REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mohamed Khider – BISKRA
Faculté des Sciences Exactes, des Sciences de la Nature et de la Vie
Département d'informatique

N° d'ordre IVA4/M2/2021

Mémoire

Présenté pour obtenir le diplôme de master académique en

Informatique

Parcours : **Image et Vie Artificielle (IVA)**

Direct illumination using cube mapping technique on GPU

Par :

RAHAL HAKIMA RYM

Soutenu le 06/07/2021 devant le jury composé de :

Nom Prénom	grade	Président
BABAHENINI Djihane	MCB	Rapporteur
Nom Prénom	grade	Examinateur

Année universitaire 2020-2021

Dedication

This work is dedicated to my parents, who have raised me to the person I am today, and to the rest of my big family, and to my friends, specially my teacher Djihane Babahenini for helping me develop my knowledge and skills, thank you all for everything.

Acknowledgements

First, I want to thank almighty ALLAH for giving me the will, patience and health to develop this work.

I would like to express my profound gratitude to my supervisors Dr. Babahenini Djihane, for her involvement in this research work and for the support they have given me, their patience, their availability and the relevance of their advice that have been of invaluable assistance throughout this work.

Also, I would like to thank all my professors in the computer science department.

I extend my sincere thanks to the members of the jury, for having accepted to judge this work.

Finally, I thank my big family and my friends for encouraging me during this year, and to have always been available when I needed it.

Abstract

In this thesis, we present a Cube Mapping algorithm via the GPU for the 3D computer graphics direct lighting, this algorithm uses a special type of texture called the cube map texture, to calculate the direct lighting in each point of the scene, this technique can help accelerating the execution time. We first load a 3D scene, and then we generate the cube map and use it to calculate the direct lighting of this scene. Then we render our final image. In the stage of generating the cube map we could develop an advanced method that eliminates 5 rendering passes from the whole program, keeping only one for generating the cube map plus the display pass, which accelerates the calculation time even more. The test results show that the method can generate the final image in small calculating time with a good quality.

Key-words : Cube Map, Direct Lighting, Calculating Time, Quality, GPU.

Résumé

Dans cette mémoire, nous présentons un algorithme de Cube Mapping via le GPU pour l'éclairage direct de l'infographie 3D, cet algorithme utilise un type particulier de texture appelé la texture cube map, pour calculer l'éclairage direct en chaque point de la scène, cette technique peut aider à accélérer le temps d'exécution. Nous chargeons d'abord une scène 3D, puis nous générons le cube map et l'utilisons pour calculer l'éclairage direct de cette scène. Ensuite, nous rendons notre image finale. dans la partie de la génération de cube map, nous pourrions développer une méthode avancée qui élimine 4 passes de rendu de l'ensemble du programme, ce qui accélère encore plus le temps de calcul. Les résultats des tests montrent que la méthode peut générer l'image finale en un temps de calcul réduit avec une bonne qualité.

Mots-clés : Cube Map, Éclairage direct, Temps de calcul, Qualité, GPU.

ملخص

في هذه الأطروحة ، نقدم خوارزمية رسم خرائط المكعب عبر وحدة معالجة الرسومات للإضاءة المباشرة لرسومات الكمبيوتر ثلاثية الأبعاد ، وتستخدم هذه الخوارزمية نوعًا خاصًا من النسيج يسمى نسيج خريطة المكعب ، لحساب الإضاءة المباشرة في كل نقطة من المشهد ، ويمكن لهذه التقنية تساعد في تسريع وقت التنفيذ. نقوم أولاً بتحميل مشهد ثلاثي الأبعاد ، ثم نقوم بإنشاء خريطة المكعب ونستخدمها لحساب الإضاءة المباشرة لهذا المشهد. ثم نقدم صورتنا النهائية. في مرحلة إنشاء خريطة المكعب ، يمكننا تطوير طريقة متقدمة تقضي على ٥ تمريرات عرض من البرنامج بأكمله ، مع الاحتفاظ بواحد فقط لإنشاء خريطة المكعب بالإضافة إلى ممر العرض ، مما يؤدي إلى تسريع وقت الحساب أكثر. تظهر نتائج الاختبار أن الطريقة يمكن أن تولد الصورة النهائية في وقت حساب صغير بجودة جيدة.

الكلمات المفتاحية: خريطة المكعب ، الإضاءة المباشرة ، حساب الوقت ، الجودة ، وحدة معالجة الرسومات.

Contents

List of Figures

Introduction

The 3D Graphics Rendering Pipeline accepts the description of 3D objects in terms of vertices of primitives (such as triangle, point, line, and quad), and produces the color-value for the pixels on the display. One of the most important steps in the pipeline is the lighting step. For most of us lighting is something usual, something we do not pay must attention to. 90 percent of our experience with light is absolutely passive. But in 3D computer graphics and design lighting is an essential aspect!

Sometimes we can see that a well-modeled 3D object looks flat or has no specific shape because of the bad lighting. Well-chosen lighting techniques can help giving a meaning to our scene. There are numbers of well established 3D lighting paradigms, and the type of scene usually determines which one is most appropriate. For example, techniques that work well for an interior environment usually doesn't have an effect in exterior scenes. Though they may give excellent results when it comes to quality, not all lighting techniques are suitable for real-time rendering applications such as interactive video games, simulators... etc. which requires finding other solutions or techniques that can approximate the effect of these expensive techniques but with lower calculation time.

This problem leads us to many questions such as what are the techniques that can give us good results with an acceptable rendering time? Which technique is the most favorable? Can these techniques work in real-time?

This thesis describes a widely used lighting technique that can eliminate usual 3D lighting problems such as the quality and the calculating time by approximating the real calculations of the lighting of the 3D scenes.

The main goal of this thesis is to generate an image with a good quality of a lighted 3D scene generated in an acceptable calculating time.

The rest of this thesis is organized as follows: chapter 1 gives a brief Direct and Indirect Illumination and the different Radiometric quantities and types of light sources. The different types of Environment Mapping and the cube mapping technique are represented in chapter 2.

The conception of this work is described in chapter 3. And the implementation details, results, and discussions are presented in chapter 4. Conclusions are in the end.

Chapter 1

Direct and Indirect Illumination

1.1 Introduction

Lighting (illumination) represents the energy that transports from the light source to the points of a surface. What the human eye sees is a result of light coming off of an object or other light source to the receptors in the eye.
3D Lighting is a collection of tools and techniques used to simulate light in a computer-generated 3D environment. There are several 3D lighting techniques that can offer a huge amount of flexibility regarding the level of detail and functionality. They also operate at different levels of complexity. Lighting artists can choose from a variety of light sources, effects, tools, and techniques that suit their needs.

1.2 Radiometric quantities

Radiometric quantities are quantities related to electromagnetic radiation.

1.2.1 Radiant Energy

Radiant Energy is the energy carried from any electromagnetic field.
- It is denoted by Qe.
- Its SI unit is the joule (J).

1.2.2 Radiant Flux

Radiant Flux is the radiant energy per unit time (also called Radiant Power); it is considered the fundamental radiometric unit.
- It is denoted by Pe.
- Its SI unit is the watt (W).

1.2.3 Radiant Exitance

Radiant Exitance, or Radiant Emittance, is the radiant flux emitted from an extended source per unit area.
- It is denoted by Me.
- Its SI unit is: watt per square meter (W/m^2)

1.2.4 Irradiance

Irradiance is the radiant flux incident on a surface unit area.
- It is denoted by Ee.
- Its SI unit is: watt per square meter (W/m^2).

1.2.5 Radiant Intensity

Radiant Intensity is the radiant flux emitted from a point source per unit solid angle.
- It is denoted by Ie.
- Its SI unit is: watt per steradian (W/sr).

1.2.6 Radiance

Radiance is radiant flux emitted from an extended source per unit solid angle and per unit projected source area.
- It is denoted by Le.
- Its SI unit is: watt per steradian and square meter ($W/(sr.m^2)$).[1]

1.3 Types of light sources

Before setting the lighting it is necessary to decide which kind of lighting type will be used in the scene since there are several types of computer-generated light sources (such as point source light (Omni light), spotlight, directional light, area light...etc).

1.3.1 Directional Light

A directional light (also known as infinite light) is great for simulating the sun or moonlight because the light it creates travels in parallel rays; like a far-away light source. This light source strikes the polygons in the scene with equal intensity. The next figure represents the effect of the directional light.[2]

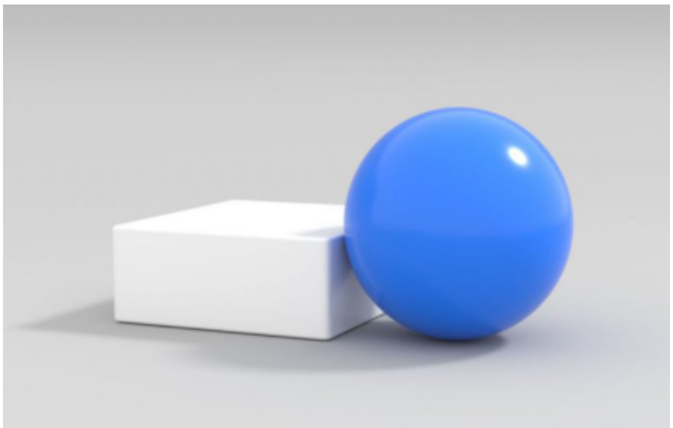


Figure 1.1 – Directional Light [2]

1.3.2 Point light

Point light (also known as Omni light), emits light from a single small point, in all directions. It is often used to create fill light because it has no specific shape or size. The closer the object is to the light source, the brighter it appears. A light bulb is an example of Omni/point lights in the real world. The next figure represents the effect of the point light.[2]

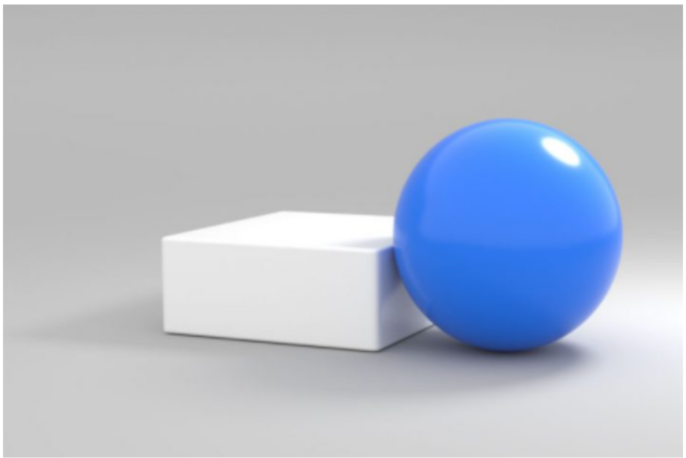


Figure 1.2 – Point light [2]

1.3.3 Area Light

Area light sources are common in the real world, and thus important in realistic images[3]. An area light emits light from a specified surface with a specific shape and size; such as a window, fluorescent light fixture, or back-lit panel. In other words, an area light is a physically-based light that casts directional rays from within a specified boundary; creating soft and realistic shadows. These properties make it a popular choice in product lighting or architectural visualization. Area lights do have an overall direction, but they don't emit parallel rays like a directional light. The next figure represents the effect of the area light.[2]

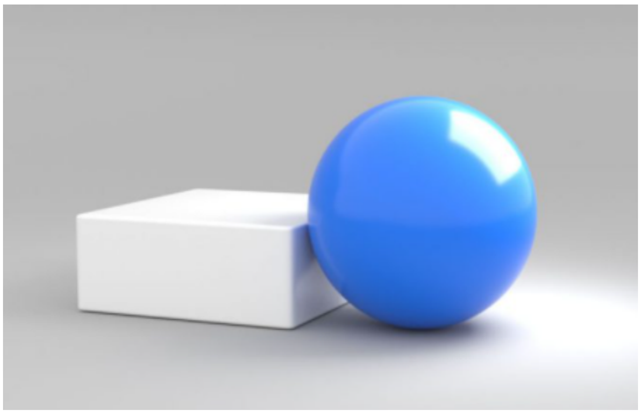


Figure 1.3 – Area Light [2]

1.3.4 Spotlight

A spotlight produces a cone of light in a single direction; with the light getting more intense closer to the source and to the center of the cone. The lighting artist can control the cone angle, determine the size of the light or soften the outside edge of the cone to create different looks. A flashlight is an obvious example of spotlights in the real world. The next figure represents the effect of the spotlight.[2]

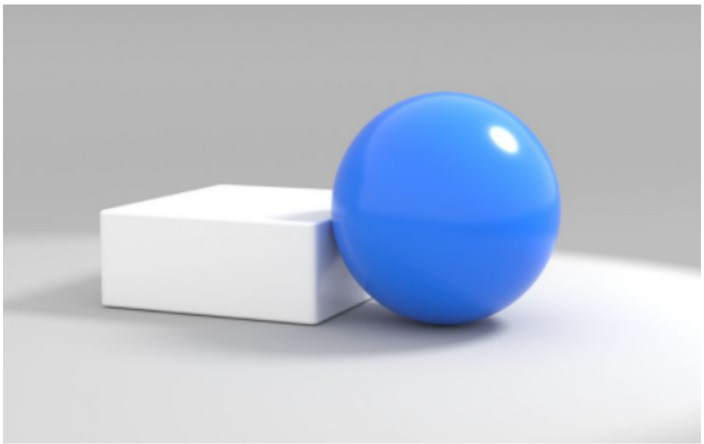


Figure 1.4 – Spotlight [2]

1.4 Direct Illumination

Direct Illumination is the lighting/shading method that was originally used by computer graphics programs. When rendering, direct illumination considers only light from the original source, there are no bounce lights or light emissive polygons (or sources other than lights) will add to the lighting within the rendered scene. the final color of an object is actually a combination of 3 different components Ambient, Diffuse, Specular Reflections (Phong Model), where:

Final Color = Ambient + Diffuse + Specular.

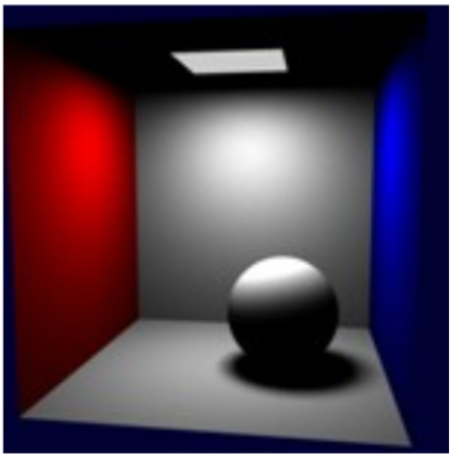


Figure 1.5 – Direct Illumination [4]

1.4.1 Ambient Reflection

Even when it is dark there is usually still some light somewhere in the world (the moon, a distant light) so objects are almost never completely dark. To simulate this we use an ambient lighting constant that always gives the object some color[5]. Creates the effect of having light hit your object equally from all directions, where:

I = Ia . Ka

I: intensity.
Ia: intensity of Ambient light.
Ka: object's ambient reflection coefficient, 0.0 - 1.0 for each of R, G, and B.
The next figure represents the result of ambient light.

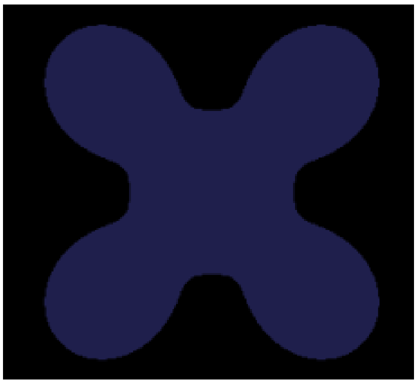


Figure 1.6 – Ambient Reflection [6]

1.4.2 Diffuse Reflection

Simulates the directional impact a light object has on an object. This is the most visually significant component of the lighting model. The more a part of an object faces the light source, the brighter it becomes[5], where:

I = Ip . Kd . (N' * L')

I: intensity.

Ip: intensity of point light.

Kd: object's diffuse reflection coefficient, 0.0 - 1.0 for each of R, G, and B.

N': normalized surface normal.

L': normalized direction to light source.

(N' * L'): represents the dot product of the two vectors.

The next figure represents the result of diffuse light.

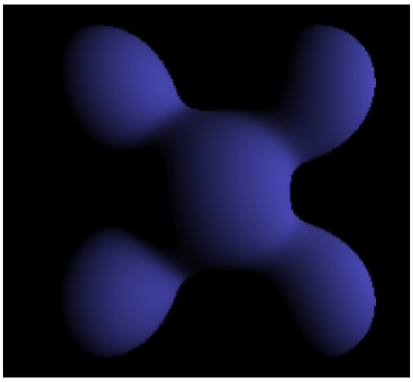


Figure 1.7 – Diffuse Reflection [6]

1.4.3 Specular Reflection

Simulates the bright spot of a light that appears on shiny objects. Specular highlights are more inclined to the color of the light than the color of the object[5]

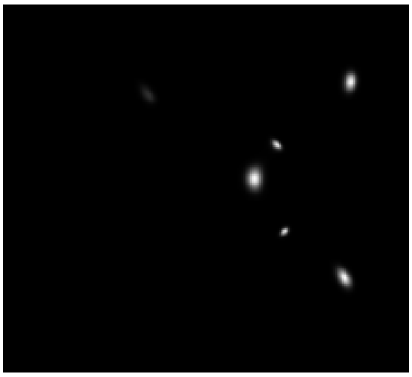


Figure 1.8 – Specular Reflection [6]

Final illumination of a point (vertex):

$$amb + diff + spec = (Ia.Ka) + (Ip.Kd.(N' * L')) + (Ip.Ks.(U' * V')^f).$$

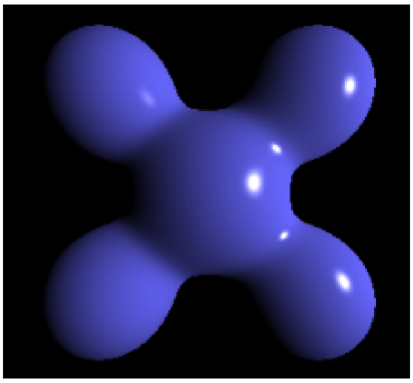


Figure 1.9 – Phong Model [6]

1.5 Indirect Illumination

When light rays bounce only once from the surface of an object to reach the eye, we speak of direct illumination. But when light rays are emitted by a light source, they can bounce off of the surface of objects multiple times before reaching the eye. This is what we call indirect illumination because light rays follow complex paths before entering the eye. Some surfaces are not exposed directly to any light sources (often the sun), and yet they are not completely black. This is because they still receive some light as an effect of light bouncing around from surface to surface. The next figure represents how indirect illumination works.[7]

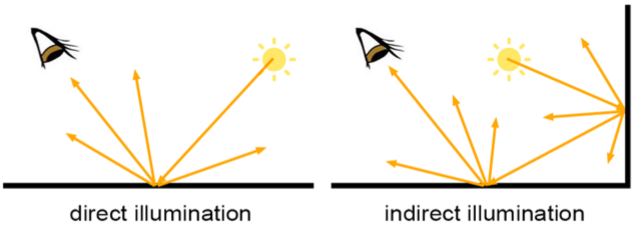
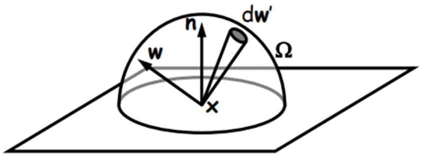


Figure 1.10 – The difference between Direct and Indirect Illumination. [7]

Indirect illumination is an important element for realistic image synthesis, but its computation is expensive and highly dependent on the complexity of the scene and of the BRDF of the surfaces involved.[8]

Global illumination involves to simulate both effects (direct plus indirect illumination). Simulating both effects is important to produce realistic images. It is represented by a Rendering Equation known as Kajiya [1986], an integral equation which generalizes a variety of known rendering algorithms [9]. Describes the flow of light energy throughout a scene, assuming that all objects of a scene (not just light sources) may reflect light. The next figure represents Kajiya equation.[10]



$$L_o(\mathbf{x},\mathbf{w}) = L_e(\mathbf{x},\mathbf{w}) + \int_\Omega f_r(\mathbf{x},\mathbf{w}',\mathbf{w})\, L_i(\mathbf{x},\mathbf{w}')(\mathbf{w}'\cdot\mathbf{n})\,d\mathbf{w}'$$

Figure 1.11 – Kajiya Equation. [10]

- x is a surface point. n is the normal. w is a unit vector (direction).
- Lo(x, w) is the light energy reflected outwards from point x in direction w.
- Le(x, w) is the light energy emitted at x in direction w.
- Omega denotes the hemisphere above the surface patch at x. The integral is

taken over all differential directional elements dw' on Omega.
- Li(x, w') is the incoming light energy incident on x arriving from direction w'.
- fr(x, w', w) is the fraction of light energy arriving at x from direction w' , that is reflected to direction w. (In general this depends on w and w'.)
- The (w' n) term captures the attenuation of arriving light, similar to Lambert's law. (The bigger the angle, the less the energy per unit area.[10])

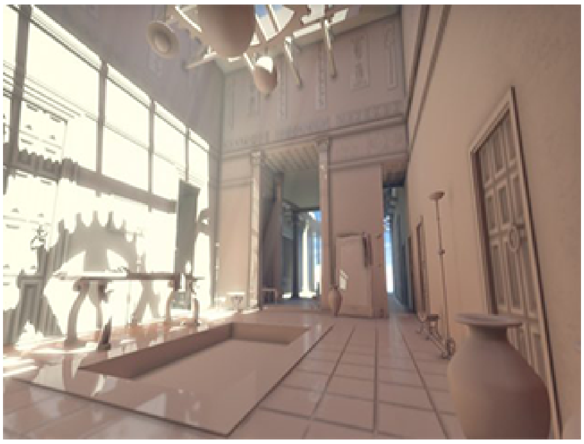


Figure 1.12 – Global Illumination [11]

Global illumination provides a visual richness not achievable with the direct illumination models used by most interactive applications. To generate global effects, numerous approximations attempt to reduce global illumination costs to levels feasible in interactive contexts. One such approximation, reflective shadow maps, environment maps,...etc.[12]

1.6 Conclusion

3D lighting is an important aspect of every 3D animation project. It is A combination of light sources to either draw attention to a special part of the scene or represent natural properties such as time of the day or even weather.
Direct Illumination is an interaction between the light source and the object only without any bouncing lights from other objects, it helps differentiate the objects of the scene, plus it is important for giving the scene a realistic view.

In the other hand indirect illumination takes in consideration all the bouncing lights that reflect the light in the scene not only the light source, which illuminates all the darker corners that can't be reached by the direct light, this gives the scene more realistic view since it approximates the behavior of the light in real life.

Using both techniques leads us to a global illumination that is considered as the ultimate lighting technique that makes the 3D scene so close to reality though its complexity in the calculations since the light can be reflected as an infinity of rays in all directions and each ray bounces in so many surfaces until it reaches the eye.

In the second chapter we will show some of the very optimized lighting techniques that can help us create a very good lighting in a small calculating time.

Chapter 2

Environment Maps

2.1 Introduction

In this Chapter we'll be talking about Environment Mapping and the different types of Environment Maps that are used in this discipline.[13]
There are so many lighting techniques that were developed through the years, and each one of them is seeking the goal of creating realistic lighted scenes, but the major problem is that most of them though the great quality of the results, takes so much time rendering due to the complexity of the calculations, some of them may take days if not months, also if some techniques can be done in real time, generally the quality of their results is so low and unsatisfying, that's what leaded us to environment maps.

2.2 Why Environment Maps?

Before talking about environment mapping, let us first think what we would mean by the environment? So far the examples have mostly been with a black background, as if the objects were all located in empty space. This is usually not the case and we want our scene to be located in either a closed environment (for example in a room) or in an open environment (outside, with a visible sky). As it turns out it is more efficient to create an illusion of an open environment, rather than having an actual simulation of the Earth, stars, the Sun or the Moon, mountains or whatever we want to see in the distance. Also Environment maps can be applied to a texture, to use it for faking reflections. reflections are usually created by ray-tracing, but using the Environment Maps reflections can be much faster than ray tracing reflections, In certain situations they need to be calculated only once, and may be re-used like any ordinary texture. So the main goal of this work is to reduce the calculation time and the GPU ressorces.

2.3 Environment Mapping Types

There are different types of Environment Mapping techniques that are used for creating reflections and refractions, which are calculated with the direct lighting technique:

Calculating the reflection vector R based on direction to eye I [14]:

$$R = 2(N.I)N - I \tag{2.1}$$

The following figure represents the reflection vector according to the eye and the normal of the surface.

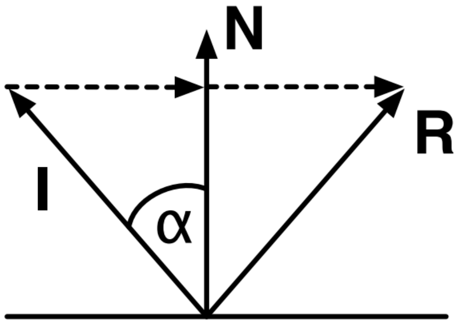


Figure 2.1 – Reflection [14]

We'll take a look at some techniques in the next sections:

2.3.1 Shadow Mapping

Shadow mapping is an old technique [Williams 78], but very widely used and developed with many extensions and implementations (Renderman). the technique precomputes images to avoid retracing shadow rays for every frame. The basic idea is to take an image of what the light sees in the scene and then compare it to what the eye sees by projecting the depth back to the light. The places where it is seen by both the light and the eye we know is lit. Otherwise, it is in the shadow. The saved image from the light is called the shadow map, which is essentially a bitmap where depth corresponds to a gray scale value.[15]

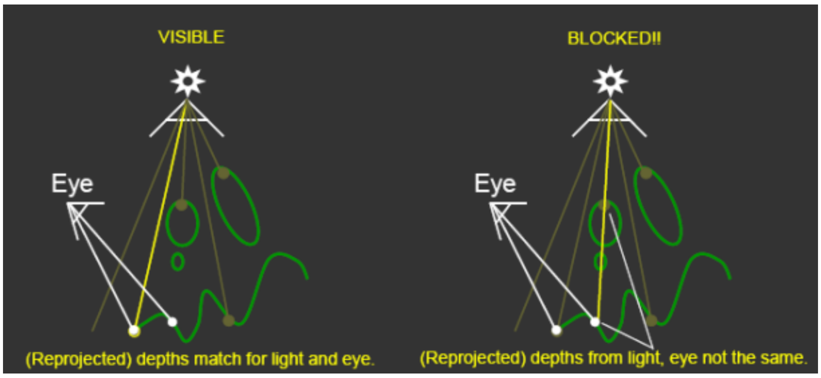

VISIBLE
Eye
(Reprojected) depths match for light and eye.
BLOCKED!!
Eye
(Reprojected) depths from light, eye not the same.

Figure 2.2 – Shadow Mapping [15]

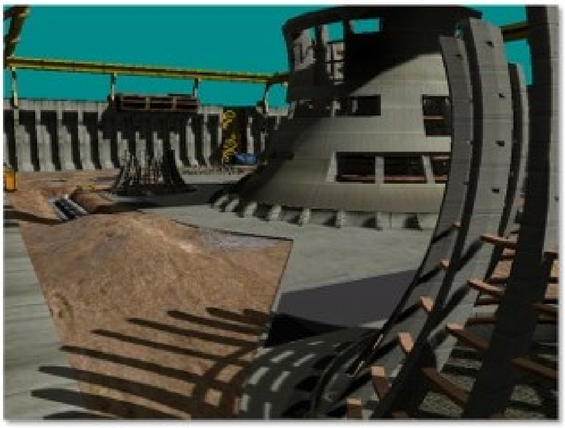


Figure 2.3 – Shadow Mapping example [**heitz2018combining**]

2.3.2 Sphere Mapping

Sphere mapping is the first environment map by Jim Blinn, in 1976. It is a type of reflection mapping that approximates the effect of reflective surfaces by storing the environment as a texture, and map it in an infinitely far spherical wall. This texture contains reflective data for the entire environment, except for the spot directly behind the sphere.

Issues with sphere mapping

— Cannot change the viewpoint (requires recomputing the sphere map).

— Highly non-uniform sampling.

— Uses high resolution polygons.

— Linear interpolation of texture coordinates picks up the wrong texture pixels which is represented in the next figure [14]:

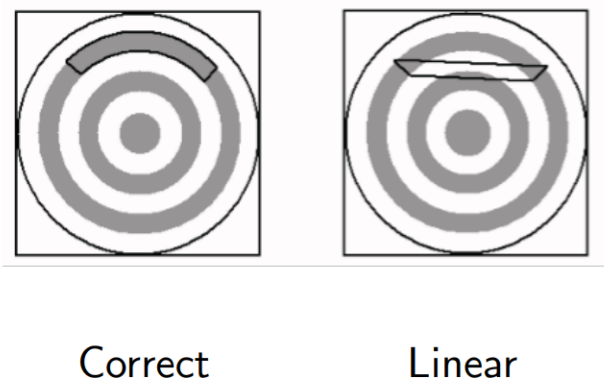


Figure 2.4 – correct and Linear sphere mapping [14]

Some results of Sphere Mapping

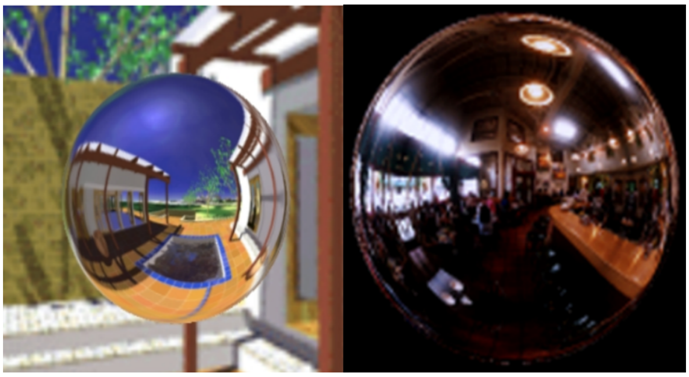


Figure 2.5 – Results of Sphere Mapping [14]

2.3.3 Cube Mapping

A cube map is a type of environment maps that is essentially a box made out of six 2D textures, which can be provided to the program or generated at runtime, mostly used for creating reflections and refractions [16]. It has become a common environment mapping technique for reflective objects. [17]

Generally these six cube map faces images are an artist created static scene of mountains or other such far away features.[18] The next figure represents a cube map texture.

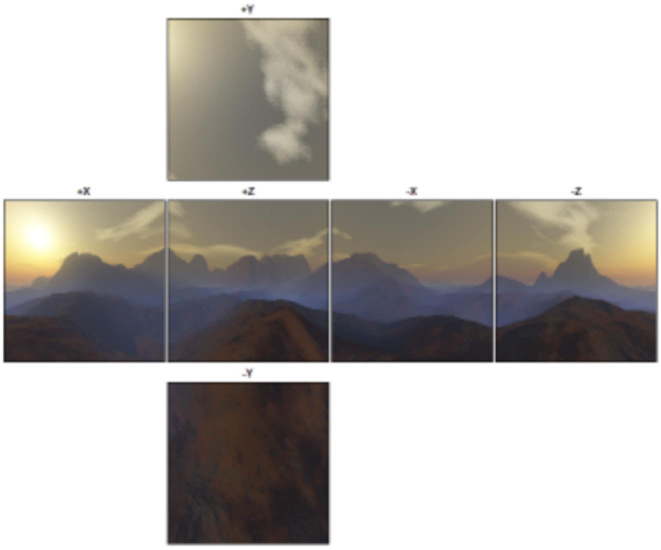


Figure 2.6 – Cube Map faces [18]

2.3.3.1 Main idea

Sampling the cube map using a normalised direction vector (a ray that begins in the middle of the cube formed by the cube mapped textures, with the point where the ray intersects the cube being the texel sampled).

The figure below shows how the 45 degree cube map corner can be sampled (using a 45 degree reflection vector and a bi-linear interpolation between the two textures).

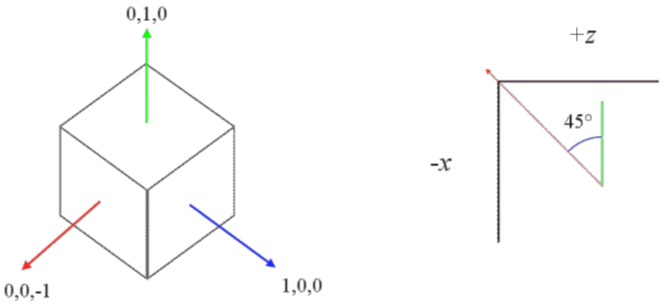


Figure 2.7 – Sampling direction vectors [18]

To calculate the reflection vector for a surface, we need its normal, and the incident vector that runs from the view point to the surface, to use them for calculating the angle of reflection from the angle of incidence.
Another popular use for cube maps is that of skyboxes. Imagine that there is a mountain in the distance inside a game. A mountain that the player can never travel to, because it is so far away. Modeling the actual geometry of that mountain and then rendering it, would take resources of the GPU. It is much more efficient and quite effective to just have a pre-rendered image of that mountain to display as a background. Then we can use that background as a texture in a cube. This is usually called a skybox or a skydome.[19]
The next figure represets how the skybox work.

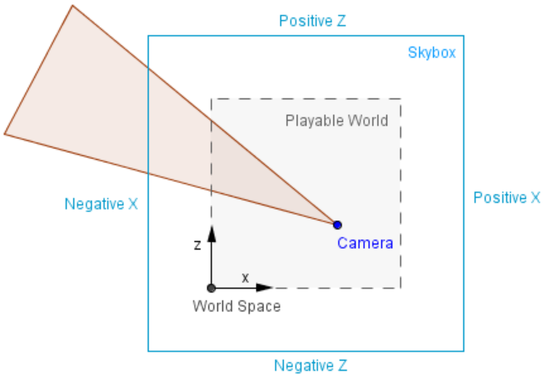


Figure 2.8 – Skybox [19]

2.3.3.2 Advantages of Cube Mapping

- Cube mapping is preferred over other methods of environment mapping because of its simplicity.
- Cube mapping produces results that are similar to those obtained by ray tracing, but much more computationally efficient.
- A solution for the problem of viewpoint dependency in Sphere mapping.
- A texture mapped onto a sphere’s surface must be stretched and compressed, and warping, and distortion, unlike cube mapped textures.
- Can be rendered only once during the process.

2.3.3.3 Limits of Cube Mapping

- Doesn’t work well with moving objects because it requires regenerating the cube map.

Dynamic Cube Mapping

The major problem of static cube map is that it doesn’t work with dynamic scenes because it requires re-generating the cube map, this is where dynamic Cube map have been developed as a solution for static cube map problem. Dynamic cube Mapping is a form mapping where the cube map could be rendered every single frame (In real-time) by rendering our scene six times before the final render pass, once for each face on the cube. So when textures are provided in the program at run-time we call it a dynamic cubemap.

There are some differences between Dynamic Cube and Cube Mapping:

1. Dynamic Cube Map is an advanced method of Cube Map.
2. The Lighting of Dynamic Cube Mapping is more realistic than Cube Mapping.
3. The objects of Cube Map are static but in the other hand dynamic cube mapping objects are dynamic.
4. Creating a dynamic cube map is more expensive than cube map because of the amount of textures that must be rendered.

2.3.3.4 Related work

In this section we will present some works which use the cube mapping technique.

Cube Maps: Sky Boxes and Environment Mapping by Anton Gerdelan [2 October 2016][20]:

- Sampling the cube map texture coordinates using a direction vector with R, S, and T, components.

- Creating a big box encase the camera as it moves around. It represents the Skybox as shown in the figure.

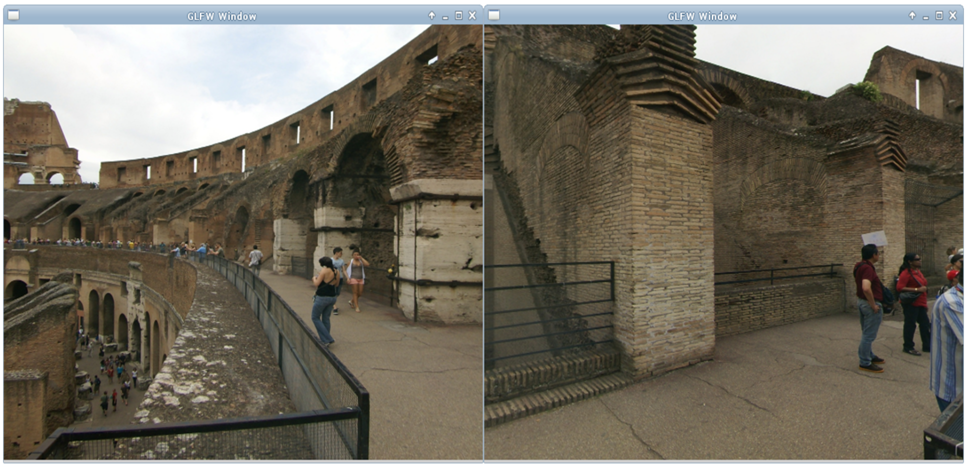


Figure 2.9 – Skybox Implementation [20]

- Reflecting the Skybox onto the surface using a direction vector from the view point (camera position) to the surface, then reflect it based on the surface normal, after that we use the reflected one to sample the cube map as shown in the figures.

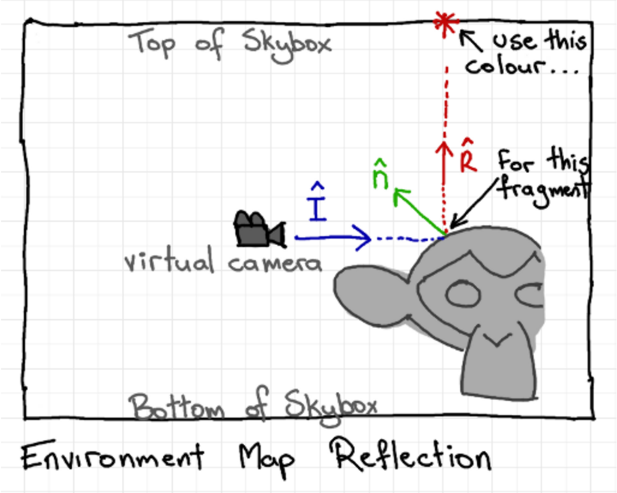


Figure 2.10 – Environment Map Reflection [20]

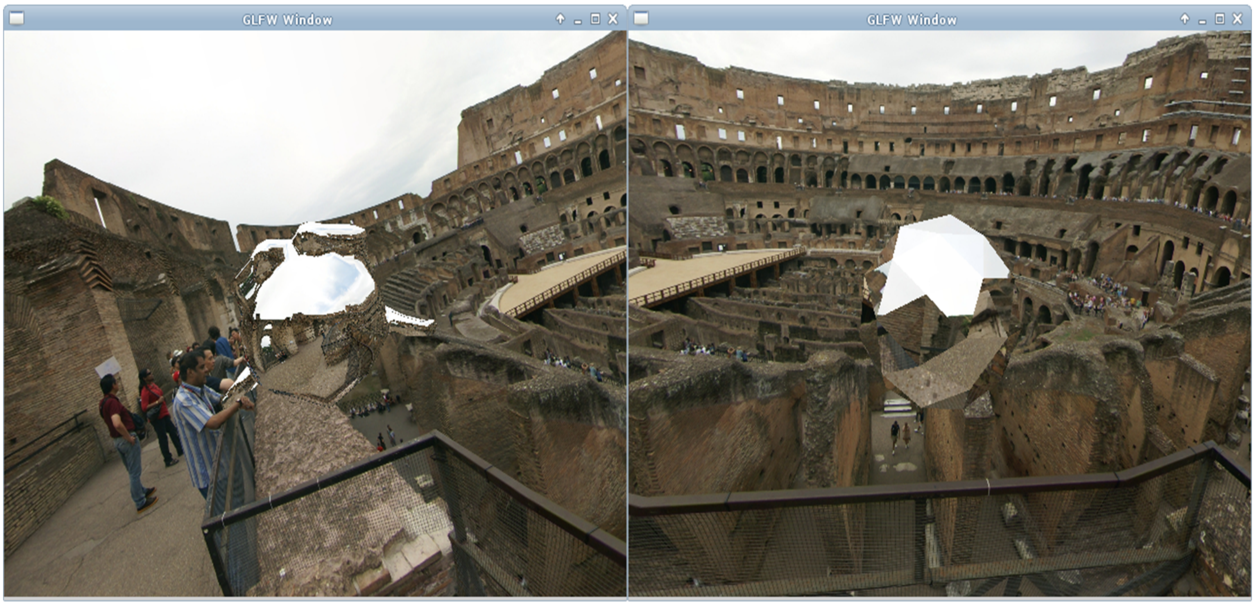


Figure 2.11 – Environment Map Reflection Implementation [20]

Advantages:

— Can be done in real-time.

Limits:

— Somme of the common mistakes is that the reflection is upside-down or the wrong side, so we need to check the direction of our incident and normal vectors, and normalise them.

— The mesh reflections are not smooth, so we need to use interpolated surface normals.

— Doesn't work with dynamic scenes.

OpenGL Cube Map Texturing by G. Zachmann [1999][21]:
The texture is generated by capturing a 360 degree view of our surroundings by standing in one place and taking six images, each at an orthogonal 90 degrees view from the others, as represented in The next image.

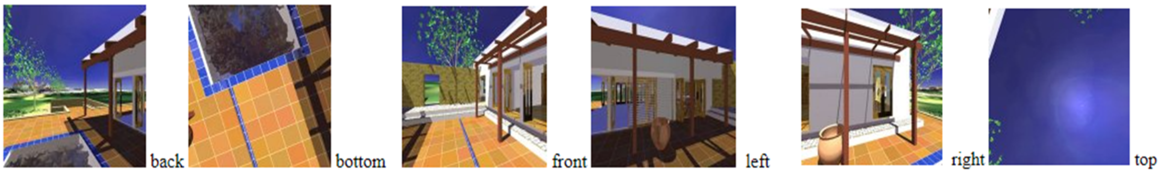


Figure 2.12 – Cube map texture [21]

- Mapping texture coordinates to Cube Map faces by indexing into the texture using a 3D direction vector (rx,ry,rz).
- Then we calculate the reflection based on the camera direction vector and the normal of the surface as scene in the figure below.

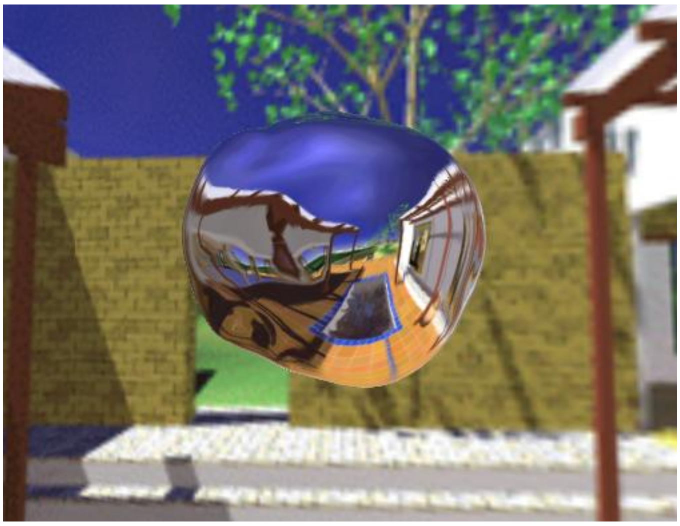


Figure 2.13 – Object Reflection [21]

Advantages:

— The reflection is in the right side.

— Can be done in real-time.

— The mesh reflections are smooth.

Limits:

- Doesn't work with dynamic scenes.

Converting to/from cubemaps by Paul Bourke [July 2020][22]:

Transforming a cube map (90 degree onto the face of a cube) into a cylindrical panoramic image as scene in the figure below.

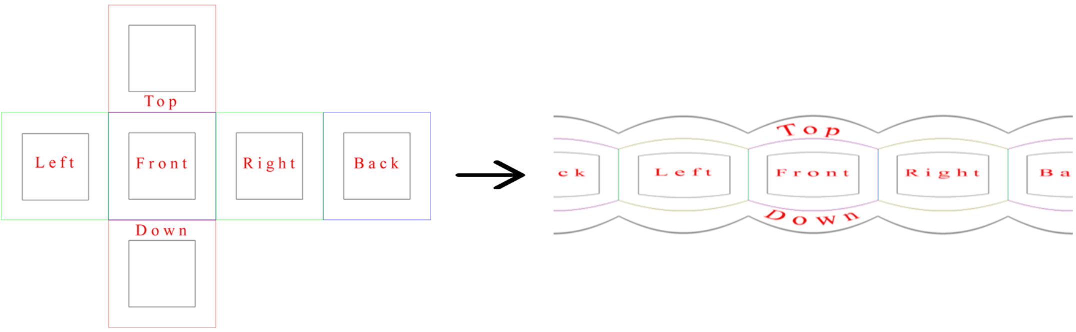


Figure 2.14 – From cubic to panoramic [22]

In particular, if (i,j) is the pixel index of the panoramic normalised to (-1,+1) the the direction vector is given as follows.[22]

x = cos(i . pi)
y = j tan(v/2)
z = sin(i . pi)

The desired image should be as the following figures.

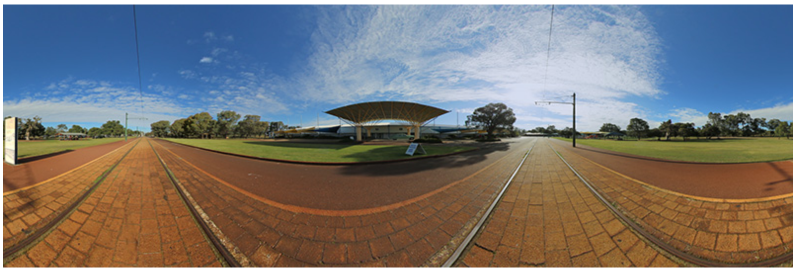


Figure 2.15 – Cylindrical panoramic (120 degrees) [22]

The following cube map texture can be transformed to a cylindrical panoramic image.

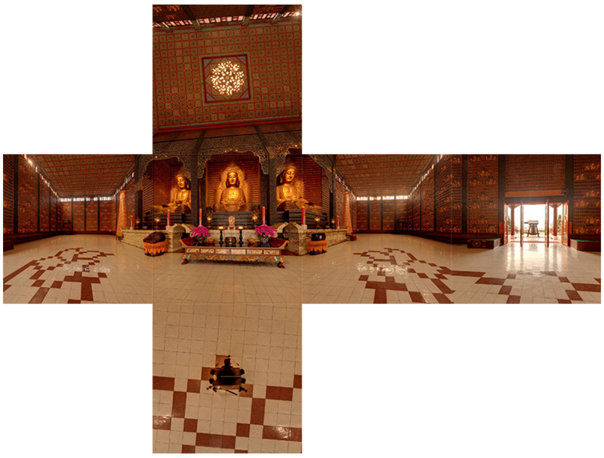


Figure 2.16 – Cube to perspective projection 1 [22]

Which gives us the following results with a better perspective vision.

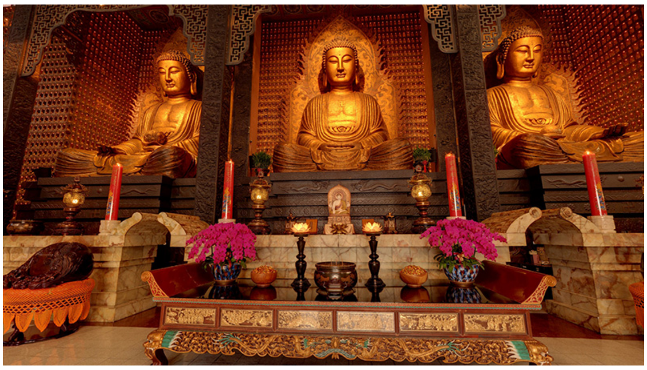


Figure 2.17 – Cube to perspective projection 2 [22]

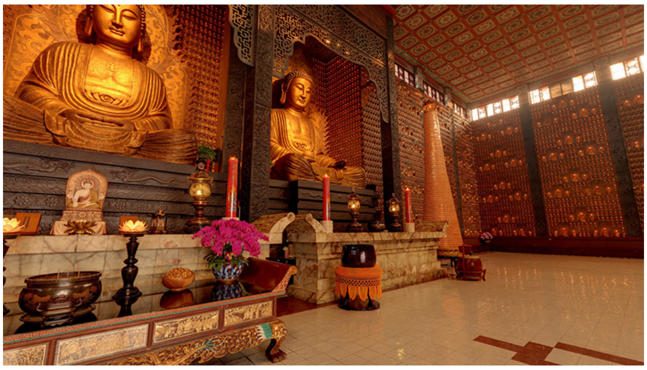


Figure 2.18 – Cube to perspective projection 3 [22]

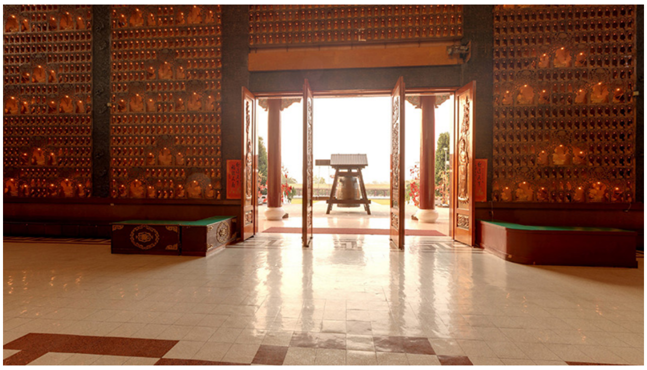


Figure 2.19 – Cube to perspective projection 4 [22]

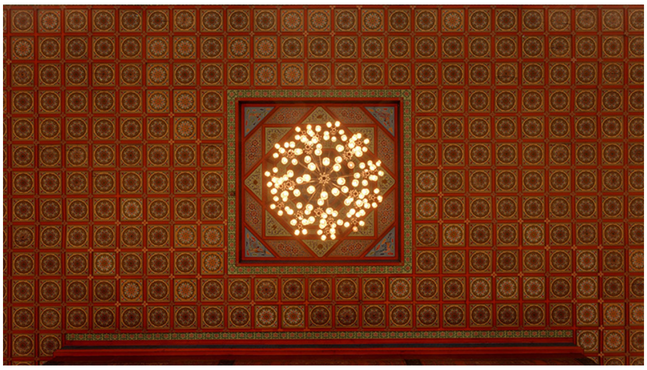


Figure 2.20 – Cube to perspective projection 5 [22]

Figure 2.21 – Cube to perspective projection 6 [22]

Advantages:

- optimize the perspective vision which makes the scene more realistic.

Limits:

- Doesn't work with dynamic scenes.

Dynamic-Cubemaps by Kevin Hongtongsak [2016][16]:

- Creating a cube map and assigning the images to each side of it, and then sample them using the cube map sampler (the direction vector) as shown in the figure below.

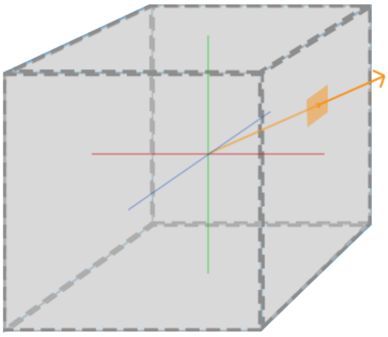


Figure 2.22 – Cube Maps example [16]

- This is the cube map texture that have been used in the program.

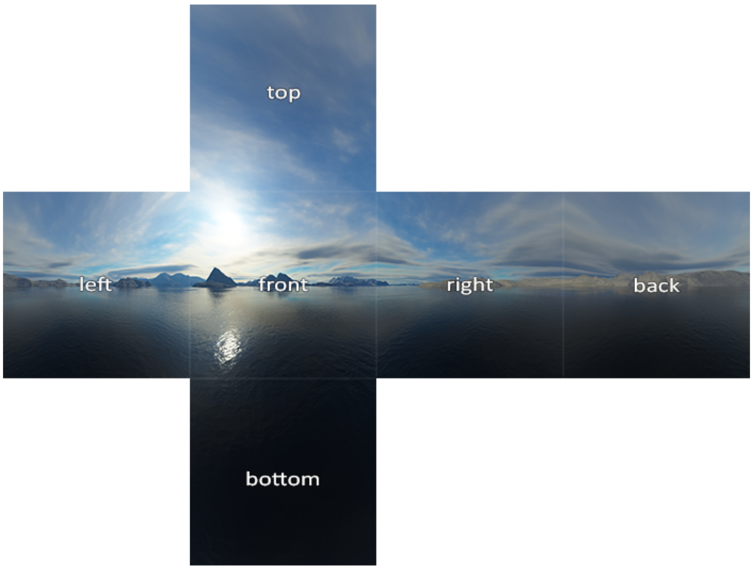


Figure 2.23 – Cube Maps texture [16]

- Creating a huge cube and applying the texture to it from the inside. The skybox can be seen in the background and reflected off this sphere in the figure.

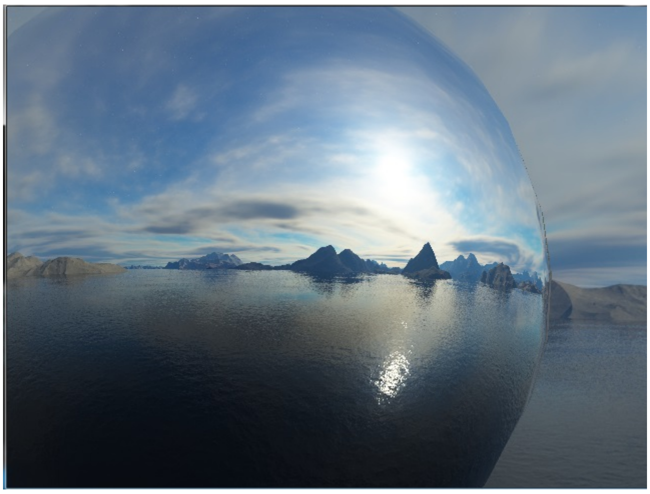


Figure 2.24 – Final Skybox [16]

Advantages:

- Can be done in real-time.

Limits:

- Doesn't work with dynamic scenes.

Cubemaps by Joey de Vries [2019][23]:

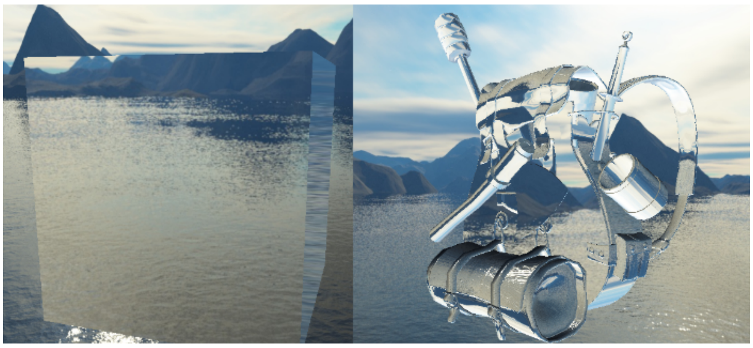


Figure 2.25 – Cube Maps Refraction Result [23]

Advantages:

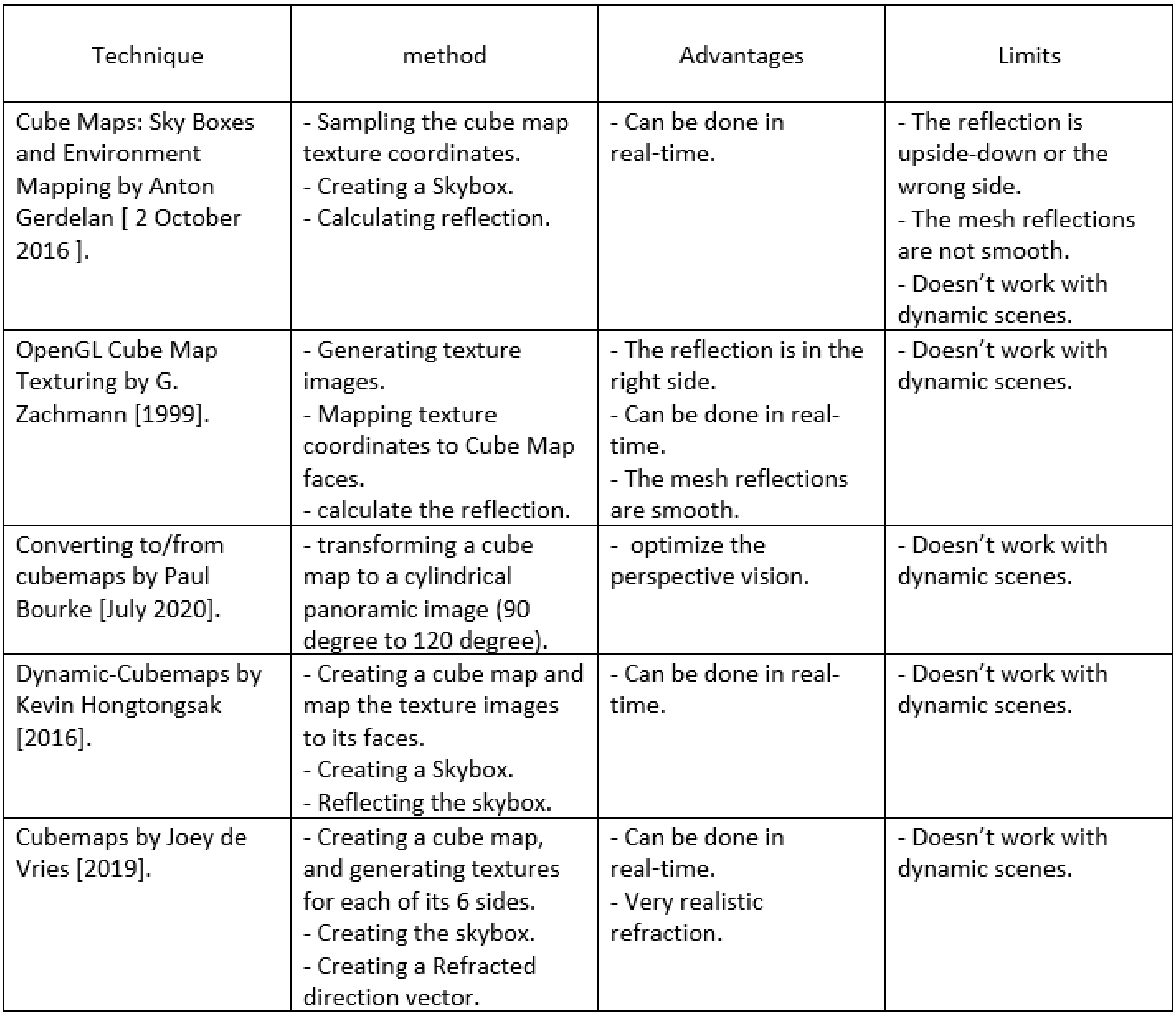
— Can be done in real-time.

— Very realistic refraction.

Limits:

- Doesn’t work with dynamic scenes.

Comparison between the techniques:



Technique	method	Advantages	Limits
Cube Maps: Sky Boxes and Environment Mapping by Anton Gerdelan [2 October 2016].	- Sampling the cube map texture coordinates. - Creating a Skybox. - Calculating reflection.	- Can be done in real-time.	- The reflection is upside-down or the wrong side. - The mesh reflections are not smooth. - Doesn't work with dynamic scenes.
OpenGL Cube Map Texturing by G. Zachmann [1999].	- Generating texture images. - Mapping texture coordinates to Cube Map faces. - calculate the reflection.	- The reflection is in the right side. - Can be done in real-time. - The mesh reflections are smooth.	- Doesn't work with dynamic scenes.
Converting to/from cubemaps by Paul Bourke [July 2020].	- transforming a cube map to a cylindrical panoramic image (90 degree to 120 degree).	- optimize the perspective vision.	- Doesn't work with dynamic scenes.
Dynamic-Cubemaps by Kevin Hongtongsak [2016].	- Creating a cube map and map the texture images to its faces. - Creating a Skybox. - Reflecting the skybox.	- Can be done in real-time.	- Doesn't work with dynamic scenes.
Cubemaps by Joey de Vries [2019].	- Creating a cube map, and generating textures for each of its 6 sides. - Creating the skybox. - Creating a Refracted direction vector.	- Can be done in real-time. - Very realistic refraction.	- Doesn't work with dynamic scenes.

2.4 Conclusion:

Environment mapping is generally the fastest method of rendering a reflective or a refractive surface.

It is pretty slow to calculate reflections with usual lighting methods, like recursive ray-tracing. If we use environment mapping , it can obtain global reflection and lighting results in real-time with a good quality.

Cube mapping is preferred over other methods of implementing environment mapping because of its quality and simplicity.

There are many uses for cube mapping such as Skyboxes (perhaps the most advanced application of cube mapping of creating pre-rendered panoramic sky images), Dynamic reflection (a cube map texture can be consistently updated to represent a dynamically changing environment), Global illumination, Projection textures (It relies on cube maps to project images of an environment onto the surrounding scene)...etc.

Chapter 3

Conception of a cube map method using GPU

3.1 Introduction

We have seen in the previous chapter the method and different implementations of the cube mapping technique.
In this chapter we will present the architecture of our system as well as the description of each part, and the role of each component.

note: we are using diffuse cube map for diffuse scenes in our system.

3.2 Goals

Our main goals in this project is to accelerate the calculating time of the lighting of our scene, but without any loss in the quality.

3.3 Main idea and motivations

The problem of getting a good lighting quality in a short rendering time leaded us to a very optimized technique called cube mapping that can approximate the effect of most of lighting techniques with a good quality and a minimum calculating time and also can be rendered at real-time.

3.4 General Architecture

Our system takes the scene components as input and produces a rendered image of the scene as output, passing by the most important step that produces the final image, which is the cube mapping.

Our architecture can be divided into three important parts. In the first part (input), we load our obj file using the Assert Import Library (Assimp) and we define the different components of the scene that we will use (the camera coordinates, and the light source), then, the goal of the second part (Cube Mapping) is to allow us computing the direct illumination of the scene. Basically, it represents the lighting technique that we're going to use to create the shading and the shadow of the scene. Finally, we visualize the final image of the scene with the shading and the shadow calculated in the previews step (display) (see figure 3.1).

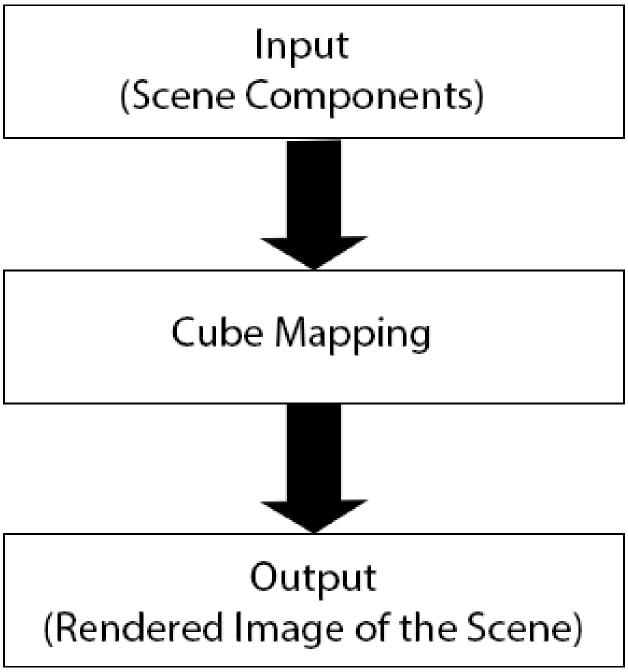


Figure 3.1 – Global Architecture of the system

3.5 Detailed Architecture

We have seen in the previous section 3.4, the global architecture of our system, we are now going to detail it into subsystems, to fully understand the functioning of each component and its role in our system.

3.5.1 Input

In this part, we enter the components of the scene and develop the scene tree which contains the objects, the point light source, and the camera, this step is so important because it focuses on the geometry of the scene and the position, direction of the camera and the point light source, if one of the components isn't in the right place we won't get the result we need. (see figure 4.1)
We use Assimp to load the scene, which is represented as an obj and mtl file. The obj file contains the geometry of the 3D scene, when the mtl file contains the material components (texture, kd, ka,...).
The camera and the point light source parameters has described in the .ini file.

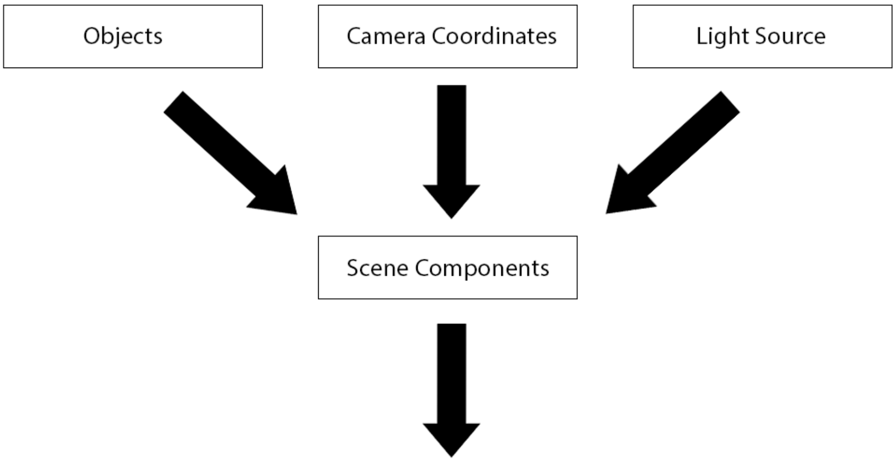


Figure 3.2 – Input Part

As we can see in the Figure 4.1, the scene components are divided to three important components, each one of them has it's own way to set it up, we will specify each and every component of them as follows:

3.5.1.1 Objects

In this step we must choose and select the 3D objects that we need to place them in our scene, then we load them to our program using the objects loader. When we finish, we place each object where it's supposed to be in our scene (controlling the objects coordinates)(see figure3.3).

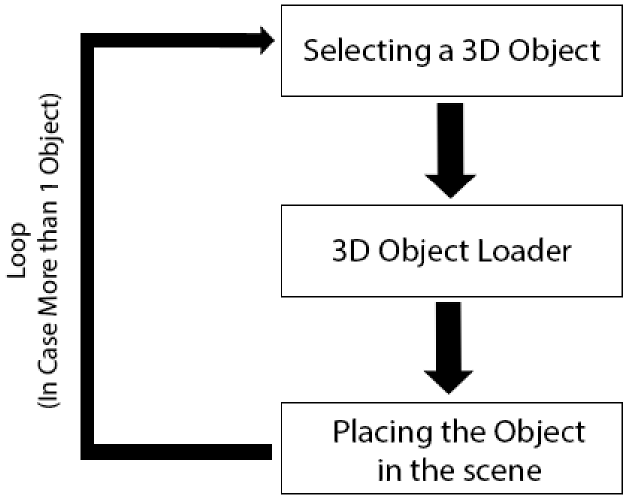


Figure 3.3 – Objects of the scene

3.5.1.2 Camera coordinates

In Computer Graphics, a camera is considered as any other 3D object, it has position coordinates in the world space, but unlike the 3d objects, the camera has other parameter called the direction coordinates, also, we need to specify the angle of the camera view (named the Field Of View (FOV)). So We need to set the camera parameters (of both the position, direction and FOV)

3.5.1.3 Light Source

Since we have several types of light sources (point light, area light,...), in our system we have placed a point light source in our scene, which has position and direction coordinates (see 4.2)

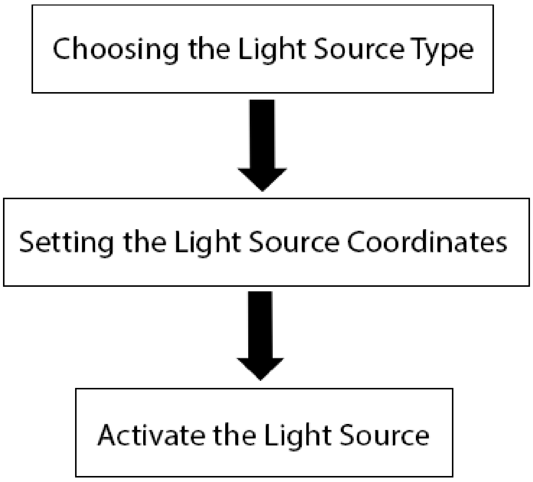

Choosing the Light Source Type
Setting the Light Source Coordinates
Activate the Light Source

Figure 3.4 – Objects of the scene

3.5.2 Cube Mapping

This is the most important part in our system. After the scene components are all set, and the structure of the scene has been prepared inside the camera frame, we can now apply our lighting technique which is the cube mapping, for each of the scene vertices. As mentioned in the previous chapter, this technique can reduce the calculating time so much, and even make the rendering done at real-time.

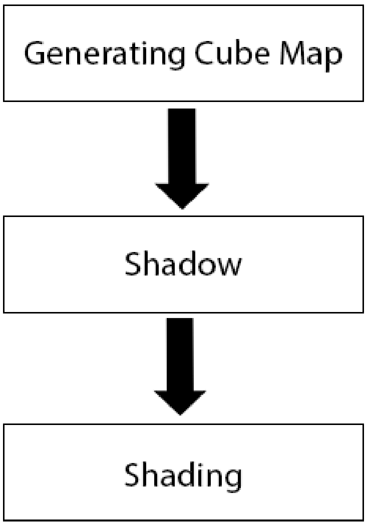


Figure 3.5 – Objects of the scene

As we can see in the figure 4.3 , this part can be divided into three important steps:

3.5.2.1 Generating the cube map

The first step is to generate the cube map that we are going to use in our system, we do that by creating a cube map, then sampling it with the cube map sampler. Here we managed to develop an optimized method, that is an advanced version of the classic one:

- **The classic method:** requires 6 rendering passes to generate the cube map, and one additional pass to render the scene and display the final image. This can add so much to the calculating time.
- **The advanced method:** this one can reduce the number of rendering passes to 2 passes only to get the final result, which means less calculating time. The first pass allows us to generate the cube map and the second pass aims to generate the final image.

3.5.2.2 Computing the shadows

In this step, we calculate the shadow for each of the scene components based on the light source position using the cube map sampler, basically creating a shadow map using the cube map, where we calculate the closest and the farest depths to determinate the positions of the shadow according to the light source position in the scene. The program takes as an input the vertices positions of the scene, the camera coordinates, and the lightsource position, and produces as output the shadow. (see figure 4.4)

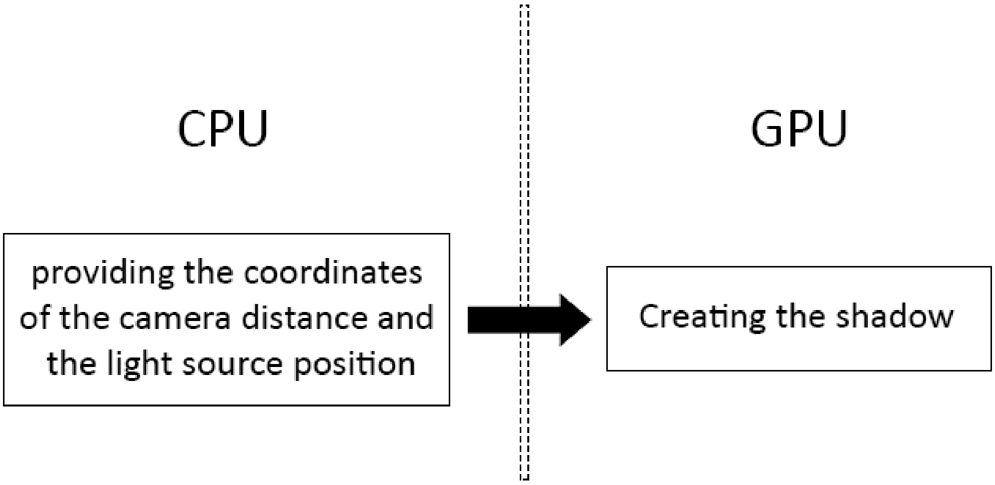

CPU
GPU
providing the coordinates of the camera distance and the light source position
Creating the shadow

Figure 3.6 – CPU and GPU of the shadow

3.5.2.3 Shading

The last step, is where we calculate the shading (direct illumination) of the scene using the sampled cube map. In this step we calculate the diffuse color of our scene, and that is done using the cube map sampler. The program takes as an input the normal of the surface, the previous shadow value (to detect the visibility), and the light source position, then it produces the shading or the diffuse color of our scene as an output. (see figure 4.5)

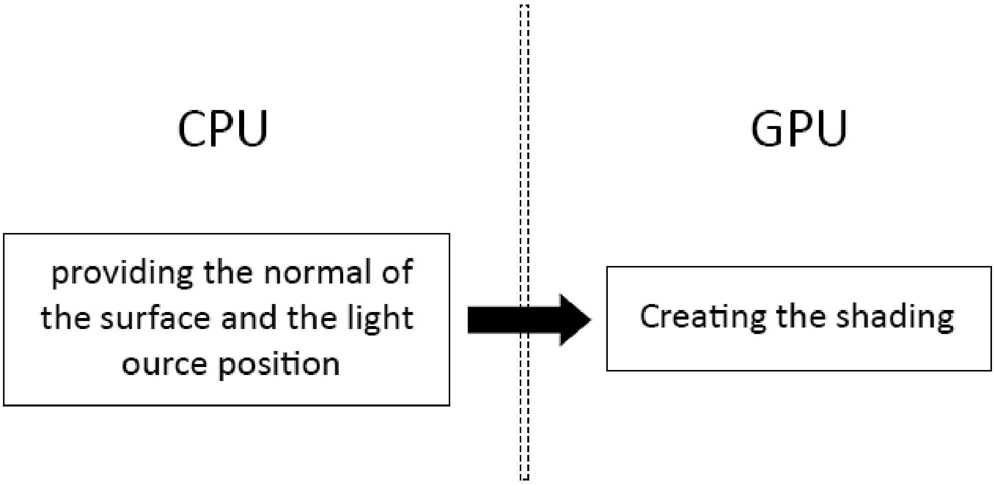

CPU
GPU
providing the normal of the surface and the light ource position
Creating the shading

Figure 3.7 – CPU and GPU of the shading

3.5.2.4 Display part

The last part of our system is the display part, after calculating our scene lighting using cube mapping method, we now can save and display the result, which is the final rendered image of our scene (see figure 4.4).

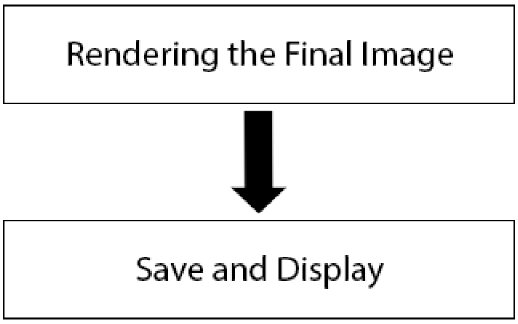


Figure 3.8 – Display Part

3.6 Conclusion

In this chapter, we have given a specification on our system conception, this phase represents one of the most important phases in the software development process.

It describes the system from a general and detailed point of view to understand and succeed in the programming phase.

In the next chapter, we will illustrate the realization of our system by representing some interfaces and some results obtained, with the structures which are chosen to implement this system.

Chapter 4

Implementation, results and discussion

4.1 Introduction

In this chapter we will describe the implementation of the different stages of our application.
Firstly, we will present the hardware configuration and the environment and library of the machine we used in our project, next we will detail the data structure as well as the algorithms used in our implementation, and finally we will present and discuss our cube maps results.

4.2 Hardware configurations

Our hardware of DESKTOP-KC0RTVR laptop includes the following devices:

— **OS:** Windows 10 Professional 2018 ,64bits.

— **CPU:** Intel(R) Core(TM) i5430OU CPU @ 1.90GHZ 2.50 GHz.

— **GPU:** Intel(R) HD Graphics 4400.

— **RAM capacity:** 8.00 Go.

— **Storage:** 237 GB.

— **Screen:** 35.6 cm.

4.3 Environnement and libraries

To implement the cube map technique, we use the following libraries:

— **Environnement:** Visual Studio 2008.

— **Programming language:** C++.

— **Libraries:** GLEW, ASSIMP, SFML, GLM.

4.3.1 Programming Environment (Visual Studio)

Microsoft Visual Studio is a development software suite for Windows designed by Microsoft.
Visual Studio is a comprehensive set of development tools for building ASP.NET Web applications, XML Web Services, desktop applications, and mobile applications. Visual Basic, Visual C ++, Visual C, and Visual J all use the same Integrated Development Environment (IDE), which allows them to share tools and makes it easy to build solutions that use multiple languages.

In addition, these languages make it possible to take better advantage of the functionalities of the .NET Framework, which provides access to key technologies simplifying the development of ASP Web applications and XML Web Dervices thanks to visual Web Develper.

4.3.2 Programming Language: C++

In our application, we used C++ as the programming language. C++ is a compiled programming language, allowing programming under multiple paradigms such as procedural programming, object-oriented programming, and generic programming. The C ++ language does not belong to anyone and therefore anyone can use it without the need for authorization or the obligation to pay for the right to use it. C ++ is one of the most popular programming languages, with a wide variety of hardware platforms and operating systems.

4.3.3 Libraries

4.3.3.1 GLEW

The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension loading library.
GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. OpenGL core and extension functionality is exposed in a single header file.
We are using GLEW 1.9.0 support for OpenGL 4.3.

4.3.3.2 SFML

Simple and Fast Multimedia Library (SFML) is a cross-platform software development library designed to provide a simple application programming interface (API) to various multimedia components in computers(system, window, graphics, audio and network), and we are using SFML 2.5.1.

4.3.3.3 Assimp

Open Asset Import Library (Assimp) is a cross-platform 3D model import library which aims to provide a common application programming interface (API) for different 3D asset file formats.
Written in C++, it offers interfaces for both C and C++, and we are using Assimp 3.0.0.

4.3.3.4 GLM

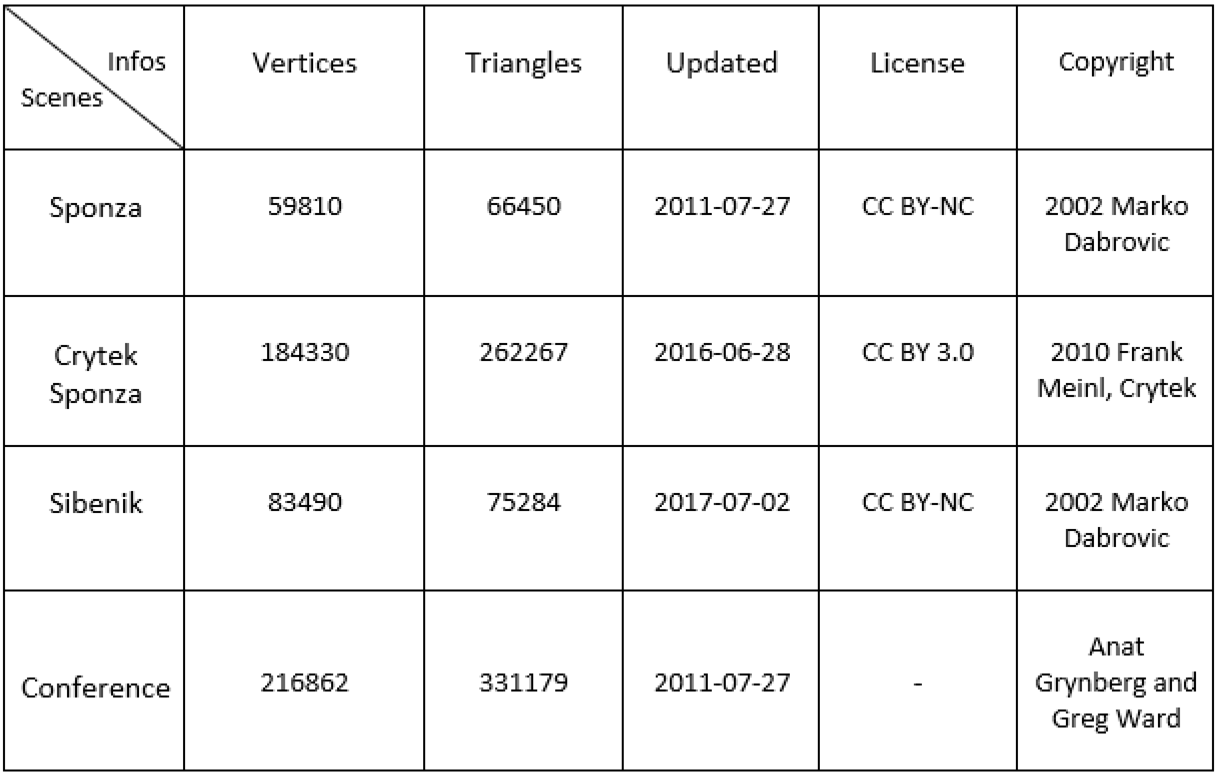
OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications.
GLM provides classes and functions designed and implemented with the same naming conventions and functionalities than GLSL so that anyone who knows GLSL, can use GLM as well in C++.
GLM isn't limited to GLSL features. An extension system, based on the GLSL

extension conventions, provides extended capabilities: matrix transformations, data packing, random numbers, noise, etc...
GLM is a good library for software rendering (ray tracing / rasterisation), image processing, physic simulations and any development context that requires a simple and convenient mathematics library.
GLM is written in C++98 but can take advantage of C++11 when supported by the compiler.
We are using GLM 1.9.0.1.

4.4 Implementation details

In this section we'll present a set of our obtained results. We have chosen the following scenes: Sponza, Crytek Sponza, Sibenik, and Conference.

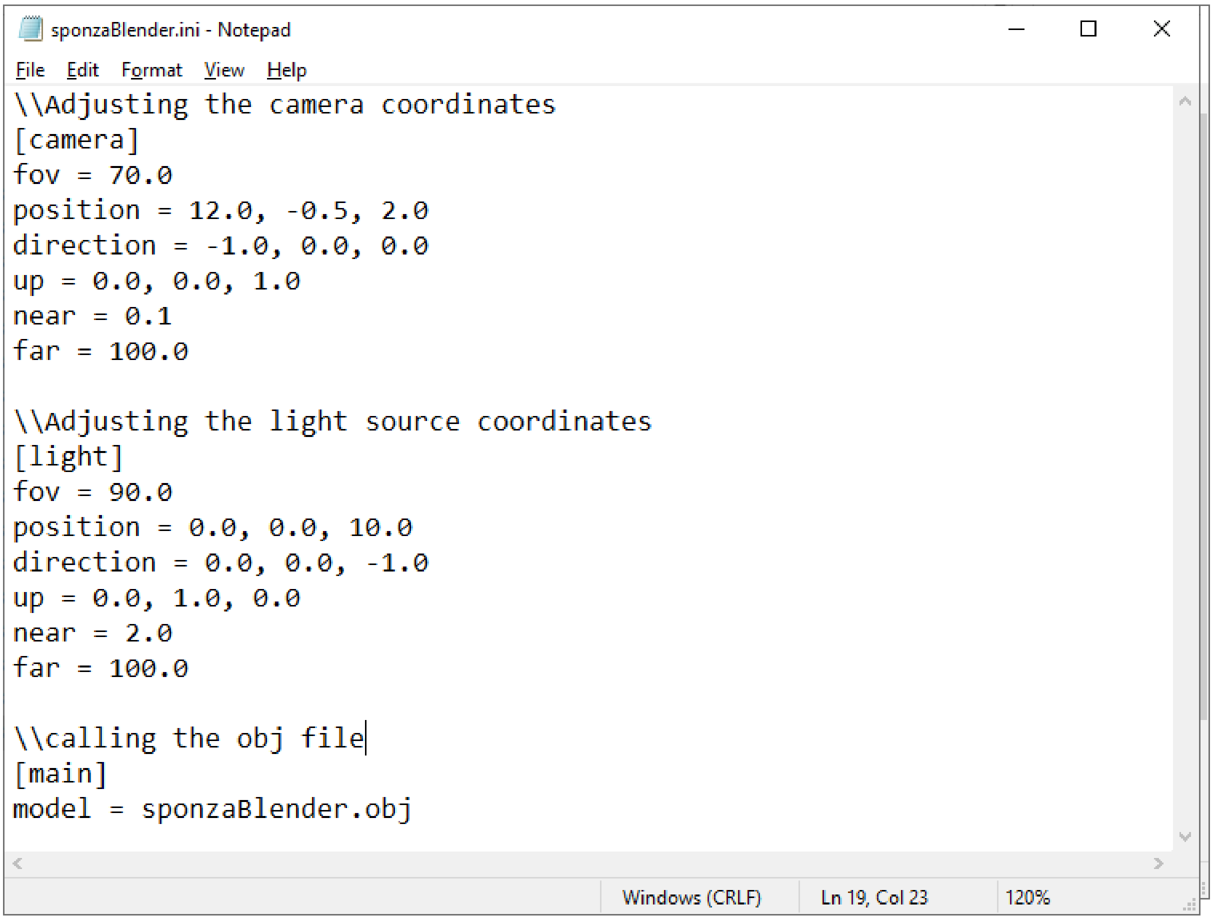
4.4.1 Scenes informations



Infos Scenes	Vertices	Triangles	Updated	License	Copyright
Sponza	59810	66450	2011-07-27	CC BY-NC	2002 Marko Dabrovic
Crytek Sponza	184330	262267	2016-06-28	CC BY 3.0	2010 Frank Meinl, Crytek
Sibenik	83490	75284	2017-07-02	CC BY-NC	2002 Marko Dabrovic
Conference	216862	331179	2011-07-27	-	Anat Grynberg and Greg Ward

4.4.2 Input

As we have seen in the previous chapter, this part is where we enter the components of the scene, by loading the obj file of our 3D scene model, and adjusting the coordinates of the light source and the camera. All these steps can be done in the ini file. (see figure 4.1)



```
\\Adjusting the camera coordinates
[camera]
fov = 70.0
position = 12.0, -0.5, 2.0
direction = -1.0, 0.0, 0.0
up = 0.0, 0.0, 1.0
near = 0.1
far = 100.0

\\Adjusting the light source coordinates
[light]
fov = 90.0
position = 0.0, 0.0, 10.0
direction = 0.0, 0.0, -1.0
up = 0.0, 1.0, 0.0
near = 2.0
far = 100.0

\\calling the obj file
[main]
model = sponzaBlender.obj
```

Figure 4.1 – The ini file

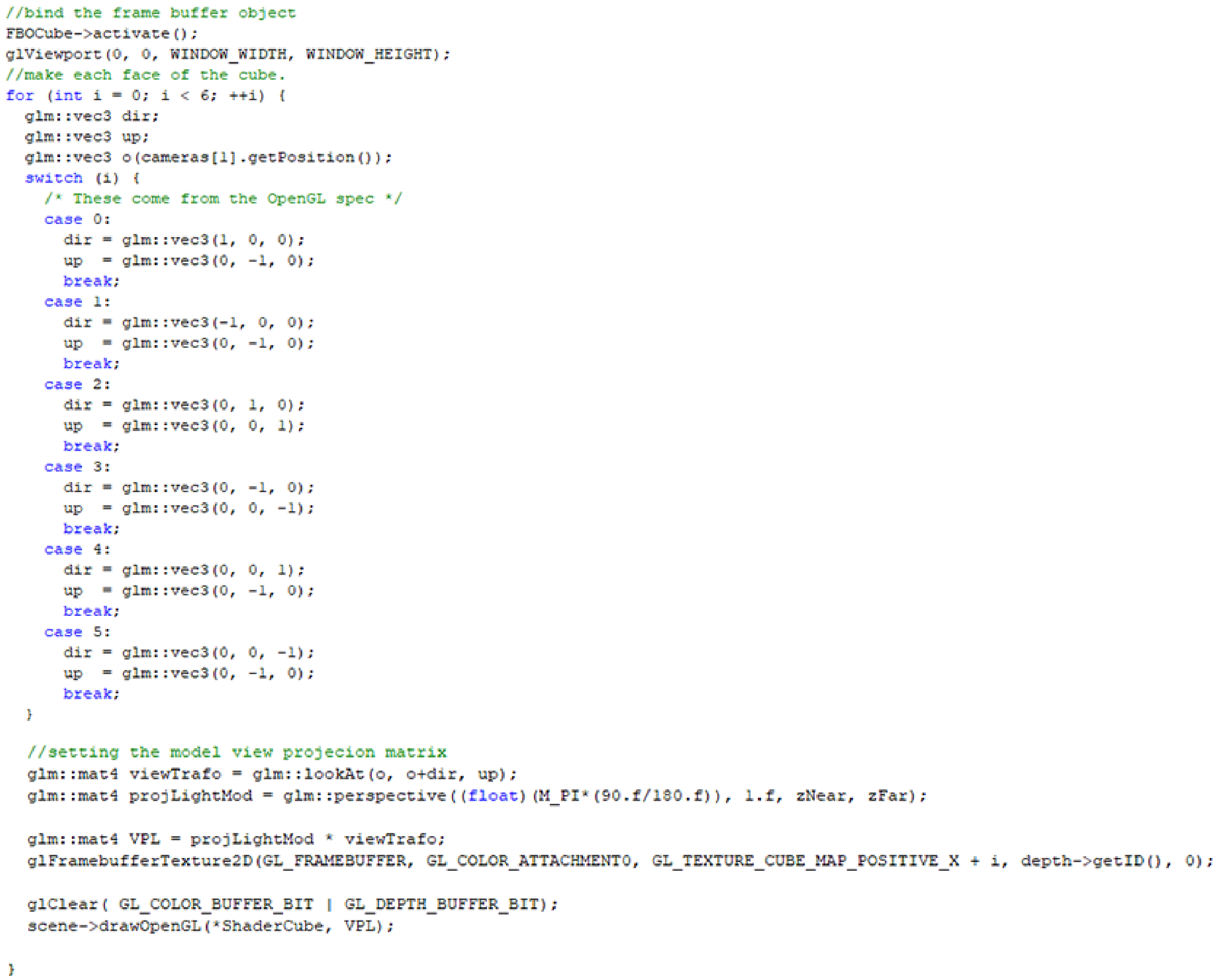
And now we finished setting our scene components to be used in the next parts.

4.4.3 Generating cube map

In the conception chapter we clarified that in this part we used two different methods, classic and advanced to generate the cube map, we are going now to explain each one of them in details.

4.4.3.1 Classic

This method doesn't use the geometry shader, which means the rendering function is called six times for making each face of cube map, which adds a lot to the calculating time. (See figure 4.2)



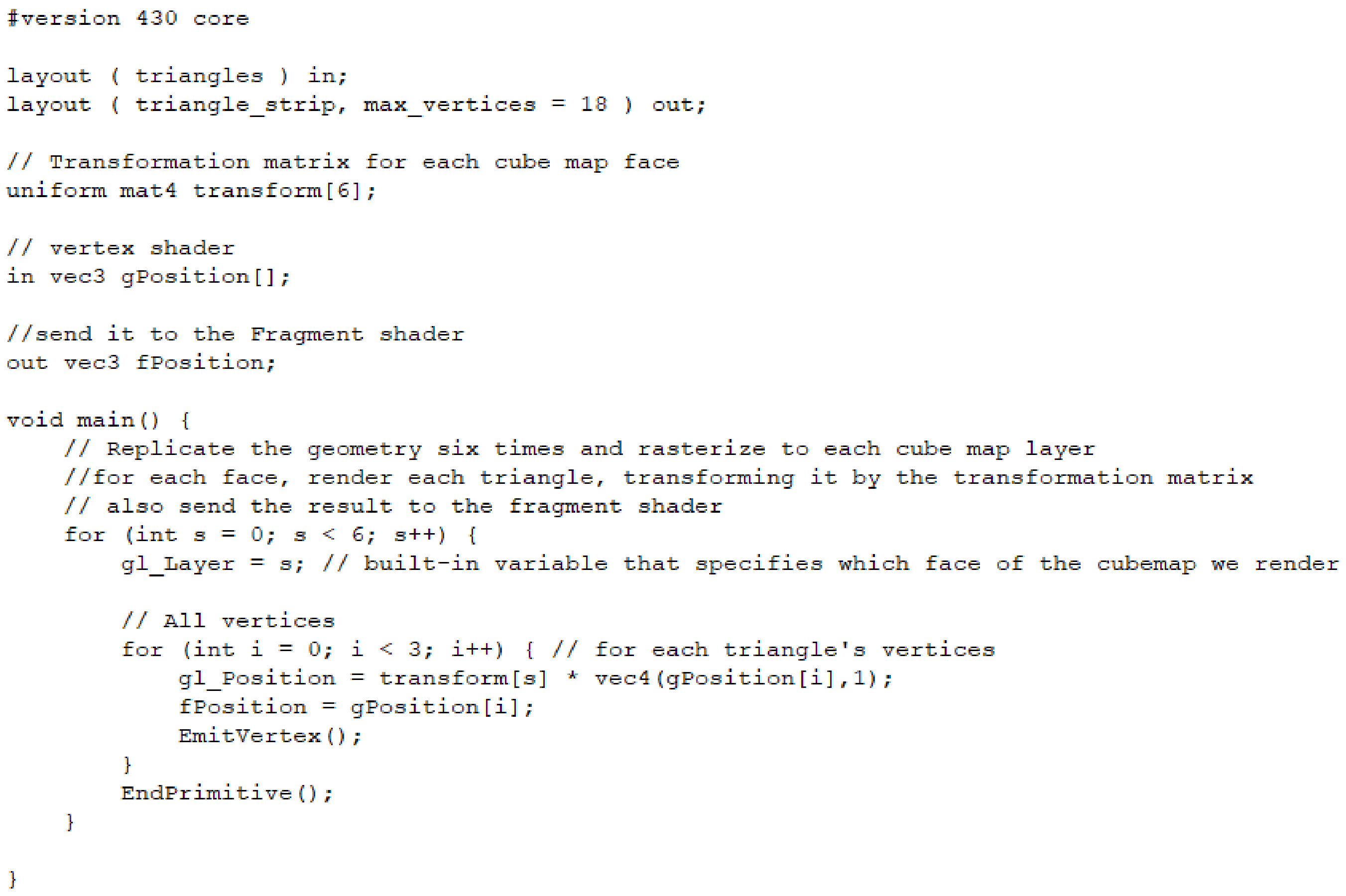
```
//bind the frame buffer object
FBOCube->activate();
glViewport(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT);
//make each face of the cube.
for (int i = 0; i < 6; ++i) {
  glm::vec3 dir;
  glm::vec3 up;
  glm::vec3 o(cameras[1].getPosition());
  switch (i) {
    /* These come from the OpenGL spec */
    case 0:
      dir = glm::vec3(1, 0, 0);
      up  = glm::vec3(0, -1, 0);
      break;
    case 1:
      dir = glm::vec3(-1, 0, 0);
      up  = glm::vec3(0, -1, 0);
      break;
    case 2:
      dir = glm::vec3(0, 1, 0);
      up  = glm::vec3(0, 0, 1);
      break;
    case 3:
      dir = glm::vec3(0, -1, 0);
      up  = glm::vec3(0, 0, -1);
      break;
    case 4:
      dir = glm::vec3(0, 0, 1);
      up  = glm::vec3(0, -1, 0);
      break;
    case 5:
      dir = glm::vec3(0, 0, -1);
      up  = glm::vec3(0, -1, 0);
      break;
  }

  //setting the model view projecion matrix
  glm::mat4 viewTrafo = glm::lookAt(o, o+dir, up);
  glm::mat4 projLightMod = glm::perspective((float)(M_PI*(90.f/180.f)), 1.f, zNear, zFar);

  glm::mat4 VPL = projLightMod * viewTrafo;
  glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, depth->getID(), 0);

  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  scene->drawOpenGL(*ShaderCube, VPL);

}
```

Figure 4.2 – Cube Map creation code

4.4.3.2 Advanced

This method uses the geometry shader (see figure 4.3), where it is used to render to all faces of the cube map once instead of 6 times to create the cube map. (see figure 4.4)

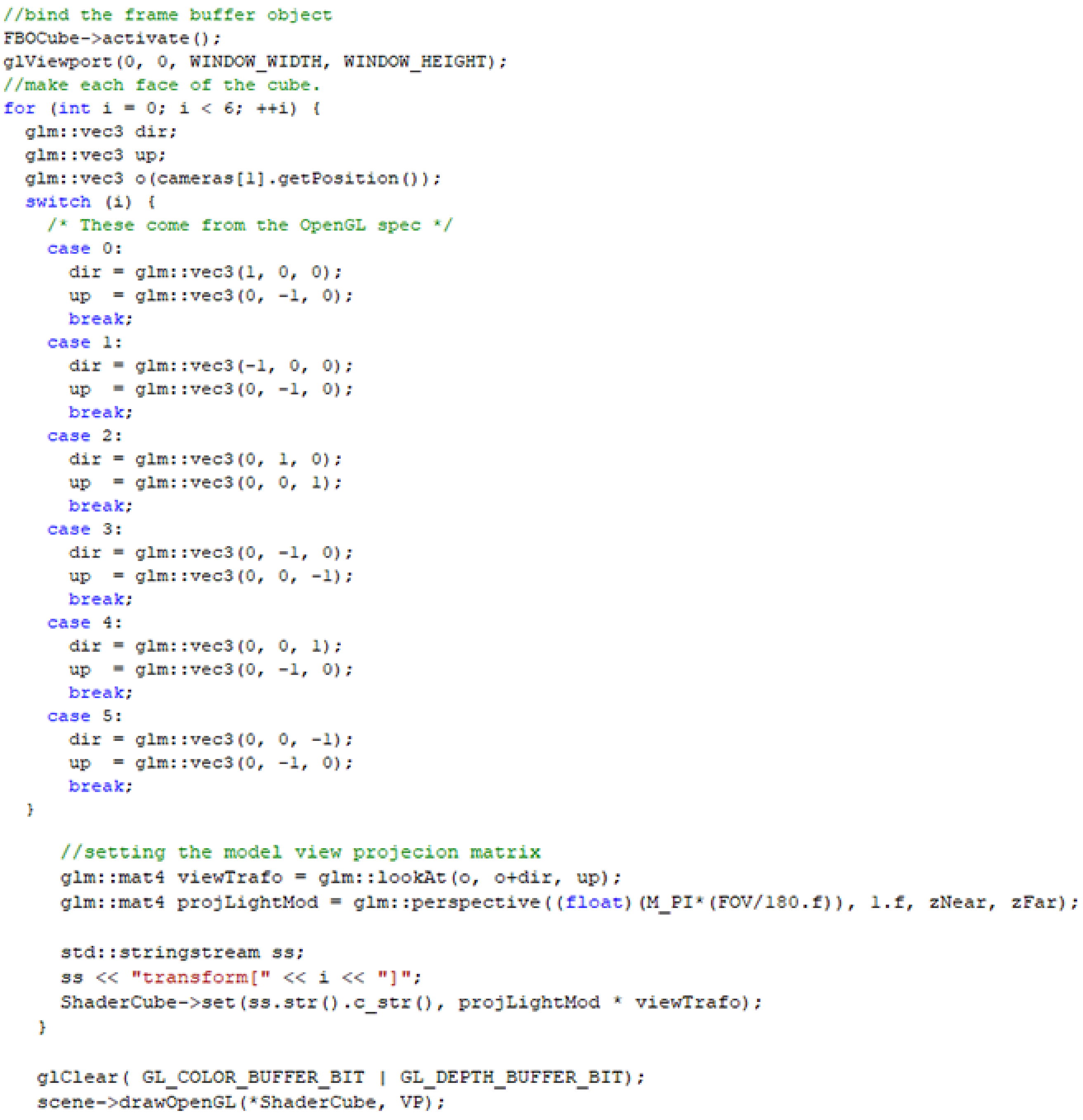
The geometry:



```
#version 430 core

layout ( triangles ) in;
layout ( triangle_strip, max_vertices = 18 ) out;

// Transformation matrix for each cube map face
uniform mat4 transform[6];

// vertex shader
in vec3 gPosition[];

//send it to the Fragment shader
out vec3 fPosition;

void main() {
    // Replicate the geometry six times and rasterize to each cube map layer
    //for each face, render each triangle, transforming it by the transformation matrix
    // also send the result to the fragment shader
    for (int s = 0; s < 6; s++) {
        gl_Layer = s; // built-in variable that specifies which face of the cubemap we render

        // All vertices
        for (int i = 0; i < 3; i++) { // for each triangle's vertices
            gl_Position = transform[s] * vec4(gPosition[i],1);
            fPosition = gPosition[i];
            EmitVertex();
        }
        EndPrimitive();
    }

}
```

Figure 4.3 – Geometry shader code

Cube map creation:



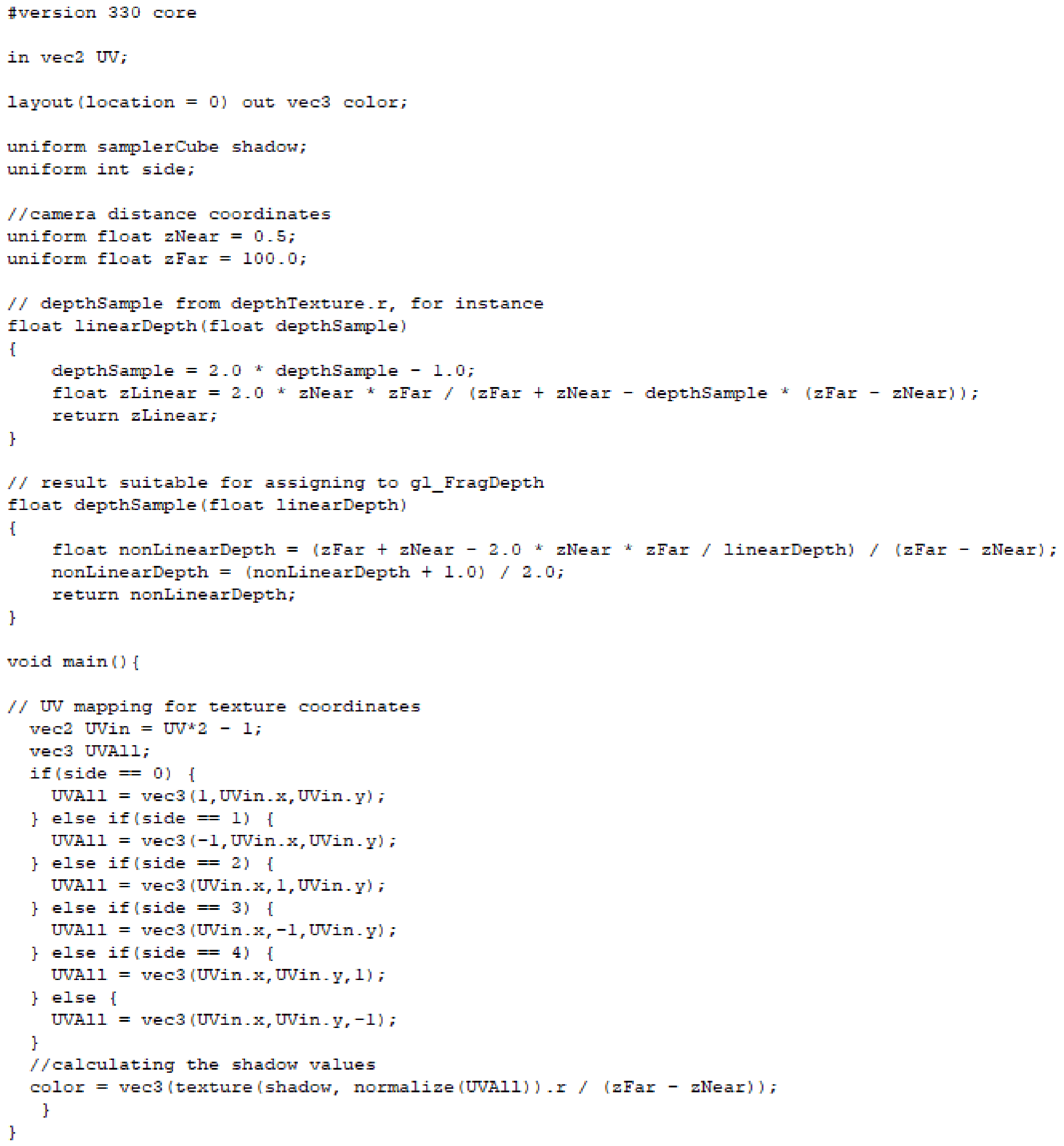
```
//bind the frame buffer object
FBOCube->activate();
glViewport(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT);
//make each face of the cube.
for (int i = 0; i < 6; ++i) {
  glm::vec3 dir;
  glm::vec3 up;
  glm::vec3 o(cameras[1].getPosition());
  switch (i) {
    /* These come from the OpenGL spec */
    case 0:
      dir = glm::vec3(1, 0, 0);
      up  = glm::vec3(0, -1, 0);
      break;
    case 1:
      dir = glm::vec3(-1, 0, 0);
      up  = glm::vec3(0, -1, 0);
      break;
    case 2:
      dir = glm::vec3(0, 1, 0);
      up  = glm::vec3(0, 0, 1);
      break;
    case 3:
      dir = glm::vec3(0, -1, 0);
      up  = glm::vec3(0, 0, -1);
      break;
    case 4:
      dir = glm::vec3(0, 0, 1);
      up  = glm::vec3(0, -1, 0);
      break;
    case 5:
      dir = glm::vec3(0, 0, -1);
      up  = glm::vec3(0, -1, 0);
      break;
  }

     //setting the model view projecion matrix
     glm::mat4 viewTrafo = glm::lookAt(o, o+dir, up);
     glm::mat4 projLightMod = glm::perspective((float)(M_PI*(FOV/180.f)), 1.f, zNear, zFar);

     std::stringstream ss;
     ss << "transform[" << i << "]";
     ShaderCube->set(ss.str().c_str(), projLightMod * viewTrafo);
   }

   glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   scene->drawOpenGL(*ShaderCube, VP);
```

Figure 4.4 – Cube Map creation code

Since we finished creating the cube map now we are going to use it to create the shadow and the shading.

4.4.4 Shadow

The shadow in our program is generated using the cube map that we created in the previous part, to create it we need a special sampler of the type of cube-Sampler, also we need the coordinates of the camera and the position of the light source. (see figure 4.5)

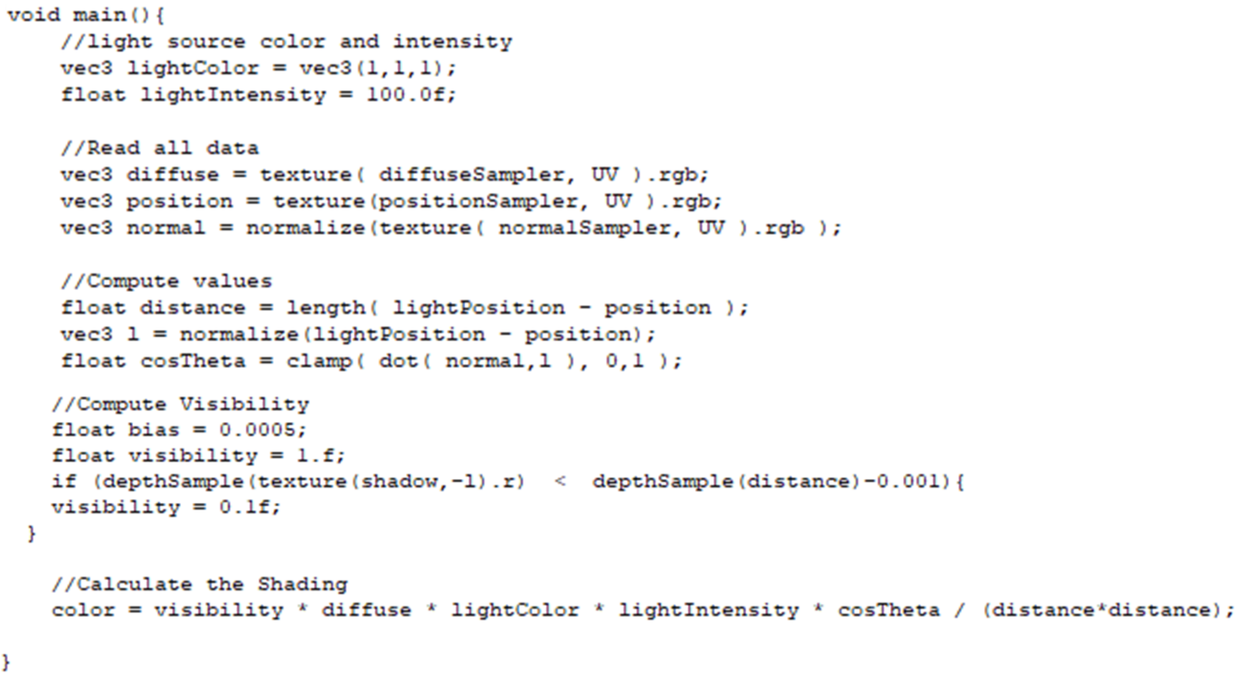


```
#version 330 core

in vec2 UV;

layout(location = 0) out vec3 color;

uniform samplerCube shadow;
uniform int side;

//camera distance coordinates
uniform float zNear = 0.5;
uniform float zFar = 100.0;

// depthSample from depthTexture.r, for instance
float linearDepth(float depthSample)
{
    depthSample = 2.0 * depthSample - 1.0;
    float zLinear = 2.0 * zNear * zFar / (zFar + zNear - depthSample * (zFar - zNear));
    return zLinear;
}

// result suitable for assigning to gl_FragDepth
float depthSample(float linearDepth)
{
    float nonLinearDepth = (zFar + zNear - 2.0 * zNear * zFar / linearDepth) / (zFar - zNear);
    nonLinearDepth = (nonLinearDepth + 1.0) / 2.0;
    return nonLinearDepth;
}

void main(){

// UV mapping for texture coordinates
  vec2 UVin = UV*2 - 1;
  vec3 UVAll;
  if(side == 0) {
    UVAll = vec3(1,UVin.x,UVin.y);
  } else if(side == 1) {
    UVAll = vec3(-1,UVin.x,UVin.y);
  } else if(side == 2) {
    UVAll = vec3(UVin.x,1,UVin.y);
  } else if(side == 3) {
    UVAll = vec3(UVin.x,-1,UVin.y);
  } else if(side == 4) {
    UVAll = vec3(UVin.x,UVin.y,1);
  } else {
    UVAll = vec3(UVin.x,UVin.y,-1);
  }
  //calculating the shadow values
  color = vec3(texture(shadow, normalize(UVAll)).r / (zFar - zNear));
   }
}
```

Figure 4.5 – Shadow calculation code

4.4.5 Shading

The shading in our program is also generated using our cube map, to create it we will need four samplers, the position sampler, the normal sampler, the diffuse sampler, and our previous shadow sampler. Also we will need the light source position since the shading is based on the light. (See figure 3.8 and figure 4.6)



```
void main(){
    //light source color and intensity
    vec3 lightColor = vec3(1,1,1);
    float lightIntensity = 100.0f;

    //Read all data
    vec3 diffuse = texture( diffuseSampler, UV ).rgb;
    vec3 position = texture(positionSampler, UV ).rgb;
    vec3 normal = normalize(texture( normalSampler, UV ).rgb );

    //Compute values
    float distance = length( lightPosition - position );
    vec3 l = normalize(lightPosition - position);
    float cosTheta = clamp( dot( normal,l ), 0,1 );

   //Compute Visibility
   float bias = 0.0005;
   float visibility = 1.f;
   if (depthSample(texture(shadow,-l).r)  <  depthSample(distance)-0.001){
   visibility = 0.1f;
  }

   //Calculate the Shading
   color = visibility * diffuse * lightColor * lightIntensity * cosTheta / (distance*distance);

}
```

Figure 4.6 – Shading calculation

4.5 Results and discussion

4.5.1 Results

In this section, we will present the results of the direct illumination obtained with the classical shadow for each of the scenes, and make some comparisons. Here we converted the results images from HDR to LDR, since our screen is an LDR screen, while the output of our program is an HDR image. We used Luminance HDR 2.4.0 program to convert these images.

4.5.1.1 According to the FPS

Sponza

- Classic method (Figure 4.7):

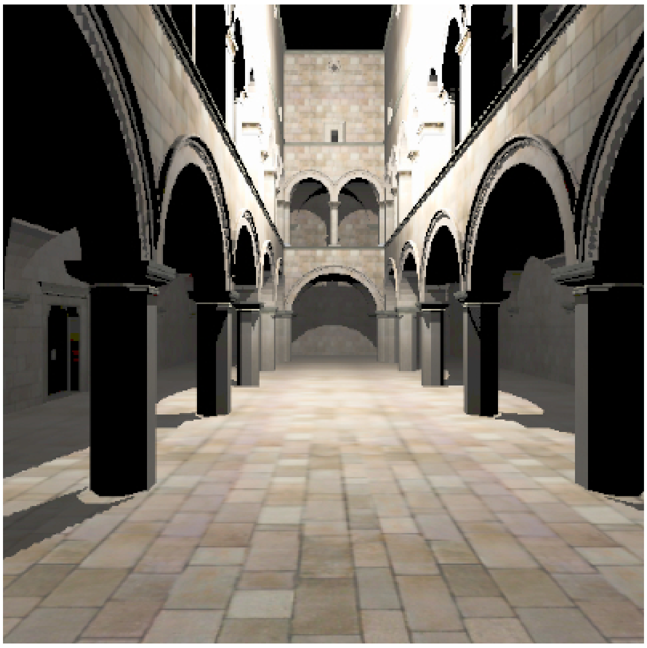


Figure 4.7 – Sponza with the classic method

The FPS with this method is 72.6586

- Advanced method (Figure 4.8):

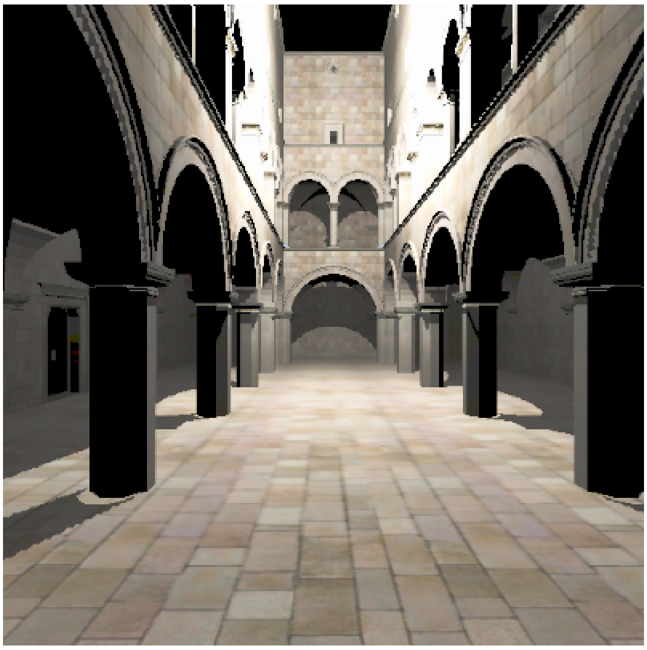


Figure 4.8 – Sponza with the advanced method

The FPS with this method is 102.838, which is better than the first method.

Crytek Sponza

- Classic method (Figure 4.9):

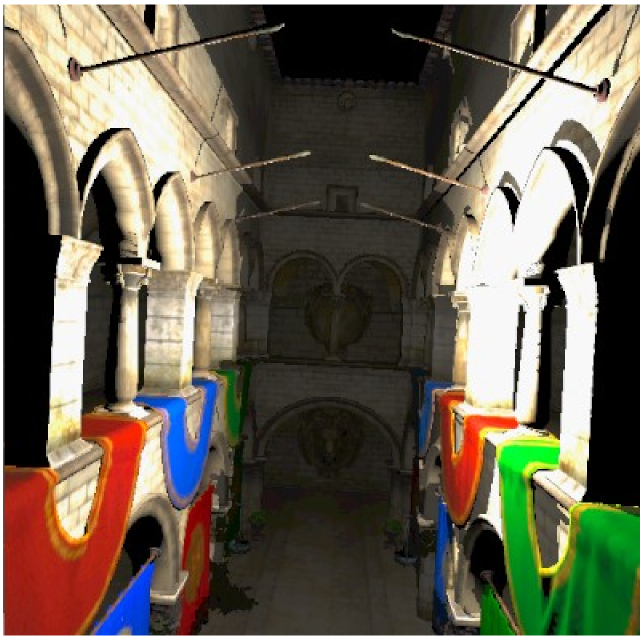


Figure 4.9 – Crytek Sponza with the classic method

The FPS with this method is 72.5953.

- Advanced method (Figure 4.10):

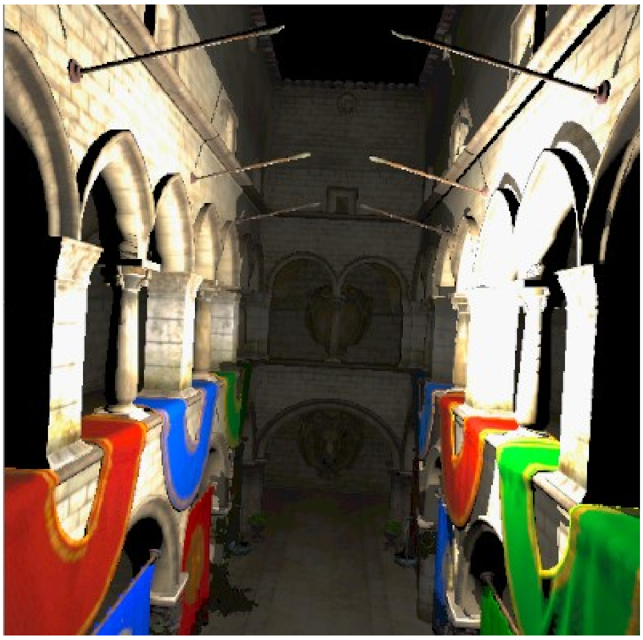


Figure 4.10 – Crytek Sponza with the advanced method

The FPS with this method is 102.312, which is better than the first method.

Sibenik

- Classic method (Figure 4.11):

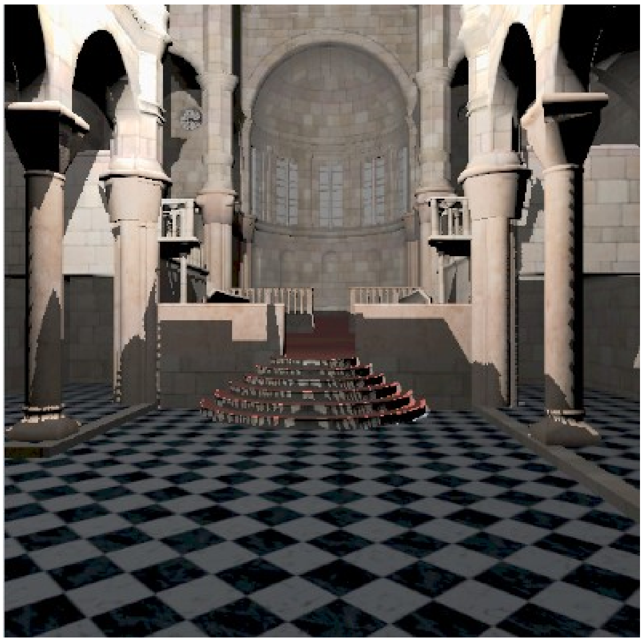


Figure 4.11 – Sibenik with the classic method

The FPS with this method is 4.64391.

- Advanced method (Figure 4.12):

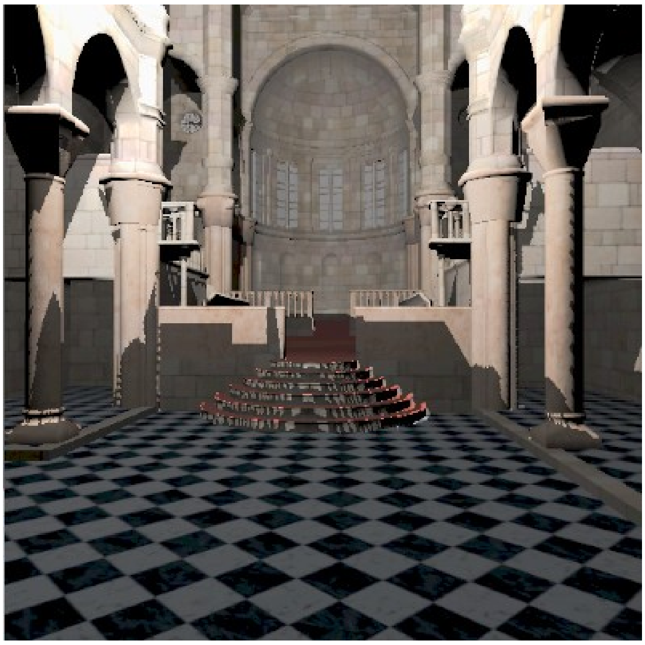


Figure 4.12 – Sibenik with the advanced method

The FPS with this method is 17.9488, which is better than the first method.

Conference

- Classic method (Figure 4.13):

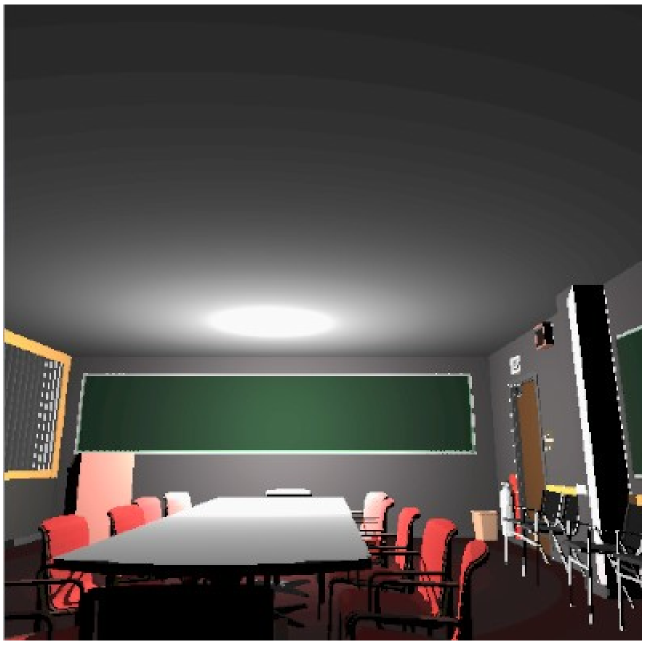


Figure 4.13 – Conference with the classic method

The FPS with this method is 65.1211.

- Advanced method (Figure 4.14):

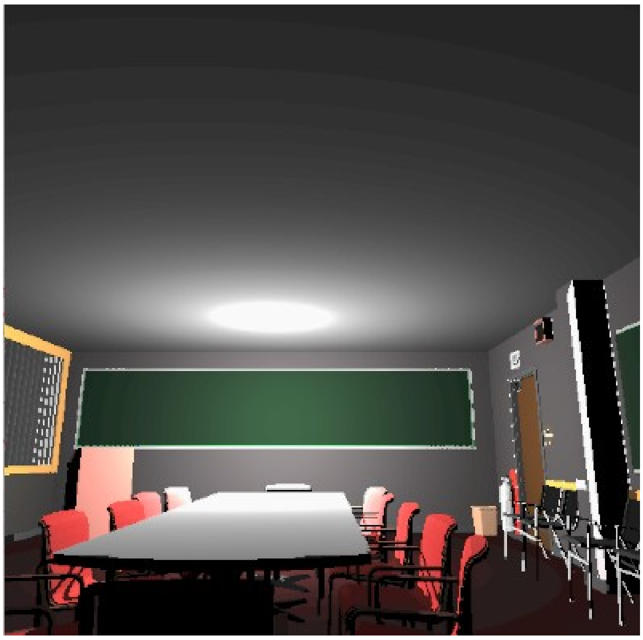


Figure 4.14 – Conference with the advanced method

The FPS with this method is 83.0082, which is better than the first method.

4.5.2 Discussion

In our system we could create both the shading and the shadow for our scenes, so we are going to point out some details in our final scenes images to show them well.

4.5.2.1 Sponza

Shading:

In the (Figure 4.15) we can clearly see the details of the shading and the color gradient in the arches.

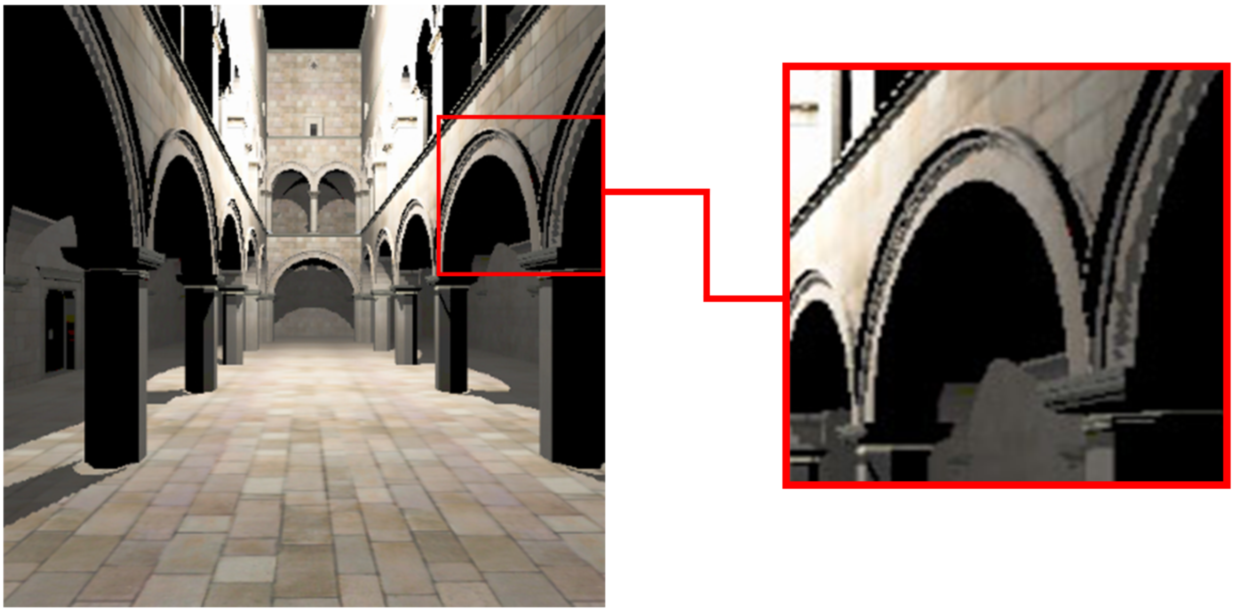


Figure 4.15 – Sponza Shading details

Shadow:

Also in the (Figure 4.16) we can see the details of the shadow of the columns that was created according to the light source position in the middle of the scene.

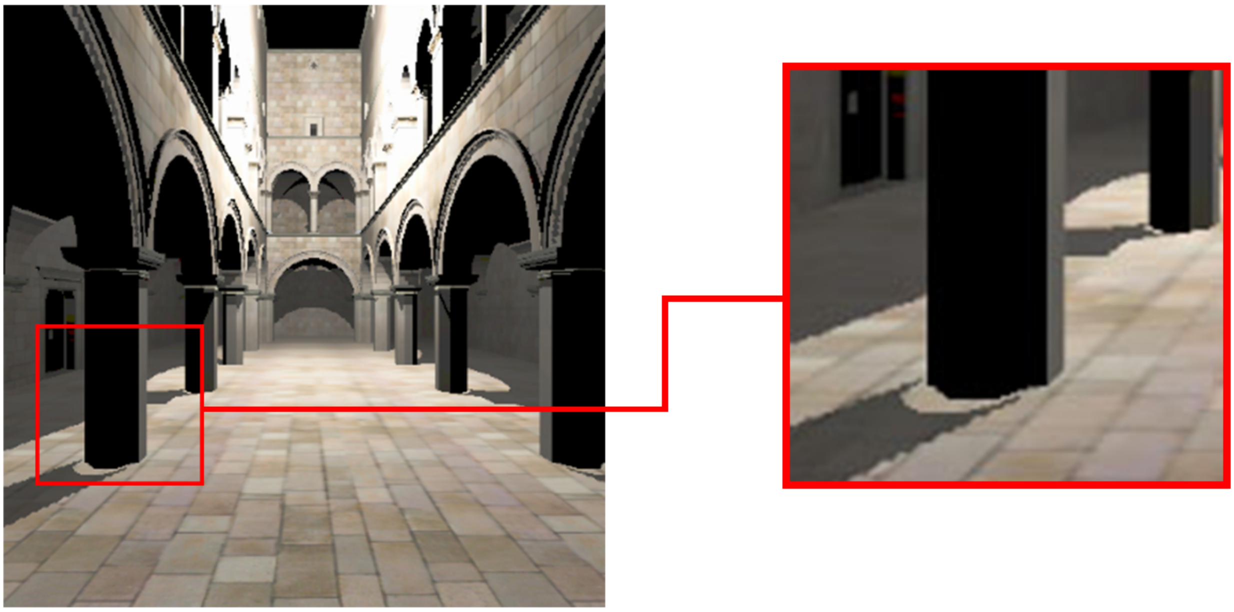


Figure 4.16 – Sponza Shadow details

4.5.2.2 Crytek Sponza

Shading:

In the (Figure 4.17) we can clearly see the details of the shading and the color gradient in the windows.

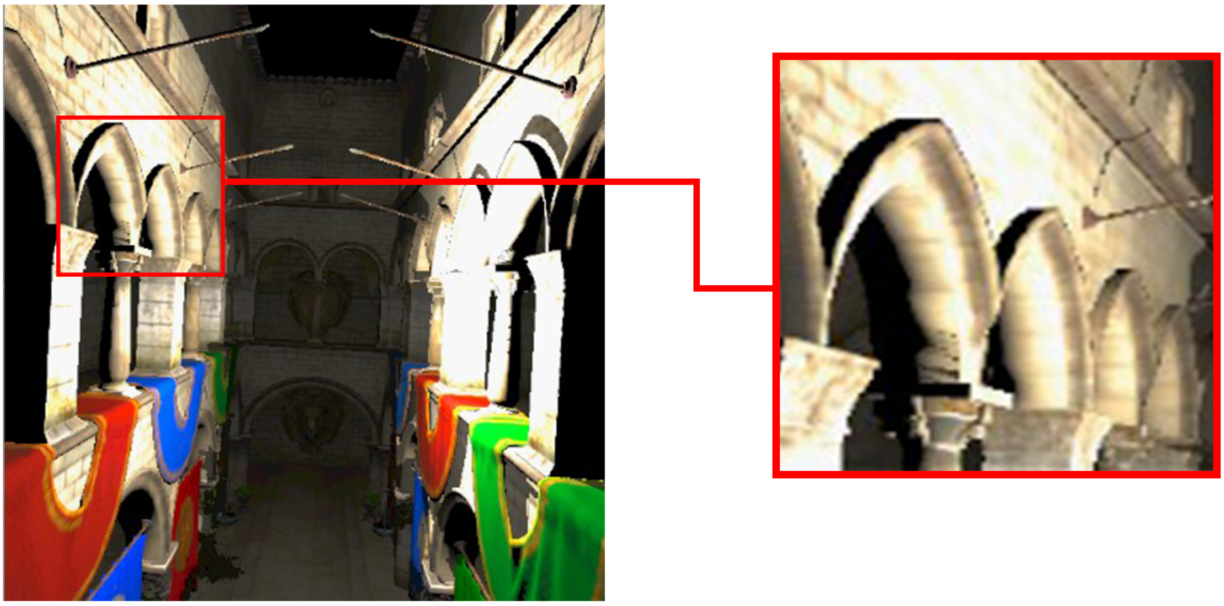


Figure 4.17 – Crytek Sponza Shading details

Shadow:

Also in the (Figure 4.18) we can see the details of the shadow of the the other side windows that was created according to the light source position in the middle of the scene.

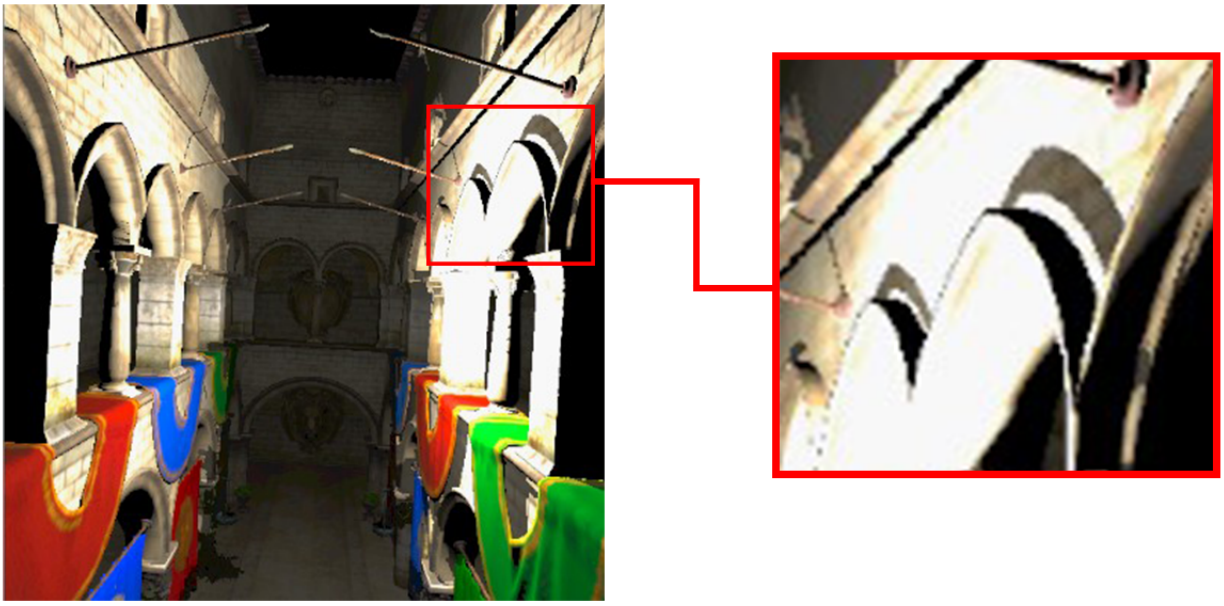


Figure 4.18 – Crytek Sponza Shadow details

4.5.2.3 Sibenik

Shading:

In the (Figure 4.19) we can clearly see the details of the shading and the color gradient in the columns.

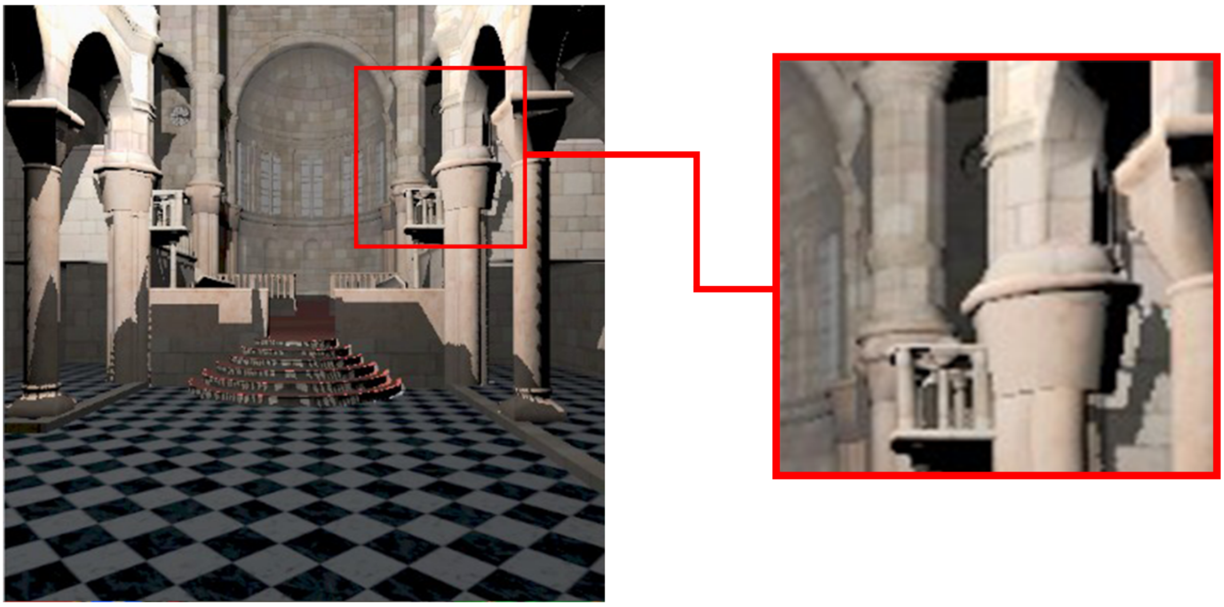


Figure 4.19 – Sibenik Shading details

Shadow:

Also in the (Figure 4.20) we can see the details of the shadow of the walls that was created according to the light source position behind the scene.

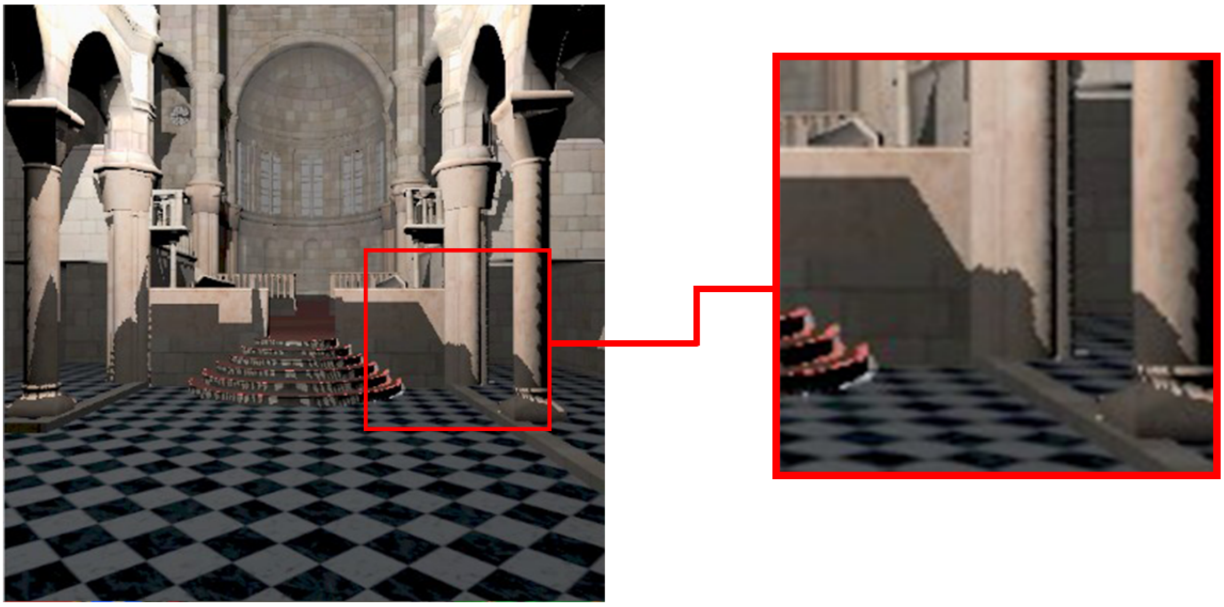


Figure 4.20 – Sibenik Shadow details

4.5.2.4 Conference

Shading:

In the (Figure 4.21) we can clearly see the details of the shading and the color gradient in the door and the chairs' edges.

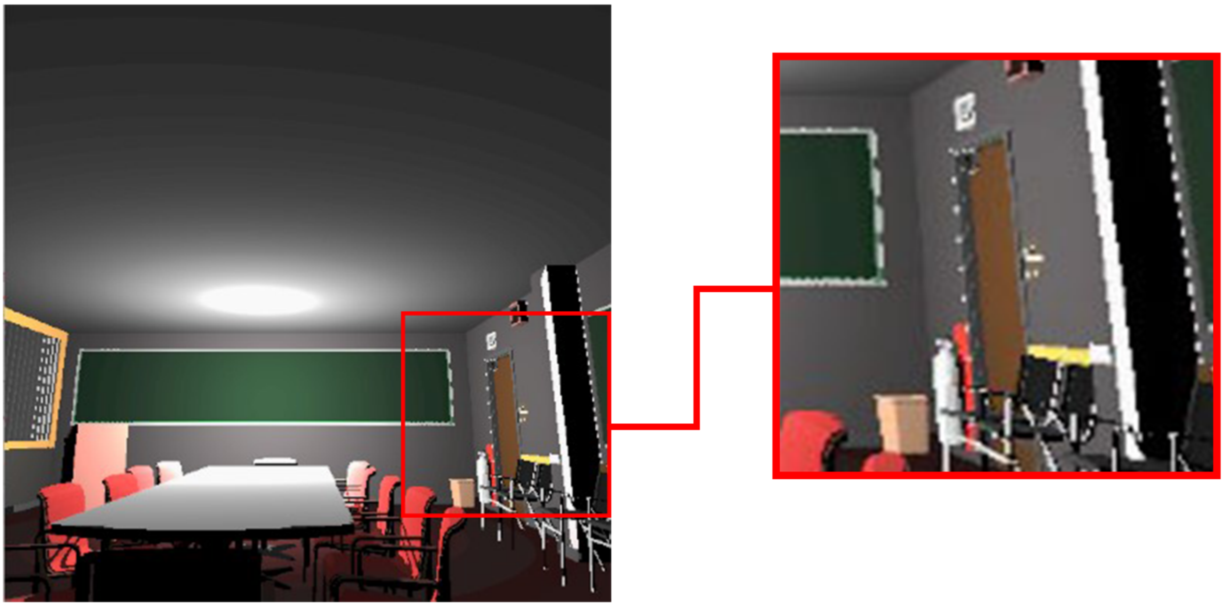


Figure 4.21 – Conference Shading details

Shadow:

Also in the (Figure 4.22) we can see the details of the shadow of the chairs that was created according to the light source position in the middle close to the roof.

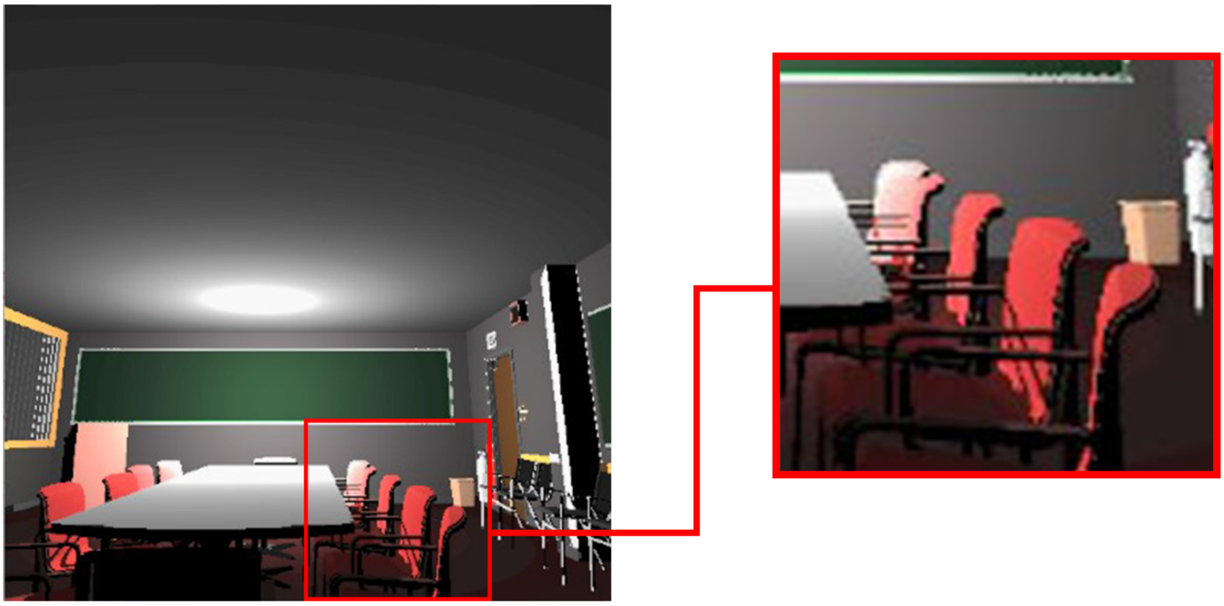
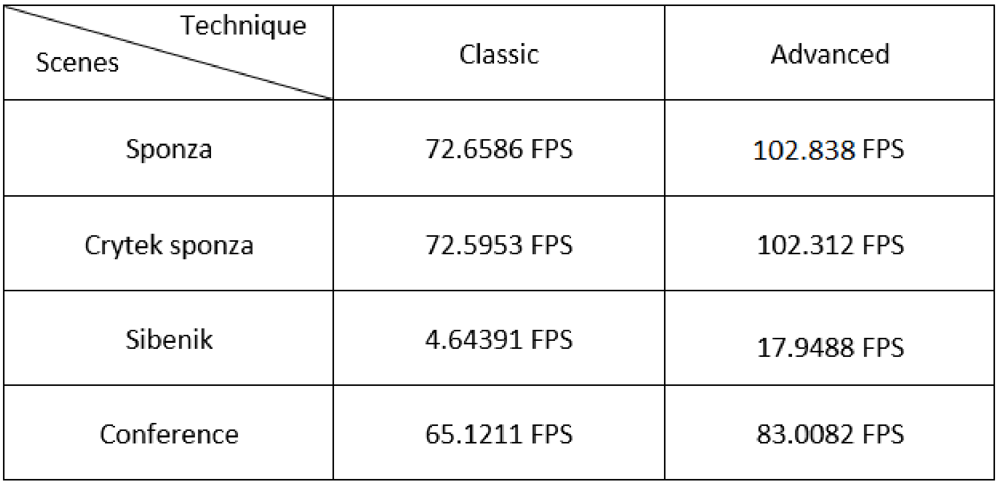


Figure 4.22 – Conference Shadow details

Comparison between the 2 methods:

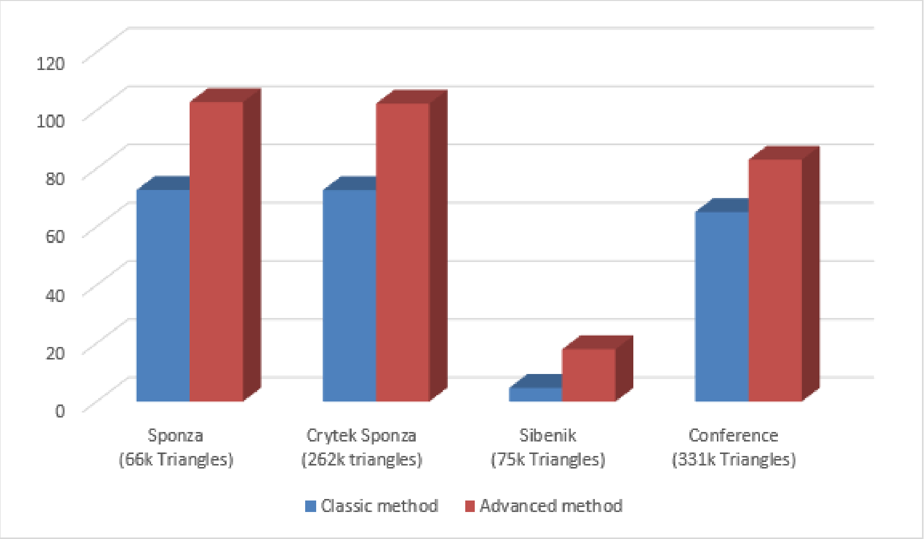
Table:



Technique Scenes	Classic	Advanced
Sponza	72.6586 FPS	102.838 FPS
Crytek sponza	72.5953 FPS	102.312 FPS
Sibenik	4.64391 FPS	17.9488 FPS
Conference	65.1211 FPS	83.0082 FPS

As we can see in the previous table, the FPS has increased so much in the advanced method than the classic one. In Sponza scene the FPS has increased from 72.6586 to 102.838 which makes it a 30.1794 FPS difference. In Crytek Sponza scene the FPS has increased from 72.5953 to 102.312 which makes it a 29.7167 FPS difference. In Sibenik scene the FPS has increased from 4.64391 to 17.9488 which makes it a 13.30489 FPS difference. In Sibenik scene the FPS has increased from 65.1211 to 83.0082 which makes it a 17.8871 FPS difference.

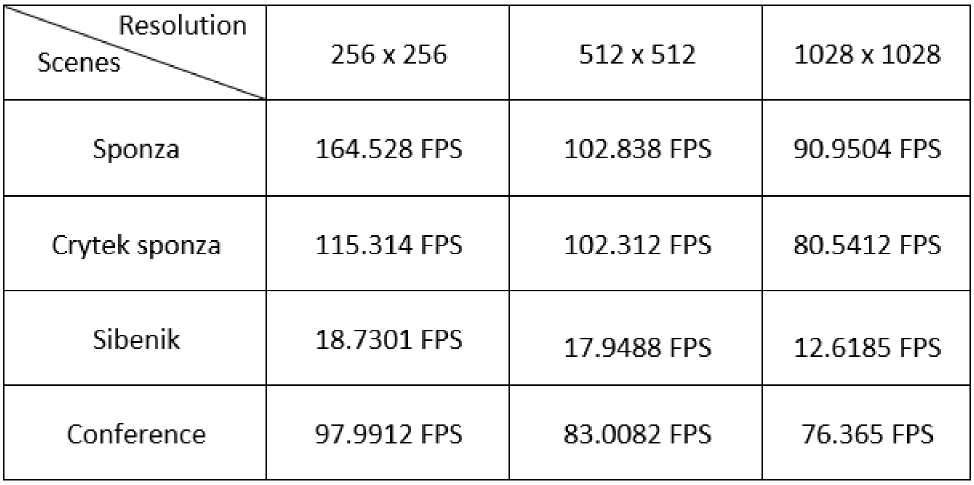
Column Chart:



In the previous column chart, we can see that the FPS of the advanced method is always higher than the FPS in the classic method, which confirms that our method is much better to use.

Comparison between the different resolutions with our method:

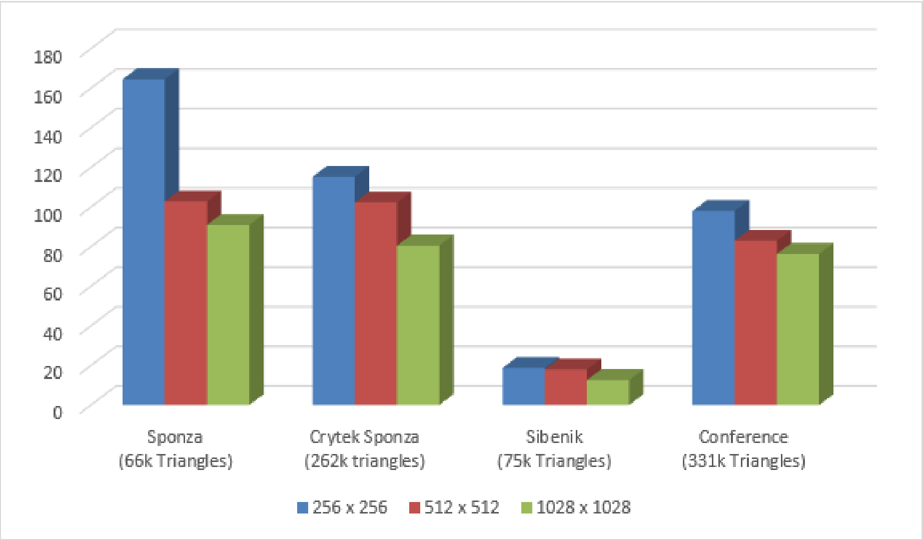
Table:



Resolution / Scenes	256 x 256	512 x 512	1028 x 1028
Sponza	164.528 FPS	102.838 FPS	90.9504 FPS
Crytek sponza	115.314 FPS	102.312 FPS	80.5412 FPS
Sibenik	18.7301 FPS	17.9488 FPS	12.6185 FPS
Conference	97.9912 FPS	83.0082 FPS	76.365 FPS

As we can see in the previous table, the FPS changes in each resolution. In Sponza scene the FPS in the resolution 256x256 was 164.528, and gets down in the second resolution to 102.838, and it continues to decrease in the third resolution 1028x1028 to 90.9504. In Crytek Sponza scene the FPS in the resolution 256x256 was 115.314, and gets down in the second resolution to 102.312, and it continues to decrease in the third resolution 1028x1028 to 80.5412. In Sibenik scene the FPS in the resolution 256x256 was 18.7301, and gets down in the second resolution to 17.9488, and it continues to decrease in the third resolution 1028x1028 to 12.6185. In Conference scene the FPS in the resolution 256x256 was 97.9912, and gets down in the second resolution to 83.0082, and it continues to decrease in the third resolution 1028x1028 to 76.365.

Column Chart:



In the previous column chart whenever the resolution gets higher the FPS gets lower, but the difference is always smaller than the first method so our method is still better.

4.6 Conclusion

In this chapter we have described the tools for carrying out our project, presenting the different results that we obtained (Shadowed and Shaded scenes) using certain method of lighting called Cube Mapping to reduce the calculating time, Where we used OpenGL and C++ as a programming language, in a Visual Studio 2008 environment.

We made a comparison between two methods, a classic one that doesn't use the geometry shader, and our advanced method that uses the geometry shader, this last one can reduce the number of rendering passes from 6 to 2 rendering passes, which means less calculating time (as we saw in the FPS of the results).

Conclusion

Lighting takes a really important part in the 3D graphics, a scene without lighting can not be seen by the camera. Although there are a lot of lighting techniques, not all of them give a good quality result in a small calculating time. Here we could highlight a specific method called Cube Mapping.

In this project we have presented the direct illumination using the Cube Mapping technique on GPU, where we saw how to apply the Cube Map to calculate the lighting of a scene. This technique can generate a really good quality images in a really small calculating time, which makes it a widely used technique in 3D computer graphics, specially video games that needs to be generated in real-time.

We could specify two different methods in the Cube Mapping technique, a classic one that doesn't use the geometry shader, and our method that uses the geometry shader. The second one is an advanced method based on the first one. This method can reduce the rendering passes from six to two rendering passes only which makes the calculating time lower.

We believe that this lighting technique can improve the work in the technical field of production of 3D movies and video games.

Bibliography

[1] Günter Wyszecki and Walter Stanley Stiles. "Color science: concepts and methods, quantitative data and formulas". In: (1982).

[2] Maryam. "The ultimate guide to lighting fundamentals for 3D". In: *DreamFarm Studios* (2021).

[3] Greg Nichols, Rajeev Penmatsa, and Chris Wyman. "Direct illumination from dynamic area lights with visibility". In: *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. 2010, pp. 1–1.

[4] Ben Herila. "Introduction to computer graphics". In: 2010.

[5] Joey de Vries. *Basic Lighting*. 2016.

[6] Brad Smith. "Illustration of the components of the Phong reflection model (Ambient, Diffuse and Specular reflection)". In: *CC BY-SA 3.0* (2006).

[7] Scratchapixel 2.0. "Global Illumination and Path Tracing". In: 2020.

[8] Cyril Crassin et al. "Interactive indirect illumination using voxel cone tracing: A preview". In: *Symposium on Interactive 3D Graphics and Games*. 2011, pp. 207–207.

[9] James T Kajiya. "The rendering equation". In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. 1986, pp. 143–150.

[10] Dave Mount A. Varshney D. M. Mount. "Global Illumination Models CMSC 427: Global Illumination Models". In: CMSC 427, 2007.

[11] Alexander Majercik Morgan McGuire Adam Marrs. "RTX Global Illumination Part I". In: *NVIDIA Developper* (2019).

[12] Greg Nichols and Chris Wyman. "Multiresolution splatting for indirect illumination". In: *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. 2009, pp. 83–90.

[13] Greg Zaal. "How to Create Your Own HDR Environment Maps". In: (2016).

[14] Tom Thorne. "Computer Graphics 9 - Environment mapping and mirroring". In: 2014.

[15] Aaron Hong Ravi Ramamoorthi. "CS294-13: Special Topics, Advanced Computer Graphics, Lecture 8". In: *University of California, Berkeley* (2009).

[16] Kevin Hongtongsak. “Dynamic-Cubemaps”. In: 2016.

[17] Chris Brennan. *Diffuse Cube Mapping*. 2002.

[18] “Tutorial 13: Cube Mapping”. In: 2018.

[19] Margus Luik Raimond Tunnel Jaanus Jaggo. “Computer Graphics Learning Materials, Environment Mapping”. In: 2018.

[20] Anton Gerdelan. “Cube Maps: Sky Boxes and Environment Mapping”. In: 2016.

[21] G. Zachmann. “OpenGL Cube Map Texturing”. In: 1999.

[22] Paul Bourke. “Converting to/from cubemaps”. In: (2020).

[23] Joey de Vries. “Cubemaps”. In: 2019.