

Surface Detail with Textures – Relief Texture Mapping

by: Theresia Hansson

(theresia.hansson@gmail.com)

3 April 2006



Table of Contents

1. INTRODUCTION	2
1.1. THE GOAL OF THIS PROJECT	2
1.2. A SHORT VISUAL COMPARISON OF DIFFERENT TECHNIQUES	2
2. IMPLEMENTATION CHOICES AND SYSTEM REQUIREMENTS.....	4
2.1. SYSTEM REQUIREMENTS.....	4
2.2. PROGRAMMING LANGUAGE, GRAPHICS API AND SHADER LANGUAGE	4
3. THEORY AND IMPLEMENTATION OF SIMPLE TECHNIQUES	5
3.1. BASIC LIGHTING	5
3.2. BUMP MAPPING.....	6
3.3. PARALLAX MAPPING	7
4. THE RTM IMPLEMENTATION	9
4.1. INTRODUCING THE ALGORITHM FOR SIMPLE PLANAR OBJECTS	9
4.2. FOR ARBITRARY SHAPED OBJECTS	12
4.3. SELF-SHADOWS	12
4.4. SILHOUETTE EDGES AND DEPTH CORRECTION	13
4.5. EFFICIENCY	16
5. APPLICATION: OBJECT RENDERING	18
5.1. ONE- AND TWO-SIDED OBJECTS	18
5.2. OBJECTS DEFINED ON A BOUNDING BOX.....	18
5.3. APPLYING OBJECT RELIEF RENDERING TO A CHESS KNIGHT MODEL	20
6. CONCLUSION.....	22
APPENDIX A: SHADER CODE	23
A.1. COMPARETECHNIQUES SHADERS.....	23
A.2. SHADERS FOR RELIEFOBJECT.....	29
APPENDIX B: MESH AND MAP GENERATION.....	32
B.1. THE DEPTH MAP	32
B.2. THE NORMAL MAP	32
B.3. CALCULATING THE TANGENT, BINORMAL AND NORMAL OF A MESH	33
B.4. PER-VERTEX CURVATURES	33
APPENDIX C: REFERENCES	34



1. Introduction

Relief Texture Mapping (RTM) uses depth data stored in a texture to generate a detailed depth surface depending on the viewpoint of the observer.

The technique of relief mapping was originally introduced as a forward mapping technique by Manuel Menezes de Oliveira Neto in his article [RTMOrig00]. This means that the textures were “pre-warped” before rendering took place. Since then, relief texture mapping has developed, and with the arrival of more powerful graphics hardware it can be implemented as a real-time inverse mapping technique (texture coordinates calculated depending on view). The latter of the two approaches is treated in this project, that is, the inverse mapping one.

1.1. The goal of this project

This project was done as a part of the course “Synthèse d’images animées” given at École Polytechnique in France. The goal, as described in the project requirements, was to read and understand some scientific articles in the chosen field of Relief Texture Mapping, as well as implement the techniques described in this article. Besides that, there was also the suggestion to compare the techniques with similar ones.

The decision here was to understand and implement the different relief mapping techniques described in several articles as well as to compare the results and efficiency with the simpler techniques of bump mapping and parallax mapping. I’ve also set up a goal trying to apply the relief mapping technique in some application.


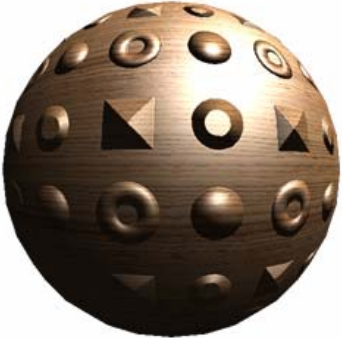
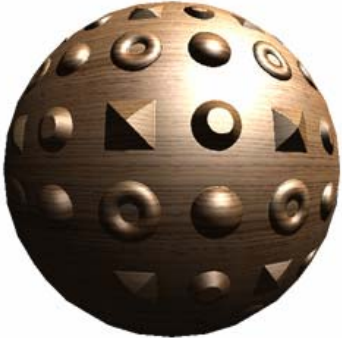

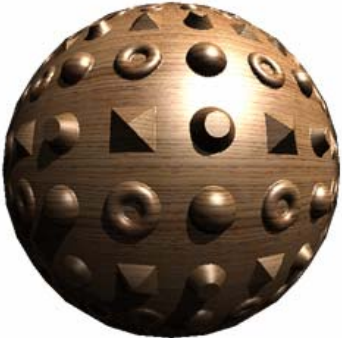
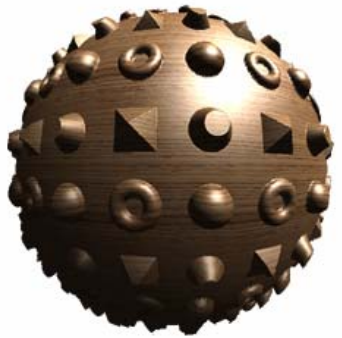
1.2. A short visual comparison of different techniques

As mentioned in the previous section, this project intends to compare and explain some different techniques used. As an introduction, I here present some images rendered with the different techniques described in this document. Figure 1.1 shows a sphere rendered in different ways. The associated demo program – [CompareTechniques](#) – has its shader source codes in Appendix A. It will be explained in parts 3 and 4 of this report.

The image rendered using conventional texture mapping presents no depth at all. In the simple technique of bump mapping one tries to give an illusion of a simple depth structure by applying a normal map to the object in addition to the color texture map used in the conventional texture mapping image. This is then used in lighting calculations so that the sphere is lit in a way that gives the illusion of a depth. But still, the image rendered with bump mapping gives a rather flat impression compared to the simplest of depth mapping techniques: parallax mapping. This technique uses in addition to the color and normal texture maps also a depth texture (these texture types are explained in Appendix B). However the algorithm used to project the pixels in this technique is approximal and doesn’t always give accurate results. There can only be small displacements with parallax mapping. Here relief texture mapping (RTM) comes as a rescue. It gives believable depth mapping results, but uses rather complex calculations which require it to be run on a graphics card supporting shader model 3.0. The images are ordered by increasing complexity and believability. The first one only implements the depth mapping, while the second adds the enhancement of self-shadows and the third takes geometry curvature into account to discard pixels that ought not to be visible at the edges, an enhancement known as silhouette edge correction.

All these techniques are described here, except for the conventional texture mapping one, which is trivial and should already be known to the reader. The approach is a build-up approach which means that every one of these techniques more or less extends previous described ones.



FIGURE 1.1. <i>Renders from CompareTechniques</i>		
Simple techniques (part 3)		
Conventional Texture Mapping (CTM)	Bump Mapping (BM)	Parallax Mapping (PM)
		
RTM techniques – arbitrary shaped objects (part 4)		
Simple	With self-shadows	With correct silhouettes
		



2. Implementation choices and System requirements

2.1. System requirements

The shaders written in this project are very long and complex. With the exception of the Bump Mapping and Parallax Mapping shaders, they all require Shader Model 3.0 functionality which is found on Graphics cards of NVIDIA's Geforce 6 series and higher or on ATI's corresponding cards. This presented a problem on the test computer which didn't have a sufficient graphics card.

2.2. Programming language, Graphics API and Shader language

All programming was done in C++ because of the power and usability of this language.

The original thought was to use the OpenGL API and NVIDIA's CG shader language for the graphical implementations. However, when discovering that the development computer didn't meet the system requirements (it has only Shader Model 2.0 functionality) the choice of Graphics API was **DirectX**. DirectX has a reference driver which enables development of advanced applications on computers that do not actually support all functionality. This was of course ideal in this case. Because of using DirectX enables HLSL (Microsoft's High Level Shader Language), which, as a language, is almost identical to CG, **HLSL** was a natural choice. However the shader codes should, perhaps with small modification, be able to run with OpenGL and CG.



3. Theory and implementation of simple techniques

3.1. Basic lighting

Basic per-pixel lighting according to the phong model uses the here described equations for the lighting calculation.

Normally the lighting calculations are divided into different components: ambient (background color), diffuse (normally reflected color), specular (color from specular highlight) and emissive (self-illuminative color). Here follow the equations to calculate each of these components.

$$\text{ambient_color} = \text{material_ambient} * \text{light_ambient}$$

$$\text{diffuse_color} = \text{material_diffuse} * \text{light_diffuse} * \text{diffuse_intensity}$$

$$\text{specular_color} = \text{material_specular} * \text{light_specular} * (\text{specular_intensity})^{\text{shininess}}$$

$$\text{emissive_color} = \text{material_emissive}$$

For these equations `diffuse_intensity` and `specular_intensity` are calculated in world space as follows:

$$\mathbf{n} = \text{normalize}(\text{surface_normal})$$

$$\mathbf{v} = \text{normalize}(\text{view_pos} - \text{pos})$$

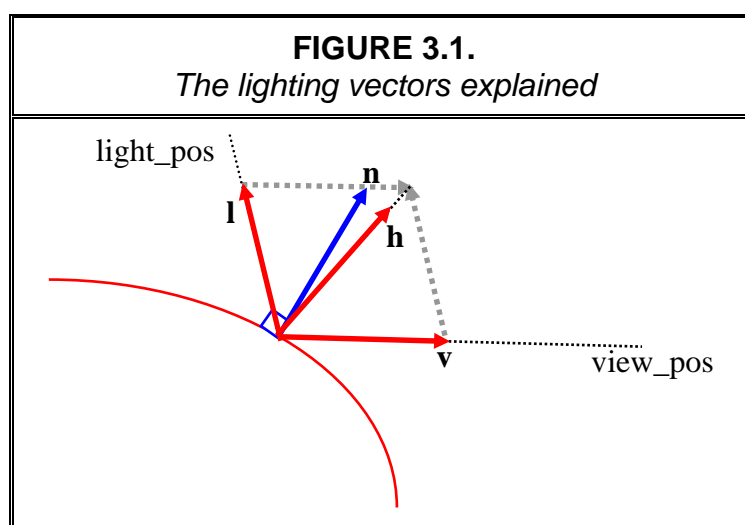
$$\mathbf{l} = \text{normalize}(\text{light_pos} - \text{pos})$$

$$\mathbf{h} = \text{normalize}(\mathbf{l} + \mathbf{v})$$

$$\text{diffuse_intensity} = \max(\mathbf{l} \cdot \mathbf{n}, 0)$$

$$\text{specular_intensity} = \max(\mathbf{h} \cdot \mathbf{n}, 0)$$

The lighting vectors used ($\mathbf{n}, \mathbf{v}, \mathbf{l}, \mathbf{h}$) are explained in figure 3.1.



The final color becomes:

$$\text{final_color} = \text{ambient_color} + \text{diffuse_color} + \text{specular_color} + \text{emissive_color}$$



It should be mentioned that the material components when used with textures in this project take the following values:

material_ambient = *texture_color*

material_diffuse = *texture_color*

material_specular = *white* = 1

material_emissive = *black* = 0

All the demos in this project use only a single light which is an infinite range point light.

Remark

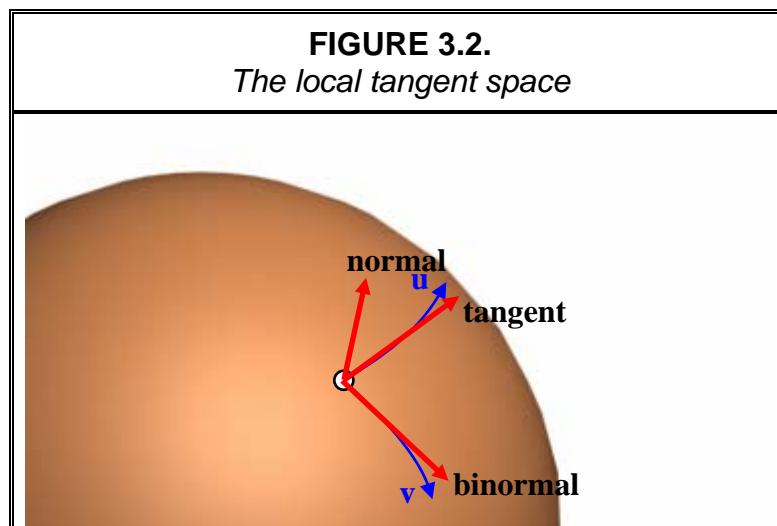
We use per-pixel calculations of the vectors **v**, **l** and **h** instead of interpolating them between vertices, because this will be necessary to get correct lighting of the displaced pixel in the depth mapping algorithms described in this report.

3.2. Bump mapping

Bump mapping is a per-pixel technique that uses a normal map for lighting calculations. The principles are very simple and use almost the same code as in the lighting model described above, however, there are some things that need to be taken into consideration. It is important to transform either the normal or the light and view vectors to be in the same space before doing any calculations. Here, the choice was made to transform the normal into world space. Currently they are defined in a space called local tangent space, and need to be transformed by

$$\mathbf{n}_{\text{world}} = \text{normalize}(\mathbf{n}_{\text{tangent}} \cdot x \cdot \mathbf{tangent} - \mathbf{n}_{\text{tangent}} \cdot y \cdot \mathbf{binormal} + \mathbf{n}_{\text{tangent}} \cdot z \cdot \mathbf{normal}).$$

(Depending on the generation of the normal map there might have to be some sign changes)



The local tangent system, shown in red in figure 3.2 uses the position of the rendered pixel as origo and its tangent, binormal and normal as x,y and z axis respectively. These parameters are passed from the vertex shader. All this requires of course that the vertices fed into the



graphics pipeline have information of this kind. The generation of these extra parameters is described in Appendix B.

Once all lighting information is in the same space the lighting calculations proceed exactly as described in section 3.1. The only difference is that self shadows are introduced by modulating the diffuse and specular components by a parameter *att* defined by

$$att = 1 - (1 - \max(0, \mathbf{normal} \cdot \mathbf{l}))^2.$$

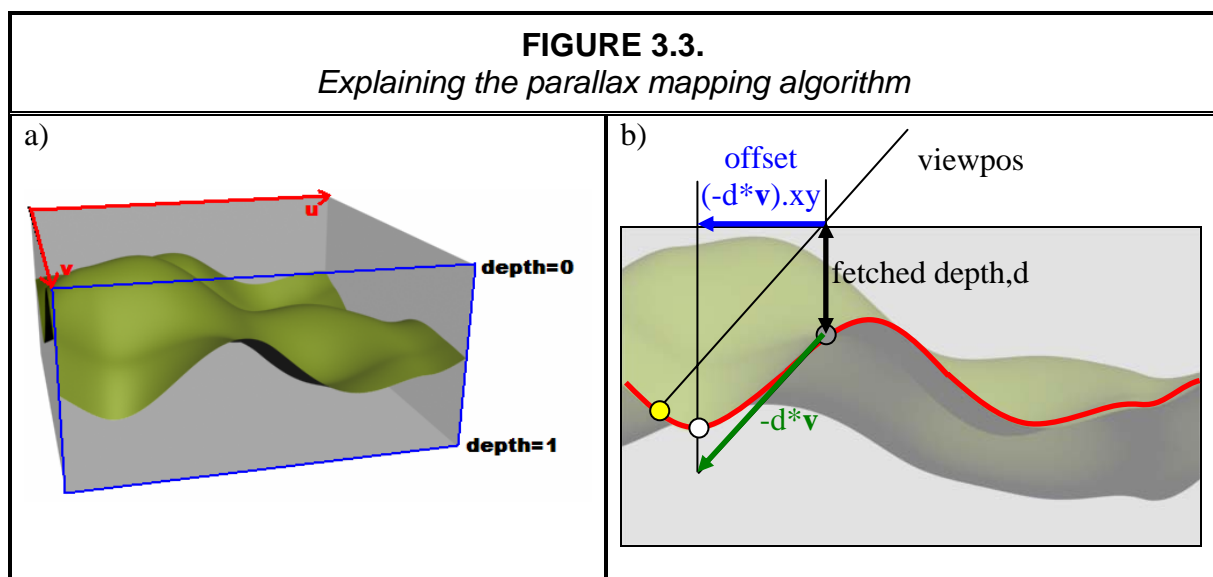
This is necessary because a bump map normal could be pointing a bit towards the light even if it is in shadow from the object (object normal points away from light).

3.3. Parallax mapping

Parallax mapping is a simple depth technique that uses a depth map to define the surface. A simple example of a surface defined by a depth map is shown in figure 3.3.a. To demonstrate the depth techniques in this report we will be using the cross-section outlined in blue, but it is important to remember the third dimension as well. Figure 3.3.b shows the principle of parallax mapping. Basically the algorithm proceeds as follows:

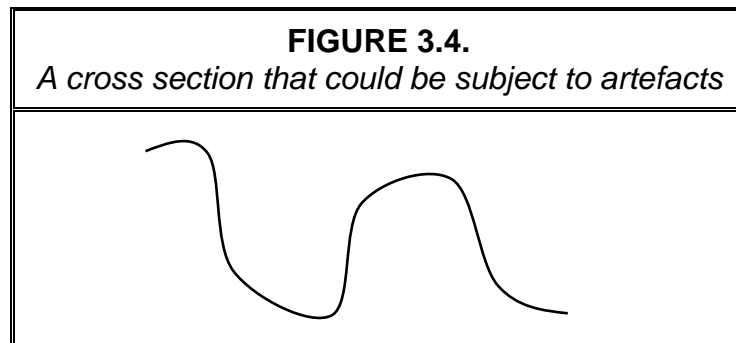
1. Fetch the depth value at the current point (u,v) (depth at the grey point in figure 3.3.b)
2. Use a version of the *v* vector introduced in section 3.1 which is transformed into local tangent space
3. Multiply the negative of this local tangent space view vector with the depth fetched in step 1 (illustrated as the green vector in figure 3.3.b)
4. Add the resulting vector's x and y coordinates to the texture coordinates u and v respectively (the offset arrow drawn in figure 3.3.b)
5. Fetch normal and color information with these new texture coordinates
6. Continue in the same way as for bump mapping with normal-to-world transformation followed by lighting calculations

As can be seen in the figure, the texel used by the algorithm (the white point) is only approximal. The correct value would have been the texel where the intersection between the surface defined by the height map and the view line occurs (the yellow point).





Sometimes there are visual artefacts due to the physical incorrectness of the method. This occurs mainly when the slopes of the depth map surface are too big (for example with a depth map surface cross section as in figure 3.4) and when trying to displace as if the depth was large. For this reason the fetched depth is scaled by a factor of about 0.06.





4. The RTM implementation

4.1. Introducing the algorithm for simple planar objects

The inverse mapping version of Relief Texture Mapping enhances the technique of parallax mapping with a new and better way of finding the intersection of the depthmap-defined surface and the view ray. RTM uses a linear search followed by a binary search to find the correct intersection point. The reason why there are two different searches involved is explained here as are some limitations to the searches.

The linear search technique proceeds as follows:

1. Input parameters are **dp** and **ds** expressed in texture coordinates
2. Let $\text{current_depth} = 0$, $\text{best_depth} = 1$ and $\text{depth_step} = 1/\text{NUM_SEARCH_STEPS}$
3. Increase current_depth by depth_step
4. Fetch the depthmap value for $\text{dp} + \text{ds} * \text{current_depth}$
5. If $\text{best_depth} = 1$ (no intersection found yet) and result from 4 is smaller than or equal to current_depth ($\text{current_depth} - \text{depth_step} < \text{intersect_depth} \leq \text{current_depth}$) Then $\text{best_depth} = \text{current_depth}$
6. If $\text{current_depth} < 1$ go to step 3
7. Return best_depth which corresponds to the first point inside the object

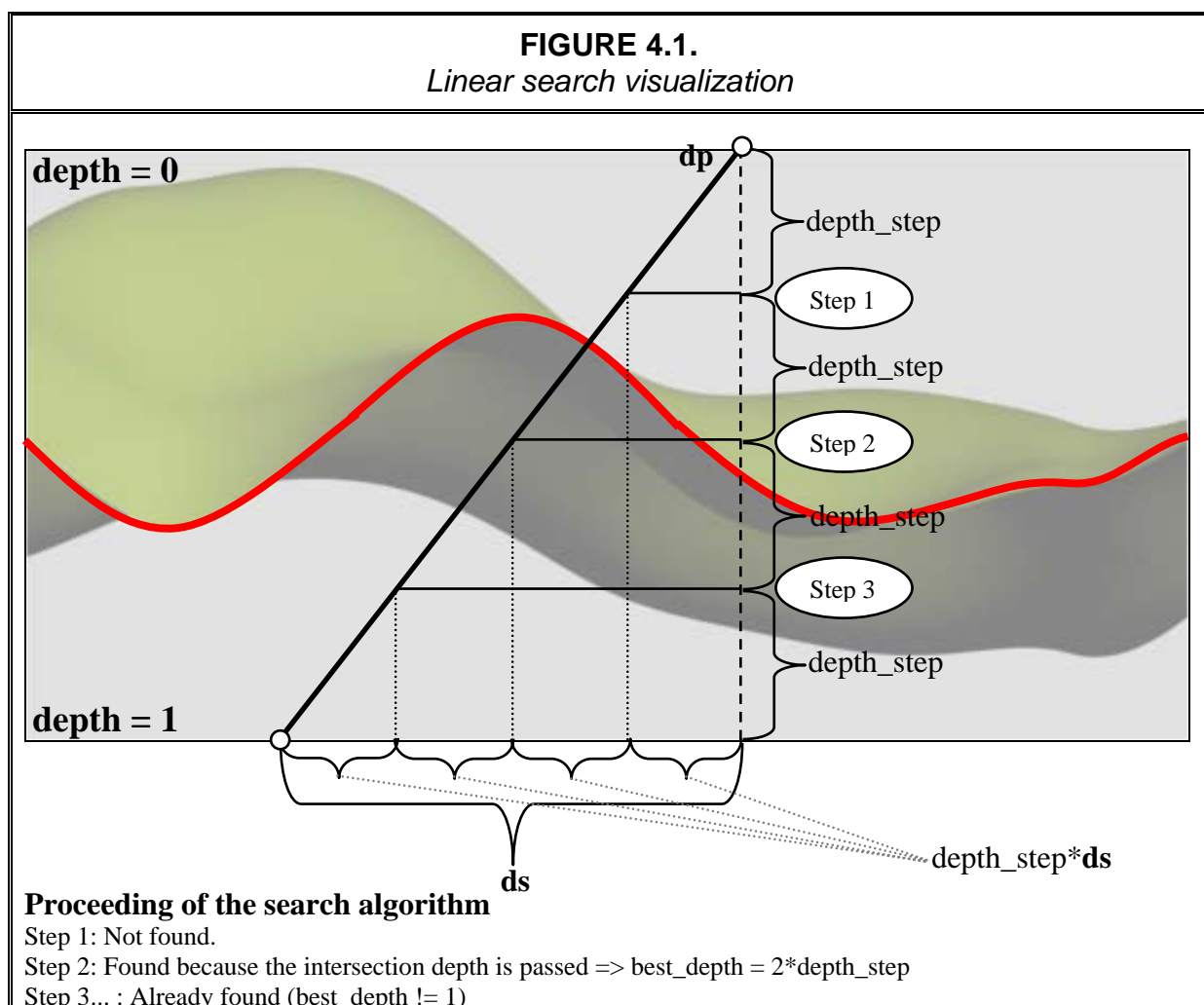




Figure 4.1 explains this further by visualizing the search procedure and explaining the parameters. Here, the number of search steps is taken to be 4.

There is a problem with this linear search technique. If the geometry details are smaller than the linear search step, the algorithm would not detect the intersection that is closest to the viewer. This limits the complexity of the surface that the depth map can define. A complex surface simply needs more search steps.

The linear search alone doesn't produce a very accurate result. For this a binary search is performed on the interval $(\text{best_depth} - \text{depth_step}, \text{best_depth}]$, where there ought to be an intersection.

A general binary search on the interval $(\text{current_depth} - \text{size}, \text{current_depth}]$ proceeds as follows:

1. In parameters are **dp** and **ds** expressed in texture coordinates
2. Let $\text{size} = \text{size} * 0.5$
3. Fetch the depthmap value for $\text{dp} + \text{ds} * \text{current_depth}$
4. If result is smaller than or equal to current_depth , Then
(this means that the point is inside the object)
 $\text{best_depth} = \text{current_depth}$ and $\text{current_depth} = \text{current_depth} - \text{size}$
Else
(this means that the point is outside the object)
 $\text{current_depth} = \text{current_depth} + \text{size}$
5. If the total number of search steps are not yet performed, go to step 2
6. Return best_depth which is an accurate approximation of an intersection point within the start interval

As for linear search the binary search procedure is illustrated in a figure (figure 4.2). In the figure, the total number of binary search steps is 6. The yellow point is the one returned as the best depth found. Notice that the search increases the depth by the current size if the current point is outside the surface, and decreases it if it is inside.

As understood by the binary search algorithm it doesn't necessarily return the desired depth if there is more than one intersection point in the search interval. This is why it cannot be used as a stand-alone search algorithm, but has to be used in combination with the linear search.

So, to find the first intersection point seen from the viewer, we need to do a linear search followed by a binary search. We call this process `find_intersection` and use it to define the RTM algorithm. With the help of the explanation of figure 4.3 it can be said to proceed as follows:

1. Calculate **v** and **s** = (the vector that corresponds to the view ray inside the depth range $[0,1]$) in pixel's local space
2. Let **dp** = (texture coordinates) and **ds** = (**s** projected into texture space)
3. Let $d = \text{find_intersecion}(\text{dp}, \text{ds})$
4. Calculate the light vector, **l**, with respect to the intersection point (**pixelposition** + $d * \text{s}$)
5. Offset texture coordinates with $d * \text{ds}$, and extract the normal, **n**, and the color data from the corresponding textures in the same way as for bump mapping
6. Perform lighting calculations



FIGURE 4.2.
Binary search visualization

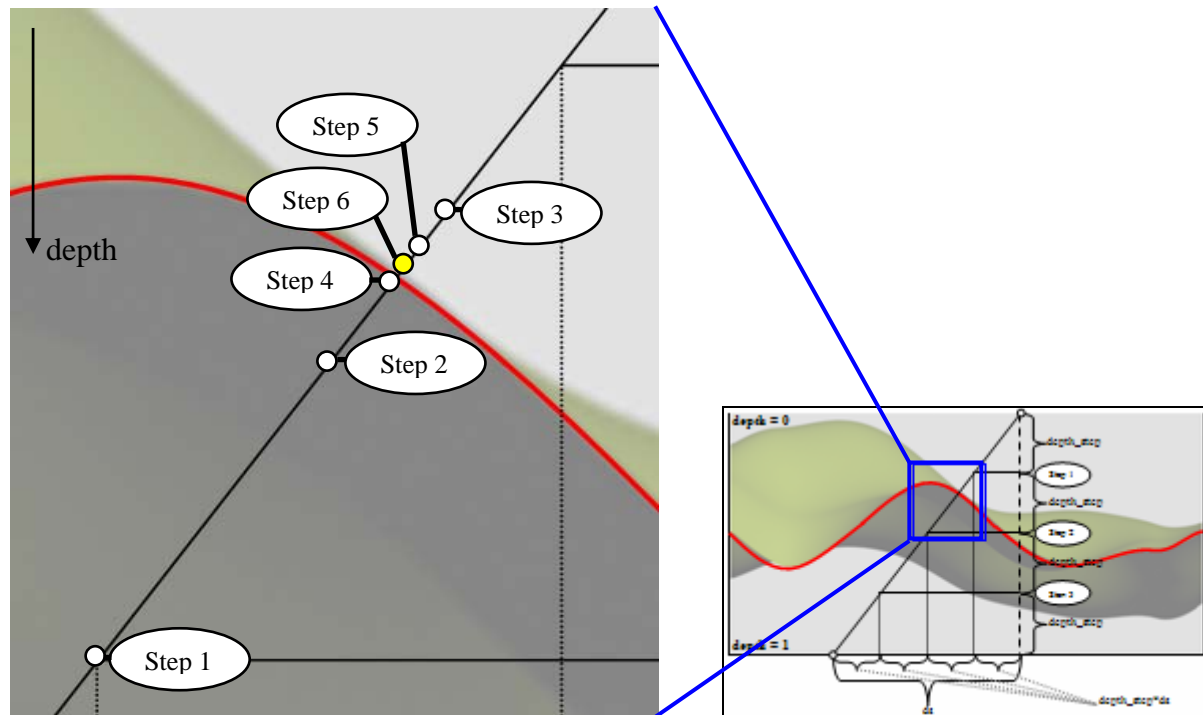
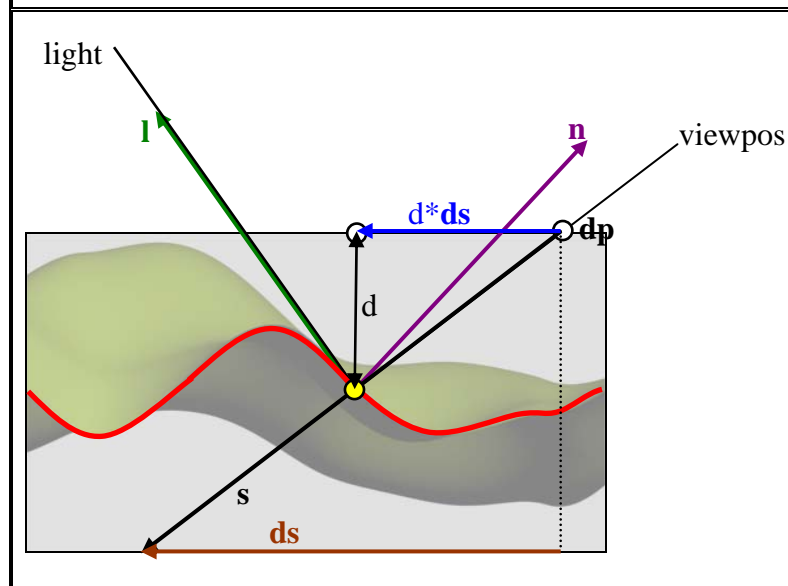


FIGURE 4.3.
Displacements based on find_intersection



A note on the hardware requirements: The main reason why the RTM algorithm will not run on a graphics card with shader model 2.0 is that this model has a maximum number of 96 (up to 512 for shader model 2.x) instructions allowed in the pixel shader. Also, all loops will



be “hard-coded” by the compiler, so that if you have a loop running 10 times, the compiler will actually output 10 times the same code. Because the `find_intersection` function contains two rather complex loops that need to be run at least about 5 times each, the compiled program will have a very large number of instructions and can thus not be run on a graphics card with shader model 2.0. Shader model 3.0, however, has almost no limit to the number of instructions allowed, and also supports “genuine” loops whose code are put in the compiled program only once.

4.2. For arbitrary shaped objects

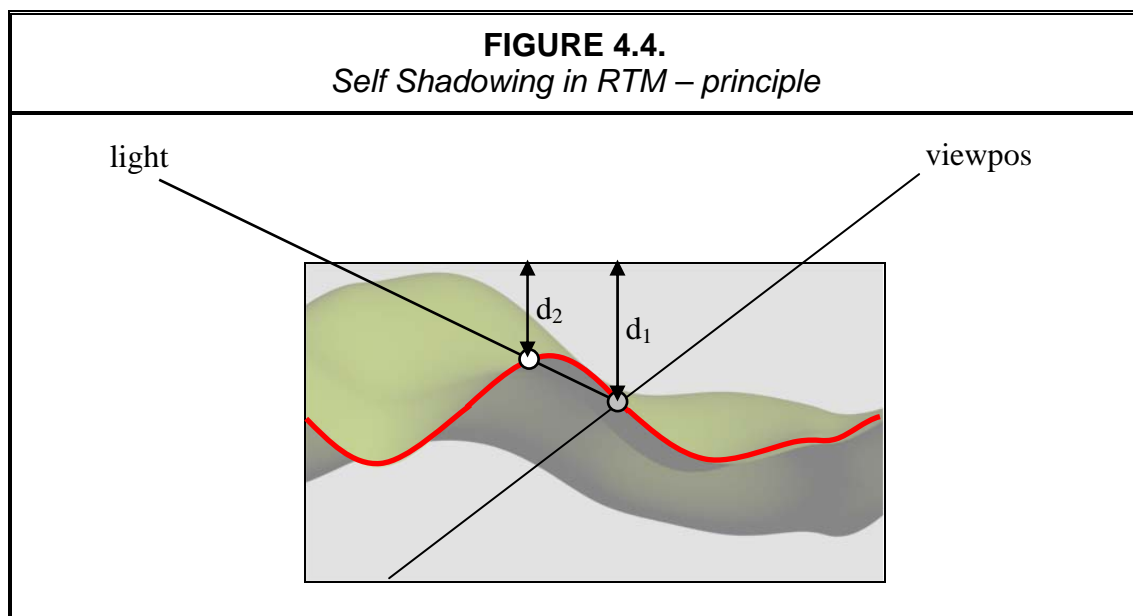
Generalizing the RTM algorithm to be able to handle arbitrary shaped geometrical objects involves some complications. First, as in the case of bump mapping, the calculations need tangent space information to be passed to the shader.

The main algorithm structure is the same as in the case of planar objects (but with planar objects it was easy to define the tangent space data from the plane rendered). However, even when using the local tangent space for the intersections, this approach will not produce accurate results in the case when the object is too deformed because it assumes it to be locally planar in tangent space.

This leads us to the other complication, which is dealt with in section 4.4. In that section we will be approximately transforming the rays into the tangent space using an estimated precalculated curvature of the surface. This enables us to deal with silhouette edges.

4.3. Self-shadows

Self shadows in the RTM algorithm consist of the shadows cast by the surface geometry onto other parts of the surface. A simple addition to the previous algorithm can be implemented to take self-shadows into consideration. Have a look at figure 4.4.



We could simply use `find_intersection` with the light ray in similar way as with the view ray. We compare this depth, d_2 , with the depth obtained when using `find_intersection` with the view ray, d_1 . If the light intersection depth (d_2) is less than the view intersection depth (d_1) then the pixel that is currently to be rendered is in shadow and



its material (color texture map) color is therefore modulated by some shadow intensity value before applying lighting calculations.

4.4. Silhouette edges and depth correction

The previously presented algorithm suffers from two problems. First, in the case of arbitrary shaped objects, the algorithm will not produce correct results at the edges of the geometry. Secondly, while having multiple objects in the scene, there will be errors because the z-buffer will not be correctly updated with the true depth given from the depth map. These two issues will be treated here, beginning with the easiest of them: depth correction.

Depth correction

When using a left-handed coordinate system in DirectX, the matrix that transforms the coordinates from view space (camera space) to projected screen space, e.g. the projection matrix, has the following form:

$$\mathbf{MProj} = \begin{pmatrix} X_{scale} & 0 & 0 & 0 \\ 0 & Y_{scale} & 0 & 0 \\ 0 & 0 & \left(\frac{far}{far - near} \right) & 1 \\ 0 & 0 & \left(\frac{near \cdot far}{near - far} \right) & 0 \end{pmatrix}$$

When transforming a point $p_{view} = (x, y, z, w)$ from view space into $p_{proj} = (x', y', z', w')$ in projection space we perform the operation

$$p_{proj} = p_{view} \cdot \mathbf{MProj},$$

from which we get the z-buffer value as

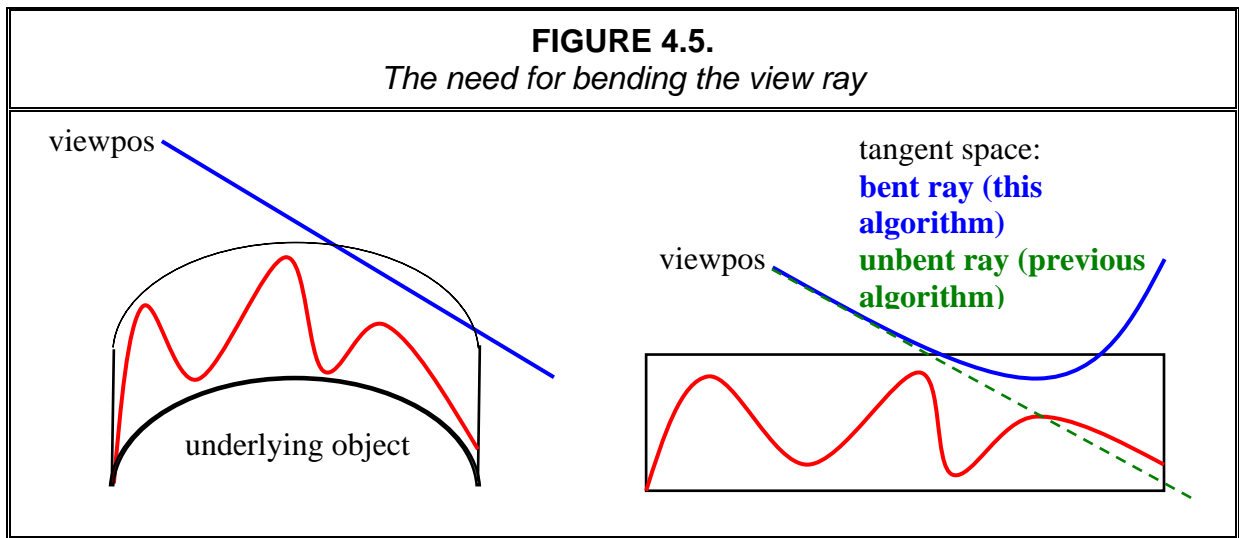
$$z_{buf} = z' / w' = \frac{\left(\frac{far}{far - near} \right) z + \left(\frac{near \cdot far}{near - far} \right)}{z}.$$

So, we transform the view space z position of the point resulted from the find_intersection algorithm (that is the point to be rendered) to get our corrected z-buffer value.

Silhouette edge correction

As previously mentioned, the goal of the silhouette edge correction is to transform the view and light rays into tangent space using a curve approximation. The reason for this could easily be understood when taking a look at figure 4.5.

As seen in the second figure (tangent space) the ray should be bent to give a correct result (no intersection). The amount of this bending depends on the object geometry's curvature. This will force us to change the find_intersection algorithm to take the curvature into account. We will also need to determine if the ray doesn't intersect at all, which basically means that the pixel should not be rendered. This happens when the ray re-exits the depth texture space without having made any intersections.



The curvature of the surface is defined as a quadratic curve estimate, calculated offline and stored per vertex (See Appendix B), and of the form:

$$Ax^2 + By^2 + z = 0$$

Changes to find_intersection (producing find_curved_intersection):

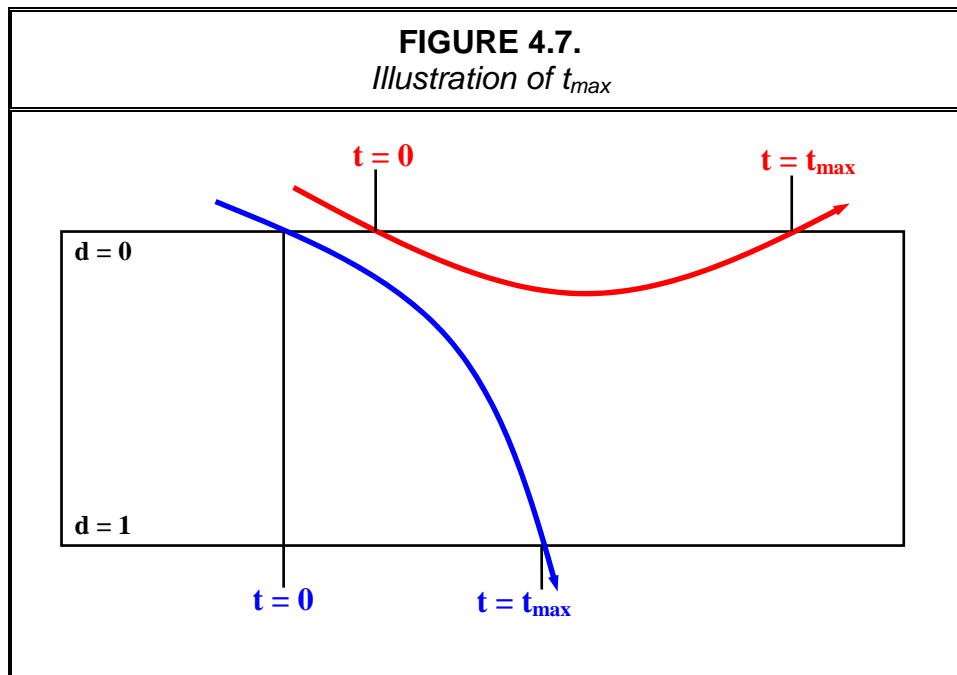
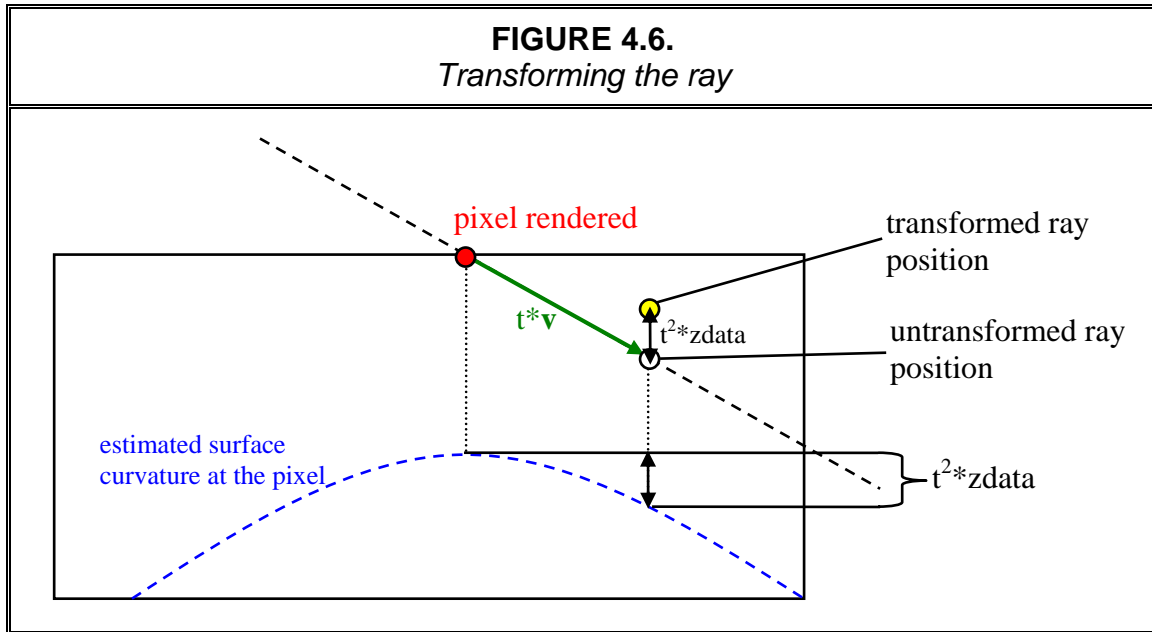
We introduce a function `transform_ray` which applies the curvature to a ray defined in texture space¹. This function operates on the ray with a parameter t which defines the position on the ray (see figure 4.6). The function takes as inputs the parameter t , the ray's direction in texture space \mathbf{v} , the original texture coordinate \mathbf{tx} , and the z -coordinate of the curvature at $t=1$ (see figure 4.6) in texture space (multiplied by $1/\text{depth}$) $zdata$. From this the x, y and z components of the transformed ray are calculated as follows:

$$\begin{aligned} x &= t \cdot \mathbf{v}_x + (\mathbf{tx})_x \\ y &= t \cdot \mathbf{v}_y + (\mathbf{tx})_y \\ z &= t \cdot \mathbf{v}_z - t^2 \cdot zdata, \quad \text{where } zdata \text{ is pre-calculated as } (A \cdot \mathbf{V}_x^2 + B \cdot \mathbf{V}_y^2) / \text{depth}, \\ &\quad \mathbf{V} \text{ being the vector direction in tangent space} \end{aligned}$$

The `transform_ray` function is then called from `find_curved_intersection` (which is based on the parameter t and will be returning it instead of the best depth, which is useful to find the intersected position in different spaces) to find the ray position for a certain t .

Now though, we have another problem. Which will be the interval for the parameter t when performing the linear search? The solution is that it has to be taken as the interval between 0 (the entrance in the texture space volume) and the coordinate where the bent ray first exits texture space, either at the depth $d=0$ or at $d=1$ (See figure 4.7).

¹ Texture space is obtained by scaling the tangent space accordingly. $XScale = \text{tile}/\text{length}(u\text{-axis})$, $YScale = \text{tile}/\text{length}(v\text{-axis})$, $ZScale = 1/\text{depth}$.



The intersection points can be found mathematically by setting d as 1 respectively 0, in

$$d = t \cdot \mathbf{v}_z - t^2 \cdot (A \cdot \mathbf{V}_x^2 + B \cdot \mathbf{V}_y^2) / \text{depth} \equiv t \cdot \mathbf{v}_z - t^2 \cdot zdata$$

which gives the first encountered non-zero roots in texture space as

$$d = 0 : t_{\max} = \frac{\mathbf{v}_z}{zdata} \quad d = 1 : t_{\max} = \frac{\mathbf{v}_z - \sqrt{\mathbf{v}_z^2 - 4 \cdot zdata}}{2 \cdot zdata}.$$

From these two solutions the one producing the smallest t_{\max} is used.



If the intersection parameter t , resulting from `find_curved_intersection`, is greater than t_{\max} , then the ray doesn't intersect at all, which means the pixel should be discarded!

Remark: Shadows

When using corrected silhouette edges a shadowing algorithm such as shadow volumes will no longer give a sufficient result. However, with depth correction, a shadow mapping algorithm will. This forces us to apply shadow maps if we wish to have shadows cast from our relief mapped objects onto other objects in the scene. A shadow mapping algorithm can also easily be applied to shadows cast onto the relief objects. This automatically generates self shadows as well.

4.5. Efficiency

After having introduced all the different techniques we are ready to do an efficiency comparison as a complement to the visual comparison presented in the introductory section. This was done with the very same demo program by running the techniques one after the other and measuring the time it took to render 10 frames for each of them. The results on the test computer¹ are shown in table 4.1.

Because using a reference driver slows down the frame rates considerably, the frame rates are not measured in FPS, but relatively to each other. For example when comparing bump mapping frame rates with those of relief mapping, we might say that bump mapping is X times faster than relief mapping.

TABLE 4.1. <i>Results from efficiency test</i>		
Technique	Time to render 10 frames with reference driver (ms)	Time divided by Time for CTM
Conventional texture mapping (CTM)	12688	1,0
Bump mapping	19047	1,5
Parallax mapping	20141	1,6
Simple RTM	79031	6,2
RTM with self-shadows	141313	11,1
RTM with silhouette edges	255969	20,2

A way to find an approximate order of magnitude of the frame rate on a computer supporting a hardware implementation is to test the techniques that are hardware-implementable on the current test computer and measure the time used to render 1000 frames. The times for reference driver and hardware driver could then be compared to find an estimate of how fast the reference driver is compared to the hardware driver. This is of course a very approximate method and needs a lot of different tests to give accurate results. But using the first three techniques, I found an estimation that the hardware driver is about 90 times faster than the reference driver. This shows that the RTM algorithms could almost be considered as real-time for the test computer (if they could have been run in hardware mode in which case they would have the estimated FPS values: 11,4 for Simple RTM, 6,4 for RTM with self-shadows and 3,5

¹ A laptop with: Graphics card: ATI RADEON XPRESS 200M, CPU: AMD Athlon 64 3200



for RTM with silhouette edges) – and are in fact considered as real-time techniques for newer graphics cards for which the hardware/reference factor would have been even larger due to an increase in speed of the graphics processor.

The fact that RTM is a per-pixel technique makes it run slower with objects that cover a larger area of the screen, as well as for increased screen resolution.



5. Application: Object rendering

Besides the fact that RTM techniques can be used to enhance surface detail on geometric meshes, they can also be used to define simple objects. This can be very memory-saving when using detailed objects, and it also discards the need for a level-of-detail system (possibly needed for polygon objects) because it is always pixel-based and thus renders fewer pixels at a greater distance from the viewer. The principles of object representation and rendering are described here. We build on the planar RTM algorithm introduced in section 4.1.

5.1. One- and Two-sided objects

To use RTM to render one- or two-sided objects only one texture is needed with the alpha respectively alpha and blue components representing the depths. One could tell the renderer to discard a pixel if the best found depth is 1.0, which corresponds to the case when no intersection is found. The one-sided objects are then simple (exactly the same as the planar RTM). The two-sided needs, on the other hand, to consider the back depth as well. This is done in `find_intersection` by inserting another check when testing if the depth is inside the object or not. Thus:

For one-sided: $depth \geq front_depth$

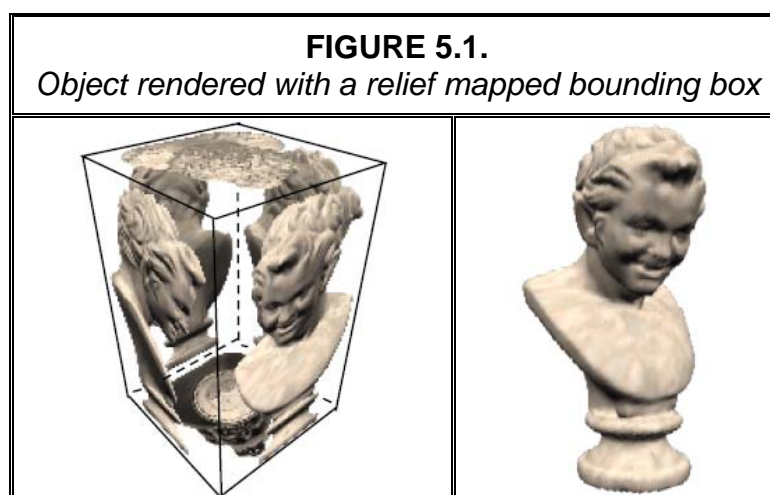
For two-sided: $back_depth \geq depth \geq front_depth$.

This, two-sided approach, however, doesn't give correct results when rendered from the back, because the normals wouldn't be right. Two different relief texture objects would be needed to be rendered if this was desired.

5.2. Objects defined on a bounding box

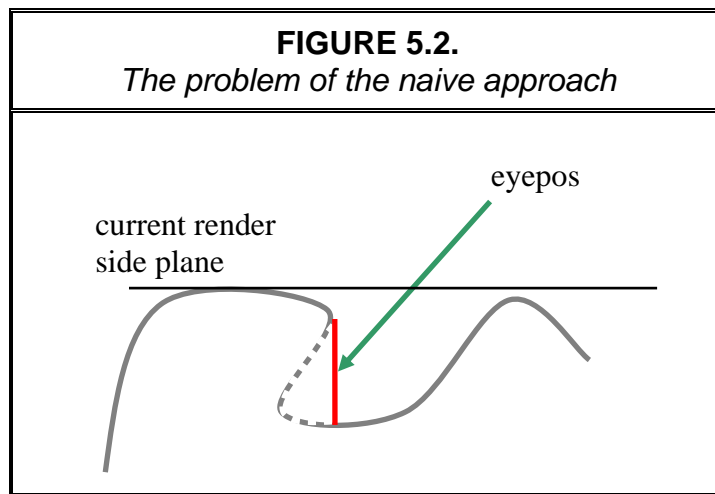
To represent more complex objects one could use an approach with a total number of 6 relief object planes representing the sides on the object's bounding box. Each of the 6 relief textures are generated from the object with a view direction perpendicular to the current side (see Appendix B). However, naturally, when looking at a box one can always only see a maximum of 3 sides, and these are the only three sides that need to be rendered, although one has to take two depth values into account (front and back depth) for each side. Basically, the test if the relief object side is to be rendered or not is easily performed by making use of back face culling. The front-facing sides are rendered.

A commonly rendered object is the statue head. See figure 5.1 for an image taken from [RTMOrig00].





A naive approach is to just render the relief texture mapped sides independently. However, if this is done the situation shown in figure 5.2 could occur.



Here we have an object defined with the silhouette in grey. When rendering the current render side with its reliefmap, the intersection of the depth map data and the view ray would occur at a point on the red “discontinuity” line. This is natural, because the current depth map has to be defined that way. Because of this the set of objects that can be represented and rendered with this naive approach is very limited.

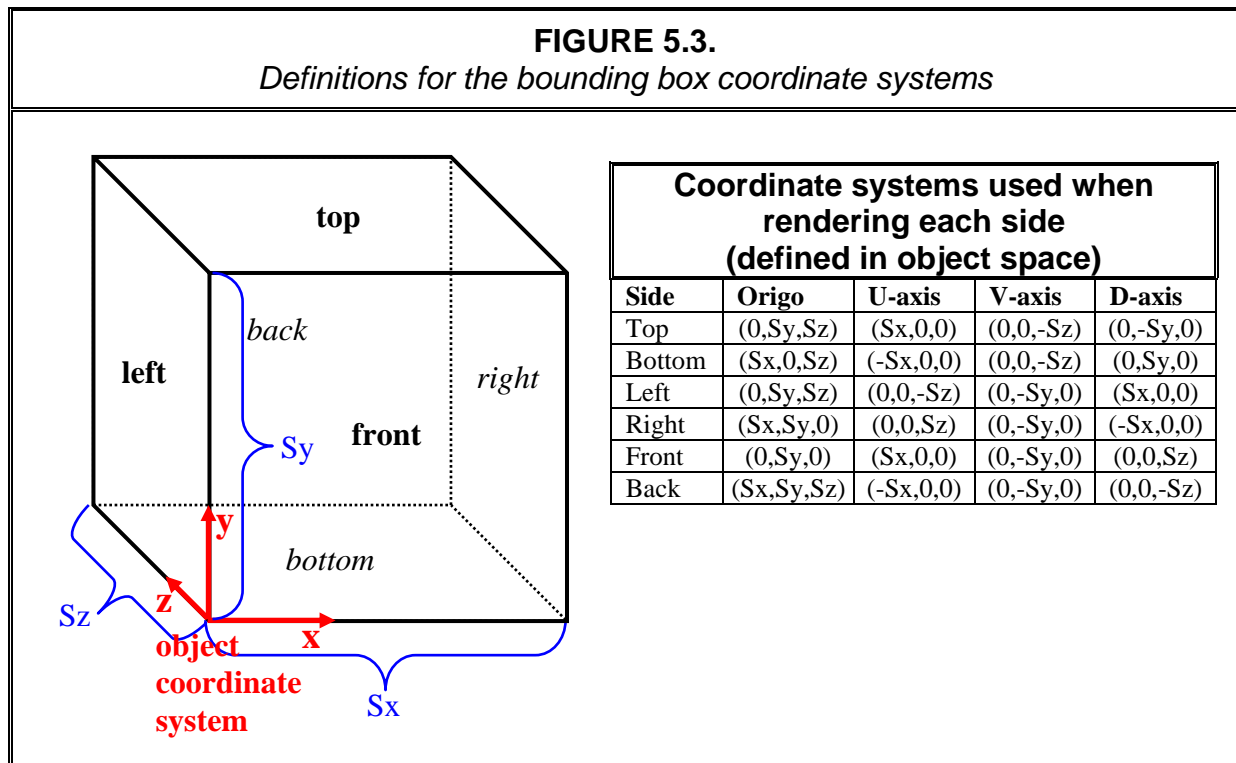
Luckily, there is a cure for this. Consider checking the depth data of the sides that are perpendicular to the current one. If the found intersection point is outside some of the other depth data, then it is to be discarded. In this case, the side having its normal pointing out of this page would find that the point found on the red line is outside its depth-range. Therefore, the pixel could and should be discarded. To fill in the discarded pixel we need to render one pass for each of the visible relief maps that are perpendicular to the current side. But how do we implement such a system? Consider defining the box as in figure 5.3.

Say that we are rendering the front relief map. We will need to pass not only the textures associated with this side, but also the textures that contain double depth data for the top and left sides (these contain in their double depth information about the right and bottom depths). When a point is found, we transform it into the coordinate system of the left and top sides each and check it against their depth data. The transformation matrices used are easily precalculated and passed to the shader. In the same way, we can render the other relief maps with the relief maps to “the left of” and “over” them. How these sides are assigned depends on the coordinate systems chosen for each side and become in this case the ones shown in table 5.1.

TABLE 5.1. <i>Associated sides</i>		
Current Side	Left of	Over
Top	Left	Back
Bottom	Right	Back
Left	Back	Top
Right	Front	Top
Front	Left	Top
Back	Right	Top



Using this technique solves the problem of figure 5.2 if the object contains non-depth data on at maximum all but one side at a time, otherwise it will not work. If the object to be rendered is too complex, there is still another possibility: using Multilayer RTM. This technique, though, is out of the scope of this project and the reader is referred to [RTMMulti06].



5.3. Applying object relief rendering to a chess knight model

Using a handwritten tool program, the textures in figure 5.4 were generated from a mesh model of a chess knight. The tool uses the depth- and normal-map generation algorithms described in Appendix B.

The defined chess knight was then rendered with a bounding-box technique. The result is shown from different views in figure 5.5. The first figure was rendered using the naive approach which doesn't support convex objects. The other three were rendered with the "cure" described in section 5.2.

The demo program used is [ReliefObject](#). The shader code is found in Appendix A.



FIGURE 5.4. <i>Maps for the chess knight</i>			
Side	Depth Texture	Normal Texture	Relief Texture
Top			
Bottom			
Left			
Right			
Front			
Back			

FIGURE 5.5. <i>Renders of the chess knight (first image was rendered with naive approach)</i>			



6. Conclusion

In this project we have studied and implemented different techniques for representing surface detail – ranging from conventional texture mapping to relief texture mapping with correct silhouette edges. These techniques are becoming usable in real-time as graphics hardware gets better (which happens fast). In some years we will be seeing these techniques in computer games. In fact, on his home-page¹, Manuel Oliveira mentions having implemented relief mapping for the Doom3 engine.

The possibilities with RTM are many, but one always needs to trade between performance. When viewed from a close distance a relief texture mapped object is very expensive to render because it covers a bigger part of the screen than an object farther away. Some kind of LOD scheme could be used for this. Consider rendering a mesh detailing the relief surface at a close distance and a simpler relief mapped mesh at a far distance.

When using relief mapping to represent objects, a method has here been developed to handle rather complex object shapes with the only requirement that it is completely defined by ALL relief maps together. The developed method could surely be optimized in some ways but this was out of the time scope of the project and is left for further investigation.

¹ <http://www.inf.ufrgs.br/~oliveira/RTM.html>



Appendix A: Shader code

A.1. CompareTechniques Shaders

Table A.1. shows the source codes for the HLSL shader used in CompareTechniques. I chose to have one common vertex shader for all techniques, but each technique has its own pixel shader.

TABLE A.1. <i>Shaders for CompareTechniques</i>	
Structures	
<pre> struct a2v { float3 position : POSITION; float2 texcoord : TEXCOORD0; float3 normal : NORMAL; float4 tangent : TEXCOORD1; float4 binormal : TEXCOORD2; float2 curvature : TEXCOORD3; }; struct v2ps { float4 projpos : POSITION; float3 position : TEXCOORD0; float2 texcoord : TEXCOORD1; float3 normal : TEXCOORD2; float3 tangent : TEXCOORD3; float3 binormal : TEXCOORD4; float2 curvature : TEXCOORD5; float3 scale : TEXCOORD6; }; struct v2p { float3 position : TEXCOORD0; float2 texcoord : TEXCOORD1; float3 normal : TEXCOORD2; float3 tangent : TEXCOORD3; float3 binormal : TEXCOORD4; float2 curvature : TEXCOORD5; float3 scale : TEXCOORD6; }; struct p2s { float4 color : COLOR; }; struct p2sd { float4 color : COLOR; //float depth : DEPTH; }; </pre>	
Global Variables	
<pre> //Vertex Shader float4x4 worldviewproj; float tile; //Pixel Shader float3 light_pos; float3 view_pos; float3 light_ambient; float3 light_diffuse; </pre>	



```
float3 light_specular;
float shininess;
float depth;
float2 planes;

//Textures
sampler2D colormap : TEXUNIT0;
sampler2D normalmap : TEXUNIT1;
sampler2D reliefmap : TEXUNIT2;
```

Vertex Shader

```
v2ps mainvs(in a2v IN)
{
    v2ps OUT;

    OUT.projpos = mul(float4(IN.position,1),worldviewproj);
    OUT.position = IN.position;
    OUT.texcoord = IN.texcoord*tile;
    OUT.normal = IN.normal;
    OUT.tangent = IN.tangent.xyz;
    OUT.binormal = IN.binormal.xyz;
    OUT.curvature = IN.curvature;
    OUT.scale = float3(IN.tangent.w,IN.binormal.w,1.0)/tile;

    return OUT;
}
```

Conventional Texture Mapping

```
p2s convtexps(in v2p IN)
{
    p2s OUT;

    //Lighting vectors
    float3 v = normalize(view_pos - IN.position);
    float3 l = normalize(light_pos - IN.position);
    float3 h = normalize(l+v);

    //Normal
    float3 n = normalize(IN.normal);

    //Fetch Color
    float3 color = tex2D(colormap,IN.texcoord).rgb;

    //Calculate lighting
    float diffuse_intensity = max(dot(l,n),0);
    float specular_intensity = max(dot(h,n),0);

    //Output color
    float3 ambient_color = color*light_ambient;
    float3 diffuse_color = color*light_diffuse*diffuse_intensity;
    float3 specular_color = light_specular*pow(specular_intensity,shininess);
    OUT.color.rgb = ambient_color+diffuse_color+specular_color;
    OUT.color.a = 1;

    return OUT;
}
```

Bump Mapping

```
p2s bumpmapps(in v2p IN)
{
    p2s OUT;

    //Lighting vectors
    float3 v = normalize(view_pos - IN.position);
    float3 l = normalize(light_pos - IN.position);
    float3 h = normalize(l+v);

    //Fetch normal
    float3 n = tex2D(normalmap,IN.texcoord).xyz;
    n = n*2.0-1.0;
    n = normalize(n.x*IN.tangent + n.y*IN.binormal + n.z*IN.normal);
```



```
//Fetch color
float3 color = tex2D(colormap,IN.texcoord).rgb;

//Calculate lighting
float diffuse_intensity = max(dot(l,n),0);
float specular_intensity = max(dot(h,n),0);
float att = 1.0-max(0,dot(IN.normal,l)); att = 1.0-att*att;

//Output color
float3 ambient_color = color*light_ambient;
float3 diffuse_color = color*light_diffuse*diffuse_intensity;
float3 specular_color = light_specular*pow(specular_intensity,shininess);
OUT.color.rgb = ambient_color+att*(diffuse_color+specular_color);
OUT.color.a = 1;

return OUT;
}
```

Parallax Mapping

```
p2s paramapps(in v2p IN)
{
    p2s OUT;

    //Lighting vectors
    float3 v = normalize(view_pos - IN.position);
    float3 l = normalize(light_pos - IN.position);
    float3 h = normalize(l+v);

    //View vector in tangent space
    float3x3 tbn = float3x3(IN.tangent,-IN.binormal,IN.normal);
    float3 v_t = mul(tbn,-v);

    //Parallax code
    float2 uv = IN.texcoord;
    float d = tex2D(reliefmap,uv).w*0.10 - 0.05;
    uv.xy += d * float2(v_t.x,v_t.y);

    //Fetch normal
    float3 n = tex2D(normalmap,uv).xyz;
    n = n-0.5;
    n = normalize(n.x*IN.tangent+n.y*IN.binormal+n.z*IN.normal);

    //Fetch color
    float3 color = tex2D(colormap,uv).rgb;

    //Calculate lighting
    float diffuse_intensity = max(dot(l,n),0);
    float specular_intensity = max(dot(h,n),0);
    float att = 1.0-max(0,dot(IN.normal,l)); att = 1.0-att*att;

    //Output color
    float3 ambient_color = color*light_ambient;
    float3 diffuse_color = color*light_diffuse*diffuse_intensity;
    float3 specular_color = light_specular*pow(specular_intensity,shininess);
    OUT.color.rgb = ambient_color+att*(diffuse_color+specular_color);
    OUT.color.a = 1;

    return OUT;
}
```

find_intersection

```
float find_intersection(float2 dp,float2 ds)
{
    //Linear search
    const int linear_search_steps = 10;
    float size = 1.0/linear_search_steps;
    float current_depth = 0.0;
    for(int i=0;i<linear_search_steps-1;i++)
    {
        float d = tex2D(reliefmap,dp+ds*current_depth).w;
```



```

        if(current_depth<d)
            current_depth += size;
    }

    //Binary search
    const int binary_search_steps = 6;
    for(int i=0;i<binary_search_steps;i++)
    {
        size *= 0.5;

        float d = tex2D(reliefmap,dp+ds*current_depth).w;

        if(current_depth<d)
            current_depth += 2*size;
        current_depth -= size;
    }

    return current_depth;
}

```

Simple RTM

```

p2s rtmsimpleps(in v2p IN)
{
    p2s OUT;

    //View vector
    float3 p = IN.position;
    float3 v = normalize(p - view_pos);

    //RTM code
    float a = dot(IN.normal,-v);
    float3 s = float3(dot(v,IN.tangent),dot(v,-IN.binormal),a);
    s *= depth/a;
    float2 ds = s.xy;
    float2 dp = IN.texcoord;
    float d = find_intersection(dp,ds);
    float2 uv = dp+ds*d;
    p += v*d/(a*depth);

    //Fetch normal
    float3 n = tex2D(normalmap,uv).xyz;
    n = n-0.5;
    n = normalize(n.x*IN.tangent + n.y*IN.binormal + n.z*IN.normal);

    //Fetch color
    float3 color = tex2D(colormap,uv).rgb;

    //Light and Half vectors
    float3 l = normalize(p - light_pos);
    float3 h = normalize(-l-v);

    //Calculate lighting
    float diffuse_intensity = max(dot(-l,n),0);
    float specular_intensity = max(dot(h,n),0);
    float att = 1.0-max(0,dot(IN.normal,-l)); att = 1.0-att*att;

    //Output color
    float3 ambient_color = color*light_ambient;
    float3 diffuse_color = color*light_diffuse*diffuse_intensity;
    float3 specular_color = light_specular*pow(specular_intensity,shininess);
    OUT.color.rgb = ambient_color+att*(diffuse_color+specular_color);
    OUT.color.a = 1;

    return OUT;
}

```

RTM with SelfShadows

```

p2s rtmselfshadowps(in v2p IN)
{
    p2s OUT;

```



```
//Shadow constants
const float shadow_threshold = 0.08;
const float shadow_intensity = 0.4;

//View vector
float3 p = IN.position;
float3 v = normalize(p - view_pos);

//RTM code
float a = dot(IN.normal,-v);
float3 s = float3(dot(v,IN.tangent),dot(v,-IN.binormal),a);
s *= depth/a;
float2 ds = s.xy;
float2 dp = IN.texcoord;
float d = find_intersection(dp,ds);
float2 uv = dp+ds*d;
p += v*d/(a*depth);

//Fetch normal
float3 n = tex2D(normalmap,uv).xyz;
n = n-0.5;
n = normalize(n.x*IN.tangent + n.y*IN.binormal + n.z*IN.normal);

//Fetch color
float3 color = tex2D(colormap,uv).rgb;

//Light and Half vectors
float3 l = normalize(p - light_pos);
float3 h = normalize(-l-v);

//Calculate lighting
float diffuse_intensity = max(dot(-l,n),0);
float specular_intensity = max(dot(h,n),0);
float att = 1.0-max(0,dot(IN.normal,-l)); att = 1.0-att*att;

//SelfShadow RTM code
a = dot(IN.normal,-l);
s = float3(dot(l,IN.tangent),dot(l,-IN.binormal),a);
s *= depth/a;
ds = s.xy;
dp = uv - d*ds;
float dl = find_intersection(dp,ds);
if(dl < d-shadow_threshold)
{
    color *= shadow_intensity;
    specular_intensity = 0;
}

//Output color
float3 ambient_color = color*light_ambient;
float3 diffuse_color = color*light_diffuse*diffuse_intensity;
float3 specular_color = light_specular*pow(specular_intensity,shininess);
OUT.color.rgb = ambient_color+att*(diffuse_color+specular_color);
OUT.color.a = 1;

return OUT;
}
```

transform_ray

```
float3 transform_ray(float t,float2 tx,float3 v,float dataz)
{
    float3 r = v*t;
    r.z -= t*t*dataz;
    r.xy += tx;
    return r;
}
```

find_curved_intersection

```
float find_curved_intersection(float2 tx,float3 v,float tmax,float dataz)
{

```



```
//Linear search
const int linear_search_steps = 30;
float size = (tmax+0.001)/linear_search_steps;
float current_t = 0.0;
for(int i=0;i<linear_search_steps;i++)
{
    float3 p = transform_ray(current_t,tx,v,dataz);

    float t = tex2D(reliefmap,p.xy).w;

    if(p.z<t)
        current_t += size;
}

//Binary search
const int binary_search_steps = 6;
for(int i=0;i<binary_search_steps;i++)
{
    size *= 0.5;

    float3 p = transform_ray(current_t,tx,v,dataz);

    float t = tex2D(reliefmap,p.xy).w;

    if(p.z< t)
        current_t += 2*size;
    current_t -= size;
}

return current_t;
}
```

Complete RTM

```
p2sd rtmcompleteps(in v2p IN)
{
    p2sd OUT;

    //View vector
    float3 view = normalize(IN.position-view_pos);

    //View vector in tangent space
    float3 v = normalize(float3(dot(view,IN.tangent),dot(view,-IN.binormal),dot(IN.normal,-view)));

    //Scale for transforming from tangent to texture space
    float3 mapping = float3(1,1,1/depth)/IN.scale;

    //Curvature value
    float dataz = IN.curvature.x*v.x*v.x + IN.curvature.y*v.y*v.y;
    dataz = sign(dataz)*max(abs(dataz),0.001); //Protect against being 0 for future divisions

    //Find tmax
    float d = v.z*v.z-4*dataz*depth;
    float tmax = 50;
    if(d>0)
        tmax = min(tmax,(-v.z+sqrt(d))/(-2*dataz));
    d = v.z/dataz;
    if(d>0)
        tmax = min(tmax,d);

    //Transform v and dataz to texture space
    v *= mapping;
    dataz *= mapping.z;

    //Find parameter for intersection
    float t = find_curved_intersection(IN.texcoord,v,tmax,dataz);

    //Discard if no intersection found
    if(t > tmax)
        discard;

    //Get intersection position in texture space
    float3 p = transform_ray(t,IN.texcoord,v,dataz);
}
```



```
//Fetch normal
float3 n = tex2D(normalmap,p.xy).xyz;
n = n-0.5;
n = normalize(n.x*IN.tangent + n.y*IN.binormal + n.z*IN.normal);

//Fetch color
float3 color = tex2D(colormap,p.xy).rgb;

//Depth correction
//float p_z = length(IN.position - view_pos) + t;
//OUT.depth = ((planes.x*p_z+planes.y)/-p_z);

//Intersection position in world space
p = IN.position + view*t;

//Light and Half vectors
float3 l = normalize(p - light_pos);
float3 h = normalize(-l-view);

//Calculate lighting
float diffuse_intensity = max(dot(-l,n),0);
float specular_intensity = max(dot(h,n),0);
float att = 1.0-max(0,dot(IN.normal,-l)); att = 1.0-att*att;

//Output color
float3 ambient_color = color*light_ambient;
float3 diffuse_color = color*light_diffuse*diffuse_intensity;
float3 specular_color = light_specular*pow(specular_intensity,shininess);
OUT.color.rgb = ambient_color+att*(diffuse_color+specular_color);
OUT.color.a = 1;

return OUT;
}
```

A.2. Shaders for ReliefObject

Table A.2. shows the source of the ReliefObject shader.

TABLE A.2. <i>Shaders for ReliefObject</i>	
Structures	
<pre>struct a2v { float3 position : POSITION; }; struct v2ps { float4 projpos : POSITION; float3 position : TEXCOORD0; }; struct v2p { float3 position : TEXCOORD0; }; struct p2s { float4 color : COLOR; };</pre>	
Global Variables	
//Vertex Shader	



```
float4x4 worldviewproj;
```

```
//Pixel Shader
float4 origo;
float4 u;
float4 v;
float4 depth;
float4 view_pos;
float4 light_pos;
float3 light_ambient;
float3 light_diffuse;
float3 light_specular;
float shininess;
float4x4 to_left;
float4x4 to_top;
```

```
//Textures
sampler2D colormap : TEXUNIT0;
sampler2D reliefmap : TEXUNIT1;
sampler2D leftrelief : TEXUNIT2;
sampler2D toprelief : TEXUNIT3;
```

Vertex Shader

```
v2ps main(in a2v IN)
{
    v2ps OUT;

    OUT.projpos = mul(float4(IN.position,1),worldviewproj);
    OUT.position = IN.position;

    return OUT;
}
```

project_uv

```
float3 project_uv(float3 p)
{
    return float3(dot(p,u.xyz)/u.w,dot(p,v.xyz)/v.w,dot(p,depth.xyz)/depth.w);
}
```

find_intersection

```
float find_intersection(sampler2D rm,float3 dp,float3 ds)
{
    //Linear search
    const int linear_search_steps = 20;
    float depth_step = 1.0/linear_search_steps;
    float size = depth_step;
    float current_depth = 0.0;
    float best_depth = 1.0;
    for(int i=0;i<linear_search_steps-1;i++)
    {
        current_depth += size;

        float2 d = tex2D(rm,dp.xy+ds.xy*current_depth).wz - dp.z;

        if(best_depth > 0.996)
            if(current_depth >= d.x)
                if(current_depth <= d.y)
                    best_depth = current_depth;
    }

    //Binary search
    current_depth = best_depth;
    const int binary_search_steps = 4;
    for(int i=0;i<binary_search_steps;i++)
    {
        size *= 0.5;
        float2 d = tex2D(rm,dp.xy+ds.xy*current_depth).wz - dp.z;

        if(current_depth >= d.x)
```



```
        if(current_depth <= d.y)
        {
            best_depth = current_depth;
            current_depth -= 2*size;
        }
        current_depth += size;
    }

    return best_depth;
}
```

Pixel Shader

```
p2s main(in v2p IN)
{
    p2s OUT;

    //View vector
    float3 v = normalize(IN.position - view_pos);

    //RTM code
    float3 p = IN.position - origo;
    float3 s = depth.w*v/dot(depth.xyz,v);
    float3 dp = project_uv(p);
    float3 ds = project_uv(s);
    float d = find_intersection(reliefmap,dp,ds);
    float2 uv = dp.xy + ds.xy*d;

    //Check if pixel is to be discarded
    if(uv.x < 0.0 || uv.x > 1.0 || uv.y < 0.0 || uv.y > 1.0 || d > 0.996)
        discard;
    float3 p_top = mul(float4(uv,d+dp.z,1),to_top).xyz;
    float2 top_i = tex2D(toprelief,p_top.xy).wz;
    if(p_top.z < top_i.x - 0.008 || p_top.z > top_i.y + 0.008)
        discard;
    float3 p_left = mul(float4(uv,d+dp.z,1),to_left).xyz;
    float2 left_i = tex2D(leftrelief,p_left.xy).wz;
    if(p_left.z < left_i.x - 0.008 || p_left.z > left_i.y + 0.008)
        discard;

    //Fetch normal
    float3 n = tex2D(reliefmap,uv).xyz;
    n = n*2.0 - 1.0;
    n.z = sqrt(1.0 - dot(n.xy,n.xy));
    n = normalize(n.x*u.xyz+n.y*v.xyz+n.z*depth.xyz);

    //Fetch color
    float3 color = tex2D(colormap,uv).xyz;

    //Get intersection position in world space
    p += origo + s*d;

    //Light and Half vectors
    float3 l = normalize(p - light_pos);
    float3 h = normalize(-l-v);

    //Calculate lighting
    float diffuse_intensity = saturate(dot(-l,n));
    float specular_intensity = saturate(dot(h,n));

    //Output color
    float3 ambient_color = light_ambient*color;
    float3 diffuse_color = light_diffuse*color*diffuse_intensity;
    float3 specular_color = light_specular*pow(specular_intensity,shininess);
    OUT.color.rgb = ambient_color + diffuse_color + specular_color;
    OUT.color.a = 1;

    return OUT;
}
```




Appendix B: Mesh and Map generation

B.1. The depth map

The depth map is a texture map with one color component that represents the depth. If the map is to represent the shape of an object it has to be generated from the object. Depending on the objects complexity one may need one or more depth maps to represent it.

When representing objects as in section 5, a rather simple way to generate a depth map is to set up a scene where the reference mesh object is scaled so that its bounding box has the dimensions 2x2x1 when viewed from the desired angle. In this case the vertices in world space can be assumed to be transformed (The viewport's x and y coordinates reach from -1 to 1, while the depth coordinate reaches from 0 to 1. No matrix transformation is needed when rendering. The rendering is done to a texture and saves the depth of the rendered pixels, which corresponds to the final depth in the texture.

B.2. The normal map

A normal map used for relief mapping purposes (and the other techniques treated in this project as well) can be generated from a depth map. Looping through all texels (u,v) in the depth map we get the normal as

$$\begin{aligned} \text{tangent}(u, v) &= (1, 0, \text{depth}(u + du, v) - \text{depth}(u, v)) \\ \text{binormal}(u, v) &= (0, 1, \text{depth}(u, v + dv) - \text{depth}(u, v)) \\ \text{normal}(u, v) &= \text{normalize}(\text{tangent}(u, v) \times \text{binormal}(u, v)) \end{aligned}$$

If this is done in a system where u and v reach from 0 to 1 then we could take du and dv as $1/\text{texture_dimension}$. The resulting normal coordinates reach from -1 to 1, but in a texture we can only store positive values from 0 to 1. Therefore we compress the normal coordinates by dividing them by 2 and adding 0.5 before storing them in the normal texture. When we later are to use the normal, we need to decompress it again by inverting this operation.

Another way to generate a normal map is to render the object mesh in a similar way as for depth map generation, but this time, we store interpolated normals instead of depth data in the output pixel. This enables, as opposed to generation from the depth map a generation of a “smooth” normal map. This alternative technique is the one used in this project.

Remark:

When using normal maps with depth maps (relief maps), they are placed into the same texture with the channels representing the data as follows:

R: normal x (*compressed*) **G:** normal y (*compressed*) **B:** back depth **A:** front depth

The normal's z component can easily be calculated due to the fact that the normal is normalized. It is always positive and becomes thus

$$z = \sqrt{1 - x^2 - y^2}.$$



B.3. Calculating the tangent, binormal and normal of a mesh

In this project we have only used a mathematically generated sphere, but what if we were to load an arbitrary model? We would need some way to calculate the tangent space coordinate system of each vertex. Luckily, DirectX comes with a built-in function to do this: `D3DXComputeTangentFrame`. But for understanding reasons, I here include a brief explanation of how this is done.

The tangent, binormal, and normal of one triangle is calculated as

$$\mathbf{T} = (u_2 - u_0)(\mathbf{P}_2 - \mathbf{P}_0) + (u_1 - u_0)(\mathbf{P}_1 - \mathbf{P}_0)$$

$$\mathbf{B} = (v_2 - v_0)(\mathbf{P}_2 - \mathbf{P}_0) + (v_1 - v_0)(\mathbf{P}_1 - \mathbf{P}_0)$$

$$\mathbf{N} = \mathbf{T} \times \mathbf{B}$$

(\mathbf{P}_0 , \mathbf{P}_1 and \mathbf{P}_2 are the vertex positions of a triangle and u_i, v_i ($i=0,1,2$) are their corresponding texture coordinates. \mathbf{P}_0 is the vertex for whom we are currently generating the tangent space.)

For every vertex in the mesh, the contributions from all triangles sharing that vertex are added together and the result is then normalized.

B.4. Per-vertex curvatures

To calculate the curvature parameters A and B for a mesh to represent the curvature function

$$Ax^2 + By^2 + z = 0$$

one repeats for each vertex:

1. Transform n neighbour vertices to current vertex' tangent space.
2. Apply a least square algorithm on their coordinates to find A and B.

a. Let

$$E = \begin{pmatrix} x_0^2 & y_0^2 \\ x_1^2 & y_1^2 \\ x_2^2 & y_2^2 \\ \vdots & \vdots \\ x_n^2 & y_n^2 \end{pmatrix}, \quad x = \begin{pmatrix} A \\ B \end{pmatrix}, \quad F = \begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix}$$

and solve

$$Ex = F$$

b. Solving:

$$Ex = F$$

$$E^T Ex = E^T F$$

$$x = (E^T E)^{-1} (E^T F)$$

3. Save the retrieved parameters in the vertex data passed to the vertex shader so that the parameters are interpolated for all pixels.

For the sphere used in this project, however, the solution was already known and hard-coded.



Appendix C: References

[RTMOrig00] Oliveira, Manuel M., Gary Bishop, David McAllister. **Relief Texture Mapping**. *Proceedings of SIGGRAPH 2000 (July 23-28, 2000, pp. 359-368)*
http://www.inf.ufrgs.br/%7Eoliveira/pubs_files/RTM.pdf

[RTMBasic05] Fabio Policarpo, Manuel M. Oliveira, João Comba. **Real-Time Relief Mapping on Arbitrary Polygonal Surfaces**. *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, Washington, DC, (April 3-6, 2005, pp. 155-162)*
http://www.inf.ufrgs.br/%7Eoliveira/pubs_files/Policarpo_Oliveira_Comba_RTRM_I3D_2005.pdf

[RTMAdv05] Manuel M. Oliveira, Fabio Policarpo. **An Efficient Representation for Surface Details**. *UFRGS Technical Report RP-351. (Jan 26, 2005)*
http://www.inf.ufrgs.br/%7Eoliveira/pubs_files/Oliveira_Policarpo_RP-351_Jan_2005.pdf

[RTMMulti06] Fabio Policarpo, Manuel M. Oliveira. **Relief Mapping of Non-Height-Field Surface Details**. *ACM SIGGRAPH 2006 Symposium on Interactive 3D Graphics and Games, Redwood City, CA, (Not yet published)*
http://www.inf.ufrgs.br/%7Eoliveira/pubs_files/Policarpo_Oliveira_RTM_multilayer_I3D2006.pdf

[AGDbook05] Alan Watt, Fabio Policarpo. **Advanced Game Development with Programmable Graphics Hardware**. *ISBN: 1-56881-240-X (chapters 2, 3, 4 and 5)*