A light manager for a dynamic light environment

Mikael Eriksson

September 23, 2006
Master's Thesis in Computing Science, 20 credits
Supervisor at CS-UmU: Anders Backman
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

Lighting is necessary when rendering a 3d-scene. By using dynamic lights and shadows, the quality of the scenes lighting is improved. In this thesis a theoretical study of different light models, ways to render shadows and level of detail for lights, has been performed in order to create an improved system for lighting. A light manager and a shadowing system were implemented, the light manager uses both user-defined and calculated priorities to select the number of lights that should be active at one time.

The shadowing system is based on shadow mapping and renders the scene to a texture from the view point of the light. The texture is then applied to the receiver of the shadow as an additional texture, projected with the transformation of the light.

The resulting system is integrated with the game engine that is used at Coldwood Interactive and is used in their latest game.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Coldwood Interactive is a game company and was founded 2003 from, mostly, ex-employees of Daydream Interactive. Their first game was published in late 2004, called Herman Maier's Ski Racing 2005, and they have since released the sequel Herman Maier's Ski Racing 2006. The use of lighting in their games today is limited, but by creating a framework to allow both dynamic lighting and shadows the company hopes to improve the graphical look of their games. A key component of any game is good graphics, and lighting of the scene is a large part of it.

1.1 Problem description

The complexity of 3d-scenes for games are increasing every day. An important part is the lighting, but lighting of scenes today is often static and can not be affected by the player. By using pixel shaders for rendering dynamic lights, the lighting can be changed during runtime. The problem with dynamic lights is that only a limited number of light sources can be used in the calculations. A system that can query what lights are strongest from a certain position in a scene would be desirable. The system should also use priorities for different light sources, such as the light from a weapon which should be more important than the ambient lighting.

By integrating shadows into a scene, the lighting effects will be even more realistic and dramatic. It would be useful to have a consistent system that integrates both the shadows and the dynamic lighting, using the same kind of priorities.

1.2 Goals

The goal of this thesis is to investigate and implement a method for determining which lights should be used when rendering dynamic objects/objects using pixel shaders. Implementation of pixel shaders for different number of lights and light sources will be done for testing. A dynamic shadowing system should also be implemented.

1.3 Methods

After the in depth-study is completed an implementation of the system will be done in the Gamebryo 3d-engine that is licensed by Coldwood Interactive.

Chapter 2

Theory

This section will describe the theoretical basis of scene rendering, illumination models and shaders.

2.1 Shaders

The rendering of a scene is accomplished on the graphics card or the graphics processing unit(GPU), the geometry and other information that describes the scene is sent to the graphics card and a fixed number of states are set to determine how it should be processed. This is called the fixed-function pipeline. The fixed function pipeline is described in figure 2.1. The first half of it is the part that operates on individual vertices, transforming them, clipping and so on. The second half handles individual pixels, and does texturing, fog calculation and rasterization.

Instead of using the fixed function pipeline to set the parameters for rendering, a small program can be written that is executed on the GPU. These programs are called shaders. They can be written in both assembler-like or high-level, C-like, languages such as Nvidia CG, OpenGL GLSL and Microsoft HLSL. Shaders are split into two different parts, one is the vertex shader and the other is the pixel shader. The vertex shader replaces the first part of the fixed-function pipeline, the part that operates on vertices where transforms, some lighting, culling and clipping is done. A pixel shader operates

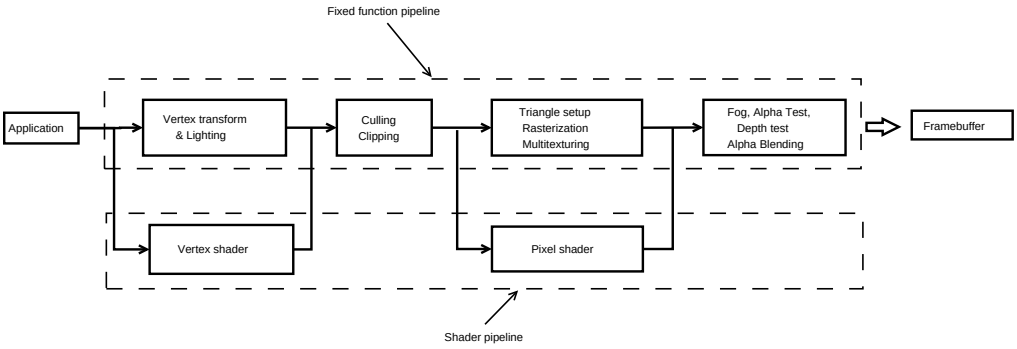

Figure 2.1: Rendering pipeline

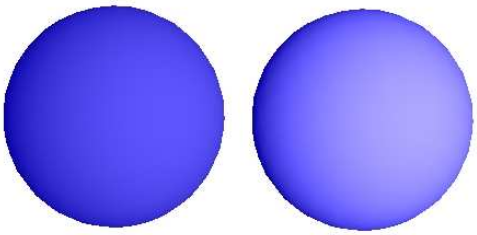

Figure 2.2: Lambert lighting for different values of I_L

on pixels and usually does texturing, fog, alpha blending and depth testing.

2.2 Illumination models and shading

Light is very important for our perception of a three-dimensional scene, because without lighting it is very hard to judge both the distance to and the shape of objects. A light model is a model of how photons interact with the materials of a scene. Given a light source, a surface/scene and a viewer it describes the intensity and color of the light that reaches the viewer from each surface. Several different light models have been developed and a few of the more common ones are described here.

2.2.1 Lambert lighting

If a surface reflects incoming light equally in all directions the surface can be said to be a perfectly diffuse reflector, or a Lambertian reflector. This means the luminosity of the surface, its brightness, is independent of the location of the viewer. According to Lambert's cosine law, the reflected luminosity in an arbitrary direction from a diffuse surface depends on the cosine of the angle between the direction of the incoming light and the normal, see figure 2.3. The consequence of this is that, for a Lambertian surface, the luminosity only depends on the normal vector N and the normalized vector L in the direction between the light and the surface, an example can be seen in figure 2.2 [FvDFH96].

The color of the surface can thus be calculated by taking the dot product of the normal vector N and the view vector L. The result is then multiplied by the intensity of the light source and the surface material color.

$$I_d = I_L k_d cos\phi, \tag{2.1}$$

where I_d is the lighted surfaces final color, I_L is the light sources intensity and k_d the materials diffuse-reflection coefficient [FvDFH96]. If L and N are normalized the equation 2.1 can be rewritten as

$$I_d = I_L k_d (L \cdot N) \tag{2.2}$$

The equation above should be applied to each drawn pixel to calculate its intensity. In practice an ambient term is also added to the equation to simulate that surfaces that are not lighted explicitly still has a small amount of light on it, the indirect light.

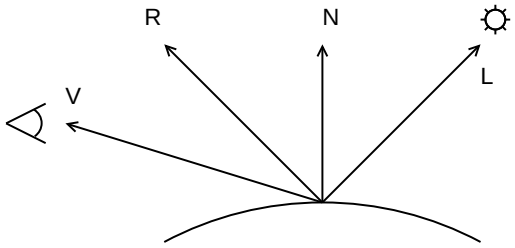

Figure 2.3: Definitions used for the light calculations

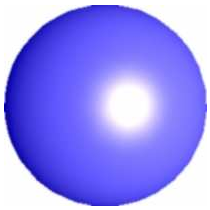

Figure 2.4: Specular highlight

$$I = I_a k_a + I_L k_d (L \cdot N), \tag{2.3}$$

where I_a is the intensity of the ambient light and k_a is the material ambient-reflection coefficient. Another property of light that needs to be accounted for is the falloff of the intensity in distance. The falloff of light will be denoted f_{att}.

$$f_{att} = min(\frac{1}{c_1 + c_2 d_L + c_3 {d_L}^2}), \tag{2.4}$$

where d_L is the distance to the light source and c_1, c_2 and c_3 are constants defined by the user. [FvDFH96].

2.2.2 Phong lighting

Bui Tuong Phong extended Lambert's light model with a component that simulates light that is reflected in a certain direction instead of equally in all directions [Pho75], it is called specular component of the light. This can for example be seen on an orange lit by a single light, the orange will get a highlight, a brighter spot on it with the color of the light, which varies with the viewers position see figure 2.4. Phong's light model assumes that surfaces are not perfect mirrors. A perfect mirror being a mirror that reflects light in only one direction and only a viewer that is on the reflected light ray can see the reflection. By approximating the scattering of the light in a certain direction the light model takes into account that the maximum amount of light is reflected when the angle between the view vector and the reflected lights vector is zero. In practice the light intensity is a more complicated function of the angle ϕ between the reflected light ray and the view angle. Phong used an empirical model to approximate the amount of reflected light that is described in equation 2.5.

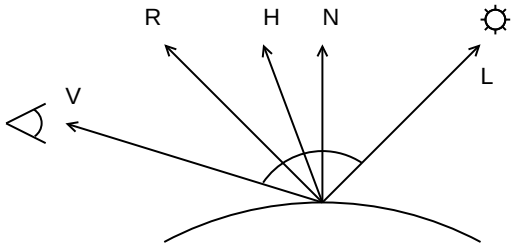

Figure 2.5: Definition of the half-vector

$$W(i)\left[cos(\phi)\right]^n, \tag{2.5}$$

where ϕ is the angle between the vector of the reflected light R and the vector V from the viewer to the surface, see figure 2.5. n is a value that determines the reflectivity of the surface, Phong uses values between one and ten in [Pho75]. $W(i)$ is a function which gives the relationship between the reflected light and the incoming light as a function of the angle of the incoming light in relation to the normal, Phong expressed this value in percent and used between 10 and 80 percent in his paper [Pho75].

Phong's original equation is often rewritten as follows.

$$I_s = (cos^n\phi) * I_L * C_s \tag{2.6}$$

As in equation 2.2, the equation can, if both R and V are normalized, be rewritten as

$$I_s = (R \cdot V)^n * I_L * C_s, \tag{2.7}$$

where I_s is intensity of the reflected light, I_L is the light source intensity and C_s is the materials specular coefficient. To combine this with the diffuse and ambient contributions the complete light equation is

$$I = I_a k_a + I_L * ((L \cdot N) * C_d + (R \cdot V)^n * C_s) \tag{2.8}$$

Blinn reformulated Phong's model for reflected light in his paper on a physically based light model [Bli77], he used a vector called the half-vector, H, instead of R. H is a vector that is defined as the vector halfway in between the light- and view vector and is defined in equation 2.9, see also figure 2.5 for the definitions used.

$$H = \frac{L + V}{|L + V|} \tag{2.9}$$

Using the half vector for calculation of the reflected light the light equation becomes

$$I_s = (N \cdot H)^n * I_L * C_s \tag{2.10}$$

Blinn's way of calculating the reflected light gives a number of advantages. If both the light and the viewer are at infinity, H is constant. The amount of calculations needed for Blinn lighting is decreased because the calculation of the half-vector is simpler than for the R vector. The difference in lighting, between Blinn's and Phong's way of calculating, is noticeable as the highlights on the object are more spread out with Blinn's, this can be seen in figure 2.6.

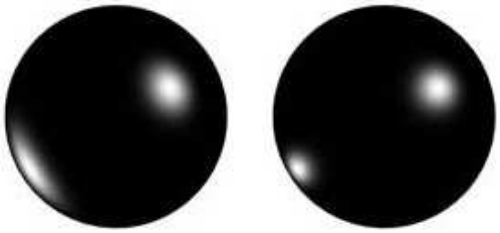

Figure 2.6: The difference in the highlights for Blinn, on the left, and Phong, the right, lighting

2.2.3 Physically based light models

There are some problems with the modeling of light that Phong and Blinn suggested, most of them caused by the fact that the models had no physical basis. The values used for the constants k_d, k_a, k_s and n are not deduced from a value for a specific material or a physical property but are rather empirical values, as is the intensity of the light.

A physically based light model was introduced by Blinn in [Bli77] which was based on a theoretical model by Torrance-Sparrow [TS67]. Blinn realized that the specular component C_s in Phong's lighting equation 2.8 is not constant but rather varies with the direction toward the light source and the peak specular highlight is not always in line with the half-vector H. Blinn used Torrance-Sparrow's model to explain and compensate for these effects and saw that it resulted in a good match between the theoretical and practical results. Torrance-Sparrow's model assumes the surface consists of a large amount of mirror-like microscopic facettes rotated in random directions and that the specular reflection is when these facets are rotated in the direction of the half-vector H. The diffuse part of the light comes from reflections between different facets. The specular component is calculated in equation 2.11.

$$R_s = \frac{FDG}{N \cdot V}, \tag{2.11}$$

where R_s is the specular term, F is the Fresnel term which describes how the light is reflected from each micro facet on the surface. The Fresnel is calculated as

$$F = \frac{1}{2}\left(\frac{sin^2(\phi-\theta)}{sin^2(\phi+\theta)} + \frac{tan^2(\phi-\theta)}{tan^2(\phi+\theta)}\right), \tag{2.12}$$

where ϕ is the angle of incidence, $sin\theta = sin\frac{\phi}{n}$ and n is the index of refraction. D is the facet slope distribution, the part of the facets oriented in the direction H. There are several different models for this, and Blinn describes one of them as

$$D = e^{-(\alpha c_2)^2}, \tag{2.13}$$

where α is the angle between H and N and c_2 is the standard deviation for the distribution.

G in equation 2.11 is the geometrical attenuation factor which accounts for shadowing and masking between individual facettes on the surface. There are three different cases, when facettes shadow each other from the lights, when facettes mask the light source and when the light is directly reflected [CT82].

First is the shadowing case

$$G_b = \frac{2(N \cdot H)(N \cdot E)}{(E \cdot H)}, \tag{2.14}$$

where E is the eye vector. Then the masking is calculated

$$G_c = \frac{2(N \cdot H)(N \cdot L)}{(H \cdot L)} = \frac{2(N \cdot H)(N \cdot L)}{(E \cdot H)} \tag{2.15}$$

And finally the minimum of these two together with, the case where the light is directly reflected, are used for the geometrical attenuation factor.

$$G = Min(G_c, G_b, 1) \tag{2.16}$$

The light model described above is only one of many different physically based light models. Some others are Oren-Nayar, Minnaert and Ward's anisotropic model.

The visual difference between the physically based light models and the Phong model are not large but the computational complexity is much greater for the physically based models. This is a big factor in choosing the right light model for the applications that Coldwood Interactive will be using the light manager for. Since Coldwood Interactive is a game company performance is very important and the Phong model is chosen for implementation. Another factor in choosing Phong is that the graphical artists at Coldwood Interactive have designed levels with shaders that use a similar lighting model.

2.3 Bump mapping

To create detailed scenes, a way of approximating not only smooth, but also rough and rugged surfaces is necessary. This can be modeled in several different ways, the number of polygons/surfaces can be increased, or texture mapping can be used. Texture mapping involves mapping a image onto a surface to give it a more realistic look. There are problems with both these approaches, texture mapping often makes the surface seem very smooth, and increasing the surface count increases the amount of work the GPU has to do and the memory requirements.

Blinn realized that the effect irregularities have on a surfaces lighting is mostly dependent on the normal of the surface rather than the position. Therefore the surface roughness can be approximated by changing the normals instead of adding extra polygons/surfaces. The normal vectors perturbation Blinn defined as a function that gives the changed position in respect to the smooth surface. The function $F(u, v)$ gives, for each point on the surface how much the normal should be perturbed. The surface is parametric with respect to the position P on the surface, see figure 2.7, the partial derivatives with respect to u and v is $\frac{dP}{du}$ and $\frac{dP}{dv}$. The geometrical interpretation of $F(u, v)$ is a height field that changes the position $P(u, v)$. In equation 2.17 the changed vector $P'(u, v)$ is defined.

$$P'(u, v) = P(u, v) + F(u, v)\frac{N(\vec{u}, v)}{|N(u, v)|} \tag{2.17}$$

The new normal is thus defined as

$$N'(u, v) = N(u, v) + \frac{\frac{dF(u,v)}{du}(N(u, v) \times \frac{dP(u,v)}{dv}) - \frac{dF(u,v)}{dv}(N(u, v) \times \frac{dP(u,v)}{du})}{|N(u, v)|} \tag{2.18}$$

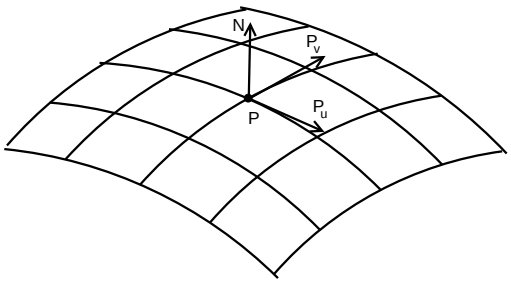

Figure 2.7: Blinn's definitions

Figure 2.8: Comparison between a bump mapped and a smooth sphere

The equation 2.18 is evaluated for each pixel. To use the new normal N' in the lighting equations it must also be normalized, the normalization must be done to keep the bumps the same size independent of the size of the surface.

Blinn suggested that the equation 2.18 could be geometrically interpreted in two different ways. The first sees the right part of the equation as an offset vector, called D, of N where D is defined as

$$D = \frac{\frac{dF(u,v)}{du}(N(u,v) \times \frac{dP(u,v)}{dv}) - \frac{dF(u,v)}{dv}(N(u,v) \times \frac{dP(u,v)}{du})}{|N(u,v)|} \tag{2.19}$$

$$N' = N + D \tag{2.20}$$

The offset vectors can be represented in a so called offset map, where the changes in the normal vector are stored in a texture.

The second geometrical interpretation is that the changed vector N' is calculated by rotating N around an axis in the tangent plane for the surface. The axis is the crossproduct of N' and N and since N, N' and D are all in the same plane

$$N \times N' = N \times (N + D) = N \times D \tag{2.21}$$

An example of bump mapping looks can be seen in 2.8.

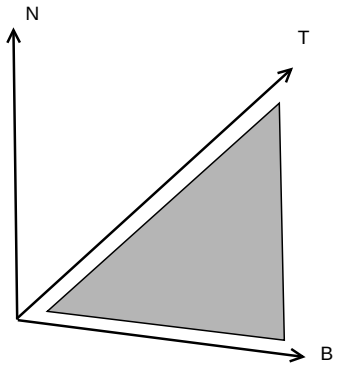

Figure 2.9: Tangent space of a polygon

2.3.1 Tangent space

Percy, Airy and Cabral suggests in [PAC97] a way of using existing hardware to calculate the bump mapping equation, see equation 2.18. They realized that the change in the normal is only a function of the local space of the surface, since the bump mapping that Blinn in [Bli78] describe assumes the surface in each point is locally flat, this space is called the tangent space. The tangent space can be defined by a matrix that is defined by the normal vector N, a tangent vector T and a binormal vector B, see figure 2.9. T is defined as $\frac{\frac{dP(u,v)}{du}}{|\frac{dP(u,v)}{du}|}$ and $B(u,v)$ as $(N(u,v) \times T(u,v))$. The binormal vector and tangent vector together with the normal creates a orthonormal coordinate system that the bump mapping can be performed in. The new normal can be defined in tangent space by

$$N'_{TS} = \frac{(a,b,c)}{\sqrt{a^2+b^2+c^2}}, \tag{2.22}$$

where

$$a = -\frac{dF(u,v)}{du}(B(u,v) \bullet \frac{dP(u,v)}{dv}) \tag{2.23}$$

$$b = -(\frac{dF(u,v)}{dv}\left|\frac{dP(u,v)}{dv}\right| - \frac{dF(u,v)}{du}(T(u,v) \bullet \frac{dP(u,v)}{dv})) \tag{2.24}$$

$$c = \left|\frac{dP(u,v)}{du} \times \frac{dP(u,v)}{dv}\right| \tag{2.25}$$

For the lighting calculations a 3x3 matrix with columns T(u,v), B(u,v) and N(u,v) is created which defines the tangent space. This is then used to transform all vectors used in the light calculations. T, B and N should ideally be calculated for each pixel but can be done only for the vertices and then interpolated across the polygon.

By performing all the calculations in tangent space the number of calculations per-pixel are decreased.

2.4 Relief mapping in hardware

Due to the need for rapid development of graphics in games an in-depth study was made in the field of image-space algorithms for adding detail more details to surfaces, specifically Relief mapping. Relief mapping is an extension of the image-space category of algorithms, such as bump mapping, for adding detail to objects. By adding another texture to every surface that describes the height of each point and using that to ray trace in texture space a more detailed representation of the object is possible.

Previous Work

There is a range of different research in image-space algorithms for adding detail to scenes and some of the previous works are detailed briefly below.

Horizon-Mapping

Horizon mapping extends the bump mapping algorithm by adding self shadowing between bumps in the texture. The algorithm works by calculating the visibility for each point on a surface and generates eight different textures describing the four compass points north, south, east and west and the four between them, northeast, northwest, southeast and southwest. These are then used for rendering the shadows during runtime. The biggest problem with this algorithm is that eight extra textures per bump map is a large increase in texture memory, this makes the technique unpractical in real-time applications [SC00].

Displacement mapping

Displacement mapping is another technique for adding detail to a surface. By changing the surface itself instead of only changing the normal such as in bump mapping many of the problems with the other techniques are avoided such as a incorrect silhouette and shadowing problems [Coo84]. The biggest problem with it is that is requires the surface to be subdivided to create the micro polygons for the surface. It is possible to implement on newer GPU:s that have support for accessing textures in a vertex shader, where the texture is used for determining how much the point should be moved along the normal of the surface. [GH99] [PF05]

Parallax mapping

Parallax mapping is an image-space technique that is an extension of texture mapping, it modifies the texture coordinate based on the parallax of the surface. By calculating an offset for the texture coordinate based on the view vector and the height map a new corrected texture coordinate can be used for indexing the texture maps and other maps. It was first mentioned by Inami et. al in [IKY+01] and later extended and popularized by Welsh in [Wel04] with a limit on the offset calculation and example code in vertex- and pixel-shaders.

Parallax occlusion mapping

Parallax occlusion mapping is a relatively recent technique that works by ray-tracing in texture space on a height map. By casting a ray from the current texture coordinate, in a pixel shader, to the eye and calculating where it intersects, the algorithm calculates

an offset for the current texture coordinate. The offset texture coordinate is linearly approximated if it is not exactly on a texel, and is then used for the rendering of the pixel. By casting another ray from the new texture coordinate to the light the texel can be determined as being shadowed or not. The algorithm also dynamically adjusts the number of samples the ray casting does based on the angle between the normal on the surface and the view vector to compensate for artifacts that cause the height field to look flatter with distance. By using the mip map levels as a level-of-detail measurement the algorithm selects between simple normal mapping and parallax occlusion mapping. Soft shadows are also added by sampling the heights along the light vector until the first visible point is met, after this a blocker-to-receiver ratio is calculated and is used to scale the amount of shadowing. [Tat05b] [Tat05a]

Steep Parallax mapping

Steep Parallax mapping uses ray tracing in texture space similarly to Parallax occlusion mapping. It differs by using the mip map level to adjust the sampling so that it is above the Nyquist rate, that is the smallest frequency needed to avoid aliasing. It also uses a linear search through texture space for the intersection point. The results of it is similar to the approach that relief mapping in hardware takes but with a slightly better performance and a little worse appearance of the surfaces. [MM05]

2.4.1 Relief texture mapping

Relief texture mapping is first mentioned in [OBM00] by Oliveira et. al as a CPU-based image-warping algorithm. Oliveira et. al. defines a relief texture as a texture with depth. The depth is stored as an extra 2d texture and is used to calculate the displacement of the texel's of the texture when rendering it with perspective. Relief texture mapping uses separable transforms, that is transforms that can be split into discrete parts such as, only transforming in the X-direction and only in the Y-direction. It uses them to re-render the texture to account for occlusion, correct the silhouettes and map it to the perspective. The separable transforms in this case is one for the vertical and one is for the horizontal perspective.

The algorithm first uses 3d-image warping to project a point from one plane to another using equation 2.26[MB97]:

$$\vec{x_t} = P_t^{-1} P_s \vec{x_s} + P_t^{-1}(C_s - C_t)\sigma_s(u_s, v_s), \tag{2.26}$$

where $\vec{x_t}$ and $\vec{x_s}$ are the coordinates for the new and the old point, P_t and P_s are camera matrices for the source- and destination plane and C_s and C_t are the center of projection for the cameras. $\sigma_s(u_s, v_s)$ is the disparity, that is the difference in an image between the right and the left eye, it is proportional to the distance from the center of projection.

By using a parallel projection camera model to change the texture before it is mapped with perspective onto the polygon, a constant sampling frequency can be used that simplifies the rendering process. Given a parallel projection a point in the new plane can be defined as

$$x = C_s + \begin{pmatrix} a_{si} & b_{si} & f_{si} \\ a_{sj} & b_{sj} & f_{sj} \\ a_{sk} & b_{sk} & f_{sk} \end{pmatrix} \begin{pmatrix} u_s \\ v_s \\ height_s \end{pmatrix} = C_s + P_s \vec{x_s} \tag{2.27}$$

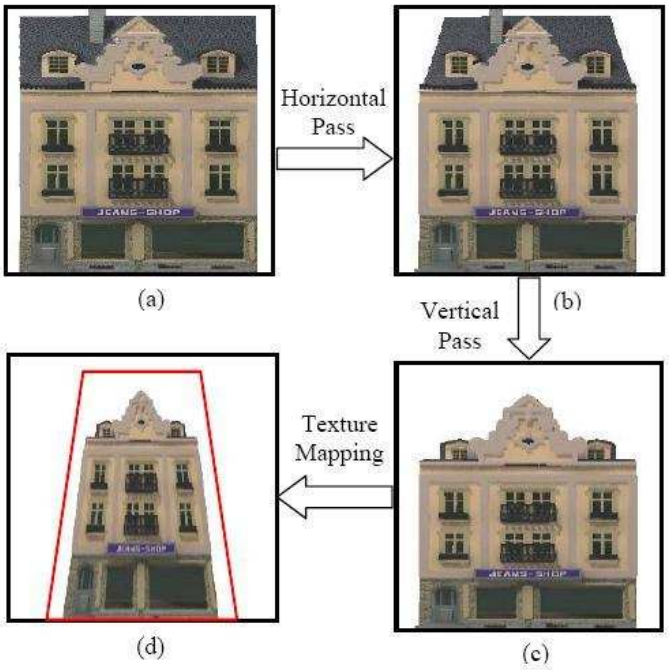

Figure 2.10: The transforms in relief texture mapping, from [OBM00].

To be able to map the new coordinate x to the perspective projection that is to be achieved Oliveira et. al [OBM00] expresses the x coordinate by using both the perspective and parallel camera. By doing this the new position can be written as

$$u_i = \frac{u_s + k_1 height(u_s, v_s)}{1 + k_3 height(u_s, v_s)} \tag{2.28}$$

$$v_i = \frac{v_s + k_2 height(u_s, v_s)}{1 + k_3 height(u_s, v_s)} \tag{2.29}$$

where

$$k_1 = \frac{f \cdot (b \times c)}{a \cdot (b \times c)} \tag{2.30}$$

$$k_2 = \frac{f \cdot (c \times a)}{b \cdot (c \times a)} \tag{2.31}$$

$$k_3 = \frac{f \cdot (a \times b)}{c \cdot (a \times b)} \tag{2.32}$$

where c is the vector from the center of projection to the origin on the perspective plane.

These equations transform all the points of the source image by the height field. After this has been done the points have to be mapped back to a texture which will be applied to the polygon. A two step process is used to reconstruct the image since the

Operation	Time taken in percent
Transformation, projection and reconstruction of image	94.1%
Uploading to texture memory	2.65%
Texture mapping	0.066%
Other	3.18%

Table 2.1: CPU-time spent on relief mapping, from [OBM00]

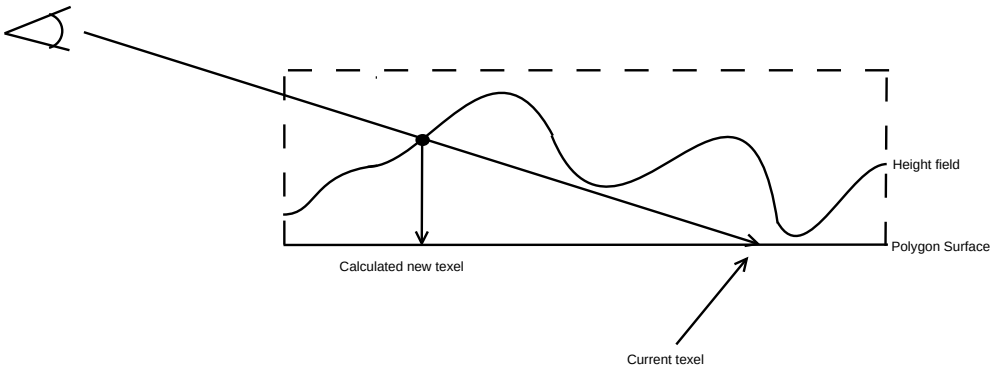

Figure 2.11: Tracing a ray into a height field

equations in 2.28 and 2.29 are separable. First the horizontal pass is done by moving each pixel in the source image to the correct column in the destination image. After this the vertical pass moves the pixels to the correct row in the destination image, see figure 2.10.

The relief texture mapping algorithm proposed is not practically usable today because of the large amount of work that must be performed on the CPU. The measured distribution of time in their implementation can be seen in table 2.1 as run on a Pentium II 400Mhz with an Intergraph 3d-card with 16MB texture memory and 16MB frame buffer memory. The table shows that over 95% of the CPU-time is used to transform, project and reconstruct the image and only 0.06% is used for the rendering of the result. This much time spent on the CPU is not acceptable since as much as possible of the work should be offloaded to the GPU maximum efficiency.

Fujita et. al attempted a hardware implementation by using register combiners on GeForce 256 GPU:s but it was shown that it was slower than the software implementation in half the cases tested and only marginally faster in the rest [FK02].

2.4.2 Hardware-accelerated relief mapping

To avoid the problems with the CPU-overhead in the earlier implementation of relief texture mapping Oliveira et. al. [POC05] revised their algorithm to do all the work on the GPU. The algorithm is based on ray tracing on the GPU.

Real-time relief mapping is, just like bump mapping, done in tangent space. All vectors are transformed to tangent space in a vertex shader and forwarded to the pixel shader which does most of the work. First a linear search down into the height field, starting from where the view vector intersects the polygon, is done to find the first

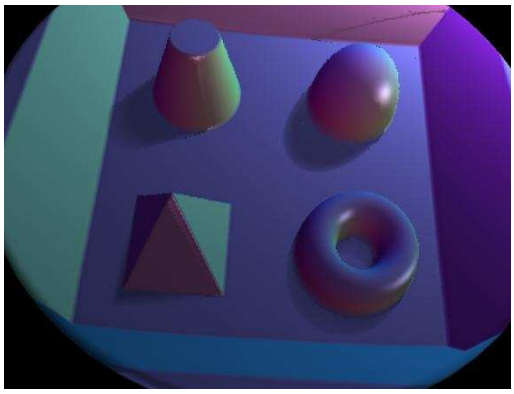

Figure 2.12: Relief mapped polygon disc with soft shadows

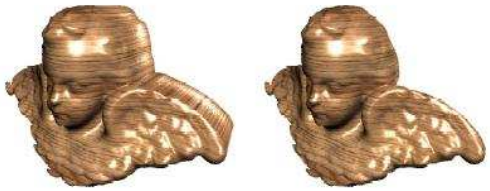

Figure 2.13: A single polygon with normal relief mapping to the left and dual-depth relief mapping to the right, from [POC05]

point inside the height field. After finding the point inside, a binary search is done by splitting the search interval in the middle, between the point inside the height field and the previous step that was outside of it. The point found in the binary search is used to index the texture- and normal map for lighting calculations[POC05].

An extension of this, that is also mentioned in the work on Parallax Occlusion mapping is shadows. By tracing a ray from the intersection point in the height field to the light source and testing for collisions with the height field, a point can be determined to be in shadow or not. By also counting the number of steps necessary for going through the height field after the intersection a soft shadow can be approximated. This is done by modulating the shadows color by the step count.

Another extension that Oliveira et. al mentions in [POC05] is dual-depth relief textures(DDRT). When rendering a relief mapped object that has a back side, the relief mapping algorithm has no information about what is there and a "'skin"' is created where the silhouette would have been. To eliminate this problem a second height field is stored, for the backside. DDRT can then be used to approximate a polygon-object with only one relief-mapped polygon, see figure 2.13.

To make the relief mapped geometry interact with the other geometry in the scene the depth value of the pixels must be calculated and output. This is done in [POC05] by using the far and near clipping planes:

$$out_{depth} = \frac{plane_{far}(z_e + plane_{near})}{z_e(plane_{far} - plane_{near})}, \tag{2.33}$$

z_e is the z-coordinate eye-space and the $plane_{near}$ and $plane_{far}$ are the distances of

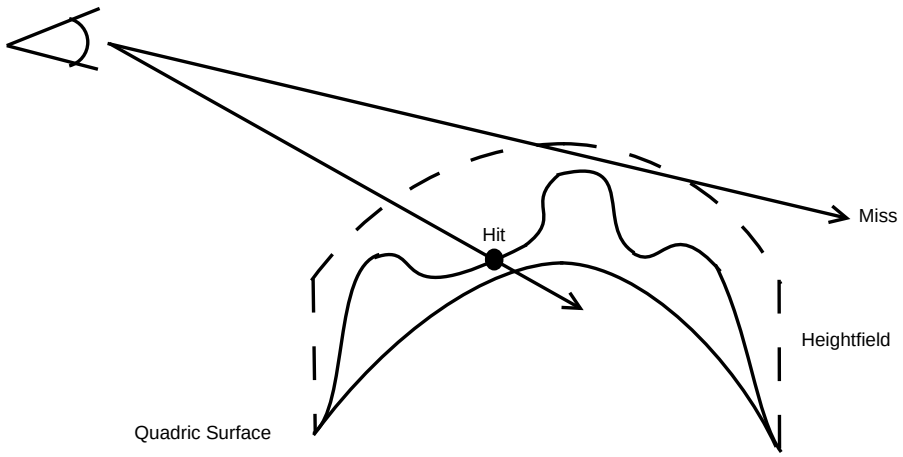

Figure 2.14: The height field is fitted to a quadric surface.

the respective clipping planes.

2.4.3 Correct silhouettes

Relief mapping in hardware gives correct looking surfaces but the same problems as bump mapping, that the silhouette is completely smooth. The problem is caused by the ray missing the height field and then continuing into the next tiled part of the height field always hitting something. To correct this Policarpo et. al uses local quadric approximations of the surface to make the ray casting take into account the silhouette[PO05].

The height field is deformed based on the shape of the quadric approximation. When casting the ray from the light onto the height field, a test against the quadric surface is also performed, and if it does not hit the deformed height field the ray is ignored, see figure 2.14.

By a pre-process where a quadric surface is fitted to the vertices of the model, the quadric surface is then interpolated across the polygon in tangent space. To calculate the approximated quadric surface Policarpo lets T be the triangles which share a vertex v_k with the coordinates (x_k, y_k, z_k) and V be a set containing all the vertices belonging to T [Pet02]. Given V' being the set that contains $(x_i - x_k, y_i - y_k, z_i - z_k)$ the coefficients for the quadric surface can be calculated by first being adapted to

$$z = ax^2 + by^2 \tag{2.34}$$

Policarpo et. al then solves for the coefficients a and b by solving the linear equation system $Ax = b$

$$\begin{pmatrix} x_1 - x_k & y_1 - y_k \\ x_2 - x_k & y_2 - y_k \\ \dots & \dots \\ x_n - x_k & y_n - y_k \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} z_1 - z_k \\ z_2 - z_k \\ \dots \\ z_n - z_k \end{pmatrix} \tag{2.35}$$

the coefficients a and b are then interpolated across the surface to enable the calculation of a distance to it. By defining P, the point on the ray that they wish to calculate the distance to and t as the parameter for the surface

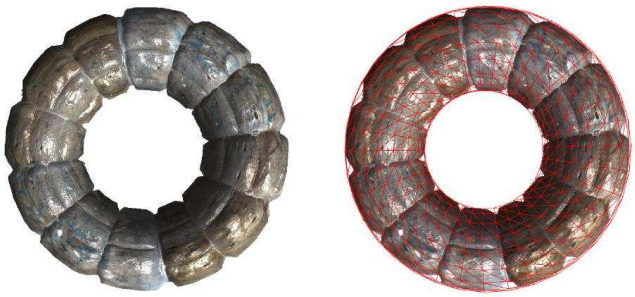

Figure 2.15: Correct silhouette on a relief mapped torus, from [POC05]

$$P = Vt \tag{2.36}$$

to approximate the distance the difference between the z-coordinate of P and the z-coordinate of the quadric surface in (P_x, P_y)

$$PQ_{distance} = P_z - (a(P_x)^2 + b(P_y)^2) \tag{2.37}$$

To get a correct result the ray-to-height field testing must be done in texture-space and when the quadric surface is in tangent space it must be transformed. To do this Policarpo first defines s_x and s_y to be the real size of the texture in tangent space and s_z that scales the height field along the z-axis. s can be seen as the scaled coordinates in tangent space and the definition of the quadric surface is then

$$z_t s_t = a(x_t s_x)^2 + b(y_y s_y)^2 \tag{2.38}$$

$$z_t = a(\frac{{s_x}^2}{s_z}){x_t}^2 + b(\frac{{s_y}^2}{s_z}){y_t}^2 \tag{2.39}$$

A picture of the result can be seen in figure 2.15.

2.5 Shadows

Shadows are as important as lighting for judging positions and shapes of objects in a scene. There are two major classes of algorithms for creating shadows, the first is shadow volumes, where the geometry of the shadow casters silhouette is extruded in the direction of the light source. The second is shadow mapping, performed by rendering the depth of the scene, as seen by the light, to a buffer and using this depth to perform a test of what is visible from the point of view of the light.

2.5.1 Shadow volumes

Crow introduced in [Cro77] the concept of shadow volumes, a geometry defining the shape of the part of the scene being in shadow. By first determining which edges that are a part of the silhouette from the lights viewpoint and then extruding these in the same direction, to a finite or an infinite distance, a shadow volume is created. The

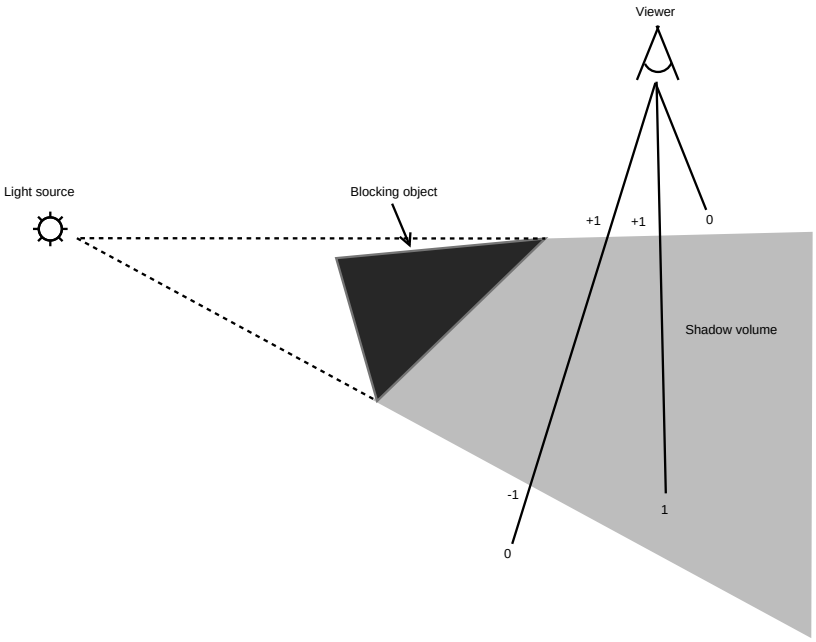

Figure 2.16: Shadow volume counting

silhouette determination is one of the most expensive part of the rendering of shadow volumes since it is mostly done on the CPU.

There are a number of different methods for determining which polygons that form the silhouette of the geometry as seen from the light. Most methods pre-calculates the connectivity information of all the edges and uses the stipulation that any edge that is shared between two polygons that have opposite facing with respect to the light, have to be a part of the silhouette. There has been some research into GPU based silhouette extrusion [BS03] but the effectiveness of it seems to be the same or worse than the CPU-based variants [Yaz06].

To be able to determine if a point is in shadow the stencil buffer is used. By counting the number of front- and back facing shadow polygons in the stencil buffer as they are rendered, increasing when encountering a front facing and decreasing on a back facing polygon, a point is determined to be in shadow if the value in the buffer is positive, see figure 2.16.

1. Scene rendered with ambient light only.
2. For each light.
   (a) Calculate the silhouettes of the geometries from the lights point of view.
   (b) Extrude the silhouettes.
   (c) Render the shadow volumes to the stencil buffer counting the front- and back facing polygons in them.
   (d) Perform the lighting, and shadowing where the stencil buffer is positive and normal lighting where it is not.

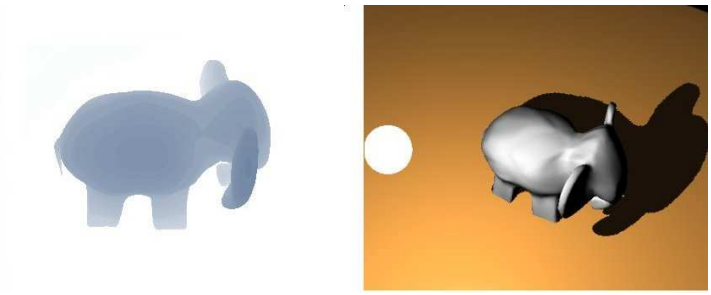

Figure 2.17: Depth map from the lights point of view to the left, and final scene to the right.

There are a lot of special cases that need to be handled to make the algorithm robust, what happens when the viewer is inside a shadow volume, how does the clipping of the shadow volumes to the view frustum work and how to cap the shadow volumes if they are not infinite in extent?

A disadvantage with this algorithm is that the amount of fill-rate that is needed to render the shadow volumes can be prohibitive for a scene with complex geometry. Complex geometry is also a problem for the performance of the silhouette detection since the algorithm needs to traverse the scene to extract all the silhouettes. If anything other than polygon data is used, for example parametric surfaces, a different silhouette extraction algorithm must be used. Another requirement is that the polygon objects are completely closed, there can be no holes in them.

The advantages of shadow volumes are that no aliasing, as is present in shadow mapping, can occur since the shadowing is determined from geometry. Another advantage is that point lights can easily be implemented, in contrast to shadow mapping with which point lights are difficult to implement efficiently.

2.5.2 Shadow mapping

Shadow mapping was first proposed by Williams in [Wil98], the algorithm is an image-based technique, in contrast to the geometry based shadow volumes, for calculating which parts of the scene are shadowed. By realizing that the points that are not visible from the lights point of view are the points that will be in shadow the algorithm uses the depth values that are written to the Z-buffer to test for visibility. As the scene is rendered from the viewer the depth values from the current point is transformed into the lights coordinate system and compared. If the distance in the lights depth map is greater than the value of the current point being rendered, the point is not shadowed since the light is not occluded, and vice versa if the lights depth maps value is greater the point is in shadow since an object is in front of the light [CEC02].

The projection of the depth map can be done in hardware by sending the projection and view matrices either to the texture coordinate generation, or to a shader that calculates the correct texture coordinates.

The steps of the algorithm are then:

1. Render the whole scene from the lights point of view.

2. Project the depth buffer of the light onto the scene.

3. Render the scene, comparing the depth values of the projected depth buffer and the scene.

There are a number of advantages with shadow mapping over shadow volumes. As was mentioned in the shadow volumes section the shadow volume algorithm can only handle polygonal meshes on which it can extrude the silhouette, shadow mapping on the other hand can handle anything that can be rendered to the frame buffer. Another advantage is that it is independent of the complexity of the geometry. [Sch05].

The fact that shadow mapping works in image-space and samples the depth in a regular grid causes aliasing problems due to the finite resolution of the depth map. Point lights are problematic when using shadow mapping because the scene must be rendered from the point of view of the light, and point lights has a spherical view which is hard to represent with a frustum. By splitting the frustum in six parts, all oriented in a different direction, a complete view can be rendered from the lights view point. Rendering six different views of the scene however, is not efficient [BAS02].

2.6 Level-of-detail for lights

The light manager must be able to handle when the number of visible light sources are larger than the number allowed, in the shaders and does this by deactivating some of them. The deactivation of light sources should be done in such a way that the viewers perception of the scene changes as little as possible. In [Ber02] Berka describes factors that affect the perception of an object. The distance of the object from the viewer is one of the factors, as the distance increases the objects perceived size decreases, the same can be said for a light. A factor that applies only to lights is the intensity of the light source, a very high intensity light will affect more of the environment and thus cause greater visual change if it is deactivated. Another factor is the angular distance of the object or light source from the focal point on the screen, the higher angular distance the less focus there is on the object.

Berka in [Ber02] uses equation 2.40 to relate the angular distance to the visual acuity

$$f(\theta) = \begin{cases} 1 & \text{when } 0 \leq \theta \leq \alpha \\ exp\left(-\frac{\theta-\alpha}{c_1}\right) & \text{when } \alpha < \theta \end{cases} \tag{2.40}$$

Where θ is the angular distance between the focal point and the object/light sources center, α is the angular distance between the focal point and the part of the object, light sources bound, that is closest and c_1 is a scale factor.

This is due to the sensitivity of the rod and cone cells in the eye decreasing with increasing angular distance. All these factors together can be used in determining a priority for an object or a light source.

The velocity of an object also affects the way it is perceived, the higher velocity the harder it is to make out details on it.

According to Funkhouser in [FS93] another way of decreasing the amount of calculations for lighting is to change the light model based on the same criteria as described above. The Cook-Torrance light model could for example be used when the light is closest to the viewer, at a farther distance Phong-Blinn could be used and farthest away a simple Lambertian light model could suffice. The difference in lighting between the models would be hidden due to the amount of screen space of the objects decreasing as the light model changes.

Chapter 3

Implementation

The implementation is done in the Gamebryo 3d-engine that is licensed by Coldwood Interactive.

3.1 Lighting model with shaders

As the shaders had to use vertex- and pixel-shader version 1.1 to be compatible with as many GPU:s as possible, some compromises had to be done in the light model. This is due to the number of instructions in the pixel shader being limited to eight. To be able to use the Phong-Blinn lighting model in a pixel shader a number of steps must be accomplished for each pixel.

1. Multiply the normal from the normal map with the light vector in tangent space.
2. Multiply the normal from the normal map with the view vector in tangent space.
3. Multiply the diffuse and ambient colors.
4. Raise specularity to the power of n.
5. Multiply together all the components with the texture used.
6. Calculate falloff value and multiply it with the final light value.
7. Output the result as a color.

This approach however causes problems if several lights has to be applied. To use several lights the rendering has to be broken up into several different passes, this due to the limitation in number of instructions of the pixel/vertex shader. By calculating only the diffuse and specular lighting in the first passes and adding the ambient lighting in the last pass, several lights can be blended together. This approach is used in the implementation. An implementation of the Phong-Blinn lighting model was also done in shader model 2.0 and was used for spotlights.

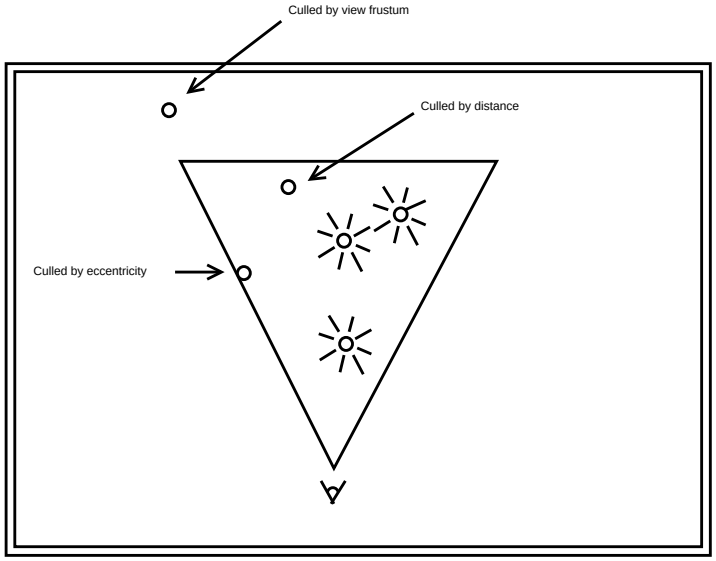

Figure 3.1: Culling with the light manager

3.2 Lighting algorithm

The algorithm used is divided into two parts, where the first is done each frame and the second part is for each object. The first part does a broad culling based on the bounding volumes of the light and the view frustum and removes the lights that are not visible. It also calculates some distances that are used in the per object part of the algorithm and stores these for each light. The bounding volumes used in the implementation were exclusively spheres since the point lights have a spherical falloff, a possible future improvement would be for other kind of lights, for example spot lights, to calculate a convex hull or a cone-like bound. The first part of the algorithm is shown below:

1. For each frame.
   (a) Test lights bounding box against view frustum.
   (b) If the light intersects or is in the view frustum.
      i. Calculate distance from the camera to the light and weigh priority based on it.
      ii. Calculate angular distance from camera focal point to the light source and weigh the priority.
      iii. Insert into visible lights list.

The second part is done for each object and sets the shaders constants before the rendering. It first tests if the object and the lights bounding volume intersects to be able to cull away lights that can not affect the object. For all lights that can affect the object the priority is calculated according to equation 3.1 and the lights are inserted into a priority queue. By then removing the lights one by one from the priority queue, with the highest priority one first, and setting the shader constants for the light, the correct lights will be used for the object. The object is finally rendered.

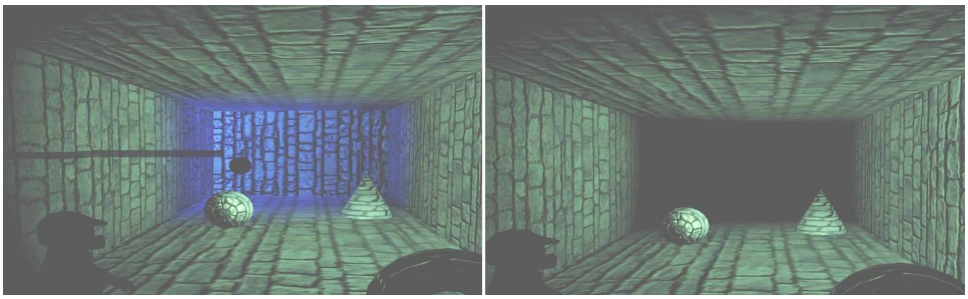

Figure 3.2: The change in lighting from one frame to another when a light is switched off due to lowered priority

1. For each object.
   (a) Test lights bounding box against objects bounding box.
   (b) If the bounds intersect or are enclosed in each other.
      i. Calculate lights distance to the object and weigh priority based on it.
      ii. Insert into priority queue.
   (c) For each object in priority queue or until maximum number of lights are enabled.
      i. Fetch highest priority light from priority queue.
      ii. Set shader constants for the light.
   (d) While number of lights enabled are less then maximum number of lights.
      i. Set intensity of light to zero.
   (e) Empty priority queue.
   (f) Render object.

A problem with this algorithm is that when the number of lights that are visible exceeds the number specified in the shader a priorization between the lights must be done and some turned off. This causes an abrupt change in lighting that is noticeable for the user. To minimize this issue an algorithm was implemented to increase or decrease the intensity for lights that are being turned on or turned off. The lighting change is then not as abrupt as in figure 3.2.

The implementation of the fading uses three lists for each object that contains the lights that affect it and a state machine that handles when the lights are on, off or fading for each object. The fading algorithm is however not used in practice since the amount of memory and processing required is too large.

3.3 Calculation of light priorities

The calculation of the priority of an object is done through a simple weighting system that accounts for several different factors.

$$P_{final} = f(\theta)w_{ecc} + d_{viewer}w_{dist} + d_{object}w_{object} + iw_{intensity} + p_{user} \tag{3.1}$$

The first part of the equation $f(\theta)w_{ecc}$ is the eccentricity of the light source with respect to the viewer, $f(\theta)$, this is not as heavily weighted as the others due to testing showing that it caused a large change in lighting often.

The second part $d_{viewer}w_{dist}$ is more important, d_{viewer} is the euclidean distance to the viewer and w_{dist} is the weight.

$d_{object}w_{object}$ is the distance between the object and the light source. Light sources closer to the objects center should be given a higher priority than the objects further away since the closer ones are more likely to affect the object more.

$iw_{intensity}$ is the intensity of the light source, since the lights intensity decreases with distance this is integrated with the distance between the object and the light.

The last part of the equation, p_{user}, is a user defined priority that can be used to give higher priority to certain objects, for example a flashlight in the hand of the player should always be on, so by increasing the priority of it it will be turned of less often.

3.4 Integration with the Gamebryo shader system

The Gamebryo shader system uses a text based format to describe how a shader is connected to the render pipeline. The format defines a number of global shader constants, vertex/texture coordinate streams and passes, each pass with an associated vertex and/or pixel shader. The passes can all contain local-pass-specific shader constants, texture samplers and render states. The local shader constants can be defined to use a specific register in the GPU, by writing a generic shader that uses some fixed registers for its parameters a framework for lighting can be created.

A shader for one point light for example, could take the point lights position, color of the diffuse light and the camera position in registers 0, 1 and 2. A Gamebryo shader would then define positions and colors for the number of lights to be handled as global shader constants and each pass would be one light. By setting the local shader constants for each pass to the registers defined(as in this example 0, 1 and 2) the information for each light would be correct. The code below shows the constants and how they map to different passes.

Global constants

```
Global:
{
PointLight1Position
DiffuseColorLight1
PointLight2Position
DiffuseColorLight2
.
.
.
PointLight<n>Position
DiffuseColorLight<n>

CameraWorldPosition
}
```

Local-per-pass constants

```
Pass 1
{
PointLight1Position  -> register 0
DiffuseColorLight1 -> register 1
CameraWorldPosition -> register 2
}
Pass 2
{
PointLight2Position  -> register 0
DiffuseColorLight2 -> register 1
CameraWorldPosition -> register 2
}
.
.
.
Pass <n>
{
PointLight<n>Position  -> register 0
DiffuseColorLight<n> -> register 1
CameraWorldPosition -> register 2
}
```

The shader system and shader format in Gamebryo is well suited to a static number of passes and can manipulate many of the render states in the pipeline [Eng04]. It is also adapted to ease the amount of control the artist has by allowing constants and variables to be manipulated through plugins for the major modeling packages. The problem with it is that it does not allow dynamic number of passes which would have been practical to allow passes to be added on the fly, a further discussion can be found in "'Limitations"'.

3.5 Shadows

The shadow system implemented is based on shadow mapping but is simplified by omitting some parts. Shadow mapping was chosen due to the ease of integration with the Gamebryo engine and the game Coldwood Interactive was developing. There are already systems in place for rendering to a texture and the existing render pipeline could be used without any large modifications. Shadow volumes would have required connectivity information to be integrated into the geometry classes and a larger modification of the rendering pipeline. The render pipeline would have to be split into several passes each with a specific render state, and since the current game engine is tightly integrated with how the rendering is done the limited time available would not allow a good integration of shadow volumes. Skinning is also used heavily in the game and is performed in a vertex shader. To get a correct shadow volume for skinned objects the skinning must be calculated in software to be able to determine the silhouette of the object, this would however be inefficient.

The algorithm implemented works by rendering only the objects that will cast a shadow to a texture with a single color, no lighting or texturing. The objects are rendered from the point of view of the light and use either a special "'shadow object"'

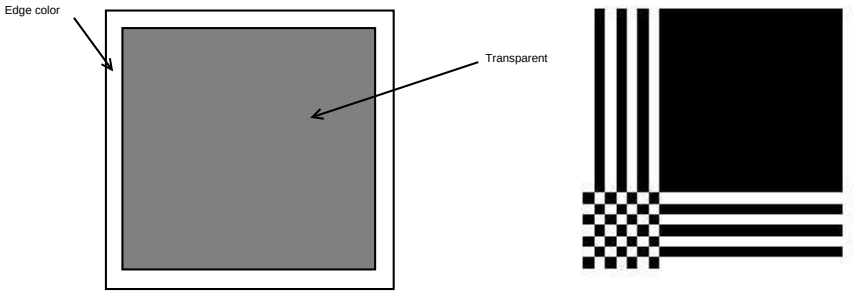

Figure 3.3: The edge texture used to fix clamping on the left and the problem with texture clamping on the right

that the artists has created as a simpler, lower polygon count, version of the original object, or the original object itself if no "'shadow object"' has been created. The texture is then projected onto the objects receiving it by using the lights view matrix in a shader.

A problem with the current approach is that the shadow texture must use the clamped texture mode so that the shadows do not repeat over the terrain. Since clamping uses the edge color of the texture, see figure 3.3, if the edge color is a shadowed pixel the edge color will be repeated/stretched across the terrain. To alleviate this a texture containing the clear color is applied on top of the shadow texture before it is applied to the terrain, see figure 3.3.

The final algorithm for rendering the shadow texture is thus:

1. Setup the camera at an infinite distance in the direction of the light and looking at the target.
2. Cull objects not in view frustum.
3. For each object.
   (a) If a "'shadow object"' exists for the current object.
      i. Get the "'shadow objects"' from the current object and insert into shadow list.
   (b) else
      i. If a clone of the object does not exist.
         A. Create clone of object with a very simple shader that only does transformations and coloring.
      ii. Insert clone into shadow list.
4. Batch render all objects in the shadow list to the shadow texture, no depth testing is necessary.
5. Apply edge texture on top of the shadow texture.
6. Render all the objects, use the shadow texture as a darkening texture map.

The shadow texture rendered is used in the shader for the terrain as a final layer of texturing. The texture coordinates are calculated based on the view matrix and

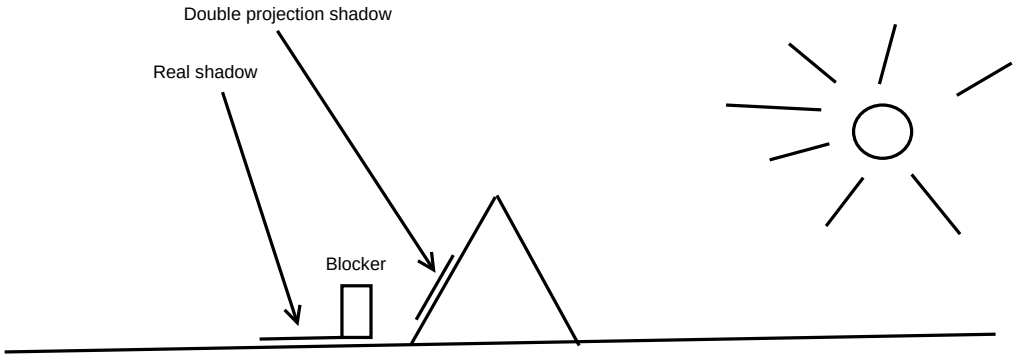

Figure 3.4: Double projection

projective texturing. There were several problems with this approach, the most obvious being double projection. Double projection is the projection that occurs on places that do not cast shadows but are blocked from the light, see figure 3.4. By using a simple test if the polygon is back-facing in the vertex shader, and if it is, not apply the shadow texture in the pixel shader, most of the artifacts caused by this can be avoided.

This algorithm is easy to extend for several lights by doing the same steps for each light to different textures and sending each, together with the projection transformations, to a pixel/vertex shader to be combined with the other.

Chapter 4

Results

The result of this thesis is a light manager that works as designed in Coldwood Interactive's current game, and is used for managing the 22 dynamic lights on the last level of the game. A two lights per pass shader with four lights in total per object was used. The visual results are good and the performance is acceptable, the fading algorithm is not used due to the large performance hit that it caused. The shadowing is used in the game as well, see figure 4.1.

A problem with the algorithm is that it will not scale well for many lights. Since the testing is done against all the lights, the performance should scale linearly. The algorithm has been tested on a Pentium 4 3.2Ghz with a ATI Radeon X700 card. With no lights on, and a standard shader with only one directional light, the performance averages around 30 frames per second, with a minimum of 18 and a maximum of 57. If all lights are turned on, 22 for this level, the frames per second average drops to around 25, a decrease by 20%. See figure 4.2 and figure 4.3. The amount of triangles rendered were between 50.000 and 140.000 depending on what was visible on the level. As can be seen in the diagram the amount of frames per second is not visibly affected by the amount of lights enabled. Even increasing the number of passes does not seem to affect the number of frames-per-second noticeably.

Another problem is that it is hard to batch render all the geometry when the shader constants must be set for each geometry. According to Wloka[Wlo03], when rendering, batching is very important and the bound on the amount of polygons rendered are the

Figure 4.1: The game SXR where the shadows and the lighting has been integrated

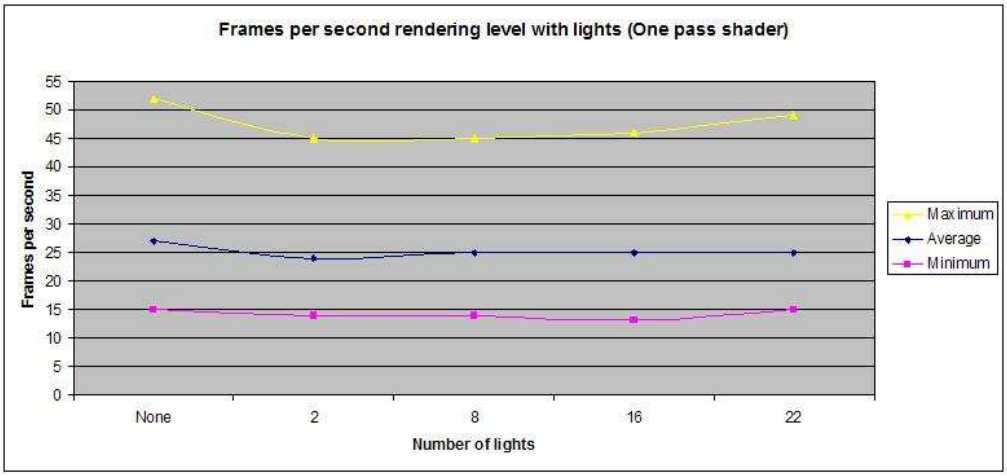

Figure 4.2: One pass shader benchmark

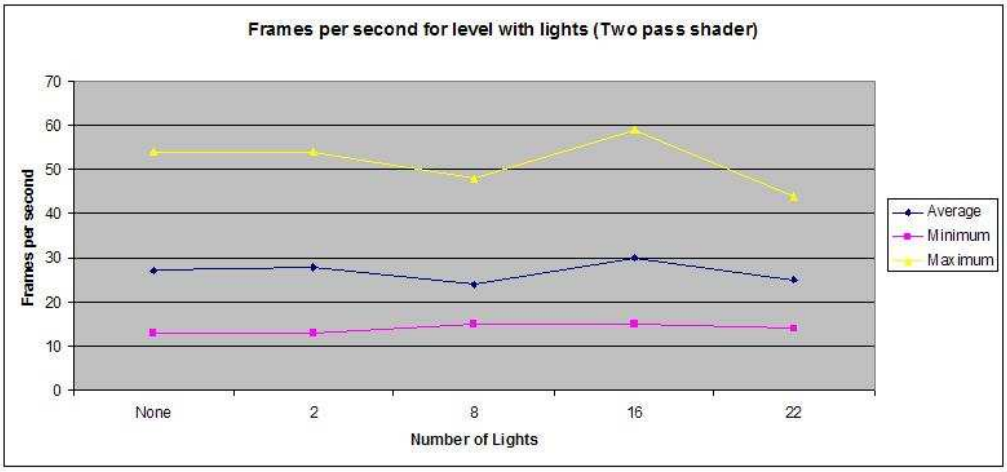

Figure 4.3: Two pass shader benchmark

number of batches that can be sent to the GPU. For a per pixel lit scene, managing the number of batches is important due to the way the scene must be structured. For example, say the scene consists of a house with ten rooms where each room has three different materials and is lit by four light dynamic light sources. To render this 4*3 batches per room are needed and with ten rooms this gives 1200 batches to be sent to the GPU every frame. Wloka cites that about 10,000-40,000 batches a second can be sent to the GPU on a 1 Ghz CPU using 100% of the CPU [Wlo03]. Normally allocating all the CPU-time to send batches is not possible, so a reasonable estimation is that ten percent can be used for that [PF05]. This means that around 1000-4000 batches would be the expected amount which gives one to four of the houses in the example could be rendered. One solution to the problem is to design the scenes so that the per-pixel light sources affect as few objects as possible and thereby decreasing the amount of batches.

The batching problem could be a large part of the slowdown seen when enabling the light manager.

See "'future work"' for further discussion.

Chapter 5

Conclusions

The goals of this thesis were reached but there are a number of areas that could be improved in the implementation. Both performance and quality of the light manager and the shadows could be improved in a number of ways, these are discussed in the sections below.

5.1 Limitations

There are a number of limitations in the approach detailed above and these are mentioned in the following sections.

5.1.1 Visual quality

The limitations in pixel shader 1.1, the low maximum instruction count and few registers, makes the visual quality of the lighting suffer. The specular, diffuse, ambient and falloff components of the light source are possible to calculate in eight instructions but there are some artifacts on the lighted surfaces. One of the most obvious artifacts are due to fact that the half-vector and light vector is only normalized in the vertex shader and is then interpolated across the surface. This interpolation denormalizes the vectors which makes highlights spread out instead of focus. Another problem that is caused by the limited number of instructions is that the specular component can't be raised higher than to the power of 16 or any values that are not a multiple of two. This makes it hard for the artist to control the highlights on objects. These limitations were removed when the shaders were ported to shader model 2.0.

5.1.2 Shader pass problems

A problem with the implementation of the light manager is that the number of passes used when the scene/geometries can not be changed dynamically. The Gamebryo shader system is not designed with this in mind and it would require a rewrite of a large portion of it to accommodate to dynamically changing the number of passes. This means that the shader must be written with a maximum number of lights in mind which causes the scene to always be rendered with a set number of passes. If for example a shader was written with four passes and one pass per light, a scene with only one light active would still require all four passes to be rendered but with three of them not affecting the scene.

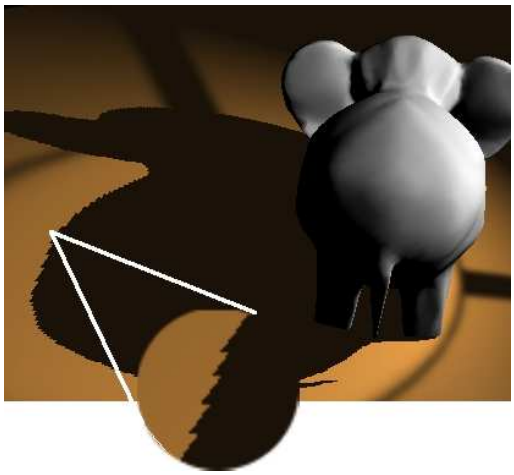

Figure 5.1: Aliasing when using shadowmapping

A possible solution to this is the create a number of different shaders/shader definitions which describes one, two, three and so on, lights and apply the shaders with a smaller number of lights on geometries in the scene where the number of dynamic lights are excepted to be small. And vice versa with large number of lights.

5.1.3 Shadow aliasing

The shadow mapping technique is prone to aliasing, as can be seen in figure 5.1, due to the limited resolution of the shadow texture. A implementation of a simple blur filter was done to help make it less obvious but the performance hit was too great to make it viable to include in the final implementation as anything other than just an option.

5.2 Future work

There are a number of things that could be done to improve the quality and performance of the implementation. One obvious thing is to improve the fading algorithm since it was not used when integrating with the game engine at Coldwood Interactive. The light manager and the shadow system are also not fully integrated, a preliminary testing using the light manager for selecting a single directional light used by the shadow system showed that it was viable. However, no time was left for a complete integration.

5.2.1 Occlusion culling

An extension of the simplistic culling implemented in the light manager could be occlusion culling. Occlusion culling is a culling technique that culls objects that are not visible because they are occluded by other objects. There are a number ways that this can be accomplished, one that could be used for the light manager is to cast a ray toward the center of the lights bounding volume to determine if anything is between the viewer and the light source center. If the light source center is tested not visible, the priority of the light could be reduced since, either the light is not visible or a lesser part of it is visible. There are also a number of techniques for doing the culling in image space, for example

Hierarchical Occlusion Maps(HOM) [ZMHH97], Hierarchical Z-Buffer(HZB) [GKM93], and several algorithms that use shadows to determine occlusions [CT96], but all these algorithms require the scene be rendered in a specific order, and would be better suited to implement as a general way of performing occlusion testing rather then only for light sources.

5.2.2 Tree structures

Another addition that could be made to the implementation, which will become necessary with larger scenes, is some sort of spatial partitioning. As scenes get larger the amount of light sources will go up and, as the algorithm used tests all lights against all objects, the amount of tests increases exponentially. By using an octree or a quadtree to partition the scene hierarchically and thus only test against objects close to the light source the amount of tests could be decreased.

5.2.3 Better shadow mapping

An implementation of real shadow mapping instead of the variant used in this thesis would be beneficial in the future since it would allow self-shadowing. Light-space perspective shadow maps [MWP04], Trapezoidal shadow maps [MT04] or Tiled shadow maps [Arv04] would help with the aliasing problems inherent in the original technique. A more general way of handling all sorts of light, from point lights to spot light and directional lights, would also be necessary to create a complete lighting system for all conditions. An improved blurring technique instead of the four-sample averaging that was used would improve the aliasing of the edges of the shadow and make the shadows more realistic looking.

Chapter 6

Acknowledgments

I would like to thank my supervisor Anders Backman for all the comments and help with the project. I would also like to extend a thank you to all the employees at Coldwood Interactive, especially my supervisor there Håkan Dalsfelt for helping me get started, Jakob Marklund and Rikard Häggström for helping me in various ways during the project, and all the rest for making me feel welcome.

References

[Arv04] Jukka Arvo. Tiled shadow maps. In *Proceedings of Computer Graphics International 2004*, pages 240–247. IEEE Computer Society, 2004.

[BAS02] Stefan Brabec, Thomas Annen, and Hans-Peter Seidel. Practical shadow mapping. *J. Graph. Tools*, 7(4):9–18, 2002.

[Ber02] Roman Berka. *Level of Motion Detail in Virtual Reality*. PhD thesis, Czech Technical University in Prague, 2002.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, New York, NY, USA, 1977. ACM Press.

[Bli78] James F. Blinn. Simulation of wrinkled surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 286–292, New York, NY, USA, 1978. ACM Press.

[BS03] Stefan Brabec and Hans-Peter Seidel. Shadow volumes on programmable graphics hardware. *Eurographics 2003*, 2003.

[CEC02] Ashu Rege Cass Everitt and Cem Cebenoyan. Hardware shadow mapping. *ACM SIGGRAPH 2002.*, 2002.

[Coo84] Robert L. Cook. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-138-5.

[Cro77] Franklin C. Crow. Shadow algorithms for computer graphics. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 242–248, New York, NY, USA, 1977. ACM Press.

[CT82] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, 1982.

[CT96] Satyan R. Coorg and Seth Teller. Temporally coherent conservative visibility. In *Proc. 12th Symp. Computational Geometry*, pages 78–87. ACM, May 1996.

[Eng04] W. Engel. *ShaderX2: Shader Programming Tips and Tricks with DirectX 9*. Wordware Publishing Inc., 2004. ISBN 1-55622-988-7. 631–650 pp.

[FK02] Masahiro Fujita and Takashi Kanai. Hardware-assisted relief texture mapping. In *Proc. EUROGRAPHICS 02*, 2002.

[FS93] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics*, 27(Annual Conference Series):247–254, 1993.

[FvDFH96] James Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics. Principles and Practice. 2nd Edition in C.* Addison-Wesley, 1996. ISBN 0-201-84840-6. FOL j 96:1 1.Ex.

[GH99] Stefan Gumhold and Tobias Hüttner. Multiresolution rendering with displacement mapping. In *HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 55–66, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-170-4.

[GKM93] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 231–238, New York, NY, USA, 1993. ACM Press. ISBN 0-89791-601-8.

[IKY+01] Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, Tomomichi Kaneko, Toshiyuki Takahei, and Susumu Tachi. Detailed shape representation with parallax mapping. *Proceedings of the ICAT 2001 (The 11th International Conference on Artificial Reality and Telexistence)*, pages 205–208, 2001.

[MB97] Leonard McMillan Mark, William (Bill) and Gary Bishop. Post-rendering 3d warping. In *1997 Symposium on Interactive 3D Graphics*, pages 27–30. Providence, RI April, 1997.

[MM05] Morgan McGuire and Max McGuire. Steep parallax mapping. *I3D 2005 Poster*, 2005.

[MT04] Tobias Martin and Tiow-Seng Tan. Anti-aliasing and continuity with trapezoidal shadow maps. *Eurographics Symposium on Rendering*, 2004.

[MWP04] Daniel Scherzer Michael Wimmer and Werner Purgathofer. Light space perspective shadow maps. *Eurographics Symposium on Rendering*, 2004.

[OBM00] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 359–368, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. ISBN 1-58113-208-5.

[PAC97] Mark Peercy, John Airey, and Brian Cabral. Efficient bump mapping hardware. *Computer Graphics*, 31(Annual Conference Series):303–306, 1997.

[Pet02] Sylvain Petitjean. A survey of methods for recovering quadrics in triangle meshes. *ACM Comput. Surv.*, 34(2):211–262, 2002.

[PF05] Matt Pharr and Randima Fernando. *Gpu Gems 2: Programming Techniques for High-Performance Graphics and General Purpose Computation.* Addison-Wesley, 2005. ISBN 0-321-33559-7.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[PO05] Fábio Policarpo and Manuel M. Oliveira. *Advanced Mapping Techniques and Ray Tracing on the GPU*, pages 119–172. A K Peters, Wellesley, MA 02482, USA, 2005. ISBN 1-56881-240-X.

[POC05] Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 155–162, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-013-2.

[SC00] P. Sloan and M. Cohen. Interactive horizon mapping. In *In Rendering Techniques '00 (Proc. Eurographics Workshop on Rendering)*, pages 281–286. Springer, 2000., 2000.

[Sch05] Daniel Scherzer. Shadow mapping of large environments. Master's thesis, der Technischen Universität Wien, 2005.

[Tat05a] Natalya Tatarchuk. Dynamic image-space per-pixel displacement mapping with silhouette antialiasing via parallax occlusion mapping. *GDC '05*, 2005.

[Tat05b] Natalya Tatarchuk. Practical dynamic parallax occlusion mapping. *SIGGRAPH '05*, 2005.

[TS67] K. E. Torrance and E. M. Sparrow. Theory for off-specular reflection from roughened surfaces. *Journal of the Optical society of America*, pages 1105–1114, 1967.

[Wel04] Terry Welsh. Parallax mapping with offset limiting: A perpixel approximation of uneven surfaces, 2004. Technical Report, Infiscape Corporation.

[Wil98] Lance Williams. Casting curved shadows on curved surfaces. *Seminal graphics: pioneering efforts that shaped the field*, pages 51–55, 1998.

[Wlo03] Matthias Wloka. Batch, batch, batch: What does it really mean? *GDC '03*, 2003.

[Yaz06] Shi Yazheng. Performance comparison of cpu and gpu silhouette extraction in a shadow volume algorithm. Master's thesis, Umeå Universitet, 2006.

[ZMHH97] Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. *Computer Graphics*, 31(Annual Conference Series):77–88, 1997.

Appendix A

Source code

The relief mapping shaders in HLSL and tested in ATI's RenderMonkey, needs a card capable of Shader Model 3.0.

A.1 Vertex shader

```
float4x4 matWorldViewProjection: register(c0);
float4 light_position: register(c8);
float4 eye_position: register(c9);
float4x4 matWorldInverse;
struct VS_INPUT_STRUCT
{
   float4 position:     POSITION;
   float3 normal:       NORMAL;
  float2 texcoord0:    TEXCOORD0;
   float3 binormal:     BINORMAL0;
   float3 tangent:      TANGENT0;
};

struct VS_OUTPUT_STRUCT
{
   float4 position:     POSITION;
   float2 texmap:       TEXCOORD0;
   float3 light_vector: TEXCOORD1;
   float3 half_angle:   TEXCOORD2;
   float3 eye_vector:   TEXCOORD3;
   float2 bumpmap:      TEXCOORD4;
};
//this vertex shader is heavily based on the samples in
//ATIs RenderMonkey 1.6, just minor adjustments to the outputs
VS_OUTPUT_STRUCT main( VS_INPUT_STRUCT vsInStruct )
{
   VS_OUTPUT_STRUCT vsOutStruct; //** Declare the output struct

   //**-------------------------------------------------------------
```

```
   //** Calculate the pixel position using the perspective matrix.
   //**------------------------------------------------------------
   vsOutStruct.position = mul( matWorldViewProjection,
   vsInStruct.position );

   //**---------------------------------------------
   //** Pass the bump and base texture coords through
   //**---------------------------------------------
   vsOutStruct.texmap = vsInStruct.texcoord0;
   vsOutStruct.bumpmap = vsInStruct.texcoord0;
   float3x3 tangentspace=float3x3(vsInStruct.tangent,
   vsInStruct.binormal,vsInStruct.normal);

   //**---------------------------------------------
   //** Calculate the light vector in object space,
   //** and then transform it into texture space.
   //**---------------------------------------------
   float3 temp_light_vector =  mul(matWorldInverse,
   light_position ) - vsInStruct.position;

   vsOutStruct.light_vector = mul(tangentspace, temp_light_vector);

   //**---------------------------------------------
   //** Calculate the view vector in object space,
   //** and then transform it into texture space.
   //**---------------------------------------------
//   float3 temp_eye_position =
   float3 temp_view_vector = mul(matWorldInverse,
   eye_position) - vsInStruct.position;

   float3 temp_view_vector2 = mul(tangentspace, temp_view_vector);
   //temp_view_vector2.z=-temp_view_vector2.z;
   vsOutStruct.eye_vector=temp_view_vector2;
   //**-------------------------
   //** Calculate the half angle
   //**-------------------------
   vsOutStruct.half_angle = temp_light_vector + temp_view_vector2;

   return vsOutStruct; //** Return the resulting output struct
}
```

A.2 Pixel shader

```
float4 specular: register(c2);
float Ka: register(c3);
float Kd: register(c4);
float Ks: register(c5);
float specular_power: register(c6);
```

```
float bumpiness: register(c7);
float4 ambient: register(c0);
float4 diffuse: register(c1);
float depth: register(c8);
float shine: register(c9);
float one_over_range;
float parallax;
float tile;
sampler base_map;
sampler reliefmap;
struct PS_INPUT_STRUCT
{
   float2 texcoord:     TEXCOORD0;
   float3 light_vector: TEXCOORD1;
   float3 half_angle:   TEXCOORD2;
   float3 eye_vector:   TEXCOORD3;
   float2 bumpmap:      TEXCOORD4;
};

struct PS_OUTPUT_STRUCT
{
   float4 color0:       COLOR0;
};

//this is based on the CG-sample code that Policarpo included
//in the Relief mapping in hardware paper.

float3 ray_intersect_rm(
      in sampler2D reliefmap,
      in float3 s,
      in float3 ds)
{
  const int linear_search_steps=40;
   const int binary_search_steps=20;

   ds/=linear_search_steps;

   for( int i=0;i<linear_search_steps-1;i++ )
   {
      float4 t=tex2D(reliefmap,s);

      if (s.z<(t.w))
         s+=ds;
   }

   for( int i=0;i<binary_search_steps;i++ )
   {
      ds*=0.5;
      float4 t=tex2D(reliefmap,s);
```

```
      if (s.z<(t.w))
         s+=2*ds;
      s-=ds;
   }

   return s;
}

float3 ray_intersect_rm_shadow(
      in sampler2D reliefmap,
      in float3 s,
      in float3 ds,
      inout float color)
{
  const int linear_search_steps=40;
  const float fadingrate=2.4;
  const float fadingchange=fadingrate/linear_search_steps;
  float3 temps;
   ds/=linear_search_steps;
 //find intersection with heightfield
   for( int i=0;i<linear_search_steps-1;i++ )
   {
      float4 t=tex2D(reliefmap,s);

      if (s.z<(t.w))
         s+=ds;
   }
   temps=s;

   for( int i=0;i<linear_search_steps-1;i++ )
   {
      float4 t=tex2D(reliefmap,s);

      //step through for soft shadows
      if (s.z>(t.w))
      {
         s+=ds;
         color-=fadingchange;
      }
   }

   return temps;
}

PS_OUTPUT_STRUCT main( PS_INPUT_STRUCT psInStruct )
{
   PS_OUTPUT_STRUCT psOutStruct;

   //first trace from current texcoord to view
```

```
float3 startpoint=float3(psInStruct.texcoord*tile, 0);

//scale viewvector
float3 eyevect = -normalize(psInStruct.eye_vector);
eyevect.xy *= depth*(2*eyevect.z-eyevect.z*eyevect.z);
eyevect /= eyevect.z;

float3 modified_texcoord=ray_intersect_rm(reliefmap,
startpoint,eyevect);

startpoint.xy+= eyevect.xy * modified_texcoord.z;

//sample textures at the new texcoord
float3 base = tex2D( base_map, modified_texcoord.xy );
float3 bump = tex2D( reliefmap, modified_texcoord.xy );

float3 lightvect = -normalize(psInStruct.light_vector);

//scale lightvector
lightvect.xy *= (depth)*(2*lightvect.z-lightvect.z*lightvect.z);
lightvect /= lightvect.z;

startpoint.xy-= lightvect.xy * modified_texcoord.z;

float shadowatt=1.0;
float3 modified_texcoord2=ray_intersect_rm_shadow(reliefmap,
startpoint,lightvect, shadowatt);

 shadowatt=max(shadowatt,0);

float3 changed_light_vector=psInStruct.light_vector;
changed_light_vector.y=-changed_light_vector.y;


float3 normalized_light_vector = normalize( changed_light_vector );
float3 normalized_half_angle = normalize( psInStruct.half_angle );

bump = normalize( ( bump * 2.0f ) - 1.0f );

float3 n_dot_l = dot( bump, normalized_light_vector );
float3 n_dot_h = dot( bump, normalized_half_angle );

float att=1.0/(length(psInStruct.light_vector)*one_over_range);

//bias shadow to prevent z-fighting
if(modified_texcoord2.z>modified_texcoord.z-0.05)
   psOutStruct.color0.rgb = ((
      ambient +
      diffuse *  max( 0, n_dot_l )+
```

```
         specular * pow( max( 0, n_dot_h ), specular_power )
         ) * base )* att ;
  else
        psOutStruct.color0.rgb = ( ( ambient+ diffuse *
        max( 0, n_dot_l )*shadowatt)*base)*att;

   psOutStruct.color0.a = 1.0f;

   return psOutStruct;
}
```