Hardware Accelerated Per-Pixel Displacement Mapping

Article · May 2004

Source: CiteSeer

CITATIONS
60

READS
712

4 authors, including:

Johannes Hirche
Luleå University of Technology
18 PUBLICATIONS **231** CITATIONS
SEE PROFILE

Stefan Guthe
Technische Universität Darmstadt
32 PUBLICATIONS **1,000** CITATIONS
SEE PROFILE

Michael Doggett
Lund University
48 PUBLICATIONS **485** CITATIONS
SEE PROFILE

Hardware Accelerated Per-Pixel Displacement Mapping

Johannes Hirche
WSI/GRIS
University of Tübingen, Germany

Alexander Ehlert
WSI/GRIS
University of Tübingen, Germany

Stefan Guthe
WSI/GRIS
University of Tübingen, Germany

Michael Doggett
ATI Research

Abstract

In this paper we present an algorithm capable of rendering a displacement mapped triangle mesh interactively on latest GPUs. The algorithm uses only pixel shaders and does not rely on adaptively adding geometry. All sampling of the displacement map takes place in the pixel shader and bi- or trilinear filtering can be applied to it, and at the same time as the calculations are done per pixel in the shader, the algorithm has automatic level of detail control. The triangles of the base mesh are extruded along the respective normal directions and then the resulting prisms are rendered by casting rays inside and intersecting them with the displaced surface. Two different implementations are discussed in detail.

1 Introduction

Displacement mapping adds real surface detail to objects in three dimensional scenes by using two dimensional maps containing height data. Displacement mapping can be used for generating large scale objects such as terrain and for adding smaller scale detail such as bumps. Displacement mapping is used in offline cinematic content creation packages to add this surface detail and as the capabilities of graphics hardware increases it can also be used in real time applications.

Displacement mapping is performed by displacing the position of a surface along the normal to the surface by a distance sampled from a map of scalar values associated with the surface. The displacement can be applied to vertices that make up the mesh of the base surface with displacement values associated with each vertex. Alternatively many algorithms insert new vertices into the surface to increase the base mesh detail either using a fixed tessellation factor or by inserting vertices adaptively based on the detail in the displacement map. The displacement map is typically a two dimensional area used in a similar manner to a texture map with the base mesh containing coordinates that indicate where the displacement map is to be sampled.

Displacement mapping was first mentioned by Cook [1] in the context of software based rendering. Techniques for ray tracing displacement maps have been presented previously by Heidrich et al.[7] and Pharr et al.[12]. But are still more complex than can be implemented on the current generation of programmable graphics hardware.

In recent years several proposals for dedicated hardware have been proposed for displacement mapping. Doggett et al [3, 4] presented a level of detail driven rasterization approach that inserts new vertices into the base mesh. A similar technique is presented by Gumhold et al[6]. Doggett and Hirche[2] presented an adaptive tessellation scheme that inserts new vertices dependent on the average displacement within the displacement map and the variance of the surface. Moule and McCool[11] improved upon the area coverage for detecting change in the displacement map to drive a similar adaptive scheme. Hirche and Ehlert[8] use a precomputed decision to drive tessellation eliminating the need for computing tessellation decisions. All of these approaches require the creation of new vertices in the vertex shader stage of existing graphics hardware, a feature which has only recently been exposed in a limited fashion through the concept of output from the vertex shader being sent to vertex arrays. This requires that the target vertex arrays are sized correctly for a CPU calculated number of vertices.

Recently introduced hardware by Matrox[10] allows fetching from a displacement map within the vertex shader, a feature not available on other hardware. But like DirectX 9 displacement mapping tessellation is done at a pre set tessellation level which is not controlable from within the vertex shader.

Using a similar approach to that presented in this paper, Kautz et al.[9] extrudes the base mesh to enclose the entire displaced surface and then composites together slices through the extruded volume using a technique similar to volume rendering. Since all slices through the volume are rendered whether visible or not this technique requires high fill rates and a high texture bandwidth.

Wang et al [14] presented a novel approach to rendering displacement maps by using a precomputed table of visibility information. The surface is extruded along the

surface normal direction and following that, viewing rays are cast from the extruded surface and an intersection test with the displacement map is performed. Of course the number of viewing rays that can be stored is limited and also the curvature of the base domain surface is only approximated.

This paper presents an approach to displacement mapping using currently available programmable graphics hardware that creates the appearance of a displaced surface on a per pixel basis. Unlike previous techniques it doesn't require any insertion of vertices to retessellate the mesh. Displacement map sampling occurs in the pixel shader so all texture filtering modes can be applied. Many of the schemes above include level of detail control tied to triangle size computed in screen space, this step is not required using this technique since the sampling of the displacement map is relative to the number of pixels contained within the bounding prisms of each displaced triangle.

This paper is broken into several sections. Section 2 describes the basic algorithm and is followed by a more detailed description in Section 3. Section 4 explains how the algorithm is improved by using tetrahedrons as the base object and how they are rendered. Section 5 presents results and section 6 concludes the paper.

2 General Idea

Most approaches to displacement mapping require that the geometry of a given base mesh can be modified, especially in the sense of adding more detail in the form of retessellated triangles. Currently available hardware, at which this algorithm is targeted, does not allow vertices to be added once the geometry has been transferred to the graphics card. To work with this restriction this algorithm does not generate geometry on the card, but instead creates triangles that cover the area on the screen that could be affected by the displaced base mesh triangle. When the covering triangles are rasterized a per pixel calculation is performed to detect an intersection with the displaced surface. If an intersection is detected the resulting fragment has to be shaded and written to the framebuffer with its correct z-value similar to the shading used by Gumhold [5]. The number of covered triangles should be kept to a minimum to reduce the geometry transfer overhead. The bounding volume of the surface with a displacement map applied to it is given by a prism obtained by displacing the base triangle along the vertex normals to the maximum displacement height. At each pixel of the prism's triangles a non-trivial intersection of the ray with the prism has to be performed, placing a very high burden on the pixel shader pipeline. Since backfacing triangles can be culled, the amount of used pixels to be drawn is relatively limited. The resulting triangles are shown in Figure 1. The sides of the prism are quads and have to be split into two triangles, the bottom and top of the prism remain unchanged resulting in eight triangles to be rendered per base triangle.

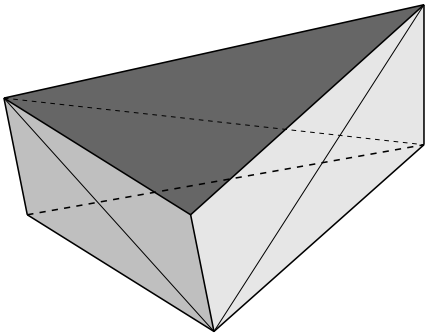

Figure 1: The prism with its resulting triangles used for rendering.

3 Single pass prism Renderer

The first approach renders the displacement prism by splitting it up into eight triangles and casting rays into the prism from every rendered pixel. However, we have to assume that the faces of the prism are flat. Otherwise calculating the intersection with the bi-linear faces is already too complex for the fragment shader. Additionally, the transformation from world coordinates to texture coordinates is not solvable.

The rays are cast in the viewing direction from each pixel position. To find out whether the ray intersects the displacement map, the height of the sampling position is compared to the height of the displacement map at the interpolated texture coordinate of the sampling position. The height ranges from zero at the base mesh level to one on the top of the prism. The 3D texture coordinates need to be interpolated inside the prism, along the viewing direction. The texture coordinates are local to the prism and a base transform has to be made at all vertex positions to obtain the viewing direction in local texture space. Given a triangle with vertices $V_i = (x_i, y_i, z_i)$ with normals N_i and texture coordinates $U_i = (u_i, v_i)$ for i = 1,2,3, the first step is to add a third coordinate defined by the height of the vertex in the prism:
$U_i' := (u_i, v_i, 0)$ for vertices of the base triangle and $U_i' := (u_i, v_i, 1)$ for vertices of the displaced triangle. To calculate the transformation for vertex V_1 for example, on the base triangle, we define a local base B_{Texture} with the texture directions e_1, e_2 along the triangle edges:

$$e_1 := U_2' - U_1'$$
$$e_2 := U_3' - U_1'$$

$$B_{\text{Texture}} := (e_1, e_2, 1)$$

In the same manner we define a local base B_{World} with the world coordinates of the vertices:

$$f_1 := V_2 - V_1$$
$$f_2 := V_3 - V_1$$
$$B_{\text{World}} := (f_1, f_2, N_1)$$

The basis transformation from B_{World} to B_{Texture} can be used to move the viewing direction at the vertex position V_1 to local texture space.

To avoid sampling outside of the prism, the exit point of the viewing ray has to be determined. In texture space the edges of the prism are not straightforward to detect and a 2D intersection calculation has to be performed. This can be overcome by defining a second local coordinates system which has its axes aligned with the prism edges. For this we assign 3D coordinates to the vertices as shown in Figure 2. The respective name for the new coordinate for a vertex V_i is O_i. Then the the viewing direction can be transformed in exactly the same manner to the local coordinate system defined by the edges between the O_i vectors:

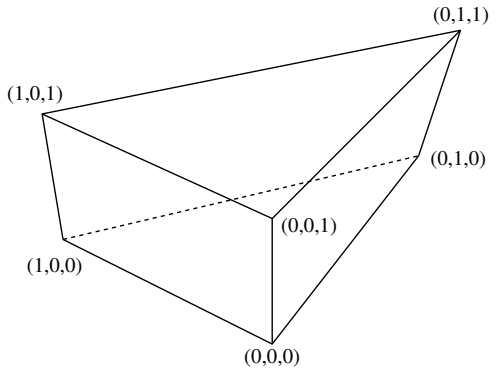

Figure 2: The vectors used to define the second local coordinate system for simpler calculation of the ray exit point.

$$g_1 := O_2 - O_1$$
$$g_2 := O_3 - O_1$$
$$B_{\text{Local}} := (g_1, g_2, 1).$$

Again this is the example for the vertex V_1. In the following the local viewing direction in texture space is called $View_T$, and in the B_{Local} base representation $View_L$. We assume that the viewing direction changes linearly over the face of a prism triangle and put the local viewing direction in both coordinate systems in 3D texture coordinates and use them as input to the fragment shader pipeline in order to get linearly interpolated local viewing directions. The interpolated $View_L$ allows us to very easily calculate the distance to the backside of the prism from the given pixel position as it is either the difference of the vector coordinates to 0 or 1 depending which side of the prism we are rendering. With this Euclidean distance we can define the sampling distance in a sensible way which is important as the number of samples that can be read in one pass is limited, and samples should be evenly distributed over the distance. An example of this algorithm is shown in figure 3. In this case four samples are taken inside the prism. The height of the displacement map is also drawn for the vertical slice hit by the viewing ray. The height of the third sample which is equal to the third coordinate of its texture coordinate as explained earlier, is less than the displacement map value and thus a hit with the displaced surface is detected. To improve the accuracy of the intersection calculation, the sampled heights of the two consecutive points with the intersection inbetween them, are substracted from the interpolated heights of the viewing ray. Because of the intersection the sign of the two differences must differ and the zero-crossing of the linear connection can be calculated. If the displacement map is roughly linear between the two sample points, the new intersection at the zero-crossing is closer to the real intersection of the viewing ray and the displaced surface than the two sampled positions.

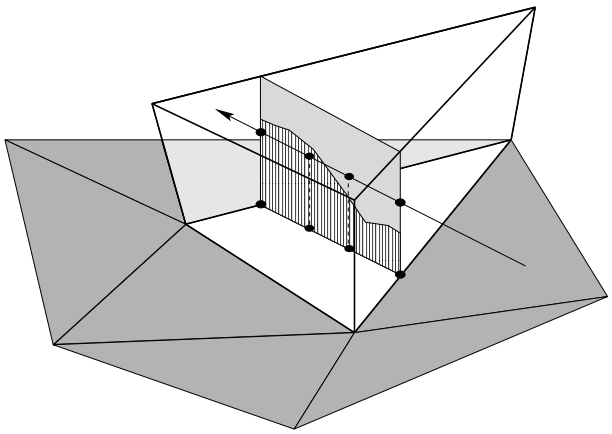

Figure 3: Sampling within the extruded prism with a slice of the displacement map shown.

Although the pixel position on the displaced surface is now calculated, the normal at this position is still the interpolated normal of the base mesh triangle. It has to be perturbed for correct shading, in this case standard bump mapping using a precalculated bump map derived from the used displacement map is used. The bump map is

obtained by convoluting the displacement map with a Sobel Operator in both horizontal and vertical direction and stored together with the displacement map in one single texture, with the displacement value stored in the alpha channel.

The algorithm was implemented using OpenGL vertex and fragment programs and run on ATI RADEON 9700 and nVidia GeForceFX class cards. The performance was similar on both cards. The implementation also showed the limitations of the fragment shader accuracy since the intersection of front- and back-face is calculated in a different way.

The assumption that the prism faces are flat is a very strong restriction that makes the algorithm in this form generally unusable. In case the faces are not flat, a viewing ray may intersect the same face it originates from which will cause holes when rendering. Especially the mapping between the different coordinate systems is curved in this case and cannot be approximated with an affine transformation as we assumed so far. Because of this a more robust algorithm was implemented that does not have these limitations.

4 Tetrahedral Renderer

To avoid the issues with non-planar prism faces, the prism has to be split into simpler shapes that can be handled more easily. Obviuosly it can be split into three tetrahedrons as shown in Figure 4. The main difference in using tetrahedrons instead of the prism is that the texture space coordinates of the entry and exit point can be interpolated at the same time by the rasterization units. The sampling points between the entry and exit point can then be obtained by just linearly interpolating in between them. The tetrahedrons can be rendered using an adaption of the Projected Tetrahedra (PT) Algorithm by Shirley and Tuchman[13].

4.1 Mesh Construction

Using tetrahedrons requires the construction of a tetrahedral mesh from the given base domain surface. It has to be ensured that neighboring tetrahedral edges are aligned in a consistent way to avoid aliasing effects between adjacent triangles. This can be achieved without knowledge of the connectivity in the tetrahedral mesh, by just setting up an enumeration of the vertices in the mesh that allows for an index comparison. The enumeration can be obtained from the vertex indices as they are usually given in an array.

The algorithm iterates over all faces in the triangle mesh folding up a prism by displacing every vertex of the base triangle along the vertex normal direction. To globally adjust the amount of displacement, the normal can be multiplied with a user defined scalar. Every prism is then split into three tetrahedrons following the ordering scheme as schematically shown in Figure 4. The indices v_0, v_1, v_2 are assigned to the lower vertices and v_3, v_4, v_5 to the upper base vertices. Now every prism is tiled into the three tetrahedrons $T(v_0, v_1, v_2, v_5), T(v_0, v_1, v_4, v_5)$ and $T(v_0, v_3, v_4, v_5)$. An additional requirement is that $v_0 < v_1 < v_2$ with respect to the consistent numbering scheme of the mesh as noted before. Hence the algorithm simply works this way:

```
FOR_EVERY_TRIANGLE_FACE(f)

  IF(v0 > v1)
    SWAP(v0, v1)
    SWAP(v3, v4)

  IF(v0 > v2)
    SWAP(v0, v2)
    SWAP(v3, v5)

  IF(v1 > v2)
    SWAP(v1, v2)
    SWAP(v4, v5)

  CREATE_TETRA(v0, v1, v2, v5)
  CREATE_TETRA(v0, v1, v4, v5)
  CREATE_TETRA(v0, v3, v4, v5)
```

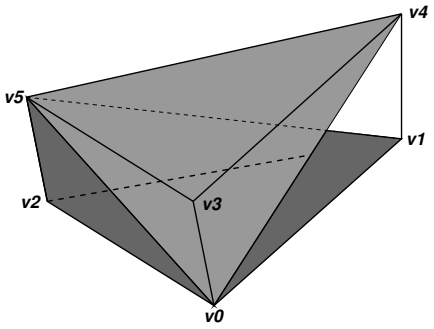

Figure 4: Subdivision of prism into three tetrahedrons (v0-v1-v2-v5, v0-v1-v4-v5, v0-v3-v4-v5)

4.2 Rendering

To adapt the PT-algorithm to displacement mapping only a few modifications have to be applied. In contrast to the standard algorithm where each vertex needs color and opacity, each vertex is attributed with its respective tangent space consisting of normal, tangent and bi-normal, each a 3d-vector. The tangent and bi-normal are necessary for performing the bump map operation while shading the surface. Additionally two texture coordinates,

one for the bump and displacement map, the other for a freely usable texture, are assigned to each vertex. Before the geometry is sent to the rendering pipeline a view-dependent preprocessing step has to be performed, where the tetrahedrons are decomposed into triangles according to the PT-algorithm. In Figure 5 the possible projections of tetrahedrons and the respective decompositions are shown. At point *S* in the diagram the connecting edge between frontside and backside of the decomposed triangles is calculated. The backside vertex is also called the secondary vertex of all the triangles.

So far all the processing has to be done on the driver side by the host computer's CPU.

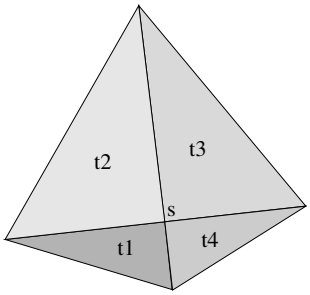

Figure 5: Possible decompositions of projected tetrahedrons into triangles.

Every triangle vertex (primary vertex) sent into the first stage is attributed with texture coordinates and tangent space vectors. Likewise the vertex on the backside (secondary vertex) of the decomposed tetrahedron is transferred as attribute including its texture coordinates and tangent space vectors. With these parameters the vertex shader computes homogeneous texture coordinates for the primary and secondary vertex. It also computes the model-view-projection transformation of the vertices and finally transforms per vertex viewing and light direction into tangent space. In the second stage of this pipeline the pixel shader performs the intersection calculation between eye vector and the displacement map. To achieve this the pixel shader performs four lookups in the displacement map given by the interpolated texture coordinates of the primary and secondary vertex and two interpolated positions in between. The intersection between eyevector and displaced surface is then calculated by substraction of the sampled displacement value from the interpolated texture coordinates. A sign change indicates the interval where the eyevector hits the displaced surface. In case no surface was hit the pixel is removed. Otherwise the pixel will undergo a final shading step. Here, bump mapping was used to perturb the interpolated normal and Phong shading using the fragment shader stage.

In order to circumvent any dynamically inserted vertices, we could also use the direct scanline conversion of tetrahedra as proposed by Weiler et al. [15]. However, the number of texture coordinates per vertex would be four times as high as for the original projected tetrahedra approach, exceeding the number of texture coordinates that can be transferred ro the pixel pipeline.

There are two possible solutions to implement a pure hardware based rendering. The first one is without introducing any dynamic vertices. For this we would need a hardware that is able to split the tetrahedra, based on its projective profile, into three or four triangles. Additionally, this hardware would have to interpolate a separate set of texture coordinates for the front and back face of the "thick" triangle.

The second solution is a programmable hardware that can not only insert additional vertices into an existing mesh, but also insert additional triangles. With this approach, we could also construct the base mesh on the GPU. This also includes the optimization of the height of the prisms if this unit has access to a texture. While the base mesh can be created once for all frames, the dynamic vertices have to be added on a pre-view basis.

The optimal solution however would require both, the programmable vertex insertion and the direct scan conversion of tetrahedra.

5 Results

The tetrahedral renderer was implemented using OpenGL vertex and fragment programs. The problems of using tetrahedrons as primitives are of course the amount of additional geometry that has to be transformed and rendered. The pixel shader needs a single rendering pass on an nVidia GeForceFX 5800 or ATI RADEON 9700. In our implementation four samples along a ray inside a tetrahedron are taken and compared with the displacement map. To avoid sampling artefacts by missing a surface completely the size of the tetrahedrons and thus the size of the used base triangle mesh has to be chosen appropriatly. Longer pixel shader programs will allow more samples to be taken improving the sampling quality. As test cases we used flat base triangle meshes and applied a half donut and the crater lake displacement map to it. Additionally we used a cylinder and a sphere shaped mesh and applied the displacement map of a laser range scan of a human head and an earth height field. It is possible to add a texture additionally to displacement mapping, even with different texture coordinates than the displacement maps, allowing for light maps, etc. Frame rates for the shown examples were clearly pixel shader limited. Our implementation is capable of rendering at 20fps

at 500x500 pixels resolution. As all rendering was done in immediate mode, there is certainly the opportunity for optimizations.

Four samples were chosen due to limited fragment shader program length. When new fragment shader pipelines become available the number of samples should be increased and the number of generated tetrahedrons reduced. As the algorithm is pixel fill rate limited a reduction of rendered tetrahedrons and thus rastered triangles would result in an increased performance.

6 Conclusion

Displacement mapping can be used to reduce the bandwidth from CPU to GPU by only requiring displacement maps to be sent to the GPU's memory to generate more complex geometry without sending large vertex arrays. This also reduces the constraints of limited GPU memory.

The pixel based algorithm presented in this paper performs at interactive rates on currently available hardware. Sampling of the displacement map is driven by visible pixels unlike most previous displacement mapping approaches that are driven by retessellation of the base mesh using various schemes.

The approach doesn't require the use of render to vertex and doesn't require any modifications to existing programmable graphics hardware. The quality can be improved by increased pixel shader lengths which would allow more samples of the displacement map to be taken, thus avoiding undersampling. The control flow in the pixel shader could also allow loops to sample incrementally along the ray until the intersection with the surface is found. This approach could be improved using many existing ray tracing techniques that improve intersection calculations with surfaces, for example octrees and space leaping.

The algorithms performance could be vastly improved if the vertex processor was capable of accessing texture memory and and adding new vertices or triangles. The number of tetrahedrons could then be adapted to the structure contained in the displacement map, adding new vertices where necessary, possibly by using curvature information as in [8]. This hybrid approach of conventional displacement map rendering by tessellation and ray casting would allow for very high quality rendering with realtime speed without any precomputation necessary.

References

[1] Robert L. Cook. Shade Trees. *Computer Graphics (Proceedings of SIGGRAPH 84)*, 18(3):223–231, July 1984. Held in Minneapolis, Minnesota.

[2] Michael Doggett and Johannes Hirche. Adaptive view dependent tessellation of displacement maps. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 59–66, August 2000.

[3] Michael Doggett and Anders Kugler. A Hardware Architecture for Displacement Mapping using Scan Conversion. Technical Report WSI–99–12, Wilhelm-Schickard-Institut für Informatik, University of Tübingen, Germany, 1999.

[4] Michael Doggett, Anders Kugler, and Wolfgang Straßer. Displacement Mapping using Scan Conversion Hardware Architectures. *Computer Graphics Forum*, 20(1):13–26, March 2001.

[5] Stefan Gumhold. Splatting illuminated ellipsoids with depth correction. In *Proceedings of 8th International Fall Workshop on Vision, Modelling and Visualization*, pages 245–252, November 2003.

[6] Stefan Gumhold and Tobias Hüttner. Multiresolution rendering with displacement mapping. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 55–66, August 1999.

[7] Wolfgang Heidrich and Hans-Peter Seidel. Ray-tracing procedural displacement shaders. In *Graphics Interface*, pages 8–16, 1998.

[8] Johannes Hirche and Alexander Ehlert. Curvature-driven sampling of displacement maps. presented at ACM SIGGRAPH 2002 as a Technical Sketch, July 2002.

[9] Jan Kautz and Hans-Peter Seidel. Hardware accelerated displacement mapping for image based rendering. In *Graphics Interface*, pages 61–70, June 2001.

[10] Matrox. http://www.matrox.com/mga/products/parhelia512/technology/disp_map.cfm.

[11] Kevin Moule and Michael D. McCool. Efficient bounded adaptive tessellation of displacement maps. In *Graphics Interface*, 2002.

[12] Matt Pharr and Pat Hanrahan. Geometry caching for ray-tracing displacement maps. In *Eurographics Workshop on Rendering*, June 1996.

[13] P. Shirley and A. Tuchmann. A polygonal approximation to direct scalar volume rendering. In *ACM Computer Graphics*, volume 24, pages 63–70, 1990.

[14] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heun-Yeung Shum. View-dependent displacement mapping. In *ACM Transactions on Graphics*, volume 22, pages 334–339. ACM, 2003.

[15] Manfred Weiler, Martin Kraus, and Thomas Ertl. Hardware-Based View-Independent Cell Projection. In *IEEE Symposium on Volume Visualization*, pages 13–22, 2002.

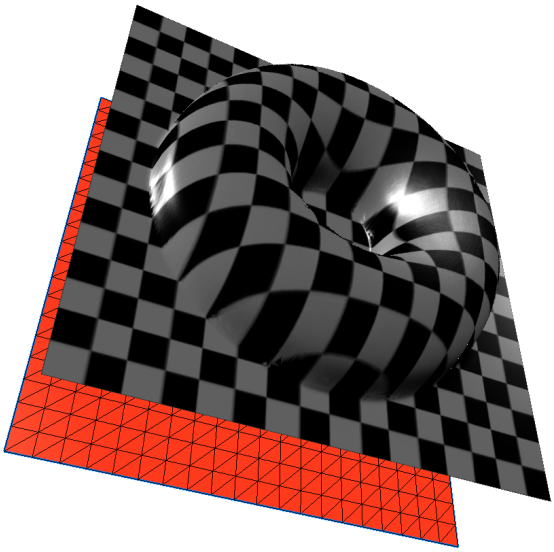

Figure 6: Flat base mesh with a half donut shape applied to it. The base mesh in red is translated away for better visibility.

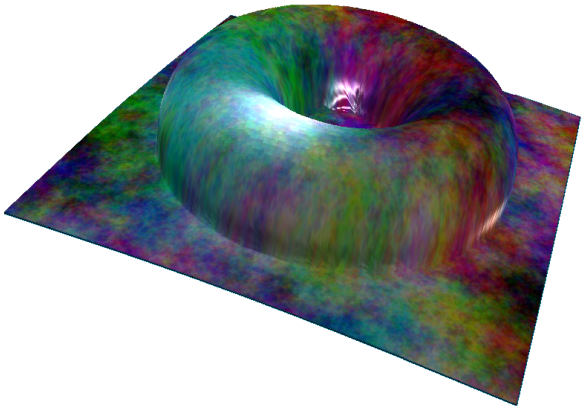

Figure 7: Same shape with a different texture applied.

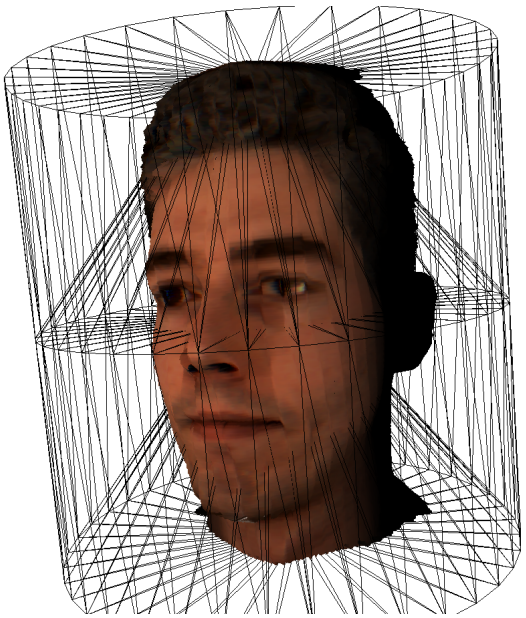

Figure 8: The head of Volker Blanz displaced from a cylindrical mesh, tetrahedral mesh show.

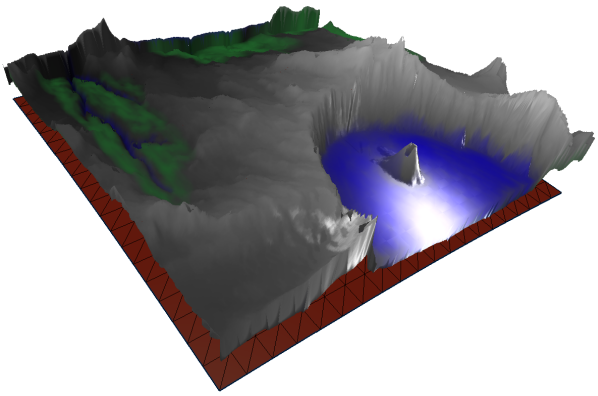

Figure 9: Displacement Map of Crater Lake applied to a flat base mesh

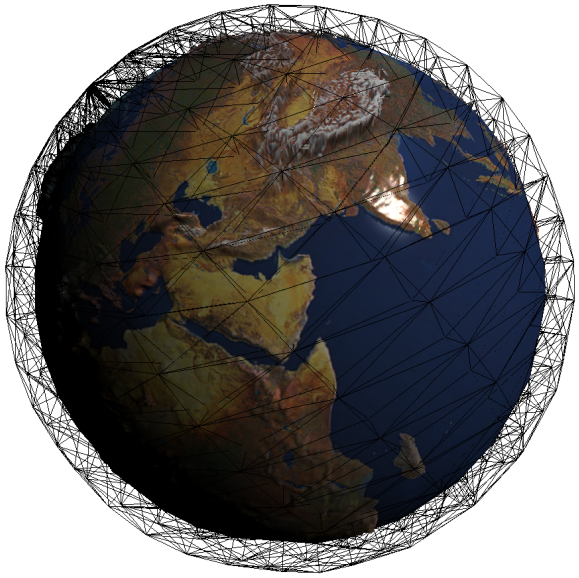

Figure 10: Sphere shaped base mesh with a earth displacement map and texture applied to it. Additionally the wire-frame of the tetrahedral mesh is shown.

Figure 11: Different angle, this time showing europe with slightly exaggerated displacements.