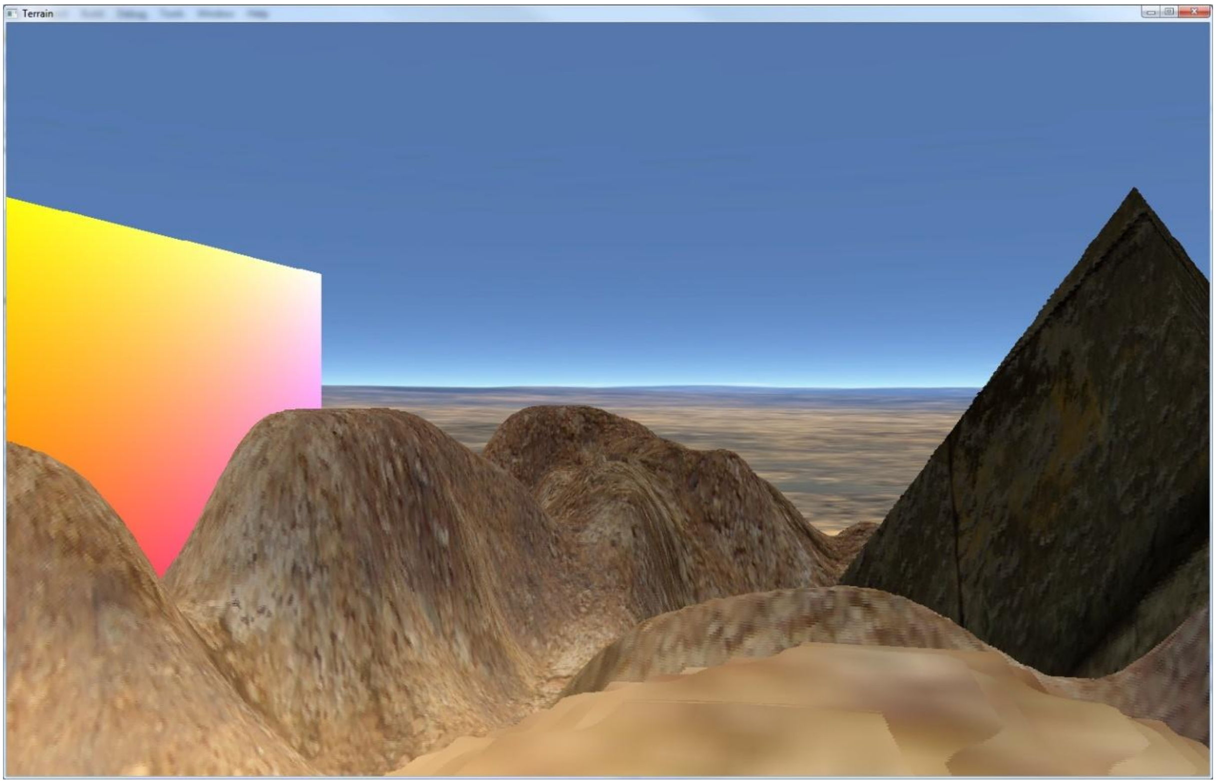
Marijn Kentie 1213660
m.a.kentie@student.tudelft.nl
IN4151 Computer Graphics Assignment 2

On-GPU ray traced terrain



1. Introduction

Before the prevalence of polygon-based 3D accelerated graphics, voxel-based terrain was a popular approach to drawing the environments for game and simulation software (see Figure 1). This terrain was always drawn by the CPU, and the introduction of fast hardware accelerated triangle rendering pushed voxels to the background. Unpopular as they currently are, voxel terrains do have a number of advantages over their polygonal counterparts: curved surfaces and real-time deformations are possible, and the terrain is extremely easy to generate from a height map texture (see Figure 2). A height map codes terrain height as an image. In a grayscale image, for example, white can mean maximum height while black means the base level.

Figure 1: Voxel terrain in the video game *Comanche*

Modern, fully programmable GPUs are equipped with flexible pixel shading units with facilities such as fast texture lookups and support for branching. As such, it seems promising to use the GPU to draw pixel-space terrain. This report details an experiment to do this: using a height map to draw terrain completely on the GPU, requiring a minimal amount of vertex data. Ease of implementation, features, flexibility and performance are investigated.

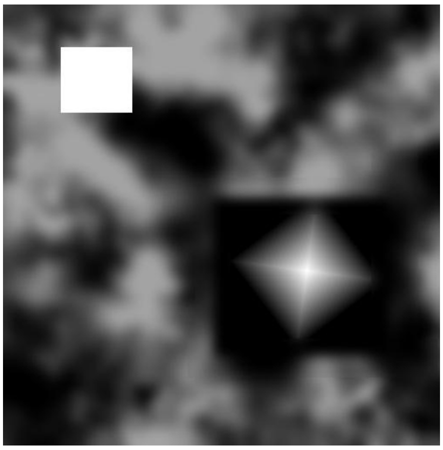


Figure 2: Height map image

2. Background

Using the GPU to create the illusion of relief is a staple technique in the video game industry. Bump mapping, a technique which uses a lookup texture to warp a surface's normal (and as such influence its lighting), has been used for years to create the illusion of depth, for example to draw the bumpy bricks of a wall. Parallax mapping builds on this by modifying the actual texture UV coordinates. Both are described in [1]. Finally, relief mapping [2], also known as parallax occlusion mapping [3], extends this further by allowing the perturbations to occlude one another. Figure 3 shows a comparison of the mentioned techniques; note the differences in quality at grazing angles. Clearly, parallax occlusion mapping creates the best illusion of relief.

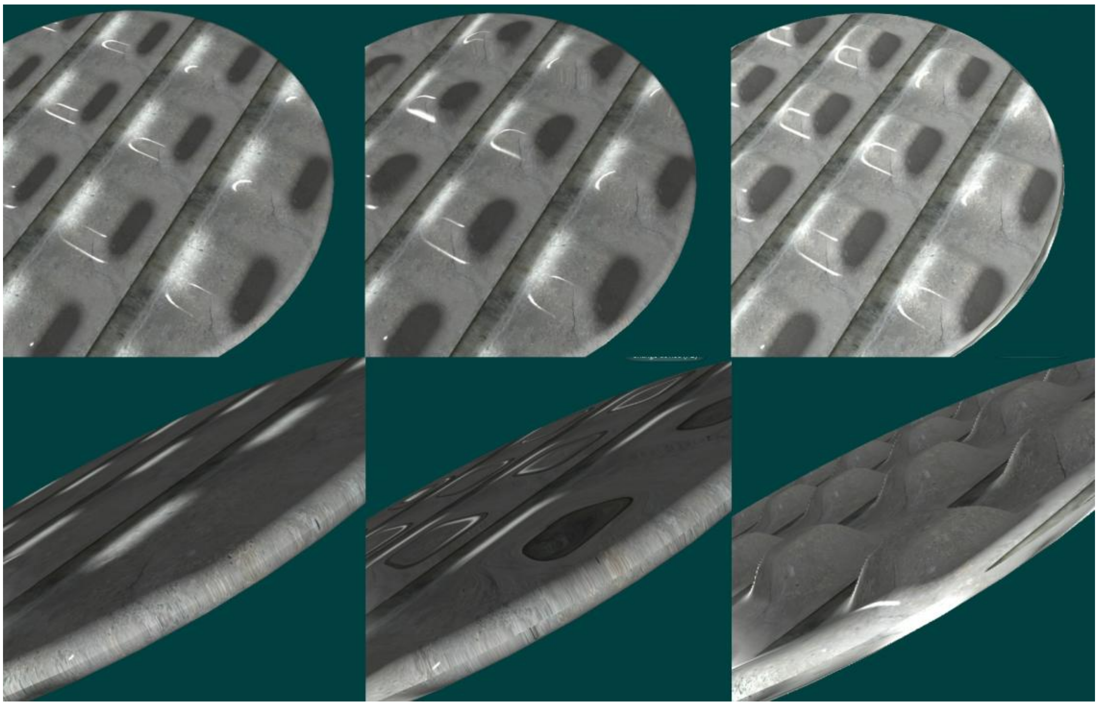


Figure 3: Left to right: bump mapping, parallax mapping, parallax occlusion mapping

Unlike the other techniques, parallax occlusion mapping is based on per-pixel ray tracing. Rays are traced from the viewpoint into the object. A height map (as with voxel terrain) is used to describe the terrain and mapped onto the surface. As the height map texture is on a surface that can have an arbitrary orientation, the ray is first transformed into texture (tangent) space; this allows comparison of the ray position with the height map.

As the ray is traced, its height is compared to the height map value. When the ray has ended up underneath the height map, tracing is stopped and the ray's position in the texture plane is used to index the diffuse texture for the surface: it is displayed as warped and the illusion of depth is created. Usually, the height map is used to create the illusion of the surface being pushed down, as shown in Figure 3. This allows the technique to be used on the original object, with no additional geometry attached specifically for the effect. Additionally, note that the technique doesn't generate proper silhouettes.

Parallax occlusion mapping was chosen as the basis for the GPU terrain implementation. Its height map-based algorithm, use in current-generation video games (i.e. real-time applications), and promising visual quality suggested it to be a good starting point.

3. Implementation

This section describes the process of creating the sample application used to investigate GPU based terrain rendering. Section 3.1 lays out the general approach, 3.2 goes into the creation of a fixed function style shader for testing, and 3.3 describes the terrain ray tracing shader. Sections 3.4 to 3.8 detail various extra functionality that was added. Finally, 3.9 goes into operation of the demo application. Figure 4 and the title page show screen captures of the final result. The final terrain pixel shader is included in appendix A.

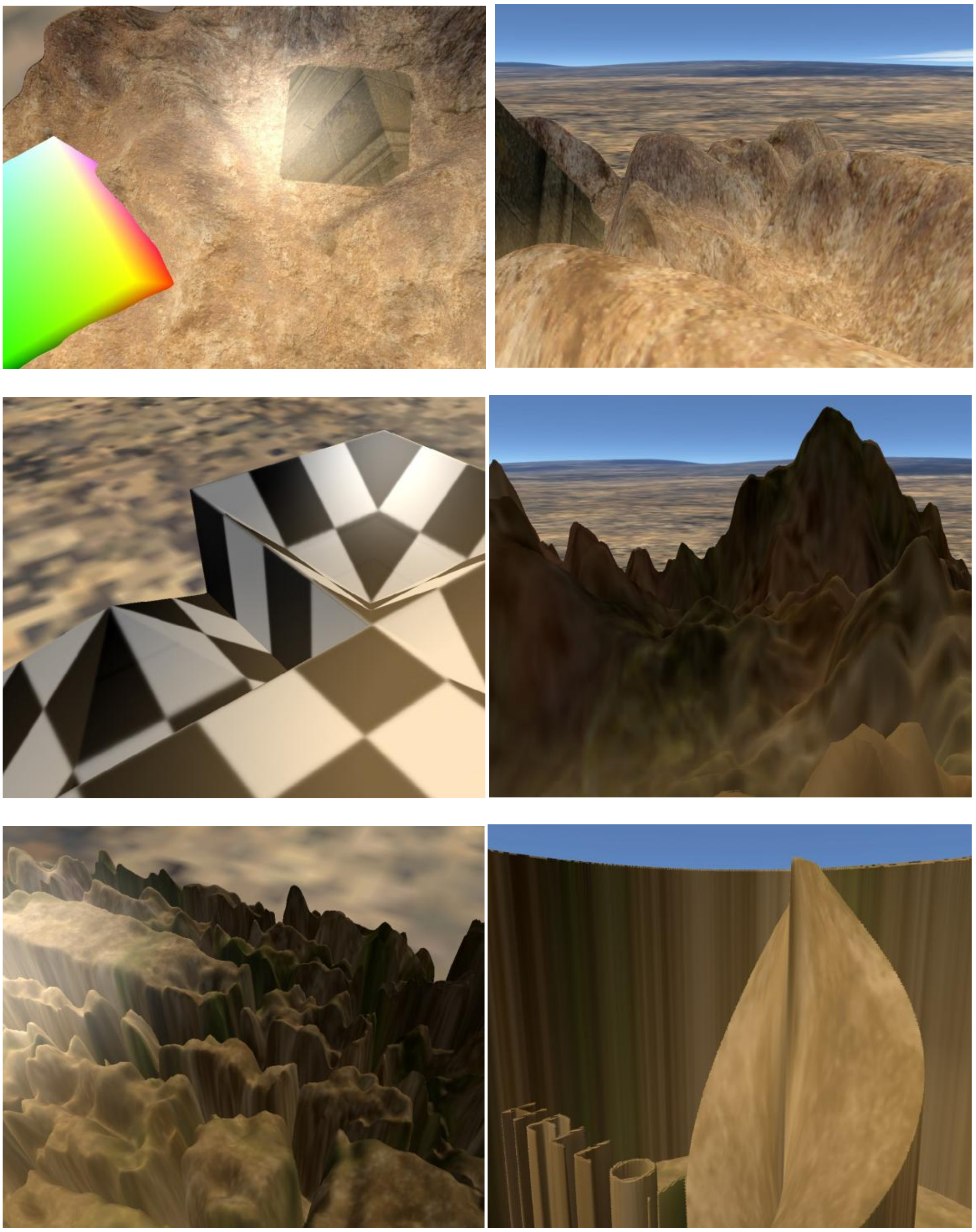


Figure 4: Final terrain

3.1 Approach

The implementation of the terrain ray tracing was done completely from scratch. The application was created using C++ and Direct3D 10. D3D10 was chosen as it is centered around using the programmable (shader) pipeline as opposed to being designed mostly for fixed function work. Additionally, it was chosen because using it would be educational to the author, who only had prior immediate mode OpenGL experience, from earlier graphics courses. Education was also the reason why the application was created from scratch: doing this would give insight in how shaders are set up and in which ways they interact with CPU-side data.

Looking at the parallax occlusion mapping papers, it was deemed possible to draw the terrain by ray tracing inside a cube. All surfaces would have terrain drawn on them where needed, giving the illusion of actually being surrounded by it, similar to in a VR cave. For simplicity's sake a unit cube was chosen for this, and as such, the first goal was to get this cube set up and rendered. Note that for all discussed shaders no geometry other than this one unit cube is sent to the GPU. Although the application uses a vertex format with support for normals, color, etc. for debugging purposes, only the vertex position is ever set; none of the other parameters are used.

3.2 Fixed function style shader

Both to get the hang of working with shaders, and to test the construction of aforementioned unit cube, first a fixed function style shader was created. This shader does nothing more than apply the viewing and projection transforms (there is no world transform as the world consists of only this unit cube) and shading using the vertex coordinates as color. Figure 5 shows this shader in wireframe mode.

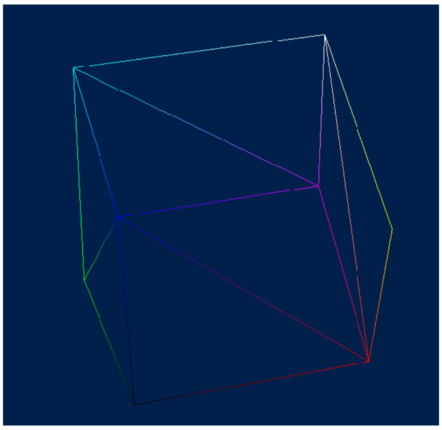


Figure 5: Vertex data shown by fixed function style shader in wireframe mode

One problem that arose at this point was the fact that rendering such as simple scene led to a frame rate of 7000+ frames per second. This actually made the application behave in a laggy fashion, perhaps due to overhead involved in processing so many frames. As such, a timer was added to restrict the frame rate to a maximum of 60 FPS.

3.3 Ray tracing

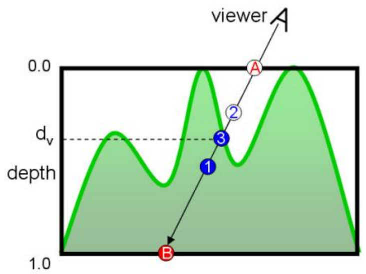


Figure 6: Height map ray tracing.

The ray tracing part of the terrain shader is based on the steps described for parallax occlusion mapping in [2]. A ray is traced from the viewer to the world space position of the pixel that is being rendered. When the ray has ended up 'under' the height map (i.e. its height is less than that assigned to the height map value at its position), its position in the height map texture is used to sample the object's (diffuse) texture. This warping of the object texture creates the illusion of relief. Figure 6 shows the procedure: tracing is done between 'A' and 'B', an intersection is found at '3'. In this 2D example, point 3's X coordinate would then be used for texture mapping.

The referenced paper called for transforming the ray to tangent space. This way, arbitrary surfaces could be parallax mapped. However, as in our case we are working with an axis-aligned unit cube, the transformation can be left out. Instead, the original, pre-transform vertex coordinates are passed to the pixel shader. These are used as the end coordinates, 'B' in figure 6.

The eye point is also passed into the vertex shader as a parameter. At first it was tried to extract the eye point from the translation part of the view transform matrix, but this proved impossible as view rotations influence this translation column. As such, the program's camera handling routines were modified to build the view matrix from a separate eye point.

The used ray coordinates match the world coordinates: X and Z are the flat plane, and Y is the height. As such, when tracing, the ray's Y coordinate is checked with the height map value, and upon intersection its X and Z values are used for the texture mapping. The pixel shader's task is to perform this tracing between the pixel-interpolated eye- and end coordinates.

The actual tracing is done in a loop, by adding a fraction of the normalized view direction vector to the eye position each iteration. This view direction vector is calculated by subtracting the eye position from the ray end position. Both have been interpolated from the vertex shader stage. The size of the fraction used each iteration determines the numbers of samples taken; in the shader it is calculated as the user-provided sample amount divided by the viewing ray length. The higher the number of samples taken, the more accurate intersections are found. This results in higher visual quality. When an intersection has been found, the tracing loop is aborted and the found intersection is used for texture mapping, as discussed.

Multiple constructions were tried for the actual tracing loop. The first attempt was a while-loop which aborted when the ray went outside of the unit cube. A disadvantage of this was that six checks had to be made (one for each box face), which was found to be relatively slow. The second attempt was to have a while-loop abort when the length of the traced ray was equal to or larger than that of the non-normalized viewing direction vector. This was fraught with glitches as inaccuracies meant tracing was often ended too late.

Finally, a simple solution was found: a for-loop which is executed once for each sample taken. This combines good accuracy with a simple, single check to determine whether the ray has been fully traced. Additionally, an advantage of the for-loop construction is that the tracing is bound to be stopped. While-loop based methods with break constructions led to GPU restarts when mistakes were made which resulted in infinite loops.

Although standard parallax occlusion mapping does not create silhouettes, this technique does. If the ray has been completely traced without finding an intersection, this means the viewer was looking over, not at, the terrain for that pixel. The current pixel is discarded; discarding all non-intersection pixels creates the terrain's silhouette. This is possible as the tracing is done inside a 3D cube: intersections will always be found for the floor, but not for the walls and roof.

As the terrain is contained in an unit cube, by default its (maximum) height and X/Z side sizes are the same. To be able to create lower terrain while still keeping the same height resolution, a height scale factor was added. All height map reads are scaled by this factor, resulting in decreased terrain height. Additionally, if the eye point is above the maximum terrain height (which can now be less than the height of the cube), the ray start is moved to that maximum height as to not waste time sampling where no terrain can ever be. Furthermore, the moved ray start point is verified to be still in the cube; if not, the ray is discarded.

Finally, the number of repeats of the diffuse texture can be set. However, a single high-resolution texture was found to look more pleasing, especially from high up where texture repeats become very apparent.

3.4 Sampling improvements

Two improvements were made to obtain a better quality versus performance tradeoff. Firstly, dynamic sampling rate selection was added. Steep rays, i.e. looking down on the terrain, do not need as high a sample rate to find correct intersection as ones grazing the terrain do (for grazing rays there is a chance of, for example, going 'through' a hill). A method to select a suitable sampling rate between a set minimum and maximum is given in [3]: linear interpolation between the minimum and maximum rates is used, using the dot product of the normal and viewing direction as the ratio. As in our case the algorithm is always applied to a cube, no normals are needed (or even set) and one minus the viewing direction Y coordinate is used as the ratio instead.

The second sampling improvement was the addition of a binary search step. After the linear search as described in 3.3 has found an intersection, a set number of binary search steps are taken between this intersection and the previous ray position, which lies outside the height volume. This way, the effect of shooting too 'deep' into the terrain is alleviated with very little performance cost. Figure 7 shows a steep section of terrain with binary search disabled and enabled (eight iterations).

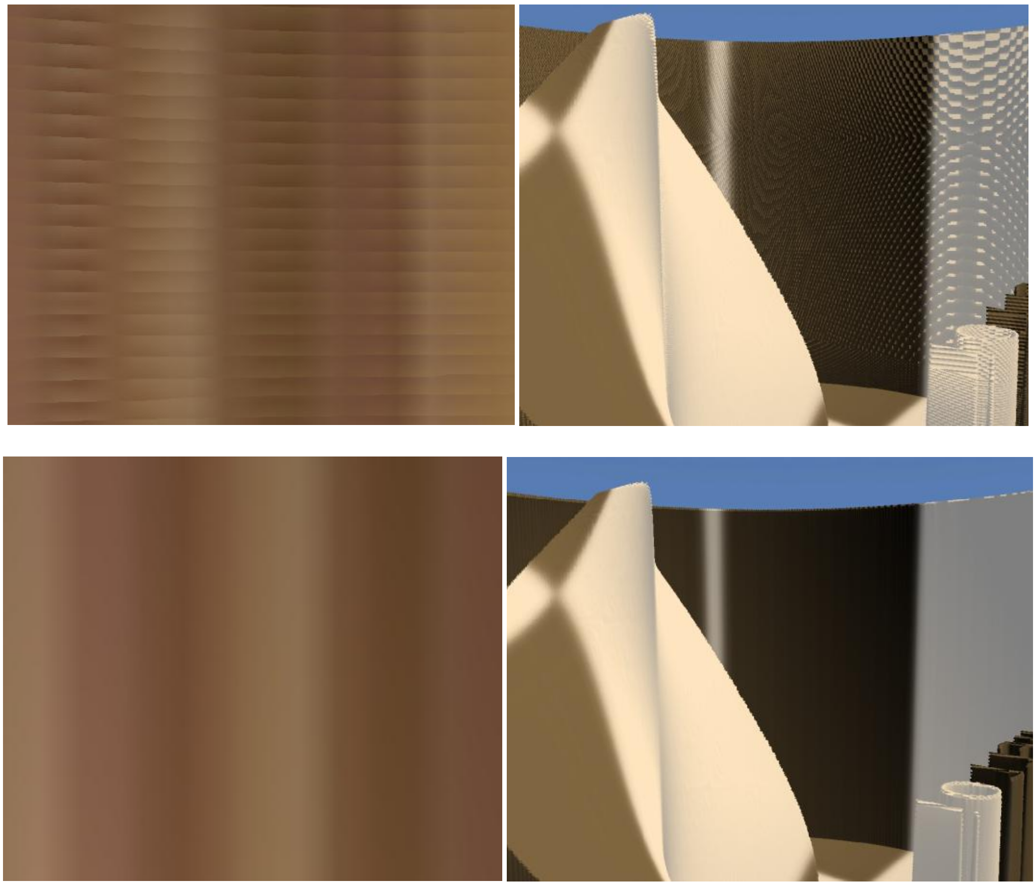


Figure 7: Binary search step disabled and enabled.

3.5 Skybox

As the terrain has proper silhouettes, non-intersection pixels are by default filled with the standard viewport color. To create a more attractive scene, a skybox was added. No extra vertex data was used for this: in an additional pass, the unit cube is resized (expanded in the XZ plane and decreased in height) and positioned around the terrain cube. A cube map texture depicting sky and terrain is applied; cubemap texture coordinates are simply generated in the box's vertex shader. The skybox is clearly visible in the various screenshots in this report.

3.6 Deformations

Height maps can easily be modified, especially compared to mesh data. As such, adding deformations such as craters to the rendered landscape is rather simple. Figure 8 shows the terrain deformed with some craters.

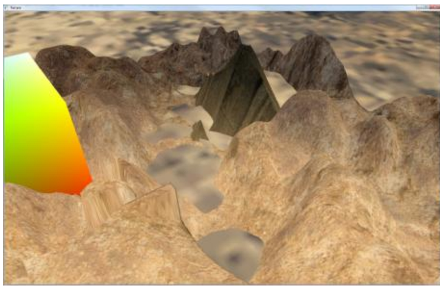


Figure 8: Deformed terrain

Note that in the demo program, deformations are created by copying a crater heightmap to a second texture, which is sampled together with the height map. As such, two texture reads are needed, but this was significantly easier to implement. Additionally, it allows different resolutions for the deformation- and height map textures and it allows the deformations to be turned off.

3.7 Occlusion

For the terrain to actually be useful in for example video games, it must be able to interact with standard polygonal objects. One interaction is collision detection, which is not covered by the demo application. Occlusion between the terrain and polygonal objects, however, is. The brightly colored cube shown in the various screen captures is a polygonal object: it is the unit cube, resized and reoriented, in an extra rendering pass. Occlusion with the terrain is accomplished by having the terrain pixel shader output a depth value in conjunction with its color. The depth value is calculated by transforming the view direction ray, which the ray tracing steps along, from world to projection space and taking its Z coordinate divided by its W coordinate.

3.8 Lighting

Applying lighting to the terrain requires normals, which are obviously different from those of the cube it is rendered on. The terrain normals can either be stored in a normal map texture, as used in bump mapping, or calculated on the fly by obtaining the height map gradient, for example using a Sobel filter [3]. Normal- and height map data can be combined together in one RGBA texture and quicky looked up. On the other hand, calculating the normals on the fly is more broadly applicable: no preprocessing is needed, and the height map can be generated or deformed in real-time. In the demo application the on the fly approach is used. A Sobel filter is used on the height map texture to calculate its derivatives. From these the normal is easily constructed. The Sobel filter code was found online [4].

Once the normal for each rendered pixel is known, standard diffuse and specular lighting are easily applied. The result of this is shown in Figure 4.

3.9 The application

Starting the demo application is simply a matter of running its executable. Apart from the viewpoint window, it also launches a console window which shows the various controls the viewport accepts. Movement is standard W/A/S/D keys, looking around can be done by holding down the right mouse button and dragging. Pressing the 'C' key adds deformations to the landscape. Features such as binary search and lighting can be toggled on and off, see the console output for more information.

The program's shaders are spread over multiple .fx files; these files define the shaders and passes used by Direct3D*.* *Header.fxh* contains definitions of the various structures, samplers and states. *Terrain3.fx* is the main effect file that is executed by the program: it describes the passes and contains the ray tracing, lighting and binary search code. In turn it includes three other files. *Skybox.fx* contains the vertex and pixel shaders which transform the unit cube into a textured skybox. *Smallbox.fx* transforms the unit cube into the small, colored cube used to test occlusion. Finally, *Sobel.fx* holds the code which calculates normal data from the height map.

4. Observations and results

While developing and running the demo application, a number of observations were made.

4.1 Ease of use, flexibility

Although creating the demo application was quite time consuming due to lack of understanding and, even more so, lack of experience, for a seasoned graphics programmer the technique should be straightforward to implement. Creating content for the technique is not very difficult: only a height map texture is needed, and polygonal objects work together with the terrain without problems.

The terrain is simple to deform, and standard lighting methods work with it. One possible disadvantage of using this type of terrain is that it locks one into using shaders: OpenGL style texture/lighting calls will not influence it. But then, modern graphics programming is already largely shader driven, as evidenced by DirectX 10's lack of fixed function support.

In short, this type of terrain is reasonably simple to implement, while not subjecting the rest of the application it is used in to any significant limitations.

4.2 Quality and performance

The visual quality of the terrain is determined by two main factors: ray sample rate and height map resolution. Higher ray sample rates result in more precise intersections, and prevent 'floating' segments as shown in Figure 9, where no intersection was found for some points. Unfortunately, increasing the sampling rate linearly increases the amount of work that needs to be done. 512 samples per ray was found to give reasonable visual quality, depending on the terrain steepness. To match the crispness of polygonal terrain, extremely high numbers of samples, such as 2000+, had to be used.

Height map resolution has the simple effect of determining how rounded the terrain looks. The height map is bilinear filtered when it is sampled, and as such it does not look grainy (Figure 10 shows the scene with filtering off). However, this is obviously no substitute for higher resolution textures, and close-up hills can be seen to consist of discrete slopes (bumpiness in Figure 11). Higher resolution height maps are of course larger in memory, and slower to sample.

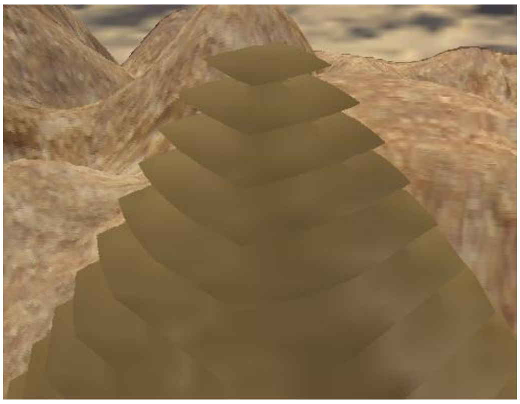


Figure 9: Sampling artifacts

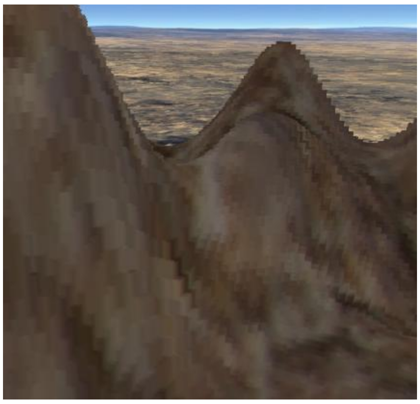


Figure 10: Terrain with no bilinear filtering

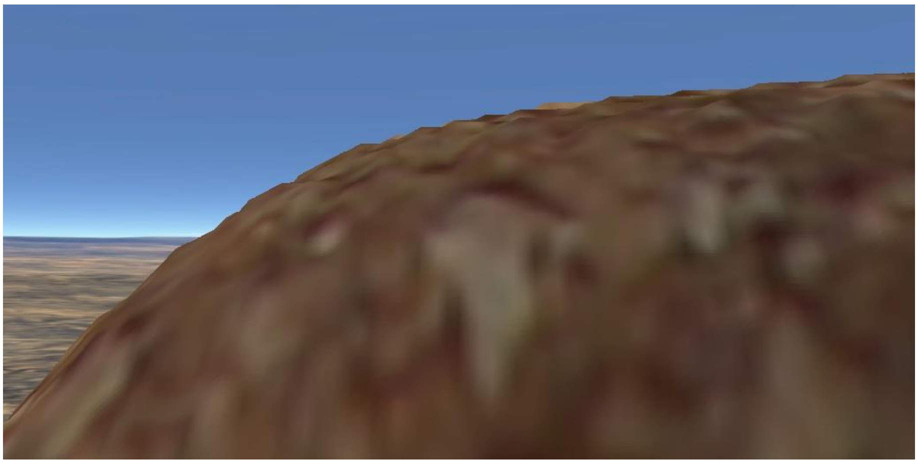


Figure 11: Bilinear filtering does not fully hide low height map resolutions

Performance was rather disappointing, especially considering the imperfect visual quality. Although frame rates of up to 300 frames per second were achieved on a Geforce GTX 275 GPU at 1440x900 pixels, performance fluctuated. With a dynamically selected sampling rate between 128 and 512 samples, frame rates dipped into the 40s when the full ray had to be traced (i.e. no intersection found). On a more modest mobile Geforce 8400, real time performance could only be reached at low (320x240) resolutions. Table 1 shows the performance impact of various features discussed in section 3. Predictably, the largest performance impacts are made by operations that are done for each ray step. Frame rates were recorded with the 'FRAPS' program.

Feature	FPS	Notes
Bilinear search	~-1	
Deformations	-~40%	Slow because second texture lookup used
Lighting	0	
Occlusion	0	
Dynamic sample rate	+0 to +300%	Dependent on viewing angle and min/max rate

Table 1: performance gain/penalty of various features on Geforce GTX 275. Frame rates are relative to a constant 50 frames per second attained when viewing a sparse hill landscape.

5. Conclusions and improvements

5.1 Performance

Terrain drawn in a pixel shader has a high performance overhead: real-time ray tracing has to be executed for every single viewport pixel. On the other hand, unlike what is the case with polygonal terrain, the terrain complexity does not influence performance much, apart from perhaps requiring a higher resolution height map and more precise tracing for small steep objects. The demo application's speed left to be desired: even on high-end hardware frame rates were inconsistent, and this was without other things such as AI, collisions detection, etc. going on. The lack of performance might partly be due to the author's lack of experience in optimizing pixel shaders. Additionally, more advanced techniques exist with which to gain speed:

- Safe-zone techniques. These techniques preprocess the height map data to find safe ray step sizes for each pixel: instead of a fixed step size, the maximum safe step without missing an intersection is used. This data is saved into unused channels of the height map texture. Various similar techniques exist, such as cone step mapping [5] and dilation/erosion based empty space skipping [6].
- Piecewise curve approximation [3] [7]. This technique works by finding the two points between which the intersection lies, and then solving for the precise point by approximating the height map as a piecewise linear curve. This technique results in higher visual quality at lower sample rates than the linear/binary search used in the demo.
- Level of detail. [3] discusses using the mip map level of the surface to fall back to standard bump mapping, which is must faster than parallax occlusion mapping. Depending on the distance from the viewer, linear interpolation between the two techniques is used.
- Visibility sets. If high-resolution height maps are required, and sampling or storage costs become prohibitive, the terrain could be split up into multiple cells, which are rendered depending on pre-calculated potential visibility sets.

5.2 Quality

Performance increasing techniques might allow the ray tracing sample rate to be increased, leading to an instant visual quality increase. Furthermore, self-shadowing should be possible by tracing from light sources [2]. Shadows from polygonal objects on the terrain could be rendered using the stencil buffer [8].

5.3 Reflection

Personally, I feel this assignment was very educational. It has acquainted me with Direct3D, shader integration and writing, and working with ray tracing. As such, it covered multiple subjects discussed in the lectures, and it was interesting to apply them in practice. The assignment was more time consuming than anticipated: lack of experience, difficulty understanding the subject material, and difficulty debugging the shaders were mostly to blame. Having been able to pick the subject myself did make it worthwhile in the end though, as I was working on something I found interesting.

6. References

1. **Pajer, Stephan.** *Modern Texture Mapping in Computer Graphics.* 2006.

2. **Policarpo, Fabio, Oliveira, Manuel M. and Comba, João.** *Real-Time Relief Mapping on Aribtrary Polygonal Surfaces.* s.l. : ACM, 2005.

3. **Tatarchuk, Natalya.** *Dynamic Parallax Occlusion Mapping with Approximate Soft Shadows.* s.l. : ACM, 2006.

4. **Zima, Catalin.** Converting Displacement Maps into Normal Maps in HLSL. [Online] [Cited: 5 2009, 14.]
http://catalinzima.spaces.live.com/?_c11_BlogPart_BlogPart=blogview&_c=BlogPart&partqs=cat%3DHLSL.

5. **Dummer, Jonathan.** *Cone Step Mapping: An Iterative Ray-Heightfield Intersection Algorithm.* 2006.

6. **Kolb, Andreas and Rezk-Salama, Christof.** *Efficient Empty Space Skipping for Per-Pixel Displacement Mapping.* 2005.

7. **Feng, Huamin Qu, et al.** *Ray Tracing Height Fields.* 2003.

8. **Microsoft Corp.** Raycastterrain Sample. *MSDN.* [Online] [Cited: 5 14, 2009.] http://msdn.microsoft.com/en-us/library/cc835725(VS.85).aspx.

9. **Dachsbacher, Carsten and Tatarchuk, Natalya.** *Prism Parallax Occlusion Mapping with Accurate Silhouette Generation.* 2007.

A. Pixel shader

Shown below is the pixel shader code which executes the ray-tracing algorithm and applies lighting. For brevity's sake, routines such as binary search and normal calculation have been omitted.

```
struct PS_INPUT
{
   float4 pos : SV_POSITION;
   float4 color: COLOR;
   float3 normal: NORMAL;
   float3 tex : TEXCOORD0;
   float3 eye: TEXCOORD2;
   float3 origPos: TEXCOORD3;
   float3 lightPos: TEXCOORD4;
};

struct PS_OUTPUT
{
   float4 color : SV_TARGET;
   float  depth : SV_DEPTH;
};

PS_OUTPUT PS( PS_INPUT input)
{
   PS_OUTPUT o = (PS_OUTPUT)0;
   float3 rayStart=input.eye;
   float3 rayDir = input.origPos-rayStart;

   if(TERRAIN)
   {
      //If ray starts outside of maximum terrain height, move it to the max height
      if(rayStart.y>HEIGHT_SCALE)
      {
         rayStart+=rayDir*((rayStart.y-HEIGHT_SCALE)/-rayDir.y);

         //If moved ray is outside bounding box, drop it
         if( rayStart.x<0 || rayStart.x>1 || rayStart.z<0 || rayStart.z>1)
         {
            discard;
         }
         rayDir = input.origPos-rayStart;
      }

      float rayLength=length(rayDir);
      rayDir/=rayLength;


      //Calculate step size by using formula to increase sampling rate along steep angles
      //  int nNumSteps = (int) lerp( g_nMaxSamples, g_nMinSamples, dot( vViewWS, vNormalWS )
); from Tatarchuk-POM
      int numSteps;
      if(DYNAMIC_SAMPLE)
         numSteps = (int) lerp(MIN_SAMPLES,MAX_SAMPLES,1-abs(rayDir.y));
      else
         numSteps = MAX_SAMPLES;


      float stepSize=rayLength/numSteps;
      float3 step = rayDir*stepSize;
      float3 rayPos=rayStart;

      float height;
      //Actual tracing
      for(int i=0;i<numSteps;i++)
      {
         //Get terrain height; stepsize is added so height zero terrain is still drawn.
```

```
        height  =getHeight(rayPos.xz).r;
        if(height+stepSize>rayPos.y)
           break;
        rayPos+=step;
     }

     //Check result of tracing
     if(i>=numSteps) //More samples were tried to be taken than should be possible
     {
        discard;
     }
     else //Succesful intersection
     {
        if(BINARY_SEARCH) //If binary searching is enabled, find a more precise intersection
        {
           rayPos.xz = binarySearch(rayPos,step);
        }

        //Lighting
        o.color =
ambientIntensity*textureDiffuse.SampleLevel(samLinear,rayPos.xz*TEXTURE_REPEATS,0);

        if(LIGHTING)
        {
           float3 lightDir=normalize(lightPos-input.origPos);
           float3 normal = ComputeNormalsPS(textureHeightMap,rayPos.xz*TEXTURE_REPEATS);
           float3 halfway = normalize(lightDir-rayDir);
           o.color += diffuseIntensity*diffuseColor*saturate(dot(lightDir,normal))   +
specularIntensity*specularColor*pow(saturate(dot(lightDir,halfway)),specularExponent);
        }

     //Assign depth to intersection point by transforming the ray to view*projection space
     if(OCCLUSION)
     {
        float4 d = mul(float4(rayPos,1),View);
        d=mul(d,Projection);
        o.depth = d.z/d.w;
     }
   }

   return o;
}
```