Politecnico di Milano
School of Industrial and Information Engineering
Master of Science in Computer Science and Engineering
Department of Electronics, Information and Bioengineering

Occluded Environment Maps Rendering

Relatore: Prof. Marco Gribaudo
Correlatore: Prof. Pietro Piazzolla

Master Thesis of:
Alessio Ambrosj Matr. 804792

Academic Year 2015–2016

"There is nothing noble in being superior to your fellow men.
True nobility lies in being superior to your former self."

Hernest Hemingway

Ringraziamenti

I miei primi ringraziamenti vanno ai miei genitori: durante questi sette anni non sempre tutto è andato come doveva andare, ma loro non hanno mai smesso di credere in me e supportarmi, e se oggi posso emozionarmi nello scrivere queste parole nella prefazione di questo testo, è solo merito loro. Vorrei ringraziare zio Gufo, zia Mokia e nonna Pezzetti perché ci sono sempre stati e sono stati per me, come dei genitori extra. Ringrazio i miei fratellini Jacopo e Lorenza, perché ormai non sono pi ú cosí "ini" ma il loro affetto è sempre stato "one". Un ringraziamento particolare va al professor Gribaudo, per avermi seguito ed avermi rassicurato ogni volta che pensavo di aver incontrato un ostacolo. A questo proposito, ringrazio Pietro Piazzolla, senza il quale sarei ancora davanti ad uno schermo con una teiera disegnata a metà. Un ringraziamento a Dorosroro, Jorge, Pieter, Anna, Gigi (grasso) perché con loro ero davvero a Casa. Grazie ad Elisa, per aver condiviso con me gran parte di questo cammino. Un ringraziamento agli amici di sempre Gio,Cup, Francesco, Fabrizio, Attilio, Davide, Larry, Dodo per le serate, le giornate, le avventure e tutto il tempo passato a ridere e, diciamoci la verità, a bere! Grazie ad Arci, Ane, Beg, Billy, Cesa, Devo, Serena, mi scenderà una lacrimuccia quando toglierò la Polibox. Grazie ai "big" dello Studio Albatros, ad Alberto, Salvo, Matteo perché il lavoro è tanto, la nostra esperienza è poca, e continuiamo ad andare avanti. Grazie alla Tigre Nera, a Sama, Ricky, Gasper e Jessica per avermi fornito la grinta necessaria a rimanere sano di mente in questi ultimi anni. Grazie a Cristiano (il Maestro), Andrea (Barbeingel), Alban(ushi), Gino, Renisa, Claudia, Lara perché sono tanti i ricordi che abbiamo insieme e spero non siano ancora abbastanza. Grazie ad Arianna, Giulia e Roberta (gnih) per le risate, le mangiate e le giocate di questi ultimi mesi, che mi hanno fatto sentire sette anni più giovane. Ed

infine grazie a tutti gli Zzipa che mi hanno sopportato: Davide, Nando, Ste, Nicho, Rodry, Tommy, Claudia (Garrison), Fabia e Claudia.

Abstract

This master's thesis deals with the issue of real time Global Illumination (GI) in Computer Graphics. The project focused on developing an open source algorithm for real time rendering, based on a Image Based Lighting (IBL) technique called Environment Mapping, and a very common real time shadowing technique called Shadow Mapping. The goal of the algorithm is to achieve photorealistic, interactive and dynamic ambient illumination, given a surrounding represented as an equirectangular panorama high dynamic range image (HDRI), namely an Environment Map (EM). The EM is preprocessed to locate a certain number of light sources (embodied by sets of pixels and a direction), and to generate a collection of new EMs which are used in the rendering process by means of their projection on Spherical Harmonics (SH) basis. For each light source a directional Shadow Map (SM) is computed and used as discriminant in assessing if a given fragment (pixel) is lit by the corresponding light source. The set of light sources that lights a fragment is mapped to a generated EM which is employed to compute the pixel's final color. The presented approach is well suited for outdoor EMs (where usually there is only one natural light source) but it has been primarly designed for indoor EMs (two or more artificial light sources) since it also offers a mechanism to switch on and off lights in real time.

Contents

1 Introduction **1**
1.1 About Global Illumination 1
1.2 Occluded EMs Rendering in a nutshell 3
1.3 Overview 4

2 State of the Art **6**
2.1 Environment Mapping for diffuse component 6
2.1.1 Prefiltering an EM 7
2.1.2 The Spherical Harmonics Approach 8
2.2 Common Projections 9
2.2.1 Sphere Mapping 9
2.2.2 Cube Mapping 10
2.2.3 Equirectangular Mapping 11

3 Technical Background **13**
3.1 LDR and HDR 13
3.2 Lambert Lighting Model 14
3.3 Blinn Reflection Model 16
3.4 K-Means Clustering 17
3.5 Spherical Harmonics 18
3.5.1 Introduction to Spherical Harmonics 18
3.5.2 Definition 19
3.5.3 Properties 20
3.5.4 SH Lighting 22
3.6 Shadow Mapping 25

4 Occluded Environemnt Maps Rendering **29**
4.1 Light Sources Location 30
4.2 OEMs SH Projection 33
4.3 Rendering cycle 34

5 Complexity and Results **40**
5.1 Complexity Evaluation 40
5.1.1 Preprocessing Phase 40
5.1.2 Rendering Cycle 41
5.2 Graphic Results 42
5.2.1 A simple scene 44
5.2.2 A comparison with Unity standard shader 45
5.2.3 A complex scene 46

Conclusion **48**

A List of the first 16 Spherical Harmonics **50**

B OpenGL Graphic Pipeline **52**

C Model Space, World Space, View Space and Screen Space **56**

Bibliography **58**

List of Figures

1.1 *Global Illumination: Cornell Box with sphere example* 2
1.2 *OpengGL's coordinates system* 5

2.1 *Comparison between Grace's Cathedral spherical EM and DIEM* 7
2.2 *Comparison between Grace's Cathedral regular spherical EM the SH generated one* 8
2.3 *Sphere mapping example* 9
2.4 *Cubemap faces explanation* 10
2.5 *Grace's Cathedral Equirectangular environment map* 12

3.1 *Comparison of reflection mapping employing LDR vs. employing HDR* 14
3.2 *Lambert's model visualization* 15
3.3 *Blinn's model visualization* 16
3.4 *Standard coordinates system* 18
3.5 Representation of the first 3 bands of real spherical harmonics 20
3.6 *Shadow mapping visual explanation* 25
3.7 *Multiple fragments mapping on the same shadow map texel* . . 26
3.8 *Shadow Acne artifact* 26
3.9 *Bias correction effect* 27
3.10 *Peter Panning artifcat* 27
3.11 *Front face culling visual explanation* 28

4.1 *Grace's Cathedrals OEMs for* $nLightSources = 2$ 33

5.1 *Stanford bunny rendered with Grace's Chatedral environment map in OER* 44

5.2 *Stanford Bunny rendered in Unity with environment mapping and two directional lights* 46
5.3 *Composite scene rendered with Grace's Chatedral environment map in OER* 47

B.1 *Diagram of the Rendering Pipeline. The blue boxes are programmable shader stages.* 53

C.1 *Vertex of the Utah Teapot in position (1,1,1)* 56
C.2 *Utah Teapots with axis of Model Space, World Space and View Space* 57

List of Tables

2.1 OpenGL Cubemap layout 11

5.1 *Software adopted for the showcase* 43
5.2 Hardware adopted for the showcase 43

List of Algorithms

1 *Phase One: Light source location and OEM generation* 32
2 *Phase Two: OEMs SH Projections* 34
3 Shadow Map Pass: Vertex Shader 35
4 Shadow Map Pass: Fragment Shader 36
5 *Final Pass: Vertex Shader* 37
6 *Final Pass: Fragment Shader* 39

Chapter 1

Introduction

1.1 About Global Illumination

Global illumination (GI) or indirect illumination is a general name for a group of algorithms used in 3D computer graphics that are meant to add more realistic lighting to 3D scenes. Such algorithms take into account not only the light which comes directly from a light source (direct illumination), but also subsequent cases in which light rays from the same source are reflected by other surfaces in the scene, whether reflective or not (indirect illumination). GI effects include the soft darkening under an object and near edges, color bleeding, etc. These effects are subtle, but important for the realism of an image. For example, if you look at figure 1.1 closely you'll notice that the green color of the wall is being cast onto the sphere on the right side of the image. That effect is referred to as indirect lighting because the green light isn't being cast directly from a light but rather is the result of a white light being cast in onto the green wall which is then bleeding onto the nearby sphere.

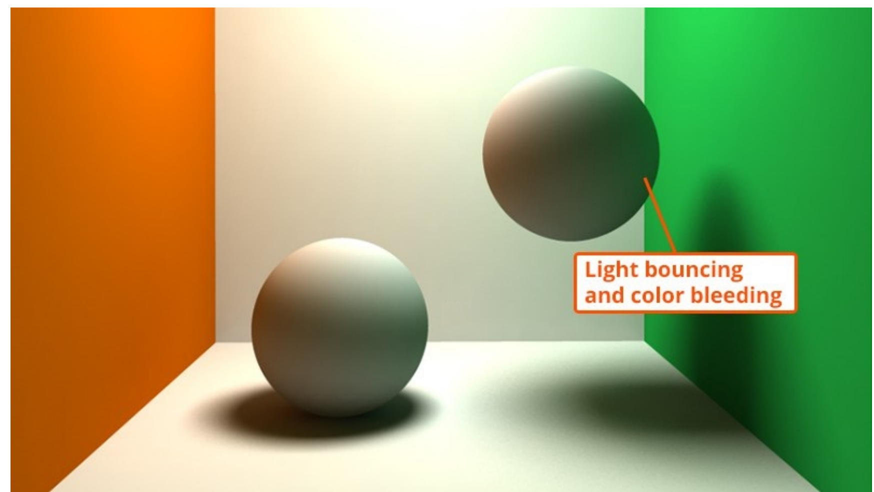


Figure 1.1: *Global Illumination: Cornell Box with sphere example*

Radiosity, Ray Tracing, Beam Tracing, Cone Tracing, Path Tracing, Metropolis Light Transport, Ambient Occlusion, Photon Mapping, and Image Based Lighting are examples of algorithms used in global illumination, some of which may be used together to yield results that are not fast, but accurate. These algorithms model diffuse inter-reflection which is a very important part of global illumination; however most of these (excluding radiosity) also model specular reflection, which makes them more accurate algorithms to solve the lighting equation (a mathematical model introduced by Kajiya in 1986) and provide a more realistically illuminated scene. The algorithms used to calculate the distribution of light energy between surfaces of a scene are closely related to heat transfer simulations performed using finite-element methods in engineering design.

Images rendered using global illumination algorithms often appear more photorealistic than images rendered using only direct illumination algorithms. However, such images are computationally more expensive and consequently much slower to generate. Traditionally, video games and other realtime graphics applications have been limited to direct lighting, while the calculations required for indirect lighting were too slow so they could only be used in non-realtime situations such as CG animated films. A way for games to work around this limitation is to calculate indirect light only for objects and surfaces that are known ahead of time to not move around (that are static). That way the slow computation can be done ahead of time, but

since the objects don't move, the indirect light that is pre-calculated this way will still be correct at runtime.One common approach is to compute the global illumination of a scene and store that information with the geometry, this method is called radiosity. That stored data can then be used to generate images from different viewpoints for generating walkthroughs of a scene without having to go through expensive lighting calculations repeatedly.

For non static objects the problem is not easily solved. In real-time 3D graphics, the diffuse inter-reflection component of global illumination is sometimes approximated by an "ambient" term in the lighting equation, which is also called "ambient lighting" or "ambient color" in 3D software packages. Though this method of approximation (also known as a "cheat" because it's not really a global illumination method) is easy to perform computationally, when used alone it does not provide an adequately realistic effect. Ambient lighting is known to "flatten" shadows in 3D scenes, making the overall visual effect more bland. However, used properly, ambient lighting can be an efficient way to make up for a lack of processing power.

Another common approach is to consider ambient light as infinite distant light that is, in jargon, a directional light. Following this reasoning, ambient lighting information can be preocomputed and stored in a texture, called Environment Map or Irradiance Map. The ambient component of the color of a fragment is then computed with a texture look up using the fragment's normal as texture coordinate. This is an efficient Image Based Lighting technique that goes under the name of Environment Mapping or Irradiance Mapping.

1.2 Occluded EMs Rendering in a nutshell

In the following, a brief overview of the presented approach is given. The Occluded EMs Rendering (OER) is an IBL technique derived from the irradiance environment maps approach, hence its goal is to achieve real time global illumination coherent with an Environment Map (EM). The procedure is divided in three conceptual phases.

The first phase of the algorithm scans the EM image in order to locate a user-specified number $nLightSources$ of light sources, embodied by a set of pixels and a direction. Given the original EM, a collection of $2^{nLightSources}$

new EMs are generated in a fashion that is described thoroughly in chapter 4.

The second phase of the algorithm employs special functions defined on the surface of a sphere (a unit sphere in our case), named Spherical Harmonics (SH), as basis of a function space which the EMs are projected onto. The projection is characterized by an "order", $expansionOrder$, that determines which and how many SH functions are used in the process. Every EM is then approximated by a set of real numbers, one per different basis function.

The third and last phase is rendering itself. The rendering cycle starts performing a so-called "Shadow Map Pass" for every light source detected in the first phase. The results of these passes are $nLightSources$ depth textures named "Shadow Maps" (SMs). Every fragment's depth is then checked against these maps to obtain a set of light sources for which the fragment is not in shadow. Depending on this set, an appropriate EM is chosen. The final color of the fragment is then computed with a convolution between a common lambertian cosine (see 3.2) and the chosen EM plus a specular component derived by the light directions of every light hitting the fragment. A convolution might seem an operation too expansive to be evaluated per frame but, as it will be shown, thanks to SH properties it results in a dot product or a matrix-vector multiplication.

1.3 Overview

This paperwork is structured so as to firstly offer to the reader a general idea of the state of the art on Environment Mapping. It follows with a quick and practical description of the mathematical tools and techniques which OER relies on, and then a precise description of the algorithm.

In detail, chapters are organized in the following manner:

- **Chapter 2** describes the current state of the art about Environment Mapping, including how EM images are used, what kind of projections are employed to store environments in 2D flat images and how EMs are then effectively exploited in shading.

- **Chapter 3** is intended to give to the reader an essential background

on high dynamic range images (HDRI), K-Means data clustering (employed in source localization), classic Lambert Lighting Model, Blinn Reflection Model, Spherical Harmonics (SH) and Shadow Mapping.

- **Chapter 4** is a detailed explanation of the algorithm, in all its phases, including its pseudo-code and some showcase result with simple famous 3D models (e.g. Stanford Bunny) and environments (e.g. Grace Cathedral).
- **Chapter 5** discusses about memory usage, achievable fps and other test's results. Furthermore it includes comparisons between OER and common IBL of some very well known and very influent modern graphic engines.
- **Chapter 6**, as a final chapter, draws conclusions about the project and proposes possible optimizations and future works.

In the following text, wherever coordinates are treated, unless clearly specified, the coordinates system of reference is always the OpenGL's one, as showed in figure 1.2:

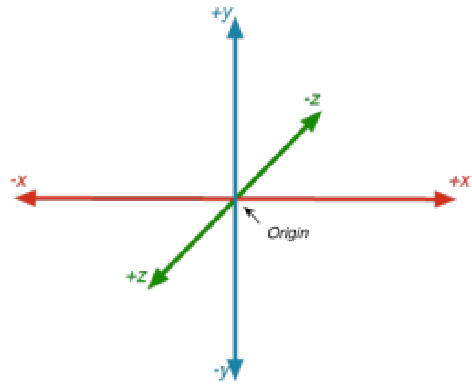


Figure 1.2: *OpengGL's coordinates system*

Chapter 2

State of the Art

2.1 Environment Mapping for diffuse component

Environment Mapping is an efficient image based lighting technique for approximating the appearance of a surface by means of a precomputed texture image. Environment Mapping assumes that an object's environment (that is, everything surrounding it) is infinitely distant from the object. The reason for the assumption is that Environment Maps (EMs) are accessed solely based on a 3D direction. Environment mapping has no allowance for variations in position to affect the appearance of surfaces. If everything in the environment is sufficiently far away from the surface, then this assumption is approximately true.

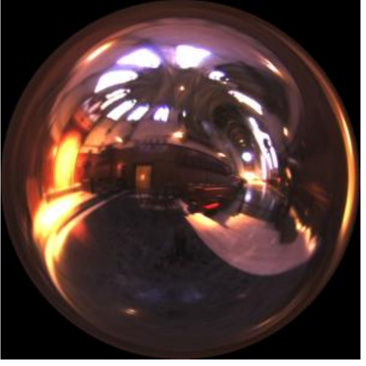
Light reflected by a surface is assumed to have a diffuse component and a specular component. The diffuse component represents light scattered equally in all directions; the specular component represents light reflected at or near the mirror direction. At the very beginning of Environment Mapping, it was used (Blinn and Newell [8]) to efficiently find reflections (specular components) of distant objects. Miller and Hoffman[3], and Greene [6] prefiltered environment maps, precomputing separate EMs for the specular and diffuse components of the BRDF (Bidirectional Reflectance Distribution Function, a function that models how a surface responds to light). OER stems from this latter approach for what concerns the diffuse component. No ambient term is required, since the diffuse illumination component accounts for all illumination from the environment.

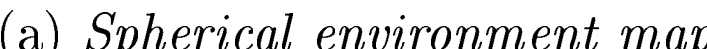
2.1.1 Prefiltering an EM

Prefiltering is generally an offline, computationally expensive process. After prefiltering, rendering can usually be performed at interactive rates with graphics hardware using texture-mapping. The idea comes from noticing that all surfaces with normal direction n should have the same value returned from the EM lookup. Furthermore, this value is dependent on just the environment and the surface normal. What this means is that one can compute the sum for a lot (e.g. 256×256) of normals in an offline process executed once per environment map, and store the result in a second environment map, indexed by the surface normal. This second environment map is known as the Diffuse Irradiance Environment Map (DIEM),or just the Diffuse Environment Map, and it allows to perform objects' illumination with arbitrarily complex lighting environments with a single texture lookup. In details, assuming that a EM has K texels, and there exists a mapping between the texel coordinates (s_i, t_i), $i \in \{0, .., K\}$, and a world space direction d_i, and similarly assuiming that the target DIEM is composed of N texels and there exists a mapping between (s_j, t_j), $j \in \{0, .., N\}$, and n_j, the value to be stored in the j-th DIEM's texel can be computed as:

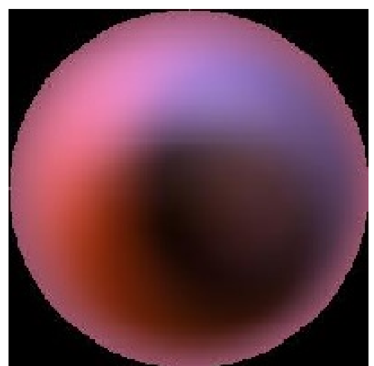
$$\sum_{i=1}^{K} max(0, dot(d_i, n_j))EM(i) \tag{2.1}$$

being $EM(i)$ the value stored in the EM in the i-th texel. Figure 2.1 shows an example of a EM and its DIEM.





(a) *Spherical environment map*

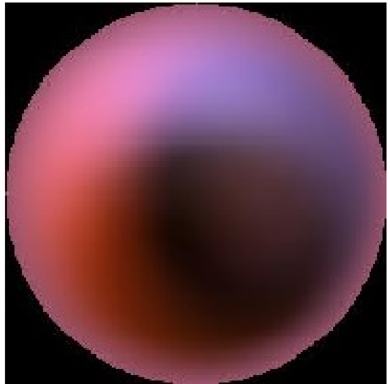


(b) *Spherical Irradiance map*

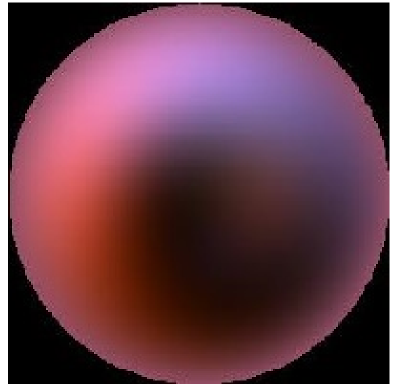
Figure 2.1: *Comparison between Grace's Cathedral spherical EM and DIEM*

2.1.2 The Spherical Harmonics Approach

Since a DIEM has little high-frequency content, it can be computed and stored at low resolution and accessed with bilinear interpolation. Thus, prefiltering the EM in this manner reduces the problem of finding the diffuse illumination at a surface point to a table lookup. The same reasoning, brought developers in trying a new approach: projecting the EM on basis of an appropriately chosen frequency-based space and employing in rendering a low-frequency reconstruction of it. It is the case of Ramamoorthy and Hanrahan[15], who adopted Spherical Harmonics (SH) basis.They showed that one needs to compute and use only the first 9 SH basis, corresponding to the lowest-frequency modes of the illumination, in order to achieve an average approximation error of only 3% with respect to the DIEM method. In other words, the irradiance is insensitive to high frequencies in the lighting, and is well approximated using only 9 parameters. In fact, they show that the irradiance can be procedurally represented simply as a quadratic polynomial in the cartesian components of the surface normal. Furthermore they showed that, once one computes the projection of a lambertian cosine lobe, $max(0, \cos\theta)$, thanks to SH properties, the final color can be computed with a simple matrix vector multiplication and a dot product. Figure 2.2 shows the same DIEM as before, compared with the one reconstructed computing for every direction stored in the DIEM the corresponding value obtained with the SH method.



(a) *Regular Spherical Irradiance map*



(b) *SH Spherical Irradiance map*

Figure 2.2: *Comparison between Grace's Cathedral regular spherical EM the SH generated one*

2.2 Common Projections

Several ways of mapping texel coordinates to 3D directions have been used. The first technique was sphere mapping, in which a single texture contains the image of the surroundings as reflected on a mirror ball. It has been almost entirely surpassed by cube mapping, in which the environment is projected onto the six faces of a cube and stored as six square textures or unfolded into six square regions of a single texture. Other projections that have some superior mathematical or computational properties include the HEALPix mapping and the equirectangular mapping.

2.2.1 Sphere Mapping

Sphere mapping considers the environment to be an infinitely far-away spherical wall. This environment is stored as a texture depicting what a mirrored sphere would look like if it were placed into the environment, using an orthographic projection. This texture contains reflective data for the entire environment, except for the spot directly behind the sphere.

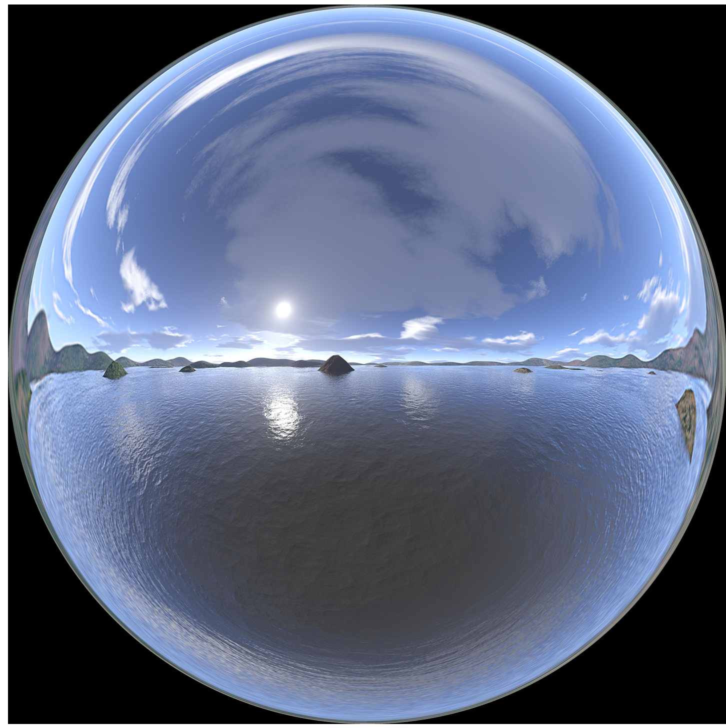


Figure 2.3: *Sphere mapping example*

In the simplest case for generating texture coordinates, suppose:

- The texture coordinate of the center of the map is (0,0), and the sphere's image has radius 1.
- We are rendering an image in the same exact situation as the sphere, but the sphere has been replaced with a object.
- The image being created is orthographic, or the viewer is infinitely far away, so that the view direction does not change as one moves across the image.

At texture coordinate (x, y), note that the depicted location on the sphere is(x, y, z) (where z is $\sqrt{1 - x^2 - y^2}$), and the normal at that location is also $< x, y, z >$. However, we are given the reverse task (a normal for which we need to produce a texture map coordinate). So the texture coordinate corresponding to normal $< x, y, z >$ is (x, y).

2.2.2 Cube Mapping

All recent GPUs support a type of texture known as a cube map. A cube map consists of not one, but six square texture images that fit together like the faces of a cube. Together, these six images form an omnidirectional image that are used to encode environment maps. Figure 2.4 shows an example of a cube map that captures an environment consisting of a cloudy sky and foggy mountainous terrain. A 2D texture maps a 2D texture coordinate set to a color in a single texture image. In contrast, you access a cube map texture with a three-component texture coordinate set that represents a 3D direction vector, usually a fragment's normal. The (s, t, r) texture coordinates are treated as a direction vector (rx, ry, rz) emanating from the center of a cube.The target column in the table below explains how the major axis direction maps to the 2D image of a particular

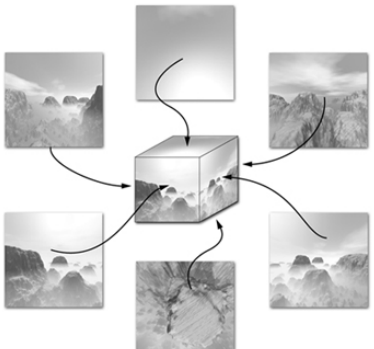


Figure 2.4: *Cubemap faces explanation*

cube map, in a right handed coordinates system, such as OpenGL's world space.

Table 2.1: OpenGL Cubemap layout

Major Axis Direction	**Target Face**	**sc**	**tc**	**ma**
+rx	Right	-rz	-ry	rx
-rx	Left	+rz	-ry	rx
+ry	Up	+rx	+rz	ry
-ry	Down	+rx	-rz	ry
+rz	Back	+rx	-ry	rz
-rz	Front	-rx	-ry	rz

Using the sc, tc, and ma determined by the major axis direction as specified in the table above 2.1, an updated (s, t) is calculated as follows:

$$s = \frac{\frac{sc}{\|ma\|} + 1}{2} \tag{2.2}$$

$$t = \frac{\frac{tc}{\|ma\|} + 1}{2} \tag{2.3}$$

2.2.3 Equirectangular Mapping

The equirectangular projection (also called the equidistant cylindrical projection, geographic projection, or la carte parallélogrammatique projection, and which includes the special case of the plate carrée projection or geographic projection) is a simple map projection attributed to Marinus of Tyre, who Ptolemy claims invented the projection about AD 100. The projection maps meridians to vertical straight lines of constant spacing (for meridional intervals of constant spacing), and circles of latitude to horizontal straight lines of constant spacing (for constant intervals of parallels). The projection is neither equal area (each texel doesn't correspnd to the same amount of area) nor conformal (circular areas are mostly distorted in ellipsis), but it's very popular in Computer Graphics because of the particularly simple relationship between the position of an image pixel and its corresponding 3D direction: the horizontal coordinate is the longitude and the

vertical coordinate is the latitude, so the standard parallel is taken as the equator. So, given the same coordinates system described in the cube mapping section, a 3D direction can be computed from image coordinates with the following formula:

$$\theta = \pi * (\frac{imageY + 0.5}{imageHeight}) \tag{2.4}$$

$$\phi = 2\pi * (1 - (\frac{imageX + 0.5}{imageWidth})) \tag{2.5}$$

$$y = \cos\theta \tag{2.6}$$

$$x = \sin\theta\cos\phi \tag{2.7}$$

$$z = \sin\theta\sin\phi \tag{2.8}$$

where $(imageX, imageY)$, $imageX \in \{0, .., imageWidth\}$, $imageY \in \{0, .., imageHeight\}$. Figure 2.5 gives an example of a EM stored in equirectangular projection.

Figure 2.5: *Grace's Cathedral Equirectangular environment map*

Chapter 3

Technical Background

This chapter provides an essential description of the mathematical tools and notions necessary to understand the OER.

3.1 LDR and HDR

HDR is abbreviation for High Dynamic Range. For a scene, dynamic range refers to ratio between the brightest and darkest parts of the scene. The Dynamic Range of real-world scenes can be quite high - ratios of 100000:1 are common in the natural world. A HDR image stores pixel values that span the whole tonal range of real-world scenes. Dynamic range of JPEG format image won't exceed 255:1, so it is considered as LDR (Low Dynamic Range). Similarly, dynamic range of CRT monitor won't exceed 100:1. HDR images show the dynamic range of real world (natural dynamic range is generally considered to be 100000:1; the dynamic range human eyes can identify is around 100000:1.), which is much higher than that of standard display equipment and images shot with common camera. As a result, HDR image cannot be displayed with this equipment. Several algorithms of tone mapping have been developed to cope with this discrepancy between what can be displayed and the real world's color information. Tone mapping is the name of a family of techniques used to map one set of colors to another to approximate the appearance of high-dynamic-range images in a medium that has a more limited range. Figure 3.1 shows a comparison between an environment mapping with reflections performed adopting an LDR image and another one rendered in HDR to a texture buffer and then tone mapped,

it's evident how real world details are more remarked in the second one.

Figure 3.1: *Comparison of reflection mapping employing LDR vs. employing HDR*

So from broad sense, image with dynamic range of higher than 255:1 (8 bit per cooler channel) is regarded as HDR image. Images with this range are particularly useful in light sources location, since the actual Luminance value (in lumens/steradian/sq.meter) of a pixel can be computed from the following formula for the standard Radiance RGB primaries:

$$luminance = 179 * (0.265 * R + 0.670 * G + 0.065 * B) \tag{3.1}$$

The value of $179\, lumens/watt$ is the standard luminous efficacy of equalenergy white light.

3.2 Lambert Lighting Model

A very common lighting model in computer graphics, is the one following the Lambert's law. The apparent brightness of a Lambertian surface to an observer is the same regardless of the observer's angle of view. More technically, the surface's luminance is isotropic, and the luminous intensity obeys Lambert's cosine law. Lambertian reflectance is named after Johann Heinrich Lambert, who introduced the concept of perfect diffusion in his 1760 book Photometria. The diffuse color is calculated by taking the dot product of the surface's normal vector, n , and a normalized light-direction

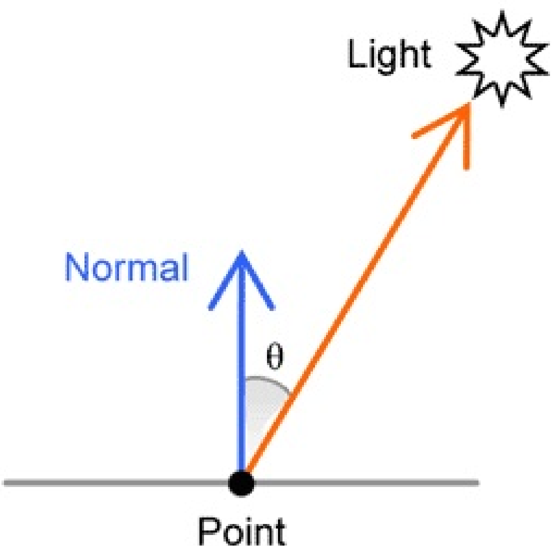


Figure 3.2: *Lambert's model visualization*

vector, l , pointing from the surface to the light source.

This number is clamped between 0 and 1, then multiplied by the diffuse costant of the material (namely, the material diffuse color) of the surface and the intensity of the light hitting the surface, to output a displayable color:

$$diffuseColor = clamp(l{\cdot}n, 0, 1) * materialDiffuseColor * lightIntensity) \quad (3.2)$$

where

$$l \cdot n = \|n\| * \|l\| * \cos\theta$$

where θ is the angle between the directions of the two vectors as shown in figure 3.2. The intensity will be the highest if the normal vector points in the same direction as the light vector ($\cos(0) = 1$, the surface will be perpendicular to the direction of the light), and the lowest if the normal vector is perpendicular to the light vector ($\cos\frac{\pi}{2}) = 0$, the surface runs parallel with the direction of the light).

3.3 Blinn Reflection Model

The Blinn Reflection Model is an empirical model of the local illumination of a surface, so common that is has been used in fixed function pipelines of OpenGL and DirectX. It considers the color of a fragment as a combination of diffuse reflection (like rough dim materials), specular reflection (like smooth shiny materials) and approximation of ambient lighting (lighting in places which aren't lightened by direct light rays). The diffuse component is computed following Lambert's law 3.1, while the ambient component is a constant term (remember 1.1?). Specular reflection emerges when direct light rays reflect from the surface in direction to the camera. Specular reflection creates highlights on the surface. A highlight is a bright spot, which is visible only when light from a source reflects directly to the camera. To calculate specular reflection in the Blinn reflection model, one has first to find the half vector H, which is the normalized average between direction to the camera V and direction to the light L, as shown in figure 3.3.

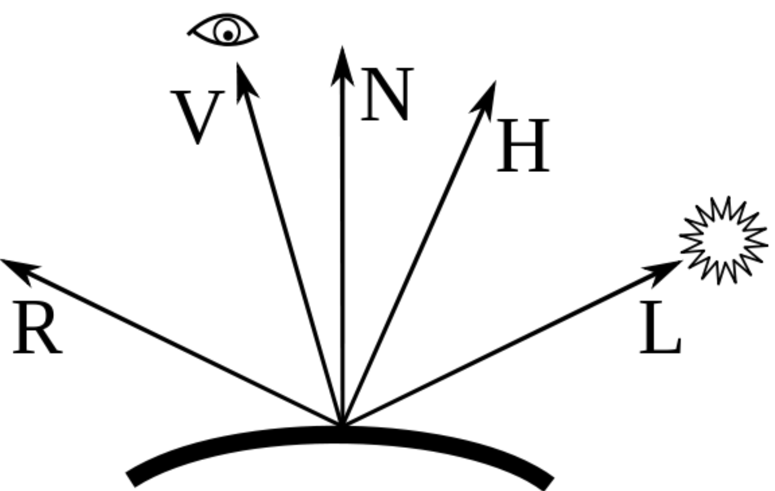


Figure 3.3: *Blinn's model visualization*

The intensity of specular reflection is the cosine of the angle between the fragment's normal N and the half vector H, raised to the power of α which is the "shininess" of the object. When α is large, in the case of a nearly mirror-like reflection, the specular highlight will be small, because

any viewpoint not aligned with the reflection will have a cosine less than one which rapidly approaches zero when raised to a high power. The specular component of the color of the fragment is finally computed multiplying the computed intensity by the specular constant of the material (namely, the material specular color). In formulas:

$$specularColor = materialSpecularColor * clamp((H \cdot N, 0, 1)^{\alpha}) \quad (3.3)$$

where

$$H = \frac{L + V}{\|L + V\|}$$

3.4 K-Means Clustering

K-means is one of the simplest unsupervised learning algorithms that solves the well known clustering problem. The procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters) fixed a priori. The most common algorithm uses an iterative refinement technique. It starts defining an initial set of k centroids $m_1^{(1)}, .., m_k^{(1)}$. Centroids represents the mean of each cluster, so the first ones shoud be placed in a cunning way because of different location causes different result. The better choice is to place them as much as possible far away from each other. The algorithm then proceeds by alternating between two steps:

1. **Assignment Step:** Assign each observation to the cluster whose mean yields the least value of a defined within-cluster distance (or error) measure. For example, assuming a squared Euclidean distance, this means intuitively choosing the "nearest" mean. Formally, given a distance measure $D(x, m)$ between two observations, a generic point x is assigned to the cluster i such that $D(x, m_i) \leq D(x, m_j) \forall j, 1 \leq j \leq k$. At each round, every observation is assigned to exactly one cluster.

2. **Update Step:** Calculate the new means to be the centroids of the observations in the new clusters.

The algorithm has converged when the assignments no longer change or (common in a lot of software packages) after a user specified number of iterations.

3.5 Spherical Harmonics

3.5.1 Introduction to Spherical Harmonics

Spherical harmonics (SH) are a frequency-space basis for representing functions defined over the sphere. They are the spherical analogue of the 1D Fourier series and the 2D cosine and sine functions. SH arise in many physical problems ranging from the computation of atomic electron configurations to the representation of gravitational and magnetic fields of planetary bodies. They also appear in the solutions of the Schrödinger equation in spherical coordinates.Spherical harmonics also have direct applicability in computer graphics. Light transport involves many quantities defined over the spherical and hemispherical domains, making spherical harmonics a natural basis for representing these functions. OpenGL's coordinates system is dropped in this section and a standard right-handed coordinate system is adopted, as showed in figure 3.4.

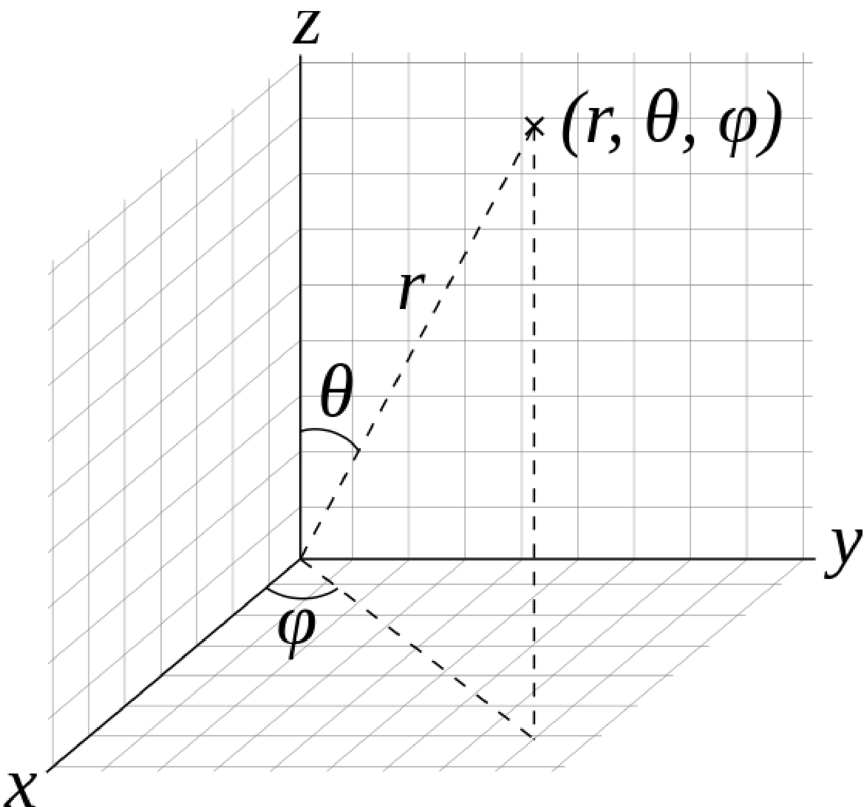


Figure 3.4: *Standard coordinates system*

3.5.2 Definition

A harmonic is a function that satisfies Laplace's equation:

$$\nabla^2 f = 0$$

As their name suggests, the spherical harmonics are an infinite set of harmonic functions defined on the sphere. They arise from solving the angular portion of Laplace's equation in spherical coordinates using separation of variables. The spherical harmonic basis functions derived in this fashion take on complex values, but a complementary, strictly real-valued, set of harmonics can also be defined. Since in computer graphics we typically only encounter real-valued functions, this discussion is restricted only to the real-valued basis. If we represent a direction vector using the standard parametrization $s = (s_x, s_y, s_z) = (\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta)$ then the real spherical harmonic basis functions are defined as:

$$Y_l^m(\theta,\phi) = \begin{cases} \sqrt{2}K_l^m \cos m\phi P_l^m(\cos\theta) & if m > 0 \\ K_l^0 P_l^0(\cos\theta) & if m = 0 \\ \sqrt{2}K_l^m \sin -m\phi P_l^{-m}(\cos\theta) & if m < 0 \end{cases}$$

where K_l^m are the normalization constants:

$$K_l^m = \sqrt{\frac{2l+1}{4\pi}\frac{(l-\|m\|)!}{(l+\|m\|)!}}$$

and P_l^m are the associated Legendre polynomials. Appendix A reports a list of the first 16 real SH function. As you can see, a SH function is usually written as $Y_l^m(s)$, where s is a point on the sphere and the index l represents the "band". Each band is equivalent to polynomials of that degree (so zero is just a constant function, 1 is linear, etc.) and there are $2l+1$ functions in a given band, indexed by m which ranges between $-l$ and l. While spherical coordinates are convenient when computing integrals, they can also be represented using polynomials, as is commonly done when evaluating them. One standard way to display SH is to distort a unit sphere, by scaling each point radialy by the absolute value of the function and coloring it based on the sign (green for positive, red for negative.) Below there are images of the

first three bands using this technique.

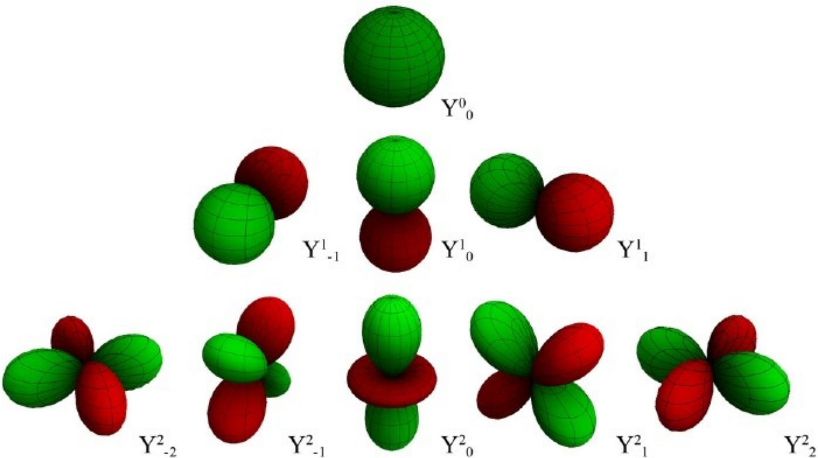


Figure 3.5: Representation of the first 3 bands of real spherical harmonics

3.5.3 Properties

Spherical Harmonics define an orthonormal basis over the sphere S. This means that the inner product of any two distinct basis functions is zero:

$$\int_S Y_{l_1}^{m_1}(s) Y_{l2}^{m_2}(s) = \begin{cases} 0 & if l_1 \neq l_2 \wedge m_1 \neq m_2 \\ 1 & if l_1 = l_2 \wedge m_1 = m_2 \end{cases} \tag{3.4}$$

Moreover, orthonormality ensures that the least squares projection of a scalar function f defined over S is done by simply integrating the function to project, $f(s)$, against the basis functions over S:

$$f_l^m = \int_S f(s) y_l^m(s) ds \tag{3.5}$$

These coefficients can be used to reconstruct an approximation of the function f:

$$\hat{f}(s) = \sum_{l=0}^{N-1} \sum_{m=-l}^{l} f_l^m Y_l^m(s) \tag{3.6}$$

Which is increasingly accurate as the number of bands N increases. This is called an "order N SH expansion" and uses all of the basis functions through degree $N-1$. Projections to Nth order generate N^2 coefficients. It is often

convenient to use a single index for both the projection coefficients and the basis function, via:

$$\hat{f}(s) = \sum_{i=0}^{N^2} f_i Y_i(s) \tag{3.7}$$

Where $i = l * (l+1) + m$. This formulation makes it clear that evaluating at direction s of the approximate function is simply a dot product between the N^2 coefficient vector composed of the f_i and the vector of evaluated basis functions $Y_i(s)$. That said, given any two functions $a(s)$ and $b(s)$ defined over the sphere S, the integral of their product can be written as:

$$\int_S a(s)b(s)ds \approx \int_S a\hat{(s)}b\hat{(s)}ds = \int_S \sum_{i=0}^{N^2} a_i Y_i(s) \sum_{j=0}^{N^2} b_j Y_j(s)ds$$

For orthonormality, recalling equation 3.4, the products between basis with different index can be canceled, leading to:

$$\int_S a(s)b(s)ds \approx \sum_{i=0}^{N^2} a_i b_i \tag{3.8}$$

which is a very handy equation considering the gap in computation time between an integral and a dot product.

The functions where $l = \|m\|$ are called Sectorial Harmonics and the zeros define regions like apple slices. The functions identified by $l = 0$ are called Zonal Harmonics (ZH) and have rotational symmetry around the Z axis and the zeros (locations where the function is zero) are contours on the sphere parallel to the XY plane (circular symmetry). Since the SH basis form effectively a Fourier domain basis defined over the sphere, it inherits a similar frequency space convolution property. If $h(z)$ is a circularly symmetric kernel, all its projection coefficients h_l^m such that $l \neq m$ are 0 and the convolution $h \star f$, with a generic function $f(s)$ over S, is equivalent to a weighted multiplication in the SH domain:

$$(h \star f)_l^m = \sqrt{\frac{4\pi}{2l+1}} h_l^0 f_l^m$$

The convolution property allows for efficient computation of prefiltered environment maps and irradiance environment maps since the lambertian co-

sine lobe (discussed in 3.2) is a kernel function symmetric with respect to the standard z-axis. SH basis are closed under rotation. Given a function $g(s)$, which represents a function $f(s)$ rotated by a rotation matrix Q, so $g(s) = f(Q(s))$ the projection of g is identical to rotating f and re-projecting it. This rotational invariance is similar to the translational invariance in the Fourier transform. This means that, for example, lighting will be stable under rotations, so there won't be any aliasing artifacts or "wobbling" of the light sources. Rotation can be performed directly on the coefficients by mean of a rotation matrix. SH rotation matrices are in a block structure, where each band is rotationally independent and has a dense $(2l + 1) \times (2l + 1)$ sub-matrix, nonetheless when the order of the expansion rises, their computation gets very cumbersome. Rotation of ZH is simpler than general SH, it can be done with a diagonal matrix and only requires evaluating the SH basis functions in the new direction $\hat{s}$. Given the ZH coefficients of a function (only the $m = 0$ terms from an SH projection) z_l they can be rotated to a new direction $\hat{s}$, obtaining new $\hat{z}_l^m$ coefficients (rotated ZH are not ensured to be ZH anymore, in fact usually they are not) using this equation:

$$\hat{z}_l^m = \sqrt{\frac{4\pi}{2l+1}} z_l Y_l^m(\hat{s}) \tag{3.9}$$

Note the similarity of this equation to the convolution. In effect, rotation of a circularly symmetric function is the same as convolving a kernel with a delta function at the desired rotation axis. This comes very useful when dealing with products with a rotated kernel.

3.5.4 SH Lighting

The color intensity of a pixel can be computed by scaling the irradiance E, incoming on a fragment with normal n, by the surface albedo ρ, to find the radiosity B, which corresponds directly to the image intensity:

$$B(n) = \rho E(n) \tag{3.10}$$

The irradiance E hitting a surface is a function of the its normal n only and is given by an integral over the upper hemisphere $\Omega(n)$:

$$E(n) = \int_{\Omega(n)} L(s)(n \cdot s)ds \tag{3.11}$$

where L(s) is the radiance incoming from direction s, which in Environment Mapping is a EM look-up in the texel coordinates corresponding to s. Noting that, being θ the angle between the normal n and the incoming light direction as in Lambert's model, $A(s) = (n \cdot s) = max(\cos\theta, 0)$ is a function circularly symmetric with respect to n, assuming n as the z-axis, one can project $A(s)$ in SH space, obtaining a reconstructed function of the form:

$$\hat{A}(s) = \sum_{l=0}^{N-1} A_l Y_l^0(s)$$

Projecting $L(s)$ aswell and approximating it with $\hat{L(s)} = \sum_{l=0}^{N-1}\sum_{m=-l}^{l} L_l^m Y_l^m(s)$, the irradiance formula for a fragment with a normal aligned with the z-axis can be expressed by:

$$\hat{E}((0,0,1)) = \int_{\Omega(0,0,1)} (\sum_{l=0}^{N-1}\sum_{m=-l}^{l} L_l^m Y_l^m(s))(\sum_{l=0}^{N-1} A_l Y_l^0(s))ds$$

Thanks to 3.8 the formula can be simplified in:

$$\hat{E}((0,0,1)) = \sum_{l=0}^{N-1}\sum_{m=-l}^{l} L_l^m A_l \tag{3.12}$$

This formula looks good but it is rather useless if it can't be used for rendering surfaces with normals other than the z-axis. To compute the correct value of irradiance for a generic n one should either rotate $L(s)$ in direction $\hat{n}$, which is n reflected with respect to the standard z-axis, or more easily rotate the cosine lobe with equation 3.9 to point in direction n. A_l coefficients can be rotated with the following:

$$\hat{A}_l(n) = \sqrt{\frac{4\pi}{2l+1}} A_l Y_l^0(n)$$

We can rewrite 3.12 to accept a generic n instead of (0,0,1) in this fashion:

$$\hat{E}(n) = \sum_{l=0}^{N-1} \sum_{m=-l}^{l} L_l^m \hat{A}_l(n) \tag{3.13}$$

The final color intensity is then computed as:

$$B(n) = \rho \sum_{l=0}^{N-1} \sum_{m=-l}^{l} L_l^m \hat{A}_l(n) \tag{3.14}$$

3.5.4.1 A practical approach

Ramamoorthi and Hanrahan ([15]) showed that an order 3 SH expansion is enough to achieve a good approximation of a DIEM. Since they only use $l \leq 2$ bands, the irradiance formula is simply a quadratic polynomial of the coordinates of the surface normal. Hence with $n^T = (x, y, z, 1)$, one can write:

$$E(n) = n^T M n \tag{3.15}$$

where M is a symmetric 4x4 matrix. Each color has an independent matrix M. Equation 3.15 is particularly useful for rendering, since only requires a matrix-vector multiplication and a dot-product to compute E. The matrix M is obtained by expanding equation 3.13:

$$M = \begin{pmatrix} c_1 L_2^2 & c_1 L_2^{-2} & c_1 L_2^1 & c_2 L_1^1 \\ c_1 L_2^{-2} & -c_1 L_2^2 & c_1 L_2^{-1} & c_2 L_1^{-1} \\ c_1 L_2^1 & c_1 L_2^{-1} & c_3 L_2^0 & c_2 L_1^0 \\ c_2 L_1^1 & c_2 L_1^{-1} & c_2 L_1^0 & c_4 L_0^0 - c_5 L_2^0 \end{pmatrix}$$

$$c_1 = 0.429043 \, c_2 = 0.511664 \, c_3 = 0.743125 \, c_4 = 0.886227 \, c_5 = 0.247708$$

The entries of M depend on the 9 lighting coefficientsL_l^m and the expressions for the SH. The constants come from the numerical values of the coefficients A_l of the cosine lobe (n.b. A_l and not $\hat{A}_l$!). Ramamoorthi and Hanrahan

showed that A_l coefficients are in the form:

$$\begin{array}{ll} l = 1 & A_l = \frac{2\pi}{3} \\ l > 1\,\textbf{odd} & A_l = 0 \\ l > 1\,\textbf{even} & A_l = (-1)^{\frac{l}{2}-1}\frac{2\pi}{(l+2)(l-1)}\left[\frac{l!}{2^l(\frac{l}{2}!)^2}\right] \end{array}$$

So the first seven terms are:

$$A_0 = 3.141593\,A_1 = 2.094395\,A_2 = 0.785398$$

$$A_3 = 0\,A_4 = -0.130900\,A_5 = 0\,A_6 = 0.049087$$

3.6 Shadow Mapping

The basic Shadow Mapping algorithm consists in two passes. First, the scene is rendered from the point of view of the light. Only the depth (distance from the light) of each fragment is computed and stored in a texture called Shadow Map (SM). Next, the scene is rendered as usual, but with an extra test to see it the current fragment is in the shadow. The "being in the shadow" test is actually quite simple. If the current sample is further from the light than the SM at the same point, this means that the scene contains an object that is closer to the light. In other words, the current fragment is in the shadow.

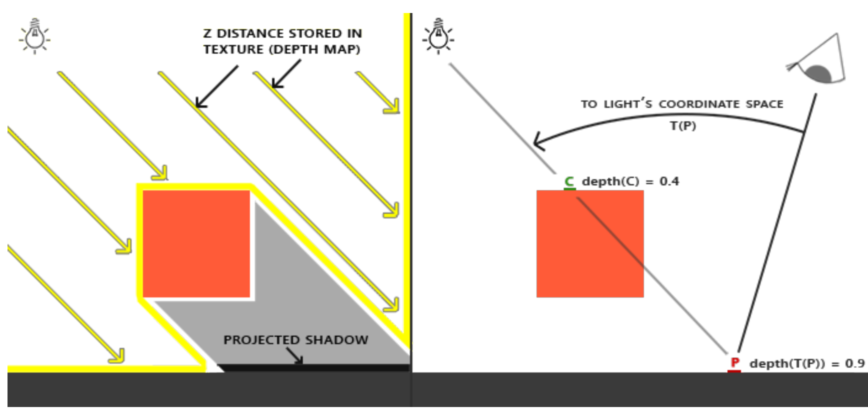


Figure 3.6: *Shadow mapping visual explanation*

Despite it's conceptual simplicity, Shadow Mapping carries a lot of tricky

pitfalls. First of all Shadow Acne.

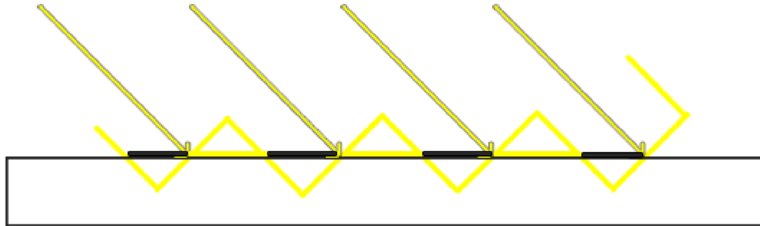


Figure 3.7: *Multiple fragments mapping on the same shadow map texel*

Because the SM is limited by resolution, multiple fragments can sample the same value from the depth map when they're relatively far away from the light source. The image shows the floor where each tilted panel represents a single texel of the depth map. The problem is that several fragments sample the same depth sample. While this is generally okay it becomes an issue when the light source looks at an angle towards the surface as in that case the depth map is also rendered from an angle. Several fragments then access the same tilted depth texel while some are above and some below the floor; we get a shadow discrepancy. Because of this some fragments are deemed in shadow and some are not, giving the striped pattern from the figure 3.8 that is called Shadow Acne.

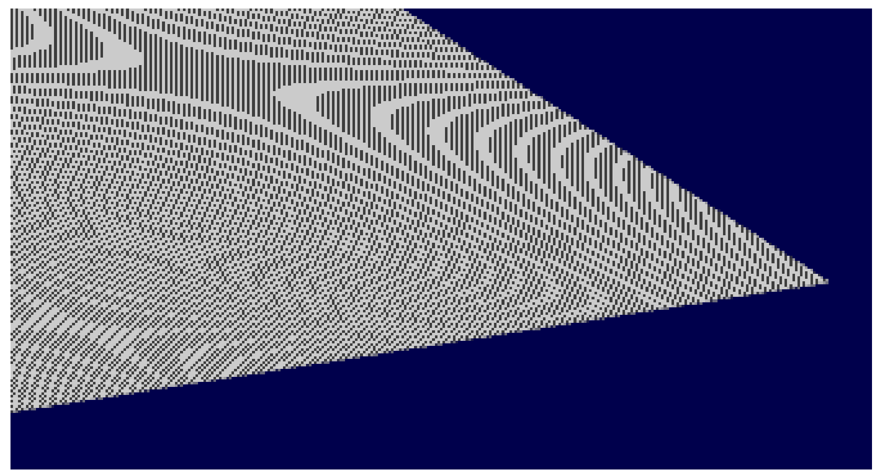


Figure 3.8: *Shadow Acne artifact*

This issue can be solved with a small little hack called a Shadow Bias

that consists in simply offsetting the depth of the surface (or the SM) by a small bias amount such that fragments are not incorrectly considered below the surface. A solid approach is to change the amount of bias based on the surface angle towards the light, as in this common formula:

$$bias = max(0.05*(1 - (normal, \cdot lightDir)), 0.005)$$

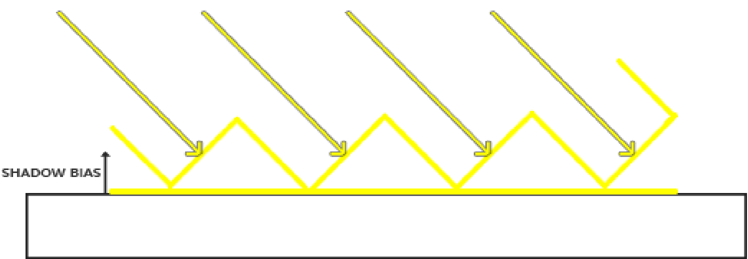


Figure 3.9: *Bias correction effect*

A disadvantage of using a shadow bias is that you're applying an offset to the actual depth of objects. As a result the bias might become large enough to see a visible offset of shadows compared to the actual object locations as you can see in figure 3.10. This shadow artifact is called Peter Panning since objects seem to slightly hover above the surface.

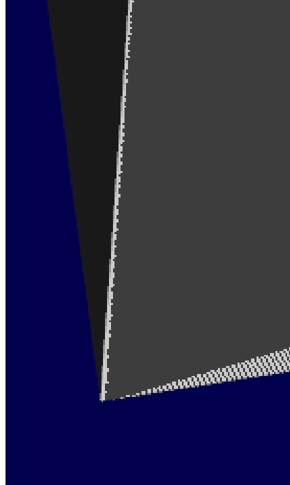


Figure 3.10: *Peter Panning artifcat*

A trick to solve most of the peter panning issue is using Front Face

Culling (FFC) when rendering the depth map. By default, graphic libraries such as OpenGL and Direct3D, culls back faces when draw commands are issued. By enabling FFC we're switching that order around. Because only depth values are needed for the SM it shouldn't matter for solid objects whether we take the depth of their front faces or their back faces. Using their back face depths doesn't give wrong results as it doesn't matter if we have shadows inside objects; figure 3.11 gives a visual explanation.

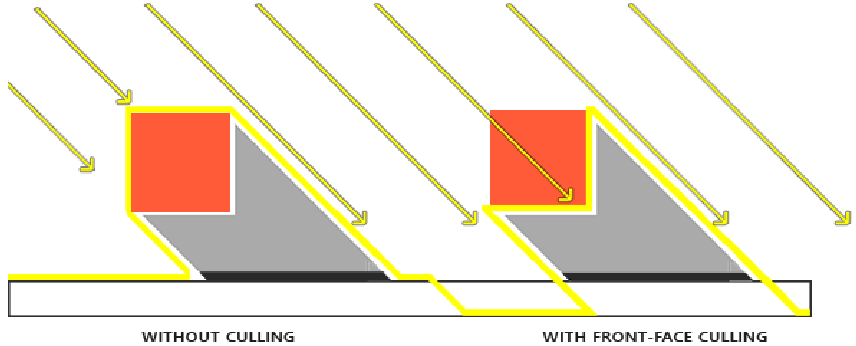


Figure 3.11: *Front face culling visual explanation*

Bias and FFC can be used together for better results, as OER does.

Chapter 4

Occluded Environemnt Maps Rendering

This chapter is intended to offer an exhaustive description of the technique. Occluded Environment Maps Rendering (OER) consists in a preprocessing phase and the rendering cycle itself. Moreover, the preprocessing phase is divided in two: the location step and the projection step. For what concerns the rendering, it's assumed an OpenGL-like or Direct3D-like graphic pipeline, coarsely described in the following. For a better understanding, see Appendix B. The input to this pipeline are the object's vertices. Every vertex is fed into a small program called Vertex Shader, which does some preliminary processing like basic transformations (translation, scaling, rotation) and can output attributes' value. At this point the vertices undergo clipping (vertices outside the view frustrum are dropped) and rasterization (decomposition in discrete points of the lines between vertices) resulting in fragments. A second small program called Fragment or Pixel Shader can then be run on each fragment before the final pixel values are output to a frame buffer for display or for being used as a texture in successive iterations of the pipeline. This second shader usually takes in input the values of the attributes computed per vertex by the first shader, linearly interpolated between any two vertices with respect to the fragment's position on the line connecting them.

4.1 Light Sources Location

An EM is effectively an image of size$(imageWidth, imageHeight)$, that is a set of $imageWidth \times imageHeight$ pixels holding color values. The EM used by OER is an Equirectangular High Dynamic Range Image. If we define a mapping between a generic index i and the image coordinates (x_{p_i}, y_{p_i}) on the EM of a pixel as:

$$x_{p_i} = \left\lfloor \frac{i}{imageWidth} \right\rfloor$$

$$y_{p_i} = i \,\mathbf{mod}\, imageWidth$$

a pixel can be represented by $p_i = (p_i^r, p_i^g, p_i^b)$, where p_i^r, p_i^g, p_i^b are, in turn, the red, green and blue value of the pixel. The objective of this phase is to locate a user-specified number *nLightSources* of light sources. A generic light source L_k, where $k \in \{0, \ldots, nLightSources\}$, is embodied by a subset of pixels of the EM, formally $L_k \subset EM$. Also, OER associates to every light source a main direction and loss factor, that represents the percentage of luminance loss when the pixel of a given light source are obscured.

Before any operation is performed, the EM is prefiltered with a gaussian blur filter, to make sources individuation easier. Then the first operation performed is the creation of a dataset D of pixel observations. This dataset is formed by one observation o_i for each pixel $p_i \in EM$, in the form $o_i = (x_{o_i}, y_{o_i}, l_{o_i})$, where $x_{o_i} = x_{p_i}$, $y_{o_i} = y_{p_i}$ and $l_{o_i} = 179 * (0.265 * p_i^r + 0.670 * p_i^g + 0.065 * p_i^b)$ is the pixel luminance, as explained in section 3.1. Two values are extrapolated from this dataset; the first one is *lightTreshold*, the percentile value corresponding to *lightPercentile*, a sensitivity parameter empirically set to 99%, the second one is *haloTreshold*, the percentile value correspondig to *HaloPercentile*, empirically set to 96%. Two new dataset are then generated from D: D^l composed of all those observation with luminance greater or equal than *lightTreshold* and D^h formed by those observations with luminance greater or equal than *haloTreshold*. D^l is considered to contain only pixels imputable to a light source or some of its reflections, but doesn't store any information of which pixel belongs to which light source. In order to achieve that information, D^l is processed by a K-Means algorithm employing a 2D Euclidean Distance accounting only for the pixels' x and y coordinates (thus ignoring the lu-

minance attribute). The dataset is divided in *nLightClusters* clusters ($nLightClusters \geq nLightSources$), where *nLightClusters* is a parameter that may differ from EM to EM, but it's empiracally defaulted to a least 10 (if the number of light sources allows it).

Given a cluster C_q and its mean $m_q = (x_{m_q}, y_{m_q}, l_{m_q},\ q \in \{0, \ldots, nLightClusters\}$, we define its magnitude value as the luminance intensity l_{m_q} of its mean . Clusters resulting from the K-Mean application, are sorted by descending magnitude and the cluster with the greatest magnitude is chosen and associated to a light source. The coordinates pair of its mean is interpreted as the source's main direction, so it is converted in 3D coordinates (as showed in 2.2.3) and stored for successive computations. In addition, a parameter called *luminanceLoss* is computed for the source as the ratio between the sum of the luminance components of every observation in the cluster and the total sum of the luminance components over the EM:

$$luminanceLoss_{C_q} = \frac{\sum_{o_i \in C_q} l_{o_i}}{\sum_{p_j \in EM} l_{p_j}}$$

The final set of pixels belonging to this first light source L_0, are obtained scanning D^h for those pixels that have as closest centroid, the one associated to L_0 and are "close enough" to it. To measure if a pixel is close enough to a light centroid, OER computes naively a light radius with the x and y components' standard deviations:

$$radius = 3 * max(\sigma_x, \sigma_y)$$

and ignores every pixel further than *radius*. In order to locate more light sources, every pixel belonging to the just individuated one are deleted from D^l and the same process is repeated decreasing *nLightClusters* of 1. This creates a loop that ends when the number of individuated sources is equal to *nLightSources*. The last operation of this phase is the creation of a collection of Occluded Environment Maps (OEM). Consider the set of all light sources $\mathcal{L} = \{L_0, \ldots, L_{nLightSources-1}\}$, and its powerset $\mathbb{P}(\mathcal{L}) = \{\emptyset, \{L_0\}, \ldots \{L_0, L_1\}, \ldots\}$, then exists a mapping between each set of $\mathbb{P}(\mathcal{L})$ and one, and only one, OEM. In particular, for each set $S \in \mathbb{P}(\mathcal{L})$, the corresponding OEM is a copy of the original EM, except for all the pixels

of the set $B = \{p_i \,|\, p_i \in L_j \wedge L_j \notin S\}$ which are blackened out.

To recap, the first phase of the algorithm is described in the following pseudocode:

Algorithm 1: *Phase One: Light source location and OEM generation*

```
Input: EM, nLightSources, lightPercentile, haloPercentile,
       nLightClusters
Output: OEMs, lightDirections, attenuationFactors
D ← {};
OEMs ← {};
ℒ ← {};
directions ← {};
losses ← {};
totalLuminance ← 0;
foreach p_i ∈ EM do
    (x_{p_i}, y_{p_i}) ← IndexToImageCoord(i);
    l_{p_i} ← 179 * (0.265 * p_i^r + 0.670 * p_i^g + 0.065 * p_i^b);
    totalLuminance ← totalLuminance + l_{p_i};
    D ← D ∪ {(x_{p_i}, y_{p_i}, l_{p_i})};
lightTreshold ← LuminancePercentile(lightPercentile);
haloTreshold ← LuminancePercentile(haloPercentile);
D^h ← Prune(D, haloTreshold);
D^l ← Prune(D, lightTreshold);
while less than nLightSources located do
    𝒞 ← K − Means(D^l, nLightClusters);
    Sort(𝒞);
    chosenCluster ← First(𝒞);
    directions ← directions ∪
      EquirectangularMapping(MeanCoordinates(chosenCluster)) ;
    losses ←
      losses ∪ SumOfLuminance(chosenCluster)/totalLuminace ;
    ℒ ← ℒ ∪ AssignPixels(C, chosenCluster, D^h);
    D^l ← Remove(D^l, chosenCluster);
    nLightClusters ← nLightClusters − 1;
foreach S ∈ ℙ(ℒ) do
    factor = 1;
    tempOEM = EM;
    foreach L|(∈ ℒ∧ ∉ S) do
        BlackenPixelsInImage(tempOEM, L);
        factor = factor − CorrespondingLoss(L, losses);
    OEMs ← OEMs ∪ tempOEM ;
    attenuationFactors ← attenuationFactors ∪ factor;
```

A collection of OEMs generated by this code with $nLightSources = 2$, $lightPercentile = 99\%$ and $nLightClusters = 10$ is showed in figure 4.1.

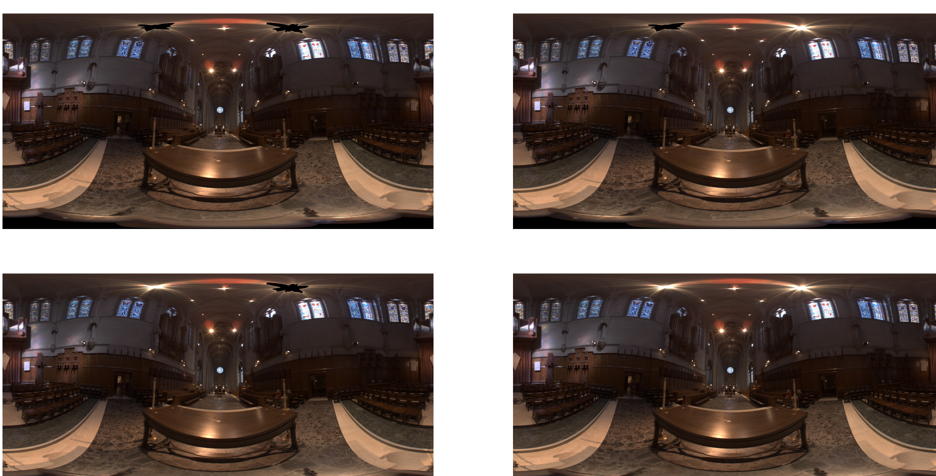


Figure 4.1: *Grace's Cathedrals OEMs for* $nLightSources = 2$

4.2 OEMs SH Projection

In this section OpenGL's coordinates system is dropped again and the standard right-handed coordinate system is adopted as previously done in 3.5.1. Before being processed in this phase, the EM is tonemapped and its channels are normalized for the range $[0, 1]$.

SH coefficients for every OEM are generated by integrating them against the spherical harmonic basis functions up to the $SHExpansionOrder - 1$ band (l). Each color channel is treated separately, so the coefficients can be thought of as RGB values. These coefficients are used to build a matrix for every color channel (3.5.4.1.

The general formula of a coefficient is given by:

$$E_l^m = \int_{\theta=0}^{\pi} \int_{\phi=0}^{2\pi} OEM(\theta, \phi) Y_l^m(\theta, \phi) \sin\theta d\phi\, d\theta$$

The integrals are simply sums of the pixels in the environment map EM, weighted by the functions Y_l^m and $\sin\theta$ to account for the stretching introduced by the equirectangular mapping (remember 2.2.3). The integrals can also be viewed as moments of the lighting, or as inner-products of the func-

tions $L(s)$ and $Y_l^m(s)$. In [15]] is showed that 9 coefficients are sufficients to approximate the environment map with an error less than the 1%. Since 9 coefficients are to be computed, the projection step takes $O(9\|OEM\|)$ time, where $\|OEM\|$ is the size (total number of pixels) of the environment map. By comparison, the standard method of computing an irradiance environment map texture takes $O(T\|OEM\|)$ time, where T is the number of texels in the irradiance environment map. Projecting will therefore be approximately $\frac{T}{9}$ times faster, even if a conventional irradiance environment map is computed at a very low resolution of 64×64, corresponding to $T = 4096$.

For an OEM stored in a equirectangular fashion, the procedure is described in the algorithm below:

Algorithm 2: *Phase Two: OEMs SH Projections*

Input: Tonemapped and normalized OEM, $SHExpansionOrder$
Output: $E_l^m coefficients$
$pixelArea \leftarrow \left(\frac{2\pi}{imageWidth}\right)\left(\frac{\pi}{imageHeight}\right)$;
foreach $(l, m) \in SHExpansion(SHExpansionOrder)$ **do**
  $E_l^m \leftarrow 0$;
foreach $p_i \in OEM$ **do**
  $(x_{p_i}, y_{p_i}) \leftarrow IndexToImageCoord(i)$;
  $(\theta, \phi) \leftarrow ImageCoordToSphericalCoord((x_{p_i}, y_{p_i}))$;
  foreach $(l, m) \in SHExpansion(SHExpansionOrder)$ **do**
    $E_l^m \leftarrow E_l^m +$
    $Y_l^m(\theta, \phi) * pixelArea * \sin\theta * GetColor(OEM, (x_{p_i}, y_{p_i}))$;

The code iterates for each pixel, over every basis in use, to find its contribution to the every SH coefficient. This way the OEM is scanned only once and the SH basis are evaluated once per pixel.

This operation yields to a set of coefficients per color channel for every OEM. From these sets matrix per color channel is built for every OEM. The matrices are finally multiplied by the OEM's attenuation factor, to be ready for rendering purposes.

4.3 Rendering cycle

This section assumes that the reader is familiar with terms like ”world space”, ”object space” and ”homogeneous coordinates”. If this is not the case,

refer to Appendix C for an overview of the common coordinates systems in Computer Graphics.

Once phase one and phase two of OER are over, the algorithm has successfully computed and stored a collection of $2^{nLightSources}$ sets of SH coefficients and $nLightSources$ 3D directions. The rendering cycle begins with a Shadow Mapping pass per light source, meaning that the entire scene is drawn from the point of view of the light source and a floating point value expressing a distance (or depth) is stored for every texel of the Shadow Map (SM). OER only consider directional lights, which are lights so far away that all the light rays can be considered parallel. As such, rendering the shadow map is done with an orthographic projection matrix. An orthographic matrix is just like a usual perspective projection matrix, except that no perspective is taken into account ; an object will look the same whether it's far or near the camera.

The Model-View-Projection (MVP) matrix used to render the scene from the light's point of view is formed by matrix composition of:

- **P**: an orthographic projection matrix which encompasses everything in a box of user-specified dimensions. Special attention should be payed not to exclude anything in the scene when dimensioning this box.
- **V**: a view matrix that rotates the world so that in camera space, the light direction is -Z (assuming OpenGL coordinates system).
- **M**: a model matrix, which is proper of every object and stores information about its position, scale and orientation.

Rendering a SM is usually more than twice as fast as the normal render, because only depth is written, instead of both the depth and the color. During these passes, front faces are culled instead of back faces, as seen in 3.6. The shaders employed are very simple, the vertex shader is a pass-through shader which simply compute the vertex's position in homogeneus coordinates C of the SM:

Algorithm 3: Shadow Map Pass: Vertex Shader

Input: MVP matrix of the light, vertexCoordinates in object space
Output: vertexShadowMapCoordinates
vertexShadoMapCoordinates = MVP * vertexCoordinates;

The fragment shader is just as simple, it just writes the depth of the fragment on the SM, encoded in the interpolated z coordinate of the fragment:

Algorithm 4: Shadow Map Pass: Fragment Shader

Input: fragmentShadowMapCoordinates = interpolated vertexShadowMapCoordinates

Output: fragmentShadowMapValue

fragmentShadowMapValue = fragmentShadowMapPosition.z;

The result of these Shadow Mapping passes are *nLightSources* SMs holding a floating point depth value in each texel.

In the subsequent rendering pass, the final color of each fragment is computed with SH lighting. In particular, the SMs are used to estabilish whether a fragment is lit or not by the corresponding light source. The set S, of light sources that light up a certain fragment, is part of the powerset $\mathbb{P}(\mathcal{L})$ of the set of light sources $\mathcal{L}$. That being, S can be mapped to an OEM, and so to set of matrices associated with the map. Hence the SMs testing is effectively used to choose a set matrices which is employed to compute the final color of the fragment (as seen in 3.5.4).

The vertex shader adopted in this pass, is more complicated than the one used for the Shadow Mapping pass. Its inputs are:

- **vPos**: the vertex position in object space.
- **vNorm**: the normal (a 3D direction), in object space, of the surface the vertex lies on.
- **vTextCoord**: the texture coordinates of the vertex.
- **MVP**: the model-view-projection matrix of the scene, used to transform a position in object space into homogeneous coordinates of the frame buffer.
- **N_M**: the normal model matrix, used to transform a normal from object space to world space.
- **SM_MVPs**: the array of model-view-projection matrices used to transform a position in object space into homogeneous coordinates of the corresponding SM, analogously as what is done in the Shadow Mapping pass.

The shader computes, for each vertex, the homogeneous coordinates over every SM besides computing the homogeneous coordinates of them over the frame buffer to display. Furthermore, it outputs a transformed 3D direction, corresponding to the vertex's normal in world's space coordinates. In details, the vertex shader outputs:

- **vScreenCoordinates** : the vertex's homogeneous coordinates over the frame buffer to display.
- **vNormalWorldSpace**: the surfaces normal in world's space coordinates.
- **vTextCoordinates**: the texture coordinates of the vertex "as is".
- **shadowCoordinates**: an array of homogeneous coordinates of the vertex over every SM.

After rasterization, these values are interpolated between any two connected vertices, for the generated fragments. The vertex shader's pseudo-code is reassumed below:

Algorithm 5: *Final Pass: Vertex Shader*

```
Input: vPos, vNorm, MVP, N_M, SM_MVPs, vTextCoordinates
Output: vScreenCoordinates, vNormalWorldSpace,
        shadowCoordinates, vTextCoordinates
vScreenCoordinates = MVP * vPos;
vNormalWorldSpace = Normalize(N_M * vNorm);
shadowCoordinates ← {};
foreach SM_MVP ∈ SM_MVPs do
  | sc ← SM_MVP * vPos;
  | shadowCoordinates ← shadowCoordinates ∪ {sc};
```

The fragment shader elaborates the final color of the screen pixel. The program is fed in with the interpolated vertex shader's output, the SMs, the SH matrices of every OEM and informations about the object's material, the light sources and the camera. To be precise, fragment shader's input are:

- **fNormalWorldSpace** : the interpolated 3D direction the fragment is orientated toward, in world's space coordinates.
- **fShadowCoordinates**: the interpolated fragment's shadow coordinates on each SM.

- **fTexCoordinates**: the interpolated texture coordinates.
- **SMs**: an array of depth (single valued floating point) textures.
- **materialTexture**: the texture of the material of the object.
- **SHMatrices**: a set triplets of matrices associated the OEMs.
- **sourceDirections**: the directions in world space of each light source.
- **eyeDirection**: the direction in world space pointing at the camera position.
- **shininess**: the shininess parameter of the material of the object.

The fragment shader's pseudo-code is the following:

Algorithm 6: *Final Pass: Fragment Shader*

Input: fNormalWorldSpace, fShadowCoordinates, fTexCoordinates, SMs, materialTexture, SHMatrices, sourceDirections, eyeDirection, shininess

Output: color

$specularComponent \leftarrow 0$;

$specularColor \leftarrow$
  $FetchSpecularColor(materialTexture, fTexCoordinates)$;

$diffuseColor \leftarrow$
  $FetchDiffuseColor(materialTexture, fTexCoordinates)$;

$lightSet \leftarrow \{\}$;

foreach $SM \in SMs$ **do**
  if *ShadowMapTest(SM, fShadowCoordinates, SMmapDirection(SM,sourceDirections))* **then**
    $specularComponent \leftarrow specularComponent +$
      $BlinnReflection(eyeDirection, shininess)$;
    $lightSet \leftarrow lightSet \cup SMmapLight(SM)$;

$MatR \leftarrow SelectMatR(SHMatrices, lightSet)$;

$MatG \leftarrow SelectMatG(SHMatrices, lightSet)$;

$MatB \leftarrow SelectMatB(SHMatrices, lightSet)$;

$n_T \leftarrow (fNormalWorldSpace, 1)$;

$shR \leftarrow n^T MatR\, n$;

$shG \leftarrow n^T MatG\, n$;

$shB \leftarrow n^T MatB\, n$;

$ambientLight \leftarrow (shR, shG, shB)$;

$color \leftarrow ambientLight * diffuseColor +$
  $specularComponent * specularColor$;

Chapter 5

Complexity and Results

5.1 Complexity Evaluation

5.1.1 Preprocessing Phase

The two phases of preprocessing are performed sequentially, so the time complexity of this first part of OER depends on the "slower" of the two. In order to locate the bottleneck of preprocessing, a computational analysis of the two phases is now given. For this purpose, an EM of width W, height H, $n = W \times H$ pixels and three color channels is adopted.

The main operations of the first phase are:

1. Gaussian Blur Filter.
2. Creation of the dataset D.
3. Creation of the datasets D^l and D^h.
4. K-Means loop.
5. Pixels Assignment.

Given a radius r, the Gaussian Blur has complexity $O(nr)$, but since r is usually a number lower than 5 (OER adopts a default radius of 4, but it's not a fixed parameter) we can simplify the complexity of 1 in $O(n)$. To generate the dataset D, the algorithm has to compute a luminance value for every pixel. This yields a $O(n)$ for operation 2. The creation of D^l and D^h both requires the calculation of a percentile that is usually implemented

with one sorting of the dataset plus a lookup per percentile. Assuming quick sort for sorting algorithm, this operation takes $O(n \log n)$ time. K-Means is repeated $nLightSources$ times over D^l which decreases in dimensionality after each iteration. That said, if we consider fixed the dimensionality of the set after each iteration, we can obtain a good lower bound estimation. Let $m = \|D^l\|$ then the complexity of 4 is $O((m^{2k+1} \log n) nLightSources)$. For every light source, pixels assignment is done by scanning D^h, which yields to $O(\|D^h\| nLightSources)$ complexity for 5. Since $n \gg m > \|D^h\|$ we can conclude that the complexity of the first phase is $O(n \log n)$.

The second phase is articulated in two operations:

1. OEM projections.
2. SH Matrices generation.

The complexity of the projection of a single EM depends on the number of coefficients to compute, 9 in this case. Since the number of OEMs is $nOEM = 2^{nLightSources}$ and there are three channels the complexity of 1 is $O(nOEM\, 27n)$. As seen in 3.5.4.1 each one of the 4×4 matrices are built performing 16 products, thus complexity of 3 is also constant $O(16)$. Hence the resulting complexity of phase two is $O(nOEM\, n)$ and since $n \gg nOEM$, the complexity of the first phase is predominant.

5.1.2 Rendering Cycle

The time complexity of the rendering cycle depends on the number of vertices to elaborate, the resolution of the employed SMs and the resolution of the displayed frame buffer. For the following discussion are assumed a scene of v vertices, SM of $s = sWidth \times sHeight$ texels and a frame buffer of $p = fWidth \times fHeight$ pixels.

The rendering comprises $nLightSources$ shadow mapping passes and one color pass.

Every shadow mapping pass performs a matrix vector multiplication $((4 \times 4) \times (4 \times 1))$ per vertex, and a memory write per texel of each SM. This translates into $O(16v)$ for the vertex shader and $O(s)$ for the fragment shader. Since both v and s are usually very big numbers (think about a 1024×1024 SM and a scene of 1 million vertices), the bottom line is that this first part of the rendering cycle takes $O(nLightSources(v + s))$.

The color pass is slightly more complicated than the shadow mapping one. The vertex shader has to compute the homogeneous coordinates of the vertex for every SM, plus its screen coordinates and its normal in world space. This results in $O((16nLightSources + 16 + 9)v) \rightarrow O(nLightSources\, v)$. The fragment shader performs $nLightSources$ textures lookup in the SMs and Blinn model's calculations, a matrix vector multiplication plus a dot product per color channel for the SH part, a texture lookup for the diffuse color and the specular color, and finally two multiplications and a sum to compute the final pixel color. SM lookups yields $O(p\, nLightSources)$ complexity. Blinn model's calculation can be considered $O(1)$, which leads to $O(p\, nLightSources)$ for the computation of the specular components. Texture lookups for the material properties accounts for $O(1)$ but are not repeated for each light source since the material is related to the object, so they result in $O(p)$. With three color channels, 4×4 matrices and 4×1 vectors, the SH part takes $O(3(16 + 4)p) \rightarrow O(v)$. The final color computation requires $O((2+1)p)$ that is asyntotically $O(p)$. We can then conclude that the rendering cycle is dominated by a complexity of $O(nLightSources(v+s+p))$.

5.2 Graphic Results

The algorithm has been implemented in an ad hoc graphic engine to show some result of the technique. Table 5.1 reports the software involved in the implementation of this engine:

Table 5.1: *Software adopted for the showcase*

	Microsoft Visual Studio 2015
	OpenTk.Next version 1.1.16
	Accord version 3.2
	Magick.NET-Q16-HDRI-AnyCPU version 7.0.3.1
Assimp Open Asset Import Library	AssimpNet version 3.3.1

In table 5.2 instead are specified the specifications of the hardware used to render the showcase scenes:

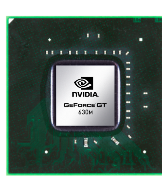
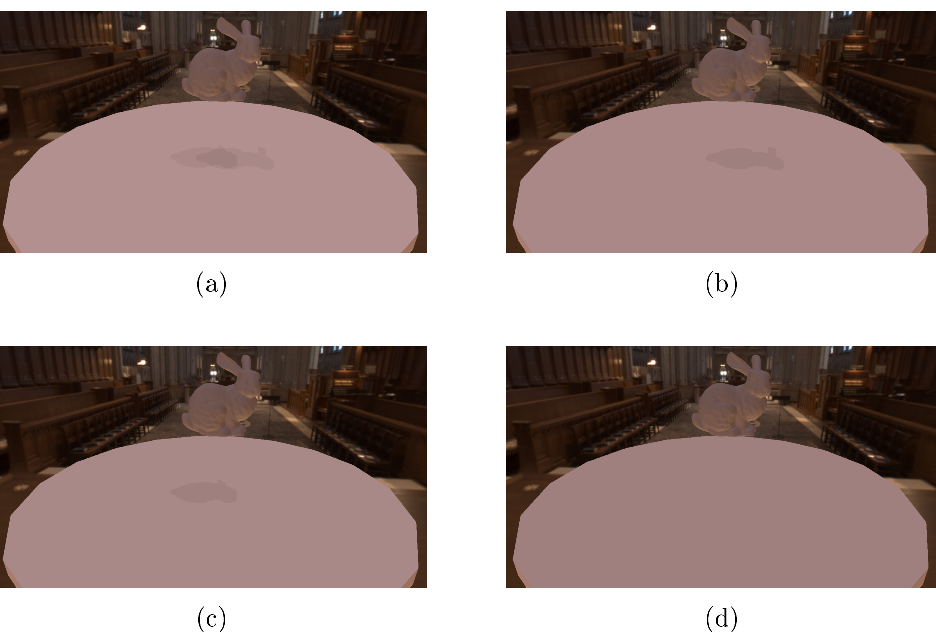
	96 CUDA cores Graphics Clock 800 MHz Texture fill rate up to 12.8 billion/sec Memory Clock 900 MHz Memory interface DDR3 Memory Size 2 GB OpenGL 4.5

Table 5.2: Hardware adopted for the showcase

5.2.1 A simple scene

The first showcase scene is a simple one. It is composed by the famous model ”Stanford Bunny” floating on a flattened cylinder. The chosen environment is the equally famous Grace’s Cathedral and it is rendered on a skybox around the camera. The scene comprises 209538 vertices with positions and normals, and it is rendered on a 1366×768 viewport. The chosen diffuse color is white both for the cylinder and the buddha. The preprocessing phase returns exactly the same EM showed at the end of 4.1, so the shadows of the two located light sources are being cast on the cylinder. The result is showed in figure 5.1:



(a) (b) (c) (d)

Figure 5.1: *Stanford bunny rendered with Grace’s Chatedral environment map in OER*

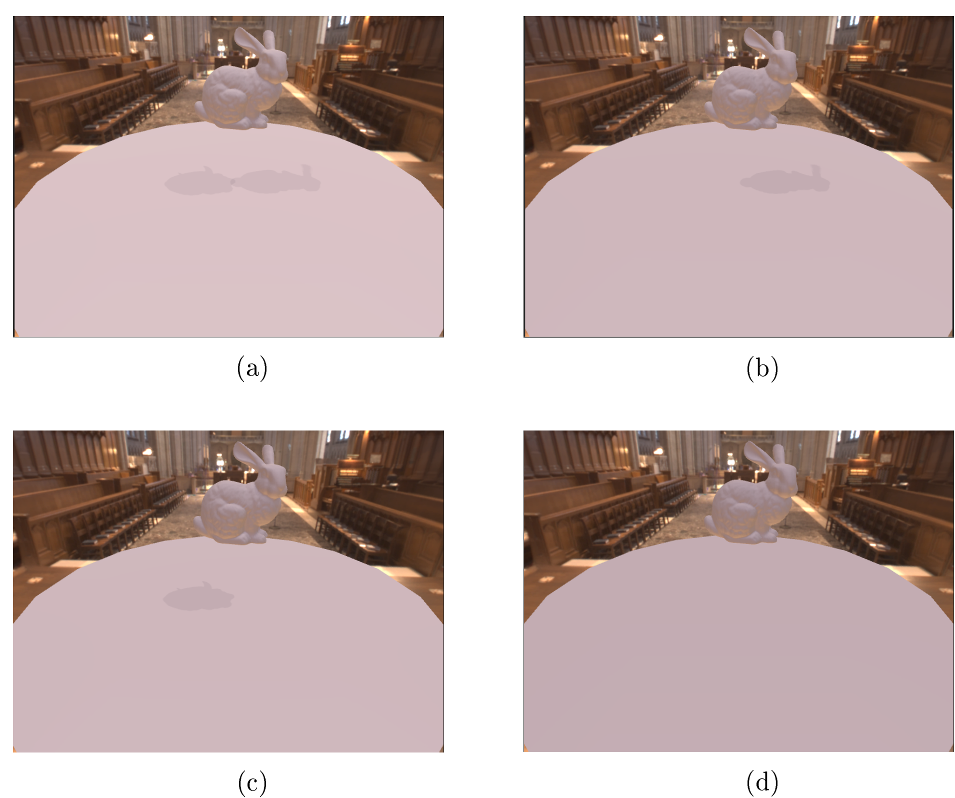
5.2a shows the scene lit by both light sources. The subsequent figures, show instead the result of a mechanism of switching off the light sources. The trick is very simple: no SM depth test is performed for the switched off light source, instead any fragment is considered never lit by that source and the SH matrices are chosen accordingly. 5.2b and 5.2c shows the case where one light source at a time is switched off, while 5.2d shows the case where both light sources are turned on. All of the four settings showed an

interactive refresh rate of $40fps$ (frames per second) circa.

5.2.2 A comparison with Unity standard shader

Unity is a cross-platform game engine developed by Unity Technologies and used to develop video games for PC, consoles, mobile devices and websites. With an emphasis on portability, the engine targets the following APIs: Direct3D on Windows and Xbox 360; OpenGL on Mac, Linux, and Windows; OpenGL ES on Android and iOS; and proprietary APIs on video game consoles.

In order to render a scene comparable to the one showed in the last section, an imitation of the OER technique employing classic environment mapping for ambient light and two directional lights has been rendered in Unity using the standard shader with specular setup. The first issue encountered has been the one of tuning the lights' intensities, colors and directions instead of having them "baked" into the environment: a light too dim might not reproduce the desired shadowing effect while one too bright might falsify the effect coming from the environment lights. The scene is presented in figure 5.2 also in the same four light setups of the previous showcase and it's rendered on a 1024×768 frame buffer. It appears brighter than the one rendered in the ad-hoc game engine but it depends on the fact that ImageMagick and Unity don't adopt the same tone mapping algorithm.



(a) (b)

(c) (d)

Figure 5.2: *Stanford Bunny rendered in Unity with environment mapping and two directional lights*

The frame rate settles also around $40fps$, but the memory consumption is higher due to the fact that classic environment mapping loads the EM on the GPU while OER encodes it with only 16 floating points per channel.

5.2.3 A complex scene

The rendering of a more complex scene is showed in figure 5.3. The scene is composed by 5 object for a total of 3478892 vertices. The structure of a vertex depends on the belonging object: the sword and the sad (very sad, he lost his parents, don't laugh) mummy have, in addition to positions and normals, a pair of texture coordinates for each vertex. To raise the bar, every object is rotating about its local y-axis and the user is able to press a button to make the whole structure of objects rotate of 90 degrees. Of course this is not something one can represent with an image but it has to be kept in consideration while evaluating the goodness of the frame rate. The OER

setup is the same adopted in 5.1:

Figure 5.3: *Composite scene rendered with Grace's Chatedral environment map in OER*

The achieved framerate is $32fps$ thus a loss of 20% against an increment of the number of vertices of the 1500%.

Conclusions

OER is a technique whose goal is to extrapolate lighting information from a surrounding mapped euqirectangularly on a image. The algorithm has proven able to generate dynamic shadows and coherent lighting leading to a realistic enough graphic result, while mantaining an interactive frame rate on a GPU not really up to date with those currently on the market (entry level gpu released in 2012). Furthermore it showed a good scalability when tested against a heavy workload. The bottleneck of the technique lies in the preprocessing phase of course, but it is an offline, una tantum operation. OER has been developed with a side purpose of a possible integration in a ”dressing room” application, an environment where users can visualize products from a shop's wardrobe on a mannequin, but this doesn't limit the range of OER's potential applications. The algorithm could be adopted in solutions for virtual & interactive visits to museums or point of interests, as well as in CAD programs, graphics engines and basically wherever a cheap, image-based, real time GI algorithm can be employed. Furthermore, adopting a very simple model for specular reflection, and being that the only viewer dependent part of the illumination algorithm, OER could be well suited in virtual and augmented reality environments.

The actual status of the technique is experimental, and a couple of improvements are being researched. The first one is related to the localization phase. The attempt is to exploit the covariance matrix of the cluster associated to a light source to generate a ”panel”, a black ellipse with the right dimensions and rotation, to cover the light source. This should get rid of the necessity of computing the attenuation factors.

The second improvement is introducing HDR rendering in the technique.This modified version of OER would project the original EM without

tonemap it first, and then render to a three channeled floating point texture. This texture is then processed by a proper shader, to find parameters to tune a tonemapping algorithm, which is then performed on the texture in order to make it displayable.

Appendix A

List of the first 16 Spherical Harmonics

Considering a standard coordinate system, in increasing order of index $i = l * (l+1) + m$:

1. $Y_0^0 = \frac{1}{2}\sqrt{\frac{1}{\pi}}$
2. $Y_1^{-1} = \sqrt{\frac{3}{4\pi}}\frac{y}{r}$
3. $Y_1^0 = \sqrt{\frac{3}{4\pi}}\frac{z}{r}$
4. $Y_1^1 = \sqrt{\frac{3}{4\pi}}\frac{x}{r}$
5. $Y_2^{-2} = \frac{1}{2}\sqrt{\frac{15}{\pi}}\frac{xy}{r^2}$
6. $Y_2^{-1} = \frac{1}{2}\sqrt{\frac{15}{\pi}}\frac{yz}{r^2}$
7. $Y_2^0 = \frac{1}{4}\sqrt{\frac{5}{\pi}}\frac{-x^2-y^2+2z^s}{r^2}$
8. $Y_2^1 = \frac{1}{2}\sqrt{\frac{15}{\pi}}\frac{zx}{r^2}$
9. $Y_2^2 = \frac{1}{4}\sqrt{\frac{15}{\pi}}\frac{x^2-y^2}{r^2}$
10. $Y_3^{-3} = \frac{1}{4}\sqrt{\frac{35}{2\pi}}\frac{(3x^2-y^2)y}{r^3}$
11. $Y_3^{-2} = \frac{1}{2}\sqrt{\frac{105}{\pi}}\frac{xyz}{r^3}$

12. $Y_3^{-1} = \frac{1}{4}\sqrt{\frac{21}{2\pi}}\frac{y(4z^2-x^2-y^2)}{r^3}$

13. $Y_3^0 = \frac{1}{4}\sqrt{\frac{7}{\pi}}\frac{z(2z^2-3x^2-3y^2)}{r^3}$

14. $Y_3^1 = \frac{1}{4}\sqrt{\frac{21}{2\pi}}\frac{x(4z^2-x^2-y^2)}{r^3}$

15. $Y_3^2 = \frac{1}{4}\sqrt{\frac{105}{\pi}}\frac{z(x^2-y^2)}{r^3}$

16. $Y_3^3 = \frac{1}{4}\sqrt{\frac{35}{2\pi}}\frac{x(x^2-3y^2)}{r^3}$

Appendix B

OpenGL Graphic Pipeline

In 3D computer graphics, the graphics pipeline or rendering pipeline refers to the sequence of steps used to create a 2D raster representation of a 3D scene. Plainly speaking, once a 3D model has been created, for instance in a video game or any other 3D computer animation, the graphics pipeline is the process of turning that 3D model into what the computer displays. There is no unique graphic pipeline, every graphic library has its own. For an example, let's have a look at OpenGL's pipeline (B.1). The OpenGL's rendering pipeline works in the following order:

1. Prepare vertex array data, and then render it

2. Vertex Processing:

   - Each vertex is acted upon by a Vertex Shader. Each vertex in the stream is processed in turn into an output vertex.
   - Optional primitive tessellation stages.
   - Optional Geometry Shader primitive processing. The output is a sequence of primitives.

3. Vertex Post-Processing, the outputs of the last stage are adjusted or shipped to different locations.

   - Transform Feedback happens here.
   - Primitive Clipping, the perspective divide, and the viewport transform to window space.

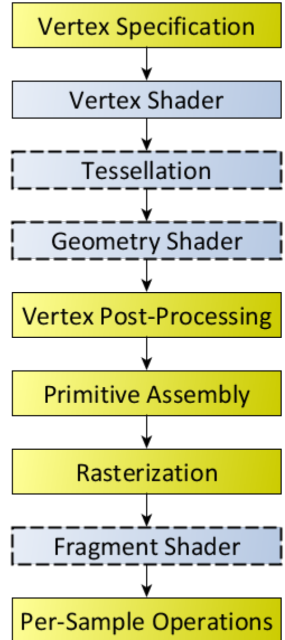


Figure B.1: *Diagram of the Rendering Pipeline. The blue boxes are programmable shader stages.*

4. Primitive Assembly

5. Scan conversion and primitive parameter interpolation, which generates a number of Fragments.

6. A Fragment Shader processes each fragment. Each fragment generates a number of outputs.

7. Per Sample Processing.

The process of vertex specification is where the application sets up an ordered list of vertices to send to the pipeline. These vertices define the boundaries of a primitive. Primitives are basic drawing shapes, like triangles, lines, and points. Exactly how the list of vertices is interpreted as primitives is handled via a later stage. A vertex is represented by a series of attributes. Each attribute is a small set of data that the next stage will do computations on. While a set of attributes do specify a vertex, there is nothing that says that part of a vertex's attribute set needs to be a position or normal.Attribute data is entirely arbitrary; the only meaning assigned to

any of it happens in the vertex processing stage. Once the vertex data is properly specified, it is then rendered as a primitive via a drawing command. The call to the draw command marks the beginning of a processing stage composed by almost all programmable operations. This allows user code to customize the way vertices are processed.Firstly vertex shaders perform basic processing of each individual vertex. Vertex shaders receive the attribute inputs from the vertex rendering and converts each incoming vertex into a single outgoing vertex based on an arbitrary, user-defined program. Vertex shaders can have user-defined outputs, but there is also a special output that represents the final position of the vertex. If there are no subsequent vertex processing stages, vertex shaders are expected to fill in this position with the clip-space position of the vertex, for rendering purposes. One limitation on vertex processing is that each input vertex must map to a specific output vertex. Subsequently, primitives can be tessellated using two shader stages and a fixed-function tessellator between them, but this process is optional. The successive operation named Geometry Shading is also optional. Geometry shaders are user-defined programs that process each incoming primitive, returning zero or more output primitives. The shader is able to remove primitives, or tessellate them by outputting many primitives for a single input. Geometry shaders can also tinker with the vertex values themselves, either doing some of the work for the vertex shader, or just to interpolate the values when tessellating them. Geometry shaders can even convert primitives to different types; input point primitives can become triangles, or lines can become points. The outputs of the geometry shader or primitive assembly are written to a series of buffer objects that have been setup for this purpose. This is called transform feedback mode; it allows the user to do transform data via vertex and geometry shaders, then hold on to that data for use later.The primitives are then clipped. Clipping means that primitives that lie on the boundary between the inside of the viewing volume and the outside are split into several primitives, such that the entire primitive lies in the volume. The vertex positions are transformed from clip-space to window space via the perspective divide and the viewport transform. Primitive assembly is the process of collecting a run of vertex data output from the prior stages and composing it into a sequence of primitives. The type of primitive the user rendered with determines how this process works.The output of this process

is an ordered sequence of simple primitives (lines, points, or triangles). If the input is a triangle strip primitive containing 12 vertices, for example, the output of this process will be 10 triangles. The rendering pipeline can also be aborted at this stage. This allows the use of Transform Feedback operations, without having to actually render something. Triangle primitives can be culled (ie: discarded without rendering) based on the triangle's facing in window space. This allows you to avoid rendering triangles facing away from the viewer. Primitives that passes all the described stages are then rasterized in the order in which they were given. The result of rasterizing a primitive is a sequence of fragments. A fragment is a set of state that is used to compute the final data for a pixel (or sample if multisampling is enabled) in the output framebuffer. The state for a fragment includes its position in screen-space, the sample coverage if multisampling is enabled, and a list of arbitrary data that was output from the previous vertex or geometry shader.This last set of data is computed by interpolating between the data values in the vertices for the fragment. The style of interpolation is defined by the shader that outputed those values. The data for each fragment from the rasterization stage is processed by a fragment shader. The output from a fragment shader is a list of colors for each of the color buffers being written to, a depth value, and a stencil value. Fragment shaders are not able to set the stencil data for a fragment, but they do have control over the color and depth values.The fragment data output from the fragment processor is then passed through a sequence of steps. The first step is a sequence of culling tests; if a test is active and the fragment fails the test, the underlying pixels/samples are not updated (usually). Many of these tests are only active if the user activates them. After this, color blending happens. For each fragment color value, there is a specific blending operation between it and the color already in the framebuffer at that location. Logical Operations may also take place in lieu of blending, which perform bitwise operations between the fragment colors and framebuffer colors. Lastly, the fragment data is written to the framebuffer.

Appendix C

Model Space, World Space, View Space and Screen Space

When an artist authors a 3D model he creates all the vertices and faces relatively to the 3D coordinate system of the tool he is working in, which is the Model Space. All the vertices are relative to the origin of the Model Space, so if we have a point at coordinates (1,1,1) in Model Space, we know exactly where it is C.1. Every model in the game lives in its own Model Space and if you want them to be in any spatial relation (like if you want to put a teapot over a table) you need to transform them into a common space (which is what is often called World Space). With all the objects at the right place the next operation is to project them to the screen. This is usually done in two steps. The first step moves all the object in another space called the View Space.

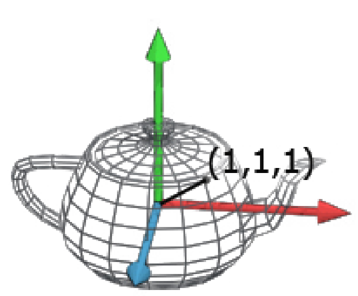


Figure C.1: *Vertex of the Utah Teapot in position (1,1,1)*

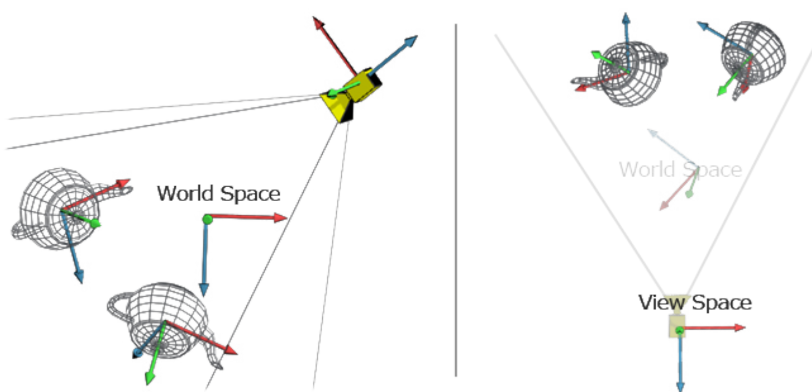


Figure C.2: *Utah Teapots with axis of Model Space, World Space and View Space*

The second step performs the actual projection using the projection matrix. This last step is a bit different from the others and we will see it in detail in a moment. The View Space is an auxiliary space that we use to simplify the math and keep everything elegant and encoded into matrices. The idea is that we need to render to a camera, which implies projecting all the vertices onto the camera screen that can be arbitrarily oriented in space. The math simplifies a lot if we could have the camera centered in the origin and watching down one of the three axis, let's say the Z axis to stick to the convention. So it's convenient to create a space that remaps the World Space coordinates so that the camera is in the origin and looks down along the Z axis.The scene is now in the most friendly space possible for a projection, the View Space. Before flattening the image, the coordinates are projected in Screen Space. This space is a cuboid which dimensions are between -1 and 1 for every axis and whose coordinates are called homogeneous coordinates. This space is very handy for clipping (anything outside the $[-1, 1]$ range is outside the camera view area) and simplifies the flattening operation (we just need to drop the z value to get a flat image).

Bibliography

[1] Paul Debevec. “Image-Based Lighting”. In: (). URL: http://ict.usc.edu/pubs/Image-Based%20Lighting.pdf.

[2] *Equirectangular Projection.* URL: https://en.wikipedia.org/wiki/Equirectangular_projection.

[3] Robert Hoffman Gene Miller. “Illumination and reflection maps: Simulated objects in simulated and real environments.” In: (). URL: http://www.pauldebevec.com/ReflectionMapping/illumap.pdf.

[4] *Global Illumination.* URL: https://en.wikipedia.org/wiki/Global_illumination.

[5] Robin Green. “Spherical Harmonic Lighting: The Gritty Details”. In: (). URL: http://silviojemma.com/public/papers/lighting/spherical-harmonic-lighting.pdf.

[6] Ned Greene. “Environment mapping and other applications of world projections.” In: ().

[7] *High-dynamic-range imaging.* URL: https://en.wikipedia.org/wiki/High-dynamic-range_imaging.

[8] Martin Newell James Blinn. “Texture and Reflection in Computer Generated Images”. In: (). URL: http://cumincad.architexturez.net/system/files/pdf/186e.content.pdf.

[9] *Jim Blinn Model for Specular Reflection.* URL: https://www.siggraph.org/education/materials/HyperGraph/illumin/specular_highlights/blinn_model_for_specular_reflect_1.htm.

[10] *Lambert’s cosine law.* URL: https://en.wikipedia.org/wiki/Lambert%27s_cosine_law.

[11] *Learn OpenGL: Shadow Mapping*. URL: http://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping.

[12] Matteo Matteucci. "A tutorial on clustering algorithms". In: (). URL: http://home.deib.polimi.it/matteucc/Clustering/tutorial_html/kmeans.html.

[13] *OpenTK GitHub's pages*. URL: https://opentk.github.io/.

[14] Mark J. Kilgard Randima Fernando. "The CG Tutorial". In: Addison-Wesley Longman Publishing Co., 2003. Chap. 7.

[15] Pat Hanrahan Ravi Ramamoorthi. "An Efficient Representation for Irradiance Environment Maps". In: (). URL: http://graphics.stanford.edu/papers/envmap/envmap.pdf.

[16] Peter-Pike Sloan. "Stupid Spherical Harmonics (SH) Tricks". In: (). URL: http://www.ppsloan.org/publications/StupidSH36.pdf.

[17] OpenGL Tutorial. *Tutorial 16: Shadow mapping*. URL: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/.

[18] Greg Ward. "Radiance (.hdr) file format specification". In: (). URL: http://radsite.lbl.gov/radiance/refer/filefmts.pdf.