BLEKINGE TEKNISKA HÖGSKOLA

Review of Displacement Mapping Techniques and Optimization

Ermin Hrkalovic
Mikael Lundgren
2012-05-01

Contents

1 Introduction 3

2 Purpose and Objectives 4

3 Research Questions 4

4 Research Methodology 4

5 Normal Mapping 4

6 Relief Mapping 5

7 Parallax Occlusion Mapping 7

8 Quadtree Displacement Mapping 8

9 Implementation 8

9.1 Optimization 12

10 Experiment Design 13

10.1 Test Environment 14

11 Results 14

11.1 Optimization Results 16

12 Discussion 16

13 Conclusion and Future Work 17

14 References 18

Appendix A 20

1 Introduction

This paper explores different bump mapping techniques and their implementation. So the reader should have an understanding of computer graphics and shading language such as HLSL.

Bump mapping is a technique that is used in computer games to make simple 3D objects look more detailed than what they really are. The technique involves using a texture to change the objects normals to simulate bumps and is used to avoid rendering high polygonal objects.
Over the years some different techniques have been developed based on bump mapping, these are normal mapping, relief mapping, parallax occlusion mapping, quadtree displacement mapping and so on.

The techniques that visually look the best are those that also cost the most in terms of performance to use. Relief mapping and parallax mapping are an example of these techniques.
Since they are computationally expensive, they are only used on objects that are close to the player. Other, less expensive techniques are then used on objects further away. This switch between techniques creates a transition that is hard to cover up.
To best avoid this transition is to use the more expensive algorithm for objects at longer distances and for this to be possible the algorithm has to be optimized.

The concept of bump mapping was introduced by [Blinn 1978] with his paper Simulation of Wrinkled Surfaces. The idea of bump mapping is to change the normals of a flat surface to simulate irregularities which makes the surface look more real. Bump mapping is easy to use and very efficient but it does not display correct depth at all angles.

Relief Texture Mapping was introduced [Oliviera et al. 2000] which works similar as the bump mapping, but it also displace texture coordinates. Afterwards a real time Relief Mapping was presented by [Oliviera et al. 2005]. This made it possible to use relief mapping in real time applications.

Parallax Mapping was introduced by [Kaneko et al. 2001] and it is similar to relief mapping, but it uses a different algorithm to calculate the offset for the texture coordinates. It was later improved by [Welsh 2004] with an offset limitation.

Parallax Occlusion Mapping [Brawley and Tatarchuk 2004; Tatarchuk 2006] is an improvement of parallax mapping. The main difference between these two techniques is that Parallax occlusion mapping uses an inverse height map tracing when computing the texture offset, more on this in section 7 Parallax Occlusion Mapping.

Pyramidal Displacement Mapping was introduced by [Kyoungsu et al. 2006] which is a technique that uses height maps with maximum mip maps for displacing texture coordinates. More on this in section 8 Quadtree Displacement Mapping. A more optimized version was presented by [Drobot 2009] called Quadtree Displacement Mapping.

The first part of this paper we go through our goals and our research methodology. We then write about four different techniques and describe how they work. We also go through how they are implemented. After that we start our experiments and measure the different techniques against each other. When the first testing has been done, we start to optimize the techniques and run a

second test to see how much faster, if it is faster, the optimization is compared to the previous tests. When the tests are done, we present our test data and analyse them. Finally we discuss the techniques and the testing. Then we finish up with a conclusion.

2 Purpose and Objectives

We want to expand our knowledge about creating realistic 3D environments with as simple geometry as possible. There are a number of different techniques that can achieve this. We will explore which one is the best to use in a real time application and how well are those techniques when compared to each other.
The goal is to develop an efficient algorithm for bump mapping that gives similar results to current technologies, or improve one of the existing techniques.

3 Research Questions

Is it possible to improve current techniques for bump mapping in terms of performance, e.g. is it possible to decrease the execution time for the technique?
Is it possible to improve the performance for current bump mapping techniques by decreasing the execution time?

4 Research Methodology

We are going to analyse different bump mapping techniques. Learn how they are implemented and Implement them. We are then going to design a test system which will be able to measure the time it takes to execute each technique. There will be a couple of tests based on; viewing angle, texture size and sampling filter. After the tests are done we will compare the result and try to optimize some of the techniques.

5 Normal Mapping

Figure 5.1 *Demonstration of normal mapping technique.*
On the left is normal mapping in front of the camera, figure on the right shows the disadvantage of normal mapping

Normal mapping is computationally inexpensive and easy to implement. To achieve bumps with this technique a texture with normal offsets is required. The texture is illustrated in Figure 5.2.

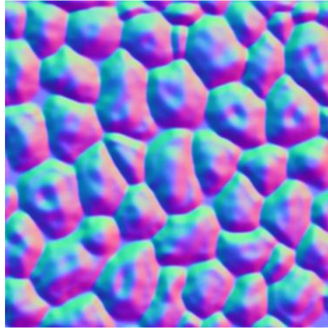

Figure 5.2 *Offset normals stored in a texture and mapped to the [0, 1] range*

The object that is being drawn has its normals facing one direction per polygon, this makes the object look flat. To create bumps you change the normals per pixel by sampling the normal map texture and transform the current normal with the sampled one.

Normal mapping is only affecting the normals, so then exposed to lights the object seems to be more detailed than it really is.
This technique has a drawback, when looking at objects from an oblique angle the illusion disappears as the normal mapping does not displace texture coordinates.

The normal mapping technique and its disadvantage is shown in Figure 5.1.

6 Relief Mapping

Figure 6.1 *Demonstration of relief mapping*

The biggest difference between normal mapping and relief mapping is that relief mapping actually displaces the texture coordinates to create real bumps, this is demonstrated in figure 6.1.

To achieve this displacement, a new texture is needed. While relief mapping has normal map texture, it also needs a height map texture, which is shown in figure 6.2.

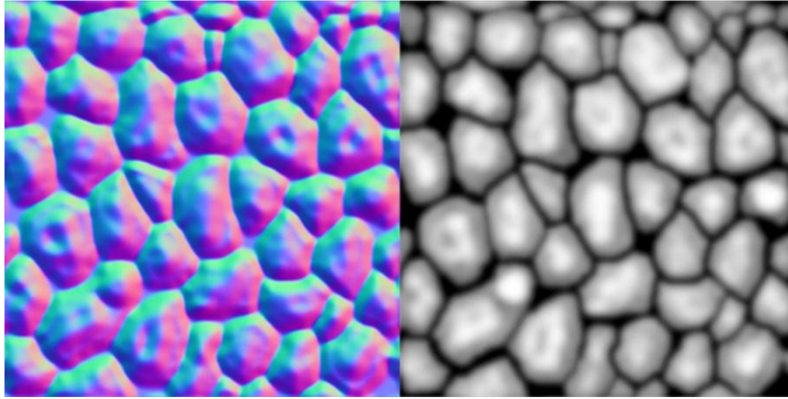

Figure 6.2 *To the left is the normal map texture and to the right is the height map texture, both are stored in the [0, 1] range.*

The height map shows the depth of the texture and it is used to correctly displace the texture coordinates, more on this in section 7 Implementation.

After the offset texture coordinate is calculated, we calculate the lightning the same way as normal mapping.

As described by [Tatarchuk 2006] the relief mapping uses a depth bias to remove artefacts. The depth bias clamps the offset vector so that the step size for relief calculation does not get to big. If the step size is too big then the linear search will miss its intersection point and get an incorrect one instead. This creates artefacts, as illustrated in figure 6.3.
This depth bias does provide an artefact free displacement, but it also flattens the depth at a distance. With this depth bias, the relief mapping is not extruding as much depth from far away compared to Parallax Occlusion Mapping. This gives the relief mapping almost even performance on all the angles.

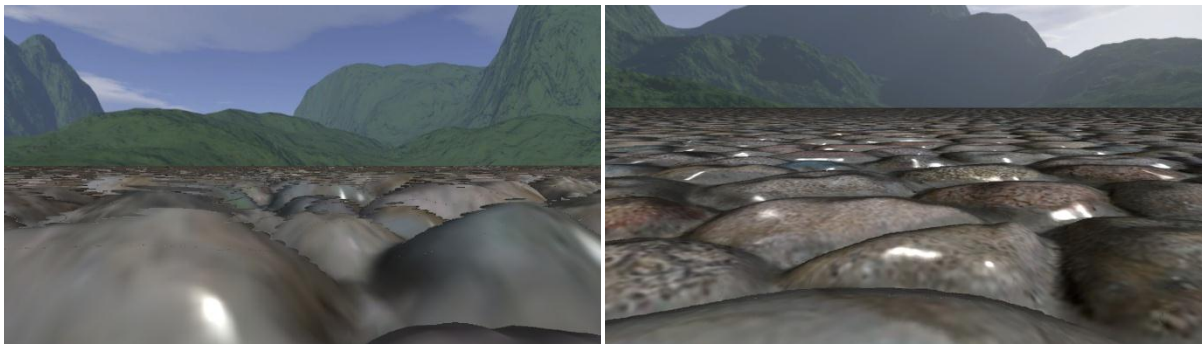

Figure 6.3 *To the left is relief mapping without depth bias and to the right is with depth bias*

7 Parallax Occlusion Mapping

Parallax Occlusion Mapping is using a slightly different approach when it comes to displace the texture coordinates. Instead of using a combination of a linear and a binary search, parallax occlusion only uses a linear search and approximating the height profile as a stepwise linear curve. As illustrated in Figure 7.2.

Figure 7.1 *Parallax occlusion mapping*

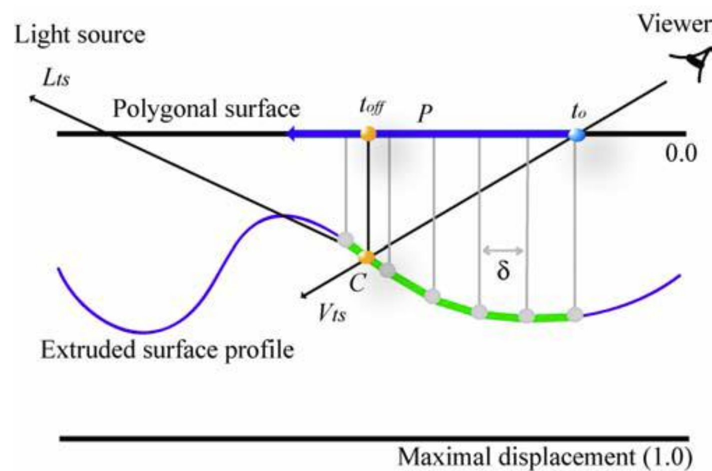

Figure 7.2 *Displacement based on sampled height field and current view direction [Tatarchuk 2006].*

To determine the right texture offset we step along the parallax offset vector *P* until the view direction vector *Vts* intersects with the height field. When we have the desired texture offset we add it to the original texture coordinates and do the lightning calculation based on new texture coordinates.

Parallax occlusion mapping also uses dynamic flow control to control the number of samples based on the viewing angle. When the viewing angle is oblique the number of samples increase to avoid aliasing artefacts. The dynamic flow control is based on a min and a max sample value which the artist can change depending on the texture.

8 Quadtree Displacement Mapping

Quadtree Displacement Mapping differs a lot from relief and parallax occlusion. It uses mipmaps when calculating the height field intersection. A mipmap texture is composed of several levels where the first level is the original image and the other levels are smaller versions of the previous level. This is illustrated in figure 8.2. For Quadtree Displacement Mapping each pixel from a level store the minimum displacement value for corresponding four pixels in the level below. This height map is then used for finding the intersection with the view ray. By stepping down from the smallest mipmap with highest height value to the lowest level, it will find the intersection. More on this in section 9 Implementation.

Figure 8.1 *Demonstration of quadtree displacement mapping*

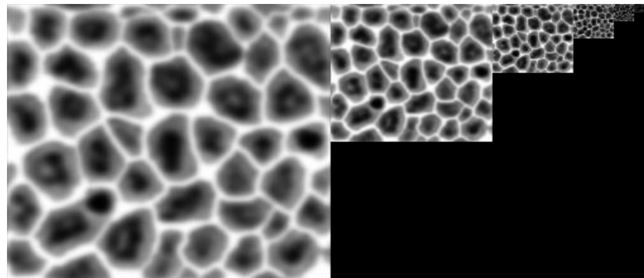

Figure 8.2 *Mipmaps for Quadtree Displacement Mapping*

9 Implementation

Normal Mapping:

Normal mapping achieves bumps by using normal maps to change the objects normals, which were explained in section 5 Normal Mapping.

The normal map texture has its normals in tangent space and are mapped to the [0, 1] ranged. The new offset normals needs to be mapped to the [-1, 1] ranged before being used. To get the bumped normals in correct space as the original normals, The bump normals needs to be transformed into world space, or transform the original normals to tangent space.
To change from or to tangent space a matrix is required. This matrix is composed by normal, binormal and tangent from the object, this matrix is also known as TBN (Tangent Binormal Normal).

Relief Mapping:
Relief mapping uses a height map and a view vector to calculate a texture offset, which is used to create an illusion of bumps. This was described in section 6 Relief Mapping.

To find the correct offset, a ray between the camera position and pixel position is created and then a form of ray tracing is done to see where on the height map the intersection is.

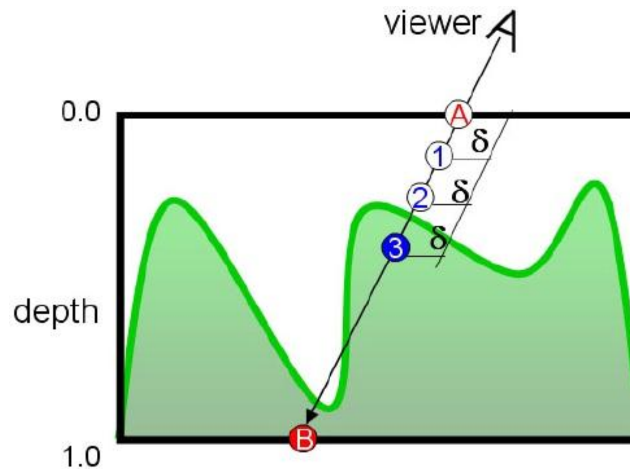

Figure 9.1 *Ray intersection on height map [Oliviera et al. 2005]*

There are few ways to find the intersection. Binary search on the ray is the fastest way to find the intersection. As described by [Oliviera et al. 2005] the binary search can get incorrect values if more than one intersection point is present, as shown in figure 9.2

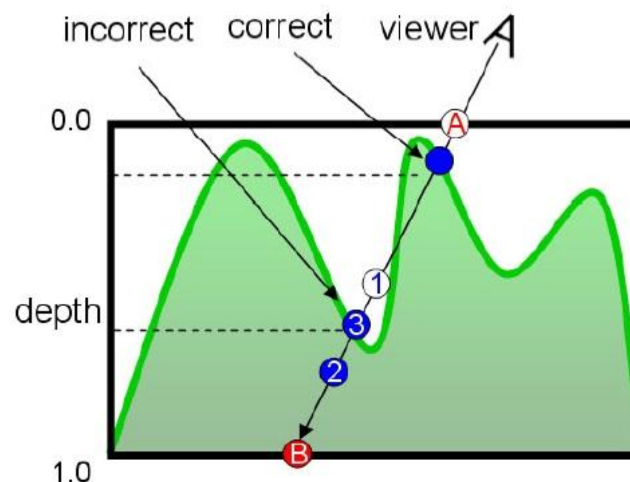

Figure 9.2 *Several intersection points [Oliviera et al. 2005]*

To correct this, the method can be switched to linear search.
Linear search steps on the ray in a fixed size until the intersection is found. This technique passes the intersection point, this will give some strange artefacts. These artefacts can be eliminated by using a binary search at the end, more information about this is presented by [Oliviera et al. 2005].

Most of the calculations for relief mapping are done in the pixel shader.
In the vertex shader we transform the objects position, normal, tangent and binormal to world space and pass it through to the pixel shader.

In the pixel shader we calculate the view direction (VD) which is a vector between the cameras and the pixels position. The VD is then transformed to tangent space, so that the texture space can be used to find the intersection point.

Step along the VD and sample the height map per iteration. Check the height map value with the VD z position, if the VD position is below the height map value, then the intersection point has been found.

The intersection value that the linear search returns has already passed the intersection point, so to find the approximate value the binary search is used between the current and the previous point.

Binary search return our final texture coordinate offset. The lightning calculation is then using this offset to get the correct shading.

Parallax Occlusion Mapping:
To create the illusion of bumps, parallax occlusion calculates a texture offset by finding the intersection between a view ray and a height field. This was explained in section 7 Parallax Occlusion Mapping.

To begin with we calculate and transform the view direction vector and the light direction vector to tangent space in the vertex shader. We also calculate the parallax offset vector in tangent space which determine the maximum parallax offset which is described in [Brawley and Tatarchuk 2004], see figure 9.3 (P).
To be able to compute how many samples that are necessary based on the viewing angle in the pixel shader we need to pass the world space normal and the view directional vector from the vertex shader.

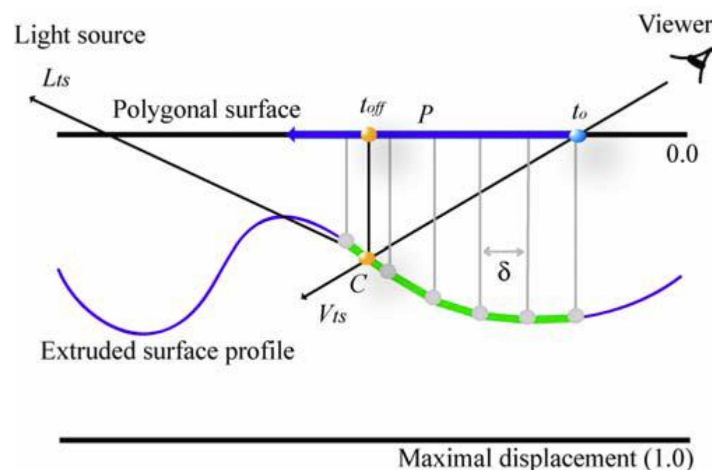

Figure 9.3 *Displacement based on sampled height field and current view direction [Tatarchuk 2006].*

In the pixel shader we start with normalizing the view, light and normal vectors, then we calculate the number of samples based on the viewing angle. We then calculate the step size based on number of steps. After that we calculate the texture offset per step based on the step size and the parallax offset.
We use the step size to step from 1 to 0 during the iteration and we call this the current bound. The

texture offset per step is used for sampling the height map during the iteration. When the sampling value is bigger than the current bound we've found our intersection, all this is shown in figure 9.4.

```
float StepSize   = 1.0 / (float) NrOfSteps;
float2 TexOffsetPerStep = StepSize * In.ParallaxOffsetTS;
while (StepIndex < NrOfSteps)
{
    TexCurrentOffset -= TexOffsetPerStep;
    CurrentHeight = NormalHeight.SampleGrad(filter, TexCurrentOffset, dx, dy).a;
    CurrentBound -= StepSize;
    if (CurrentHeight > CurrentBound)
    {
        Pt1 = float2(CurrentBound, CurrentHeight);
        Pt2 = float2(CurrentBound + StepSize, PreviousHeight);
        StepIndex = NrOfSteps + 1;
    }
    else
    {
        StepIndex++;
        PreviousHeight = CurrentHeight;
    }
}
```

Figure 9.4 *Intersection calculation*

When the intersection has been found, we calculate the new texture offset coordinate and do the lightning calculation based on it.

Quadtree Displacement Mapping:

As Relief and Parallax Occlusion mapping it calculates the intersection between the view ray and the height field. This is described in section 8 Quadtree Displacement Mapping.

To find the intersection point between the viewing ray and the height field we start sampling the height from the highest level with the smallest resolution. We then compare the sampled height with the view ray. If the height of the ray is above the sampled height we advance the ray to that height. We then continue with the iteration by traversing down one level. This continues until we find our intersection with the height field. See Figure 9.5.

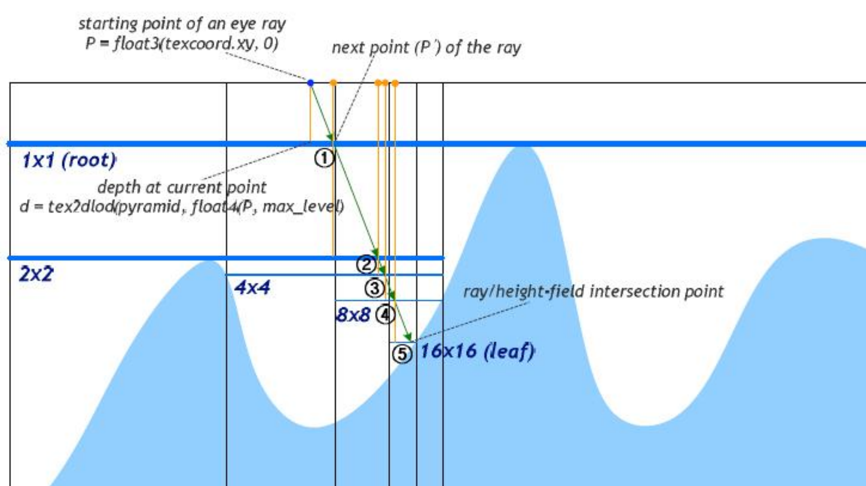

Figure 9.5 *An example of a ray/height field intersection using minimum mipmaps [Kyoungsu et al. 2006].*

We might miss an intersection when advancing down the ray. To avoid node crossing, we go up one level and continue the iterations. This is illustrated in Figure 9.6.

When the offset texture coordinate has been found, we proceed to do the lightning calculation.

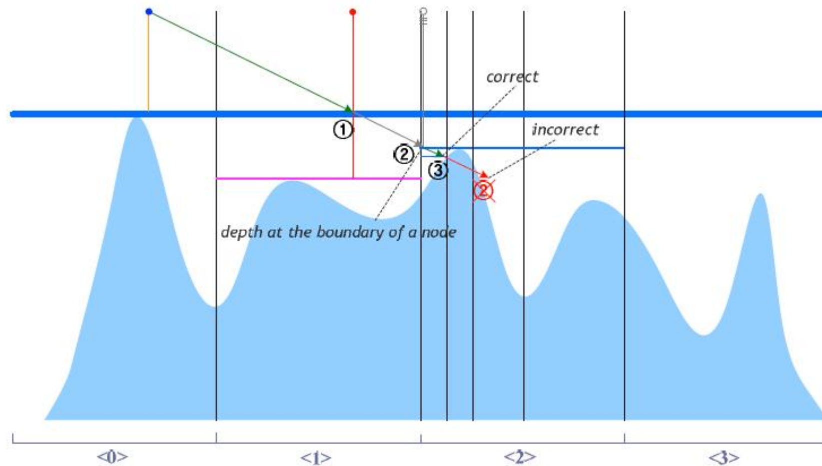

Figure 9.6 *This shows how the ray may miss an intersection when crossing a node.*

9.1 Optimization

We found an optimization for parallax occlusion mapping. Instead of using SampleGrad when sampling from the height map we used SampleLevel with a calculated mip level.
SampleGrad uses gradient to influence the way the sample location is calculated while SampleLevel samples on a specific mipmap level.
These two sample techniques results are almost identical, at a far distance the two sample techniques start to get different results.

While experimenting we found out that SampleLevel was much faster than SampleGrad. The code we used to calculate the current mip level was found in two different DirectX SDK examples. The mip level calculations is illustrated in figure 9.9.

The visual result does not differ very much from the original and we did gain a performance boost, see figure 9.7.

Figure 9.7 *Difference on parallax occlusion mapping.*
To the left is unoptimized, and to the right is the optimized.

The Quadtree displacement mapping iterates from highest mip level to the lowest. This requires a high number of iterations per pixel.

As mentioned in [Kyoungsu et al. 2006] the use of current mip level on the pixel instead of lowest mip level would decrease the number of iterations which does not affect the visual result but gives a significant performance boost.

Figure 9.8 *Difference on quadtree displacement mapping.*
To the left is unoptimized, and to the right is the optimized.

Both these optimizations use the current mip level for their calculations, figure 9.9 show a piece of code that we use to calculate the mip level.

```
float2 dxSize, dySize;
float2 dx, dy;
float  MipLevel;
float2 TexCoords;
float2 TexCoordsPerSize;

TexCoordsPerSize = In.TextureCoord * float2(g_TexSize,g_TexSize);

float4( dxSize, dx ) = ddx( float4( TexCoordsPerSize, In.TextureCoord ) );
float4( dySize, dy ) = ddy( float4( TexCoordsPerSize, In.TextureCoord ) );

TexCoords = dxSize * dxSize + dySize * dySize;

// Compute the current mip level
MipLevel = max( 0.5f * log2( max( TexCoords.x, TexCoords.y ) ), 0 );
```

Figure 9.9 *Code for calculation of current mip level*

10 Experiment Design

We used two different systems to test the different techniques. The tests are based on the GPU so for the purpose of the tests we have tested the algorithms on both Nvidia and AMD hardware.

The two test systems are listed below

1.
OS: Windows 7, 64bit
CPU: Intel Core 2 Quad Q6600 @2.40Ghz
RAM: 4 Gigabyte
GPU: AMD Radeon HD 6950

2.
OS: Windows 7, 64bit
CPU: Intel Core 2 Quad Q9550 @2.83Ghz

RAM: 4 Gigabyte
GPU: Nvidia Geforce GTX 560Ti

Our tests are based on DirectX 11.

We have used the same lightning calculation for all the techniques to make the tests as fair as possible. All the techniques uses normal mapping to get correct lightning, that means that all techniques will be slower compared to normal mapping. Since normal mapping is not displacing textures, more of this in section 6 Relief Mapping, the tests are more focused on relief, parallax occlusion and quadtree displacement mapping.

10.1 Test Environment

For our experiments, we created a test environment that suites our need the best.
It has support for reloading shaders in real time so that the changes in the code can be seen without having to restart the application.

The environment also has support to change between our four techniques in real time. It can also change between three different angles, four different texture sizes and 2 different sample filter algorithms.
For ease of use, the environment can also record the time it takes for a shader/technique to execute, this is crucial for us then we are going to optimize the techniques.

And finally all this data that we record, e.g. the time for executions of a shader, is stored in a container so that it can be saved on to a file.
With our data saved we can look back and see the difference in the techniques on different hardware and to measure how well we have optimized the techniques.

11 Results

Below we see some of the result from the test where we measured the draw time for different techniques with different sampler filters. To get a more accurate execution time, we measured 1000 draw calls for each and saved the average value.

Relief mapping uses 8-25 samples on both the linear and binary search, while parallax occlusion mapping uses 8-50 samples. Quadtree displacement mapping uses a maximum of 50 iterations.

Here we used textures with the size 512x512 pixels and tried three different viewing angles; 0°, 45° and 90°. All the values are in milliseconds.

For a complete list of the result see Appendix B.

Figure 11.1 *The different view angles (*0°, 45° and 90°).

The table results below are from the first test. To see the second test results go to 11.1 Optimization Results.

Radeon HD6950

Fastest Fast Slow Slowest.

Filter	Normal	Relief	Parallax Occlusion	Quadtree Displacement
Linear	0.0414592	0.988021	0.597455	1.22001
Anisotropic 4x	0.144124	1.19647	1.6966	1.23389
Anisotropic 8x	0.254578	1.31984	2.83429	1.2656
Anisotropic 16x	0.317614	1.43258	3.71574	1.34473

Table 11.1 *Rendering times in ms for viewing angle 0°.*

Filter	Normal	Relief	Parallax Occlusion	Quadtree Displacement
Linear	0.116145	1.12741	0.910676	1.43893
Anisotropic 4x	0.214509	1.20899	1.65031	1.45568
Anisotropic 8x	0.207091	1.21245	1.62604	1.4638
Anisotropic 16x	0.207046	1.22482	1.63927	1.47446

Table 11.2 *Rendering times in ms for viewing angle* 45°.

Filter	Normal	Relief	Parallax Occlusion	Quadtree Displacement
Linear	0.231026	1.0222	0.741321	1.06061
Anisotropic 4x	0.23116	1.05558	0.785202	1.06757
Anisotropic 8x	0.230637	1.05596	0.785526	1.06769
Anisotropic 16x	0.230836	1.05578	0.785479	1.0675

Table 11.3 *Rendering times in ms for viewing angle* 90°.

Nvidia 560Ti

Filter	Normal	Relief	Parallax Occlusion	Quadtree Displacement
Linear	0.0522142	1.27029	0.694496	2.13065
Anisotropic 4x	0.164093	1.44114	2.04695	2.19099
Anisotropic 8x	0.335241	1.58348	3.79533	2.26532
Anisotropic 16x	0.495902	1.79593	5.77	2.41668

Table 11.4 *Rendering times in ms for viewing angle* 0°.

Filter	Normal	Relief	Parallax Occlusion	Quadtree Displacement
Linear	0.143816	1.89456	1.14114	2.56082
Anisotropic 4x	0.29945	2.04922	2.52369	2.60873
Anisotropic 8x	0.299494	2.07468	2.5501	2.62108
Anisotropic 16x	0.299492	2.10885	2.60283	2.63976

Table 11.5 *Rendering times in ms for viewing angle* 45°.

Filter	Normal	Relief	Parallax Occlusion	Quadtree Displacement
Linear	0.296405	1.60084	1.29313	1.79316
Anisotropic 4x	0.289469	1.67712	1.32395	1.80066
Anisotropic 8x	0.289893	1.67778	1.32408	1.80099
Anisotropic 16x	0.290078	1.67837	1.32423	1.80085

Table 11.6 *Rendering times in ms for viewing angle* 90°.

Since the parallax occlusion mapping uses SampleGrad, it gets a big performance hit when using anisotropic filtering.
At more oblique angles the quadtree displacement mapping is faster because it has less iterations compared to relief and parallax occlusion mapping.
Relief mapping has almost the same performance on all the angles, this is because of depth flattening at longer distances, see section 6 Relief Mapping.

Radeon 6950 Average				
Angle	Normal	Relief	Parallax Occlusion	Quadtree Displacement
0	0,192662256	2,578993563	2,201465438	2,341294313
45	0,172152288	3,798389688	1,375012188	2,9595675
90	0,195427694	3,389628813	0,687281188	2,241958313

GeForce 560 Ti Average				
Angle	Normal	Relief	Parallax Occlusion	Quadtree Displacement
0	0,270060619	2,30991125	3,154385438	2,62711375
45	0,236245063	3,521849375	2,00498625	3,43129125
90	0,24766925	3,293705625	1,139626063	2,621240625

Table 11.7 *Average values per technique in milliseconds.*

Table 11.7 shows the average values for all techniques where all the filters and texture sizes are included. As we can the parallax occlusion is faster than the others, normal mapping not included.

11.1 Optimization Results

Textures with the size 2048x2048 where used in this tests

Angle	Filter	Parallax Occlusion Before/After		Quadtree Displacement Before/After	
0°	Linear	0.598871	0.457204	4.48161	0.387646
0°	A 4x	2.00846	0.529452	4.81232	0.472049
0°	A 8x	3.15276	0.574531	4.95884	0.576332
0°	A 16x	4.03739	0.650836	5.14188	0.679151
45°	Linear	0.911373	0.85679	6.61346	1.0459
45°	A 4x	1.70703	0.937171	6.9566	1.10159
45°	A 8x	1.66943	0.922072	6.9726	1.10821
45°	A 16x	1.68331	0.932344	6.98718	1.1091
90°	Linear	0.742833	0.737984	5.47207	1.25578
90°	A 4x	0.789082	0.781308	5.55074	1.26208
90°	A 8x	0.788846	0.781625	5.55378	1.26192
90°	A 16x	0.789054	0.781953	5.551	1.26194

Table 11.8 *Results from the optimization.*

As discussed in section 9.1 Optimization, while the visual difference was almost unnoticeable we gained a significant performance boost with our optimization, as you can see in Table 11.8 above.

12 Discussion

From the first test we see that the techniques differ a lot when it comes to different viewing angles, texture size and filters. For Relief and Quadtree Displacement Mapping the biggest issue was the size of the textures, the bigger they were the worse was the performance. Relief with 256 to 2048 texture

sizes went from 0.78 ms to 9.06 ms. Quadtree displacement went from 1.06 ms to 6.61 ms. While parallax occlusion was not much effected by the texture sizes.

For Parallax Occlusion Mapping the issues were on the other hand the viewing angle and the anisotropic filtering. At oblique angles the sampling rate was much higher which led to bad performance. Parallax occlusion with angle 0 and texture size 512 went from 0.61 ms on linear filter to 3.72 ms with anisotropic filtering 16x.

Even if these techniques are similar both in implementation and visual they all got their strength and weaknesses. For example the Relief and the Quadtree Displacement Mapping handles the different filters very well. Parallax occlusion works great with linear filter and big textures.

From our optimization result we see that both Parallax Occlusion and Quadtree Displacement become much faster when using the sampling function SampleLevel with calculated mipmaps.
At an angle 0 with anisotropic filtering 16x which is worst case scenario for parallax occlusion, the result went from 4.04 ms to 0.65 ms. This is a significant improvement in execution time.
At an angle 45 with anisotropic filtering 16x which is worst case scenario for quadtree displacement, the results went from 6.99 ms to 1.11 ms. This is also a significant improvement over the unoptimized code.

As you can see in figure 12.1, the visual result from before and after the optimization is almost identical.

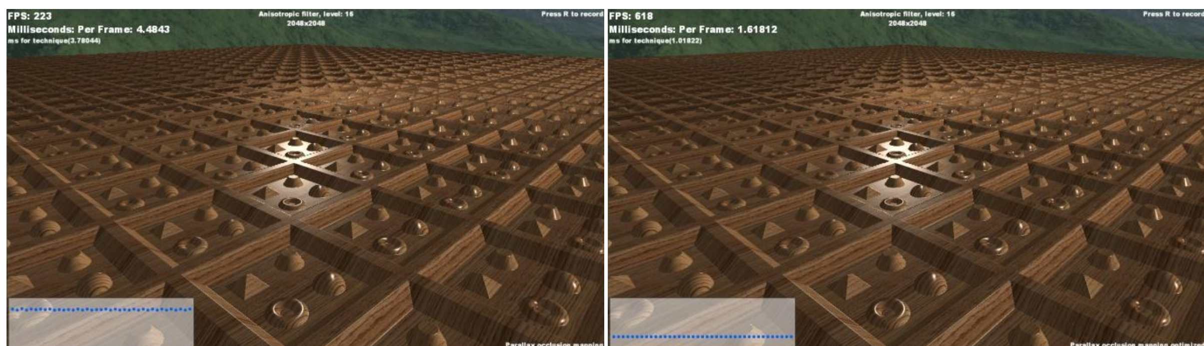

Figure 12.1 *Parallax occlusion mapping, before and after.*

With this new optimization for Parallax Occlusion we saw from our tests that it is the fastest and visually the best looking compare to the other techniques.

We tried to combine Parallax Occlusion with Quadtree displacement Mapping. We gained a small performance boost, but the transaction between the two techniques was visible.

13 Conclusion and Future Work

With our tests we found out that it was possible to optimize few of these techniques.
The different bump mapping techniques we studied showed how they differed in performance and visual results while they are based on the same principle idea, to create real depth on all angles.

From our test we found out that Parallax Occlusion Mapping is the best one suited for real time applications.
While the others work, they are still too expensive to use compared to Parallax Occlusion.

Games often have different geometry on screen at any given time. All from advanced like a human character to more simple, such as a box. All these shapes interact and adding different techniques could have different results based on the interaction. Our experiments were all based on a plane. While this was sufficient for our thesis, how would the results been when introduced these techniques to a game world based on different geometry?

In today's graphics hardware and software there have been some advancements, such as tessellation. Tessellation is a technique when using the graphics card to draw a simple geometry it adds more triangles to the geometry. When tessellation is applied with a displacement map, the triangles get new positions and with that create a real bumpy road instead of an illusion.
This is something we also wanted to try but never had a chance. How would the different displacement techniques stack against tessellation with displacement mapping?

In conclusion, our tests and optimization were successful. For future work, there are probably some things that can be improved, especially with the new DirectX 11 hardware and software.

For example, [Drobot 2006] mentioned that the quadtree displacement mapping can be further optimized if some of the calculations were switched to integer based math instead. While we did not have time to implement it our self, this is something that can be done in the future.

Tessellation is the new technique that has hardware support, in theory this plus a displacement map should make the tessellation faster than the other techniques. This can be tested in the future.

Applying these techniques on a plane is simple yet not something that is useful for games. To better get an understanding of how these techniques work, they need to be tested in a game scene with a lot more geometry. This is also something that can be tested in the future.

14 References

BLINN, J. F. 1978. "Simulation of Wrinkled Surfaces". In Proceeding of the 5th annual conference on Computer graphics and interactive techniques, ACM Press, pp. 286-292.

OLIVEIRA, M. M., BISHOP, G., AND MCALLISTER, D. 2000. Relief texture mapping. In Siggraph 2000, Computer Graphics Proceedings, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, K. Akeley, Ed., pp. 359–368.

POLICARPO, F., OLIVEIRA, M. M., COMBA, J. 2005. Real-Time Relief Mapping on Arbitrary Polygonal Surfaces. In ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games Proceedings, ACM Press, pp. 359-368.

KANEKO, T., TAKAHEI, T., INAMI, M., KAWAKAMI, N., YANAGIDA, Y., MAEDA, T., TACHI, S. 2001. Detailed Shape Representation with Parallax Mapping. In Proceedings of ICAT 2001, pp. 205-208.

WELSH, T. 2004. Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces. Tech. rep., Infiscape Corporation.

BRAWLEY, Z., AND TATARCHUK, N. 2004. Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Mao Tracing. In ShaderX3: Advance Rendering with DirectX and OpenGL, Engel, W., Ed., Charles River Media, pp. 135-154.

TATARCHUK, N. 2006. Dynamic Parallax Occlusion with Approximate Soft Shadows. In Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games, ACM Press, pp.63-69.

Kyoungsu, O., Hyunwoo, K., Cheol-Hi, L. 2006. Pyramidal Displacement Mapping: A GPU based Artifacts-Free Ray Tracing through an Image Pyramid.

Drobot, M. 2009. Quadtree Displacement Mapping with Height Blending. In GPU Pro, Engle, W., Ed., A K Peters, Ltd, pp. 117-148.

Appendix A

Radeon HD6950

Viewing angle: 0°

Fastest Fast Slow Slowest

Resolution	Filter	Normal	Relief	Parallax Occlusion	Quadtree Displacement
256x256	Linear	0.041145	0.596895	0.594725	0.964414
256x256	Anisotropic 4x	0.0903284	0.638763	1.32802	0.976285
256x256	Anisotropic 8x	0.143335	0.677777	2.01784	1.0077
256x256	Anisotropic 16x	0.201423	0.734231	2.86991	1.07642
512x512	Linear	0.0414592	0.988021	0.597455	1.22001
512x512	Anisotropic 4x	0.144124	1.19647	1.6966	1.23389
512x512	Anisotropic 8x	0.254578	1.31984	2.83429	1.2656
512x512	Anisotropic 16x	0.317614	1.43258	3.71574	1.34473
1024x1024	Linear	0.0414404	2.9647	0.598766	1.99511
1024x1024	Anisotropic 4x	0.192374	3.26631	2.00338	2.18982
1024x1024	Anisotropic 8x	0.309524	3.3881	3.14386	2.31102
1024x1024	Anisotropic 16x	0.371774	3.50505	4.02538	2.48106
2048x2048	Linear	0.0415791	4.83137	0.598871	4.48161
2048x2048	Anisotropic 4x	0.198054	5.1244	2.00846	4.81232
2048x2048	Anisotropic 8x	0.31593	5.24546	3.15276	4.95884
2048x2048	Anisotropic 16x	0.377914	5.35393	4.03739	5.14188

Viewing angle: 45°

Resolution	Filter	Normal	Relief	Parallax Occlusion	Quadtree Displacement
256x256	Linear	0.0978696	0.780093	0.820452	1.06564
256x256	Anisotropic 4x	0.112552	0.787664	1.1392	1.06948
256x256	Anisotropic 8x	0.111455	0.791952	1.1363	1.07336
256x256	Anisotropic 16x	0.111454	0.795276	1.14013	1.07793
512x512	Linear	0.116145	1.12741	0.910676	1.43893
512x512	Anisotropic 4x	0.214509	1.20899	1.65031	1.45568
512x512	Anisotropic 8x	0.207091	1.21245	1.62604	1.4638
512x512	Anisotropic 16x	0.207046	1.22482	1.63927	1.47446
1024x1024	Linear	0.117483	3.61549	0.911144	2.2524
1024x1024	Anisotropic 4x	0.231834	3.95278	1.7055	2.46843
1024x1024	Anisotropic 8x	0.219522	3.94943	1.66788	2.48093
1024x1024	Anisotropic 16x	0.21942	3.9742	1.68215	2.5022
2048x2048	Linear	0.117173	9.06299	0.911373	6.61346
2048x2048	Anisotropic 4x	0.231792	9.43471	1.70703	6.9566
2048x2048	Anisotropic 8x	0.21953	9.41696	1.66943	6.9726
2048x2048	Anisotropic 16x	0.219561	9.43902	1.68331	6.98718

Viewing angle: 90°

Resolution	Filter	Normal	Relief	Parallax Occlusion	Quadtree Displacement
256x256	Linear	0.086931	0.75769	0.421737	0.848571
256x256	Anisotropic 4x	0.0869273	0.753898	0.419655	0.848711
256x256	Anisotropic 8x	0.0869414	0.753606	0.41951	0.848712
256x256	Anisotropic 16x	0.0869094	0.753537	0.41942	0.848779
512x512	Linear	0.231026	1.0222	0.741321	1.06061
512x512	Anisotropic 4x	0.23116	1.05558	0.785202	1.06757
512x512	Anisotropic 8x	0.230637	1.05596	0.785526	1.06769
512x512	Anisotropic 16x	0.230836	1.05578	0.785479	1.0675
1024x1024	Linear	0.231331	2.69979	0.742324	1.50041
1024x1024	Anisotropic 4x	0.231153	2.78077	0.788847	1.52896
1024x1024	Anisotropic 8x	0.231513	2.78261	0.789135	1.52796
1024x1024	Anisotropic 16x	0.231471	2.78664	0.788528	1.52827
2048x2048	Linear	0.232283	8.92121	0.742833	5.47207
2048x2048	Anisotropic 4x	0.232555	9.01823	0.789082	5.55074
2048x2048	Anisotropic 8x	0.23245	9.01802	0.788846	5.55378
2048x2048	Anisotropic 16x	0.232719	9.01854	0.789054	5.551

Nvidia 560Ti

Viewing angle: 0°

Resolution	Filter	Normal	Relief	Parallax Occlusion	Quadtree Displacement
256x256	Linear	0.0523135	0.857604	0.681431	1.7273
256x256	Anisotropic 4x	0.105029	0.948786	1.39052	1.76378
256x256	Anisotropic 8x	0.176597	1.03575	2.3011	1.81138
256x256	Anisotropic 16x	0.322191	1.17702	4.12239	1.9217
512x512	Linear	0.0522142	1.27029	0.694496	2.13065
512x512	Anisotropic 4x	0.164093	1.44114	2.04695	2.19099
512x512	Anisotropic 8x	0.335241	1.58348	3.79533	2.26532
512x512	Anisotropic 16x	0.495902	1.79593	5.77	2.41668
1024x1024	Linear	0.0523865	2.34466	0.696047	2.49477
1024x1024	Anisotropic 4x	0.239827	2.65563	2.73663	2.61558
1024x1024	Anisotropic 8x	0.430849	2.82939	4.703	2.71535
1024x1024	Anisotropic 16x	0.592554	3.04999	6.72397	2.89009
2048x2048	Linear	0.0524907	3.62715	0.695953	3.51562
2048x2048	Anisotropic 4x	0.236977	3.94255	2.73569	3.71016
2048x2048	Anisotropic 8x	0.426431	4.09983	4.68665	3.83896
2048x2048	Anisotropic 16x	0.585874	4.29938	6.69001	4.02549

Viewing angle: 45°

Resolution	Filter	Normal	Relief	Parallax Occlusion	Quadtree Displacement
256x256	Linear	0.121405	1.17685	1.01767	1.8415
256x256	Anisotropic 4x	0.14328	1.18737	1.24402	1.85566
256x256	Anisotropic 8x	0.143311	1.18996	1.24825	1.86072
256x256	Anisotropic 16x	0.143388	1.19054	1.25241	1.86753
512x512	Linear	0.143816	1.89456	1.14114	2.56082
512x512	Anisotropic 4x	0.29945	2.04922	2.52369	2.60873
512x512	Anisotropic 8x	0.299494	2.07468	2.5501	2.62108
512x512	Anisotropic 16x	0.299492	2.10885	2.60283	2.63976
1024x1024	Linear	0.144141	3.80114	1.14088	3.73334
1024x1024	Anisotropic 4x	0.315874	4.12118	2.65735	3.85565
1024x1024	Anisotropic 8x	0.316203	4.15927	2.68737	3.88012
1024x1024	Anisotropic 16x	0.31607	4.20558	2.7478	3.91029
2048x2048	Linear	0.143838	6.47818	1.14151	5.22849
2048x2048	Anisotropic 4x	0.3167	6.87463	2.6672	5.44388
2048x2048	Anisotropic 8x	0.316762	6.89909	2.69943	5.48492
2048x2048	Anisotropic 16x	0.316697	6.93849	2.75813	5.50817

Viewing angle: 90°

Resolution	Filter	Normal	Relief	Parallax Occlusion	Quadtree Displacement
256x256	Linear	0.12134	1.11529	0.616717	1.4101
256x256	Anisotropic 4x	0.121221	1.093	0.606568	1.40991
256x256	Anisotropic 8x	0.121462	1.09173	0.6057	1.40994
256x256	Anisotropic 16x	0.121456	1.09102	0.605662	1.41009
512x512	Linear	0.296405	1.60084	1.29313	1.79316
512x512	Anisotropic 4x	0.289469	1.67712	1.32395	1.80066
512x512	Anisotropic 8x	0.289893	1.67778	1.32408	1.80099
512x512	Anisotropic 16x	0.290078	1.67837	1.32423	1.80085
1024x1024	Linear	0.295829	4.00303	1.29502	2.70879
1024x1024	Anisotropic 4x	0.289111	4.1001	1.32941	2.73816
1024x1024	Anisotropic 8x	0.289106	4.09971	1.3299	2.73892
1024x1024	Anisotropic 16x	0.28905	4.10134	1.33005	2.73833
2048x2048	Linear	0.291434	6.26123	1.28669	4.50833
2048x2048	Anisotropic 4x	0.285635	6.36871	1.32077	4.55672
2048x2048	Anisotropic 8x	0.285619	6.36933	1.32093	4.55757
2048x2048	Anisotropic 16x	0.2856	6.37069	1.32121	4.55733