Hardware Accelerated Displacement Mapping for Image Based Rendering

Jan Kautz
Max-Planck-Institut für Informatik
Saarbrücken, Germany

Hans-Peter Seidel
Max-Planck-Institut für Informatik
Saarbrücken, Germany

Abstract

In this paper, we present a technique for rendering displacement mapped geometry using current graphics hardware.

Our method renders a displacement by slicing through the enclosing volume. The α-test is used to render only the appropriate parts of every slice. The slices need not to be aligned with the base surface, e.g. it is possible to do screen-space aligned slicing.

We then extend the method to be able to render the intersection between several displacement mapped polygons. This is used to render a new kind of image-based objects based on images with depth, which we call image based depth objects*.*

This technique can also directly be used to accelerate the rendering of objects using the image-based visual hull. Other warping based IBR techniques can be accelerated in a similar manner.

Key words: Displacement Mapping, Image Warping, Hardware Acceleration, Texture Mapping, Frame-Buffer Tricks, Image-Based Rendering.

Figure 1: *Furry donut (625 polygons) using displacement mapping. It was rendered at 35Hz on a PIII/800 using an NVIDIA GeForce 2 GTS.*

1 Introduction

Displacement mapping is an effective technique to add detail to a polygon-based surface model while keeping the polygon count low. For every pixel on a polygon a value is given that defines the displacement of that particular pixel along the normal direction effectively encoding a heightfield. So far, displacement mapping has mainly been used in software rendering [21, 29] since the graphics hardware was not capable of rendering displacement maps, although ideas exist on how to extend the hardware with this feature [9, 10, 20].

A similar technique used in a different context is image warping. It is very similar to displacement mapping, only that in image warping adjacent pixels need not to be connected, allowing to see through them for certain viewing directions. Displacement mapping is usually applied to a larger number of polygons, whereas image warping is often done for a few images only. Techniques that use image warping are also traditionally software-based [12, 16, 19, 25, 26, 27].

Displacement mapping recently made its way into hardware accelerated rendering using standard features. The basic technique was introduced by Schaufler [24] in the context of warping for layered impostors. It was then reintroduced in the context of displacement mapping by Dietrich [8]. This algorithm encodes the displacement in the α-channel of a texture. It then draws surface-aligned slices through the volume defined by the maximum displacement. The α-test is used to render only the appropriate parts of every slice. Occlusions are handled properly by this method.

This algorithm works well only for surface-aligned slices. At grazing angles it is possible to look through the slices. In this case, Schaufler [24] regenerates the layered impostor, i.e. the texture and the displacement map, according to the new viewpoint, which is possible since he does have the original model that the layered impostor represents.

We will introduce an enhanced method that supports arbitrary slicing planes, allowing orthogonal slicing directions or screen-space aligned slicing commonly used in volume rendering, eliminating the need to regenerate the texture and displacement map.

On the one hand, we use this new method to render

traditional displacement mapped objects; see Figure 1. This works at interactive rates even for large textures and displacements employing current graphics hardware.

On the other hand, this new method can be extended to render a new kind of image-based object, based on images with depth, which we will refer to as *image based depth objects*. How to reconstruct an object from several images with depth has been known for many years now [2, 3, 6]. The existing methods are purely software based, very slow, and often working on a memory consuming full volumetric representation. We introduce a way to directly render these objects at interactive rates using graphics hardware without the need to reconstruct them in a preprocessing step. The input images are assumed to be registered beforehand.

We will also show how the image-based visual hull algorithm [13] can be implemented using this new method, and which runs much faster than the original algorithm.

Many other image based rendering algorithms also use some kind of image warping [12, 16, 19, 25, 26, 27]. The acceleration of these algorithms using our technique is conceivable.

2 Prior Work

We will briefly review previous work from the areas of displacement mapping, image warping, object reconstruction, and image-based objects.

Displacement Mapping was introduced by Cook [4] and has been traditionally used in software based methods, e.g. using raytracing or micro-polygons. Patterson et al. [21] have introduced a method that can raytrace displacement mapped polygons by applying the inverse of this mapping to the rays. Pharr and Hanrahan [22] have used geometry caching to accelerate displacement mapping. Smits et al. [29] have used an approach which is similar to intersecting a ray with a height-field. The REYES rendering architecture subdivided the displacement maps into micro-polygons which are then rendered [5].

On the other hand many image-based rendering (IBR) techniques revolve around image warping, which was e.g. used by McMillan et al. [16] in this context. There are two different ways to implement the warping: forward and backward mapping. Forward mapping loops over all pixels in the original image and projects them into the desired image. Backward mapping loops over all pixels in the desired image and searches for the corresponding pixels in the original image. Forward mapping is usually preferred, since the search process used by backward mapping is expensive, although forward mapping may introduce holes in the final image. Many algorithms have been proposed to efficiently warp images [1, 15, 20, 28]. All of them work in software, but some are designed to be turned into hardware.

The only known hardware accelerated method to do image warping was introduced by Schaufler [24]. Dietrich [8] used it later on for displacement mapping. This algorithm will be explained in more detail in the next section. It has the main problem of introducing severe artifacts at grazing viewing angles.

Many IBR techniques employ (forward) image warping [12, 19, 20, 25, 26, 27] but also using a software implementation.

New hardware has also been proposed that would allow displacement mapping [9, 10, 20], but none of these methods have found their way into actual hardware.

The reconstruction of objects from images with depth has been researched for many years now. Various different algorithms have been proposed [2, 3, 6] using two different approaches: reconstruction from unorganized point clouds, and reconstruction that uses the underlying structure. None of these algorithms using either approach can reconstruct and display such an object in real-time, whereas our method is capable of doing this.

There are many publications on image based objects; we will briefly review the closely related ones. Pulli et al. [23] hand-model sparse view-dependent meshes from images with depth in a preprocessing step and recombine them on-the-fly using a soft z-buffer. McAllister et al. [14] use images with depth to render complex environments. Every seen surface is stored once in exactly one of the images. Rendering is done using splatting or with triangles. Layered depth images (LDI) [27] store an image plus multiple depth values along the direction the image was taken; reconstruction is done in software. Image-based objects [19] combine six LDI arranged as a cube with a single center of projection to represent objects. An object defined by its image-based visual hull [13] can be rendered interactively using a software renderer.

Our method for rendering image-based objects is one of the first purely hardware accelerated method achieving high frame rates and quality. It does not need any preprocessing like mesh generation, it only takes images with depths.

3 Displacement Mapping

The basic idea of displacement mapping is simple. A base geometry is displaced according to a displacement function, which is usually sampled and stored in an array, the so-called displacement map. The displacement is performed along the interpolated normals across the base geometry. See Figure 2 for a 2D example where a flat line is displaced according to a displacement map along the interpolated normals.

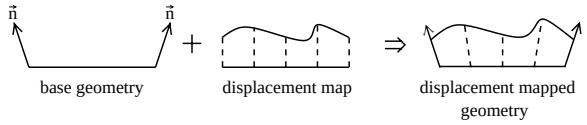
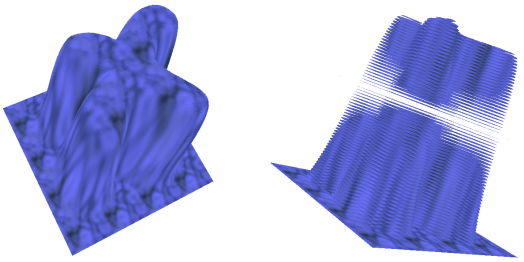


Figure 2: *Displacement Mapping.*

3.1 Basic Hardware Accelerated Method

First we would like to explain the basic algorithm for doing displacement mapping using graphics hardware as it was introduced by Dietrich [8] (and in a similar way by Schaufler [24]).

The input data for our displacement mapping algorithm is an RGBα-texture, which we call *displacement texture*, where the color-channels contain color information and the α-channel contains the displacement map. In Figure 3 you can see the color texture and the α-channel of a displacement texture visualized in different images. The displacement values stored in the α-channel represent the distance of that particular pixel to the base geometry, i.e. the distance along the interpolated normal at that pixel.



top view side view

Figure 4: *Top view and side view of a displaced polygon using the basic method (64 slices).*

In order to render a polygon with a displacement texture applied to it, we render slices (i.e. polygons) through the enclosing volume extruded along the surface's normal directions, which we will call the *displacement volume*; see right side of Figure 3. Every slice is drawn at a certain distance to the base polygon textured with the displacement texture. In every slice only those pixels should be visible whose displacement value is greater or equal the height of the slice. This can be achieved by using the α-test. For every slice that is drawn we convert its height to an α-value h_α in the range $[0, 1]$, where $h_\alpha = 0$ corresponds to no elevation; see Figure 3. We then enable the α-test so that only fragments pass whose α-value is greater or equal h_α.

As you can see in Figure 3 this method completely fills the inside of a displacement (which will be needed later on).

Schaufler [24] used a slightly different method. In every slice only those pixels are drawn whose α-values lie within a certain bound of the slice's height. For many viewpoints this allows to see through neighboring pixels whose displacement values differ more than the used bound. This is suited to image warping in the traditional sense, where it is assumed that pixels with very different depth values are not connected. The method we described is more suited to displacement mapping, where it is assumed that neighboring pixels are always connected.

Both methods have the problem that at grazing angles it is possible to look through the slices; see Figure 4 for an example. Schaufler [24] simply generates a new displacement texture for the current viewpoint using the original model. In the next section we introduce an enhanced algorithm that eliminates the need to regenerate the displacement texture.

3.2 Orthogonal Slicing

It is desirable to change the orientation of the slices to avoid the artifacts that may occur when looking at the displacement mapped polygon from grazing angles as seen in Figure 4.

Meyer and Neyret [17] used orthogonal slicing directions for rendering volumes to avoid artifacts that occurred in the same situation. We use the same possible orthogonal slicing directions, as depicted in Figure 5. Depending on the viewing direction, we choose the slicing direction that is most perpendicular to the viewer and which will cause the least artifacts.

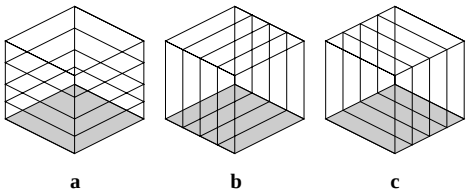


Figure 5: *The three orthogonal slicing directions. Only slicing direction* **a** *is used by the basic algorithm.*

Unfortunately, we cannot directly use the previously employed α-test since there is no fixed α-value h_α (see Figure 3) that could be tested for slicing directions **b** and **c**; see Figure 5. Within every slice the α-values h_α vary from 0 to 1 (bottom to top); see the ramp in Figure 6. Every α-value in this ramp corresponds to the pixel's distance from the base geometry, i.e. h_α.

A single slice is rendered as follows. First we extrude the displacement texture along the slicing polygon, which is done by using the same set of texture coordi-

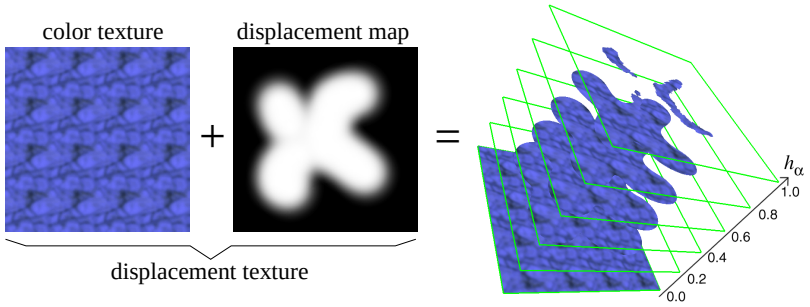


Figure 3: *Displacement Mapping using Graphics Hardware.*

nates for the lower and upper vertices. Then we subtract the α-ramp (applied as a texture or specified as color at the vertices) from the α-channel of the displacement texture, which we do with NVIDIA's register combiners [18] since this extension allows to perform the subtraction in a single pass. The resulting α-value is greater than 0 if the corresponding pixel is part of the displacement. We set the α-test to pass only if the incoming α-values are greater than 0. You can see in Figure 6 how the correct parts of the texture map will be chosen.

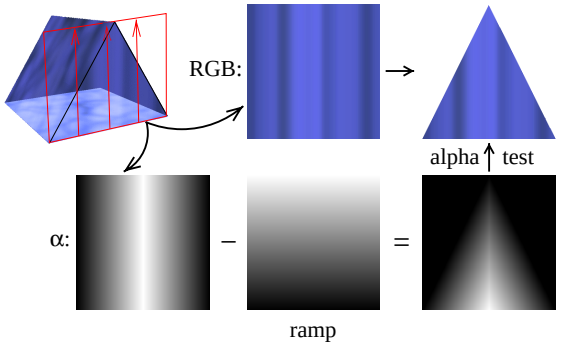


Figure 6: *The necessary computation involved for a single orthogonal slice. First the displacement texture is extruded along the slicing polygon. The resulting α and RGB channels of the textured slicing polygon are shown separately. Then, the shown α-ramp is subtracted. The resulting α-values are > 0 if the pixel lies inside the displacement. The α-test is used to render only these pixels.*

Now that we know how this is done for a single slice, we apply this to many slices and can render the displacement mapped polygon seen in Figure 4 from all sides without introducing artifacts; see Figure 7.

This algorithm works for the slicing direction **b** and **c**. It can also be applied for direction **a**, we just use the register combiners to subtract the per-slice h_α-value from the α-value in the displacement map (for every slice) and

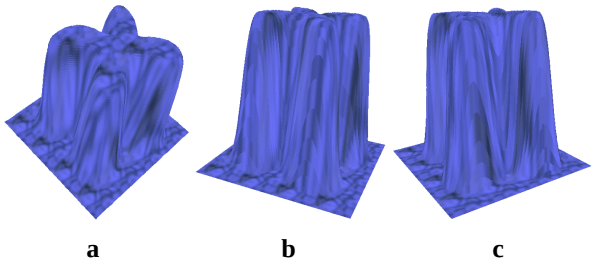


Figure 7: *Displacement mapped polygon rendered with all three slicing directions (using 64 slices each time).*

perform the α-test as just described.

Using the same algorithm for all slicing directions treats displacement map values of 0 consistently. The basic algorithm does render pixels if the displacement value is 0, which corresponds to no elevation. The new method does *not* draw them, it starts rendering pixels if their original displacement value is greater than 0. This has the advantage that parts of the displaced polygon can be masked out by setting the displacement values to 0.

3.3 Screen-Space Slicing

Orthogonal slicing is already a good method to prevent one from looking through the slices. From volume rendering it is known that screen-space aligned slicing, which uses slices that are parallel to the viewplane, is even better. In Figure 8 it is shown why this is the case. The screen-space aligned slices are always orthogonal to the view direction and consequently preventing him/her from seeing through or in-between the slices.

The new method described in the last section can be easily adapted to allow screen-space aligned slicing.

Our technique can be seen as a method that cuts out certain parts of the displacement volume over the base surface. The parts of the volume which are larger than the specified displacements are not drawn.

In Figure 9 you can see an arbitrary slicing plane in-

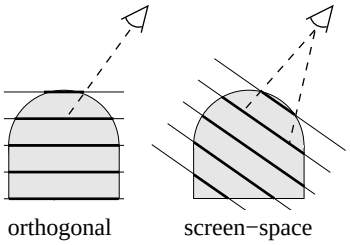


Figure 8: *Orthogonal vs. screen-space aligned slices.*

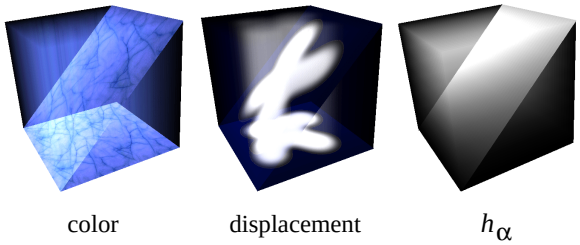


Figure 9: *Intersection of arbitrary slicing plane with the displacement volume.*

tersecting this volume. Three intersections are shown: with the extruded color texture, the extruded displacement map, and with the h_α-volume. Of course only those parts of this slicing plane should be drawn that have a displacement-value (as seen in the middle) that is equal or greater h_α (as seen on the right). To achieve this, we use the exact same algorithm from the previous section, i.e. we subtract the (intersected) α-ramp from the (intersected) displacement map and use the resulting α-value in conjunction with the α-test to decide whether to draw the pixel or not.

The only difficulty is the computation of the texture coordinates for an arbitrary slicing plane, so that it correctly slices the volume. For screen-space aligned slices this boils down to applying the inverse modelview matrix, which was used for the base geometry, to the original texture coordinates plus some additional scaling/translation, so that the resulting texture coordinates lie in the [0,1] range. This can be done using the texture matrix.

Now it is possible to render the displacement using screen-space aligned slices, as depicted in Figure 8.

The actual implementation is a bit more complicated depending on the size and shape of the slices. The simplest method generates slices that are all the same size, as seen in Figure 8. Then one must ensure that only those parts of the slices are texture mapped that intersect the displacement volume. This can be done using texture borders where the α-channel is set to 0, which ensures that nothing is drawn there (pixels with displacement values of 0 are not drawn at all, see previous section). Unfortunately, this takes up a lot of fill rate that could be used otherwise. A more complicated method intersects the slices with the displacement volume and generates new slicing polygons which exactly correspond to the intersection. This requires less fill rate, but the computation of the slices is more complicated and burdens the CPU.

3.4 Comparison of Different Slicing Methods

The surface-aligned slicing method presented in Section 3.1 is the simplest method. It only works well when looking from the top onto the displacement, otherwise it is possible to look through the slices.

The orthogonal slicing method is already a big improvement over the simplistic basic method. But it should be mentioned that slicing in other directions than orthogonally to the base surface usually requires more slices. This is visualized in Figure 10. The orthogonal slicing direction **a** achieves acceptable results even with a few slices, whereas the slicing direction **b** produces unusable results. This can be compensated if the number of slices used is adjusted according to the ratio of the maximum displacement and the edge length of the base geometry. For example, if the base polygon has an edge length of 2 and the maximum displacement is 0.5, then 4 times as many slices should be used for the slicing direction **b** (or **c**). This also keeps the fill rate almost constant.

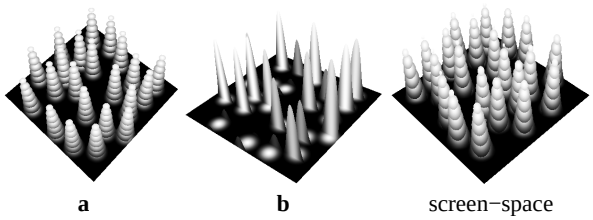


Figure 10: *Comparison of different slicing directions (**a**, **b**, and screen-space).*

Screen-space aligned slicing should offer the best quality since the viewing direction is always orthogonal to the slices. While this is true (see Figure 10), screen-space aligned slicing can introduce a lot of flickering, especially if not enough slices are used. In any case, the screen-space method is more expensive than orthogonal slicing since some more care has to be taken that only the correct parts are rendered; see the previous section.

The absolute number of slices that should be used depends on the features of the displacement map itself and also on the size the displacement takes up in screen-space. Different criteria that have been proposed by Schaufler [24] and Meyer and Neyret [17] can be applied here as well.

4 Image Based Depth Objects

So far, we have shown how we can efficiently render polygons with a displacement map. We can consider a single displacement mapped polygon as an object with heightfield topology. The input data for this object is a color texture and a depth image, which we assume for a moment to have been taken with an (orthogonal) camera that outputs color and depth. What if we take more images with depth of this object from other viewpoints? Then the shape of the resulting object, which does not necessarily have heightfield topology anymore, is defined by the intersection of all the displaced images. This is shown in Figure 11 for two input images with depth. As you can see the resulting object has a complex non-heightfield shape. Many software-based vision algorithms exist for reconstructing objects using this kind of input data, e.g. [2, 3, 6].

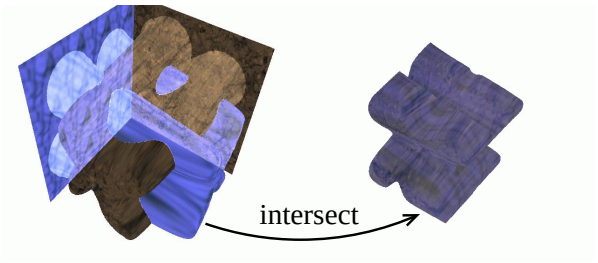


Figure 11: *Intersection of two displacement maps.*

Our displacement mapping technique can be easily extended to rendering this kind of object without explicitly reconstructing it. What needs to be done is to calculate the intersection between displacement mapped polygons. We will look at the special case, where the base polygons are arranged as a cube and the intersection object is enclosed in that cube; other configurations are possible. This algorithm can use screen-space aligned slices as well as orthogonal slices. In our description we focus on orthogonal slicing for the sake of simplicity.

Let us look at a single pixel that is enclosed in that cube and which is to be drawn. We have to decide two things: firstly, is it part of the object. If so, then it should be rendered or otherwise be discarded. And secondly, given the pixel is part of the object, which texture map should be applied. We will first deal with the former problem and in the next section with latter.

4.1 Rendering

The decision whether to render or discard a pixel is fairly simple. Since we assume a cube configuration, we know that the pixel is inside the displacement volumes of all polygons. A pixel is part of the object, if the α-tests succeeds for all six displacement maps.

In Figure 16 you can see how this works conceptually: One slice is cutting through the cube defined by four enclosing polygons (usually six but for clarity only four). For every polygon we apply our displacement mapping algorithm with the given slicing polygon. The pixels on the slicing plane are colored according to the base polygon where the α-test succeeded. Only the pixels that are colored with all colors belong to the object resulting in white pixels in Figure 16, whereas the other pixels have to be discarded.

With an imaginatory graphics card that has a lot of texture units and that allows many operations to be done in the multitexturing stage the rendering algorithm is simple. The slicing polygon is textured with the projections of all the displacement textures of the base polygons as well as the according α-ramps. For every displacement map we compute the difference between its displacement values and the α-value h_α from the ramp texture (see Section 3.2). The resulting α-value is greater zero if the pixel belongs to the displacement of that particular displacement map. We can now simply multiply the resulting α-values of all displacement maps. If it is still greater 0, we know that all the α-values are greater 0 and the pixel should be drawn, otherwise it should be discarded. As explained before we check this with an α-test that lets only pass fragments with α greater 0.

Although it is expected that future graphics cards will have more texture units and even more flexibility in the multitexturing stage, it is unlikely that they will soon be able to run the just described algorithm. Fortunately, we can use standard OpenGL to do the same thing, only that it is a bit more complicated and requires the stencil buffer:

1. Clear frame buffer and disable depth-test.
2. Loop over slices from front to back
   (a) Loop i over all base polygons
      i. Set stencil test to pass and increment if stencil value equals $i - 1$, otherwise keep it and fail test
      ii. Render slice (using the α-test)
   (b) *// Stencil value will equal total number of base // polygons where all α-tests passed*
   (c) *// Now clear frame buffer where stencil value is // less than total number of base polygons:*
   (d) Set stencil test to pass and clear, if stencil $\neq$ total number of base polygons, otherwise keep stencil (those parts have to remain in the frame buffer)
   (e) Draw slice with background color
   (f) *// Parts with stencil = total number of base polygons // will remain, others are cleared*

Please note that we slice the cube from front to back in the "best" orthogonal direction.

4.2 Texture Mapping

So far, we have only selected the correct pixels, but we still have to texture map them with the "best" texture map. There are as many texture maps as base polygons and the most appropriate is the one that maps onto the pixel along a direction which is close to the viewing direction.

Instead of using only one texture map, we choose the three texture maps which come closest to the current viewing direction. First we compute the angles between the normals of the base polygons and the viewing direction. We then choose those three base polygons with the smallest angles and compute three weights, summing up to one, that are proportional to the angles. The weights for the other base polygons are set to zero. When we now render a slice in turn with all the displacement textures defined by the base polygons (see algorithm in previous subsection), we set the color at the vertices of the slice to the computed weights. The contributions of the different textures are summed up using blending. This strategy efficiently implements view-dependent texturing [7].

5 Image Based Visual Hull

The algorithm that was described in the previous section can also be used to render objects based on their visual hull, for which Matusik et al. [13] proposed an interactive rendering algorithm that uses a pure software solution.

These objects are defined by their silhouette seen from different viewpoints. Such an object is basically just the intersection of the projections of the silhouettes. The computation of the intersection is almost exactly what our algorithm does, only that we also take into account per-pixel depth values. The only thing that we need to change in order to render a visual hull object is the input data. The α-channel of the displacement maps contains 1s inside the silhouette and 0s outside. Then we can run the same algorithm that was explained in the previous section.

If the input images are arranged as a cube, the algorithm can be streamlined a bit more, since opposing silhouettes are the same. A graphics card with something similar to NVIDIA's register combiner extension and four texture units would then be able to render a visual hull object in only a single pass per slice.

6 Results and Discussion

We have verified our technique using a number of models and displacement textures. All our timings were measured on on a PIII/800 using an NVIDIA GeForce 2 GTS.

Figure 1, Figure 17, and Figure 18 show different displacement maps applied to a simple donut with 625 polygons. We used between 15 and 25 slices together with the orthogonal slicing technique. The frame rates varied between 35 and 40Hz. This technique is heavily fill rate dependent and the number of additional slicing polygons can be easily handled by the geometry engine of modern graphics cards.

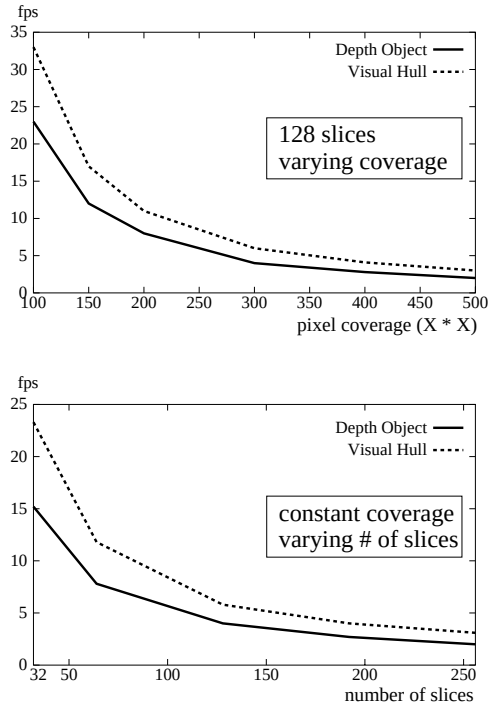


Figure 12: *Comparison of timings for rendering the creature. The upper graph shows how the frame rate varies with different pixel coverage but constant number of slices (128 in this case). The lower graph shows frame rates depending on the number of slices but for a fixed size (300 $\times$ 300 pixels).*

In Figure 13 the input data for our image-based depth object algorithm is shown — a creature orthogonally seen through six cube faces. In Figure 14 you can see the creature rendered with our method. The achieved frame rates are heavily fill rate dependent. When the object occupies about 150×150 pixels on the screen, we achieve about 24 frames per second using 70 slices (high quality). For 400×400 pixels about 150 slices are needed for good quality yielding about 2.7 frames per second. In Figure 12 two graphs show the variation in frame rates depending on the pixel coverage and the number of slices.

We also noted that the rendering speed depends on the viewing angle relative to the slicing polygons. The more the slicing polygons are viewed at an angle, the better the frame rate (up to 20% faster). This is not surprising, since less pixels have to drawn.

With the next generation graphics cards (e.g. GeForce 3), which have four texture units, the frame rate

is likely to almost double.

As you can see under the creature's arm, naïve view-dependent texturing is not always ideal. Even if a part of the object has not been seen by any of the images, it will be textured anyway, which can produce undesirable results.

In Figure 15 you can see our algorithm working on the same input data, only that all the depth values greater than 0 were set to 1. This corresponds to the input of a visual hull algorithm. You can see that many artifacts are introduced, because there are not enough input images for an exact rendering of the object. Furthermore, many concave objects, e.g. a cup, cannot be rendered correctly at all using the visual hull, unlike the image-based depth objects that can handle concave objects. Frame rates are increased for the visual hull compared to the depth objects (see Figure 12), because only the three front-facing polygons of the cube are used (opposing cube faces have the same silhouettes).

7 Conclusions and Future Work

We have presented an efficient technique that allows to render displacement mapped polygons at interactive rates on current graphics cards. Displacement mapped polygons are rendered by cutting slices through the enclosing displacement volume. The quality is improved over previous methods with a flexible slicing method.

This flexible slicing method allows the introduction of image-based depth objects. An image-based depth object is defined by the intersection of displacement mapped polygons. These depth objects can be rendered using our displacement mapping technique at interactive frame rates. The quality of the resulting images is high, but can be sacrificed for speed by choosing fewer slicing planes. Depth objects can handle fairly complex shapes, especially compared to the similar image-based visual hull algorithm.

Shading of the image-based depth objects is handled by using view-dependent texture mapping. Reshading can be accomplished by using not only colors as an input but also using a texture map storing normals, which can then be used to perform the shading [11]. This can also be used to shade the displacement mapped polygons, which doesn't even require more rendering passes on NVIDIA GeForce class graphics cards since only the first texture unit is needed for the displacement mapping algorithm keeping the second unit available.

Furthermore, animating the displacement maps is possible much in the same way as it was proposed by Meyer and Neyret [17]. Also animated depth objects are easily possible, only prerendered texture maps have to be loaded the graphics card.

For the image-based depth objects we have only used images with "orthogonal" depth values. The technique can be easily extended to images with "perspective" depth values.

Acknowledgements

We would like to thank Hiroyuki Akamine for writing the 3D Studio Max plugin to save depth values. Thanks to Hartmut Schirmacher for the valuable discussions about this method.

8 References

[1] B. Chen, F. Dachille, and A. Kaufman. Forward Image Warping. In *IEEE Visualization*, pages 89–96, October 1999.

[2] Y. Chen and G. Medioni. Surface Description Of Complex Objects From Multiple Range Images. In *Proceedings Computer Vision and Pattern Recognition*, pages 153–158, June 1994.

[3] C. Chien, Y Sim, and J. Aggarwal. Generation of Volume/Surface Octree From Range Data. In *Proceedings Computer Vision and Pattern Recognition*, pages 254–260, June 1988.

[4] R. Cook. Shade Trees. In *Proceedings SIGGRAPH*, pages 223–231, July 1984.

[5] R. Cook, L. Carpenter, and E. Catmull. The Reyes Image Rendering Architecture. In *Proceedings SIGGRAPH*, pages 95–102, July 1987.

[6] B. Curless and M. Levoy. A Volumetric Method for Building Complex Models from Range Images. In *Proceedings SIGGRAPH*, pages 303–312, August 1996.

[7] P. Debevec, Y. Yu, and G. Borshukov. Efficient View-Dependent Image-Based Rendering with Projective Texture-Mapping. In *9th Eurographics Rendering Workshop*, pages 105–116, June 1998.

[8] S. Dietrich. Elevation Maps. Technical report, NVIDIA Corporation, 2000.

[9] M. Doggett and J. Hirche. Adaptive View Dependent Tessellation of Displacement Maps. In *Proceedings SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 59–66, August 2000.

[10] S. Gumhold and T. Hüttner. Multiresolution Rendering with Displacement Mapping. In *Proceedings SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 55–66, August 1999.

[11] W. Heidrich and H.-P. Seidel. Realistic, Hardware-accelerated Shading and Lighting. In *Proceedings SIGGRAPH*, pages 171–178, August 1999.

[12] W. Mark, L. McMillan, and G. Bishop. Post-Rendering 3D Warping. In *Symposium on Interactive 3D Graphics*, pages 7–16, April 1997.

[13] W. Matusik, C. Buehler, R. Raskar, S. Gortler, and L. McMillan. Image-Based Visual Hulls. In *Proceedings SIGGRAPH*, pages 369–374, July 2000.

[14] D. McAllister, L. Nyland, V. Popescu, A. Lastra, and C. McCue. Real-Time Rendering of Real-World Environments. In *10th Eurographics Rendering Workshop*, pages 153–168, June 1999.

[15] L. McMillan and G. Bishop. Head-Tracked Stereoscopic Display Using Image Warping. In *Proceedings SPIE*, pages 21–30, February 1995.

[16] L. McMillan and G. Bishop. Plenoptic Modeling: An Image-Based Rendering System. In *Proceedings SIGGRAPH*, pages 39–46, August 1995.

[17] A. Meyer and F. Neyret. Interactive Volumetric Textures. In *9th Eurographics Rendering Workshop*, pages 157–168, June 1998.

[18] NVIDIA Corporation. *NVIDIA OpenGL Extension Specifications*, November 1999. Available from http://www.nvidia.com.

[19] M. Oliveira and G. Bishop. Image-Based Objects. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 191–198, April 1999.

[20] M. Oliveira, G. Bishop, and D. McAllister. Relief Texture Mapping. In *Proceedings SIGGRAPH*, pages 359–368, July 2000.

[21] J. Patterson, S. Hoggar, and J. Logie. Inverse Displacement Mapping. *Computer Graphics Forum*, 10(2):129–139, June 1991.

[22] M. Pharr and P. Hanrahan. Geometry Caching for Ray-Tracing Displacement Maps. In *7th Eurographics Rendering Workshop*, pages 31–40, June 1996.

[23] K. Pulli, M. Cohen, T. Duchamp, H. Hoppe, L. Shapiro, and W. Stuetzle. View-based Rendering: Visualizing Real Objects from Scanned Range and Color Data. In *8th Eurographics Rendering Workshop*, pages 23–34, June 1997.

[24] G. Schaufler. Per-Object Image Warping with Layered Impostors. In *9th Eurographics Rendering Workshop*, pages 145–156, June 1998.

[25] G. Schaufler and M. Priglinger. Efficient Displacement Mapping by Image Warping. In *10th Eurographics Rendering Workshop*, pages 183–194, June 1999.

[26] H. Schirmacher, W. Heidrich, and H.-P. Seidel. High-Quality Interactive Lumigraph Rendering Through Warping. In *Proceedings Graphics Interface*, pages 87–94, 2000.

[27] J. Shade, S. Gortler, L. He, and R. Szeliski. Layered Depth Images. In *Proceedings SIGGRAPH*, pages 231–242, July 1998.

[28] A. Smith. Planar 2-Pass Texture Mapping and Warping. In *Proceedings SIGGRAPH*, pages 263–272, July 1987.

[29] B. Smits, P. Shirley, and M. Stark. Direct Ray Tracing of Displacement Mapped Triangles. In *11th Eurographics Workshop on Rendering*, pages 307–318, June 2000.

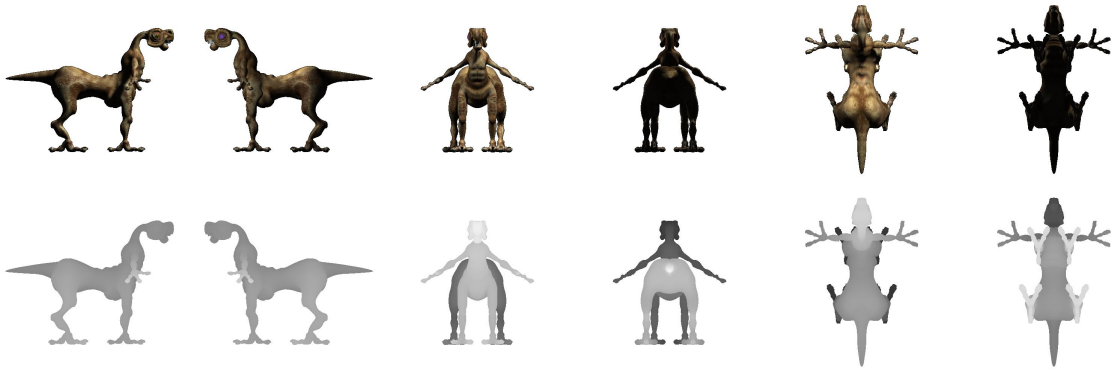


Figure 13: *The input data for the creature model (color and depth).*

Figure 14: *Image-Based Depth Object.*

Figure 15: *Image-Based Visual Hull.*

Figure 16: *One slice through an image-based depth object.*

Figure 17: *Displacement mapped donut (20 slices, 38Hz).*

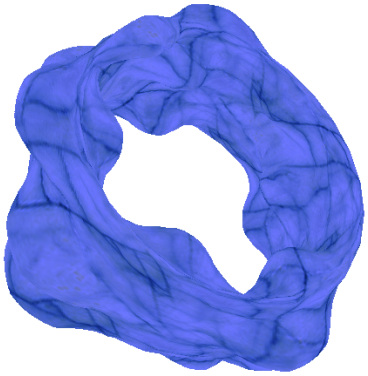


Figure 18: *Displacement mapped donut (15 slices, 41Hz).*