, initialising the $VLF$ data structure. After this first step, the *irradiance* values will still be zero. When the initialization is finished, radiance will be propagated throughout the data structure, starting from emitters, as described in the next subsection.

### 2.1.4 Light field propagation

After initializating the data structure, one wants to consider how to propagate radiance information throughout this data structure. The radiance has its origin in some light source, called an *emitter*. Consider an emitting object $L$. The radiance values for each object will be calculated as follows: During the first iteration (the first lighting bounce), all rays that have an intersection $f_n$ with $L$ are found. This can be achieved by using the same method as described in the previous subsection. For each of the rays intersecting with $L$, an attempt is made to find the previous/next intersection, depending on the irradiation direction, name it $f_{rad}$. For $f_{rad}$ one can then set the irradiance and *unshot radiance* values appropriately. The object in question (the object lying at intersection $f_{rad}$) will then be marked as 'containing unshot radiance'. Unshot radiance, as its name implies, is radiance that will be irradiated during the next lighting bounce.

In the next iteration, all objects containing unshot radiance are considered, and the same process that was executed for object $L$ in the beginning is repeated. Object $L$ is marked as 'no longer containing unshot radiance' after this iteration. The iteration process can stop when the unshot radiance falls below a certain threshold value, when the amount of radiance absorbed by the whole system converges, or when a limited amount of lighting bounces was set. Again, to visualize this algorithm, figure 2.4 is provided. Light source $L$ deposits radiance into object 1. This sets a value for the unshot radiance of object 1. In the second bounce, this unshot radiance is reflected onto object 2, setting a value for its unshot radiance. This process can be repeated multiple times, depending on how many lighting bounces the configuration allows.

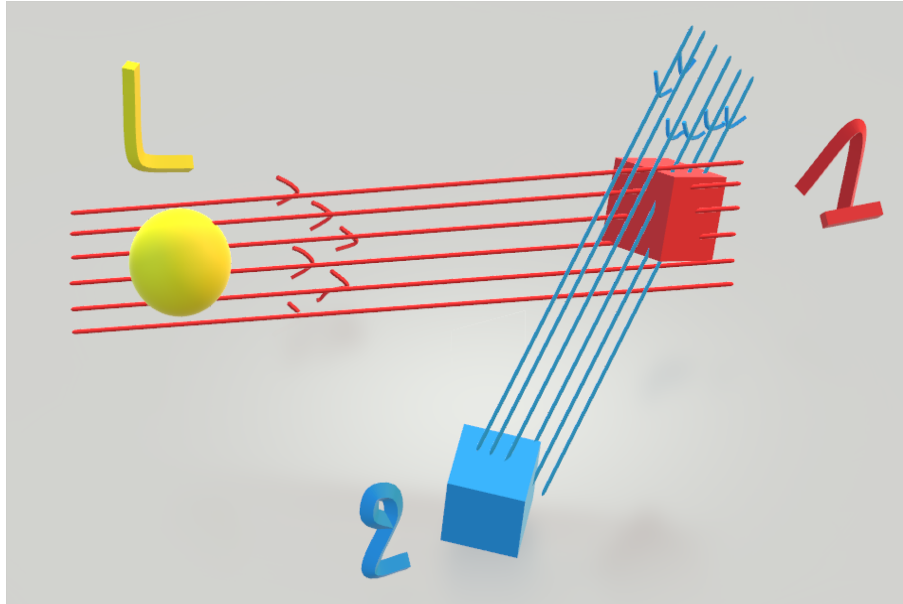



**Figure 2.4:** Simplified visualization of the radiance conversion algorithm. Lighting bounce 1 is marked in red, lighting bounce 2 is marked in blue. During the first, direct illumination phase *unshot radiance* is accumulated in the red cube. Since the red cube contains *unshot radiance* after lighting bounce 1, it will be considered an *emitter* during lighting bounce 2.

### 2.1.5 Diffuse objects

Because this thesis will only cover the *diffuse* component of the global illumination, it might be interesting what the situation considered in the subsection above would look like for *diffuse* objects specifically. Since a *diffuse* object gathers (or *accumulates*) all incoming energy to later reflect this (light) energy in all directions (according to the bidirectional reflectance distribution function ( [6]), this can also be translated to the used data structure. To do this, for each intersection $f_n$, the system keeps track of a diffuse reflection radiance accumulator field (as mentioned above). This field can contain 2 values: an accumulated radiance in that intersection and an iteration cycle number. In the propagation algorithm there are 2 phases for diffuse objects: a *'gathering' phase* and *a 'shooting' phase*. Both phases' names imply what they mean. In the gathering phase, all rays intersecting with the *diffuse* object are considered. For each of these rays, the normal at the intersection point is taken, and the closest ray $r$ in a parallel subfield to this normal is found. The main takeaway here is that because a *diffuse* object is used, the closest ray $r$ to the normal will represent the radiance value (from our data structure) for all outgoing rays in this intersection point. This is because a *diffuse* object reflects incoming radiance in a uniform manner.

Moving on, the $(i, j, u, v)$ value of ray $r$ will then represent all intersections along that ray, thus including the intersection point with the diffuse object. One can then accumulate the unshot irradiance in that intersection point in the data structure. For the shooting phase, the exact same thing will be done, except that one now looks up the value instead of writing it into the data structure. The looked up value is then propagated along the original intersecting ray. It should also explicitly be pointed out that only 2 phases are necessary for the algorithm to work correctly, that is, a 3rd phase is unnecessary for the accumulated radiance to be reset to zero after the shooting phase. This is where the iteration cycle number comes into play. During the gathering phase, for each intersection, the current iteration cycle is compared with the one written into the data structure for that intersection. If they match, the accumulated irradiance can be updated with the incoming value, but the value that is currently in needs to be taken into account. If they do not match, the current accumulated irradiance value in the data structure is from a previous iteration, so it can be overwritten by the new incoming value.

This is merely an example of how to organize the virtual light field data structure. It was used to illustrate the main concepts, but as chapter 3 will show, there are definitely other options possible.

## 2.2 WebRTC

Another concept that the implementation strongly relies on is the *WebRTC* framework. Therefore, to better understand the implementation's design choices and architecture, it is crucial to cover an introduction on the parts of this topic that are relevant for this thesis.

### 2.2.1 What is WebRTC

Briefly said, *Web Real-Time Communication* or better known as WebRTC [34] is an open-source project that consists of a collection of protocols and APIs. It first started as an open-source project released by Google, but gained the attention of many which resulted in the ongoing standardization of the relevant protocols by the *Internet Engineering Task Force*(IETF) and the *World Wide Web Consortium*(W3C). WebRTC provides the infrastructure for audio and video communication, but also arbitrary data transfer between web browsers. This is made possible by allowing direct peer-to-peer connections between clients. One could possibly argue why someone would use WebRTC in contrary to the other available technologies, like WebSockets or *Voice over IP* (VoIP). One of the great advantages is that it is free for use and open-source. Additionally, it is standardized by the W3C and IETF, which gives a good indication that it will have wide support in the future (and already has). As its name suggests, it has a relatively good real-time performance, which is crucial in real-time applications. There are also some disadvantages to WebRTC, which are browser incompatibility, its sensitivity to varying round trip times [19], and no standardization of the video codec that is being used. For the purposes of this thesis, these disadvantages should not form a problem.

### 2.2.2 Datachannels

The WebRTC standard includes an API to send arbitrary data between different peers over a peer-to-peer connection. This is done by using a so called *datachannel* [34]. Datachannels possess a number of characteristics that are highly relevant and beneficial to this thesis, such as the ability to create a custom arbitrary bidirectional data stream. For this research problem, this is somewhat perfect, because we are not constrained by any format that our data should have. This leaves us many open doors to try out, and options to optimize later, which is exactly what we want.

### 2.2.3 Connection establishment - Prerequisites

Before datachannels between any pair of peers can be deployed, one first needs to establish a connection between those peers. This peer in question can be any personal computer connected to the internet *(client peer)*, but also a server *(server peer)*, that in our case will have to distribute calculated data to the client peers. Now, take two arbitrary peers within the internet. In order to find each other, they need to know each other's IP address (location within the internet). Without knowing this information, a direct peer-to-peer connection cannot be set up because the peer simply does not know where to send the data to. In the world wide web (and other networking applications), this is usually solved by the *Domain Name System* (DNS), which translates domain names (e.g. *https://www.google.com*) that are human-readable, into IP addresses that computers can understand and work with more easily. For a server-client approach in which the server is publicly available, this method is indeed suitable, since each public server can be registered in the DNS by a registrar. However, in the case of WebRTC, we want to directly connect peers that could be any computer within the internet, and do not need to be public by any means. One naive solution could be to register any arbitrary host within the internet in the DNS. This solution is naive in one way because the amount of devices connected to the internet was already at around 22 billion [3] at the end of 2018, and is still growing every single day. This makes it a depleting (if not almost impossible) task for the DNS administrators to keep managing the system. Another, more important reason why this solution is very naive are the dynamics of IP addresses. Internet connected devices are assigned an IP address based on the subnet they connect to. If they physically move to another subnet, they will be assigned another IP address accordingly. Now, imagine how many times this happens in a world containing so many mobile devices. This behaviour makes it even harder for the DNS to deal with. That is why the creators of WebRTC had to come up with another workaround for direct peer-to-peer connections. This workaround is in the form of a *signaling server*.

### 2.2.4 Connection establishment - Signaling Server

The signaling server [4] can be thought of as a necessary middleman to establish the connection between peers. Simply said, it introduces the peers to each other, and serves as a transport mechanism for messages between the peers before any direct connection between them is set up. This can indeed resolve the problem of how to connect two arbitrary peers that do not know each other's IP address, given the fact that they do know the IP address of the signaling server. This means that the signaling server needs to be directly and publicly available, otherwise the problem of peers not knowing each other's IP address would be recurring. However, it should be mentioned that WebRTC does not specify or provide a transport mechanism to handle the signaling information (the messages between the signaling server and the peers). This is left open for the implementer to choose. The exchange of this signaling information between the signaling server and the peers can be seen as a certain handshake between the two peers trying to interconnect. This handshaking process is illustrated in figure 2.5 .

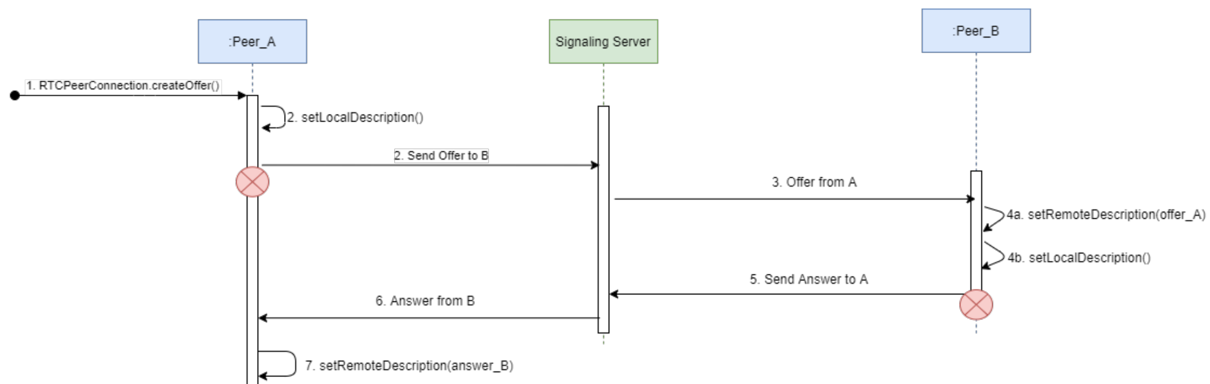



**Figure 2.5:** Simplified representation of the exchange of signaling information between peers during the handshaking process.

The signaling server has a list of connected peers which each have their ID. All of these peers are possible candidates to send (indirect) messages to. First of all, when a peer $A$ wants to connect directly with another peer $B$, it needs to connect to the signaling server. Once it is connected, it decides to create an *offer* for peer $B$. Note that the offer is made by the peer that wants to initiate a connection. The complement of an offer is an *answer*. These offer and answer messages are exchanged between the peers so they learn about each other's session configuration. This session configuration describes the peer-to-peer connection that is about to be set up. The messages exchanged are conform to the *Session Description Protocol* (SDP). An SDP message contains session details, like the IP address of the sender, the ID of the peer that we want to connect to, the codec to be used (in case of a media stream), and so on. The clue is that both peers need to agree on which session parameters they want to use, because they will need to dispose of the same information in order for the direct connection to be established. The calls to 'setLocalDescription()' and 'setRemoteDescription()' set the configuration parameters for the local peer and the remote peer respectively. It is important to note that these offer and answer messages are routed through the signaling server, since there is no existing direct connection between peers $A$ and $B$ yet.

### 2.2.5 Connection establishment - *Interactive Connectivity Establishment* (ICE)

Even when both peers obtained session details from each other, a direct connection between them still cannot be established because there is no guaranteed information on the topology layout of each peer in a pair of peers. Therefore, before a direct connection can be set up, peer $A$ needs to let peer $B$ know which possibilities for direct communication it has. Using more specific terms, one of these options is called an *ICE candidate* [2]. Each ICE candidate is considered to define the best option that that peer has available, according to the ICE policy (see below). Before the decision can be made of which candidate to choose, both peers need to know each other's ICE candidates. This is why they first exchange their ICE candidates, so they can agree on a mutually-compatible candidate. The chosen candidate's session description is then used to open a (direct) peer-to-peer connection, through which data can start to flow. The ICE policy is as follows (high priority first):

1. A direct UDP connection between peer $A$ and peer $B$. A STUN server (see 2.2.6) is then used to find the address of the opposing peer.
2. Direct TCP connection (via HTTP port)
3. Direct TCP connection (via HTTPS port)
4. Indirect connection via a TURN (section 2.2.7) server, when a direct connection fails or a firewall is blocking NAT traversal.

This policy shows that one strives for as least overhead and as much privacy as possible. Finally, it should be mentioned that within each peer there is an *ICE layer* operating, which exchanges candidates conform to the ICE protocol. In figure 2.5, the red crossed dots indicate the start of operation for the ICE layer of each peer respectively.

### 2.2.6 Connection establishment - *Session Traversal Utilities for NAT* (STUN)

Since we are operating in the field of peer-to-peer connections, chances are great that many of these peers reside in home networks. These home networks are often connected to a *Network Address Translation* (NAT) router. A NAT router translates a local IP address of a device in a local subnet to a certain public IP address that can be addressed by the rest of the internet. As a consequence of this, all devices within the same local subnet appear to have the translated public IP address to the rest of the internet. Additionally, the devices within the local subnet do not know which public IP address they get assigned. However, as mentioned in 2.2.3 and 2.2.4, a peer needs to know its IP address in order to insert it into an offer/answer message to let the other peer know. This is where *STUN* serves a solution. A STUN server is a server that a peer can contact to track down its own public IP address, hence to overcome the disadvantage of network address translators. This is a form of NAT traversal. STUN makes use of a light-weight client-server protocol, of which the details will not be discussed, because they are not relevant to this thesis. The important service that the STUN server offers is a query-response infrastructure which peers can use to learn about their public IP address. A simplified schematic representation is shown in figure 2.6. Note that both ICE and STUN are needed to make the whole system work. ICE simply offers a policy, which indicates what kind of connection should be instantiated. On the other hand, STUN is only used in number 1. of the ICE policy, and it ensures that NAT does not introduce errors into the system, which is an important difference.

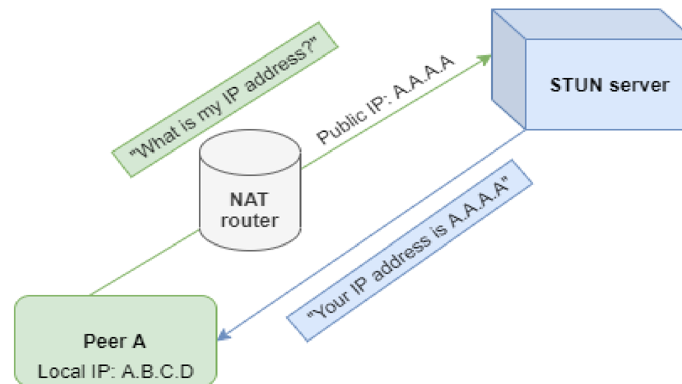



**Figure 2.6:** Simplified representation of communication between a STUN server and client.

### 2.2.7 Traversal Using Relays around NAT (TURN)

As mentioned in the ICE policy declaration in section 2.2.5, sometimes it is impossible to set up a direct connection between 2 peers, because of a failing connection, router restrictions or a firewall blocking NAT traversal. One of these possible router restrictions is called 'Symmetric NAT', which means that the router will only accept connections from peers that you have previously connected to. TURN is meant to bypass this restriction (as well as other inconveniences), and this is done by relaying all (normally direct) traffic between the peers through a TURN server. Note that this results in losing the 'direct connection' characteristic between two peers. Very simplified, the TURN server is just a intermediary node that is publicly available to the peers (available on an IP address that is known by the peer at the start of the connection setup). This will ensure that the destination IP address that a connection was opened to (the IP address of the TURN server) is the same as the source IP address of an incoming connection, resulting the 'Symmetric NAT' restriction is overcome. This is illustrated in figure 2.7. As one can expect, using a TURN server comes with quite some overhead, which is why it is only used when there is no other option available (according to the ICE policy). For the purposes and experimental character of this thesis, it was not necessary to make use of a TURN server, although it would be crucial to use one in a real-world deployment scenario. That is why it is briefly mentioned here.

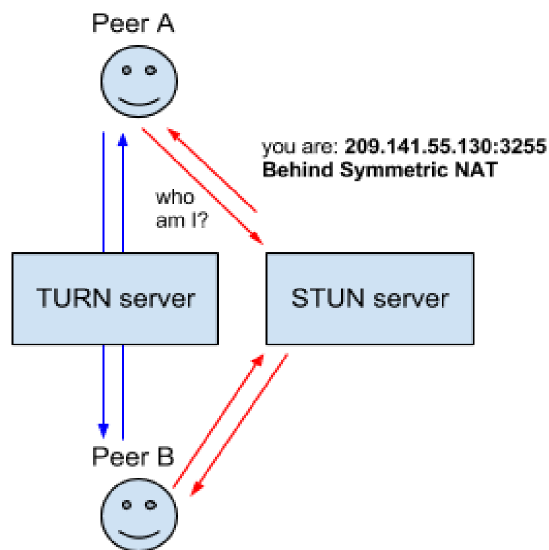



**Figure 2.7:** Simplified representation the STUN and TURN infrastructure combined.
*Source: Mozilla Developer - Introduction to WebRTC protocols*

## 2.3 Related work

In the final section of this chapter I want to mention several works that are related to these problem areas. These artifacts were of great help to me, getting introduced to the subjects of *Virtual Light Fields* (VLFs) and *shared rendering computation* in general. The ideas and implementation of this thesis were also inspired by several results of these works. Two great works which gave me a better understanding on VLFs and how to theoretically construct them are [32] and [24]. Another great inspiration to me was the Master's thesis of my promoter, *Dr. Put* [29]. For the networking part of this research problem, [22] gave a good insight on how I could possibly organize the application's high-level architecture.

# Chapter 3

# Implementation

The implementation, just like the *cloud gaming* architecture, consists of 2 main components: The client and the server. Two sections in this chapter are dedicated to these components. Both will discuss the general approach, more details on the used algorithms and how they were implemented, but also the used technologies and tools to create both applications.

## 3.1 High-level Architecture

### 3.1.1 Architectural design choices

Before moving on to the implementation details of the main components, it would be useful to first discuss the overall high-level architecture that was created as an attempt to tackle this research problem. In order to come up with an architecture, one first needs to think about what the requirements and constraints are, given a certain problem. In the particular case of this thesis, the objective is to render a scene onto a texture at server side, then send this texture to the client side to use it to enhance the scene. Additionally, two dynamic (movable) cubes will be added to the scene, each to be controlled by one of the clients. This will demonstrate that the implementation can respond to a dynamic changes in the scene (because a game's scene is always dynamic). A very simplified set of requirements then would be:

1. The server application needs to calculate the diffuse global illumination in the scene, and render it into a texture.
2. The client application, that receives a texture into which the diffuse global illumination is 'baked', needs to render the scene's geometry and map this texture to that geometry in order to obtain the resulting diffuse illuminated scene.
3. The application needs to have a certain data flow between the server and client (to transfer the resulting textures).
4. The application needs an interactive method to influence the scene's lighting. (To demonstrate that the application can handle dynamic changes in illumination).
5. The service needs to be available on most platforms with internet and browser access.

Similarly, a very simplified set of constraints would be:

1. The client has limited computing resources available and can therefore not perform intense lighting calculations.
2. There is a time constraint which is of critical interest to the experience of the user. In order to maintain an acceptable frame rate, the data flow between the server and the client needs to have a minimal latency. Additionally, the server's render engine needs to be able to calculate the diffuse global illumination at real-time frame rates.

Given this (very simplified) set of requirements and constraints, the first major decision made was to implement the client side in a web browser. A web browser is an ideal candidate for multi-platform

applications, and that is exactly what this thesis is aiming for. Another benefit that this choice brings is that it is not necessary to download or install any additional software, which makes it very accessible and appealing. Looking at constraint number 2, a sophisticated data transfer mechanism which can transfer images over the network consistently and with as low latency as possible would need to be implemented. Given the fact that a web browser was chosen as the client, and considering the technologies that already exist, two viable options remained: *WebSockets* and *WebRTC*. The networking and data transfer of the application was eventually implemented using the WebRTC framework (for an introduction to WebRTC, first read 2.2.) WebRTC was chosen over WebSockets because the application needs to be able to cope with a constant bidirectional stream of relatively large data objects between client and server. WebRTC's datachannels are very suitable for this task. However, it does need to be mentioned that in order to use WebRTC, minimal use of WebSockets was made as well, to support the 'handshake process' between peers (see 3.1.3). WebRTC makes use of peer-to-peer connections, which might seem questionable in a server-client architecture. We can still obtain a server-client architecture by perceiving the server as a separate peer. The rest basically stays the same, client peers just communicate with the server peer over direct peer-to-peer connections.

### 3.1.2 Architectural layout

With the design choices out of the way, a simplified representation of the architecture is presented in figure 3.1. Besides the components, the figure also shows the basic steps in order to set up a peer connection between the server peer and a client peer. As mentioned in 2.2, the architecture also needs to make use of a signaling server and STUN server to be able to create a peer-to-peer connection. In this thesis' implementation, the signaling server was implemented in a *Node.js* script. Additionally, a public STUN server provided by *Google* was used.

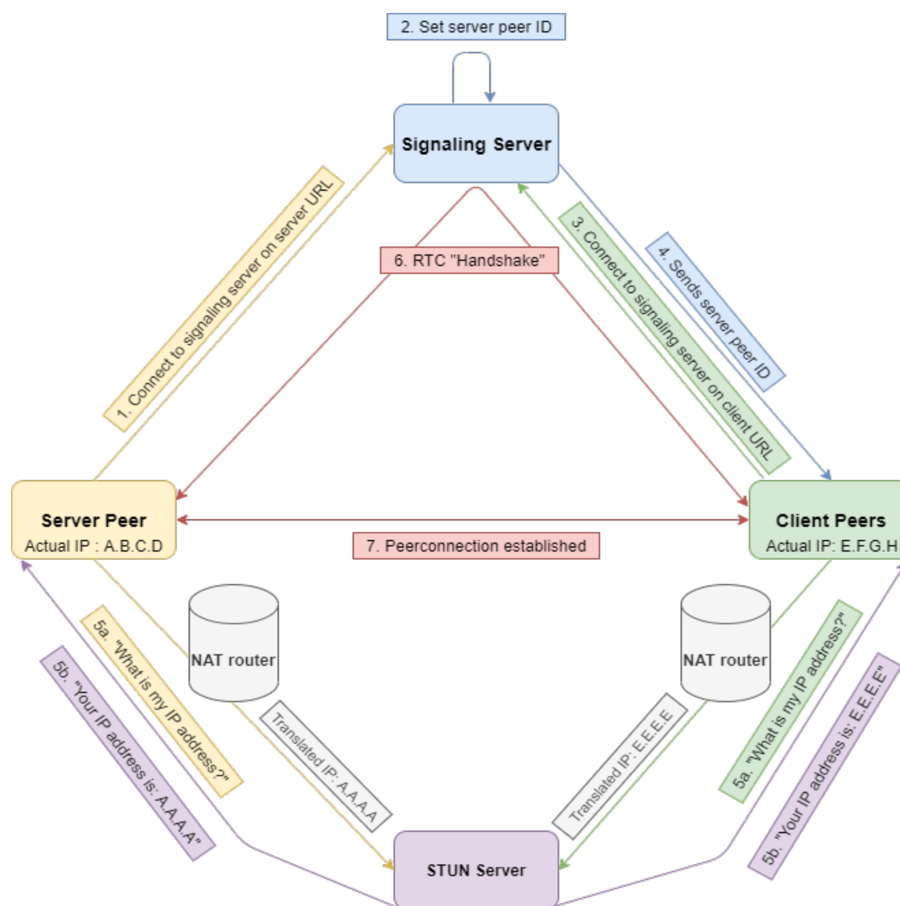



**Figure 3.1:** Simplified representation of the application architecture

Referring to the tags on each arrow in figure 3.1, I would like to explain each step of the connection establishment between a client peer and the server peer in more detail.

1. First of all, the server peer will need to let the signaling server know that it exists. It does so by connecting to the signaling server on a special server URL, so the signaling server can distinguish the server peer from other peers trying to connect. Without knowledge of the server peer, the signaling server cannot serve as a middle man between a server peer and client peers that are trying to connect to the server peer. It should also be noted that the special server URL currently forms a vulnerability, because any peer could pretend to be the server and connect to that URL. Since this is merely a proof-of-concept implementation, it is no great issue for the moment, but it should be kept in mind for the future.
2. The signaling server then notices that a peer is trying to connect on the special server URL, and saves the ID of this peer as a local variable.
3. A client application started running and connects with the signaling server on the client URL.
4. The signaling server notices that the peer is trying to connect on the client URL. It then saves the client's ID in its table of connected peers. Since it is a normal client trying to connect, the signaling server 'expects' that the client will later want to try to set up a peer-to-peer connection with the server peer. Therefore, it sends the server peer ID as a response to the client peer.
5. In the meantime, both the client and the server have been trying to connect with the provided STUN server in order to obtain their translated IP address. The peer in question initially has no knowledge over its translated address, but this knowledge is needed before a peer connection can be offered to another peer.
6. Once the client decides that it wants to do so, it can offer a peer connection to the server (since it knows the server peer's ID from step 4). Note that this offer first has to pass via the signaling server, since there is no direct connection between the server peer and client peer yet. This offer is the beginning of the handshake process, of which an example log is shown in figure 3.2. All the messages interchanged during this handshake process have to pass the signaling server.
7. Finally, a **direct** peer-to-peer connection between the client peer and the server peer is established. The client peer can now directly send messages to the server peer and vice versa, without making use of the signaling server anymore.


```
Client j7xN << {"id":"x9Bo","type":"offer","description":"v=0\r\no=- 6480162383686817019 2 IN IP4 127.0.0.1\r\ns=-\r\nt=0 0\r\na=group:BUNDLE 0\r\na=extmap-allow-mixed\r\na=msid-semantic: WMS\r\nm=application 9 UDP/DTLS/SCTP webrtc-datac
hannel\r\nc=IN IP4 0.0.0.0\r\na=ice-ufrag:u0u/\r\na=ice-pwd:FF9Kf/hZJO2c2ckQn/LMDU01\r\na=ice-options:trickle\r\na=fingerprint:sha-256 6C:63:2A:25:58:1A:38:77:23:93:64:14:6E:FF:F7:10:CD:8F:3D:4A:E8:A8:8D:A8:68:0B:DF:0B:46:F9:D5:93\r\na=s
etup:actpass\r\na=mid:0\r\na=sctp-port:5000\r\na=max-message-size:262144\r\n"}
Client x9Bo >> {"id":"j7xN","type":"offer","description":"v=0\r\no=- 6480162383686817019 2 IN IP4 127.0.0.1\r\ns=-\r\nt=0 0\r\na=group:BUNDLE 0\r\na=extmap-allow-mixed\r\na=msid-semantic: WMS\r\nm=application 9 UDP/DTLS/SCTP webrtc-datac
hannel\r\nc=IN IP4 0.0.0.0\r\na=ice-ufrag:u0u/\r\na=ice-pwd:FF9Kf/hZJO2c2ckQn/LMDU01\r\na=ice-options:trickle\r\na=fingerprint:sha-256 6C:63:2A:25:58:1A:38:77:23:93:64:14:6E:FF:F7:10:CD:8F:3D:4A:E8:A8:8D:A8:68:0B:DF:0B:46:F9:D5:93\r\na=s
etup:actpass\r\na=mid:0\r\na=sctp-port:5000\r\na=max-message-size:262144\r\n"}
Client j7xN << {"id":"x9Bo","type":"candidate","candidate":"candidate:3404171324 1 udp 2113937151 f6472ffb-be2b-4e7e-ba24-1ae6ba31ce9e.local 61232 typ host generation 0 ufrag u0u/ network-cost 999","mid":"0"}
Client x9Bo >> {"id":"j7xN","type":"candidate","candidate":"candidate:3404171324 1 udp 2113937151 f6472ffb-be2b-4e7e-ba24-1ae6ba31ce9e.local 61232 typ host generation 0 ufrag u0u/ network-cost 999","mid":"0"}
Client j7xN << {"id":"x9Bo","type":"candidate","candidate":"candidate:1852373585 1 udp 2113939711 e205cf1d-e7a8-4203-90ce-3b130cc12ba4.local 61233 typ host generation 0 ufrag u0u/ network-cost 999","mid":"0"}
Client x9Bo >> {"id":"j7xN","type":"candidate","candidate":"candidate:1852373585 1 udp 2113939711 e205cf1d-e7a8-4203-90ce-3b130cc12ba4.local 61233 typ host generation 0 ufrag u0u/ network-cost 999","mid":"0"}
Client j7xN << {"id":"x9Bo","type":"candidate","candidate":"candidate:842163049 1 udp 1677729535 84.192.233.149 61232 typ srflx raddr 0.0.0.0 rport 0 generation 0 ufrag u0u/ network-cost 999","mid":"0"}
Client x9Bo >> {"id":"j7xN","type":"candidate","candidate":"candidate:842163049 1 udp 1677729535 84.192.233.149 61232 typ srflx raddr 0.0.0.0 rport 0 generation 0 ufrag u0u/ network-cost 999","mid":"0"}
Client x9Bo << {"description":"v=0\r\no=rtc 1754750276 0 IN IP4 127.0.0.1\r\ns=-\r\nt=0 0\r\na=group:BUNDLE 0\r\na=msid-semantic:WMS *\r\na=setup:active\r\na=ice-ufrag:ceyb\r\na=ice-pwd:rnkzM4szMp4YZl9IzPGs2e\r\na=ice-options:trickle\r\n
a=fingerprint:sha-256 DE:34:F0:D3:99:DE:D4:7D:49:FB:85:A2:30:DE:97:48:0C:15:6B:A8:EE:D6:4A:C0:C6:D9:23:30:6B:17:81:E9\r\nm=application 9 UDP/DTLS/SCTP webrtc-datachannel\r\nc=IN IP4 0.0.0.0\r\na=bundle-only\r\na=mid:0\r\na=sendrecv\r\na=
ice-options:trickle\r\na=sctp-port:5000\r\na=max-message-size:262144\r\n","id":"j7xN","type":"answer"}
Client j7xN >> {"description":"v=0\r\no=rtc 1754750276 0 IN IP4 127.0.0.1\r\ns=-\r\nt=0 0\r\na=group:BUNDLE 0\r\na=msid-semantic:WMS *\r\na=setup:active\r\na=ice-ufrag:ceyb\r\na=ice-pwd:rnkzM4szMp4YZl9IzPGs2e\r\na=ice-options:trickle\r\n
a=fingerprint:sha-256 DE:34:F0:D3:99:DE:D4:7D:49:FB:85:A2:30:DE:97:48:0C:15:6B:A8:EE:D6:4A:C0:C6:D9:23:30:6B:17:81:E9\r\nm=application 9 UDP/DTLS/SCTP webrtc-datachannel\r\nc=IN IP4 0.0.0.0\r\na=bundle-only\r\na=mid:0\r\na=sendrecv\r\na=
ice-options:trickle\r\na=sctp-port:5000\r\na=max-message-size:262144\r\n","id":"x9Bo","type":"answer"}
```


**Figure 3.2:** Example log of the handshaking process at the signaling server

### 3.1.3 WebSockets

Given all this information, it has not been clarified in detail yet how the connection between the signaling server and the peers is established. This is where WebSockets come into play. Since WebRTC does not specify or provide a transport mechanism for the signaling information during the handshake process, it is up to the creator of the application to implement this in a certain way. This thesis chose to do this by using WebSockets, given the fact that it is already widely used and approved, and many examples recommended implementing the signaling server communication with WebSockets. Another small advantage that comes with it is that the syntax is very similar to the WebRTC syntax for setting up a connection. The implementation itself is fairly simple: A WebSocket is opened from a client/server peer to the signaling server, using the signaling server's URL. As mentioned in the explanation of figure 3.1, a server peer has a special server URL to connect to, so the signaling server can identify that peer as a server peer. Once this connection is opened, messages can flow from the signaling server to the client/server peer and vice versa. There is an event listener callback function that can be given a custom implementation by the application programmer. In this case the peer keeps listening for *peer connection* offers, answers to these offers, or ICE candidates (explained in 2.2).

### 3.1.4 Limitations

**Hardware**

First of all, it should be mentioned that this thesis required to be implemented and tested on personal hardware, no additional hardware was granted. The used hardware has the following specifications:

1. CPU: *Intel Core i7-5820K (3.30 GHz)*
2. GPU: *NVIDIA GeForce GTX 970 4G (MSI manifactured)*
3. RAM: *16GB DDR4-2133MHz*

Therefore, all results that will be presented in this thesis will come from tests performed on this system. Since hardware (mainly the GPU) has gained very much power since this system was built, performance could likely be increased by simply upgrading the hardware. Furthermore, this is a proof-of-concept implementation that in practice should be implemented on a server environment with many powerful resources. This hardware limitation should be kept in mind when evaluating the performance results of this thesis.

**Time frame**

Another aspect that should be mentioned is the limited time frame. Since the application that this concept aims at are video games, a real-time frame rate is important. As will become clear in the implementation, it is not evident to obtain high frame rates for **dynamic scenes**, due to the nature of the render loop (it consists of many render passes). However, since the time available for this thesis was so limited, there was barely any time left after creating a working, representative implementation. Therefore, it should be noted that the current implementation is not yet optimized for dynamic scenes. These optimizations can be considered as future work. Additionally, the limited time frame is also the reason why only the diffuse component of global illumination has been implemented. However, specular illumination is also very important in order to obtain a realistic lighting in the scene. The implementation details of the specular component will differ from the diffuse implementation, however, the same workflow can be used. That is why the specular component can also be thought of as future work.

## 3.2 Server (virtual light field rendering)

### 3.2.1 General approach

The server application consists of the following high-level components: The *renderer*, which will calculate the diffuse global illumination of the scene in question, and the *connection handler*, which will as its name implies handle everything that has to do with communicating with and connecting to the client application. The main purpose of the server as a component is thus to produce a diffuse global illumination map (using the *renderer*), and sending it to the client (using the *connection handler*). This thesis opted to implement the renderer using the concepts of *virtual light fields* and *light field propagation* [24] [32], theoretically described in section 2.1. The resulting diffuse global illumination map can then later be used on the client side in combination with the scene geometry to deliver the final result. This section will mainly cover the details on the *renderer*. More details on the networking aspect were covered in the previous section, and will be further covered in section 3.3.

### 3.2.2 Technologies and Tools

This section will cover the used technologies and tools (programming language, libraries, etc.) for the server-side application.

**C++ and Vulkan**

*Vulkan* [14] is a high performance next generation low level graphics and compute API developed by the Khronos Group. It can be thought of as the successor to the *OpenGL* graphics API. Both, however, are widely used. Vulkan has far less overhead and allows the implementer much more control over the application than other APIs like OpenGL since it is much lower level. Like OpenGL, it supports multiple operating systems (Windows, Linux, Android, third-party support for iOS and macOS). The main reason why this thesis opted for Vulkan, however, is the support for multi-GPU programming. After the thesis'

limited time frame has ended, I intend to continue working on this topic, and since the rendering software will be hosted on a server, that server is likely to have a large pool of resources. Therefore, it would be desirable if that pool of resources could be used to its full potential. There are other graphics APIs that support this, but since I already had some (although little) experience with OpenGL, Vulkan seemed to be the obvious alternative. Additionally, since Vulkan is relatively new, but much more performant than OpenGL when used correctly, it is expected to be the future for graphics programming. OpenGL, however, is still being widely used as well, and is likely to remain so for a long time. The main reason behind this is is the high cost of switching technologies within a development team that has accumulated a lot of experience with the previous technology. Since this thesis is starting from scratch, Vulkan is a justified choice.

Secondly, Vulkan is written in C, therefore it being the default language for development. However, C++ appears to be the most widely used language in the community (and industry). Another factor that played a role is the fact that I had more experience with C++ than C. For these reasons, C++ was chosen as the used programming language for the back-end renderer implementation.

#### Used libraries

Several libraries were used to facilitate the implementation process. The first one is *OpenGL Mathematics (GLM)*[1]. GLM is an open-source, header-only C++ mathematics library based on the *OpenGL Shading Language (GLSL)* [13] specifications. GLM can be used to perform matrix and vector operations.

Another library that was useful was *tinyobjloader*[2]. This library was used to facilitate loading in OBJ models into the scene. Additionally, the *stb image*[3] library was used to load in texture images from secondary storage. Finally, *GLFW*[4] was used to facilitate application window creation, surface creation and manage input events.

#### NVIDIA Nsight

A very useful, modern tool that has been of great help as well is *NVIDIA Nsight*[5]. NVIDIA Nsight is a graphical development environment that allows developers to run, analyze and profile their graphics applications. It has a number of really useful features, such as the ability to examine each render pass separately, the ability to view the content of each texture in the framebuffer, and the ability to show the information for each pipeline stage in detail. This tool has been of great help, in the sense that it has saved a lot of time during debugging. It allows a step-by-step evaluation of render passes that are not intuitive. Furthermore, it also gives the implementer a good idea of the performance of their application, and where the bottlenecks may be located.

#### Blender (UV unwrapping)

Another aspect that should be mentioned is the method that was used for texture unwrapping. Since this thesis makes use of a relative simple scene, a manual texture unwrapping would be possible. However, to save time, the 3D modeling application *Blender* was used. It supports UV unwrapping and modification of any model brought into the scene. The resulting unwrapped UV map (the scene's diffuse texture) of the scene used for this thesis is presented in figure 3.3. For greater, more complex scenes, this method is not ideal. If the world scene is dynamic (the world is reshaped/parts of the world are changed removed by interaction), this method would even fail to deliver a correct result. For these use cases a more sophisticated method is obligatory. A solution for this would be to implement a UV unwrapping algorithm within the rendering application. These algorithms, however, are quite complex on their own and do not fall within the scope of this thesis, which is attempting to prove the concept. Therefore, this thesis made use of Blender to complete this task. Besides UV unwrapping, Blender supports numerous other functionalities, but since they are irrelevant to this thesis they will not be covered here.

[1] `https://github.com/g-truc/glm`
[2] `https://github.com/tinyobjloader/tinyobjloader`
[3] `https://github.com/nothings/stb`
[4] `https://www.glfw.org/`
[5] `https://developer.nvidia.com/nsight-graphics`

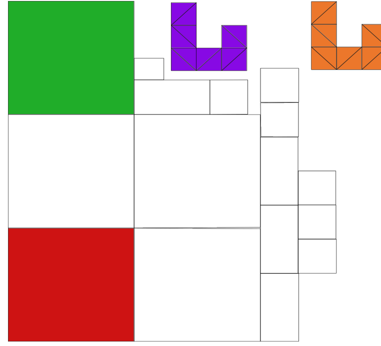


**Figure 3.3:** Unwrapped diffuse texture of the Cornell box scene and 2 movable cubes.

### 3.2.3 Producing a global illumination map

#### Preproccessing phase

The renderer's main goal is to obtain a diffuse global illumination map of the scene in question. The process of producing the diffuse global illumination map consists of multiple steps. This section will shortly describe the necessary steps from a high-level point of view, after which a more detailed explaination of each step will be given. First of all, the execution of the renderer can be divided into 2 big parts: **Preprocessing and processing**. During the preprocessing phase, all the dependencies that the processing phase is depending on are taken care of. The preprocessing phase is only executed once, at the start of the program. This means that it would not be a great problem if any relatively high-cost operations need to happen here, since they are only executed once at startup time. Although it would be a very nice addition if the startup times were low, the first priority is to keep the processing stage as efficient as possible, since this stage needs to be executed every frame.

To briefly sum up what the preprocessing phase consists of: it initializes the renderer's infrastructure and prepares the data needed in the processing phase. First of all, the renderer will need to walk through multiple render passes in order to come to its final result, as will be described in more detail in the sections down below. In Vulkan, each of these render passes are recorded into a command buffer, and they all have their corresponding pipeline, framebuffer objects (FBOs) and descriptor sets (which bind and provide resources to the shader stages). All these resources need to be allocated and initialized accordingly. This is the initialization of the pipelines, which is done in the preprocessing phase. The scene also has to be loaded, which corresponds with loading the models. The last operation which is performed in the preprocessing phase, more related to our VLF implementation and less to the renderer's infrastructure, is the generation of PSFs (see section 2.1.2). A schematic overview of the preprocessing phase is given in figure 3.4.

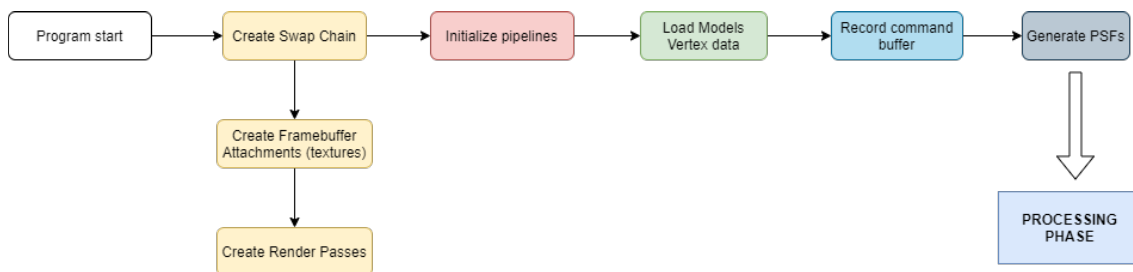



**Figure 3.4:** Schematic overview of preprocessing phase

#### Initializing sampling data structure

In order to produce a diffuse global illumination map, the scene needs to be irradiated from different points of view. To do this, the implementation uniformly generates points on a sphere with radius 1, resulting in the sampling data structure. In this implementation, it is done by performing a recursive subdivision on an octahedron, illustrated in figure 3.5. These uniformly generated points are in fact the earlier mentioned PSF-viewpoints, from which a PSF is constructed by creating an orthogonal projection towards the scene's origin. It is important that the whole scene is scaled within the bounds of the sphere

that contains the viewpoints, to ensure that each part of the scene gets irradiated equally. To realize this, an orthogonal projection space within the $[-1;1]$ range on each axis was utilized. The irradiance of the scene along 1 PSF viewing direction is demonstrated in figure 3.6.

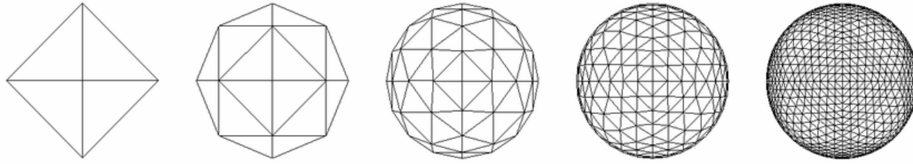


**Figure 3.5:** Recursive subdivision of an octahedron (recursive depths 0 to 4). Source: [5]

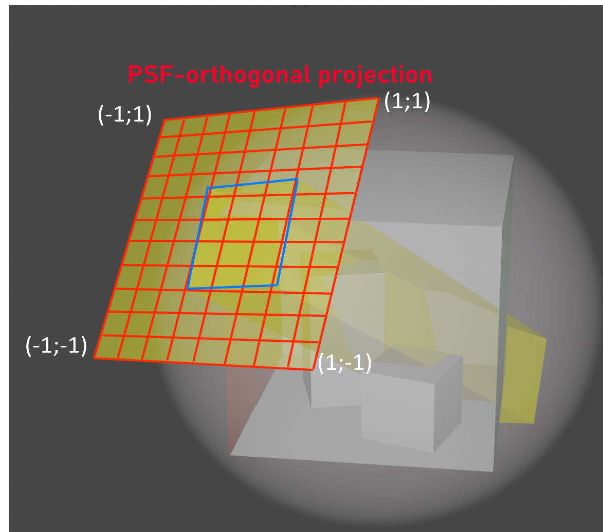



**Figure 3.6:** Simplified representation of the irradiance along one PSF direction. Within the presented orthogonal projection space, all light rays parallel to this direction are merged to generate the PSF. A subset of these light rays is made visible in this illustration (the outgoing light bundle marked in blue).

### Correction factor

The recursive octahedron subdivision algorithm delivers triangles which cover approximately the same area, although there is a certain margin of error, that may reach 70% [23]. This is due to the fact that each subdivision subdivides a triangle into 4 new triangles, which are then projected onto the surrounding sphere. This causes the middle triangle to be relatively larger than the other 3 triangles, since its distance to the sphere is greater (illustrated in figure 3.7). Because of this error, some PSFs would contribute more than others, which is not desirable. Therefore, a correction factor alpha is introduced to maintain uniformity:

$$\alpha = triArea * \frac{PSF_amount}{4\pi} \tag{3.1}$$

so that:

$$\sum_{i=0}^{PSF_amount-1} \frac{triArea}{\alpha} = 4\pi \tag{3.2}$$

In these equations, $triArea$ is the area of the triangle for which the correction factor is being calculated. $PSF_amount$ represents the amount of PSF viewing directions that are produced by the subdivision process. We divide by $4\pi$ to normalize the contribution of each PSF direction.

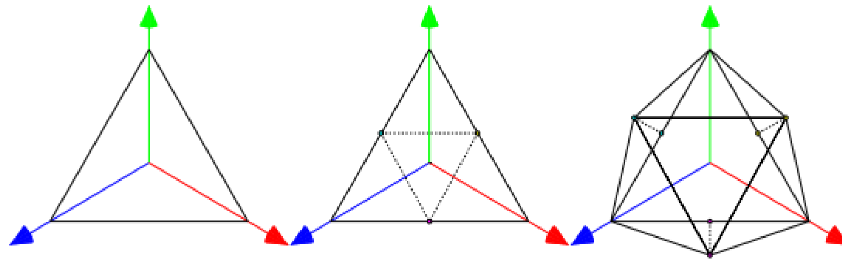


**Figure 3.7:** Subdivision of the positive quadrant of the sphere ($x = 1$, $y = 1$, $z = 1$). Each edge of the triangle gets divided in the middle, creating 4 new triangles, which are then projected onto the surrounding sphere. As becomes visible in this illustration, the resulting middle triangle is larger than the others. Source: [23].

### Processing phase

Secondly, there is the processing phase, which will be executed every frame once the preprocessing phase is completed. This is basically just the execution of the render loop consisting of the render passes which were set up in the preprocessing phase. The render loop consists of 2 sub loops. The outer sub loop iterates over the amount of lighting bounces that we want to capture. For example, if we set our lighting bounces variable to 3, this sub loop will have 3 iterations, each of which will calculate the lighting contribution of that lighting bounce respectively. The inner sub loop iterates over all the PSF-viewpoints that were generated in the preprocessing phase. In this way, for each lighting bounce, the render loop will calculate the lighting contribution for each PSF. The incoming light in each location is denoted as *irradiance*. The outgoing light at each location is denoted as *radiance*. Both are stored into textures (2D images).

The render loop is summarized by the pseudocode presented in 'Algorithm 1'. The functionality of each render pass encountered is explained briefly underneath the pseudocode, and more thoroughly in the resting subsections of this section.

**Algorithm 1** VLF Render loop

```
 1: procedure RENDER(amount_PSFs, amount_light_bounces)
 2:     for current_bounce in amount_light_bounces do
 3:         for current_PSF in amount_PSFs do
 4:             lookAtScene(current_PSF);            ▷ Place the camera in the current PSF-viewpoint
 5:             emitRadiancePass();                  ▷ Emit radiance render pass
 6:             stencilRoutingPass();                ▷ Stencil routing render pass
 7:             blendLight(PSF_total, bounce_total)  ▷ PSF contribution to current lighting bounce
 8:         end for
 9:         BRDFPass();                              ▷ Apply BRDF to objects containing radiance
10:         blendLight(bounce_total, scene_total);   ▷ Bounce contribution to scene's total
11:     end for
12:     toneMap();                                   ▷ Tone map the final irradiance texture
13:     presentTexture();                            ▷ Present the final irradiance texture to the screen
14: end procedure
```

1. **Emit radiance pass:** In the first render pass radiance will be emitted from all emitters. This light emission will be performed throughout all depth layers of the scene. Rays that pass through an emitter 'gain energy', rays that intersect with an occluder lose their energy (if they had any). During the first lighting bounce, direct light sources are the only emitters in the scene. In subsequent bounces, objects that have accumulated irradiance (incoming light, which will be reflected according to the BRDF) in the previous bounce will be an emitter as well.

2. **Stencil routing pass:** In order to propagate the emitted light throughout the scene correctly, the different depth layers that were written to in the emit radiance pass still have to be sorted from front-to-back based on their depths. This is done in the stencil routing pass. After sorting the samples front-to-back per fragment, the light is propagated throughout the scene.

3. **Blending the irradiance contribution of the current PSF with the irradiance of the current lighting bounce:** In this render pass the irradiance produced by the current PSF is taken and is blended with the total irradiance of the current bounce by performing additive blending. By doing this, the resulting total irradiance of a bounce is built up by taking the irradiance generated by each PSF into account.

4. **BRDF render pass:** This render pass makes use of multiplicative blending to obtain a good approximation of the diffuse component of the BRDF. Visible effects of this render pass are color bleeding. It should be noted that this is merely an approximation, but accurate enough for this application field.

5. **Blending the irradiance contribution of the current lighting bounce with the total irradiance**: Same as in render pass 3, only now the resulting irradiance from the bounce is taken and blended with the total irradiance. Therefore, the resulting total irradiance of the scene is built up by taking the resulting irradiance of each lighting bounce into account.

6. **Tone mapping + presentation pass**: Finally, the resulting total irradiance map can be used to render the final scene into a presentable image. Before it can be presented, a tone mapping algorithm is applied to transform the resulting irradiance color spectrum into a presentable range.

The most important part of this render loop is thus the irradiance calculation for each PSF (render pass 1 and 2), and then blending the results for each PSF into the resulting irradiance for the current lighting bounce. Ultimately, the total resulting irradiance for the scene is then obtained by blending the irradiance maps of each lighting bounce. This blending process is illustrated in figure 3.8.

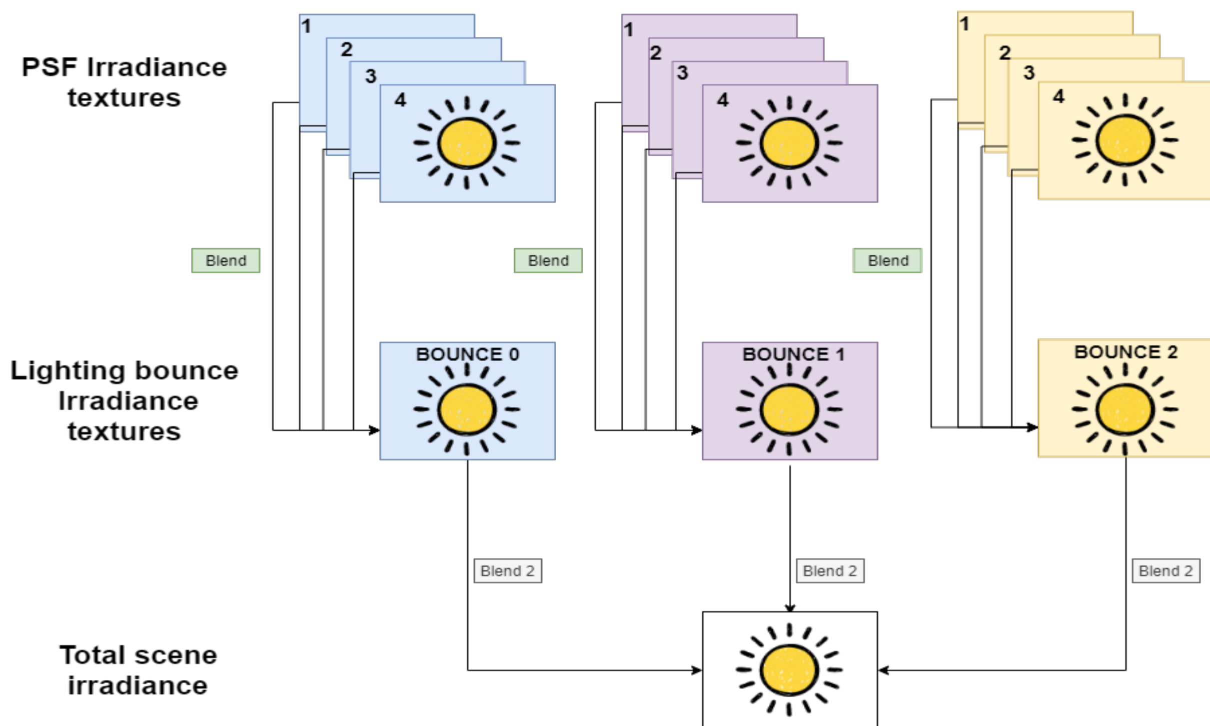



**Figure 3.8:** Schematic overview of blending the irradiance intermediate results to the total scene irradiance, supposing there are 4 PSFs and 3 lighting bounces in this setup.

### 3.2.4 *Emit radiance* render pass

#### Stencil buffer initialization

Before the radiance can be 'emitted', the framebuffers used by this render pass should be initialized. Therefore, the render pass consists of two subpasses. The first subpass will initialize the stencil buffer with the correct sample mask values. The second render pass will then emit the radiance throughout the scene, saving the correct luminance information in each sample. The first subpass will be described in this subsubsection. The stencil buffers are initialized as [25] proposes. Since the fragment shader

is executed per fragment, but the stencil operation is executed per sample, this first pass ensures that that the correct sample masks are set for each sample. Because the stencil operation is executed for each sample, the next subpass will render the fragments into the samples (since each pixel contains 8 samples in this implementation, each pixel can have up to 8 fragments rendered into it). Note that these samples are filled in rasterization order, since every fragment is rendered by the fragment shader. The rasterization order is not equal to the front-to-back order of the fragments. That is why in the next render pass (*stencil routing* pass), discussed in section 3.2.5, the fragments still need to be sorted front-to-back. The next subsubsection will discuss the second subpass of the *emit radiance* render pass.

#### Radiance emission

Starting off, it is completely dark in the scene. In order for light to appear in the scene, rays are shot *orthogonally* along each PSF, throughout all the depth layers (also called *radiance interfaces* in this context). Every fragment that the ray intersects with is checked whether it is front facing to the ray or not. If it is front facing, this fragment is said to be an *occluder*, so it will not send out any radiance. On the contrary, if the fragment is back facing, it is checked whether it is an emitter. If it is indeed is an emitter (either a direct light light source, or emitting irradiance from previous lighting bounces), it is assigned the appropriate color value. To be able to store the results for fragments on each depth layer along the ray, multisampled color and depth buffers were used. Briefly said, for each PSF, the fragment shader will shoot out a ray for each fragment, and every time that ray intersects with an object, the color and depth value at that intersection point will be written to a sample for that fragment in the respective buffer. This way, all the color/depth information for each depth layer from this certain PSF-viewpoint is obtained. Since this thesis works with a relatively simple scene, color and depth buffers with 8 samples for each pixel were sufficient. However, for more complex scenes, the sample count per pixel would have to be increased. Another possible way to solve this could be subdividing the scene and executing the whole render loop for each subdivision, without forgetting to take in account propagation between 2 neighboured subdivisions (which is obviously not necessary when the scene only exists out of 1 subdivision), to obtain a final result for the whole scene. Since this thesis mainly aims to prove the concept, merely the implementation with 8 samples is sufficient. The fragment shader for this render pass also receives an *emit radiance texture*, in which it can look up the outgoing radiance color value for each fragment (whether that fragment is a direct light source or sending out irradiance from previous lighting bounce). During the first lighting bounce, the only emitters in the scene will be any direct light sources. In the example that this thesis presents, the ceiling of the Cornell box was used as a light source. The radiance emission process is illustrated in figure 3.9.

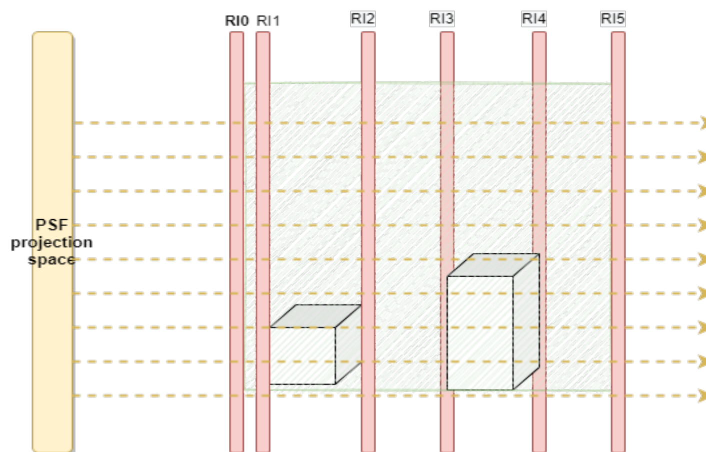



**Figure 3.9:** Simplified representation of the radiance emission pass. The current PSF viewing direction is marked by yellow dotted arrows. Along this viewing direction, different depth layers, also known as *radiance interfaces*, are marked in red. For each intersection that is encountered along a ray, a color value will be written to the corresponding sample. The color value will differ based on whether the intersecting object is an *occluder* or *emitter*.

#### Size of the light source

Before moving on to any further details, it is worth noting that using this method to compute the global illumination model heavily depends on the size of the light source. That is, the smaller the light source is, the more directions that are necessary to cover the same area of irradiation. Figure 3.10 illustrates this. It should become clear that, with the used method of global illumination in mind, it is more efficient to use emitters (light sources) of a greater size. That is why this thesis opted to assign the whole ceiling of the Cornell box as a direct light source. Hopefully this demonstrates that this technique is not very suitable to calculate direct lighting for scenes that do not make use of a skylight. Since a PSF contains the whole scene, the technique is also viable for indirect lighting. This is due to the fact that each object in the scene has the potential to emit radiance. For a certain irradiation direction, a PSF considers all objects in one render pass.

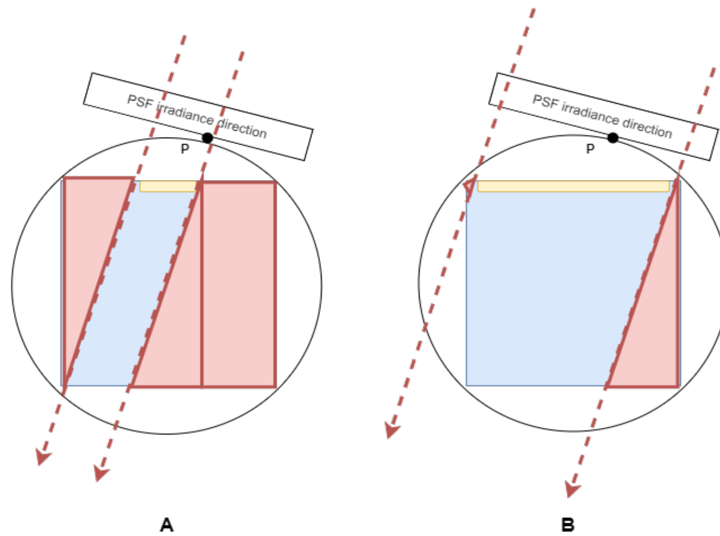



**Figure 3.10:** Illustration of the impact of the size of the emitter(s) that is/are used with this method of global illumination calculation. A small emitter is indicated in yellow in illustration A, a greater emitter is used in illustration B. In both illustration the irradiation area is indicated in blue, while the lost area of irradiation is indicated in red.

### 3.2.5 *Stencil routing* render pass

After rendering the irradiance for each depth layer correctly into the multisampled texture, the render loop is ready to move on to the next pass, which will take care of the light propagation throughout the scene. To start off, consider 1 PSF viewing direction, which is pointed towards the origin (middle of the scene). This viewing direction is delivering a certain (orthogonal) view of the scene. Briefly explained, this render pass would like to propagate the light along this viewing direction throughout the scene. The light will therefore start at the PSF-viewpoint, and go further along this viewing direction, resulting in the correct direct lighting from this point of view. However, this program would like to capture the illumination information of all viewpoints in 1 texture (which can then later be sent over the network and used to render the scene at client side). This is why this render pass will render into 2D texture space, according to the UV map [17] of the scene.

Note that no scene geometry data (vertex positions/normals/color values...) are directly passed to the shaders of this render pass through vertex buffers, because it is rendering into a 2D texture space (fullscreen quad). However, this pass still needs the position and normal of each fragment in the scene, because it will need to be able to verify whether a fragment is visible from the current PSF-viewpoint (whether it is front facing). Additionally, it will also need some idea of the depth value (relative to the current point of view) of the fragment in question to be able to fill it with the correct color value. To store this information, an additional render pass (see 3.15) was implemented, which renders the scene's world coordinates and normals into texture space. It should also be noted that the whole scene is scaled within $[-1; 1]$ on each axis. Before using them, the vertex world positions need to be transformed to the viewing space of the PSF currently being evaluated. This is visualized in figure 3.11.

The description that follows now, up until the end of this section, is a procedure that is executed per

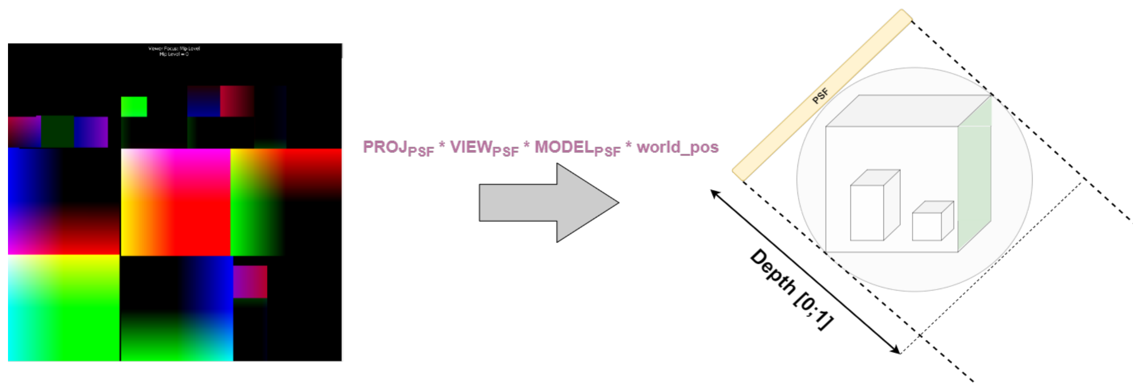



**Figure 3.11:** World positions stored in a texture are transformed into PSF viewing space. Since the world positions lie within the $[-1;1]$ range on each axis, they can easily be transformed to $[0;1]$ range. The transformed coordinates can then be used to read from the irradiance texture (figure 3.12).

fragment (by the fragment shader). The world position of the fragment is first transformed to the space of the current PSF-viewpoint. With the fragment being in the correct position, relative to the current point of view, the next step would be to read the color value for this fragment out of the multisampled irradiance texture that was the result of the *emit radiance* render pass. In order to read from this texture correctly, it is desirable that the coordinates are scaled within a $[0, 1]$ space. This could be seen as an $x$-percentage and a $y$-percentage that, after multiplying it with the width/height of the texture, signify which pixel should be read from the irradiance texture. An example of this is presented in figure 3.12.

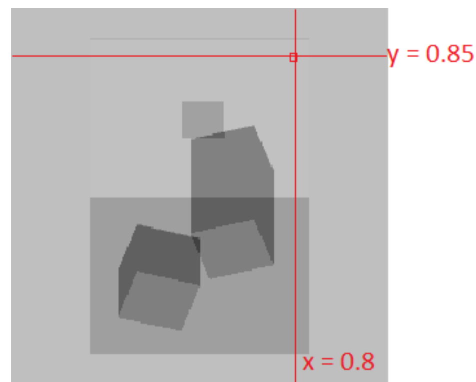



**Figure 3.12:** Reading from the irradiance texture (result coming from *emit radiance* render pass). The read pixel is decided by the $x$- and $y$-coordinates of the fragment currently considered by the *stencil routing* render pass, after transforming the position of that fragment to the space of the current PSF-viewpoint, and rescaling the coordinates to $[0;1]$ space. For the fragment in this illustration, an $x$-value of 0.8 and a $y$-value of 0.85 were obtained after performing the previously mentioned transformations.

### Front-to-back sorting

Once the pixel to be read from is decided, the depth value for each depth layer (each sample, remember that this is a multisampled texture) is read into an array. The same procedure is repeated for the color values. Because the samples are by default not necessarily in a front-to-back order (rather, it would be a great coincidence if this was the case), they still need to be sorted front-to-back. This is obligatory because ultimately the fragment currently being processed in the fragment shader needs to be compared to these values, and in order for light to propagate correctly, this needs to happen in a front-to-back order. For example, consider a box that a flashlight is shining upon from a 45°angle with the floor. The position that this light source is in can be thought of as the PSF-viewpoint, and the angle it is emitting light from presents the irradiation direction or the PSF viewing direction. Given this scenario, a part of the box will be visible (e.g. the front and top panels), due to the light that shines on it. However, behind

the box there will be shadows, because the box is front facing the light and is therefore an occluder. So for a certain ray that we send from this flashlight, it will intersect with the top panel of the box, the back panel of the box, and the floor, although the pixel selected from the irradiance texture, like illustrated in figure 3.12, will be the same for each of these intersections, right because they lay on the same ray for this viewing direction! However, as mentioned in the introduction of this example, each of these intersections have another color value. The top panel is illuminated, so it will receive the color of the light source, the back panel is back facing this viewing direction, so should not be considered, while the floor intersection is in the occlusion area and should have a black color. This example is illustrated by figures 3.13 and 3.14.

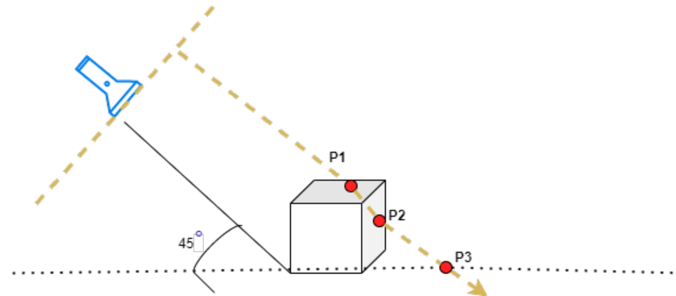



**Figure 3.13:** Schematic illustration of the example of a flashlight illuminating a box, like mentioned in the paragraph above. The irradiation area is marked by a dotted yellow line, orthogonal to the flashlight. An example ray is marked by the dotted yellow arrow. The three intersection points along this ray are marked by $P1$, $P2$ and $P3$.

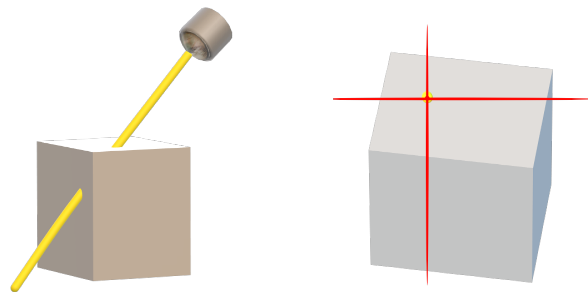


**Figure 3.14:** On the left, the 3D representation of the example illustrated in figure 3.13. On the right, the selection of which pixel to read from the irradiance texture for the current viewpoint (see figure 3.12), applied to this example. Since the ray is a straight line, it becomes clear that for each of the intersections ($P1$, $P2$ and $P3$) the same pixel is selected.

### Light propagation

This is where both the light propagation and the radiance emittance come into play, and very crucial to understand. The *emit radiance* render pass stores the color information for each depth layer according to the current PSF-viewpoint, while the *stencil routing* render pass propagates the light along the current PSF viewing direction, and decides for each fragment in the scene which color it should be assigned, in a front-to-back order. The fragment currently being processed is tested against each sample to decide whether it is in front or behind this sample. If it is in front, the test succeeds, if it is behind, the test fails. We keep track of every test that succeeds. Because the samples are sorted front-to-back, the tests are also being evaluated front-to-back and this is how the correct color value for the fragment in question is found. This is correct because as long as the sample is in front of the fragment, the test will fail, from

the moment that the sample depth is greater than or equal to the fragment's depth, it will succeed. Once a test succeeds, the color value of the sample that corresponds with this test can be used to overwrite the fragment's color. Because multiple tests can succeed, (e.g. multiple samples can have a depth value that is greater than or equal to the fragment's depth value) it is, once again, important to evaluate the tests and overwrite the fragment's color in a front-to-back order. This will ensure that the fragment gets the color of the sample corresponding to the last succeeded test (and thus the sample with a depth value closest to its own). This implementation could also explain why it is called a *light propagation* algorithm. The samples correspond with the light values on each depth layer, and they are each evaluated front-to-back for the current viewing direction, just like light would travel from the front of the scene (where the light source is located) to the back of the scene.

### 3.2.6 *Geometry texture* render pass

As briefly discussed in the previous section, an additional render pass was added to store the world positions (scaled between $[-1;1]$ on every axis) and normals of each fragment into two textures. This procedure is executed by the *geometry texture* render pass. The performed actions by this render pass are fairly simple, however subtle. In order to store the desired information for all fragments into two textures, the world positions and normals respectively were rendered into texture space. This basically results in a UV map of the scene, but instead of containing the color values for each fragment (like a UV map texture is usually used for), it contains the world position or normal. It should be noted that in case of a static scene, this render pass should only be executed once, since the world positions / normals will not change throughout the program's runtime. In the case of dynamic scenes, this is one extra render pass that will need to be executed every frame, although there are definitely some optimizations that could be applied here later on. For example, the UV map of dynamic objects could be stored in a separate UV map, so the world positions of the static objects will not be recalculated unnecessarily. The generated world position UV map for the Cornell box scene used in this thesis is presented in figure 3.15.

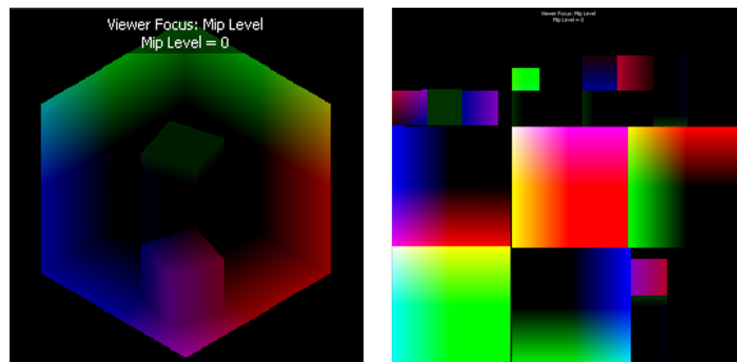



**Figure 3.15:** Rendering of vertex world positions. On the left, the vertex world positions are rendered into view space. On the right, the vertex world positions are rendered into texture space (UV map). The approach on the right is used by the *stencil routing* render pass, since it stores the world positions of all vertices, independent of the current viewing position, unlike the approach on the left.

### 3.2.7 *Blend light* render passes

Once the resulting irradiance for the current PSF-viewpoint is obtained after performing light propagation in the *stencil routing* pass, it is time to let it contribute to the irradiance generated by the current lighting bounce. This is done by performing additive blending [10]. The irradiance generated for the current PSF is simply added to the total irradiance for the current lighting bounce. Note that the total irradiance for the current lighting bounce needs to be cleared every time a new lighting bounce is initiated, unless separate textures are used (which is less efficient in terms of memory usage). The same thought process goes for the other blending render pass, which accumulates the total irradiance of each lighting bounce and performs additive blending to become the resulting total scene irradiance. Because these two render passes are very similar (almost the same), this section is dedicated to both of them.

### 3.2.8 Converting irradiance into radiance

It should also be noted that once a lighting bounce finishes, the irradiance (incoming energy) accumulated in the current lighting bounce should be used as radiance (outgoing energy) in the next lighting bounce. This is quite intuitive, every object that receives incoming light energy will partially or fully reflect that light energy afterwards. In the implementation this can easily be done by using the resulting irradiance texture for the current bounce as the *emit radiance texture* in the *emit radiance* pass for next lighting bounce.

### 3.2.9 *BRDF* render pass

To obtain realistic color bleeding and diffuse transfer, the diffuse component of the BRDF should also be taken into account. Since this thesis aims to illuminate a 3D scene realistically, but not necessary 100% physically correct, an approximation was used for this component. This render pass calculates this approximation by making use of two inputs: a *diffuse lookup texture*, that contains the diffuse material colors of each fragment in the scene, and the irradiance for the current bounce. For example one can consider a fragment on the green wall of the Cornell box. Once the diffuse component of the global illumination is computed for this fragment, the fragment shader for this render pass will look up the color of this fragment in the lookup texture and will read the green color value. Once that value is read, it can also look up the irradiance that this fragment accumulated during the current lighting bounce (since that will be the radiance that this fragment will emit during the next render pass). Finally, multiplicative blending is applied to both the lookup values that were found. This results in the color (radiance) that the fragment in question will send out during the next lighting bounce. Figure 3.16 illustrates color bleeding, which is a consequence of this render pass.

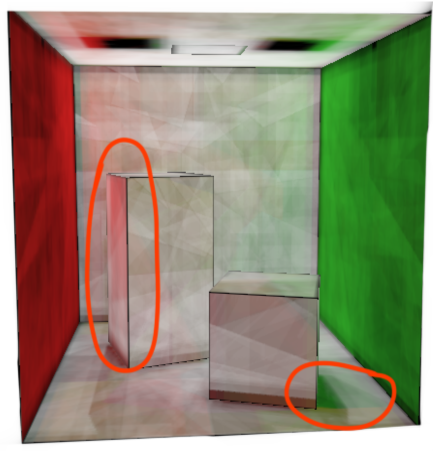


**Figure 3.16:** Visible color bleeding on the boxes and back wall of the Cornell box. (Virtual light field setup: 384 PSF directions, 2 lighting bounces, (256 x 256) texture)

### 3.2.10 *Tone mapping* and presentation render pass

#### Dynamic range limitation

Last but not least, once the final irradiance map is calculated for all PSF directions and all lighting bounces, it contains all the diffuse illumination information necessary to render the scene. However, due to the many additive light blending passes that were executed, the result will have a much higher *dynamic range*[6] (HDR) than the $[0, 1]$ range in which *RGBA* tuples are encoded. Since the color values of the calculated illumination will be located far out of this range, but since the maximum value is limited to 1, (almost) every pixel would display as a completely white pixel, even though in reality there might be differences between the actual color values. For example, consider color $A$ with value $(10.0, 10.0, 10.0, 1.0)$ and color $B$ with value $(1000.0, 1000.0, 1000.0, 1.0)$. Both colors would be clamped to color $(1.0, 1.0, 1.0, 1.0)$, which is plain white, while the light represented by color $B$ is in reality 100 times more intense than color $A$! This is obviously a problem, because this would mean almost pixel

[6] `https://web.archive.org/web/20150426032033/http://www.electropedia.org/iev/iev.nsf/display?openform&ievref=723-03-11`

in the scene would appear as white, while in reality there might be a great diversity in color values throughout the scene.

### Reinhard tone mapping

This problem can be solved by artificially 'enlarging' the dynamic range by mapping all the high range values into the $[0, 1]$ interval. Note that, in case of a 32 bit RGBA texture, one will only obtain a good approximation of the HDR image by performing this mapping. This is because the 32 bit texture will obviously have a finite set of 255 possible mappings per component (smaller than the set of values for the high dynamic ranged image). A floating point texture may be used to solve this problem and extend the number of values that can be mapped to. This operation is called *tone mapping* [21]. There are various tone mapping algorithms available. This thesis opted to go for *Reinhard tone mapping* [30], since it is a relatively simple tone mapping algorithm, and it is sufficient for this application. The *Reinhard* tone mapping operator is denoted as follows:

$$L_d(x, y) = \frac{L(x, y)}{1 + L(x, y)} \tag{3.3}$$

In equation 3.3, $L_d$ stands for the resulting luminance (the resulting color value for a certain light bundle, represented by a pixel $(x, y)$), whereas $L$ stands for the average of the logarithmic total luminance in the scene. This means that, for great luminance values, the equation will converge to 1, whereas for smaller luminance values, the equation will converge to $1/L$. In this way, the resulting luminance is guaranteed to be located in displayable range. An example of *Reinhard* tone mapping applied to the thesis' presented scene with HDR lighting is illustrated in figure 3.17.

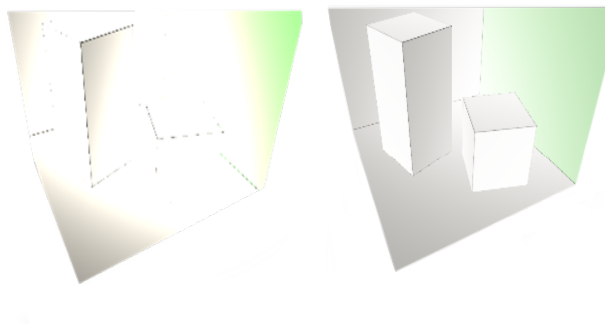


**Figure 3.17:** Cornell box scene illuminated by HDR lighting. On the left, the resulting image without tone mapping. On the right, the resulting image after *Reinhard* tone mapping was applied.

### Exposure

In the real world, the human eye is adapting to the environment. For example, if a person walks into a completely dark room after staring into a bright light for 5 minutes, that person will not be able to see anything for a moment, until the eye adapts itself to the dark and the person feels like they got used to the dark. This phenomenon happens because, as illustrated in this example, the amount of light that reaches our retina differs over time. The luminosity that the human eye is exposed to is called the level of *exposure*. Similarly, this term is also used in the photography industry, where it represents the amount of light reaching the camera's sensor (or film in case of an analog camera). As can be expected, the higher the exposure, the brighter the resulting image will be. In the scene represented by this thesis the amount of light being sent out stays relatively constant, which is why a constant exposure setting could be used.

However, it should be noted that for more complex and dynamic scenes, it could be desirable (and even necessary) to use an adaptive form of exposure. During this thesis, a (brief) look was taken into this, however, there was not enough time left to finally finish the implementation of this. The proposed method to calculate the (adaptive) exposure would generate a *logarithmic luminance histogram* in order to get a good idea of the luminance distribution in the current image. This histogram can then be used to calculate

the average luminance in the image. The average luminance gives a good indication of the amount of light that is present in the image, which is why it can be used to adapt the exposure accordingly. Finally, as with tone mapping algorithms, there exist many algorithms concerning (adaptive) exposure. However, the full theory behind these are out of the scope of this thesis.

### 3.2.11 Argumentation

One could argue that the proposed rendering approach has a rather large memory requirement, having to store all the irradiance information for each PSF, for multiple bounces, which is in principle a collection of 2D textures (images). This is a good point, but since the server has a substantial amount of resources, this should form no significant problem. Compression could be a possible solution for this problem as well. The rendering algorithm itself initially looks quite heavy: we have relatively many render passes, which need to be passed for each PSF, and for each lighting bounce. However, right because this algorithm exploits the rendering capabilities of the GPU, by simply rasterizing 2D images from different viewpoints, this is an operation that is executable very efficiently by the GPU with a high degree of parallellism, which makes it feasible.

With that being said, there are still some bottlenecks, like the sorting of the fragments during the stencil-routing stage. This sorting operation is where we lose some degree of parallellism. In terms of time complexity for rendering a scene once the diffuse global illumination map is calculated, this approach is very efficient: we can render a static scene from a certain viewpoint practically in $O(1)$, by simply performing *UV mapping* onto the scene's geometry, accordingly to the viewpoint, using the diffuse illumination texture that was previously calculated. For dynamic scenes, there will be the need to recompute (parts of) this texture, but this should not be a great problem either, since 2D image operations can be performed very efficient on a GPU, and the server will ideally consist of multiple GPUs. It should taken into account that this is merely a proof-of-concept implementation. Many optimalizations and alternatives to the used techniques are possible, as will thoroughly be discussed in section 4.

### 3.2.12 Virtual light fields vs. traditional ray tracing

Another question that should be considered is why one would prefer the use of virtual light fields to calculate global illumination over traditional ray tracing techniques (e.g. [20]). First of all, because rays in a virtual light field can only 'gain energy' by an *emitter*, the size of that emitter has an impact on the effectiveness of the light field. This is illustrated in figure 3.10 Therefore, virtual light fields are ideal for scenes with very large emitters, such as skylights.

Furthermore, the data that a virtual light field represents has a much higher coherence than it has in traditional ray tracing methods. The data represented by a virtual light field is in fact a collection of **parallel bundles of light rays**. This is very different from traditional ray tracing, where a ray is split into multiple new rays with different, non-coherent directions at each point of intersection. This automatically results in much lower coherence, because these new rays have nothing in common with each other, and all need to be evaluated separately. In contrary, the dense parallel light bundles with a high coherence all have the same direction. A PSF is in fact nothing more than an orthogonal projection towards the origin of the scene from a certain viewpoint, such that the projection contains the whole scene. Because of this, the rasterization hardware within a GPU can be exploited to find each ray's points of intersection with the scene's geometry. This high coherence of data is likely one of the most important advantages that virtual light fields offer over traditional ray tracing techniques.

## 3.3 Client (and client-server communication)

### 3.3.1 General approach

The client side of the architecture will be enhanced by the diffuse global illumination map received from the server. It will simply render the scene geometry (a relatively lightweight task). The client will receive this diffuse global illumination texture periodically from the server, and will then map it to its rendered geometry in order to obtain the final illuminated scene. Since the geometry is the same on both client and server side, the resulting scene at client side will also be exactly the same as the one rendered on server side. However, the amount of calculations that the client had to make in order to obtain this result is severely lower, which is exactly what we desire. The client application also provides an interactive scene

in which the user can move around in first-person view to inspect the resulting scene. Additionally, up to 2 clients can connect to the same session, in which each client is able to control a cube. Every time a client moves a cube, this is communicated to the server, which can then update the diffuse global illumination map accordingly. In this way, the responsiveness of the dynamic virtual light field is demonstrated.

### 3.3.2 Technologies and Tools

This section will cover the used technologies and tools (programming language, libraries, etc.) for the client-side application and client-server communication.

#### React.js

*React* is an open source JavaScript framework initially developed by Facebook in 2011 [27]. It can be used to create front-end web applications for the web browser (or native apps in case of React Native). Since this thesis aims to create a representative *shared rendering cloud gaming* implementation, it is very desirable that the client application is supported by as many platforms as possible. Therefore, a web application is an ideal choice. Furthermore, as of today, *React* is one of the most, if not the most popular front-end framework available [28]. Finally, another thing that led to this decision, is the fact that I already had a decent amount of experience with *React*, which would definitely be beneficial to the implementation process.

#### Three.js

*Three.js* is a JavaScript library and application programming interface (API) built on top of *WebGL*[7]. It allows 3D computer graphics and computer animations to be displayed in a web browser, without having to rely on any additional browser plugins. The high-level nature of the library makes it very attractive to use, compared to much lower-level *WebGL*. The client application was built using a higher-level framework because the majority of this thesis involved implementing a complete low-level render engine from scratch, using the *Vulkan* API, which was far more time consuming. This has the trade-off of less control over the implementation, but since the client-side implementation is relatively simple in the subject of this thesis, this does not form a real problem. There are other high-level 3D JavaScript libraries that are perfect candidates as well, such as *Babylon.js*, *D3.js*, and more. It was just a matter of picking one of these, and this thesis opted to use *Three.js*.

#### WebRTC

An introduction to WebRTC was already given in section 2.2. The benefits that WebRTC has to thesis and therefore the reason why it was chosen is also discussed there. Besides WebRTC one other possible candidate was considered, which is WebSockets [18]. Although both are part of the HTML5 specification, WebRTC (originally) offers bidirectional communication between browsers, while WebSockets offer a bidirectional communication between browser and web servers. This phrasing makes it seem that WebRTC would not be suitable for use in this thesis, but because WebRTC has its native code available, it is possible to write native applications that make use of the WebRTC standards. Furthermore, WebRTC was designed for high-performance, high-quality communication of audio, video and arbitrary data between peers. While it is possible to do the same communication over WebSockets, it was not designed for these purposes. WebRTC was ultimately chosen because the purposes for which it was designed meet the requirements of this implementation very well.

### 3.3.3 Communication protocol

To make the communication between the client and server possible in a structured way, a very simple communication protocol was designed. Before one can design a protocol, one first needs to evaluate which communication there will be between the nodes in question. In the case of this implementation this is fairly limited and simple: Besides the client-server communication discussed in section 3.1, the only additional communication necessary between client and server is the shipping of textures (server to client), the current position of the movable cubes (server to client), and possible moving commands for the cubes (client to server). Message formats for each of these will be covered in this section.

[7] `https://en.wikipedia.org/wiki/Three.js`

#### Texture shipping (server to client)

Each time a new texture image becomes available at server side, the server sends it into the datachannel that was established between client and server. It sends it into this datachannel as a binary byte stream, which is recognized by the client side. To let the client side know that the end of an incoming image is reached, the following message is lastly sent: `'PNG_TRANSFER_COMPLETE'` . The client then recognizes this as the end of an incoming texture image on which it will flush its data buffer and parse the image to a texture. Following this process, the texture will then be used to render the scene.

#### Movable cube positions (server to client)

In order to keep the cube positions synchronized for each client in the session, the server needs to let each client know what the current positions of the cubes are. To do this, it periodically sends a message to each client in the following format:

```
'x1!y1/x2!y2'
```

In this format, `(x1;y1)` and `(x2;y2)` represent the coordinates of cube 1 and 2 respectively. The separator tokens `'!'` and `'/'` were used to separate the coordinates and make string operations easier at client side. The client can use this message in turn to update the local positions of the movable cubes.

#### Moving commands (client to server)

Each client also gets assigned a cube that they can move, based on the order in which they joined the session. To move this cube, keyboard inputs are used to trigger events that will send commands to the server. These commands are in the following very simple format:

```
'up'
'down'
'left'
'right'
```

On arrival of these commands at server side, the server will distinguish between these commands and update the position of the cubes according to which client sent the command.

One could criticize the fact that these protocols are really simple and not sophisticated in any way. Although this is true, this was merely a first (fast) approach, that was sufficient to obtain a representative, working implementation, which is the goal of this thesis.

# Chapter 4

# Results and Conclusions

This chapter will take a critical look into the implementation (process), based on varying parameters, and evaluate the results obtained during this thesis. It will do so by looking into the separate artifacts that have an interesting impact on the performance of the application. In section 4.6, the resulting implementation is presented. It should be noted from the beginning of this chapter that on the client side there are surely still many optimizations possible. Since the majority of time had to be invested in understanding the concept of virtual light fields and more in particular the implementation, as well as debugging the implementation, there was barely any time left for optimizations. However, the goal of the thesis was to find a **representative working setup** for the *shared rendering cloud gaming* architecture, which is still met in this way.

This thesis results in a simplified **proof of concept implementation** of a shared rendering infrastructure, making use of virtual light fields to calculate the diffuse global illumination in the scene. The implementation is a merging of concepts that have already been implemented before separately, although this combination makes it very interesting. It can be considered as a foundation for future work, that I intend to continue working on during my Master's degree.

## 4.1 Setup

First of all, let us consider the **virtual light field setup** that was used during all of the experiments/trade offs that are presented in this chapter. I was able to test two configurations:

1. 384 PSF-viewpoints, 2 lighting bounces
2. 1536 PSF-viewpoints, 2 lighting bounces

Configuration 1. can be thought of as a good lower bound that is still acceptable. However, it should be mentioned in the beginning of this chapter that due to hardware limitations, it was unfortunately impossible to test the implementation with higher configurations. Higher configurations might be desirable for some scenes, but in context of this thesis these configurations suffice.

## 4.2 Texture formats

During the course of this thesis, a few image encoding options for the global illumination map were tried out. First of all, an attempt was made using the *JPEG* format [26]. JPEG is by far one of the most popular image encoding formats available. However, since it uses 'lossy compression', which means it downgrades the image's quality in order to compress it, it might not have been the ideal choice for the purpose of this implementation. Secondly, the *PNG* [33] format was used. In contrary to JPEG, PNG uses a lossless compression method and is therefore more suitable for storing the global illumination map, since it is desirable to maintain the texture's quality.

After trying out both formats, no noticeable difference in performance was observed. The main bottleneck seemed to be the encoding of the texture image at server side, and then decoding it at client side. This was quite expectable, since the amount of data that needs to be sent over the network lays in the same

order of magnitude for each suitable format. To overcome this bottleneck, another approach was made to send the raw texture image data over the network, to directly use it on client side without it being necessary to encode/decode this data. The attempt seemed to be almost working, the binary data was being received at client side. However, there were constraints in the *Three.js* library, which has a class that supports raw texture data, but it produced a distorted, unusable image in the end. Although there was no time left to further explore and resolve this option, it is definitely worth mentioning it because it would be a more efficient approach than the one that was implemented. This was already noticeable in the arrival rate of the textures at client side.

## 4.3 Texture sizes

Besides texture formats, different texture sizes were also experimented with. However, due to hardware limitations, this is where problems arose. In the early stages of the implementation, while I was testing the texture streaming with arbitrary textures, textures of an acceptable resolution (1024 x 1024) were used. The performance was mediocre, but not good enough for real-time frame rates. The main bottleneck, however, was found in the encoding/decoding of the image according to its format, like discussed in section 4.2. The use of raw texture data could therefore be a solution to this issue.

Unfortunately, due to the limited video memory that my system disposes of, I was forced to transition to textures with a maximum size of (512 x 512). Although that this is not an acceptable resolution in a gaming environment, provided that the machine on which the application runs is given the required resources, this implementation is still representative for textures with higher resolutions. Three resolutions, (128 x 128), (256 x 256) and (512 x 512) were tested respectively. The (512 x 512) resolution, however, resulted in a very poor performance which results in a non-interactive experience. The resulting frame rates for the other resolutions are presented in figure 4.1. It shows that by halving the resolution, the frame rate increased with a factor $> 3$. This is merely to demonstrate that, due to the high memory demands of this implementation, texture resolution has a significant effect on application performance. Note that the current implementation is still highly unoptimized, all the PSF directions and lighting bounces are being recalculated for each frame (in the case of a dynamic scene). Although both frame rates presented here (9.4 FPS and 2.9 FPS) are not sufficient to deliver an enjoyable real-time experience, this is likely to change after hardware upgrades (more video memory!) and optimizations are applied.

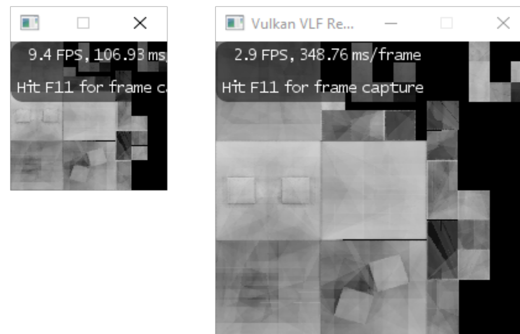



**Figure 4.1:** Frame rate comparison for different texture resolutions. The (128 x 128) resolution reaches an average frame rate of 9.4 FPS, while the (256 x 256) resolution only reaches a frame rate of 2.9 FPS. Screenshots were made in NVIDIA Nsight.

## 4.4 Streaming

Another aspect of the implementation that should be considered in this chapter is the streaming aspect. The quality and performance of the streaming of the global illumination textures over the network has a great impact on the resulting experience of the application. For example, one could create a very efficient back-end render engine which calculates the global illumination in real-time at high frame rates, but this would be of no use if the streaming mechanism is causing the major bottleneck. The different components are linked together like a chain, and a chain is only as strong as the weakest link. During the course

of the implementation, several iterations have led to the streaming mechanism that is currently being used.

Firstly, a very naive and poorly performing mechanism was set up in the following way: The server and client would communicate with each other over TCP/IP, using HTTP as the application layer protocol. The (again, naive) reasoning behind this was the fact that the implementation made use of a web browser as the client, and HTTP is the protocol used between web servers and web clients for applications on the web. Of course, this approach is far from ideal, for several reasons:

- The back-end application would need to save the image to its disk in a certain format, in order to answer to the HTTP request appropriately, which requests an image from a certain directory path. Although the 'save to disk' could be prevented by keeping the most recent texture image in memory at all times, but in this way the server would no longer be able to work correctly according to the HTTP protocol, which would be a very 'hacky' solution.
- The front-end application needs to 'poll' each time it wants to receive a texture. HTTP is a 'pull protocol', which means that the information needs to be pulled from the web server. Every time a client wants information from the web server, it needs to send a request first. This is obviously very inefficient, it would result in a great overhead of network traffic, besides unnecessary delay.
- A bidirectional data stream is impossible. With this method, the client can only send HTTP requests to the server, but it cannot send any additional messages, like the commands to move the cubes. This would obviously prevent the application from being interactive in any way.

Hopefully that makes clear that this first attempt was a mistake, mainly caused by of the lack of knowledge in this domain during the early stages of the implementation. After doing more research on web technologies for real-time, bidirectional communication, two good candidates remained, namely WebRTC and WebSockets. The motivation behind why WebRTC has been chosen in the end was already covered in section 3.3.2. Lastly, the current implementation of the streaming mechanism (client-server communication) is basic and functional. It leaves a lot of room for optimizations and improvements, efficiency wise, but also in terms of security. However, the current mechanism is not causing any bottlenecks (it currently takes longer to produce a texture than to send it over the network). Therefore, since this mechanism is not the weakest link in the chain at the moment, it would be wise to first optimize the other significant bottlenecks. The final conclusion on the streaming mechanism: It can be a lot better/safer/cleaner, and it will likely have to be improved in the future, however, for the goals of this thesis and the current state of the application, it offers sufficient performance and functionality.

## 4.5 Render passes

Taking a look at the render passes that are currently in use, they all seem to be inevitable. The world coordinates need to be saved somewhere (*texture geometry pass*), radiance needs to be emitted throughout the scene (*emit radiance pass).* Furthermore, the light needs to be propagated correctly through the scene (*stencil routing pass*). The remainder render passes are simply performing basic color blending in order to bring together results from different lighting bounces / PSF-viewpoints. However, analyzing the render loop with NVIDIA Nsight provided better insight into which render passes are the most expensive. The results are presented in figure 4.2.

It becomes clear that the render passes that are causing the most overhead are the *stencil routing* and *emit radiance* passes. For the *emit radiance* pass, this could possibly be explained by the multisampled image that the fragment shader needs to render to. Since the render pass is not doing much more besides rendering the scene geometry, the fact that the fragment shader has to be executed for each sample could explain a lower performance than expected. However, this is something that cannot be avoided using the current technique. More sophisticated methods that do not make use of multisampled textures could increase this performance, but more research is necessary in order to find and test these methods. Secondly, the *stencil routing* pass currently has a significant contribution to the render time per frame. The front-to-back fragment sorting that must be completed before the light propagation starts, in order to retrieve a correct result, accounts for the majority of the overhead here. Two methods of sorting, *bitonic sort* and a *compare and swap* algorithm, were experimented with. The winner was unmistakably the compare and swap algorithm. Since this algorithm still does not deliver a satisfying performance, the implementation will need to resort to more efficient algorithms or techniques in the future. It should already be mentioned that the sorting algorithm does redundant work in certain situations because it

takes into account all of the samples for the fragment currently being processed. This despite the fact that many of these samples are unused, because the fragment in question has a limited number of depth layers. Therefore, it is very likely that there is (a lot) of room for improvance in this area.

One possibly more efficient solution could be to sort polygons front-to-back instead of samples. This would greatly reduce the overhead that is caused right now by sorting all the samples **per fragment**. Although sorting polygons would not deliver a 100% correct result in every situation, it might be accurate enough for this purpose. Another very interesting topic to consider is *order-independent transparency* [9] [31]. This technique (partially) avoids per-fragment sorting operations but still offers plausible results. Since the render loop's main bottleneck is this sorting operation, order-independent transparency could likely solve this issue.

Finally, besides optimizations within the render passes individually, there is also room for high-level optimizations. Since the implementation needs to be able to manage dynamic scenes (dynamic virtual light fields), the entire virtual light field is recalculated for each frame. This is very redundant, because many parts of the scene are in fact static, there are just several (possibly small) dynamic objects. In principle, the diffuse global illumination for the static parts of the scene only needs to calculated once, whereas the illumination for the dynamic objects needs to be updated every frame. This could be an optimization that would significantly improve the performance. However, the implementation is definitely not trivial to add, and unfortunately there was too little time left within this thesis' time frame to do so. Additionally, it might not be necessary to recalculate the entire global illumination map **for each frame** in order to obtain an experience that is perceived as correct and real. It could be sufficient to calculate this every 2, 3, ... frames, which could considerably improve the performance.

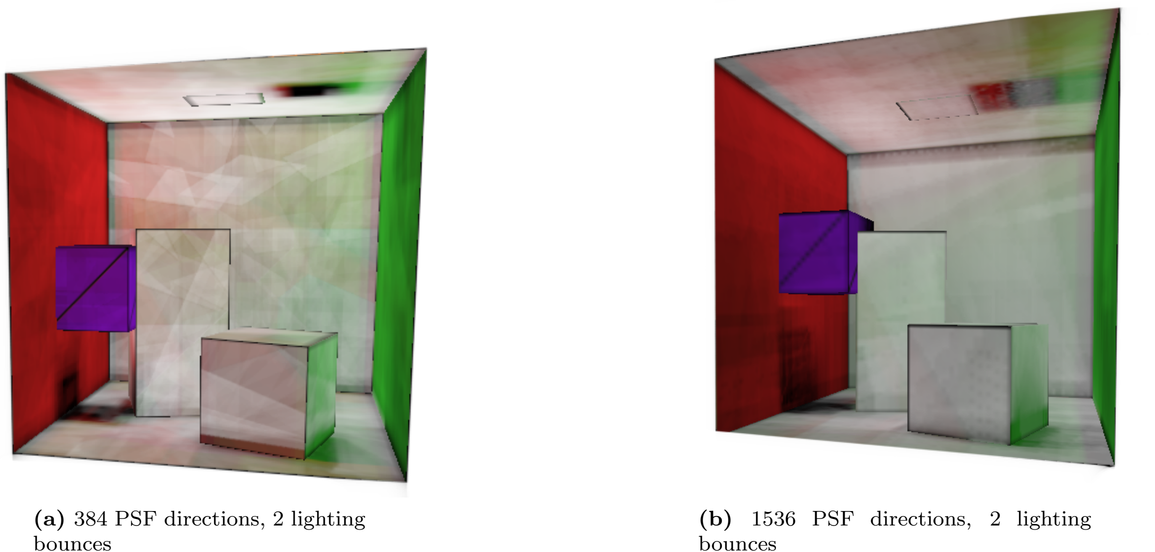
| Render pass | GPU time ($ms$) |
|---|---|
| Texture geometry | 0.02 |
| Emit radiance | 0.03 |
| Stencil routing | 0.04 |
| Blend light | 0.02 |
| Tone mapping + presentation | $< 0.01$ |

**Figure 4.2:** Render pass execution times in $ms$. Results obtained from profiling with NVIDIA Nsight.

## 4.6 Overall result and learning process

Since this chapter should also have some form of personal reflection on the subject and the obtained results, I would like to dedicate this section to reflect on the learning process. However the current performance of the implementation is not in its most optimal state yet, the goal of this thesis was reached. The thesis aimed to implement a representative proof-of-concept implementation for the *shared rendering cloud gaming* architecture. More specifically, the computation of diffuse global illumination of a scene needs to be handled by the server, in order to enhance a client that wants to render the same scene. Despite the limited time frame, we can conclude that this proof-of-concept implementation has been developed, and further enhancements can be added on top of it in the future. Different useful topics in the domain of *computer graphics* and *vision* were touched upon, such as (virtual) light fields, (diffuse) global illumination, ray tracing, tone mapping, rendering in texture space, *order-independent transparency* and more. The foundation for developing more interesting, new ideas with this technique has been made during this thesis.

Finally, to demonstrate the overall result, some screenshots of the final resulting client application in action is presented in figure 4.3. Note, once again, that the diffuse global illumination map was calculated at server side, sent over a direct peer-to-peer datachannel to the client web browser application, and there mapped onto the scene geometry to obtain the final result. The client has moved the cube by sending a movement command the the server. The server received this command, modified its internal model so the positions of the cubes are updated correctly and recalculated the diffuse global illumination accordingly. Lastly, the server sent back the updated diffuse global illumination map, and the updated positions of the movable cubes, which are then presented by the client application.



**(a)** 384 PSF directions, 2 lighting bounces

**(b)** 1536 PSF directions, 2 lighting bounces

**Figure 4.3:** Diffuse illuminated Cornell box computed with the virtual light field rendering method. Screenshots captured in web browser client. The texture resolution used for these screenshots is (512 x 512).

Unfortunately, the resulting implementation is not yet able to operate at a real-time performance. The lack of optimizations for dynamic virtual light fields in the back-end rendering application is the main cause of this. The employed rendering technique has a comparatively large memory complexity, although this should not be a major issue in a server environment. Furthermore, the render loop currently has a time complexity $O(l \text{ x } k)$, with $l$ the amount of lighting bounces and $k$ the amount of PSF-viewpoints. Note that this render loop needs to be performed for each frame, so in order to obtain an interactive frame rate this is currently too slow. Future optimizations should be able to resolve this, since there is a lot of redundant work that is currently being done (as mentioned in section 4.5).

## 4.7 Other methods of global illumination computation

This thesis was focused on calculating the diffuse component of global illumination using virtual light fields, but it might be useful to compare the technique presented in this thesis to other methods. Another important question to ask is when one would opt to use rendering through virtual light fields, and in which situations other methods would be more suitable. First of all, if this technique were to be adopted, it would likely not be used to compute direct lighting. Although it does have the capability to complete this task, there are several other techniques (e.g. *shadow mapping* [35]) that are more suitable in this situation. To understand this, it should be noted that, as discussed in section 4.3, the virtual light field implementation has a relatively large memory complexity because of the amount of texture data that needs to be stored. Therefore, a trade-off between texture resolution and performance has to be made. However, in a server environment with sufficient resources, this constraint might still allow high resolution textures. Direct lighting contains a lot of high frequency content (colors that are rapidly changing over a small distance, e.g. hard shadows). On the contrary, indirect lighting (e.g. color bleeding, diffuse global illumination) contains content of a much lower frequency. For content with high frequencies, a high texture resolution is required to ensure that the content is properly displayed. Lower texture resolutions reduce memory consumption and increase the performance of the technique used in this thesis. Since low frequency material is not susceptible to these lower texture resolutions, it makes this technique ideal for indirect illumination.

Another reason why this is the case is the impact of the light source's size, as discussed in section 3.2.4. This makes this method again less suitable for direct lighting, unless the only direct light source in the scene is a skylight. With smaller light sources, a higher amount of PSFs are required to avoid 'irradiation gaps', therefore decreasing the performance of this technique.

## 4.8 Future work

In the introduction of this chapter was mentioned that the implementation of this thesis is merely a preparatory work and foundation for future work. Since the implementation exists of many (sub)components, it has great potential to be optimized in the future. Furthermore, it leaves many open paths to still be explored. During the course of this thesis, I was already able to uncover several promising avenues that could be pursued in the future. Some of the most interesting ideas for the future will be discussed in this section.

### 4.8.1 Coherence of data

Like briefly discussed in 4.7, it is useful to mention that the virtual light field technique that was utilized in this thesis, profits from a much higher data coherence than traditional ray tracing methods. Additionally, in a virtual light field, many objects will stay in the same state as they were in during the previous frame (except for dynamic objects with a high probability to move), therefore, the data of the previous frame could be reused. Furthermore, each ray sent out from a certain PSF viewpoint is surrounded by other rays, whose data is stored very close to the data of the ray in question (the information is stored into the pixels of a texture). Additionally, if a scene would be subdivided into multiple virtual light fields, light will need to be propagated between these subdivisions as well, in which the coherence would play a significant role, the result from a certain subdivision could be reused in the contribution for its neighbours. This high coherence of data could play a very interesting role in future optimizations.

### 4.8.2 PSF interpolation

The coherence of data within virtual light fields has several benefits, one of which is the possibility to interpolate between PSFs. Instead of rendering a high amount of PSFs to cover a high amount of irradiance directions, one could opt to render a lower amount of PSFs and interpolate between each pair of PSFs to obtain results for intermediate spaces. Without the coherence of data, this would not be possible, because there would be no correlation between a random pair of rays built up from 2 different PSFs. More concisely, with virtual light fields, we can tell with certainty that all rays from PSF $A$ make the same angle with each ray from PSF $B$. Because the implementation's performance severely relies on the amount of PSFs that are rendered, lowering the amount of PSFs to be rendered is a very interesting concept to look into in the future.

### 4.8.3 Sparse resources and resident textures

Something else that was encountered during the course of this thesis is the concept of *sparse resources and resident textures* [12]. This is a more advanced method of resource allocation in Vulkan, which allows the implementer to allocate a memory block, and (re)bind resources to that memory block on the fly. A resource is not bound to a single continuous memory block, but is subdivided into smaller, equally sized parts or pages. Furthermore, not all pages of a certain resource are required to be bound to a certain memory block in order to be accessible by the GPU. Since the virtual light field rendering implementation makes use of a great amount of textures (even for a small scene), this dynamic streaming of data in and out of the video memory looks very interesting at first sight. It provides some form of virtual memory to the GPU, which allows the GPU to run applications that require more video memory than it disposes of.

### 4.8.4 Simplification of geometry

The current implementation renders the whole scene's geometry in multiple render passes, each frame. For the scene that was presented in this thesis, which is very low in complexity, that did not cause any issues performance-wise. However, for more complex scenes, this could be more problematic. Therefore, another optimization that could be done is simplifying the geometry by using *volumetric billboards* [8] or *voxel based rendering* [16] [7]. These topics are far from trivial, and do not fall under the scope of this thesis. However, to allow the render engine to support more complex scenes, it could be very interesting to look into these subjects in the future.

### 4.8.5 Synchronization of lighting

As the resulting application demonstrates, this *shared rendering* infrastructure ensures that the global illumination is being calculated and synchronized on the server, so each client has the same perception of lighting in the scene at all times (assuming that they have an internet connection of the same quality). Now another interesting possible future task is to find applications that can really benefit from this. In the introduction (section 1.3) of this thesis some toy example was given to demonstrate the usefulness of the topic, but there might be much more creative approaches that would demonstrate the effectiveness of this subject.

### 4.8.6 Specular reflection

An aspect that can definitely not be forgotten is the specular component of global illumination. Although this thesis only focused on diffuse global illumination, the proof-of-concept implementation that was set up can later be expanded to also support specular materials. Something that needs to be kept in mind, however, is that specular light has different frequencies of detail. It exists of a spectrum of diffuse, glossy and perfectly reflected light. It was mentioned before that this technique is less suitable for high frequency content. Therefore, the virtual light field method could be used for the diffuse and glossy parts of the spectrum. The perfectly reflected light can then be handled by traditional ray tracing techniques.

# Appendix A

# Nederlandstalige samenvatting

## A.1 Introductie

De gaming industrie is constant geëvolueerd doorheen de geschiedenis. Een van de meer recente innovaties in de gaming infrastructuur is het concept van *cloud gaming*. Cloud gaming verplaatst het zware werk dat het renderen en managen van een video game inhoudt van de client naar een server. De server streamt dan de resultaten naar de client. Gelijkaardig, maar niet hetzelfde, bestaat een *gedeelde render berekening (shared rendering computation)* infrastructuur uit een server die de client versterkt. Het verschil zit hier in het feit dat client en server beiden een vorm van rendering/managing verrichten, terwijl dat bij cloud gaming enkel de server hiervoor instaat. De server krijgt in het ideale geval de zwaardere taken toegewezen, de client verricht relatief lichte taken. Om overtollig werk te vermijden, kan de server best de taken uitvoeren die voor elke client gemeenschappelijk zijn. Op deze manier kan de client zijn rekenkracht sparen voor andere taken die het moet uitvoeren. Beter zelfs, een client die niet beschikt over de nodige rekenkracht om een bepaalde applicatie uit te kunnen voeren, kan mits ondersteuning van de server dit nu wel.

### A.1.1 Objectief

Deze thesis tracht een efficiënte techniek te ontwikkelen om een *Gedeelde Render Berekening Cloud Gaming Service (Shared Rendering Computation Cloud Gaming Service)* te organiseren. Meer specifiek, de computationeel intensieve taak die aan de server wordt toegewezen is het berekenen van de **diffuse globale belichting** in een virtuele scene. De berekening van globale belichting is een moeilijke taak om te volbrengen in real-time. Het vereist dat de render applicatie een vorm van ray tracing hanteert, wat snel een zware taak wordt voor complexe scenes. Het idee is om deze taak toe te wijzen aan de server applicatie, die het resultaat opslaat in een diffuse globale belichtingsmap (een textuur), en deze dan over het netwerk naar de client verstuurt. De client rest dan nog de relatief simpele taak om de geometrie van de scene te renderen, en hierop de textuur te mappen. In de context van single player games betekent dit dat de client applicatie veel minder werk moet verrichten. Een interessanter scenario is dat van een multiplayer game. In dit geval moet de diffuse globale belichting slechts éénmaal worden berekend, aangezien dat deze voor alle clients gelijk is. Dit komt omwille van het *diffuus oppervlak*, dat het licht uniform reflecteert in alle richtingen. Dat betekent dat het waargenomen licht voor een bepaald punt in de scene hetzelfde is, ongeacht het standpunt of de kijkrichting van de kijker (aangenomen dat enkel de diffuse component van de globale belichting wordt beschouwd). Om deze reden kan een groot aantal overtollige berekeningen vermeden worden door de diffuse lichtberekening te centraliseren op een server, en het resultaat hiervan te verdelen over iedere client.

Ten slotte, omdat het tijdsbestek van deze thesis relatief beperkt is en de diffuse belichting een zeer specifiek voordeel oplevert, zoals hierboven beschreven, beschouwt de implementatie gemaakt tijdens deze thesis enkel de diffuse component van de globale belichting. De speculaire component is desalniettemin ook belangrijk om een realistisch resultaat te bekomen. Hoewel deze thesis niet de focus zal leggen op dit aspect, kan het toevoegen van de speculaire belichting beschouwd worden als toekomstig werk. Ondanks dit, levert de diffuse component op zichzelf een representatief resultaat waarvan aanzienlijke voordelen van afgeleid kunnen worden.

### A.1.2 Probleemgebieden

De twee disciplines die het gebied van deze thesis definiëren zijn Networking en Computer Vision. Er zullen uitdagingen zijn in beide velden. Een systeem dat een video game afspeelt voert continu berekeningen uit. In de context van deze thesis, zal de server de zwaarste berekeningen uitvoeren. Daarom is het wenselijk dat er een efficiënte graphics pipeline wordt ontwikkeld zodat de berekeningen van de server zo performant mogelijk kunnen verlopen. Bovendien moet de server een update uitvoeren in de globale belichting elke keer dat er een wijziging aan de scene wordt aangebracht die de belichting beïnvloedt. Bij elke wijziging moet dan ook de diffuse globale belichtingsmap opnieuw worden verstuurd naar de client, zodat deze ook de wijzigingen kan doorvoeren. Dit allemaal tezamen moet gebeuren op real-time frame rates, om een goede ervaring te kunnen verzekeren. Hoewel dat dit een probleem is dat zeker niet triviaal is om op te lossen, tracht deze thesis om een werkende, representatieve implementatie te ontwikkelen.

## A.2 Implementatie

De implementatie van deze thesis bestaat uit twee high-level componenten: de client en de server. Deze sectie zal eerst de high-level architectuur van de applicatie kort bespreken, waarna meer details voorzien worden voor iedere component.

### A.2.1 High-level architectuur

Gegeven het onderzoeksprobleem, zou de volgende opsomming een gesimplificeerde verzameling van vereisten voor de applicatie kunnen zijn:

1. De server applicatie moet de diffuse globale belichting in de scene berekenen, en deze opslaan in een textuur.
2. De client applicatie, die een textuur ontvangt waar de diffuse globale belichting is 'ingebakken', moet de geometrie van de scene renderen en deze textuur hierop mappen. Zo verkrijgt deze de resulterende diffuus belichte scene.
3. De applicatie moet een bepaalde stroom van data voorzien tussen de server en client (om de texturen over te brengen).
4. De applicatie moet een interactieve methode voorzien om de belichting van de scene te beïnvloeden (om te demonstreren dat de applicatie overweg kan met dynamische veranderingen in belichting).
5. De service moet beschikbaar zijn op de meeste platforms met een internetverbinding.

Om vereiste 5 te vervullen, werd de keuze gemaakt om de client applicatie in een web browser te implementeren. Naast multi-platform ondersteuning heeft dit nog andere voordelen, zoals het direct gebruiksklaar zijn van de applicatie.

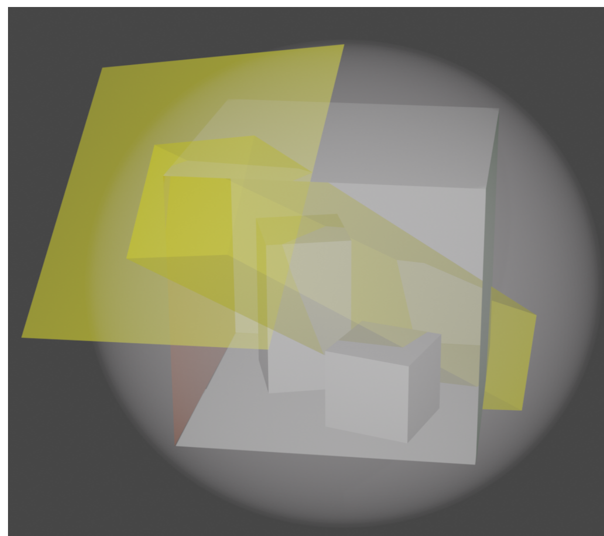
Het streaming mechanisme dat werd gebruikt om de diffuse globale belichtingsmap en andere arbitraire data tussen client en server te streamen werd geïmplementeerd door gebruik te maken van *WebRTC specificatie*. Meer specifiek, WebRTC's datachannels werden gebruikt om een constante bidirectionele arbitraire data stream op te zetten. Aangezien dat WebRTC gebruikt wordt om peer-to-peer connecties op te zetten, wordt de server beschouwd als een speciale peer, om zo de server van de clients te kunnen onderscheiden. Aangezien dat de peers elkaars IP adres niet kennen, kan de peer-to-peer connectie niet direct worden opgezet. Om dit op te lossen, is er tussenin een extra server voorzien (de signalerende server) die de server peer en client peer met elkaar laten 'handjes schudden'. Zo kunnen de client peer en server peer elkaars IP adres te weten komen. Voor meer gedetailleerde informatie over WebRTC wordt de lezer verzocht om sectie 2.2 te raadplegen.

### A.2.2 Server (rendering met virtuele lichtvelden)

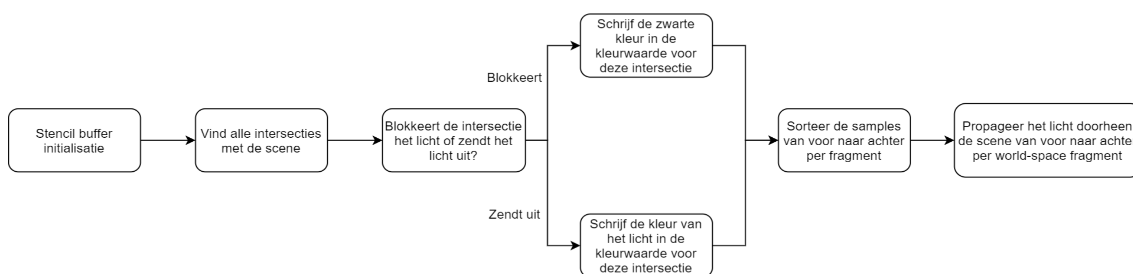
De server applicatie neemt de verantwoordelijkheid voor het berekenen en voorzien van de diffuse globale belichtingsmap van de gerenderde scene. Hiervoor werd een render engine die een scene rendert met behulp van een *virtueel lichtveld*. Een zeer korte samenvatting voor de term 'virtueel lichtveld' is: "*De complete verzameling van alle lichtstralen in een virtueel volume*". Dit is echter slechts een zeer gelimiteerde definitie van een virtueel lichtveld. Omdat deze samenvatting enkel instaat voor het kort

samenvatten van deze thesis, wordt de lezer verzocht om sectie 2.1 te raadplegen voor een uitgebreidere introductie tot dit onderwerp.

Het algoritme (de render loop) uitgevoerd door deze engine is als volgt: De scene wordt omvat door een virtuele, denkbeeldige bol. Een uniform verdeeld punt $P$ op deze bol wordt genomen. De camera wordt in punt $P$ geplaatst, hierna wordt er een orthogonale projectie gemaakt gericht naar de oorsprong van de scene, die overlapt met het middelpunt van de bol. Het volume dat wordt bedekt door deze orthogonale projectie noemt men een *parallel subveld* (PSF, van het Engelse *parallel subfield*). In het beeld dat resulteert uit deze projectie wordt er voor elke pixel een straal uitgezonden. Dit resulteert in $n$ x $m$ stralen, met $n$ x $m$ de resolutie van het beeld. Om kleur en diepte informatie op te kunnen vangen voor meer dan één dieptelaag werden er multisampled framebuffers gebruikt. Een visuele representatie van een parallel subveld is voorgesteld in figuur A.1. De volgende stappen in de render loop voeren een techniek uit genaamd *stencil routed a-buffer*, gelijkaardig aan hetgeen voorgesteld in [25]. Deze stappen worden uitgevoerd per PSF straal en zijn samengevat in figuur A.2. Verder wordt dit proces herhaald voor $k$ uniform verdeelde punten $P_i$ op de bol, met $i \in [0; k[$, en het kan mogelijk ook herhaald worden voor meerdere lichtbotsingen. De eerste lichtbotsing stelt de directe belichting van de scene voor.



**Figuur A.1:** Visuele voorstelling van een *parallel subveld*. De scene wordt volledig omringd door een denkbeeldige bol, hier voorgesteld in doorzichtig wit. Een punt $P$ wordt genomen op de bol, waaruit een orthogonale projectie wordt gemaakt richting de oorsprong van de scene. Dit wordt hier voorgesteld door een gele lichtbundel (niet alle stralen zijn getekend). Merk op dat een PSF de volledige scene bedekt. Namelijk, voor ieder PSF snijdt elk punt in de scene met precies één straal van dat PSF.




**Figuur A.2:** Gesimplificeerde schematische voorstelling van het *stencil routed a-buffer* algoritme, dat wordt uitgevoerd voor iedere straal in een PSF.

Vervolgens, om de finale diffuse globale belichting te bekomen, moeten de tussenresultaten van elk kijkpunt en lichtbotsing worden gecombineerd, omdat ze allemaal bijdragen aan het eindresultaat. Dit werd opgelost door additieve color blending toe te passen. Bovendien werd er een extra stap van multiplicatieve color blending toegepast om de inkomende radiantie van de huidige lichtbotsing om te zetten in uitgaande radiantie voor de volgende lichtbotsing. Voor de doeleinden van deze thesis volstaat dit als benadering van een echte *Lambertiaanse diffuse bidirectionele reflectie distributie functie [6]*, terwijl dit

de implementatie een stuk simplificeert.

Ten slotte, aangezien de kleurwaarden verschillende additief werden gemengd, zal het eindresultaat een veel hogere *dynamisch bereik* hebben dan het $[0; 1]$ bereik waarin de *RGBA* tupels (kleurwaarden) zijn gecodeerd. Dit zou resulteren in een scene die volledig wit is, wat onacceptabel is. Om dit op te lossen werd er een *tone mapping* algoritme gebruikt. Een tone mapping algoritme, zoals de naam impliceert, mapt de kleurwaarden van een hoger dynamisch bereik terug naar het $[0; 1]$ bereik zodat deze weergegeven kunnen worden op non-HDR displays. In de implementatie van deze thesis werd het *Reinhard tone mapping [30]* algoritme gehanteerd om deze taak te vervullen.

### A.2.3 Client (en client-server communicatie)

De client applicatie zal worden versterkt door de diffuse globale belichtingsmap die het ontvangt van de server. Verder heeft het ook nog de relatief lichte taak om de geometrie van de scene te renderen. Het zal de diffuse globale belichtingsmap op deze geometrie mappen om de resulterende belichte scene te bekomen. Een bijkomende opmerking is dat de server al het zware werk heeft uitgevoerd, namelijk het berekenen van de diffuse globale belichting. De gebruiker van de client applicatie heeft de mogelijkheid om een kubus die aan hen werd toegewezen te bewegen d.m.v. toetsenbord input. Deze input triggert een event dat een commando naar de server verstuurt, die dan positie van de kubus in kwestie zal updaten. De diffuse globale belichting in de geüpdatete scene wordt herberekend, waarna de nieuwe textuur naar de client wordt verzonden, zodat deze de belichting updates kan zien.

De client-server communicatie wordt beheerd door een simpel custom protocol. Dit protocol regeert de data die wordt uitgewisseld tussen client en server. Er zijn momenteel 3 situaties waarin het nodig is dat de server en client met elkaar communiceren:

1. Het verzenden van de diffuse globale belichtingsmap (server naar client)
2. Het verzenden van de (nieuwe) huidige posities van de beweegbare kubussen (server naar client)
3. Het verzenden van beweging commando's om een kubus te bewegen (client naar server)

In situatie 1, elke keer dat er een nieuwe textuur beschikbaar is aan server kant, zendt de server deze in de datachannel (opgezet tussen server en client) in de vorm van een binaire byte stream, die wordt herkend door de client. Het einde van een binnenkomende textuur wordt aangegeven door een bericht van volgend formaat: `'PNG_TRANSFER_COMPLETE'`.

Volgens situatie 2 zendt de server periodiek naar iedere client een bericht van het volgend formaat: `'x1!y1/x2!y2'`. `'(x1;y1)'` en `'(x2;y2)'` stellen respectievelijk de coördinaten van kubus 1 en 2 voor (er werden slechts 2 beweegbare kubussen voorzien in deze implementatie).

Ten slotte, volgens situatie 3 krijgt iedere client een beweegbare kubus toegewezen. De commando's die de client kan verzenden naar de server zijn van het volgende formaat: `'up'`, `'down'`, `'left'`, `'right'`. De server maakt dan het onderscheid tussen deze commando's en update de positie van de kubussen afhankelijk van welke client het commando verstuurde.

## A.3 Resultaten en conclusies

Om de resulterende implementatie te demonstreren werd er een klein experiment opgezet waarin twee clients met de server verbinden. Elke client krijgt een beweegbare kubus toegewezen, die ze kunnen bewegen m.b.v. toetsenbord input. Aan de hand van deze bewegingen wordt de diffuse globale belichting door de server geüpdatet. Ik heb twee configuraties van het virtuele lichtveld kunnen testen:

1. 384 PSF-kijkpunten, 2 lichtbotsingen
2. 1536 PSF-kijkpunten, 2 lichtbotsingen

Meer lichtbotsingen / PSF richtingen kunnen wensbaar zijn, afhankelijk van de scene. Helaas was het door hardware gerelateerde beperkingen niet mogelijk om met hogere configuraties te testen tijdens deze thesis.

Er werd verder ook geëxperimenteerd met verschillende parameters. Één van deze parameters is het formaat van de texturen. Er werd getest met beiden het *JPEG* [26] en *PNG* [33]. Er was geen merkbaar

verschil in performantie tussen deze twee, hoewel dat de grootste flessenhals leek te zitten in het encoding/decoding proces. Omwille hiervan werd er een alternatieve methode gebruikt om deze flessenhals te omzeilen. Deze methode verzendt ruwe textuur data over het netwerk, zodat er geen encoding/decoding hoeft te gebeuren. Dit leverde al een significante verbetering in de performantie. Desondanks werd de finale implementatie genoodzaakt om terug het PNG formaat te hanteren, door problemen met het mappen van ruwe textuur data in het front-end framework. Dit is echter een probleem wat hoogstwaarschijnlijk kan opgelost worden in de toekomst.

Verder moet er opgemerkt worden dat door geheugen limitaties (512 x 512) de maximale textuur resolutie is die gebruikt kon worden voor de diffuse globale belichtingsmap. Het is ook vermeldenswaard dat de textuur resolutie een grote impact heeft op de performantie van de applicatie, omdat het de hoeveelheid van video geheugen dat gebruikt wordt bepaalt. Om een interactieve ervaring te bekomen met de hardware setup gebruikt tijdens deze thesis, werd ik genoodzaakt om (256 x 256) texturen te gebruiken, hoewel dat sommige statische schermafbeeldingen gemaakt werden op de (512 x 512) resolutie.

Belangrijker in termen van performantie, werd er bij het evalueren van de render loop gevonden dat de hoofdzakelijke flessenhalsen zitten in de *stencil routing* en *emit radiance* render passes. De *emit radiance* render pass rendert naar samples van een multisampled textuur. De lagere performantie zou eventueel verklaard kunnen worden door het feit dat deze render pass naar 8 samples per fragment moet renderen, in plaats van één. Ten tweede is de *stencil routing* pass, de render pass die samples van voor naar achter sorteert en het licht doorheen de scene propageert. De hoogste kost zit hier in de sorteeroperatie. Twee sorteeralgoritmen, namelijk *bitonic sorting* en een *compare and swap* algoritme werden uitgeprobeerd. Het *compare and swap* algoritme kwam er duidelijk uit als winnaar. Het aantal per-fragment sorteeroperaties die moeten gebeuren heeft een grote impact op de performantie. Verder kunnen er alternatieve methoden beschouwd worden in de toekomst, zoals het voor naar achter sorteren van veelhoeken in plaats van samples. Dit zou niet resulteren in een 100% accurate oplossing in elke situatie, maar het zou zeker voldoende kunnen zijn voor de doeleinden van deze applicatie. Omdat de huidige implementatie de grote hoeveelheid van overtollig werk voor dynamische virtuele lichtvelden nog niet in acht neemt, zijn er ook nog zeker high level optimalisaties mogelijk. Een ander interessant onderwerp waarnaar gekeken kan worden in verband met de stencil routing pass is *order-independent transparency [9] [31]*.

Bij het evalueren van de finale implementatie (degene gebruikt voor het experiment) is de conclusie dat de huidige applicatie helaas nog niet genoeg is geoptimaliseerd om real-time frame rates te kunnen behalen. Desalniettemin zijn de optimalisaties die hierboven werden opgemerkt hoogstwaarschijnlijk al voldoende om een real-time performance te behalen. **De gebruikte rendering techniek heeft een relatief grote geheugencomplexiteit, hoewel dit niet een ernstig probleem zou mogen vormen in een server omgeving. Bovendien heeft de render loop, die momenteel nog geen verdere optimalisaties heeft voor dynamische virtuele lichtvelden, een tijdscomplexiteit van $O(l \text{ x } k)$, met $l$ het aantal lichtbotsingen en $k$ het aantal PSF-kijkpunten.**

Ten slotte is nuttig om de techniek die wordt voorgesteld in deze thesis te vergelijken met andere methodes. Indien de methode die virtuele lichtvelden hanteert ooit commercieel gebruikt zal worden, zal het waarschijnlijk niet zijn om directe belichting te berekenen. Hoewel het wel de mogelijkheid biedt om dit te doen, zijn er voor directe belichting alternatieve technieken (bv. *shadow mapping [35]*) die meer geschikt zijn voor dit doeleinde. Voor indirecte belichting is deze techniek echter zeer geschikt, aangezien het feit dat indirecte belichtingseffecten bestaan uit laag frequent materiaal, in tegenstelling tot directe belichtingseffecten. Voor laag frequent materiaal volstaan texturen met een lagere resolutie vaak, wat voordelig is voor de virtueel lichtveld techniek, omdat haar geheugenverbruik exponentieel opschaalt met texturen van een hogere resolutie. Dit zou leiden tot een minder performante applicatie.

# Bibliography


[1] History of video games - four decades of video entertainment. https://cdn.hackaday.io/files/1649347056536256/History%20of%20Video%20Games-Four%20Decades%20of%20Video%20Entertainment.pdf. Accessed: 2021-05-09.

[2] Ice - mozilla developer. https://developer.mozilla.org/en-US/docs/Glossary/ICE. Accessed: 2021-03-21.

[3] Statista - number of internet of things (iot) connected devices worldwide in 2018, 2025 and 2030. https://www.statista.com/statistics/802690/worldwide-connected-devices-by-access-technology/. Accessed: 2021-04-06.

[4] Webrtc: Signaling and video calling - mozilla developer. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling. Accessed: 2021-03-21.

[5] Ian Ashdown and Marc Salsbury. A near-field goniospectroradiometer for led measurements. 06 2006. Accessed: 2021-05-17.

[6] Robert L Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics (ToG)*, 1(1):7–24, 1982. Accessed: 2021-05-20.

[7] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 15–22, 2009. Accessed: 2021-05-20.

[8] Philippe Decaudin and Fabrice Neyret. Volumetric billboards. In *Computer Graphics Forum*, volume 28, pages 2079–2089. Wiley Online Library, 2009. Accessed: 2021-05-21.

[9] Eric Enderton, Erik Sintorn, Peter Shirley, and David Luebke. Stochastic transparency. *IEEE transactions on visualization and computer graphics*, 17(8):1036–1047, 2010.

[10] Kenneth Paul Fishkin. Color mixture in computer graphics. *Winconsin Academy of Sciences, Arts, and Letters*, 71(2):41–44, 1983. Accessed: 2021-05-04.

[11] Donald P Greenberg, Michael F Cohen, and Kenneth E Torrance. Radiosity: A method for computing global illumination. *The Visual Computer*, 2(5):291–297, 1986. Accessed: 2021-03-02.

[12] The Khronos Group. 32. sparse resources - vulkan® 1.1.178 specification. https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#sparsememory. Accessed: 2021-05-14.

[13] The Khronos Group. The opengl® shading language (4.50) - specification. https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf. Accessed: 2021-05-11.

[14] The Khronos Group. Vulkan® 1.1.178 - a specification. https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html. Accessed: 2021-05-11.

[15] Anmol Gupta and Kamlesh Dutta. Cloud gaming: Architecture and quality of service. *CPUH-Research Journal*, 1(2):19–22, 2015. Accessed: 2021-02-01.

[16] Taosong He, Lichan Hong, Arie Kaufman, Amitabh Varshney, and Sidney Wang. Voxel based object simplification. In *Proceedings Visualization'95*, pages 296–303. IEEE, 1995. Accessed: 2021-05-21.

[17] Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, 1986. Accessed: 2021-05-04.

[18] IETF. Rfc 6455: The websocket protocol. https://datatracker.ietf.org/doc/html/rfc6455. Accessed: 2021-05-11.

[19] Bart Jansen, Timothy Goodwin, Varun Gupta, Fernando Kuipers, and Gil Zussman. Performance evaluation of webrtc-based video conferencing. *ACM SIGMETRICS Performance Evaluation Review*, 45(3):56–68, 2018. Accessed: 2021-04-06.

[20] Henrik Wann Jensen, James Arvo, Phil Dutre, Alexander Keller, Art Owen, Matt Pharr, and Peter Shirley. Monte carlo ray tracing. In *ACM SIGGRAPH*, volume 5, 2003. Accessed: 2021-05-22.

[21] Chris Kiser, Erik Reinhard, Mike Tocci, and Nora Tocci. Real time automated tone mapping system for hdr video. In *IEEE International Conference on Image Processing*, volume 134. IEEE Orlando, FL, 2012. Accessed: 2021-05-07.

[22] Chang Liu, Wei Tsang Ooi, Jinyuan Jia, and Lei Zhao. Cloud baking: Collaborative scene illumination for dynamic web3d scenes. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 14(3s):1–20, 2018. Accessed: 2021-02-01.

[23] Jesper Mortensen. *Virtual light fields for global illumination in computer graphics*. PhD thesis, UCL (University College London), 2011.

[24] Jesper Mortensen, Pankaj Khanna, and Mel Slater. Light field propagation and rendering on the gpu. In *Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 15–23, 2007. Accessed: 2021-02-01.

[25] Kevin Myers and Louis Bavoil. Stencil routed a-buffer. In *SIGGRAPH Sketches*, page 21, 2007. Accessed: 2021-05-15.

[26] JPEG org. Jpeg 1. https://jpeg.org/jpeg/. Accessed: 2021-05-10.

[27] React.js org. React - a javascript library for building user interfaces). https://reactjs.org/. Accessed: 2021-05-10.

[28] The Overflow. The top 10 frameworks and what tech recruiters need to know about them. https://stackoverflow.blog/2019/12/17/the-top-10-frameworks-and-what-tech-recruiters-need-to-know-about-them. Accessed: 2021-05-10.

[29] Jeroen Put. Dynamische virtuele lichtvelden voor games. Accessed: 2021-05-20.

[30] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 267–276, 2002. Accessed: 2021-05-07.

[31] Marco Salvi, Jefferson Montgomery, and Aaron Lefohn. Adaptive transparency. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 119–126, 2011.

[32] Mel Slater. A note on virtual light fields. *University College London*, 2000. Accessed: 2021-02-01.

[33] W3C. Portable network graphics (png) specification (second edition). https://www.w3.org/TR/2003/REC-PNG-20031110/. Accessed: 2021-05-10.

[34] W3C. Webrtc 1.0: Real-time communication between browsers. https://www.w3.org/TR/webrtc/. Accessed: 2021-04-06.

[35] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274, 1978. Accessed: 2021-05-21.