Real-Time Relief Mapping on Arbitrary Polygonal Surfaces

Fábio Policarpo*
Paralelo Computação

Manuel M. Oliveira†
Instituto de Informática
UFRGS

João L. D. Comba‡
Instituto de Informática
UFRGS

Figure 1: Teapot rendered with different relief textures using per-pixel lighting and self-shadowing.

Abstract

This paper presents a technique for mapping relief textures onto arbitrary polygonal models in real time. In this approach, the mapping of the relief data is done in tangent space. As a result, it can be applied to polygonal representations of curved surfaces producing correct self-occlusions, interpenetrations, shadows and per-pixel lighting effects. The approach can be used to consistently add surface details to geometric models undergoing deformations, such as in the case of animated characters commonly found in games. The technique uses an inverse formulation (*i.e.*, pixel driven) based on an efficient ray-height-field intersection algorithm implemented on the GPU. It supports extreme close-up views of the surfaces, mip mapping and anisotropic texture filtering. Also, contrary to high-dimensional representations of surface details, the low memory requirements of the proposed technique do not restrict its use to tiled textures.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: image-based rendering, relief mapping, motion parallax, real-time rendering, surface details.

1 Introduction

Texture mapping [Catmull 1974] is a fundamental component of modern image synthesis systems, playing a major role in enhancing the realism of scenes. It adds significant amount of details to surfaces by simulating the appearance of different materials, the existence of surface wrinkles [Blinn 1978] or even by modifying the underlying geometric model [Cook 1984]. In recent years, a few texture mapping extensions have been introduced, for instance, for rendering texture appearance under different lighting orientation [Malzbender et al. 2001] and 3D surface details [Oliveira et al. 2000]. Relief texture mapping [Oliveira et al. 2000] simulates the existence of 3D surface details using image warping techniques. Thus, correct views of geometrically rich objects can be obtained by rendering just a few textured polygons.

Figure 2: Bronto's tea hour. The surface details for the character, the teapot and the walls were rendered using relief mapping.

In this paper, we show how relief textures can be mapped onto arbitrary polygonal models by performing the mapping in tangent space [Peercy et al. 1997]. Figures 1 and 2 show several objects rendered with different surface details. These images also include per-pixel lighting and shadows cast by the surface relief. Compared to other recently proposed techniques to represent surface de-

*e-mail:fabio@paralelo.com.br
†e-mail:oliveira@inf.ufrgs.br
‡e-mail:comba@inf.ufrgs.br

tails [Wang et al. 2003; Wang et al. 2004], our approach presents several desirable features. In particular, it uses a much more compact representation, is easy to implement, supports arbitrary close-up views without introducing noticeable texture distortions, and supports mip mapping and anisotropic texture filtering.

The contributions of this paper include:

- A technique for mapping relief textures onto arbitrary polygonal models in real time (Section 3). The resulting renderings present correct self-occlusions, interpenetrations, shadows and per-pixel lighting. Moreover, it supports extreme close-up views of the added surface details;
- An efficient algorithm for computing ray-height-field intersections using programmable graphics hardware (Section 3);
- A new relief texture representation based on two depth layers (Section 4). Such a representation significantly improves the rendering quality of single-depth-layer relief textures when mapped onto single polygons. This is achieved with no extra storage cost and small additional computational cost.

2 Related Work

Bump mapping [Blinn 1978] simulates the existence of surface details using normal perturbation. Since the actual surface is not modified, self-occlusions, shadows and silhouettes are not accounted for. Horizon maps [Max 1988] provide a solution for shadowing bump-mapped surfaces and can be implemented using the graphics hardware [Sloan and Cohen 2000]. An alternative solution that can also be implemented using hardware acceleration was presented by Heidrich et al. [Heidrich et al. 2000].

Displacement mapping [Cook 1984] changes the actual geometry of the underlying surface and as a result produces correct self-occlusions, shadows and silhouettes. Unfortunately, the use of displacement maps requires rendering a large number of micro-polygons, which is undesirable for interactive applications. In recent years, several approaches have been devised to accelerate the rendering of displacement maps and to avoid explicit rendering of micro-polygons. These approaches are based on ray tracing [Patterson et al. 1991; Pharr and Hanrahan 1996; Heidrich and Seidel 1998; Smits et al. 2000], 3D inverse image warping [Schaufler and Priglinger 1999] and 3D texture mapping [Meyer and Neyret 1998; Kautz and Seidel 2001]. The demonstrated ray-tracing and inverse-warping-based approaches are computationally expensive and not appropriate for real-time applications. The 3D-texture approaches render displacement maps as stacks of 2D texture-mapped polygons. This may introduce objectionable artifacts depending on the viewing direction. Hirche et al. [Hirche et al. 2004] use graphics hardware to ray cast displaced surfaces inside extruded tetrahedra and Gumhold [Gumhold 2003] implemented a pixel shader ray-caster to render ellipsoids.

More recently, Wang et al. [Wang et al. 2003] presented a technique that pre-computes the distances from each displaced point to a reference surface. These distances are computed along many sampling viewing directions. The result is a five-dimensional function called a view-dependent displacement map (VDM) that can be queried at rendering time. Due to the large sizes of these datasets, VDMs need to be compressed before they can be stored in the graphics card memory. The compression is done using principal component analysis techniques. This approach works in real time and can produce nice results, but has some drawbacks: it introduces significant texture distortions and can only be applied to closed surfaces. Due to the large sizes of these representations, usually a single patch is created and tiled over the entire surface. Also, due to the pre-computed resolution of these representations, they are intended for rendering from a certain distance and should not be used for close-up renderings.

In order to reduce texture distortions and handle surfaces with boundaries, Wang et al. [Wang et al. 2004] introduced another five-dimensional representation called GDM for rendering non-height-field structures. GDM also produces large sampling databases that need to be compressed. Likewise, GDMs are more appropriate for tiling and renderings from a certain distance.

Parallax mapping [Kaneko et al. 2001] uses textures augmented with per-texel depth. In this approach, the texture coordinates along the view direction are shifted based on the depth values using an approximate solution. While this technique can produce interesting results at very low cost, it is only appropriate for noisy irregular bumps, as the surfaces are inaccurately and dynamically deformed as the viewing position changes. No support for shadows has been demonstrated for parallax mapping.

2.1 Review of Relief Texture Mapping

Relief texture mapping [Oliveira et al. 2000] uses image warping techniques and textures enhanced with per-texel depth to create the illusion of complex geometric details when mapped onto flat polygons. The depth information is computed as the distance from a reference plane to the sampled surface. Figure 3 shows an example of a relief texture. On the left, one sees a diffuse pre-shaded color texture with its corresponding depth data on the right.

Figure 3: A relief texture: diffuse pre-shaded color (left) and depth map (right). In the depth map, dark means closer.

Ideally, portions of a relief texture should only be warped on demand. However, the rendering of a height field is not as simple as conventional texture mapping, requiring a search for the closest surface along any given viewing ray. In order to avoid this search, which tends to be computationally expensive, Oliveira et al. factored the mapping into a two-step process [Oliveira et al. 2000]. First, the height field is converted into a regular 2D texture using a forward transformation. Then, the resulting texture is mapped onto the polygon in the conventional way. Figure 4 compares the renderings produced by different techniques from the same viewpoint. In (a), the image was conventionally texture-mapped onto a quadrilateral. Figure 4 (b) shows the result obtained when applying relief texture mapping to the same polygon. For comparison, Figure 4 (c) shows the color and depth data presented in Figure 3 rendered as a mesh of micro-polygons. Essentially, the results in (b) and (c) are indistinguishable from each other.

Oliveira et al. [Oliveira et al. 2000] represented 3D objects by relief texture-mapping the visible faces of parallelepipeds. This situation is illustrated in Figure 5, where one sees an object rendered as two

(a) (b) (c)

Figure 4: Rendering comparison from the same viewpoint. (a) Color image rendered as a conventional texture. (b) Relief texture mapping rendering. (c) Rendering of the color and depth data in Figure 3 as a mesh of micro-polygons.

relief texture-mapped polygons. The borders of the two polygons are shown on the left. This method, however, has not been extended to arbitrary surfaces. ElHew and Yang [ElHelw and Yang 2003] used cylindrical versions of relief textures (*i.e.*, cylindrical images with depth measured along the normal directions to the cylinder) to render images of endoscopic simulations. They create inside-looking-outside renditions by warping the cylindrical textures according the the viewer's position and by texture-mapping the result onto a reference cylinder. Their technique cannot be generalized to arbitrary surfaces.

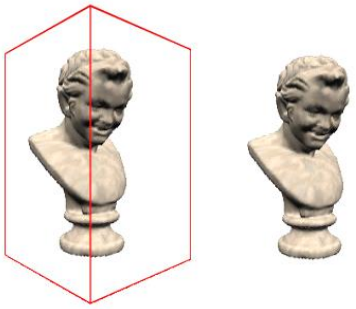

Figure 5: 3D objects are rendered by relief texture-mapping the visible faces of a parallelepiped. Object rendered showing the borders of the quads (left) and without borders (right).

3 Relief Mapping on Polygonal Surfaces

We exploit the programmability of modern graphics hardware to effectively render surface details onto arbitrary polygonal surfaces. Since the rendering is performed using fragment shaders, we can also perform per-pixel shading and compute shadows. Thus, the color texture originally used to store pre-computed diffuse shading can be discarded and replaced by a normal map. Any 2D texture can be mapped onto the resulting representation. Figure 6 shows a relief texture represented by its corresponding depth and normal maps. The depth map is quantized and represented using the alpha channel of the RGBα texture used to store the normal map. This way, a single 32-bit per texel texture suffices to represent the structure of a relief texture.

We normalize the height values to the $[0,1]$ range. Figure 7 shows the representation (cross-section) of such a height-field surface.

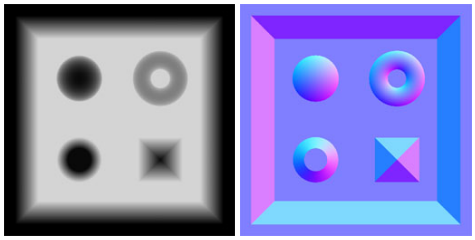

Figure 6: A relief texture represented by a depth (left) and a normal map (right). The normals are mapped to the [0,1] range and stored as an RGB image.

From top to bottom, the depth values vary from 0.0 to 1.0.

The process of mapping relief data to a polygonal surface can be conceptually understood as following. For each fragment to be rendered:

- compute the viewing direction (VD) as the vector from the viewer to the 3D position of the point on the polygonal surface;
- transform VD to the tangent space (defined by the tangent, normal and bi-normal vectors) associated with the current fragment;
- use VD' (the transformed VD) and A, the (s,t) texture coordinates of the fragment, to compute B, the (u,v) texture coordinates where the ray reaches the depth value 1.0 (see Figure 7);
- compute the intersection between VD' and the height-field surface using a binary search starting with A and B;
- perform the shading of the fragment using the attributes (*e.g.*, normal, depth, color, etc.) associated with the texture coordinates of the computed intersection point.

This process is illustrated in Figure 7. Point A has an associated depth equal to zero, while B has depth equal to 1.0. At each step, one computes the midpoint of the current interval and assigns it the average depth and texture coordinates of the endpoints. In the example shown in Figure 7, the circle marked "1" represents the first midpoint. The averaged texture coordinates are used to access the depth map. If the stored depth is smaller than the computed one, the point along the ray is inside the height field surface, as in the case of point 1 in Figure 7. The binary search proceeds with one endpoint inside and other outside the surface. In the example shown in Figure 7, the numbers indicate the order in which the midpoints are obtained. In practice, we have found that eight steps of binary subdivision is sufficient to produce very satisfactory results. This is equivalent to subdivide the depth range of the height field in $2^8 = 256$ equally spaced intervals. Other researchers have used 64 axis-aligned equally-spaced 2D texture slices to render displacement maps using 3D textures [Meyer and Neyret 1998; Kautz and Seidel 2001]. The reader should also notice that our approach takes advantage of texture interpolation. Thus, while in techniques based on 3D texture mapping one may see in between slices, our technique does not suffer from this problem. As the depth map is treated and accessed as a texture, texture filtering (*e.g.*, bilinear) guarantees that the height-field surface will be continuous. As a result, the proposed technique can be used to produce extreme close-up views of the surface without noticeable artifacts (see Figures 16 and 17).

The binary search procedure just described may lead to incorrect results if the viewing ray intersects the height field surfaces in more than one point, as illustrated in Figure 8. In this example, the depth value associated with the first midpoint has a depth value smaller

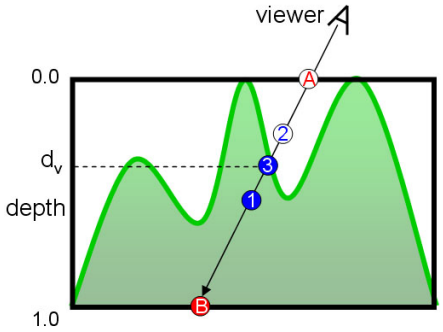

Figure 7: Ray intersection with a height-field surface using binary search. Starting with A and B, the numbers indicate the sequence of computed midpoints.

than the one retrieved from the depth map. Since the point is above the height field surface, the binary search would continue its way going deeper into the bounding box and find point 3 as the intersection, which is clearly incorrect. In order to avoid missing the first intersection, we start the process with a linear search. Beginning at point A, we step along the AB line at increments of δ times the length of AB looking for the first point inside the surface (Figure 9). If the graphics card supports shading model 3.0, δ varies from fragment to fragment as function of the angle between VD' and the interpolated surface normal at the fragment. As this angle grows, the value of δ decreases. In our current implementation, no more than 32 steps are taken along the segment AB. Notice that since the linear search does not involve any dependent texture accesses, this process is very fast as we can make several texture fetches in parallel.

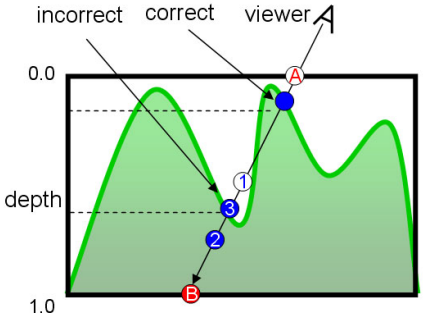

Figure 8: Problem with binary search. The midpoint between A and B is outside the height-field surface but the viewing ray has already pierced the surface.

Once the first point under the height field surface has been identified, the binary search starts using the last point outside the surface and current one. In this case, a smaller number of binary subdivisions is needed. For example, if the depth interval between two linearly searched points is 1/8, a six-step binary search will be equivalent to subdividing the interval into 512 ($2^3 \times 2^6$) equally spaced intervals.

3.1 Surface Self-Shadowing

Rendering shadows is a visibility problem [Williams 1978]. Therefore, a similar procedure can be used to determine whether a fragment is lit or in shade. In this case, we check if the light ray intersects the height-field surface between point C and the actual point being shaded (Figure 10). In case an intersection exists, the point must be in shade. Notice that there is no need to find the actual intersection point, but simply decide whether such an intersection exists, which can also be done using a similar strategy. Figure 14 and 19(c) show examples of relief renderings containing self-shadowing.

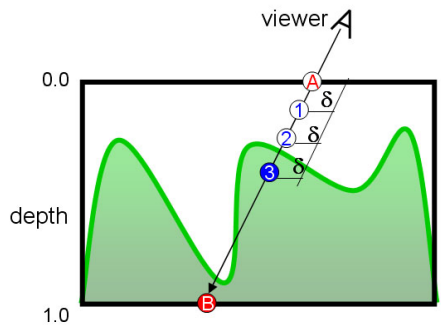

Figure 9: Linear search, from A to B, for the first point inside the height-field surface.

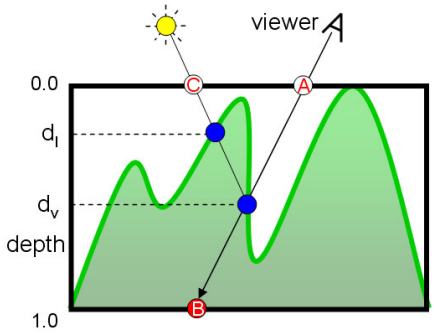

Figure 10: Shadow computation. One needs to decide if the light ray intersects the height-field surface between point C and the point where the viewing ray first hits the surface.

4 Dual-Depth Relief Textures

This section introduces an extension to relief textures that uses two layers of depth information. Such an extension, called *dual-depth relief textures*, can be used to produce approximate representations for opaque, closed-surface objects using only one relief-mapped polygon.

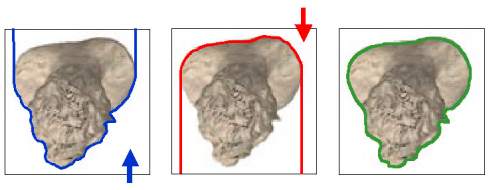

Figure 11: Dual-depth relief textures. The combined use of front and back depth layers produces tight bounds for an object representation.

As one tries to sample an object using a single relief texture, not enough information will be available to produce a proper reconstruction. In particular, no information will exist about what lays behind the object (Figure 11 left). In these cases, inverse rendering techniques may extend the ends of these surfaces forming "skins" [McMillan 1997]. The occurrence of skins can be eliminated with the use of one extra layer of depth that represents the back of the object (Figure 11 (center)). The combined effect of the two depth layers produces a much tighter boundary for the object (Figure 11 (right)) and leads to better quality renderings.

Notice that this representation is not exactly a layered-depth image (LDI) [Shade et al. 1998]: the two layers of depth are computed as orthographic distances measured with respect to one of the faces of the depth bounding box and it does not store color information. Moreover, the second depth layer is not used directly for rendering, but for constraining the search for ray-height-field intersections. Like other impostor techniques, this representation is not intended to be seen from arbitrary viewpoints. However, we show that they can be used for quite large range of angles.

The two depth maps and the normals can be stored in a single texture. Since all normals are unit length, we can store only the x and y components in the normal map, using the other two channels to represent the two depth layers. The z component of the normal can be recovered in the shader as $z = \sqrt{1-(x^2+y^2)}$. Figure 12 shows dual-depth maps for two models: angel (top) and Christ (bottom). The depth values of both layers are defined with respect to the same reference plane. In Figure 12, the maps on the left represent the front of the object, while the ones on the right represent the back surface. The rendering process using two depth layers is similar to what was described in Section 3. In this case, however, a point is considered inside the represented object if *front depth* $\leq$ *point depth* $\leq$ *back depth*.

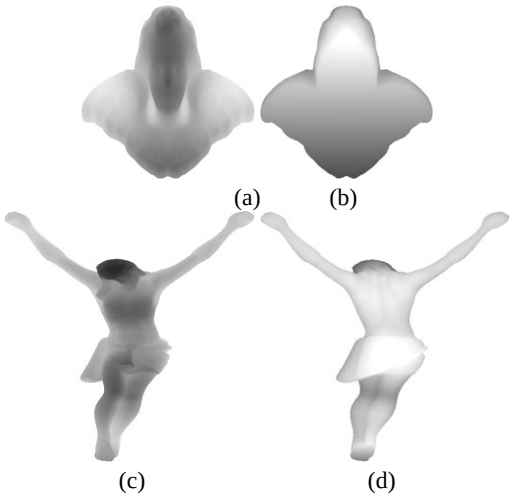

Figure 12: Dual-depth maps. Front (left) and back (right) layers. The top row samples an angel model. The bottom row is the sampling of a Christ model.

Figure 13 compares the renderings of the sampled objects shown in Figure 12 using single and dual-depth relief textures. In the case of single, only the front depth was used. In all cases, the images were created by rendering a single texture-mapped polygon. On the left, one sees the renderings produced with the use of a single depth layer. Notice the existence of skins on the angel's hair and over its right wing, and on Christ's hair and silhouettes. The Christ image was cropped to focus on some details.

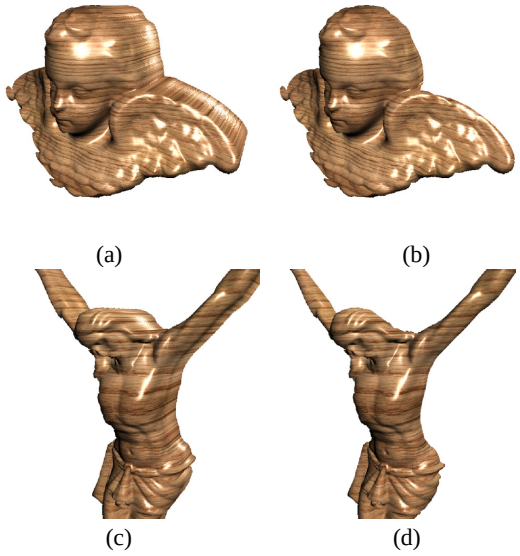

Figure 13: Renderings produced using single (left) and dual-depth (right) relief textures. Notice the existence of skins on the left images, which are not present in the right ones. A 2D wooden texture was added to the models.

5 Results

We have implemented the techniques described in the paper as fragment programs written in Cg and used them to map surface details to several polygonal objects. The mapping process is straightforward, using the texture coordinates and the tangent, normal and binormal vectors associated to the vertices of the model. Except for the rock relief texture used to render Bronto, the character shown in Figures 2 and 15, and the teapots in Figures 1 (right) and 17, all textures used in the paper were 512x512 RGBα textures. This includes the dual-depth relief representations. The stone texture used for Bronto was a 256x256 texture. The depth maps were quantized using 8 bits per texel. The quantized values represent evenly spaced depths, which can be scaled during the rendering using a parameter of the shader. All scenes were rendered at a resolution of 800x600 pixels at 85 frames per second, which is the refresh rate of our monitor. These measurements were made on a 3 GHz PC with 512 MB of memory using a GeForce FX6800 GT with 256 MB of memory.

Figure 1 shows the Utah teapot rendered using three different relief textures with per-pixel shading and shadows. Figure 2 shows a scene where all surfaces details for the character, the teapot and the brick walls were produced using relief mapping. The relief mapped objects naturally integrate themselves with the other elements of the scene. Notice the shadows on the teapot, which are cast by its own relief. Figure 14 shows a closer view of the same teapot, where all the surface details can be appreciated. Notice the correct shadows and occlusions due to parallax. The relief used for the teapot was obtained from the depth map shown in Figure 3.

Figure 15 shows another view of Bronto with its stone texture. Note how the texture correctly adjusts itself to the polygonal surface, producing a very realistic look. The relief details are emphasized by the per-pixel lighting.

Figure 19 compares the renderings of a single polygon with the data shown in Figure 6 using three different techniques. This height field contains both smooth and sharp features and tests the ability of the

Figure 14: A close view of the same teapot from the scene shown in Figure 2. Note the shadows. The relief used for the teapot was obtained from the depth map shown in Figure 3.

Figure 15: Bronto, a game character rendered using relief mapping.

techniques to correctly represent surface details. The images were created from the same viewpoint using: (a) bump mapping, (b) parallax mapping and (c) relief mapping with self-shadowing. Notice how relief mapping succeeds in producing a correct image for this object, while both bump and parallax mapping fail. The images produced by these techniques present some flatening. In Figure 19 (c) one can also observe the shadows properly cast by the surface relief. The accompanying videos provide some more examples of scenes containing shadows and per-pixel lighting recorded in real time.

Figure 16 and 17 show two extreme close-ups of relief mapped surfaces. The resolutions of the textures used to produce these renderings are 512x512 and 256x256, respectively. Notice how sharp these close-up images are. Correct interpenetration of relief-mapped surfaces (*e.g.*, involving multiple relief-textured objects) can be achieved by appropriately modulating the Z-buffer. The 3D coordinates associated with a fragment corresponding to the polygonal surface are available to the fragment program. From this, we can compute the 3D position of the intersected point at the surface height field. We then compute its corresponding depth value to test and/or update the Z-buffer. Figure 18 shows a relief mapped surface interpenetrated by three textured spheres. Notice the correct interpenetration boundaries.

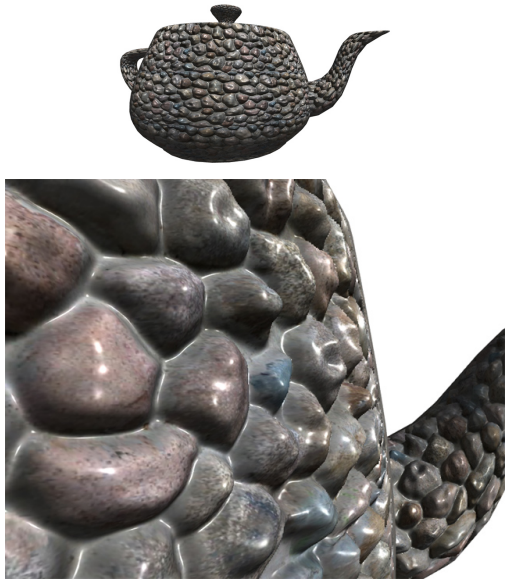

Figure 16: Relief texture-mapped teapot (top). Close-up view produced with a 512x512 stone relief texture (bottom).

If a feature represented by a height field is too thin, it might be possible that the linear search misses it. This is an aliasing problem for which there is no perfect solution. In practice, although we have modeled and rendered sharp features, we have not encountered such artifacts.

Since we use mip mapping [Williams 1983], texture minification causes the resampling of the height-field surface to be done on a filtered version of the depth map. While this is not strictly correct, it saves us the need to create a separate version of the depth map for each level of the mip map (which would then have to be interpolated anyway) and helps to reduce animation aliasing artifacts. It also improves rendering performance, as the use of smaller textures tends to increase cache coherence.

6 Conclusion

We have presented a technique for mapping relief textures onto arbitrary polygonal models in real time. The mapping is done in tangent space [Peercy et al. 1997], guaranteeing its generality and applicability to deforming surfaces. The technique produces correct self-occlusions, interpenetrations, shadows and all standard per-pixel lighting and filtering effects. We also described an efficient algorithm for finding the intersection between a ray and a height field based on binary search. Since the depth maps representing the heights undergo texture filtering, the resampled height-field surface is continuous, allowing the user to obtain extreme close-up views of the surface details. The compactness and ease of implementation of the presented technique make it an attractive alternative for games.

We have also extended relief textures with dual-depth maps. This new representation allows for approximate representations of 3D objects using a single texture. Renderings of these objects can be

(a) (b) (c)

Figure 19: One polygon rendered from the same viewpoint using three different techniques: (a) Bump mapping, (b) Parallax mapping and (c) Relief mapping with self-shadowing. A 2D wooden texture was mapped to the surface.

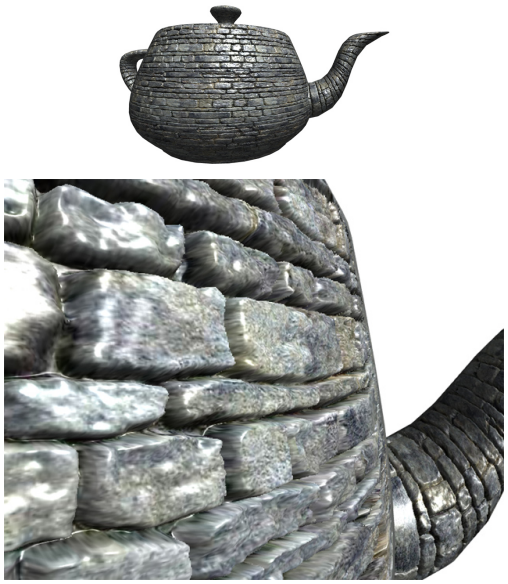

Figure 17: Relief texture-mapped teapot (top). Close-up view produced with a 256x256 rock relief texture (bottom).

generated rendering a single polygon, while producing dynamic impostors that can be used for a considerable angular range.

Our current implementation still does not add details to the underlying object's silhouette. This is a valuable feature to enhance realism and we are exploring ways of changing the silhouettes to correctly represent these details.

References

BLINN, J. F. 1978. Simulation of wrinkled surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, ACM Press, 286–292.

CATMULL, E. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, Salt Lake City, Utah.

COOK, R. L. 1984. Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM Press, 223–231.

ELHELW, M. A., AND YANG, G.-Z. 2003. Cylindrical relief texture mapping. *Journal of WSCG 11* (feb).

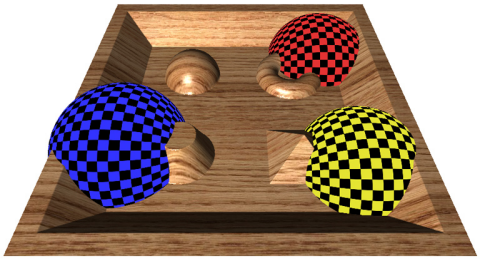

Figure 18: Interpenetration among a relief mapped surface and some textured spheres. Notice the correct boundaries.

GUMHOLD, S. 2003. Splatting illuminated ellipsoids with depth correction. In *8th International Fall Workshop on Vision, Modelling and Visualization*, 245–252.

HEIDRICH, W., AND SEIDEL, H.-P. 1998. Ray-tracing procedural displacement shaders. In *Graphics Interface*, 8–16.

HEIDRICH, W., DAUBERT, K., KAUTZ, J., AND SEIDEL, H.-P. 2000. Illuminating micro geometry based on precomputed visibility. In *Siggraph 2000, Computer Graphics Proc.*, 455–464.

HIRCHE, J., EHLERT, A., GUTHE, S., AND DOGGETT, M. 2004. Hardware accelerated per-pixel displacement mapping. In *Graphics Interface*, 153 – 158.

KANEKO, T., TAKAHEI, T., INAMI, M., KAWAKAMI, N., YANAGIDA, Y., MAEDA, T., AND TACHI:, S. 2001. Detailed shape representation with parallax mapping. In *Proceedings of the ICAT 2001*, 205–208.

KAUTZ, J., AND SEIDEL, H.-P. 2001. Hardware accelerated displacement mapping for image based rendering. In *Proceedings of Graphics Interface 2001*, 61–70.

MALZBENDER, T., GELB, D., AND WOLTERS, H. 2001. Polynomial texture maps. In *Siggraph 2001, Computer Graphics Proceedings*, 519–528.

MAX, N. 1988. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer 4*, 2, 109–117.

MCMILLAN, L. 1997. *An Image-Based Approach to Three-Dimensional Computer Graphics*. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC.

MEYER, A., AND NEYRET, F. 1998. Interactive volumetric textures. In *Eurographics Rendering Workshop 1998*, 157–168.

OLIVEIRA, M. M., BISHOP, G., AND MCALLISTER, D. 2000. Relief texture mapping. In *Siggraph 2000, Computer Graphics Proceedings*, 359–368.

PATTERSON, J., HOGGAR, S., AND LOGIE, J. 1991. Inverse displacement mapping. *Comp. Graphics Forum 10*, 2, 129–139.

PEERCY, M., AIREY, J., AND CABRAL, B. 1997. Efficient bump mapping hardware. In *SIGGRAPH '97*, 303–306.

PHARR, M., AND HANRAHAN, P. 1996. Geometry caching for ray-tracing displacement maps. In *Eurographics Rendering Workshop 1996*, Springer Wien, 31–40.

SCHAUFLER, G., AND PRIGLINGER, M. 1999. Efficient displacement mapping by image warping. In *Eurographics Rendering Workshop 1998*, Springer Wein, 175–186.

SHADE, J. W., GORTLER, S. J., HE, L.-W., AND SZELISKI, R. 1998. Layered depth images. In *Siggraph 1998, Computer Graphics Proceedings*, 231–242.

SLOAN, P.-P. J., AND COHEN, M. F. 2000. Interactive horizon mapping. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, Springer-Verlag, 281–286.

SMITS, B. E., SHIRLEY, P., AND STARK, M. M. 2000. Direct ray tracing of displacement mapped triangles. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, Springer-Verlag, 307–318.

WANG, L., WANG, X., TONG, X., LIN, S., HU, S., GUO, B., AND SHUM, H.-Y. 2003. View-dependent displacement mapping. *ACM Trans. Graph. 22*, 3, 334–339.

WANG, X., TONG, X., LIN, S., HU, S., GUO, B., AND SHUM, H.-Y. 2004. Generalized displacement maps. In *Eurographics Symposium on Rendering 2004*, EUROGRAPHICS, 227–233.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. In *Siggraph 1978, Computer Graphics Proceedings*, 270–274.

WILLIAMS, L. 1983. Pyramidal parametrics. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, ACM Press, 1–11.

Appendix: Fragment Shaders in Cg

```
f2s main_frag_relief( v2f IN,
      uniform sampler2D rmtex:TEXUNIT0, // rm texture map
      uniform sampler2D colortex:TEXUNIT1, // color texture map
      uniform float4 lightpos, // light position in view space
      uniform float4 ambient, // ambient color
      uniform float4 diffuse, // diffuse color
      uniform float4 specular, // specular color
      uniform float2 planes, // near and far planes info
      uniform float tile, // tile factor
      uniform float depth) // scale factor for height-field depth
{
      f2s OUT;
      float4 t,c; float3 p,v,l,s; float2 dp,ds,uv; float d;
      // ray intersect in view direction
      p = IN.vpos; // pixel position in eye space
      v = normalize(p); // view vector in eye space
      // view vector in tangent space
      s = normalize(float3(dot(v,IN.tangent.xyz),
            dot(v,IN.binormal.xyz),dot(IN.normal,-v)));
      // size and start position of search in texture space
      ds = s.xy*depth/s.z;
      dp = IN.texcoord*tile;
      // get intersection distance
      d = ray_intersect_rm(rmtex,dp,ds);
      // get normal and color at intersection point
      uv=dp+ds*d;
      t=tex2D(rmtex,uv);
      c=tex2D(colortex,uv);
      t.xyz=t.xyz*2.0-1.0; // expand normal to eye space
      t.xyz=normalize(t.x*IN.tangent.xyz+
            t.y*IN.binormal.xyz+t.z*IN.normal);
      // compute light direction
      p += v*d*s.z;
      l=normalize(p-lightpos.xyz);
#ifdef RM_DEPTHCORRECT
      // planes.x=-far/(far-near); planes.y =-far*near/(far-near);
      OUT.depth=((planes.x*p.z+planes.y)/-p.z);
#endif
      // compute diffuse and specular terms
      float att=saturate(dot(-l,IN.normal));
      float diff=saturate(dot(-l,t.xyz));
      float spec=saturate(dot(normalize(-l-v),t.xyz));
      float4 finalcolor=ambient*c;
#ifdef RM_SHADOWS
      // ray intersect in light direction
      dp+= ds*d; // update position in texture space
      // light direction in texture space
      s = normalize(float3(dot(l,IN.tangent.xyz),
            dot(l,IN.binormal.xyz),dot(IN.normal,-l)));
      ds = s.xy*depth/s.z;
      dp-= ds*d; // entry point for light ray in texture space
      // get intresection distance from light ray
      float dl = ray_intersect_rm(rmtex,dp,ds.xy);
      if (dl<d-0.05) // if pixel in shadow
            { diff*=dot(ambient.xyz,float3(1.0,1.0,1.0))*0.3333; spec=0; }
#endif
      finalcolor.xyz+=att*(c.xyz*diffuse.xyz*diff+
            specular.xyz*pow(spec,specular.w));
      finalcolor.w=1.0;
      OUT.color=finalcolor;
      return OUT;
}

float ray_intersect_rm(in sampler2D reliefmap, in float2 dp, in float2 ds)
{
      const int linear_search_steps=10;
      const int binary_search_steps=5;
      float depth_step=1.0/linear_search_steps;
      float size=depth_step; // current size of search window
      float depth=0.0; // current depth position
      // best match found (starts with last position 1.0)
      float best_depth=1.0;
      // search from front to back for first point inside the object
      for ( int i=0; i< linear_search_steps-1;i++){
            depth+=size;
            float4 t=tex2D(reliefmap,dp+ds*depth);
            if (best_depth>0.996) // if no depth found yet
            if (depth ≥ t.w)
                best_depth=depth; // store best depth
      }
      depth=best_depth;
      // search around first point (depth) for closest match
      for ( int i=0; i<binary_search_steps;i++) {
            size*=0.5;
            float4 t=tex2D(reliefmap,dp+ds*depth);
            if (depth≥ t.w) {
                best_depth = depth;
                depth -= 2*size;}
            depth+=size;
      }
      return best_depth;
}
```