FACIAL RENDERING

Master thesis

AUTHOR:

N. A. Nijdam

SUPERVISOR:

M. Poel

Z. Ruttkay

J. Zwiers

Human Media Interaction

Universiteit Twente
de ondernemende universiteit

Enschede, 20/08/07

PREFACE

This document describes my graduation on the University of Twente, Human Media Interaction division. This document is intended for those who are interested in the development and use of virtual environments created with the use of OpenGL and the rendering procedure for facial representation.

The Human Media Interaction(HMI) division is part of the "Electrical Engineering", "Mathematics" and "Computer Science" division at the University of Twente. HMI is all about the interaction between human and machine and offers a great range of topics. One of those topics is Virtual Reality & Graphics which is a very broad topic in itself. The graduation project falls within this topic with the emphasis on graphics. The project is based on work of The Duy Bui[12], where a virtual head model has been developed for facial expressions using an underlying muscle system, and aims at visually improving the head model. Related work in this field are skin rendering, realistic lighting, organic material rendering, shadowing, shader programming and advanced rendering techniques.

N. A. Nijdam

Enschede, 20 August 2007

CONTENTS

1 Introduction 1

Chapter overview 2

2 Lighting models 5

2.1 Default OpenGL lighting 5

Emission 7
Ambient 7
Diffuse 7
Specular 7
Attenuation 8
Blinn-phong shading calculation 8

2.2 Cook-Torrance 9
2.3 Oren-Nayar 9
2.4 Environment mapping 10

Sphere mapping 10
Cube mapping 13
Paraboloid mapping 14

2.5 Environment map as Ambient light 15
2.6 Spherical harmonics 15

The definition 15
Projection 16

2.7 Lighting and texturing 18
2.8 Conclusion 19

3 Bump mapping 21

3.1 the normal vector 21

Normal vector per polygon 21
Normal vector per vertex 21
Normal vector per fragment 22

3.2 Light the bumps using normal maps 22

Normal maps and object space 22
Normal maps and tangent space 23

3.3 Creating a normal map 25
3.4 Parallax mapping 26

4 Real-time shadow casting 29

4.1 Shadow mapping 29
4.2 Soft shadows using shadow mapping 32

5 Deferred shading 35

5.1 The layout of the buffers 35
5.2 Material pass 36
5.3 Lighting pass 38
5.4 Post process pass 42

5.5 Conclusion 42

6 Facial rendering 43

6.1 The HMI face model 43
The Face mesh 43
Muscle system 44
The conversion 44
6.2 Rendering the skin 44
Basic skin rendering 45
Diffuse mapping 45
Bump mapping 45
Specular mapping 47
6.3 Rendering the eyebrow 48
The blend function 48
The eyebrow mesh 48
Parameters vertex offset and alpha blending 49
Final result 50
6.4 Rendering the eyes 50
6.5 Rendering the Eyelashes 52

7 Conclusion 55

8 Future work 59

Bibliography 61

Appendices 65

A The OpenGL render pipeline 67

A.1 Overview 67
OpenGL Geometric primitives 68
3D models 68
Coordinate spaces 69
A.2 OpenGL buffers 71
Color buffers 72
Depth buffer 72
Stencil buffer 73
Accumulation buffer 73
Frame buffer object 73

ABSTRACT

Based on the work of the TheDuy Buy the primary goal is increasing the visual realism of the virtual head model in a real-time environment. Especially the area around the eyes. In order to do so, several aspects of the head are taken apart and handled separatly.

The skin is updated by using multiple textures for the base color, specular intensity and normal maps providing skin color and skin imperfections. Using a different lightmodel then the default per vertex lighting OpenGL model, a more realistic lighting environment is created reflecting the skin more realistic. The lighting model constists of three parts: ambient lighting using Spherical harmonics, diffuse lighting using Oren-Nayar model and specular lighting using the Cook-Torrance model. In combination with bumpmapping (using the normal maps) a very detailed skin surface is constructed.

The eyebrows are replaced with a texture based patch, providing a more customizable eyebrow. A specialised shader (fur shader) is written in order make the eyebrow appear more voluminous.

The eyelashes are replaced with a more modular and generic approach. Where the original model uses hardcoded nodes, the new model uses a freely movable bouding box which provides the place where the eyelash is created (using the vertices within the bounding box).

The eyes are separated into two parts: the eyeball and the iris. The iris uses a specialised shader, providing the eye visually with sublte effects like a lens distortion effect, a reflective spot on the outside of the iris, an inside illumination on the iris and an environment reflection stretched over the eye.

The head is provided with a self-shadowing technique called shadowmapping. Extended with a softshadowing algorithm, subtle lighting effects are created on the face. Improving the way light affects the head.

A secondary goal is providing the used techniques in a modular way. This is done by the use of a system also known as Deferred shading. Deferred shading provides layers that can be stacked. In short the system splits the default way of rendering into several stages of rendering. First the material pass which takes the geometry and supplemental textures as input and outputs a set of buffers containing information used by sequential passes of rendering. The second stage is the lighting stage, using the information from the first stage (such as, basecolor, normals and material properties). Several layers (for every lightspot a layer) are blended together. The final stage is the post process stage (optionally) and can perform 2D effects(color balance, saturation, blurring, blooming etc) on the final output buffer.

1

INTRODUCTION

Recent developments in Human Computer Interaction (HCI) try to incorporate Virtual Reality (VR) techniques in situations where human-human interactions as well as human-computer interactions are important. For instance, human avatars can represent participants within a meeting situation where the real person cannot be physically present. In health care, an avatar might provide information and instructions to patients, or talk to elderly persons. In most situations, it is required that the avatar is, at the very least, a “believable” human. Such avatars should look reasonably realistic. Moreover, they should be able to show human-like facial expressions and emotions, and should show appropriate body language, including gaze behaviour. The use of graphics can give a certain visual impression for a virtual environment. The focus on graphics can be to make a virtual reality as realistic as possible or a more simplified approach with a cartoon style rendering. This is dependent on how the most important aspects of graphics, which are lighting and materials, are simulated. The use of graphics in a real time virtual reality is dependent on current hardware capabilities and software technologies. Graphical technologies which were only possible in the past by non-realtime rendering can now be performed in real time. This offers new possibilities to implement new visual presentations for virtual reality which have the requirement to be performed in real-time.

In this master’s thesis, we focus on one aspect: the visual appearance of the human face, and, more in particular, the area around the eyes. A second goal was to improve substantially the applicability and reusability of a software model based upon Parke & Water’s original head model, later on further developped by The Duy Buy.

- **The primary goal of this project is to increase the visual realism of the virtual head model in a real-time environment. Where the focus of attention is in the region of the skin, eyes and eyebrows. Related work in this field are skin rendering, realistic lighting, organic material rendering, shadowing, shader programming and advanced rendering techniques.**
- **A secondary goal is to have the improved head model and the advanced render techniques available for other systems which use OpenGL and shaders to create virtual environments. In order to visualize the new head model, a new stand-alone system, capable of rendering the improved head model, is developed utilizing the rendering techniques. The rendering system provides a modular structure for using the several rendering techniques. The modularity keeps the rendering techniques independed from each other and are more flexible to use/implement in other systems. In order to use the advanced render techniques together and still keep the modularity, a rendering structure called deferred shading[28] is designed.**

The head model from The Duy Bui uses the default OpenGL lighting (per vertex Phong-Blinn based lighting model) and has no support for texturing. The head appearance is based on a single skin color and uses a constant reflective intensity over the whole face. The per vertex based lighting model doesn't support subtle facial irregularities. This clearly doesn't look realistic. Facial irregularities are important as it defines the look of a face of a person. It shows age and imperfections of the face such as scars and birthmarks which gives an impression of the character and makes it unique. In order to support these aspects a per fragment based lighting model is used. With today's hardware shader capabilities this is easily done in real-time and can support complex lighting models in real-time. With more accurate material properties the skin can be simulated more realistically, instead of using a single color on the face. This directly involves the use of textures, which can improve the appearance of the head model by using an image providing the color information on the model. The use of textures in real-time isn't new, however, the use of textures and hardware shader capabilities give a new dimensions on how to use these textures.

Usually a texture is a two dimensional array where each entry contains a color. Using a texture to lookup the color for a specific fragment is its main purpose, but textures can be used to store all kinds of information and can be utilized in shader programs as inputs for an algorithm (such as a lighting model). A basic lighting approach is to calculate the light reflective intensity. This is usually done by a dot product of the normal vector (the normalized perpendicular vector on the surface of an object) and the light vector (the normalized vector pointing from the surface of an object towards the light source). The normal vector usually is calculated by a planar equation from the polygon, but can be manipulated. The manipulation of this vector can create the impression of bumps (so called bump mapping) and can be further extended to create facial irregularities such as wrinkles. These manipulated normals can be stored in the textures. Another use for textures is to store the specular intensity. This is useful for the face in order to make certain areas on the face more reflective. This for example is the case with the nose area which in contrast to the cheeks is more reflective. Also if sweating a lot the forehead will appear more shiny then most parts of the face. A texture can incorporate these informations at a per fragment level, and can easily be read by a shader program. This offers new possibilities on how to use the head model and on how we want to present a model.

As the eyebrow is part of the face it is taken into account on how we perceive a person (as we directly create an opinion based on looks). They are used to complement facial expressions such as frown or just raising one eyebrow. By making these more realistic it can raise the acceptance of the person and the possible expressions. The head model from The Duy Bui offers a very basic eyebrow represented by a small arc stripe with a single color. In order to make this more realistic the single stripe is replaced by a texture based eyebrow, combined with a custom designed shader program (a so called fur shader) to create a more voluminous eyebrow.

The eyes from the head model use a low quality texture containing the iris color information, and also a static light reflection spot. Using a low resolution texture can give a good result on a small object or at a greater distance, however when only rendering the face, the details of the eye are more evident. Therefore a higher resolution texture capable of containing a more detailed iris is used, which can be from a real source (such as a photo) or custom generated. The static light reflection spot gives the impression of a light source reflecting in the eye, but this can be easily done in real time by the light model. By providing the eyes with a high reflective intensity the light model can take the actual light sources in the scene to provide accurate reflection highlights in the eye. This in order to provide additional cues on how the light is affecting the head model and making it more realistic. Another aspect of light is the creation of shadows. Shadows provide additional cues on how the light is distributed over the face. A shadowing technique called shadow mapping is therefore implemented.

A minor part of the eyes are the eyelashes which are provided by a static eyelash model. Replacing this model with a more custom designed eyelashes using an external texture gives the possibility to create new eyelashes more easily and more realistic eyelashes can be provided.

Where most research papers discuss one technique, here a full scale of methods is presented and described on how they can be used together. The rest of this chapter gives a more detailed introduction/overview on the project. The first 5 chapters introduce the several rendering techniques and its uses and in chapter 6 the techniques are used on the head model.

Chapter overview

Chapter 2 “Lighting models” handles the lighting. Lighting is the way every pixel on screen is lit (or colored). Starting with the default lighting model emphasizing the important components of a lighting model. In the system of The Duy Bui the shading of the head model is done by the default lighting model in OpenGL, which is a per vertex based lighting model. Using a set of diffuse colors to colorize certain facial regions (skin, lips, eyebrows). To improve the visual appearance

of this model certain key areas are defined: lighting, materials and special effects. The lighting is basically the most important aspect of visualizing the head model, the material gives structure to the surface of the head model and the special effects can add certain modifications to the final image before it is presented on screen. The interaction between the lighting and the way materials are handled is very close, and is incorporated in a so called lighting model or bidirectional reflectance distribution function model (BRDF). The lighting in a virtual environment is an approach for real-world lighting using a mathematical algorithm calculating the "correct" color at each pixel. The material information is used as attributes for regulating the algorithm, which can be done by setting the reflective colors or special function parameters (such as a Fresnel factor used by certain BDRF models). Setting the “correct” color at each pixel is done by combining certain colors based on the lighting aspects. This can be very basic or very complex. In the system from The Duy Bui a most simplistic diffuse color is used, which defines only one color to be used as the diffuse reflection color for a part of the face. To make this more complex, lighting can be separated into multiple components, ambient, diffuse and specular, where for each component a color can be assigned. Still this is on a complete region and to create more detail the use of surface color maps are used, which contain the color information to be used at any part of the face. This is also called texturing. This approach can be further extended by multi-texturing (using multiple color maps as layers over each other) where the way of combining the textures is also defined by a algorithm. The color maps are used for the diffuse component. The way the diffuse is handled can be replaced by other models. OpenGL default is the Lambertian diffuse model which is used by the Blinn-Phong shading model. Here another well known model is presented, the Oren-Nayar reflection model and is used to replace the default diffuse lighting model. This diffuse model simulates rough surface features, which scatter light more unevenly then the Lambertion diffuse model, and makes it ideal for a skin surface. The other components can also be extended with a variety of techniques. The ambient component can be one color, or a more complex model. For example a low frequency light model, based on a so called spherical harmonics, can be used. Another method is a real-time environmental ambient illumination, by taking the colors around the object as an ambient input (environment mapping). The last component of the lighting model , called “specular lighting”, is used to show highlights and is presented here with the Blinn-Phong model and the Cook-Torrance model.

Chapter 3 “Bump mapping” introduces a technique for modulating the lighting at a per fragment level in order to simulate bumps which can be used to simulate wrinkles and skin irregularities.

Chapter 4 “Real-time shadow casting” handles the self-shadowing. Although it may seem to be a part of the lighting model, shadowing is a complete separate subject. With lighting calculations one specific point (vertex or fragment) is processed at a time and for that point in solitude a lighting calculation is performed using only the point information and the lighting information. However this doesn't include concepts like occlusion. If the light ray is intercepted by an object standing in front of the point being illuminated a specialized algorithm is used to store that information and to use it to "add" shadows to the virtual scene. In this project a technique known as "shadow mapping" has been used, which is a fast and easy to implement method and most of all it delivers good self shadowing results. This technique uses a two pass rendering, which means that the whole scene is rendered two times in one render cycle. First the scene is rendered from the point of the light source, storing the distance of each object towards the light(depth), and the second pass is the normal render pass from the users view point, which then uses the depth information to recalculate if a certain object is occluded by another point.

Chapter 5 “Deferred shading” discusses a rendering system known as deferred shading. This system makes it possible to have multiple lighting models separated from each other (each has it's own shader program) used in one render cycle. This makes the system perfect for testing new lighting models and layered usage of shader programs. Each pass can have its own shader program operating on the same output buffer and therefore it can be seen as a layer. The use of this system in this project is the main rendering method for the project because of its flexibility for adding,

removing or modifying the layers. The scene can be rendered(pushing the vertex data) one time, store information about the scene in several buffers and make the buffers available for sequent render passes.

Chapter 6 “Facial rendering” describes the usage of the previously mentioned techniques combined used with the several parts of the head model, including skin, eye, eyebrow and eyelash rendering.

The form of skin rendering that we have used for rendering the head mesh is based on a one color map and two so called normal maps for simulating wrinkles, skin structure and other skin irregularities. The base color map contains colors that have to be applied on the head mesh. This includes the combined information of skin color, lip color and colors for skin irregularities (for example freckles or birthmarks). The first normal map is used to be applied to the whole face(spanned across multiple polygons) and contains the information on wrinkles and other facial bumps. The second normal map contains information about the skin micro structure and is applied to single polygons, in order to make the light scatter a bit more on the skin.

The rendering of the eyes is broken into two pieces: the eyeball and the lens, where the lens uses a small refraction shader program to simulate a subtle lens effect. The eye uses a color map for texturing the eyeball (blood veins and other eyeball color information) and a color map for the lens which contains the iris and pupil colors. Also a complete procedural texture generation of the eyeball and iris texture is presented.

The rendering of the eyebrows is based on a so called fur shader program. Such shader programs try to emulate the appearance of 'fur-like' materials but do not rely on a potentially very complex geometric model of the material. First the eyebrow region is extracted from the head model, as this regions needs to be rendered multiple times because of the fur shader. This is done by an algorithm that first looks at which vertices are within a custom defined bounding box (which defines the area where the eyebrow is). Using the filtered vertices the polygons that use these vertices are extracted. The filtered polygons are then rendered multiple times using a fur shader program, which renders the polygons with a little offset. This simulates depth which gives the eyebrow more volume, instead of being flat on the head model surface. This is done in combination with an alpha map texture to mask out the eyebrow structure.

The eyelash rendering follows the same polygon filtering method as the eyebrow. The difference here is the usage of the filtered polygons. As the eyelash spans across a fine line above the eye, the edges of the polygons are used as a guide to generate an eyelash. By extruding the eyelash vertices and creating a quad strip OpenGL is able to render the eyelash. For every vertex also the texture coordinates are calculated in order to map an eyelash color map onto it.

Chapter 7 “Conclusion” summarizes the final results of this project by highlighting the advantage of the used techniques and the improvements but also the problems with the current approach.

Appendice Chapter A “The OpenGL render pipeline” gives a short overview of the basics of OpenGL and highlights key components that have been used in this project. As this is an engineering project the reference to the use of OpenGL is made several times throughout this document, by mentioning certain functionality from the OpenGL library or its utility library. OpenGL stands for Open Graphics Library and is widely supported over many platforms. Using OpenGL and the OpenGL Shader Language (GLSL) makes it possible to create so called shader programs which are small programs dedicated to some specialized graphics effects that are running on the graphics hardware. This gives the ability to take control on how a point in space (vertex) or a pixel (fragment) is handled. Also an OpenGL extension is used for drawing off-screen, which means rendering in the background and the result stored in a separate buffer from the default buffers so that it can be used during later stages in the render process.

2

LIGHTING MODELS

Lighting is one of the most important aspects of rendering, if not the most important. In a mathematical sense it describes the way how a surface is lit. Therefore we look at several existing lighting models which are usually called Bidirectional Reflectance Distribution Function (BDRF) models. Starting with an extensive explanation about the default OpenGL lighting model (Blinn-Phong shading model) introducing the basic light calculation aspects. The other BDRF models discussed here are Cook-Torrance and Oren-Nayar. The mathematics of these models are based on the book Advanced Lighting and Materials with Shaders by K. Dempsi & E. Viale[5] and may have some changes in the formulas because of taking account of to the way they are implemented in a shader language.

2.1 DEFAULT OPENGL LIGHTING

The OpenGL lighting is based on the Blinn-Phong[1] shading model. Where the base is the Phong reflection model and the Blinn part a modification to the way specular highlights are calculated. In contrast to ray traced and radiosity models, second-order reflections are not supported. This light model basically simplifies the normal way of calculating lighting by splitting various lighting attributes. Two main categories are: the lighting attributes and material attributes. These attributes are used together to create an approach for realistic lighting. In the following points are the distinct attributes described. Each attribute can be part of material, light or both of them have such an attribute (as noted between the square brackets)

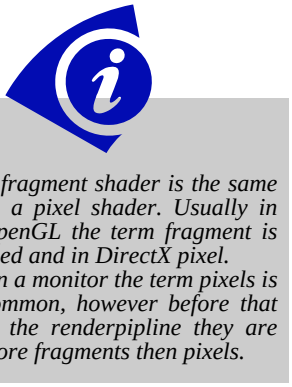
- **Emission [material]: Light emitted from the polygon. It is unaffected by any light source and also does not affect other objects (it is not acting as an additional light source, i.e. radiate light).**
- **Ambient [material and light]: Overall illumination. This can be seen as an indirect light source that affects the complete environment and actually it is a kind of simplification for a global illumination model (where light is being reflected and radiated on other objects).There is also a global ambient attribute that can be set besides the light and material ambient.**
- **Diffuse [material and light]: Light that comes from a certain direction (light source) hits a polygon and then is scattered equally in all directions.**
- **Specular [material and light]: better known as specular highlights or shininess. Just like diffuse it comes from a certain direction and acts as a sort of mirror (reflects the light almost 100%).**
- **Shininess [material]: used in combination with specular, and acts as its exponent parameter to regulate the highlight spot intensity.**

In practice this usually comes down to experiment with the attribute values until a good combination is found. But let's take a look at a more in-depth explanation (based on the OpenGL 'Red book'), with the actual equations and usage of the attributes. First of the definition of the equation given by the red book in textual format:

> *"Vertex color = the material emission at that vertex + the global ambient light scaled by the material's ambient property at that vertex + the ambient property at that vertex + the ambient, diffuse and specular contributions from all the light sources, properly attenuated."*
>
> *Red book[1]*

Notice that the vertex color is mentioned and not the fragment color, this is because the default OpenGL lighting is a vertex based lighting model. The color at the vertex is interpolated to the color of the next vertex. This interpolation method is called Gouraud shading[1], where a smooth color transition is achieved over a polygon.

There are OpenGL extension to have per fragment lighting or one can write a shader program incorporating the same equations but on fragment level. In figure 2.1 this is shown on a low polygon model. On the left you can clearly see how the specular highlight is flowing from one point (on the leg), in contrast to the model on the right. There even are highlights showing in places where a vertex based lighting model shows nothing at all (for example at the feet)



A fragment shader is the same as a pixel shader. Usually in OpenGL the term fragment is used and in DirectX pixel.
On a monitor the term pixels is common, however before that in the renderpipline they are more fragments then pixels.

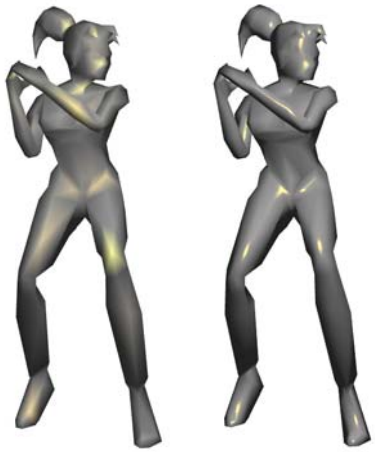


Figure 2.1: Left the model has vertex lighting and on the right per fragment lighting.

Going from a vertex based model to a fragment based model clearly gives visual improvements however at a greater computational cost. Since normally only the equation is done on each vertex and now for every fragment. To make it clearer, if the model consists of about a 1000 vertices then the vertex lighting model is executed a 1000 times, however for a fragment based lighting model this can easily add up to 1.310.720 executions every render cycle for a screen resolution of 1280x1024 with the same model.

For the lighting calculation a couple of vectors are important, they are shown in figure 2.2.

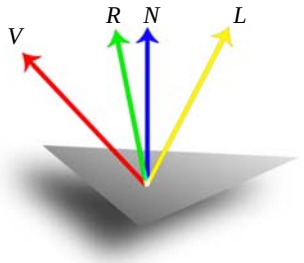


Figure 2.2: The basic vectors used in lighting equations. View, Reflection, Normal and Light vector.

- *V* **The viewing vector, pointing in the direction of the viewer.**
- *R* **The reflection vector, used for specular highlights.**
- *N* **The normal, the vector perpendicular on the polygon.**
- *L* **The light vector, pointing in the direction of the light source.**

These are all unit length vectors.

Emission

The emission is simply an RGB color that can be assigned to the fragment or vertex. It isn't affected by any other attribute. This implies, for instance, that when emission if full white, the object being rendered will always be full white. As shown in the equation below.

$$E = M_e$$

E The resulting color, the Emission component
M_e Material emission color

Ambient

The material can be set to an ambient color; a light can be set to an ambient color and the world ambient an RGB value. The world ambient color is scaled by the material ambient color. And for each light the ambient color is scaled with material ambient color. Both of them are added up and form the ambient component.

$$A = W_a M_a + \sum_{lights} (L_a M_a)$$

A Ambient component
W_a World ambient color
M_a Material ambient color
L_a Light ambient color

Diffuse

With the diffuse attribute the normal vector comes into play. Based upon the normal it checks if the light is hitting vertex.

$$D = \sum_{lights} (max\{L \cdot N, 0\}\ L_d M_d)$$

L The light unit vector from the vertex to the light source. $\begin{bmatrix} L_x & L_y & L_z \end{bmatrix} - \begin{bmatrix} V_x & V_y & V_z \end{bmatrix}$
D The diffuse component
N The unit normal vector.
L_d The light diffuse color.
M_d The material diffuse color.

The max function returns the greater value of two values. Thus here if the dot product between L and N is negative the function returns zero. If zero is the result the diffuse component is discarded, this causing one sided lighting.

This is also known as the Lambertian diffuse model, because it depends on Lambert's law; it gives ideal diffuse reflection (intensities) in any direction.

Specular

The specular component like the diffuse component also depends on the normal vector, but also uses the reflection vector. The reflection vector is calculated using the Blinn method, where it is better known as the half-vector.

Normally the Phong specular reflection color is computed:

$$R = 2(N \cdot L)N - L$$

$$S = max\{R \cdot V, 0\}^{M_x} L_s M_s$$

R The reflection vector.
S The specular component
M_x Material shininess, also referred to as the specular exponent.
L_s Light specular color (also referred as the intensity).
M_s Material specular color (same remark as L_s).

Blinn proposed an optimized reflection calculation, by introducing the half-vector.

$$H = L + V$$

$$S = max\{N \cdot H, 0\}^{M_x} L_s M_s$$

H The half-vector. Sum of the Light and View vector.

Attenuation

If the current attribute calculations are put together, a light source would have an infinity range of effect. This is often used for static environment lighting (like having sun light). For lights that are limited by a range of effect the attenuation factor is used as a scalar and is based on three coefficients.

$$A = \frac{1}{k_c + k_l d + k_q d^2}$$

A The attenuation factor.
k_c constant attenuation.
k_l linear attenuation.
k_q quadratic attenuation.
d The distance between the vertex/fragment and the light source.

Normal vectors in OpenGL can be defined per face or per vertex. If defining a normal per face then the normal is commonly defined as a vector which is perpendicular to the surface. In order to find a perpendicular vector to a face, two vectors coplanar with the face are needed. Afterwards the cross product will provide the normal vector, i.e. a perpendicular vector to the face.

Blinn-phong shading calculation

$$C = M_e + W_a M_a$$

$$+ \sum_{lights} \left(\frac{1}{k_c + k_l d + k_q d^2} ((L_a M_a) + max\{L \cdot N, 0\} L_d M_d + max\{R \cdot V, 0\}^{m_x} L_s M_s) \right)$$

The only thing that has been left out here is the "spotlight effect". The spot light effect defines a boundary (alike the attenuation factor) and is multiplied with the attenuation factor.

$$P = max\{L_{sd} \cdot L, 0\}^{L_{se}}$$

P The spot effect factor.
L_{sd} The light spot direction (unit vector).
L_{se} The light spot exponent to control the intensity distribution.

This completes the light model for the default OpenGL lighting; there are however additional settings and possibilities provided by OpenGL. Also there was no texturing involved (not to mention multi texturing), and no blending (transparency). A word on these issues is given in section 2.7 “Lighting and texturing” on page 18, since the other lighting models also have the same issues on these topics a word given on this is done afterwards.

2.2 COOK-TORRANCE

The Cook-Torrance model[35] is a specular component lighting model, and is based on the model of Torrance and Sparrow. The model takes a Fresnel term to modify the reflected amount of light in different angles. By doing this it introduces microscopic facets on a surface, where every facet is a ideal reflector.

$$S = B\frac{DGF}{(N \cdot L)(N \cdot V)}$$

B A base color, this can be the diffuse color or specular color.
D The distribution term for the facets orientations.

Which can be for example the Blinn function:

$$D = R_1 e^{-\left(\frac{\arccos(N \cdot H)}{R_2}\right)^2}$$

R_1 Roughness parameter 1 (a user selected scalar/constant).
R_2 Roughness parameter 2 (the average gradient of the facets).
H The half vector as shown in section “Default OpenGL lighting”. Can also be some other function for the reflection vector.

Other distribution function can be used (such as the Beckman distribution[5], is however more complex and performance intensive).

G Describes the shadowing and masking effects between the micro facets.

$$G = min\left\{1, \frac{2(N \cdot H)(N \cdot V)}{V \cdot H}, \frac{2(N \cdot H)(N \cdot L)}{V \cdot H}\right\}$$

F The Fresnel reflection term corresponding to a material's index of refraction. The Fresnel term is here given for completeness.

$$F = \frac{(g-c)^2}{2(g+c)^2}\left(1 + \frac{(c(g+c)-1)^2}{(c(g+c)+1)^2}\right)$$

$$c = V \cdot H$$

$$g = \sqrt{\eta^2 + c^2 - 1}$$

η The indice of refraction for a material (can be wavelength dependent or as stated by [5] "at least a color channel").

Because of the complexity often a simplification is used, created by C. Schlick[13]

$$F = f + (1+f)(1 - V \cdot N)^5$$

$$f = \frac{\left(1 - \frac{\eta_1}{\eta_2}\right)^2}{\left(1 + \frac{\eta_1}{\eta_2}\right)^2}$$

Here η_1 and η_2 are the material's indices of refraction.

2.3 OREN-NAYAR

The model presented by Oren and Nayar[34] is a diffuse reflection model and differs from the Lambertion diffuse model as it introduces retro reflection (reflection back to the light source). The

equation presented here is based on the equations provided by [5], with some notation changes (part of the equations provided in the book are only given in shader code).

$$D = max\{N \cdot L, 0\} L_d M_d (A + B \sin(\alpha) \tan(\beta) max\{\cos((C), 0)\})$$

$$A = 1 - 0.5 \frac{\sigma^2}{\sigma^2 + 0.33}$$

$$B = 0.45 \frac{\sigma^2}{\sigma^2 + 0.09}$$

$$\alpha = max\{\text{acos}(N \cdot L), \text{acos}(N \cdot V)\}$$

$$\beta = min\{\text{acos}(N \cdot L), \text{acos}(N \cdot V)\}$$

D The resulting diffuse component color.

$max\{N \cdot L, 0\} L_d M_d$ The first part of the equation is the base diffuse color and resembles the Lambertian diffuse model.

N The unit normal vector.

L The unit light vector.

L_d Light diffuse color.

M_d Material diffuse color.

σ The sigma represents the roughness of the surface.

C The angle between the light vector and the view vector.

$$C = \hat{E} \cdot \hat{F}$$

$$E = V - N(V \cdot N)$$

$$F = L - N(L \cdot N)$$

The implementation of this model is quite instruction costly. In [5] they present a optimizing method by storing particular data in a texture and use the texture as a two dimensional lookup table and reduce the amount of instruction from 70 to 27 (for comparison: the Blinn-Phong model uses about 20 instructions).

2.4 ENVIRONMENT MAPPING

Originally developed by Blinn and Newell[14] environment mapping is a technique to reflect the surrounding on an object. It can however also be used as a replacement for global ambient or for specular highlighting. The downside is that some reflective object cannot “easily” reflect itself when using realtime generated environment maps. Note that self reflection is only possible on concave objects. By default the backface culling is enabled in OpenGL (this means that the back side of a polygon is not being rendered) and therefore it would be possible to render itself on itself with a very small near offset for the camera. In practice this only gives a lot of trouble with artifacts and incorrect reflections.

> “At an abstract level, what an environment map supplies is a fast way to determine the incident irradiance in any direction at a particular fixed point in space."
>
> OpenGL.org[37]

Sphere mapping

Basically this method creates, using for example the reflection vector, texture coordinates for a lookup texture. This texture can be seen as a textured sphere containing a complete panoramic environment obtained from an external source (for example a real world image) or a real-time

render with the off-screen view from the object towards the viewer. This is also known as Sphere mapping, and OpenGL supports this. Internally OpenGL uses the following formula to create the appropriate spherical st coordinates.

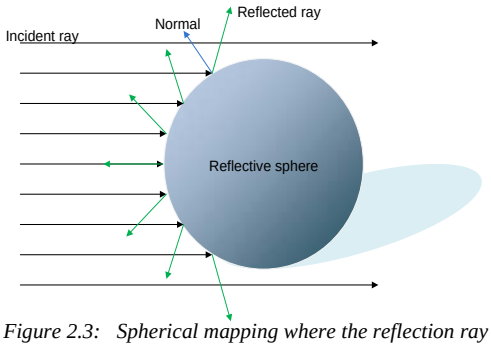


Figure 2.3: Spherical mapping where the reflection ray is used for lookup in a texture.

On a side node: "UV coordinates" is also often used instead of "ST coordinates" to denote texture coordinates, this is because they are often the same (U==S and V ==T). The st coordinates are Surface Texture Parameters and uv coordinates are Surface Parameters (for example used for NURBS).

$$s = \frac{R_x}{2p} + 0.5$$

$$t = \frac{R_y}{2p} + 0.5$$

$$R = V - 2(N \cdot V)N$$

$$p = \sqrt{R_x^2 + R_y^2 + (R_z + 1)^2}$$

s and t are the texture space coordinates.
R The reflection vector.
V The unit view vector.
N The unit normal vector.

The reflection vector after normalization lies in the range of $[-1, 1]$ however for a texture it must be in the range of $[0, 1]$ therefore the scale by 0.5 and the bias of 0.5.

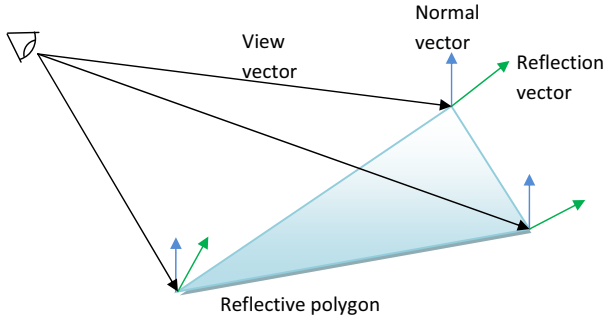


Figure 2.4: A polygon showing on each vertex the view, normal and reflect vector.

In practice, any image loaded as a texture can be used for this method and the surprising part is that even when not "really" reflecting the actual surrounding, it often is sufficient realistic instead

of real-time environment mapping. Most important when using a non-realtime texture is to have the same coloring/tone as the environment.

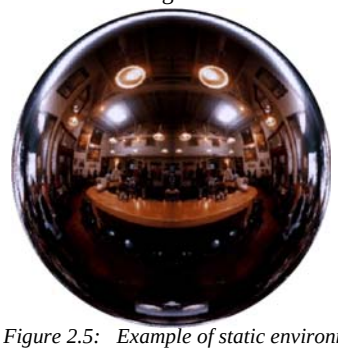


Figure 2.5: Example of static environment map (SGI).

Another method often used when looking at real-time environment-mapping is sharing of the environment map. Normally for every object that has some reflection it should have its own "per render cycle" updated environment map. However common practice is to use "one" environment map (created from a predefined point) for all reflective objects. These are performance related considerations, and real-time environments cannot afford to lose too much performance if the effect gain is minimal. Especially in video games this method is often used.

So we know now that environment mapping is a technique where we get the reflection from an image, and that this image can be a real photo (external source), or a real-time environment map. The most interesting part here is the real-time environment mapping. There are several techniques to create environment maps and to map them onto an object. The first one mentioned is spherical mapping. The dynamic creation for a spherical map is tricky, because one needs to render the reflective part with a possible optimal angle of 180 degrees (if not more for some mirror objects). To be precise, this is an off screen render pass, where the reflection view point is set to the origin of the object and the viewing direction is set to the inverted direction of the viewer (looking at the camera so to say). Then it captures (renders) the scene (without himself) into a texture (a texture buffer object bound to a logical color buffer bound to an FBO). However in perspective mode with a wide angle the resulting image gives a strong fisheye distortion. A solution to this is an orthographic mode, thus orthographic parallel rendering. To illustrate this, figure 2.6 simulates the effects of having a perspective camera (with a large angle, 180 degrees) in the middle of the object giving undesired 'pinch' effects in the resulting sphere map.

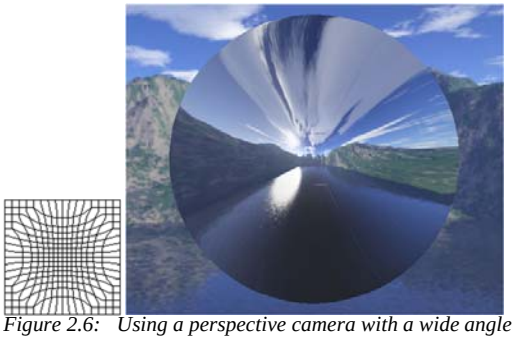


Figure 2.6: Using a perspective camera with a wide angle gives a pinch effect in the environment map.

The image below shows a better effect of spherical mapping using an orthographic camera.

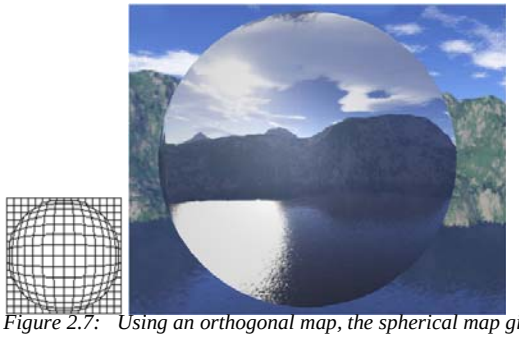


Figure 2.7: Using an orthogonal map, the spherical map gives better results.

The main downside of spherical mapping is that it is view dependent, because the reflection map is created with the use of the inverted viewer direction. View independent methods are cube mapping and paraboloid mapping.

Cube mapping

With cube mapping, all six directions are being rendered to textures, also called a cube map.

> *Cube map textures provide a new texture generation scheme for looking up textures from a set of six two-dimensional images representing the faces of a cube.*

Each face is rendered by a different view direction with a 90 degrees angle (horizontal and vertical, in perspective mode), combining them to a cube map.

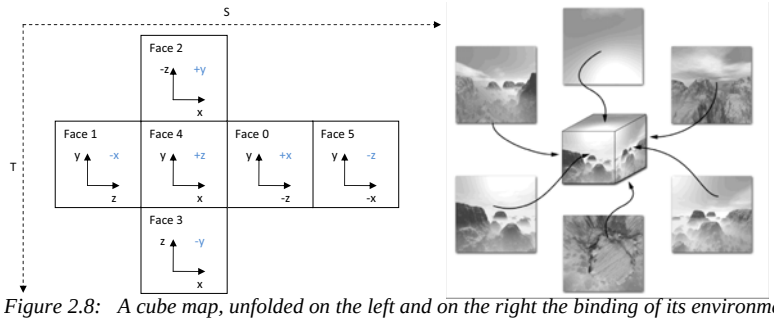


Figure 2.8: A cube map, unfolded on the left and on the right the binding of its environment is illustrated. (http://www.developer.com/img/articles/2003/03/24/EnvMapTecIm01.gif)

Retrieving the appropriate ST coordinates for mapping the reflection vector is sufficient.

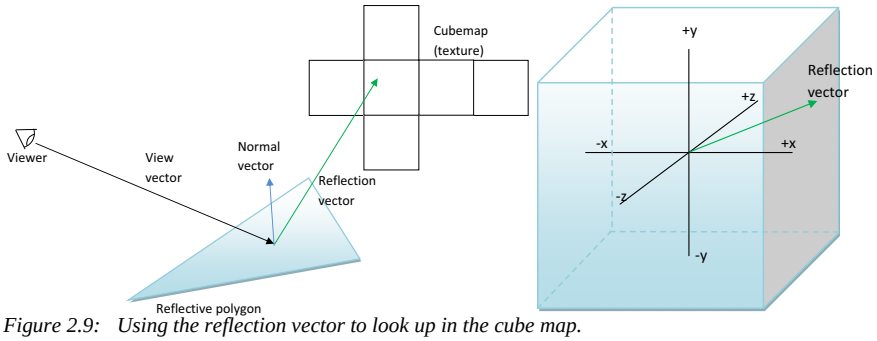


Figure 2.9: Using the reflection vector to look up in the cube map.

The reflection vector is used from the center of the cube. A face is then selected based on the largest magnitude coordinate. The major axis, and in figure 2.9 this would be the "+X" face where

the value x is >0 (otherwise it would hit the -X face) and x is in magnitude larger than the Y and Z values (x>Y , x>Z). To get the ST coordinates, the other two values are divided by the major axis (in the example: s=y/x,t=z/x) The new coordinates can then be used for the lookup in the texture[1].

This method creates realistic environment mapping however it uses significantly more resources since it has to render 6 times. If the environment map isn't updated every render cycle it is a good solution (or if graphics power doesn't matter).

Paraboloid mapping

The paraboloid mapping, created by W. Heidrich & H. Seidel [15] is similar to spherical mapping, however it uses two textures instead of one. Although this is known as a two pass rendering, it is possible to do it in one pass by using multi render targets (only if supported by the graphics hardware). The environment map is taken from the center of the reflective object and then the scene is divided into two hemispheres by rendering the front and the back or better said from the object perspective "rendering forward and backwards".

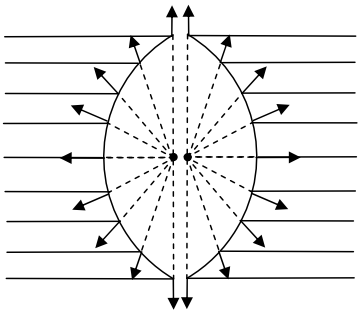


Figure 2.10: Paraboloid mapping, using only the front and back to map its environment.

The encoding of the incoming rays reflected on the surface is given by:

$$f(x,y) = \frac{1}{2} - \frac{1}{2}(x^2 + y^2)_{for\ x^2+y^2 \le 1}$$

The formula describes the paraboloid information using the image created by the camera (viewpoint) in an orthographic setup.

Notice that no reference to the z coordinate is used, this is because the viewing is along the z axis. In practice this means that the viewing point is set up in the middle of the object looking in some direction. A specific direction is not important since the calculations are in eye space (and therefore looking along the z axis), however usually the environment maps are aligned to the default coordinate system.

The texture lookup for the front face is given by

$$\begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix} = PSM_l^{-1} \begin{bmatrix} R_x \\ R_y \\ R_z \\ 1 \end{bmatrix}$$

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad S = \begin{bmatrix} -1 & 0 & 0 & d_x \\ 0 & -1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} s \\ t \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix}$$

Where R is the reflection vector calculated using the normal and view vector ($R = V - 2(N \cdot V)N$).

2.5 ENVIRONMENT MAP AS AMBIENT LIGHT

Instead of only reflection the environment map can be used as a cheap method for having a better ambient lighting. The Ambient light part for example in the Blinn-phong model can be replaced with this approach. The idea is to take the colors from the environment and use them as a light source. The negative side on this approach is that bright colors have great impact, for there is no light intensity. For example a red ball gives the same light intensity as a red light. But this is acceptable for it gives the impression of a better light environment.

This is usually done in several steps

- **Take an environment map using one of the earlier mentioned techniques.**
- **Blur the map, by down sampling and/or a blur filter like Gaussian blur.**
- **Use the blurred environment map as the ambient source color.**

By down sampling the environment map it gives also a blur effect, because of the linear interpolation during the texture mapping (sampling down and then stretch it up).

2.6 SPHERICAL HARMONICS

With spherical harmonics it is possible to approach accurate diffuse reflection based on its surrounding (light probe). R. Ramamoorthi and P. Hanrahan presented in 2001 a method to use spherical harmonics for the diffuse term based on a light probe[16] and in 2002 a paper was presented by J. Kautz, P. Sloan & J. Snyder for using spherical harmonics as a low-frequency lighting in a BRDF shading model[17].

The definition

Spherical harmonics are basis functions, which can be seen as signals that can be scaled and combined together to approximate an original function[18] over a 2 dimensional unit sphere.

The definition for Spherical harmonics ("real spherical harmonics function y") is:

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

K The scaling factor to normalize the functions.
P The polynomial, from the set of associated Legendre polynomials.

With two arguments l and m. l is the band index from the natural numbers. A slight difference from the original Legendre polynomials is the range, defined: $l \in N, 0 \le m \le l$ becomes the definition: $l \in N, -l \le m \le l$.

The Legendre polynomials are defined in a recursive manner:

$$(l-m)P_l^m(x) = x(2l-1)P_{l-1}^m(x) - (l+m-1)P_{l-2}^m(x)$$

$$P_m^m(x) = (-1)^m(2m-1)!!(l-x^2)^{m/2}$$

$$P_{m+1}^m(x) = x(2m+1)P_m^m(x)$$

And the scaling factor is given by:

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

In [5] and [18] the following simplification is used for the spherical harmonics:

$$y_l^m = (\theta,\varphi) = y_i(\theta,\varphi)$$

$$i = l(l+1)+m$$

This gives a single indexed form of the spherical harmonics and is used later on in the reconstruction/projection of the original function.

Projection

The reconstruction of the original function is approximated with the nth order function:

$$\tilde{f}(s) = \sum_{l=0}^{n}\sum_{m=-l}^{l} c_l^m y_l^m(s)$$

Where the coefficient c_l^m is calculated by integrating the product of the function s, which is a two dimensional function defined over the unit sphere, and the spherical function $y_l^m(s)$.

$$c_l^m = \int_s f(s)y_l^m(s)ds$$

Using the simplification the capped $\tilde{f}(s)$ function becomes:

$$\tilde{f}(s) = \sum_{i=0}^{n^2} c_i y_i(s)$$

$$i = l(l+1)+m$$

To fully reconstruct the function $f(s)$ replace in the band limited function n^2 with ∞.

The orange book[3] provides a detailed way to implement spherical harmonics and also provides the precomputed coefficients based on the light probes available from Devebec[39] and a optimized formula that can be easily implemented in a vertex shader program. The formula presented is based on the use of 9 basis function. Given by R. Ramamoorthi and P. Hanrahan[16]

9 basis functions are enough to reproduce the accurate diffuse reflection term (average error less than 1%).

$$\text{Diffuse} = c_1 l_{22}(N_x^2 - N_y^2) + c_3 L_{20} N_z^2 + c_4 L_{00} - c_5 L_{20} + 2c_1(L_{2-2} - xy + L_{21} xz + L_{2-1} yz) + 2c_2(L_{11} x + L_{1-1} y + L_{10} z)$$

$c_1 = 0.429043$

$c_2 = 0.511664$

$c_3 = 0.743125$

$c_4 = 0.886227$

$c_5 = 0.247708$

c_1, c_2, c_3, c_4, c_5 Are derived constants.

L Are the spherical harmonics coefficients.

N The normal vector $\begin{bmatrix} N_x & N_y & N_z \end{bmatrix}$

For completeness, the coefficients for the "Old town square lighting" are provided here. All the light probes available at Devebec's website were also processed and the coefficients are also available in the orange book[3].

$L_{00} = \begin{bmatrix} 0.871297 & 0.875222 & 0.864470 \end{bmatrix}$

$L_{1-1} = \begin{bmatrix} 0.175058 & 0.245335 & 0.312891 \end{bmatrix}$

$L_{10} = \begin{bmatrix} 0.034675 & 0.036107 & 0.037362 \end{bmatrix}$

$L_{11} = \begin{bmatrix} -0.004629 & -0.029448 & -0.048028 \end{bmatrix}$

$L_{2-2} = \begin{bmatrix} -0.120535 & -0.121160 & -0.117507 \end{bmatrix}$

$L_{2-1} = \begin{bmatrix} 0.003242 & 0.003624 & 0.007511 \end{bmatrix}$

$L_{20} = \begin{bmatrix} -0.028667 & -0.024926 & -0.020998 \end{bmatrix}$

$L_{21} = \begin{bmatrix} -0.077359 & -0.086325 & -0.091591 \end{bmatrix}$

$L_{22} = \begin{bmatrix} -0.161784 & -0.191783 & -0.219152 \end{bmatrix}$

In figure 2.11 some examples are shown using the spherical harmonics applied on the HMI face model. No other lighting is used or any texturing, just the diffuse term calculated from the spherical harmonics is directly used as the final color.

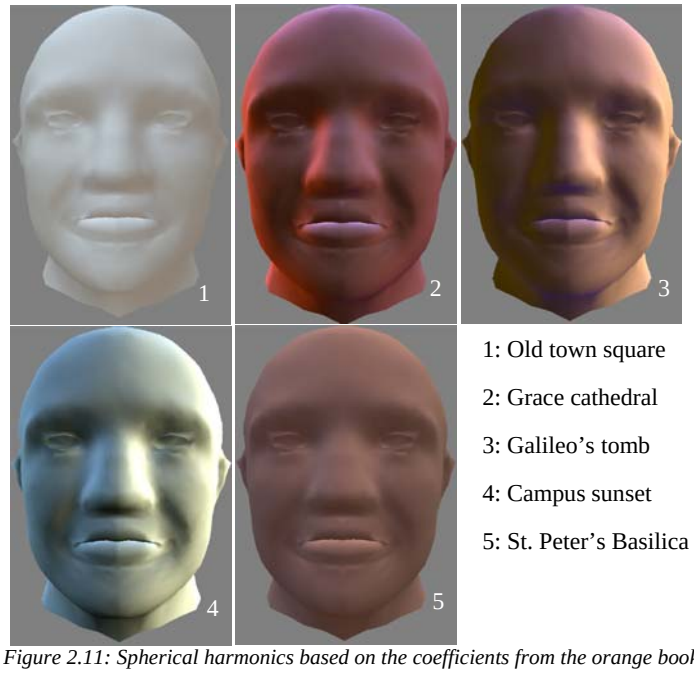


Figure 2.11: Spherical harmonics based on the coefficients from the orange book[3].

2.7 LIGHTING AND TEXTURING

Looking at all the BDRF's there is one thing left out, the surface texturing. With texturing we can simulate a surface appearance by mapping an image to it. As shown previously in figure A.4. Together with a BDRF and appropriate parameter settings (for example the material index) a more realistic surface should be the result. But where does the texture come into play?

First off what is meant by "texturing"? As stated before "simulate a surface" but based on what? Textures are external produced images or procedural generated images. Basically this can be anything as long it is some image loaded into memory and is assigned as being a texture, but there are a couple of well known types of textures. First those that are used for tiling and show a certain material or composition, for example wood, brick, grass, skin etc. These textures are made so, that when they are tiled they fit exactly together and no seams are shown. Well at least the good ones! These textures can be said as being mesh independent since they are used to fit per polygon. The other texture is more or less bound to a mesh. The mesh has a set of ST coordinates and uses the specific texture to get the appropriate color information; the texture is as it were "stretched" over multiple polygons.

Within a BDRF the texture is often used as a replacement or extension to the material diffuse color. This sounds logical, since the diffuse color is the base for coloring an object, but this is not a given fact. There is also a possibility of having multi-texturing, where multiple textures are layered on top of each other. These need to be blended by some function, and depending on the application some layers must be treated different in the BDRF. This still seems to be an open field and is left to the creator's creativity/ingenuity. OpenGL supports several blending methods for multi texturing (Replace, Modulate, Decal, Blend, Add).

In the previous section a texture from the environment is used as a substitute for reflecting the environment, or as a source for the environment light ambient. Other specialized textures might contain other information, for example a specular map which contains values of intensity of specular highlight. Such maps can be thought of as “material” maps as they contain information about the objects material properties.

2.8 CONCLUSION

It can be concluded that it is not clear to say what lighting model and methods can be used best as there are many factors involved alone with the lighting models, and developing and testing them is quite time consuming. There are even more BDRF's all focusing on a certain aspect of lighting interaction. BDRF models like those proposed by Lafortune, Minnaert, Wart, Ashikmin, etc have their strong and weak points. As shown by Havran et. all[19] certain BDRF's perform better for a set of materials then other models. Choosing a BDRF becomes even more difficult with complex objects, as for the head rendering of skin, hair and eyes. The problematic part is that each part can also be constructed in several ways and making the interaction more complex. Therefore instead of choosing one definite way, a rendering system which is more open for several lighting models is taken into consideration. This system is called deferred shading and is discussed in chapter “Deferred shading” on page 35.

3 BUMP MAPPING

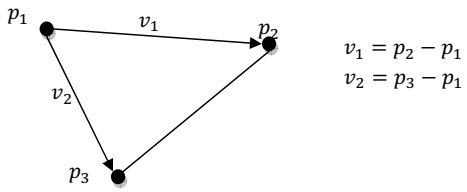
Bump mapping is a way to add details to a surface without adding new vertices[20]. This usually is done on a per fragment basis where for each fragment the normal vector is altered by some function. There are several techniques to do bump mapping, however in this section the two most prominent methods are given. First normal mapping[21] and as an extension parallax mapping based on the article from T. Welsh presented in the book ShaderX3Parallax Mapping with Offset Limiting: A PerPixel Approximation of Uneven Surfaces[11][22].

3.1 THE NORMAL VECTOR

First let's get a better view on this normal vector and how this is retrieved per fragment. This usually involves three steps: the normal vector for a plane (in this case a polygon), then the normal per vertex by averaging the normal vectors from the planes sharing the same vertex and finally the normal per fragment by interpolating the normal from vertex to vertex. However with bump mapping the normal is altered and this can be done by some function or/and a lookup texture containing normal vectors.

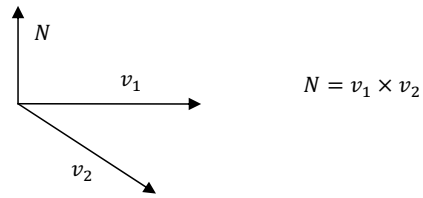
Normal vector per polygon

The best way to calculate the normal vector is from the smallest polygon unit, a triangle. A triangle is defined by three vertices where we can extract two coplanar vectors from.



Figuur 3.1: Calculate two coplanar vectors.

The normal vector for this plane is then easily retrieved by the cross product between the two vectors.



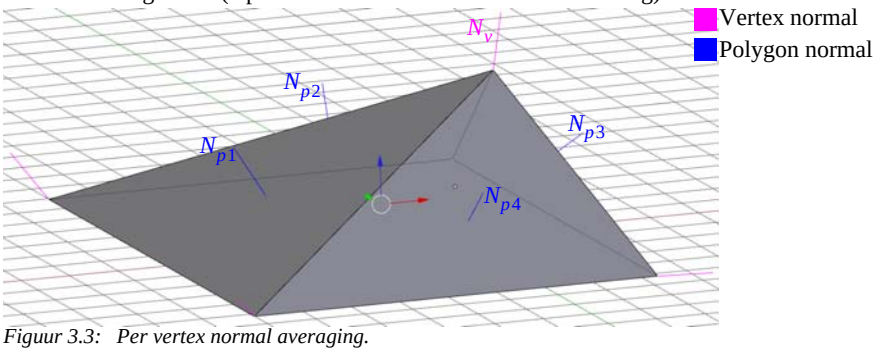
Figuur 3.2: Calculate the normal.

The only thing left to do is to normalize the normal vector $\hat{N} = \frac{N}{|N|}$.

Normal vector per vertex

The normal per polygon is comparable to the flat shading state in OpenGL and gives an object a faceted look. The reflection over the complete polygon is the same. So the next step is to provide

each vertex with the appropriate normal vector by averaging the normal vector and create a smooth shading effect (OpenGL also has a state for smooth shading).



Figuur 3.3: Per vertex normal averaging.

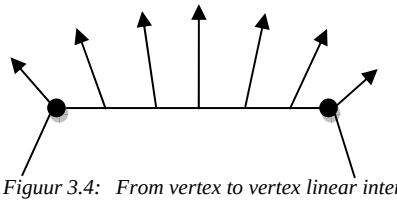
In figure 3.3 the normal vectors for the polygons are denoted by N_p and the normal vector that we want to calculate is N_v.

$$N_v = \frac{N_{p1} + N_{p2} + N_{p3} + N_{p4}}{4}$$

And at last normalize N_v again.

Normal vector per fragment

From the normal vector per vertex to per fragment is a linear interpolation. This is normally done by the rendering pipeline when going from the vextex processor to the fragment processor.



Figuur 3.4: From vertex to vertex linear interpolation.

3.2 LIGHT THE BUMPS USING NORMAL MAPS

We can actually create bumps at vertex level (using the vertex normals), thereby creating certain effects. As they are usually averaged for "smooth shading" a hard edge for example can be created. But let's get to an actual bump mapping technique called normal mapping. A normal map is a texture, an internal frame buffer or some other buffer, but usually a texture is meant, that contains the new normal vectors. In the previous sections where the normal calculation was explained, the normal resided in object space. During normal rendering the normal is moved into eye space in the vertex processor (the normal is given per vertex) and is automatically interpolated within the fragment processor. There the lighting calculation can be done on a per fragment basis (still in eye space). Now the normal comes from a texture and also resides in a certain space. The first possible space is the object space, and another possible option is a local surface space also called tangent space.

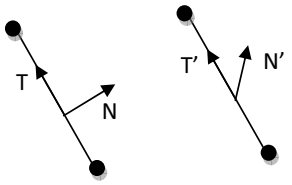
Normal maps and object space

The normal vectors are stored in object space, thus they are pointing in their actual directions and are mesh dependent. Whenever an object translates/rotates (moving into world/eye space) the normal vectors or the view and light vector must be moved into the same space. Thus every

normal vector needs to be moved into eye space, or the view and light vector must be moved in the object space of the object.

The creation of normal maps in object space is usually done in the 3d editor (modeler) , using a high polygon model as reference. The normal vectors that are being stored are actually the vertex normal vectors from the high polygon model. Then the model is reduced to a lower polygon count and the normal map can be applied.

In the fragment shader the given texture coordinates (provided with each vertex) are used for lookup. The retrieved normal vector is then multiplied with the Modelview Matrix (the matrix containing the eye space information). However the modelview matrix is not used, but a special matrix called Normalmatrix. This is done because when the modelview matrix has some non-uniform scaling applied to it, it can deliver a wrong normal resulting in a normal that is not perpendicular to a surface.

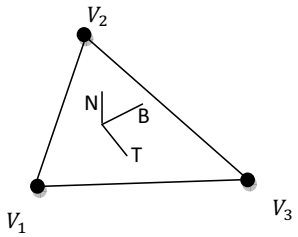


Figuur 3.5: Left the correct Tangent and Normal vector, and on the right a skewed Normal due to non-uniform scaling.

In the image above on the left is the correct normal and tangent given, on the right the incorrect normal due to a non-uniform modelview matrix. To correct this, the normal is multiplied with the transposed inverse modelview matrix $(M^{-1})^T$ [[3]. Note that when there are no special operations on the modelview other then translation and rotation the default modelview matrix can be used (the 3x3 upper left sub matrix needs to be orthogonal).

Normal maps and tangent space

The normal vectors are placed in a surface local space called tangent space. In this space it assumes that every point is at the center $\begin{bmatrix}0 & 0 & 0\end{bmatrix}$ (the origin) and that the unperturbed surface normal at each point is $\begin{bmatrix}0 & 0 & 1\end{bmatrix}$. The view and light vectors must be converted into tangent space using some matrix; this introduces the tangent and bitangent component for each vertex. The combination of the normal, tangent and bitangent vectors is called the tangent space matrix which is a 3x3 matrix containing the rotation and scaling for the tangent space per vertex. First to calculate the tangent and bitangent for a polygon.



Figuur 3.6: Calculate the Tangent and Bitangent vector.

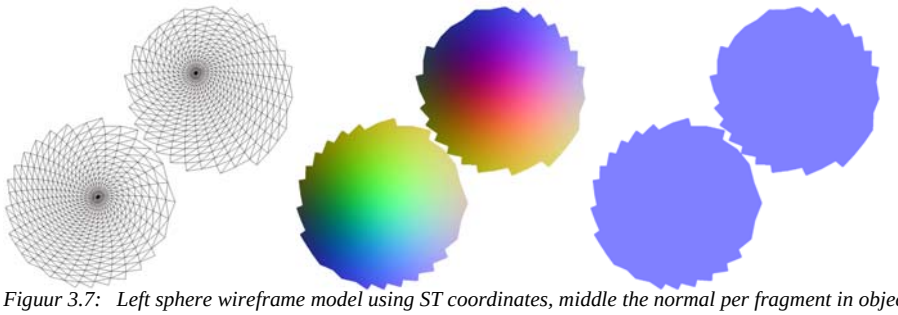
$$E_1 = V_2 - V_1$$
$$E_2 = V_3 - V_1$$
$$E_{st1} = V_{st2} - V_{st1}$$
$$E_{st2} = V_{st3} - V_{st1}$$
$$C = \frac{1}{E_{st1.s}E_{st2.t} - E_{st2.s}E_{st1.t}}$$
$$T = \begin{bmatrix} (E_{st2.t}E_{1.x} - E_{st1.t}E_{2.x})C \\ (E_{st2.t}E_{1.y} - E_{st1.t}E_{2.y})C \\ (E_{st2.t}E_{1.z} - E_{st1.t}E_{2.z})C \end{bmatrix}$$
$$B = \begin{bmatrix} (E_{st1.s}E_{2.x} - E_{st2.s}E_{1.x})C \\ (E_{st1.s}E_{2.y} - E_{st2.s}E_{1.y})C \\ (E_{st1.s}E_{2.z} - E_{st2.s}E_{1.z})C \end{bmatrix}$$

V_1, V_2, V_3 Denote the vertices representing their space position $\begin{bmatrix} x & y & z \end{bmatrix}$.
E_1, E_2 The edge vectors $\begin{bmatrix} x & y & z \end{bmatrix}$.
$V_{st1}, V_{st2}, V_{st3}$ Represent the texture coordinates for the vertices $\begin{bmatrix} s & t \end{bmatrix}$.
E_{st1}, E_{st2} The texture coordinates for the edges $\begin{bmatrix} s & t \end{bmatrix}$.
T The tangent vector $\begin{bmatrix} x & y & z \end{bmatrix}$.
B The bitangent vector $\begin{bmatrix} x & y & z \end{bmatrix}$.

The next step is to average and normalize the tangent and bitangent vectors for each vertex the same way as done with the normal vector. Eventually add the Gram-Schmidt orthogonalization to make the normal, tangent, bitangent vectors orthogonal. Note that the bitangent can also be calculated by taking the cross product of the normal and tangent vector.

The main advantage of tangent space textures is that it is mesh independent, it can be reused. For example can it be used for symmetrical parts of a model (by simple reversing the axis of symmetry) and another usage could be tiled textures for example detail maps (a multi textured normal map, one for distance and one for close up).

Let's see what this in practice means. In figure 3.7 a sphere is used and some unwrapping method is applied to define the st coordinates; this is shown below in the left image in figure 3.7. Then for every pixel in the destined texture the normal vector is saved in color space (this space ranges from 0.0 to 1.0, where the normal usually is from -1.0 to 1.0, thus a small conversion is needed). Saving the normal vectors in object space is shown in the middle figure and in tangent space shown in the right figure. In object space the color varies strongly and in tangent space the color is constant, this directly shows why the tangent space map is useable for multiple objects. Although the object is a sphere the normal in the tangent map stays constant and delivers the same effect.



Figuur 3.7: Left sphere wireframe model using ST coordinates, middle the normal per fragment in object space and on the right the normal vectors per fragment in tangent space.

This makes it easier to work with and to create them. A good programming strategy would be to have the source textures in tangent space and once in the program create object space maps from the tangent space maps; this can reduce the amount of operations. Object space maps are however not practical with dynamic models (animated) and for using the tangent space map the tangent matrix needs to be recalculated (with dynamic models the normal is usually recalculated, and in this case the tangent and bitangent vectors must also be recalculated). Mutation to the tangent map 'can' be done by hand which would be almost impossible in object space.

3.3 CREATING A NORMAL MAP

Here a possible solution is given on how to make dynamic normal maps in Tangent space. The approach is based on the shader available from RenderMonkey[40]. For each fragment the 8 direct surrounding fragments are taken then the slopes for x and y are calculated using the Sobel filter and a normal vector is retrieved. The input can be a diffuse map or a heightmap, as the color difference between the selected fragments is used to determine the slope. Note that this method can result in “undesired” bumps (example incorrect direction) as the filter kernel determines how the slope is calculated. It is not a defintive way on how to calculate dynamic “correct” normals, but it delivers acceptable normals. Here the algorithm is presented:

$$N = \begin{bmatrix} X' & Y' & 1 \end{bmatrix}$$

$$X' = X \cdot L$$
$$Y' = Y \cdot L$$
$$X = S_{00} + 2S_{10} + S_{20} - S_{02} - 2S_{12} - S_{22}$$
$$Y = S_{00} + 2S_{01} + S_{02} - S_{20} - 2S_{21} - S_{22}$$

N The normal vector (still needs to be normalized before storing).
L A control vector for adjusting the intensity of the bump and direction.
S A surrounding fragment color $\begin{bmatrix} Red & Green & Blue \end{bmatrix}$.

Table 3.1: Surrounding fragments used for filtering.

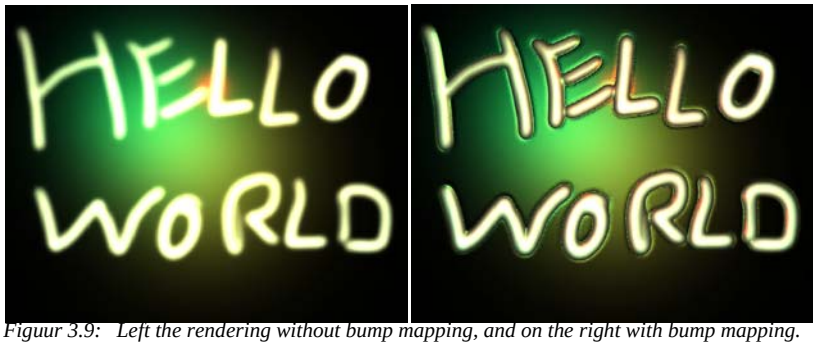
S_{00}	S_{10}	S_{20}
S_{01}		S_{21}
S_{02}	S_{12}	S_{22}

The temporary variables X, Y, X' and Y' are the components.

For creating the normal map using this filter or a similar filter the best source is a heigh map. A height map image contains as the name implies the height at each pixel. Usually this is saved in one channel and can be seen as a grey colored image (notice that the given filter above uses all the color channels). This actually means that creating a bump map is rather easy. By drawing a grey map, which can be done in the actual render program (for dynamic bumps) or "off-line" using a

drawing program (like Gimp). An example is shown in figure 3.8. Using a simple brushed text, convert the image and use it (figure 3.9).

Figuur 3.8: Left the height map and on the right the normal map based on the height map.

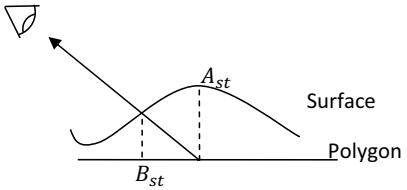


Figuur 3.9: Left the rendering without bump mapping, and on the right with bump mapping.

3.4 PARALLAX MAPPING

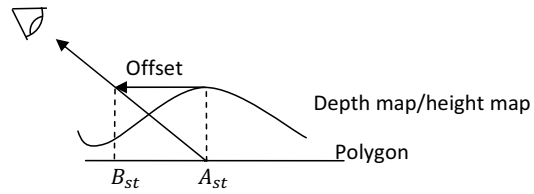
This technique is more and more used in today's 3D real-time programs (mainly games). The main advantage of parallax mapping is the ability of occlusion which is not possible with the normal mapping technique explained in the previous section, hence it is also known as 'virtual displacement mapping' or 'offset mapping'. Where the 'displacement mapping' technique actually moves vertices this technique does not, it actually moves st coordinates instead. The normal map here is optionally, as the normal can be calculated on the fly. The main input for this technique is a depth map (or height map depending on its usage). The depth map is a texture where each pixel is a depth value, this only uses one channel and is often placed in the alpha channel (the RGB channels can then be used for the normal, diffuse color or other information). Parallax mapping can be implemented in several ways, for giving a good view on this techniques it is explained here following the implementation by T. Welsh [22].

In figure 3.10 the problem is illustrated. The surface is rough and a polygon is plain flat and the texture used on the polygon is also flat (by default the st coordinates are linearly interpolated from vertex to vertex). If we look at the polygon we would get the st texture coordinates at point A, however due to the roughness is should be st texture coordinate B.



Figuur 3.10: The problem with a rough surface considering a polygon is always flat.

To approximate the correct st texture coordinate an offset is calculated using the eye vector and the surface represented by a depth map.



Figuur 3.11: An approach to simulate this roughness.

To compute the offset three components are needed. The starting texture coordinate, that is, the interpolated coordinate that would be used by default for mapping a texture to the polygon. The surface height is needed. Here the starting texture coordinate can be used for looking this up in the depth map. And third the normalized eye vector in tangent space, this involves the tangent space where for each vertex the tangent matrix must be supplied (the calculation is the same as shown in the previous section "normal maps and tangent space"). The values in the depth map range from 0.0 to 1.0. To adjust the height a scalar and a bias are used for calibration, $height = H_{st}Scale + Bias$, where H_{st} is the height retrieved from the depth map. The formula to get an approximated new texture coordinate:

$$B_{st} = A_{st} + \left(\frac{height * V_{xy}}{V_z}\right)$$

V is the View vector in tangent space. (In the vertex processor moved into tangent space, and in the fragment processor normalized.)

As can be seen in figure 3.11 the approximated coordinate is flawed, especially for a steep rough surface. This can actually end in an unrecognizable surface; the paper by T. Welsh[22] suggests a simplified "offset limiting" $B_{st} = A_{st} + (height * V_{xy})$. Using this formula the offset can never be larger than the maximum height.

The normal is in no way altered or whatsoever using this technique, the only thing that has been done is moving the st coordinate to simulate occlusion. This is done in addition to bump mapping, so the actual normal perturbations are still to be done by some method. This can be the normal mapping technique, or calculate the normal on the fly using the depth map information.

4

REAL-TIME SHADOW CASTING

Shadow can give visual cues to the position of an object in an environment and adds a great deal of realism. In this particular case for rendering a face, self-shadowing is important. The shadow from the hair, nose and eye socket give more information on how the face is lit. There are several shadow casting techniques possible to implement and each has its strong and weak points[23][24]. Instead of explaining all kinds of shadow techniques only the one used in this project will be discussed, which is shadow mapping. Previous work on shadow mapping within the HMI department has also been done by B. Kevelham[25] in a project for obtaining his Master degree (the project was carried out at MIRALab). A small summary on advantages of shadow mapping are:

- **Has full hardware support (and GLSL support).**
- **Anything visible in a scene can cast a shadow, also with self shadowing.**
- **Dynamic or static scene are handled the same.**
- **It is easy to implement.**

A shadow is created by an object blocking the light rays, this is shown in figure 4.1.

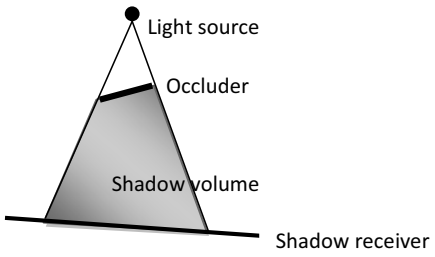


Figure 4.1: Simplified illustration of casting shadows.

Everything in the shadow volume receives no light from the light source because of the occluder. The occluder and the shadow receiver are usually part of the geometry used in the scene. Note the mention of "usually" in the previous sentence; because we are dealing with virtual generated scenes it does not necessarily have to be the geometry that creates the shadow (for example a predefined shadow map texture can be used).

4.1 SHADOW MAPPING

Shadow mapping is a 2-pass rendering method. This means that at least two sequential render passes are necessary to create/visualize shadows. The first one is the shadow pass, where a depth buffer is used to store the depth values from the lights 'viewpoint'. The shadow pass is also mentioned as being the light pass, however although the shadow pass is using light-entity information (translation and rotation) it does not have to do anything with the lighting algorithm itself. The lighting itself can also be a different pass; this is also shown in the next section on Deferred shading.

The second pass is the normal rendering pass and the depth buffer from the shadow pass is used to check whether a fragment falls in the shadow or not by comparing the depth from the current viewpoint with the depth from the light's viewpoint.
fragment in shadow $if\ \ z(light) < z(viewer)$

For the first pass to work we need to render off-screen. This can be done by using the framebuffer object extension. The framebuffer buffer object is set up to only contain de the depth values, thus

a depth buffer is added to the framebuffer. There is no need for any color buffer as it is not needed for the actual render pass. The framebuffer object now easily provides access to the depth buffer by using a texture binding with the logical depth buffer and passing it by a uniform sampler variable in the shader program. The render pass is done as follows:

STEP 1
In the modelview mode (thus using the modelview matrix stack) set the camera/viewpoint to the light source position and rotation. Depending on the light source type the amount of passes to render might differ. A light point can cast shadows in all directions; usually this is solved by rendering 6 times to create a depth environment map stored in a cube map. Note that it also is an option to use the paraboloid technique instead of a cubemap.

For a spot light a single render pass is sufficient, using the rotation from the spot light to setup the viewpoint. And for the directional light type an orthographic projection is used (note that the translation of the light's viewpoint is depended on the view of the 'user' and should cover every region the 'user' also can see)

STEP 2
Rendering the scene from the light position. Since no color information is needed, no textures are needed to be attached/loaded, no normal, tangent, bitangent texture coordinates or any other special vertex information, just the vertex translation information (the projection and modelview matrix). Important is how to render with the use of culling. Normally the back side of a polygon isn't drawn; but in this case for shadow mapping an often used approach is the front side of the polygon being discarded and using the depth value of the back side. Mostly this is done to avoid Z-fighting which happens due to the lack of precision. However self shadowing isn't possible with front-face culling; therefore to solve the z-fighting problem a vertex offset is given. This is a supported function by OpenGL called polygon offset. This displaces the polygon by a given unit and needs some calibration which is depending on the precision of the depth buffer (8, 16, 24 or 32 bit depth buffers are possible, whereby the higher the better the floating point precision value and the smaller the offset can be, if needed at all). The effects of z-fighting is shown in figure 4.2 where on the left shadow glitches can be seen in a region which is fully lit, and on the right a small polygon offset is provided.

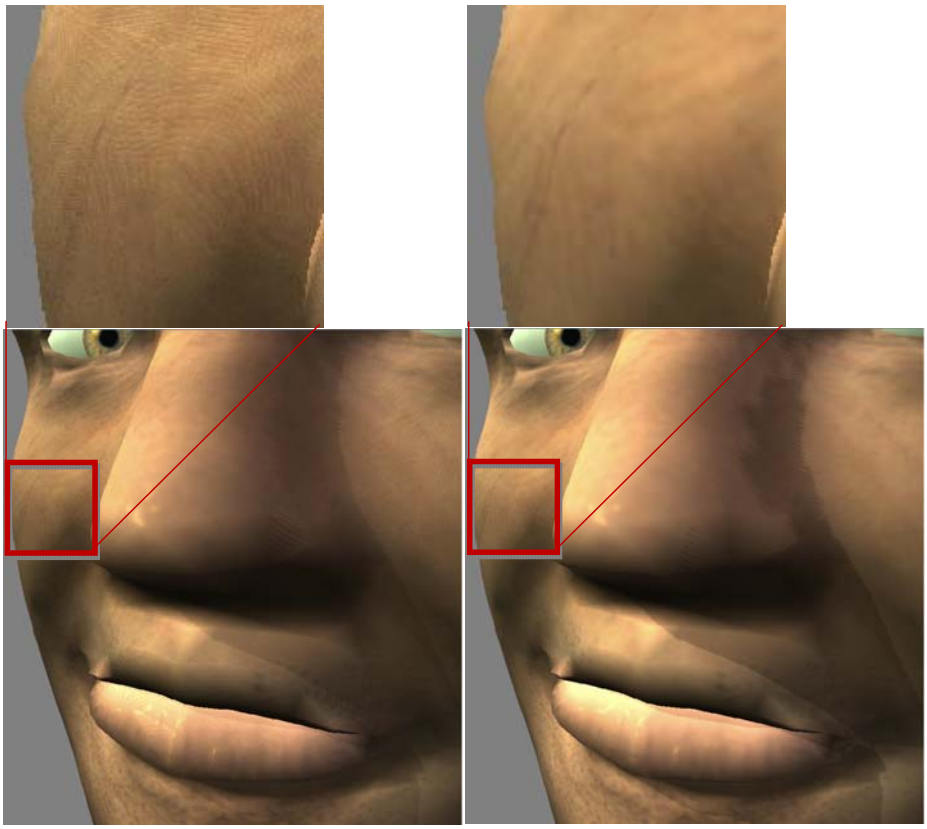


Figure 4.2: On the left there is Z-fighting and on the right a polygon offset is given.

The depth buffer needs to be setup for comparison, in practice this can be done in two ways.

- **Writing the z values in a color buffer using a shader program. Thus not using a depth buffer but a color buffer instead attached to the framebuffer object.**
- **Using the hardware support using a depth buffer attached to the framebuffer object.**

With the hardware support we don't really have to worry about this, thus the normal solution is to use a depth buffer attached to the FBO. However if for some reason it is needed to write the values our self, the depth can be written in the RGB channels by converting the depth value into a value that can be stored in the channels. This is shown in the equation below.

$$D' = Db$$
$$C_r = floor(D')$$
$$D'' = b(D' - C_r)$$
$$C_g = floor(D'')$$
$$C_r = C_r\frac{1}{b}$$
$$C_g = C_g\frac{1}{b}$$

D The depth value for a given fragment. Which is usually the z value in view space.
b The color bit depth, which is normally 8bit per channel, the value here is then 256.
C The resulting color vector $C = \begin{bmatrix} R & G & B \end{bmatrix}$

By reversing this process the depth is retrieved again.

For the hardware support the depth is automatically written in the buffer and also follows the depth test function. In figure 4.3 a sample is given of what a depth buffer looks like when it is

used as a texture. The figure on the left gives the normal "colored" result of the rendering, the middle figure the original depth texture and on the right the contrast has been slightly enhanced to make it more visible.

Figure 4.3: Left: final image, middle original depth buffer and on the right an enhanced version (higher contrast).

Note that when using the depth buffer, that has been set up for comparison (which is needed for shadow checking), as a texture the result is always blank. The image from figure 4.3 shown as the depth buffer has not been set up for depth comparison. This has been tested on an NVidia (6800gt) and ATI (x700) card and both only show the depth when they are not in comparison mode.

Now we have the depth written in the buffer, the next thing we need is the shadow matrix so that we can move a vertex into the shadow space. To get the appropriate depth value from the depth buffer, the texture coordinates must be calculated for the depth which corresponds to the current fragment.

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \begin{bmatrix} 1/2 & 0 & 0 & 1/2 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} light \\ projection \\ matrix \end{bmatrix} \begin{bmatrix} light \\ modelview \\ matrix \end{bmatrix}^{-1} V$$

V is the vertex in world space (the space between object and view space).

$$V = \begin{bmatrix} world \\ modelview \\ matrix \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

The shadow map is then accessed using $\left(\frac{s}{q}, \frac{t}{q}\right)$. The value that is retrieved from the shadow map is then compared with the current depth value.

$$\text{fragment in shadow } if \left(\frac{r}{q} > shadowMap\left(\frac{s}{q}, \frac{t}{q}\right)\right)$$

If a fragment is in the shadow, it is up to the programmer what to do next. An often used approach is to use a value < 1.0 value to darken the current color. For example using a value of 0.5 still gives the underlying color, where 1 is fully lit and 0 complete black.

4.2 SOFT SHADOWS USING SHADOW MAPPING

The shadows seen in figure 4.2 and figure 4.3 are still very hard/crisp which is not really natural. A solution for this is introducing a smoothing algorithm as an extension. Again looking at a

shadow there are two regions of shadow, the umbra (latin: shadow) which contains the full shadow and the penumbra where the light is only partially blocked. This is shown in figure 4.4.

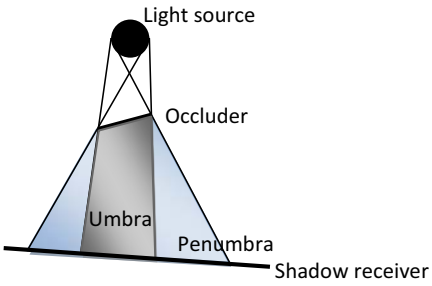


Figure 4.4: The shadow divided into the umbra and penumbra regions.

To simulate this effect with shadow mapping multiple solutions are possible. The described approach here is based on multiple sampling. By averaging surrounding depth comparison results, the final shadow intensity is determined. In the previous section only for the actual fragment the depth was compared, but with a simple extension it is able to smooth the shadow. The algorithm used is as follows:

A single smoothing run takes for each fragment the 4 depth comparisons. The offset is a minor value to select the depth next to the actual fragment.

Tabel 4.1: Soft shadows filter

$-offset, offset$		$offset, offset$
	actual fragment	
$-offset, -offset$		$offset, -offset$

The resulting values are summed and then divided by 4, to get the average. With the average we then can adjust the overall color for the current pixel (for example *diffuse*shadowIntensity*)

This can be repeated by taking multiple and greater offsets. At each loop the offset can be set by a base offset size resulting in *offset=base*loopIndex*. And the divider to average the intensity becomes is the amount of loops multiplied with 4 (the count of depth comparisons). The results

from this approach can be seen in figure 4.5, where on the left the "hard" shadow is shown and on the right the soft shadow.

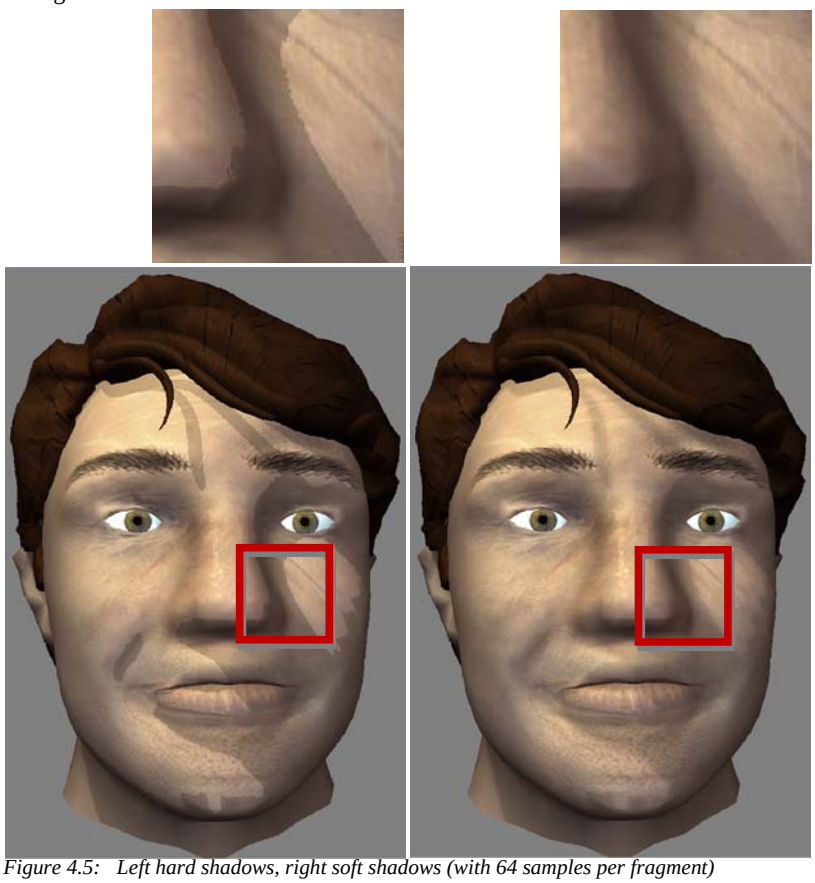


Figure 4.5: Left hard shadows, right soft shadows (with 64 samples per fragment)

Extension or other approaches to this method might be:

- **Create a shadow texture, and then use a blur filter.[26][41]**
- **Use jittering sampling, instead of taking with each render cycle the full amount of samples per fragment and only multisample at the borders of the shadow.[43][44]**
- **Use depth from fragment to light source to determine the amount of samples to be taken. The effect is that shadows that are close to the source of light are appear sharper and shadows created further away take more samples and therefore are more smoothed.[27]**

5 DEFERRED SHADING

Defer means "to delay", and that is exactly what is being done in this shading method or better said rendering system. In the previous section on shadows a 2 pass rendering sequence was needed to create shadows and the same principle is used here. Multiple passes are done to compose the final image using the textures created from the previous passes for retrieving color, material properties and other information, for example forumula parameters.

In its most basic form this method consists of two render passes, the material pass and the lighting pass. The material pass is used to store several properties, such as normal vectors, diffuse colors and specular information, in several colorbuffers. The lighting pass is then used to apply per fragment illumination using the information from the material pass. The idea behind this approach is the separation of lighting and geometry and supports an undefined amount of lighting entities in a scene with the possibility of each having it's own lighting model. In practice this means that instead of having one "huge" shader program combining all functionality, it is separated into smaller shaderprograms each with its own specific functionality.

The amount of passes can be extended with a post process pass where we can apply some 2D effects on the image, aswell other passes can be added. Think of a shadow pass and an environment mapping pass, but these passes render the scene from a different perspective and therefore not directly related with the Deferred shading system. The sequence of passes for deferred shading is shown in the figure 5.1.

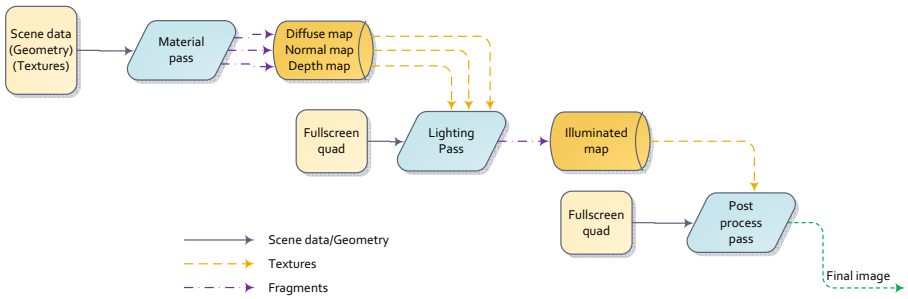


Figure 5.1: The several possible passes in a Deferred shading system.

In this section this system and its passes will be explained in more detail.

5.1 THE LAYOUT OF THE BUFFERS

For each pass a distinct frame buffer object is attached to be able to render off screen. The final image can be extracted from the last lighting pass or from the post process pass if used. This means that the last pass does not necessarily need to be an off screen pass. For example if we let the Post process pass directly write to the front-left. Further optimalization is to use only one frame buffer object and to detach and attach the several logical buffers, but this is not always possible. For example when using a 16 bit (or 32 bit) color buffer for a BDRF model instead of a normal 8bit. This is still a bit restrictive with frame buffer objects, further more they all need to be the same resolution. These are however implementation issues and depends on the shader programming language and hardware support.

5.2 MATERIAL PASS

The material pass writes all relevant properties into buffers that subsequent passes need. The Frame buffer object (FBO) which is the underlying technique for off-screen rendering is also called the Geometry-buffer or G-buffer in this context. This is because it is the only actual pass where the scene geometry (vertex data) is pushed to the graphics card. The G-buffer contains several color buffers and a depth buffer, and as said before this is dependent on the application, or better said dependant on the need of data in subsequent passes.To illustrate this, an example for the usage of the material pass is shown in figure 5.2. Here a human model, a landscape and some water surface mesh is being rendered, each using its desired (material)shader. These material shaders write their output into the G-buffer, which is in this case a diffuse map (colorbuffer 0), a normal map (colorbuffer 1) and a depth map(figure 5.3).

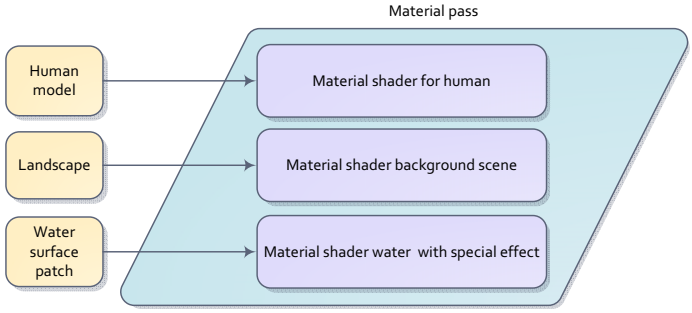


Figure 5.2: Example on the use of the material pass.

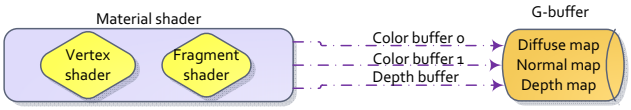


Figure 5.3: The material shader.

In the figure 5.4 the outline is given for the G-buffer in more detail, which directly shows that the G-buffer is a name given to a FBO. The color buffers are defined by a 32 bit plane and are divided into 4 components(red, green, blue and alpha). The depthbuffer is using a 24bit plane. Note that it is not necessary to have this layout, as the next passes determine what information should be stored best. The information here is enough to get a per fragment based phong-blinn lighting model.

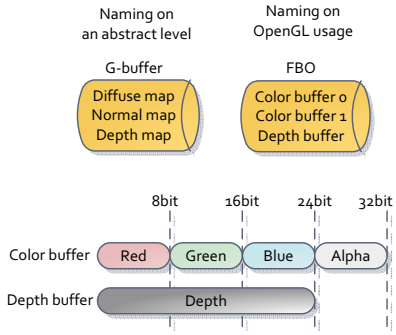


Figure 5.4: The geometry buffer storing several properties for later use.

As the G-buffer is only a name for being a container of “geometric data”, the same applies to the “diffuse map” and the “normal map” which are named after what is stored in a buffer. The RGBA values within a colorbuffer don’t need to be interpreted as colors and can be used to store other data as already mentioned earlier with normal mapping. Which stored the normal vector in

colorspace. The fragment shader writes the information into the color buffers as shown in figure 5.5.

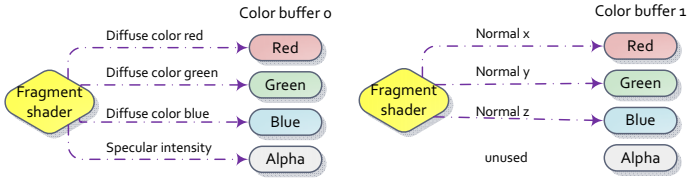


Figure 5.5: The fragment shader writes the data into multiple buffers.

Color buffer 1 from figure 5.5 has one unused component (the alpha component) and could be used to store additional data. The reason for having it unused (exept from the fact of having no data to store in it) is because the colorbuffers need to have the same bit plane (here 32 bit). This is currently an implementational issue and might be solved in the future (newer hardware and OpenGL) but it gives a good sense on what and how to store data, especially if precision is important. Each component here is only 8bit, so for the normal vector each coordinate is stored in 8bit precision (8bit in integer is a range of 0 to 255). Higher precision can be achieved by using 16bit components (or even 32bit), but then all the color buffers need the same precision and memory or process performance might suffer. This is however also an implementation issue and hardware/OpenGL related.

The depth buffer can be filled in a previous pass, which implies pushing the scene geometry twice to the graphics card. This is sometimes done to restrict the more complex calculations in later passes only to those fragments that pass the depth test. This is also called early Z culling. In this project however the scenes are not that advanced as it only contains the HMI face model; this pass is however also mentioned by Policarpo [28].

To give a better sense of what is stored in these buffers, the content of each buffer is visualized in figure 5.6, what normally is rendered in off-screen and accessible as texture. All information that

is being stored is from the eye space (modelview) and is used in the next pass, which is the lighting pass.

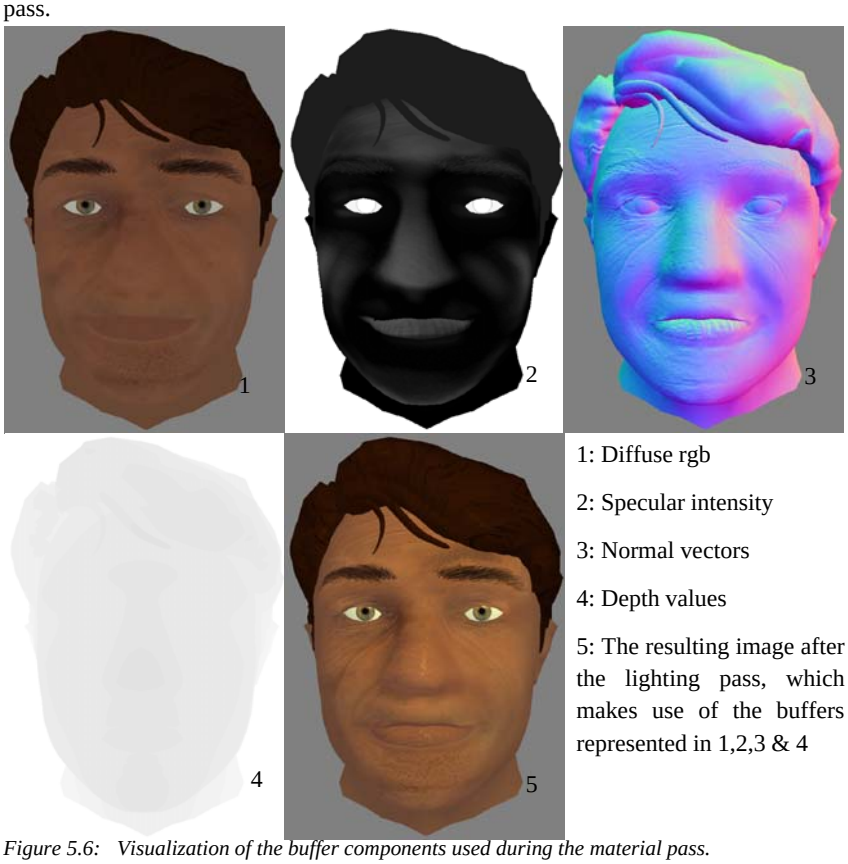


Figure 5.6: Visualization of the buffer components used during the material pass.

5.3 LIGHTING PASS

The lighting pass can be seen as layering the several light entities on each other. Each light uses the base diffuse buffer (figure 5.6 depicted with number 1), using the normal map (figure 5.6, nr 3) for getting the correct angle for diffuse and reflective/specular lighting and for regulating the specular highlights the intensity map (figure 5.6, nr 2) is used. If some lighting model uses other information that is needed and only available at the material pass, then the material pass must be extended with another color-buffer to store this information. In this project the following lighting scheme was used. The first layer served as the ambient lighting. The ambient lighting used was the spherical harmonics which are added in the diffuse buffer, as they are calculated at a per vertex basis, rather than per fragment. The layers placed on the ambient layer are the actual lighting models that work on a per fragment basis. The fragment shader sends his output to the colorbuffer

which is color buffer 0 in another FBO where it is called the “illumination map”. An example of the lighting pass and the fragment shader process are shown in figure 5.7.

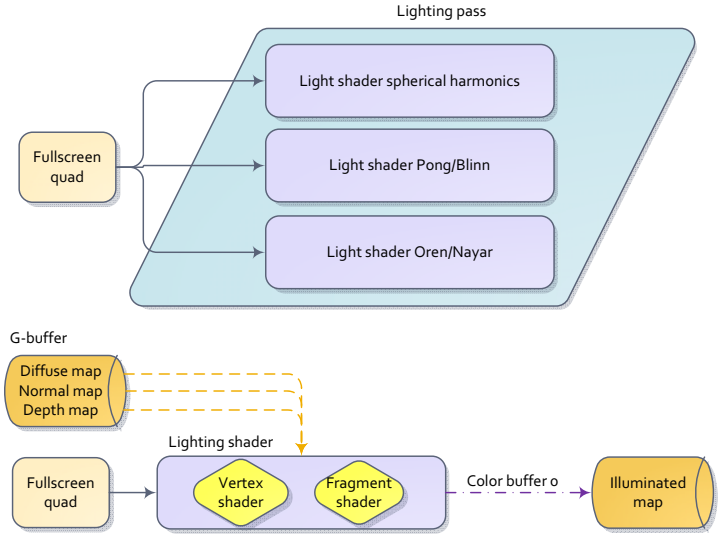


Figure 5.7: Example of the lighting pass with three lighting shaders active.

Each layer is a full screen quad, where the first is used to clear the target color buffer and the subsequent layers are blended onto it. This so called full screen quad is a simple quad polygon created with the x and y coordinates -1 and 1. Without using the modelview or depth (z coordinate) information the vertices are placed directly into clipping space. This way the quad always stretches over the complete screen as the screen ranges from -1 to 1 and can be used to draw on. figure 5.8 shows this, where the normal quad is stretched on screen. In the figure the stretching is a bit overdone just to illustrate that it is being stretched. To avoid the stretching a normal procedure is to set an appropriate scaling for the OpenGL viewport (for example an 4:3 horizontal/vertical scaling) to get the appropriate proportions.

Rendering to an off-screen buffer is the same as default rendering. Therefore setting a projection matrix for offscreen rendering is still needed. Otherwise incorrect scaling occurs.

Figure 5.8: Stretching a quad polygon on screen.

To illustrate the effect of layers, a scene with two light points is used together with an ambient setting. The first layer shown on the left in figure 5.9 represents the ambient light layer, also the background color is shown here (light gray). The middle image in figure 5.9 shows the effect of a light point with a "large" radius (as it covers the whole face region). The background here is depicted as white, however it isn't touched by the lighting model (because of depth testing). Only fragments that fall within its radius of effect are updated in the buffer. This is clearly shown in the

right image where a red light point with a smaller radius is placed. Fragments that are not effected by the light are discarded, and thus not updated in the color buffer.

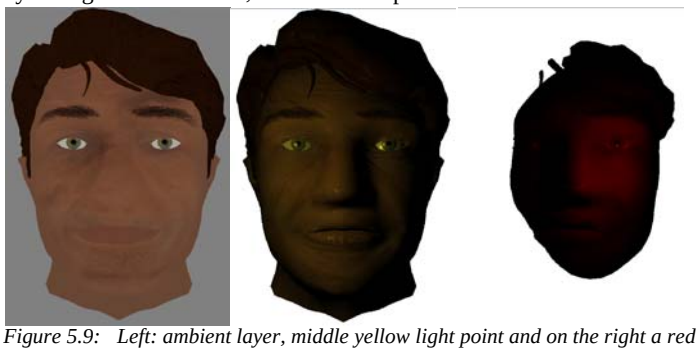


Figure 5.9: Left: ambient layer, middle yellow light point and on the right a red light point.

These three layers are then merged together using a blending function available in OpenGL. The first layer is used as a base (writing to all fragments in the target buffer) and for each fragment in the other layers its color value is blended using the “add” blend function, the final result is shown in figure 5.10.

Figure 5.10: Left: final image after blending all lights together.

This pass can be rendered off-screen or directly to the front buffer for rendering on screen.

The lighting pass seems rather easy, but there are some tricky points in its use. As there is no geometry from the scene sent to the graphics card in this pass some information is not available anymore. The fragments position or surface position in view space are not available anymore but are still needed for the lighting calculations. Also the view vector is not directly available anymore. As shown in chapter 2 Lighting models on page 5 the view vector and fragment positions are used in lighting models for example to calculate the reflection vector and attenuation value. As a reminder, the view vector is pointing in the direction of the viewer from the fragment. To retrieve the fragment’s position there are two ways to solve this problem. The first one is to store the fragment’s position (just like the normal vector) during the material pass; this would imply an extra color buffer. A second solution would be to recalculate the fragments position using the view vector and depth buffer. Another issue with storing the fragments position is the precision. An 32 bit buffer plane may cause precision errors and using a higher bit plane increases the memory hit (another let down is the restriction of equal bit planes among color buffers attached to an FBO, but this is a more technical problem). The conclusion is that recalculation seems the most logical option.

First the view vector needs to be available again at each fragment. Normally the view vector is provided in the vertex shader, where for each vertex the view vector is calculated by normalizing the multiplication of the incoming vertex with the modelview matrix. After that the view vector is interpolated from vertex to vertex, so that it's available in the fragment shader. Now we have to provide the view vector our self. To do this we calculate the view vectors at each corner of the screen (the OpenGL window). In an optimal static environment the full screen quad could be used

to retrieve the view vectors using the four vertices. But that is not the case as the quad is stretched in the vertex program by simple discarding the modelview and depth information. The solution is to calculate the view vector at each corner using the window coordinates. OpenGL provides a utility function for this called gluUnproject, which is able to take window coordinates and returns view space coordinates. This function takes a window coordinate, the modelview matrix and the projection matrix as input. Since the view vector itself is in view space, the modelview matrix can be equal to the identity matrix and only the projection matrix used for the current camera is used as input. The resulting view space coordinates then only need to be normalized to become the view vectors. The view vectors only need to be updated whenever the projection changes (thus whenever the height/width ratio of the OpenGL window changes). When drawing the quad the view vector is provided to the vertex shader as attributes. Note that the view vector now is being approximated to its original, and might differ slightly because of the linear interpolation.

In the fragment shader we have the view vector per fragment and using the depth buffer from the material pass the position of the fragment can be calculated. The depth buffer isn't linear in depth but is packed towards the near plane. I.E. the precision of the depth buffer is higher towards the near plane. This is also called Z-distortion [2][37]. This effect can be explained by looking at the perspective projection matrix.

$$P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$z' = \frac{-(f+n)}{f-n} + \frac{-2fn}{(f-n)z} \Rightarrow 1 \text{when } z = n \text{ and } -1 \text{ when } z = f$$

If $|z|$ becomes larger, then the variance of the neighboring z values becomes smaller.
l, r coordinate for the left and right vertical clipping planes.
b, t coordinate for the bottom and top horizontal clipping planes.
n, f distances for the near and far depth clipping planes.

To solve this effect a small conversion is needed to get a linear distribution of depth values which is then used as the fragments z coordinate, and together with the view vector the x and y coordinates are calculated.The depth retrieved from the depth buffer is the stored z' in the material pass.

$$a = \frac{f}{f-n}$$

$$b = \frac{fn}{f-n}$$

$$z = \frac{b}{a + depth}$$

$$x = \frac{V_x}{V_z z}$$

$$y = \frac{V_y}{V_z z}$$

n, f distances for the near and far depth clipping planes
$depth$ the depth value taken looked up in the depth buffer from the material pass.
x, y, z The coordinates for the fragment in view space
V the View vector $\begin{bmatrix} V_x & V_y & V_z \end{bmatrix}$

After recalculating the fragments position, the light model algorithm can be applied.

5.4 POST PROCESS PASS

This pass is completely optional, but here some nice effects can be added. From simple effects like adjusting saturation, contrast and color balance to blur effect (motion blur), bloom and HDR effects. The effects can be seen as 2D filter effects. The effects can also be layered just like the light entities in the previous pass. This way each effect can have its own shader program, which again gives a good separation of the distinct functionality. Some examples of these effects are shown in figure 5.11.

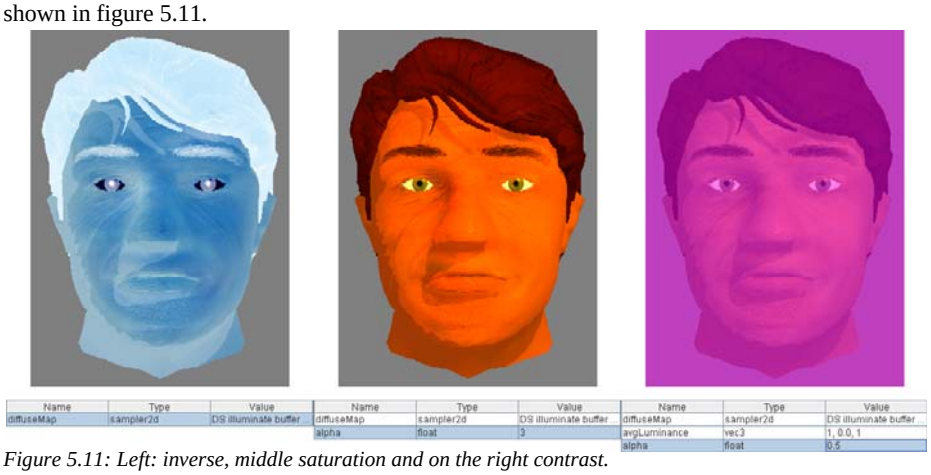


Figure 5.11: Left: inverse, middle saturation and on the right contrast.

5.5 CONCLUSION

The greatest advantage of this system is the separation of shaders, and the possibility for shader layering. This makes it great for creating new shaders and trying combinations of effects and testing BDRF's and can support an unlimited amount of light entities in a scene (where default OpenGL built-in light is limited). Deferred shading gives a good, clean, and flexible render system.

On the other hand, this method has the disadvantage that it is computationally expensive, due to its multiple buffers and performing everything on a per fragment basis one needs a high-end graphics card with a high fill rate. On a NVidia 6800GT the framerate was about 30fps (resolution of 1024x1024) with 3 dynamic per fragment lights, compared with a non-deferred system the framerate was about 90fps(resolution 1280x1024). However the used video card is already two generations old (as of writing the NVidia 8X00 series are available), and it should be noted about every 8 month a new graphics card is released, pushing the limits even further.

After implementing and working with this system, I came to the conclusion that an optimised non-deferred system greatly outperformes this system. However for post processing and specialised lighting a hybrid system or full deferred system seems to be optimal and in many high end applications (games using for example HDR lighting) this is already used.

6 FACIAL RENDERING

This section discusses the facial rendering of the HMI face model. The HMI face model developed by The Duy Bui is a muscle based 3D face model. The aim for the facial rendering is to enhance the appearance of this model using new render methods (today's possible render technology in real-time environments). With a preference to make it more realistic, several methods are discussed from rather simple to more complex models. The face is separated into several sections, skin, hair, eyes. Also the underlying muscle system is discussed as this becomes intervened (at implementation level) with certain render approaches.

6.1 THE HMI FACE MODEL

The HMI face model was developed by The Duy Bui during his Ph.D at the University of Twente[12]. The model focuses on the ability to create expressions using a muscle system. A quote from the thesis points this out.

> *"A face model that allows high quality and realistic facial expressions, which is sufficiently simple in order to keep the animation real-time and is able to assist the muscle model to control the deformations."*
>
> *The Duy Bui[12]*

The Face mesh

This section handles in short the architecture behind the face model as it is available at the HMI section. The head model from The Duy Bui's work is constructed using a base model (a mesh) defined in a "custom" format. This directly complicates the way on how to improve the model as it is not a standard format. The widely known formats such as Wavefront OBJ, Blender .blend, 3D max .3ds etc are also no standard formats, however they are in most cases good interchangeable formats with many 3d applications. An open standard called Collada (COLLAborative Design Activity; .dea extension) is also available, which is an interchange file format for interactive 3d applications and supports a very broad set of functionality (even including physics, shaders programs and other customized parts). The emphasis here on the custom defined format is because it limits the actual usability of the head for certain graphical operations. Improving the appearance is done by combining current rendering techniques.

The face model contains a mesh and a B-spline surface for the lips[12]. The mesh only contains half of the face and is mirrored during the loading of the face in the program (created by The Duy Bui). Other properties of the face mesh that need to be noted are:

- **The mesh is build out of only triangles.**
- **Vertices are ordered into specific regions and a fixed order (a muscle influences the vertices within a region).**
- **The program adds several "glue vertices" during mirroring the face mesh ("hardcode" adding vertices).**

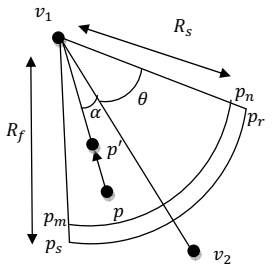
The use of regions is stated as improving the performance of the facial muscle model. The lips based on a B-spline surface provide a smooth lip model. The B-spline is distorted by the muscles and is "polygonalized into a predefined topology of vertices and triangles".

Muscle system

The muscle system is based on an improved version of Water's muscle model. Describing three types of muscles:

- **Muscles that pull**
- **Sphincter muscles that squeeze**
- **Sheet muscles (parallel fibers, lying in flat bundles)**

Waters' muscle model has a directional property and can be seen as a vector and also has a "zone of influence" or "area of effect". The Water skin vector is formulated as follows:



Vector modeled form v_2 to v_1.
R_s and R_f are the falloff radius (start and finish).
p' the new location of vertex p.

$$p' = p + kr\frac{pv_1}{\|pv_1\|}\cos\alpha$$

$$r = \begin{cases} \cos\left(1 - \frac{D}{R_s}\right) \text{ for } p \text{ inside sector } (v_1 p_n p_m) \\ \cos\left(\frac{D - R_s}{R_f - R_s}\right) \text{ for } p \text{ inside sector } (p_n p_r p_s p_m) \end{cases}$$

$$D = \|v_1 - p\|$$

k is a constant representing the skin flexibility.

Figure 6.1: Water's vector muscle.

The conversion

One of the goals of this project is to use the rendering techniques and the muscle system independent from each other. If possible we would like to use the muscle system on other face models. The face model lacks information needed for the certain rendering techniques, such as the texture coordinates. Also the addition of "hard coded" vertices during runtime to the model makes it difficult to extend the model. And because of the mirroring action the normal vectors were all pointing in the wrong direction. In the original program this was not a problem as back face culling was disabled, and with the default OpenGL lighting schema this is no problem, however with custom lighting models this will be a problem. It becomes clear that the model isn't useful in this form. To solve this problem the model was converted into a more appropriate mesh format. During the rendering of the face model in the original program, the face was written to a new file format. This way the complete model data including the mirroring of the half-face mesh and the hard coded vertices is stored. The format used was "Wavefront object format" (short with the extension .obj). This format is a simple textual format and can be read by almost any 3D modeling package. The head mesh at this stage is freely editable in a 3D program (Blender was used in this project).

The biggest problem that came up was the existing muscle system, which also was pretty much hard linked to the original face model and could not be used with the new mesh file. Making the muscle system modular and mesh independent became very hard without breaking the muscle system. An attempt to convert the muscle system failed as it took too much time. The focus hereafter is more on the render system and not on facial animations.

6.2 RENDERING THE SKIN

Rendering the skin of the face mesh can be done in many ways, from simple texturing to complex subsurface scattering methods. In this section the focus lies more on the simpler techniques(less performance intensive), and on the use of the previous discussed techniques (light models, bump mapping, shadowing)

Basic skin rendering

The most basic skin rendering would be to not using textures at all. In the original HMI head program the material color was used to set the skin color. But as we have seen with the wood texture it can give a lot of extra information on the appearance of a surface. Mix this together with bump mapping and a light model that takes advantage of the extra information a highly detailed skin surface can be constructed. The question here is how and what kind of texture to apply to the head mesh. In the section The conversion op pagina 44 it stated the extra need of adding texture coordinates to the head model. Usually this is a 3D model designers job. The 3D model is constructed in a 3D program like Blender, Maya or 3D Studio max, where also the texturing is applied, and material settings are made. Moreover customized shader programs can be used within these applications to make the 3D modeler program an all-round set for any 3D task. As for the HMI head, blender was used to assign new texture coordinates for a base diffuse map. The process of texture coordinate assigning is not always an easy thing to do especially if you are not familiar with 3D modeling.

Diffuse mapping

The texture coordinatemap (or “UV map”) for the HMI head is shown in figure 6.2 on the left. It has been separated into two sections, the front is enlarged as this needs to have the most detail and the back side is scaled down to a minimum (to the lower right in the figure of the UV map). Using this layout the coloring of the face can be done. The coloring of the diffuse map can be drawn by hand, use existing textures and modify them or create the map from photo's or some algorithm that is capable of creating such a texture. Here a texture from an NVidia demo, which comes with the NVidia SDK[42], was used and modified to fit the UV map. The result is shown on the right in figure 6.2. The base diffuse map contains the colors covering the complete face and can be seen as simulating the epidermis layer of the skin. Other models propose a multi layered skin in order to simulate the lower levels of the skin (dermis and hypodermis) and to simulate subsurface scattering[29].

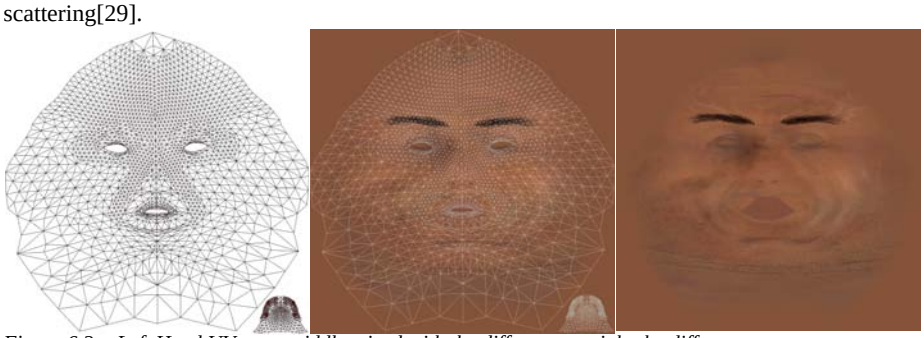


Figure 6.2: Left Head UV map, middle mixed with the diffuse map, right the diffuse map.

Bump mapping

The next step is to add the other techniques such as bump mapping and controlling the specular highlights. As stated before a normal map can be created by a 3D modeler program using a high polygon model, using it as a source for creating the normal map which is then applied to a lower polygon model. But this is not really applicable for the face model as it is already a fairly low polygon model. However another option is possible: draw the normal map yourself. Actually the best thing is to do is to use an algorithm that takes a source image and translates it into a normal map (in tangent space). Such an algorithm takes for each pixel the surrounding pixels, looks at their color, and then calculates(based on several parameter settings) the slope and retrieves the normal from it. It is basically the same approach as given in section 3.3 Creating a normal map op

pagina 25, but by using NVidia normal map plug-in [42] there are more control parameters and different filter kernels. The plug-in[42] is available for Adobe Photoshop at their developer site.

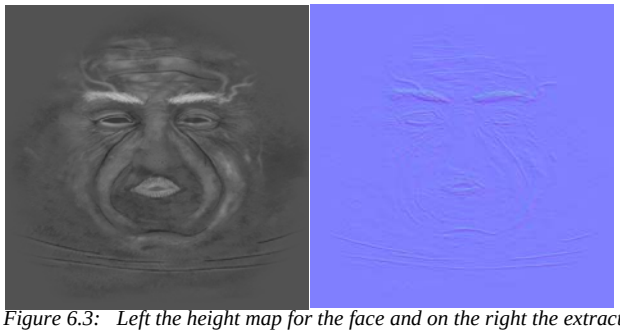


Figure 6.3: Left the height map for the face and on the right the extracted normal map.

The height map in figure 6.3 contains all the detailed information about the face structure.

The resolution of the height map texture restricts the quality of the facial information. For example a real close up to the skin still doesn't seem like real skin, as you can't see the pores in the skin. The question is wether this is important depends on the application, but still it lacks some visual cues which impairs the realism of the model. There are proposed rendering methods that take account of the skin texture at this level by taking real skin patches or by simulating the skin structure by some algorithm. A. Harot, B. Geunter and I. Essat[30] propose such a method by taking skin patches. In their article the face is divided into several regions where different skin patches are used. The patches are taken by a laser scanner and then modified into "seamless" texture patterns. Also noticeable is the use of the Lafortune shading model[36] instead of the default Pong/Gouraud shading model. The process of dividing the face into multiple regions not only provides unique (more realistic) skin textures for certain regions, it also is placed on smaller facial regions and therefore the texture can provide more detail for that region. In the article on simulation of static and dynamic wrinkles of skin by Y. Wu, P. Kalra and N.M. Thalmann[31] the skin is reconstructed using a Delaunay triangulation algorithm. Here the 2D texture is triangulated using the Delaunay algorithm. The edges of the triangles are modified in size to get different kinds of skin textures. The size is important for creating the normal map based on the texture. The method proposed uses multiple layers with different skin structures combined to create a final skin structure. These two methods provide highly detailed skin information. Creating and using such a model can be time consuming and difficult. A more simplistic approach with a similar effect is by using a noise algorithm (for example a 2D Perlin noise function). The reason for mentioning this method is because of the support of noise functions in the OpenGL Shader language. Combining this with the normal map creation presented in 3.3 Creating a normal map op pagina 25 delivers the results shown in figure 6.4.

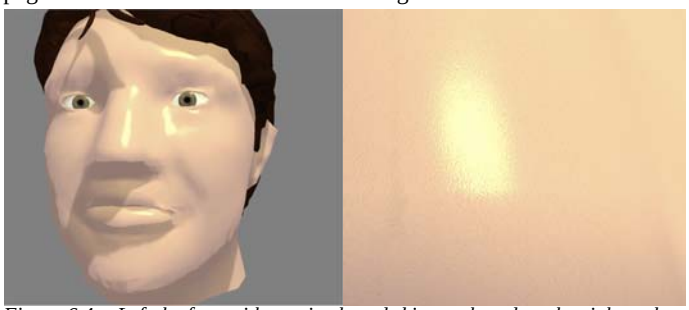


Figure 6.4: Left the face with a noise-based skin patch and on the right a close-up reflective part showing the skin structure (notice the light dithering).

In figure 6.4 the skin patch is applied to each triangle of the face, the result of this is a high detail texture used on every triangle. The downside of this approach in contrast to the first skin texturing method is that "large scale" facial details are lost. As can be seen in figure 6.2 even the eye-brows

are contained in the skin texture which are not visible anymore in figure 6.4. To resolve this, the best way is to have multiple texture coordinates and use some texture blending method for the skin texture. As the normal maps for both skin texturing methods are in Tangent space, they can be easily added and normalized to get the new normal vector. This is shown in figure 6.5 where an additional noise normal map is applied.

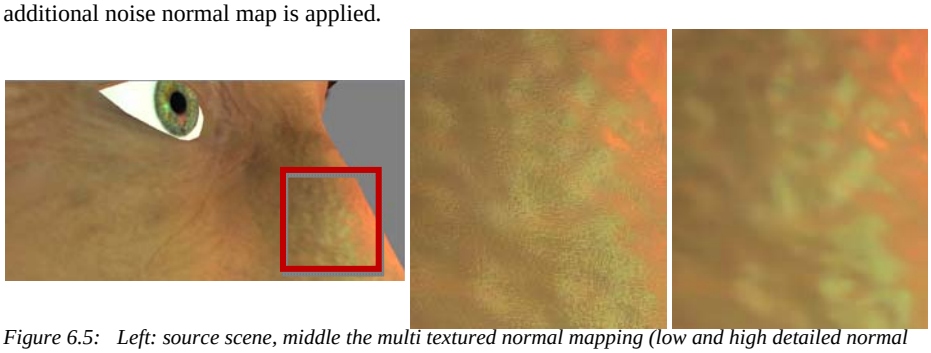


Figure 6.5: Left: source scene, middle the multi textured normal mapping (low and high detailed normal map) and on the right the single "low" detail normal map.

Specular mapping

The idea of specular mapping is to control the specular highlights over a mesh. In figure 6.6 the specular map is depicted. The map contains in each pixel a value for the specular highlight intensity.

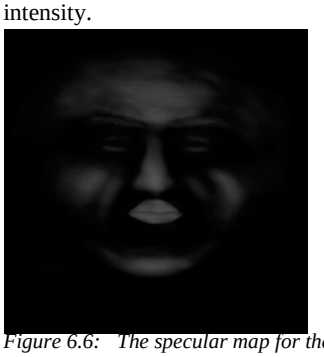


Figure 6.6: The specular map for the HMI face model.

A possible usage of the value can be the following. Looking at the specular highlight calculation from the Blinn-Phong shading model we can use the value at two points.

$$S = max\{N \cdot H, 0\}^{M_x} L_s M_s$$

In the original formula given above we can simply add the value from the specular map as a multiplier.

$$S = max\{N \cdot H, 0\}^{M_x} I L_s M_s$$

I Denotes the value from the specular map, used as an intensity regulator.

This is possible because a texture lookup returns the value from the texture in color space which ranges from 0.0 to 1.0. Thus white is max specular color and black zero specular color. Another use of the value is to use it as the specular exponent in the given formula. To give a bit of comparison the exponent in default OpenGL usage ranges from 1 to 128. The higher the exponent the smaller the specular highlight will be. This can give the following formula for calculating a new shininess value.

$$M_x = 1 + 127I$$

M_x Denotes the new specular exponent(shininess component)

The value 1 is just a small offset. Anything is fine as long it is not 0. And the value 127 is the max exponent that is allowed and is the same here as in default OpenGL. For the HMI face rendering a max value of 64 gives slightly better results.

The specular map can be expanded with more functionality. Currently only one color channel is used and can be extended to three to also regulate the specular color (the red, green and blue channels). Also the exponent value could be based on actual measurement data from real skin.

6.3 RENDERING THE EYEBROW

The rendering of the eyebrow is basically just a texture applied to the face mesh. This can be seen in figure 6.2 where the eyebrow is simply drawn into the diffuse map. This gives at a distance from the face the impression of eyebrows, but at a close range the eyebrow is visually not really satisfying. Therefore a simple fur shader has been implemented to give better eyebrows without heavy performance costs.

The fur shader algorithm is run in multiple passes, creating layers to simulate the fur. Prior to the multiple render passes an appropriate blending function is set. Then for each pass the polygons that contain the "fur" are pushed to the graphics card. In the vertex shader the vertices are displaced by a certain value which increases with each pass. In the fragment shader the color data is retrieved from the diffuse map (for example the diffuse map in figure 6.2) and an additional texture lookup for the alpha value. The alpha value from the texture lookup is multiplied with the alpha value for the 'current' pass to set the final alpha value, which is then used in the blending function for creating the opacity.

The blend function

The blend function from the default OpenGL capabilities is used[1]. This means that the color values for the incoming fragment are combined with the color values of the current stored fragment. The blend function from OpenGL:

$$C = \left[R_s S_r + R_d D_r \;\; G_s S_g + G_s D_g \;\; B_s S_b + B_s D_b \;\; A_s S_a + A_d D_a\right]$$

S_r, S_g, S_b, S_a The source factor

D_r, D_g, D_b, D_a The destination factor

R, G, B, A values of the fragments of the source and destination, indicated with a subscript s or d.

The source factor used is $\left[A_s \; A_s \; A_s \; A_s\right]$ (in OpenGL the constant GL_SRC_ALPHA)

The destination factor used is $\left[1 \; 1 \; 1 \; 1\right] - \left[A_s \; A_s \; A_s \; A_s\right]$

This way the alpha value from the source is used to filter or regulate the opacity.

The eyebrow mesh

As the algorithm is run for several passes, the polygon data needs to be pushed to the graphics card every time, where the vertex shader displaces the vertices with each pass a little. This directly constructs a small problem. A first possibility is to have the eyebrows have their own mesh which is placed into location on the face mesh. However keeping in mind that there is still a muscle system underneath, this would complicate matters as the eyebrows also can be displaced by certain muscles. This means that the actual vertices of the face mesh need to be used, but not all of them as that would be a waste of processing power. For this a bounding box is used and is located at the eyebrow locations. Then by prior looping the vertices of the mesh the vertices that

are located in the bounding box are stored. As it concerns here an indexed array of vertices, the indices are actually stored instead. Then using the indices to retrieve the polygons (the polygons here are all triangles) which use them and for each eyebrow the polygons needed are filtered. The result is shown in figure 6.7 and these reduced meshes are used within the fur algorithm.

Figure 6.7: The eyebrow mesh filtered from the face mesh.

Parameters vertex offset and alpha blending

With each pass another layer of fur is placed on top of the other. To do this there are two parameters that set with each layer. The vertex offset and the alpha value(transparancy) for the corresponding layer. The vertex offset displaced the layer and the alpha value blends the layer.

With each pass the vertices are a little bit displaced to the outside.

$$V' = \left[V_x + O_x \; V_y + O_y \; V_z + O_z\right]$$

$$O = \left[N_x o \; N_y o \; N_z o\right]$$

V' The new displaced vertex
V The incoming vertex position
N The normal vector for the current vertex
O The calculated offset
o The offset.

The offset is calculated by the pass number multiplied with a constant

$o = pc$
$c = 0.01$
p The pass number.
The alpha value is calculated by
$a = 1 - pr$

$$r = \frac{1}{l}$$

l The amount of passes (l referring to loops)
r The amount of alpha value to be reduced with each pass.
a the alpha value.

Within the fragment shader this alpha value is multiplied with the alpha value retrieved from the texture lookup. The value from the texture lookup is used to filter out the eyebrows from the skin. As can be seen in figure 6.7 the eyebrow is clearly surrounded by some skin texture. To filter this, an additional alpha map is created which is shown in figure 6.8. Using this alpha map extra detail can be added to smooth or thicken the appearance of the eyebrows.

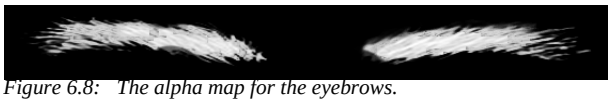


Figure 6.8: The alpha map for the eyebrows.

Notice that in the formula for displacing the vertices the y coordinate has been left out. This is only a minor cosmetic change because the eyebrows don't need to expand in height, or better said along the y-axis.

Final result

Using the alpha map to filter out the non-eyebrow fragments the only 'thing' that's left is the color texturing. Using the face diffuse map which already contains the eyebrows as a source the results are shown in figure 6.9. The settings here are 8 passes with an offset of 0.01.

Figure 6.9: Left no fur shading, right with fur shading (8 layers, 0.01 offset)

And a close up look is depicted in figure 6.10, where the eyebrow is shown from above which shows the separate layers.

Figure 6.10: The layers simulating the fur.

The extraction of the polygons or better said the vertex indices (which point to the vertex data in the vertex array) here have a two-fold purpose. First, for the fur shader the vertices need to be rendered multiple times. And secondly, now the eyebrow is modular and independend from the head model, yet still attached to the head model. If an underlying muscle system is active and modifying the position of the vertices the eyebrow would adjust itself to it because it uses the indices and not copies of the vertices. This way some other eyebrow rendering mechanism can easily replace the fur shading method and still work with a muscle system.

6.4 RENDERING THE EYES

Several options to render realistic eyes, deciding between a complex accurate high polygon mesh model or a simple sphere with a custom shader. The latter one is the choice as this project is more focused on using shader programs than modeling accurate models. Instead of dynamically creating the textures for the eye as this has been done in AGEye[32] (part of the Advanced Graphics course at HMI) a different approach has been chosen, jet trying to stay at a simple as possible basis. The eye constructed here is build out of two spheres, a big one for the eyeball and a second for the iris area. The eyeball only needs a texture or can even be left at an "almost" white color, but for the blood vessels which are also visible in the eyeball a diffuse map would can be

used. For the iris a small internal ray tracing algorithm[42] has been developed for simulating an offset effect of the iris, creating or simulating the "Aqueous humour" area as shown in figure 6.11.

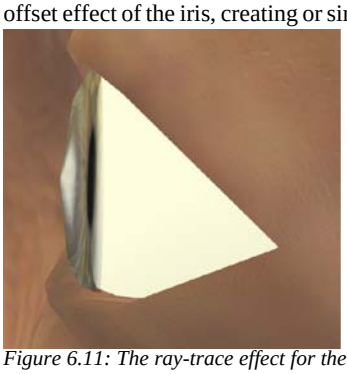


Figure 6.11: The ray-trace effect for the eye.

The algorithm uses a single bounce ray traced pass. The surface of the mesh is used to calculate the refraction vector. The refraction vector is then intersected with a pre defined plane, defined as being perpendicular to the object x axis. The intersection point can then be calculated and is used for the ST coordinates for a texture lookup. The algorithm presented here is based on the implemented shader program. Calculating the refraction vector:

$$R = \eta V + (\eta c_1 - \sqrt{c_2} N)$$

$$c_1 = -V \cdot N$$

$$c_2 = 1 - \eta^2 (1 - c_1^2)$$

V The normalized view vector to the surface.
N The normal vector in object space (normalized).
η refraction eta constant value.

Retrieving the intersection point and calculating the ST texture coordinates.

$$I = O + TR$$

$$T = \frac{-((P_{xyz} \cdot O) + P_w)}{D}$$

$$D = (P \cdot R)$$

I Point of intersection.
P plane equation which is defined over the x-axis ($\begin{bmatrix} 1 & 0 & 0 & scalar \end{bmatrix}$)
O fragment position in object space (point of origin).
R the refraction vector, or ray direction vector.

$$C = (sI) + t$$

$$t = \begin{bmatrix} 0 & 0.5 & 0.5 \end{bmatrix}$$

C The st texture coordinates stored in C_y, C_z texture coordinates.
t small addition to bring the value into color space.
s A scalar value, here to regulate the iris scale.

The retrieved texture coordinates can then be used to retrieve the diffuse color from an iris texture.

To further improve the visual quality of the eye, a reflection over the eye can be added. This effect requires the use of environment mapping which is described in 2.4 Environment mapping op pagina 10. It is possible to use a real-time environment map, or a static map. As the result is a subtle change in the eye color a static map is often more then enough. Also used often for this is replacing the light specular highlight with a texture (thus simulating the specular highlight by using only a texture). Here an environment map from a so called skybox is used. A skybox is a cubemap that is used to be placed around the users view in order to give the feeling of being in

some kind of surrounding (for example seeing in a distance the mountains). In figure 6.12. the

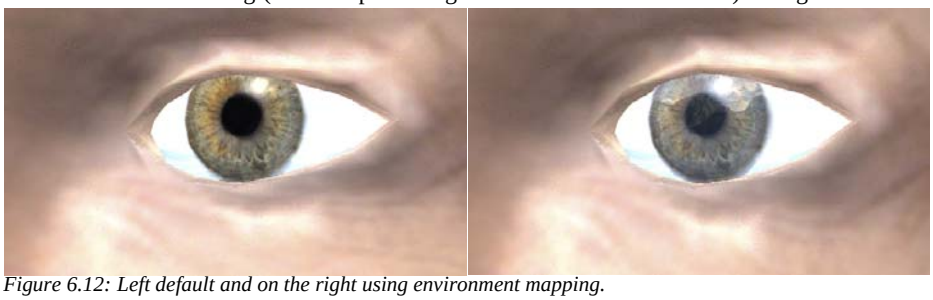


Figure 6.12: Left default and on the right using environment mapping.

skybox has been applied as an environment texture to the eye on the right.

6.5 RENDERING THE EYELASHES

The final part of the face are the eyelashes. The eyelashes here use the same approach as the eyebrow rendering. First the vertices on which the eyelash will be attached need to be extracted from the head model. This is done by a bounding box and filtering the vertices inside the box from the head model (figure 6.13). From the filtered vertices the indices are kept, this in order to keep it modular and ensure it is possible when animating the eyelit that also the eyelash is moving correctly.

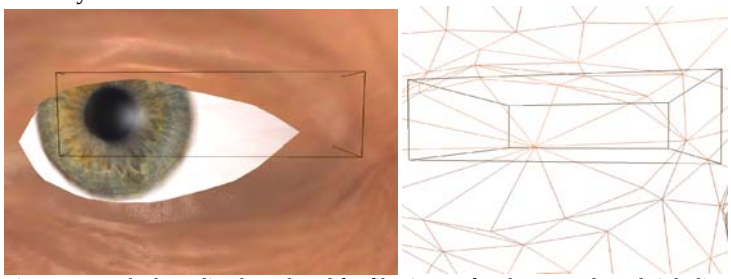


Figure 6.13: The bounding box placed for filtering. Left polygon mode and right line mode.

Now having the vertices that are withing the bounding box, we still need to filter the exact vertices onto which the eyelash will be attached. To do this a small algorithm is used to filter a line from left to right. First the most left and the most right vertices are filtered from the list of vertices in the bounding box. This is based on the x-axis value.

The leftmost vertex is taken as the start vertex and the rightmost vertex is the goal. Now using the polygons, that use the vertices in the boundingbox, the algorithm must choose the shortest route from left to right. This effect is shown in figure 6.14, showing on the left the line layout of the

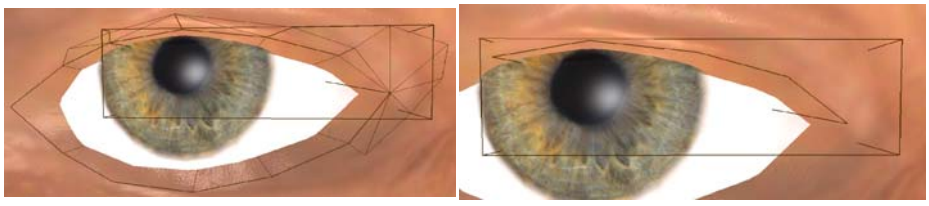


Figure 6.14: The shortest route from left to right. Left the available polygons and right the shortest route.

polygons and on the right the line depicting the shortest route from left to right.

Now it is known where the eyelash is going to be attached. If these vertices move, then the eyelash model can adjust itself to it. The eyelash model presented here is a very simple, and low performance costly.

The vertices used in the shortest route are duplicated and extruded into a preffered direction. The direction is controlled by certain parameters regulating the offset. The used parameters yOffset, zOffset and zSineOffset are shown in figure 6.15. The front, side and top view for rendering the

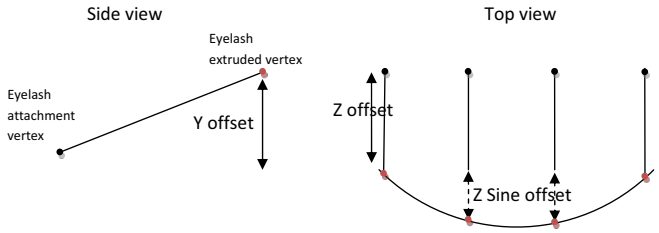


Figure 6.15: Usage of the offset parameters.

eyelash is presented in figure 6.16. Now a texture can be applied which streches from side to side.

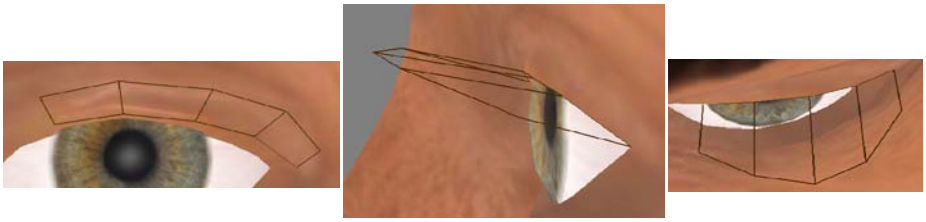


Figure 6.16: Several views on the eyelash line model. Front, Side and Top view.

To do this the texture coordinates also need to be generated using a linear interpolation from left to right.Using an image editor a simple eyelash is drawn and applied to the eyelash surface. This is shown in figure 6.17 where also a second smaller lash addition is visible.A note on the

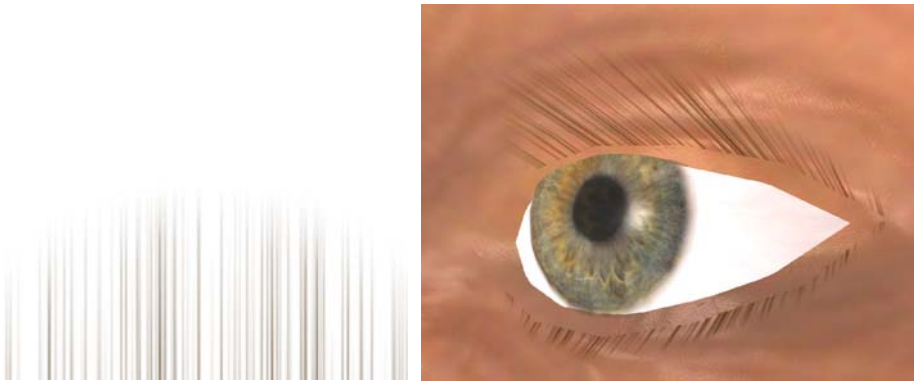


Figure 6.17: Left the custom drawn eyelash texture; Right the rendering of the eyelash.

implementation here is the use of the texture and it's blending. The texture uses an alpha channel to filter out the white part, so that only a set of small brown strokes remain. By default a new loaded texture in OpenGL is set to a linear interpolation with mip mapping. The combination of the blending, linear interpolation and the mipmapping causes the eyelash to blur and the blending shows artifacs(it becomes clearly visible that it is a quadstrip). To prevent this, the mipmapping is disabled.

7

CONCLUSION

In this chapter the conclusion of this engineering project is discussed by reviewing the end result in comparison to the original head(figure 7.1). The final result using a combination of the mentioned render techniques is shown in figure 7.2. For every topic a conclusion is presented.

By using a different lighting model than the default model from Blinn-Phong subtle differences on how the surface is lit can be achieved. In figure 7.2 the default lighting model was replaced by a combination of the Cook-Torrance and the Oren-Nayar model. The differences are hard to see and from a personal opinion difficult to say if it actually improved on realism but this can also be the cause of not using a correct diffuse maps (these are hand drawn and not photographs). The default per fragment based model (Blinn-Phong shading) gives a higher specularity which seems useable in cases if the persons skin is wet (water or oily), and the Cook-Torrance/Oren-Nayar give a softer look which seems a better overall choice. By replacing the ambient component with the spherical harmonics rendering a great improvement is achieved as it actually takes the environmental lighting into account on the head model. The overall conclusion is that replacing the light model with a more sophisticated model greatly improves visual realism.

- **Using per fragment lighting instead of per vertex provides increases the detail on how light affects the surface of a mesh.**
- **Environmental ambient lighting using spherical harmonics or environment mapping increases the realism on indirect lighting.**
- **Using different specular and diffuse reflectance models different material properties can be taken into account. The skin rendering with the Cook-Torrance/Oren-Nayar model gives a more fragmented lighting effect (which gives a visual softer look).**
- **By defining a custom lighting model based on the components ambient, diffuse and specular gives great flexibillity (implementational). Replacing parts of the lighting model (ambient, specular or diffuse) can improve quality in certain circumstances (example skin wet or dry).**

The use of normal mapping greatly improves the quality of the head model. It mainly introduces wrinkles in the face. It actually shows "age" and other skin irregularities. As presented the normal maps can be dynamically generated, which implies dynamic wrinkles.

- **Bump mapping increases the detail on the face.**
- **Normal maps in tangent space make it easy to generate these maps dynamically.**

Shadow mapping provides a fast and accurate method for self-shadowing. It provides the scene with additional visual cues on how the light effects the face/head and increases the realism.

- **Provides additional cues on how the light affects the scene, improves realism.**
- **Easy to implement, good hardware support.**
- **Fast and accurate shadows for self shadowing.**

The use of the deferred shading rendering system came out as a robust and flexible rendering method. It allows easy creation of multiple layers, partial buffer updates and advanced post processing. It's downside is the amount of fragment processing power needed. However with newer hardware this problem might not be severe and might even outperform traditional rendering methods as it handles multiple lights on a fragment level more efficiently. Currently several applications already use this method (games like S.T.A.L.K.E.R., Ghost Recon: Advanced Warfighter, Gothic 3 and the Unreal-engine 3).

- **Provides a system capable of separating the scene data from the lighting models. From an engineering viewpoint this makes everything more modular and easier to maintain.**
- **Provides a layering mechanism which makes it easy to add new light and/or effects. Experimenting with new shader effect can be done on top of the previous layers.**

Skin rendering. The rendering method here for the skin delivers a great visual improvement, especially in combination with a customized lighting model. In figure the Ambient component is the Spherical harmonics function, the diffuse model the Oren-Nayar and the specular model based on the Cook&Torrance model. By creating several maps that contain information on what color the skin has or the amount of specular intensity a great flexibility is given on how to provide the information for each component of a lighting model.

- **Using textures greatly improves the realism and detail on the head model. Even if the texture is hand drawn it already changes the head appearance completely in a more realistic way.**
- **Using a specular intensity map provides control over "per fragment" specularity on the head model. For example the nose is more reflective then the cheeks.**
- **The normal maps provide detailed information on how the light effects the skin. Creating normal maps to simulate skin irregularities and wrinkles improves the realism. Example the forehead wrinkles in figure 7.2 which are clearly not visible in figure 7.1.**

Eye rendering. The rendering of the eyes is kept relatively simple. A color map providing the eyeball and iris color information, a reflection map for having reflections in the eyes and a refraction shader used for a lens effect. The eyes are more detailed and by using real photo's for the iris texture the realism is improved. The main problem with the eyes is however the placement. As they are here perfectly round eyes they don't really fit in the eye socket of the head model. Therefore they were placed a little bit backwards in order to prevent the eyeball from coming through the head model.

- **Using a more detailed texture on the iris gives a more realistic eye appearance.**
- **Using a small lens shader provides a subtle lens effect when looking close up to the eye.**

Eyebrow rendering. The eyebrows are rendered with a fur shading program. This provides a more voluminous eyebrow at a relative low cost of rendering performance. By first extracting the polygons from the main mesh model that are used for the eyebrow makes replacing the eye brow rendering with some other method easier and independent of any underlying vertex-modifier system (for example a muscle system).

- **A fur shader is able to be used for eyebrow rendering.**
- **The method used for filtering the vertices from the head model keeps the eyebrows independent in rendering but are still "attached" to a possible underlying muscle system.**
- **The fur shading can easily be replaced by a different eyebrow rendering model.**

Eyelash rendering. The principle from the eyebrow rendering also applies here. By first extracting the vertices that form the base for the eyelash support a way of placing an eyelash rendering method onto the head model without modifying the head model itself.

- **Using the same approach as with the eyebrows for filtering the vertices. It provides independent rendering from the head model, yet still affected by a vertex modifier system on the head.**
- **The rendering model can easily be replaced.**

This project showed a variety of techniques and described how they could be combined. In virtual reality many ways are possible and are always approximations. This project:

- **Increased visual realism on the head as can be seen by comparing figure 7.1 with figure 7.2**.
- **Provides each technique in a modular way, so that it can be taken from the system to be used in other systems as they are independend from each other.**

As for framerates, using a NVidia 6800GT(agp), AMD 3200+ the following fps where measured:
One dynamic light
Deferred shading (internal resolution 768x768) : 60~65 fps
Non-Deferred shading (screen resolution 1280x1024): 190~230 fps
Three dynamic lights
Deferred shading (internal resolution 768x768) : 25~30 fps
Non-Deferred shading (screen resolution 1280x1024): 60~100 fps

Figure 7.1: The original head model.

Figure 7.2: Rendering the head with visual improvements; Final result.

8

FUTURE WORK

In this chapter the future work for this engineering project is given. In short for every technique a small summary of possible improvements is given.

Lighting models:

- **Implementing more complex/realistic lighting models.**
- **A lighting model which takes account of skin reflective parameters based on actual measured data, using a closer real-world BRDF model.**
- **Optimize the algorithm using more lookup textures in order to reduce the instructions per cycle.**

Bump mapping:

- **Dynamically generate wrinkles based on the underlying muscle system.**
- **A hybrid system. Wrinkle creation based on vertex displacement (as used by The Dui Buy) and on a per fragment level with normal mapping.**
- **Dynamic creation of skin irregularities. For example spots, birthmarks and scars.**
- **Optimize the bumpmap blending as this is performance intensive.**

Real-time shadow casting:

- **More advanced shadowing, better soft shadowing by simulating the penumbra more realistic. Example: enhancing the penumbra region by taking the lights intensity and object distance into account[33].**

Skin rendering:

- **Color information of the skin, generated or photo realistic textures (instead of hand drawn textures).**
- **Dynamic layered creation of skin color maps.**
- **Sub surface scattering, a lighting model taking account of the several layers of the skin.**
- **An extension to the sub surface scattering can be taking translucency. For example with the ears, if a light is shining on the back side of the ear then still some light is partially coming through.**

Eye rendering:

- **A more realistic eye model instead of using a single sphere.**
- **An algorithm that can adjust the eye shape or eye socket in order to fit it perfectly.**

Eye brows & eyelashes:

- **Improved textures.**
- **Use a realistic (more complex) hair rendering technique.**

BIBLIOGRAPHY

Books

[1] M. Woo, J. Neider, T. Davis & D. Shreiner. OpenGL Programming Guide, third edition (also known as the Red book). ISBN: 0-201-60458-2. URL: http://www.opengl.org/documentation/red_book

[2] D. Shreiner. The OpenGL Reference Manual, fourth edition (also known as the Blue book). ISBN: 0-321-17383-X. URL: http://www.opengl.org/documentation/blue_book

[3] R. J. Rost. OpenGL Shading Language, second edition (also known as the Orange book). ISBN: 0-321-33489-2

[4] S. Savchenko. 3D Graphics Programming, games and beyond. ISBN: 0-672-31929-2

[5] K. Dempski and E. Viale. Advanced Lighting and Materials with shaders. ISBN: 1-55622-292-0

[6] A. Watt. 3D Computer Graphics. ISBN: 0-201-63186-5.

[7] A.Watt and F. Policarpo. 3D Games, Real-time rendering and software technology (volume one). ISBN: 0-201-61921-0

[8] M. Woo, J. Neider, T. Davis and D. Shreiner. OpenGL programming guide (third edition). ISBN: 0-201-60458-2

[9] Direct3d ShaderX : vertex and pixel shader tips and tricks. ISBN 1-55622-041-3

[10] ShaderX2: Introductions and Tutorials with DirectX 9.0. ISBN 1-55622-902-X

[11] ShaderX3: Advanced Rendering with DirectX and OpenGL. ISBN 1584503572

[12] T. Duy Bui. Creating Emotions and Facial Expressions for Embodied Agents. ISBN 90-75296-10-X.

Articles

[13] C. Schlick. An Inexpensive BRDF Model for Physically-based Rendering; Computer Graphics Forum, Volume 13, Issue 3 (1994) pp. 233-246.

[14] J. F. Blinn & M. E. Newell. Texture and reflection in computer generated images.Communications of the ACM Volume 19 , Issue 10 (1976) pp. 542 - 547.

[15] W. Heidrich & H. Seidel.View-independent environment maps. SIGGRAPH/EUROGRAPHICS Workshop On Graphics Hardware archive (1998) pp. 39 - ff. ISBN 0-89791-097-X

[16] R. Ramamoorthi & P. Hanrahan. An Efficient Representation for Irradiance Environment Maps. International Conference on Computer Graphics and Interactive Techniques.(2001) pp 497-500. ISBN 1-58113-374-X.

[17] J. Kautz, P. Sloan & J. Snyder. Fast, Arbitrary BRDF Shading for Low-Frequency Lighting Using Spherical Harmonics. ACM International Conference Proceeding Series; Vol. 28(2002). pp. 291-296. ISBN 1-58113-534-3.

[18] R. Green. Spherical Harmonic Lighting: The Gritty Details. Game Developers Conference 2003. www.research.scea.com/gdc2003/ spherical-harmonic-lighting.pdf. Last visited: 28-05-2007

[19] V. Havran, A. Nuemann, G. Zotti, W. Purgathofer & H. Seidel. On cross-validation and resampling of BRDF data measurements. Spring Conference on Computer Graphics (2005). pp. 161-168.ISBN:1-59593-203-6.

[20] J. F. Blinn. Simulation of wrinkled surfaces. Proceedings of the 5th annual conference on Computer graphics and interactive techniques(1978). pp.286-292.

[21] M. Peercy, J. Airey & B. Cabral. Efficient bump mapping hardware. Proceedings of the 24th annual conference on Computer graphics and interactive techniques(1997). pp. 303-306.

[22] T. Welsh. Parallax Mapping with Offset Limiting: A PerPixel Approximation of Uneven Surfaces.(2004).

[23] E. Chan & F. Durand. An Efficient Hybrid Shadow Rendering Algorithm. Proceedings of the Eurographics Symposium on Rendering 2004.
Url: http://people.csail.mit.edu/ericchan/papers/smapSV/

[24] T. Akenine-Moeller, E. Chan, W. Heidrich, J. Kautz, M. Kilgard & M. Stamminger. Real-time shadowing techniques. ACM SIGGRAPH 2004 Course(2004).

[25] Bart Kevelham. Real-time Shadows in Augmented Reality Environments; Human Media Interaction, University of Twente.

[26] S. Brabec, H. Seidel. Hardware-Accelerated Rendering of Antialiased Shadows with Shadow Maps. Computer Graphics International (2001). pp. 209-214.

[27] U. Assarsson, M. Dougherty, M. Mounier & T. Akenine-Moller. An Optimized Soft Shadow Volume Algorithm with Real-Time Performance. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware(2003). pp. 33-40.

[28] F. Policarpo & F. Fonseca. Deferred Shading Tutorial. http://fabio.policarpo.nom.br/docs/Deferred_Shading_Tutorial_SBGAMES2005.pdf. Last visited: 28-05-2007

[29] F. Struck, C. Bohn, S. Schmidt & V. Helzle. Realistic Shading of Human Skin in Real-time. (2004) pp. 93-97.

[30] A. Harot, B. Geunter and I. Essat. Real-time Photo-Realistic Physically Based Rendering of Fine Scale Human Skin Structure. Proceedings of the 12th Eurographics Workshop on Rendering Techniques(2001). pp. 53-62.

[31] Y. Wu, P. Kalra, and N. Magnenat-Thalmann. Physically-based wrinkle simulation and skin rendering. In Eurographics Workshop on Computer Animation and Simulation(1997).
http://www.miralab.unige.ch/papers/104.pdf. Last visited: 28-05-2007.

[32] N. A. Nijdam & A. S. Tigelaar. Advanced Graphics: Simple Eyes. Advanced Graphics (AG) [216640] course assignment, University of Twente (2006).

[33] Q. Mo, V. Popescu & C. Wyman. The Soft Shadow Occlusion Camera. www.cs.uiowa.edu/~qmo/UICS-TR-07-02.pdf. Last visited: 28-05-2007.

[34] S. K. Nayar & M. Oren. Generalization of the Lambertian Model. International Conference on Computer Graphics and Interactive Techniques(1994). pp. 239-246.

[35] R. L. Cook & K. E. Torrance. A reflectance model for computer graphics. ACM Transactions on graphics(1982) pp. 7-24.

[36] E. P.F. Lafortune, S. Foo, K. E. Torrance & D. P. Greenberg. Non-Linear Approximation of Reflectance Functions. SIGGRAPH 97 Conference Proceedings, Los Angeles, California (1997), pp. 117-126.

Web sources

[37] OpenGL.org

[38] \OpenGL Frame buffer extension.
URL:http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt
Last visited: 28-05-2007.

[39] P. Devebec. personal website. http://www.debevec.org/
Last visited: 28-05-2007.

[40] AMD ATI RenderMonkey Toolsuite. http://ati.amd.com/developer/rendermonkey/index.html. Last Visited: 28-05-2007

[41] A. S. Shastry. Soft-Edged Shadows. Gamedev http://www.gamedev.net/reference/articles/article2193.asp. Last visited: 28-05-2007

[42] NVIDIA developer site. http://developer.nvidia.com/
Articles and SDK sources.Last visited: 28-05-2007

[43] NVIDIA Shadow Mapping.
http://developer.nvidia.com/object/hwshadowmap_paper.html. Last visited: 28-05-2007

[44] NVIDIA Simple Soft Shadows. http://http.download.nvidia.com/developer/SDK/Individual_Samples/DEMOS/Direct3D9/src/HLSL_SoftShadows/docs/HLSL_SoftShadows.pdf. Last visited: 28-05-2007

APPENDICES

A“The OpenGL render pipeline”

A

THE OPENGL RENDER PIPELINE

OpenGL is a software interface to graphics hardware. This interface consists of about 250 distinct commands (about 200 in the core OpenGL and another 50 in the OpenGL Utility Library) that you use to specify the objects and operations needed to produce interactive three-dimensional applications." [1]. In this chapter a brief overview of OpenGL is presented.

A.1 OVERVIEW

An abstract overview of the OpenGL 2.0 graphics pipeline[3] is shown in figure A.1 which is based on the OpenGL 2.0 specification.

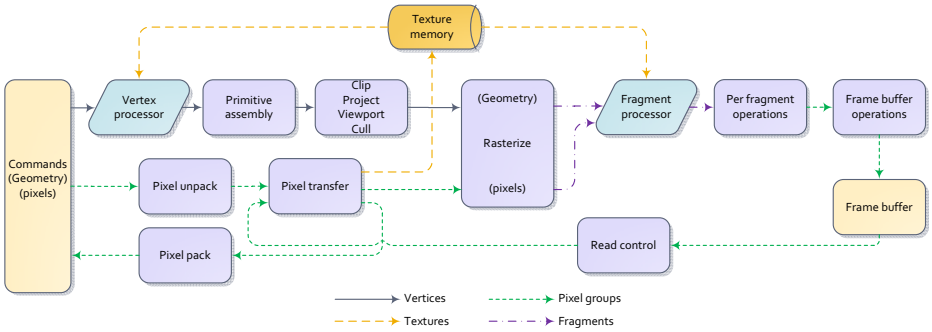


Figure A.1: Abstract overview of the Opengl pipeline.

In short : The commands inserted into the pipeline can specify geometric objects to be drawn or control commands specifying how various stages are handled. The vertices are processed by the vertex processor which is a programmable unit (using vertex shader programs). After the vertex processor the vertices are made into triangles and are rasterized. Then another programmable unit the fragment processor (using fragment shader programs) operates on every fragment. At last the fragments are places into the appropriate buffers (here are still some "frame buffer operations" possible). Finally the front buffer will be shown on screen and we can see the result as pixels.

Primarily OpenGL is used for representing virtual objects in a three dimensional space (although it supports 2D too) and uses the following coordinate system (figure A.2):

- **Horizontally the x-axis where positive goes right and negative left.**
- **Vertically the y-axis where positive goes up and negative down.**
- **Depth the z-axis where positive is forward and negative is backwards**

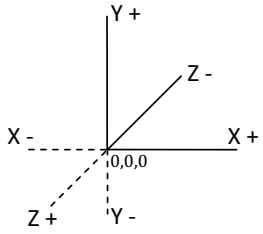


Figure A.2: OpenGL coordinate system

OpenGL Geometric primitives

The most basic primitives available in OpenGL are points, lines and polygons. For the lines and polygons several types of data streams are available. For example the polygon can be defined as triangles, triangle strips, quads, quad strips. On hardware level a polygon will be transformed into triangles in the primitive assembly stage (example: a quad will be broken into two triangles). Triangles form the minimal representation for a surface element, are convex and are linear which makes it computational more efficient/easier.

Points: A point is defined by a vertex usually denoted as $\lfloor x\ y\ z \rceil$. The vertex can also contain a fourth component, the "w component", used as a scalar.

Lines: a line is defined by two vertices.

Polygons: A polygon is defined by three or more vertices. The polygons need to be well formed. This means they need to be convex, otherwise depending on the hardware unknown rendering results occur. Convex means that when drawing a line from within a polygon to another point inside the polygon, it never crosses a border. In order to draw concave polygons, the best solution is to break the concave polygon up into multiple convex polygons.

3D models

A 3D model contains information for rendering a virtual three-dimensional object. The mesh is the central part of a 3d model. A 3D model mostly refers to a complete set of information about a virtual object, where a mesh can be a small part of that information. A mesh contains a set of data that describes a virtual object. The most basic data for a mesh are the vertices. With these vertices a mathematical representation is possible in points, lines or polygons.

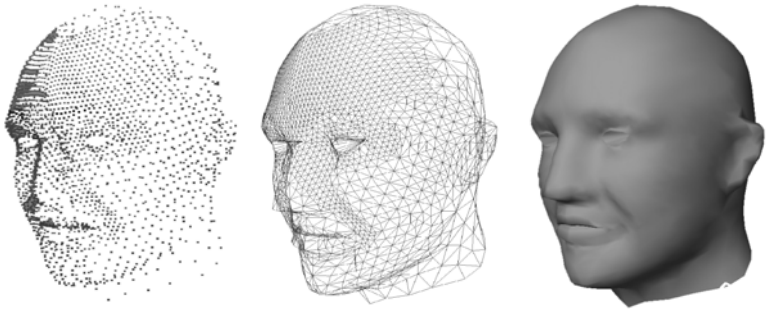


Figure A.3: The HMI head model shown in points, lines and filled

The actual rendering depends on how the information is used within a 3D render application. A mesh is usually rendered as polygons, connected to each other creating a solid object. A usual rendering procedure for a mesh is using two arrays, first a set of unique vertices and secondly an array of indices. Since most vertices are used by multiple polygons the array of indices can reduce the amount of data significantly that has to be send to the graphics card. The indices are ordered in a specific way. This in order to render polygons, also known as faces.

Other information that a mesh might contain can be colors, normal vectors, texture coordinates and other vertex/face information (such as tangent vectors).

It is common that 3D models contain textures which contain color information about the surface of an object and/or have material properties. In simple words, textures are images that are blended onto the polygons during rendering, creating an imaginary surface.

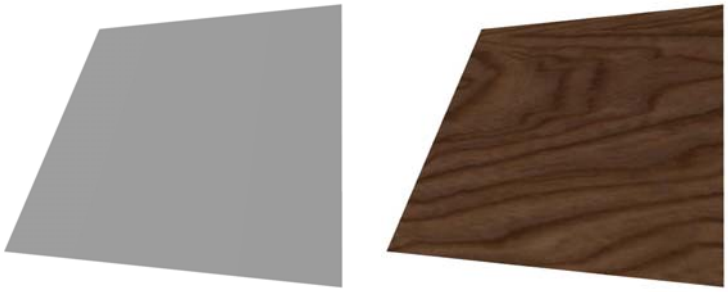


Figure A.4: The square on the left has no texturing, and on the right a wood texture is applied.

However using textures for only color information is not their only useful purpose. With today's vertex and fragment shader programs these textures are often used as a source of information for all kinds of operations. One of those operations is Normal mapping. In short normal mapping is the use of a texture which contains information about the normal vector. During rendering for each pixel the normal vector is read from the texture instead of the being provided by the mesh data. Further it is up to the designer/programmer how to use these textures, which are also known as samplers.

Coordinate spaces

There are many coordinate spaces involved with rendering. First the 3D model, the vertices defined for the model are in object space. For example a box being modeled around zero. A box has 6 quad faces (or 12 triangulated faces) and uses 8 unique vertices. During rendering the box can be placed (translated) and rotated at any other position in space (this means that the position of the vertices are altered), this space is then called model space (also known as world space). However most of the time this isn't the model space but the modelview space or better known as eye space. The eye space includes the viewing translation and rotation which is set before actually moving and rotating the box. The modification for setting up the view position and direction is also a space in itself called view space.

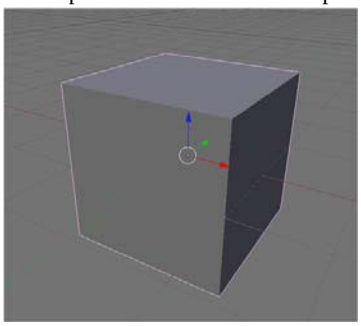


Figure A.5: If a box has no translation and rotation then model space is equal to the object space.

In order to move from one space to another, the data is multiplied by a 4 by 4 matrix and contains information about rotation, translation, scaling and projection adjustments.

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

In other words, a space is defined by this 4x4 matrix. To keep track on these spaces OpenGL uses a matrix stack (last in first out) and knows several matrix stacks. The most well known matrix stacks are the ModelView and the Projection (although the Projection matrix stack isn't normally used intensively). Other OpenGL matrix stacks are texture and color matrix stacks.

The Projection matrix stack is used for

- **Field of view (fov)**
- **Setting an perspective or orthographic view**
- **Viewing frustum (defining the clipping planes)**

As for the ModelView stack the name already implies the combination of the model space and the eye space.

Coming back to the matrix itself, the range m0 to m10 is also known as the rotation matrix. Besides the rotation it also contains the scaling modifications. The range m_{12} to m_{14} contain the translation where $m_{12} = x, m_{13} = y, m_{14} = z$. A summary of possible operations on such a matrix are rotation, translation, scaling, skewing, inverting, and projection.

In theory the vertices in object space are transformed into world space, then into eye space. Before rasterizing the vertices are thrown into clipping space and finally the window space. However there are many ways on how to actually implement and how to use these spaces. Especially the first three spaces are very depending on how the render procedure works and are all involved with the OpenGL ModelView stack.

Example "Does the world turn, or is it just me?"

Moving an object around is nothing more than an adjustment to the current matrix. This is also the case with the camera (the viewing point). Since both operate on the same matrix stack it is important to know which occurs first. A logical order to have is the following: move an object into world space by setting the transformation/rotation for that object and then going into view space by applying the transformation/rotation of the camera. This means, starting with the identity matrix it is modified according to the world space mutations (translation, rotation etc) and then again changed by the eye space mutations. A problem is then if you have multiple objects which are all located at different places in the world space then you need to apply the camera settings with every object.

```
Render frame
   Set Identity matrix
   For each object
      Push matrix (copy current matrix on the stack)
      Object world translation/rotation
      Camera Translation/rotation
      Pop matrix (remove top matrix)
```

Figure A.6: Apply camera translation&rotation on every object.

This is a waste of processing power, because you need to adjust the space matrix with the camera settings for every object. The ‘correct’ or ‘preferred’order is shown in figure A.7.This is only a

```
Render frame
   Set Identity matrix
   Camera Translation/rotation
   For each object
      Push matrix (copy current matrix on the stack)
      Object world translation/rotation
      Pop matrix (remove top matrix)
```

Figure A.7: Apply camera translation&rotation prior to processing every object.

simple example on how to order the matrices in order to get the appropriate space modifications. Knowing this is important because these matrices aren't only used to get the vertex data into eye space, often (especially with shader programs) you need these matrices in combination with custom matrices for certain operations. OpenGL also has functionality to retrieve the current matrix from the stack, and then you have to know in what space you are operating. One example is shadow mapping, where the matrix from a light point is being used to set up a projection matrix for creating shadows. It is quite easy to get confused using several spaces, or even worse if you have no idea in what space a particular object is. To give this some more depth, take a lighting pitfall for example. When defining a light point in OpenGL, thus specifying coordinates for a light point origin. You provide coordinates in object/world space and NOT directly in Eye space. The OpenGL build in lights "automatically" set these coordinates into eye space. The confusing part here is when you are writing a shader program with a custom light shader (or any shader using the build in light variables), you need to know that this is already done for you, but if you define a light source yourself, then you need to convert the position of the light source into the appropriate space which usually is the eye space. The figure A.8 gives a default sequential schema of the spaces.

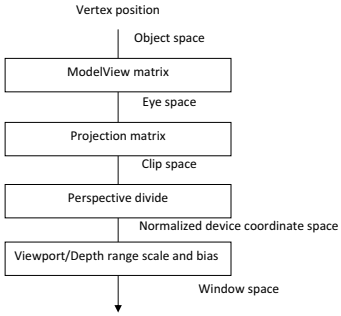


Figure A.8: Most common spaces in the OpenGL render process.

There are more spaces, in fact every newly created matrix that can be used for moving a coordinate into a different state can be seen as a new space, but only certain "types of matrices" or "matrices with the same intention" that are used often are named. Another example of a space that is used in this document is tangent space. This tangent space (also known as texture space) is a kind of intermediate space and is used for storing data that can be a source for all kinds of operations, in this case normal vectors are stored in tangent space that are to be used with a lighting model (normal mapping). The tangent space is discussed in more depth in “Bump mapping” on page 21.

A.2 OPENGL BUFFERS

Ultimately after rasterizing and fragment processing (and some other hardware build-in operations) the "pixels" are stored in the frame buffer. In 1969-70 J. Miller introduced a 3-bit frame buffer at Bells Labs. This means for every pixel there was 3-bit available, good enough for 8 unique colors. Today's frame buffers are a bit more advanced (provide up to 128-bit), however in the essence they are the same, namely storing colors. The use of "bit per pixel" is called a bit-plane, thus in the 3-bit per pixel implementation there are 3 bit-planes available.

OpenGL gives by default support for Color, depth, accumulation and stencil buffers which will be further explained in this section.

Color buffers

Available are the front-left, front-right, back-left, back-right, and depending on the hardware at least 4 auxiliary buffers. For color buffers usually the red green blue components ("RGB") are used and optionally an Alpha component ("A", thus 4 components). Here the amount of bit-planes are divided over the amount of components, for example a 24bit-plane system gives 8 bits for every component (RGB) and thus for 32bit is used for RGBA. There are extensions available to set the number of bit planes for each component, however this is hardware depended (and usually these extensions are only supported by the hardware manufacturer).

Today's hardware is also capable of higher precision giving 16bit for every color (floating point, and depending on the hardware fixed point setups), which then sums up to a 64bitplane system or even 32bit precision for each component per pixel (thus 128 bit planes, and good enough for 4 billion color values per color channel). The higher precision cannot always be seen by the human eye, however for complex lighting/coloring calculations these become necessary to increase image quality (also for special effects such as high dynamic range lighting). So the higher precision isn't necessary for output but more for the internal calculations. A quote from ID software lead programmer J. Carmack stating this issue:

> *"Eight bits of precision isn't enough even for full range static image display. Images with a wide range usually come out fine, but restricted range images can easily show banding on a 24-bit display. Digital television specifies 10 bits of precision, and many printing operations are performed with 12 bits of precision.*
>
> *The situation becomes much worse when you consider the losses after multiple operations. As a trivial case, consider having multiple lights on a wall, with their contribution to a pixel determined by a texture lookup. A single light will fall off towards 0 some distance away, and if it covers a large area, it will have visible bands as the light adds one unit, two units, etc. Each additional light from the same relative distance stacks its contribution on top of the earlier ones, which magnifies the amount of the step between bands: instead of going 0,1,2, it goes 0,2,4, etc. Pile a few lights up like this and look towards the dimmer area of the falloff, and you can believe you are back in 256-color land."*
>
> *Carmack 4/29/00*

By default the front-left buffer is used for displaying the end results on the screen, however the front-right can also be enabled to be used in stereo-scoping mode. And for a better picture quality double buffering can be enabled. This actually enables the back buffers, while the front buffer is being shown on screen the back buffer is being filled with new data, and when ready it is flipped. Meaning the back buffer becomes the front buffer and vice versa. This prevents graphical glitches as only a complete rendered frame is being shown. However there is still a chance for a lag since the flipping of the buffers can only occur at the end of a screen refresh (monitor sync). A third buffer can be used for this as an intermediate buffer, and is called Triple buffering as such (note: OpenGL does not give access to this buffer as there are still only the front and back buffers references).

Depth buffer

This buffer is also known as the Z-buffer as it actually stores z values from eye space in a 0.0 to 1.0 range. The amount of bit planes gives the precision at which the depth value can be stored for every pixel. The depth buffer is usually used for depth culling by using a depth-test function.

For example when rendering two polygons, A and B, and polygon A overlaps B. Then depending on the depth function, the fragments from polygon A may overwrite those of B or vice versa. Also the order in which the polygons are drawn have an effect with certain settings. By default the

depth function is set to "Less", meaning when the depth of a new fragment is smaller than the value in the depth buffer it is over written with the new value.

Figure A.9: Depth testing with depth functions: left lesser, middle greater and right always.

Stencil buffer

The stencil buffer is like the depth buffer and also is used with a function to test a fragment to check whether or not to update the buffer. However it doesn't store depth, but stores an unsigned integer that.

"The stencil function controls whether a fragment is discarded or not by the stencil test, and the stencil operation determines how the stencil planes are updated as a result of that test"

The stencil buffer can be used for masking, or other applications such as Shadow volumes.

Accumulation buffer

The accumulation buffer is like a color buffer, only the usage of this buffer is different. By using a special function it takes the current draw buffer (for example the front left color buffer) and uses it to update the accumulation buffer. Only whole buffer operations are possible (thus not per pixel). The function takes two parameters. First an operator (for example multiplication) and second a value.

This buffer can be used for a lot of things, especially special effects such as motion blur, depth of field, soft shadowing and anti aliasing. However with today's shader functionality there is more control over what to do with each pixel, and so it seems less attractive to use the Accumulation buffer.

Frame buffer object

The framebuffer object extension(FBO) provides a mechanism to render off-screen, context independent and capable of supporting several logical buffer types. However this is still an extension and not a part of the OpenGL 2.0 set, but the latest OpenGL drivers provide support for this extension on graphics no older than about 2 years (maybe even older cards, but that depends on their hardware capabilities).

> *Status: Complete.*
> *Approved by the ARB "superbuffers" Working Group on January 31, 2005. Despite being controlled by the ARB WG, this is not an officially approved ARB extension at this time, thus the "EXT" tag.*
>
> *OpenGL.org; Extension document[38] last viewed on: 6/5/07*

With this the FBO more or less replaces the pixel buffer system (pbuffer), which has been around for some time (from SGI's IRIX systems). The pbuffer and FBO are used to render "off-screen" in a buffer, meaning it is not directly used for displaying. By storing data that can be used for other

purposes, for example an often used application is "render to texture" where a virtual scene is rendered in the background (thus "off screen") and then used for texturing purposes.

The main difference between the pbuffer and an FBO is that the pbuffer has its own OpenGL context. This means it is separated from the main OpenGL context (for example the OpenGL context that is used for drawing on screen). The pbuffer also contains, just like a normal OpenGL context, all the default OpenGL buffers (color, depth, stencil and accumulation buffers) however these buffers cannot be directly shared with other pbuffers or the main OpenGL context. For using the data in the buffer, it needs to be copied from the buffer into an intermediate buffer (or destination buffer that is available in the other OpenGL context). Another disadvantage is the speed impairment when switching between pbuffers, it causes a complete OpenGL context switch. The FBO on the other hand can be used in a single OpenGL context and contains buffers that can be easily attached or detached (and can be exchanged between FBO's). These buffers are the so called logical buffers and resemble the default OpenGL buffers. An FBO can have several Logical color buffers, one depth buffer, one stencil and one accumulation buffer. The logical buffers are containers of "framebuffer-attachable images" which is a "texture object" or a "render buffer object". The special notion here is the texture object, because the buffer can directly be used as a texture or as a sampler in a shader program. The render buffer is used for those types of logical buffers that have no corresponding texture format (stencil, accumulation, etc) or which that don't always need one (for example depth, where only the depth test is necessary to function).

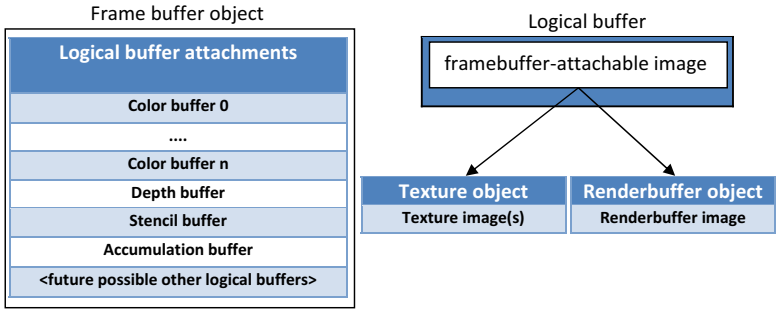


Figure A.10: The Framebuffer object and its components.

On a side note: A drawback with current off-screen rendering is the lack of hardware supported anti-aliasing. A possible solution is a custom shader program handling the anti-aliasing.