Relief Mapping of Non-Height-Field Surface Details

Fabio Policarpo*
CheckMate Games

Manuel M. Oliveira†
Instituto de Informática
UFRGS

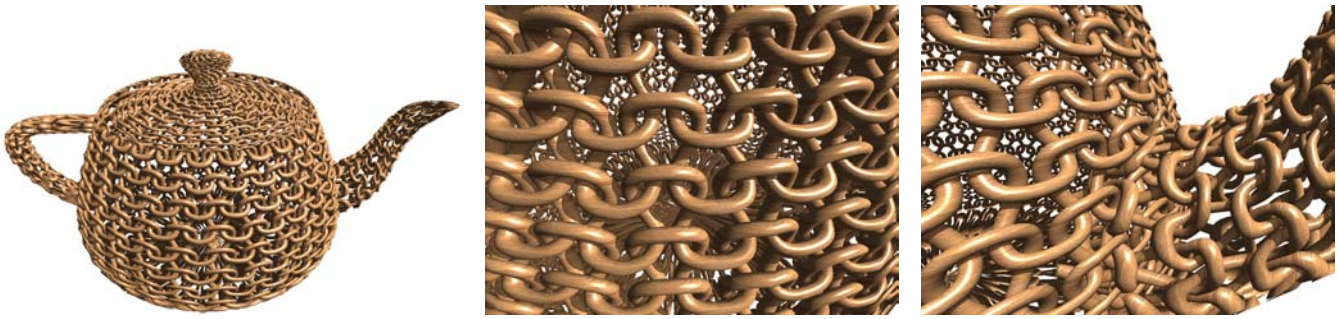


Figure 1: Relief mapping of non-height-field surface details. Teapot with a weave pattern (left). Close-up views of the teapot's body (center and right) reveal the back surface through the holes. Note the self-shadowing, occlusions and silhouettes.

Abstract

The ability to represent non-height-field mesostructure details is of great importance for rendering complex surface patterns, such as weave and multilayer structures. Currently, such representations are based on the use of 3D textures or large datasets of sampled data. While some of the 3D-texture-based approaches can achieve interactive performance, all these approaches require large amounts of memory. We present a technique for mapping non-height-field structures onto arbitrary polygonal models in real time, which has low memory requirements. It generalizes the notion of relief mapping to support multiple layers. This technique can be used to render realistic impostors of 3D objects that can be viewed from close proximity and from a wide angular range. Contrary to traditional impostors, these new one-polygon representations can be observed from both sides, producing correct parallax and views that are consistent with the observation of the 3D geometry they represent.

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

Keywords: real-time rendering, non-height-field surface representation, image-based rendering, relief mapping, impostors.

1 Introduction

Accurate representation of surface geometry is crucial for achieving the subtle shading effects commonly found in the real world. Unfortunately, explicitly representing all these details with polygons is impractical. As such, several techniques have been proposed to enrich the appearance of polygonal models by adding spatially-varying attributes such as color [Catmull 1974] and surface normals [Blinn 1978]. While powerful and extremely popular, these techniques cannot handle effects such as self-occlusions, self-shadowing or changes in the models' silhouettes, which are due to the fine-scale details they try to represent. These effects can be achieved by changing the actual geometry [Cook 1984], but it comes at the cost of rendering a larger number of extra polygons, which is undesirable for real-time applications. Moreover, displacement maps [Cook 1984] can only represent height-field structures and, therefore, is not a general representation for surface details.

This paper presents an image-based technique for mapping both height-field and non-height-field structures onto arbitrary polygonal models in real time. Compared to other techniques used to map non-height-field surface details [Meyer and Neyret 1998; Chen et al. 2004; Peng et al. 2004; Porumbescu et al. 2005; Wang et al. 2004], the proposed approach presents much lower memory requirements, is completely implemented on the GPU, exploiting the inherent parallelism of its vector units, and is easier to implement. Figure 1(left) shows a teapot mapped with a weave pattern rendered with our technique. Close-up views of the teapot's body revealing the details of the pattern are shown on the center and on the right. Note the proper occlusions, self-shadowing and silhouettes.

The main contributions of this paper include:

- A technique for mapping both height-field and non-height-field structures, represented as sets of 2D texture maps, onto arbitrary polygonal models in real time (Section 3).
- A new impostor representation for 3D objects (Section 4), which significantly improves the rendering quality and extends the lifetime of conventional impostors.

Section 2 discusses some related work. The details of the non-height-field mapping approach are described in Section 3. Section 4 discusses how to use the multilayer representation to render relief impostors. We present some of our results in Section 5, and compare the features of our algorithm with previous approaches in Section 6. Section 7 summarizes the paper.

*e-mail:fabio.policarpo@gmail.com
†e-mail:oliveira@inf.ufrgs.br

2 Related Work

In recent years, we have observed a growing interest for the use of non-height-field representations to model mesostructure details, and several techniques have been proposed to render them [Meyer and Neyret 1998; Chen et al. 2004; Peng et al. 2004; Wang et al. 2004; Porumbescu et al. 2005]. Meyer and Neyret [Meyer and Neyret 1998] use volumetric textures, which are rendered as a stack of 2D texture-mapped polygons along one of three possible directions. A similar approach was used in [Kautz and Seidel 2001]. Chen et al [Chen et al. 2004] treat a polygonal model as a shell layer around an inner homogeneous core. The shell represents the mesostructure and is filled using a texture synthesis approach that takes volumetric samples as input. This technique uses ray tracing for rendering and can produce high-quality images by simulating subsurface scattering, which our technique does not handle. However, the pre-processing time to create such representations is about 10 to 20 hours and the reported frame rates are not interactive.

Peng et al. [Peng et al. 2004] also use surface shells and volumetric textures to render surface details. Like in [Meyer and Neyret 1998], the rendering is performed by resampling the volume with a set of 2D polygons. The slices are regularly spaced across the volumetric texture and, as the geometric complexity of the represented mesostructures increases, a larger number of slices are required, which impacts the rendering performance. The memory requirements reported by the authors for storing a compressed version of a 512^3 volumetric texture is about 134MB [Peng et al. 2004].

Wang et al. [Wang et al. 2004] store pre-computed distances from each displaced point along many sampling viewing directions, resulting in a 5-D function that can be queried during rendering time. Due to their large sizes, these datasets need to be compressed before they can be stored in the graphics card memory. The approach is suitable for real-time rendering using a GPU and can produce nice results. However, in order to keep the size of these representations from growing too large, a low-resolution volumetric grid and a small number of viewing directions are used for sampling. Due to the pre-computed resolution, these representations are not intended for close-up views.

Porumbescu et al. [Porumbescu et al. 2005] use barycentric coordinates of points inside tetrahedra to define a mapping between points in a shell volume extruded from a polygonal surface and points in a volumetric texture. This technique uses ray tracing for rendering and can produce very impressive results, including refraction and caustics, which are not supported by our method. The technique, however, is not suitable for real-time applications.

Shade et al. [Shade et al. 1998] used perspective projection images with possibly multiple depth and color samples per pixel to minimize disocclusion artifacts in the context of 3D image warping.

2.1 Relief Mapping Review

Relief mapping [Policarpo et al. 2005] exploits the programmability of modern GPUs to implement a pixel-driven solution to relief texture mapping [Oliveira et al. 2000]. All the necessary information for adding surface details to polygonal surfaces is stored in RGBα textures. The RGB channels encode a normal map, while its alpha channel stores quantized depth information. The resulting representation can be used with any color texture.

A relief texture is mapped to a polygonal model using texture coordinates in the conventional way. Thus, the same mapping is used both for the relief and its associated color texture. The depth data is normalized to the $[0,1]$ range (it can be rescaled for rendering), and the implied height-field surface can be defined by a function $h : [0,1]x[0,1] \rightarrow [0,1]$. Let f be a fragment with texture coordinates (s,t) corresponding to a point on a polygonal surface. The rendering of fragment f can be described as follows:

- First, the viewing ray is transformed to f's tangent space. The ray enters the height field's bounding box BB at f's texture coordinates (s,t) (Figure 2 left).
- Let (u,v) be the texture coordinates of the point where the ray leaves BB. Such coordinates are obtained from (s,t) and from the ray direction. The actual search for the intersection is performed in 2D (Figure 2 right). Starting at coordinates (s,t), the texture is sampled along the line toward (u,v) until one finds a depth value smaller than the current depth along the viewing ray, or we reach (u,v);
- The coordinates of the intersection point are refined using a binary search, and then used to sample the normal map and the color texture.

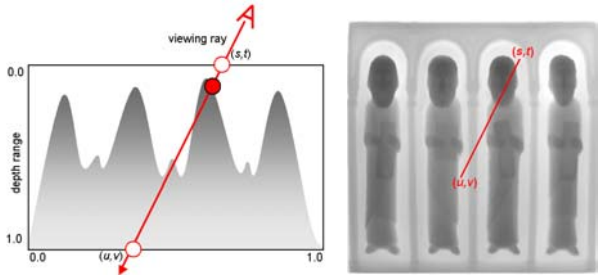


Figure 2: Ray intersection with a height-field surface (left). The search is performed in 2D, starting at texture coordinates (s,t) and proceeding until one reaches a depth value smaller than the current depth along the viewing ray, or until (u,v) is reached (right).

Self-shadowing can be computed in a straightforward way. A light ray is defined from the light source to the position of the first intersection of the viewing ray with the height-field surface. Thus, the question of whether a fragment should be lit or in shade is reduced to checking whether the light ray intersects any surface before reaching the point under consideration.

Figure 3: Detail of the teapot silhouette rendered with the technique described in [Oliveira and Policarpo 2005]. A wireframe representation of the polygonal model is shown for reference.

Oliveira and Policarpo [Oliveira and Policarpo 2005] extended the technique to render silhouettes implied by the relief data. For this, the attributes of each vertex of the polygonal model are enhanced with two coefficients, a and b, representing a quadric surface ($z = ax^2 + by^2$) that locally approximates the object's geometry at the vertex. Such coefficients are computed off-line using least-squares fitting and are interpolated during rasterization. Figure 3

illustrates the silhouette rendering produced by this technique, with a superimposed wireframe representation of the original polygonal model for reference.

3 Relief Mapping of Non-Height-Field Structures

The algorithms described in Section 2.1 can be extended to handle non-height-field representations in a rather straightforward way. Conceptually, this is done by: (i) representing several layers of depth and normals in the same domain and range (*i.e.*, $[0,1]x[0,1] \rightarrow [0,1]$) used before, and (ii) adapting the ray-surface intersection procedure to handle an arbitrary number of layers. This situation is illustrated in Figure 4, where a relief texture element (*relief texel*) may contain several pairs of depth and normal values. In the example shown in Figure 4, texel (α, β) stores four depth values, $\{d_i, d_j, d_k, d_n\}$, and their corresponding normal vectors. Texel (γ, δ), in turn, stores only two such pairs. Next, we describe how these multilayer representations can be efficiently handled by GPUs.

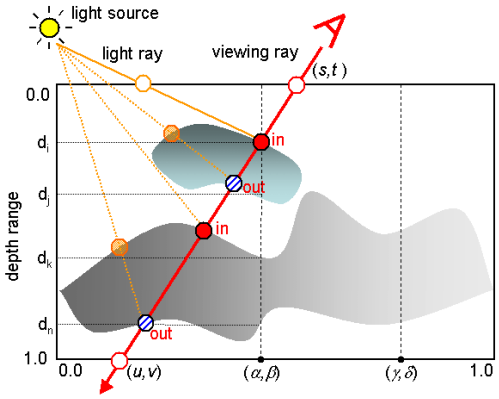


Figure 4: Ray intersection with a non-height-field closed surface.

The non-height-field surfaces are represented using 2D RGBα textures. For efficiency reasons, instead of using a separated relief representation, as defined in Section 2.1, for each layer, we use different textures to store depth and normal data. This way, each texel of a depth map can store up to four depth values. This is an important point because the ray-surface intersection is performed against the depth map. The normal map is sampled for shading, but only after an intersection has been found. Thus, one can check for ray-surface intersections with up to four layers in parallel, exploiting the vector processing capabilities of modern GPUs.

We will explain the non-height-field rendering process for the case of four-layer structures. Handling a different number of layers is similar. If the number of layers is not multiple of four, the extra channels of the depth texture are set to 1.0. Thus, for example, the depth values for texel (γ, δ) in Figure 4 are set to $\{d_p, d_q, 1.0, 1.0\}$. Using this convention, a height-field surface can be represented by setting the last three channels of the depth texture to 1.0, providing a uniform representation for both height-field and non-height-field surfaces. Figure 10 illustrates the mapping of a height-field surface on a polygonal model using this technique. Note, however, that if all surface details are height fields, one should prefer the single-layer representation described in [Policarpo et al. 2005], as in this case it would be more memory and computational efficient.

Figure 5 shows the multilayer representation of the weave pattern mapped on the teapot shown in Figure 1. The R, G, B and α channels of the first texture (Figure 5a) store the depth values associated with the four layers. The four channels of the second texture (Figure 5b) store the x components of the unit-length normal vectors for each of the layers, while their corresponding y components are stored in the third texture (Figure 5c). The z components of the normals are computed in the pixel shader as $z = \sqrt{1-(x^2+y^2)}$. Care should be taken because, due to representation errors, the stored 8-bit values for the x and y components of a normalized vector may lead to $(x^2+y^2) > 1.0$. Therefore, in practice z should be computed as $z = \sqrt{max(0, 1-(x^2+y^2))}$.

(a) (b) (c)

Figure 5: Representation of a four-layer non-height-field structure. (a) Depth values for the four layers stored in the RGBα channels of the texture. (b) x components of the unit normal vectors for the four layers. (c) y components of the unit normal vectors for the four layers.

3.1 Computing ray surface intersection

We assume that the non-height-field structures are defined by opaque closed surfaces, although they can have very complex topology and several connected components. This situation is illustrated in Figure 4, where the viewing ray intersects two disjoint components. Although a four-layer non-convex structure may possibly lead to more than four intersections along a given ray, we are only interested in the closest one. As in the case of the height-field version of relief mapping [Policarpo et al. 2005], the search for the closest intersection consists of two phases: a linear search followed by a binary-search refinement step. In the non-height-field approach, however, each step of the linear and binary searches checks for intersections against four layers in parallel. Once such an intersection is found, only the closest one is kept.

The parallel version of the linear search can be stated as follows: starting at coordinates (s,t), the depth texture is sampled along the line toward (u,v), using a step size (δ_s, δ_t). At step k, let d_k be the depth value corresponding to the current position along the viewing ray, and let $(s+k*\delta_s, t+k*\delta_t)$ be the current texture coordinates. In parallel, compare the current depth d_k with the four depth values stored at $(s+k*\delta_s, t+k*\delta_t)$ (see Figure 4). If d_k is bigger than an odd number of these depths, than the ray is inside a surface and the actual intersection point should be refined using the binary search; otherwise, the search proceeds. Algorithm 1 shows a pseudo code for the parallel version of the linear search.

The binary search starts where the linear search stopped. In case the viewing ray has pierced a surface, the current ray position is inside a surface; otherwise, it will already be close to the 1.0 limit of the normalized depth range (*i.e.*, the ray has moved forward at all steps of the linear search). Each step of the binary search consists of halving the texture and ray delta increments, sampling the depth texture and checking whether the current position along the viewing ray is outside or inside the closed surface. If it is outside,

the current texture and depth value along the viewing ray are incremented by the newly computed delta increments; otherwise, they are decremented. Algorithm 2 shows a pseudo code for the parallel version of the binary search.

```
// (δs, δt) is the linear increment in texture space
(si, vi) = (s,t) + (δs, δt); // initialize texture coords
dk = δdepth; // depth along viewing ray
for (step = 1; step < num_linear_steps; step++)
    float4 quad_depth = tex2D(quad_depth_map, (si, vi));
    float4 diff = dk - quad_depth;
    if ((diff[0]*diff[1]*diff[2]*diff[3]) > 0)
        // outside: move forward
        (si, vi)+ = (δs, δt); // increment in texture space
        dk+ = δdepth; // increment depth along viewing ray
```

Algorithm 1: Pseudo code for the parallel linear search

```
// (si, vi) stores the value computed by the linear search
//
for (step = 1; step < num_binary_steps; step++)
    (δs, δt) *= 0.5; // halves the texture space increment
    δdepth *= 0.5; // halves increment along the viewing ray
    float4 quad_depth = tex2D(quad_depth_map, (si, vi));
    float4 diff = dk - quad_depth;
    if ((diff[0]*diff[1]*diff[2]*diff[3]) > 0)
        (si, vi) += (δs, δt); // move forward
        dk += δdepth;
    else
        (si, vi) -= (δs, δt); // move backward
        dk -= δdepth;
```

Algorithm 2: Pseudo code for the parallel binary search

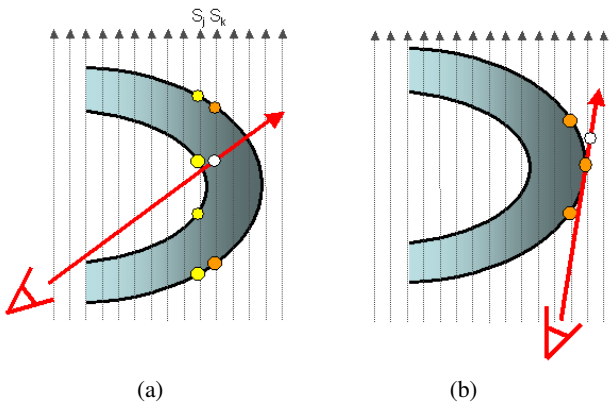
Depending on the viewing configuration and on the sampled geometry, it is possible that some rays do not intersect any surface. In this case, a base texture can be used to color the fragment with a different color, as can be seen in Figures 9(right) and 13(right). Alternatively, the fragment can be discarded, allowing the viewer to see through the underlying object, as in the examples shown in Figures 1 and 9(left).

Self-shadowing is computed in a similar way as described in Section 2.1. Once the closest intersection for a given viewing ray has been found, one computes a light ray and checks if it pierces any surface before hitting the given visible point (Figure 4).

Like the relief mapping approach described in [Policarpo et al. 2005], the non-height-field version of the technique is also prone to aliasing artifacts, which happen if the linear search skips some fine structures.

3.2 Rendering multilayer interfaces

Relief mapping non-height-field details requires that one pays special attention to the interfaces between relief texels storing different numbers of layers. This situation is depicted in Figure 6 where one sees the cross-section of a non-height-surface sampled by a set of parallel rays (indicated by the dotted arrows). The shaded dots indicate the intersections of some of these rays with the sampled surface. The number of intersections may vary from ray to ray. Without proper care, artifact may happen as we render the interfaces between texels representing different numbers of surface layers. Next, we discuss the problem and its solution.



(a) (b)

Figure 6: Multilayer interfaces. The viewing ray pierces the surface at a four-layer texel, but the sampling position (white dot) is at two-layer texel (a). Similar situation involving a two- and a zero-layer texels.

Figure 6(a) shows a configuration in which the viewing ray enters the surface through a four-layer texel sampled by ray S_j. Although the binary search tends to produce a very good approximation for the ray-surface intersection point, there is some error to this approximation. According to the pseudo code shown in Algorithm 2, such an error might cause the computed intersection to be either inside or outside the surface. Figure 6(a) illustrates this situation for the case the point is inside the surface and represented by a white dot. Although it is clear from this illustration that the inner-surface of the U-shape is the visible one, the resulting texture coordinate associated with the white dot will cause the normal map to be sampled at a position corresponding to ray S_k, which only stores normals for two intersection points. In this case, the shading would be performed using an incorrect normal (in this example, with the second normal stored at S_k), thus introducing some artifacts.

A similar situation is depicted in Figure 6(b), where the intersection was computed for a two-layer surface, but the resulting texture coordinates correspond to a texel storing zero layers. A careful examination of these cases reveals that this kind of problem can happen every time the viewing ray crosses an interface from a region with more layers to another region with less layers. In practice, these potential artifacts are avoided by keeping the last pairs of inside and outside (if any) texture coordinates visited during the binary search, and returning the one associated with the larger number of layers.

4 Relief-Mapped Impostors

Impostors [Maciel and Shirley 1995] are simplified representations of complex geometry rendered as billboards. Traditionally, parts of the scene are rendered into textures, which are then mapped onto quadrilaterals [Forsyth 2001]. Impostors are often created on the fly, being correct only from a given viewpoint, but reused as long as their associated rendering errors are under a given threshold. Schaufler [Schaufler 1995] proposed a metric for deciding when impostors need to be regenerated.

In this section, we show that the non-height-field version of relief maps can be used to produce representations of 3D objects that can

be rendered as single relief-mapped polygons. These representations can be viewed from very close proximity and from a wide angular range. A unique feature of this new class of impostors is that they can be viewed from both sides of the polygon, producing correct parallax and views that are consistent with the observation of the 3D geometry they represent. Figure 11 shows several views of a dog impostor represented by the relief and color maps shown in Figure 7. Note the parallax effect. Viewing the impostor from both sides of the polygon can be accomplished by appropriately inverting the search direction (*i.e.*, from depth value 0.0 to value 1.0) while looking for the first intersection of the viewing ray with the depth layers. In order to keep a consistent view from both sides, we shift the depth values by 0.5 so that the plane of the polygon (originally with depth = 0.0) now passes through the middle of the object (depth = 0.5). Thus, let (s,t) and (u,v) be the start and end texture coordinates for performing the search in texture space (Figure 4). This shifting effect is accomplished by simply performing the search from $(s-0.5(u-s),t-0.5(v-t))$ to $(s+0.5(u-s),t+0.5(v-t))$.

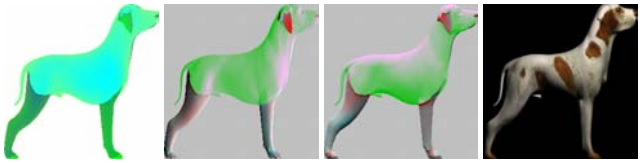


Figure 7: Representation of the four-layer dog impostor showed in Figure 11. (a) Depth values. (b) x components of the unit normal vectors. (c) y components of the unit normal vectors. (d) Color map.

(a) (b) (c)

Figure 8: Relief map representation of a horse: depth map (a) and normal maps (b) and (c).

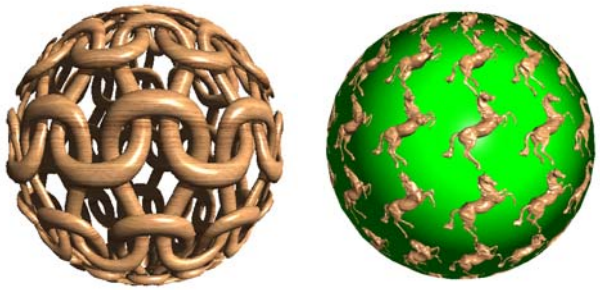


Figure 9: (left) A sphere mapped with the pattern shown in Figure 5. Note the silhouettes, self-shadowing and self-occlusion. (right) Another sphere tiled with the horse represented in Figure 8 without silhouette rendering. Note the parallax effect visible in the legs of the horses (right).

5 Results

We have implemented the techniques described in the paper as vertex and fragment programs written in Cg and used them to map both height-field and non-height-field surface details to several polygonal objects. The same programs can also render relief impostors. The multilayer relief maps were created using a ray-casting procedure implemented as a 3D Max plugin. The plugin renders parallel orthographic views of the models and saves depth and normal values at each ray-surface intersection. This sampling process is illustrated in Figure 6 by the set of parallel rays going from bottom to top. For all examples shown in the paper, the depth and normal textures were saved as tga RGBα (8 bits per channel) with resolution of 256×256 texels. Higher resolutions can be used for sampling finer structures. As before, the mapping process is straightforward, using the texture coordinates and the tangent, normal and binormal vectors associated to the vertices of the model. The same texture coordinates are used to access the depth, the normal, and the color texture maps.

Figure 9(left) shows a sphere mapped with the pattern shown in Figure 5. Note the silhouette, self-shadowing and self-occlusions. The silhouettes were rendered using the piecewise-quadric approximation described in [Oliveira and Policarpo 2005]. Figure 9(right) shows another sphere with the horse representation shown in Figure 8 tiled over its surface. For this example, the fragments for which the viewing rays miss the mesostructure were assigned a green color and shaded using the normal vector of the underlying model. At each point on the sphere, the horse model is seen from a different perspective. One should note the proper occlusions and the parallax effect (see the legs and heads of the horses).

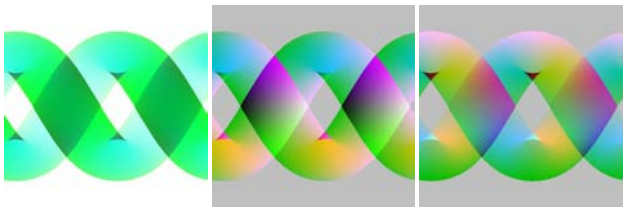
Figure 10 shows a cube mapped with a height-field surface, and illustrates the ability of the described technique to handle single-layer structures as well. In this case, the G, B and α channels of the depth map are set to 1.0.

Figure 10: Cube with a height-field surface mapped to it.

Figure 11 shows a relief impostor seen from different viewpoints. All these images were created by rendering the relief impostor shown in Figure 7 onto a single quadrilateral. Note the wide angular range supported by this kind of representation. A relief impostor, however, should not be rendered if the normal of its supporting quadrilateral approaches a direction perpendicular to the viewing ray. In such a configuration, the quadrilateral would approximately project onto a line on the screen, and not enough fragments would be available for the proper rendering of the impostor. A possible solution to this problem is to sample the model from three orthogonal directions, one of which is selected on-the-fly based on the viewing direction [Meyer and Neyret 1998], but this tends to introduce ”popping“ artifacts. Alternatively, the three orthogonal views can be combined to produce the final image, as done in the representation of 3D objects described in [Oliveira et al. 2000].

Figure 11: Multiple views of a relief impostor rendered as a single quadrilateral. The corresponding depth, normal and color maps are shown in Figure 7.

Figure 12 shows a relief map representation of a double-helix structure used to render the teapot shown in Figure 13(left). The other two teapots in Figure 13 were rendered using the the pattern shown in Figure 5. At the center, it exhibits silhouettes, whereas on the right, a second texture was used for the underlying model.



(a) (b) (c)

Figure 12: Relief map representation of a double helix. (a) Depth map. (b) x components of the unit normal vectors. (c) y components of the unit normal vectors.

The accompanying videos and illustrations used in the paper were rendered in real time using an Athlon64 4000 (2.4 GHz) PC with 2 GB of memory and a PCI express GeForce 7800 GT with 256 MB of memory. In order to evaluate the performance of our technique, we made a series of measurements. Each experiment consisted of mapping the multilayer relief map shown in Figure 5 onto an object using a 4×4 tiling factor and using a set of rendering options with a window with 640×480 pixels. Mip mapping was used only for the color textures. Depth and normal maps were sampled using nearest neighbors to avoid undesirable interpolation with non-surface texels. In each experiment, the object covered as many pixels of the window as possible, without having any part culled. Note that the sphere and the teapot have backfacing polygons visible through the holes defined by non-height-field pattern (see Figure 1). For those objects, better performance can be achieved by rendering in two passes. The first pass with backface culling and the second pass with frontface culling and taking advantage of early-z culling. For comparison, we have also measured the frame rates for mapping a height field (HF) to the same models using the technique described in [Policarpo et al. 2005]. Table 1 summarizes these results.

6 Discussion

The use of four layers is sufficient to represent a large class of non-height-field structures of interest. Representing structures with higher depth complexity can be obtained with the use of extra depth and normal textures. The intersection with the depth layers can be performed in parallel for groups of four layers at a time.

Table 1: Performance of the algorithm with different rendering options (in fps). All cases include per-fragment lighting. ML stands for multilayer and HF stands for height field.

	Plane	Sphere	Teapot
lighting only			
(ML) 1 pass / 2 passes	200 / 200	200 / 265	190 / 212
(HF)	480	530	470
self-shadowing			
(ML) 1 pass / 2 passes	127 / 127	136 / 182	120 / 140
(HF)	315	315	285
silhouette			
(ML) 1 pass / 2 passes	–	166 / 217	150 / 162
(HF)	–	440	380
self-shadow & silhouette			
(ML) 1 pass / 2 passes	–	110 /145	97 / 106
(HF)	–	305	240

While mip mapping [Williams 1983] can be directly applied continuous depth maps, such as the ones used in [Policarpo et al. 2005], textures used in the representation of non-height-field surfaces may contain many texels that do not correspond to surface points. Such texels are shown in white in Figures 5(a), 7(a), 8(a), and 12(a). For these cases, the use of regular mip map filtering produces incorrect results by introducing undesirable "skins" as it averages surface and non-surface texels. A simple approach to overcome this problem is to use a nearest-neighbors strategy to sample both the depth and normal maps, without using mipmapping. The color textures can be used with conventional mip-mapping filtering. Proper intersection with other objects in the scene is obtained by updating the z-buffer of each fragment with its perceived depth in camera space. A detailed description of this procedure can be found in [Oliveira and Policarpo 2005].

Table 2 compares some features of several techniques used for rendering non-height-field structures. All previous methods are based on the use of 3D textures or other volumetric representations. While these techniques can produce good quality images, they tend to require larges amounts of memory and most of them cannot achieve real-time frame rates. Shell Texture Functions [Chen et al. 2004] and the technique by Peng et al. [Peng et al. 2004] both require over 130MB. No memory requirements have been reported for Shell Maps [Porumbescu et al. 2005], but since its rendering is based on ray tracing, the technique is not appropriate for real-time applications. GDMs [Wang et al. 2004] are based on a 5D representation, which tend to be very large. By restricting the sampling of the viewing directions to $16x32$ and further compressing these datasets can considerably reduce their sizes. However, due to the low sampling resolution, these representations should be avoided if close-up

Figure 13: Teapot with different rendering options: (left) with the double-helix mesostructure shown in Figure 12. (center) With the pattern shown in Figure 5 with silhouettes. (right) With the pattern shown in Figure 5 and a second texture for the underlying model.

Table 2: Comparison of techniques for rendering non-height-field structures.

Technique	Represent.	Memory	Rendering / Speed
Relief Maps	2D Textures	$\sim$ 780KB (uncompr.)	Frag. shader Single pass Real time
Shell Maps	3D Textures or Geometry	Not informed	Ray tracing Non-interact.
Peng's et al.	3D texture	$\sim$ 130MB	Textured slices Interactive
Shell Texture Functions	Volume (voxels)	$\sim$ 150MB	Ray tracing Non-interact.
GDMs	5D function sampled on a volume	$\sim$ 4MB (compressed) 128x128x16	Multi-pass Real time

views of the surface details are required.

The proposed technique is based on 2D textures, requires considerably less memory than previous techniques (about 20% of what is required by a relatively low-resolution GDM - see Table 2), supports close-up views of the mapped details, and works in real time.

7 Conclusion

The ability to add non-height-field details to polygonal surfaces is of great importance for rendering object with complex mesostructures. We have extended the relief mapping technique to render non-height-field surface details in real time. This new technique stores multiple depth and normal values per texel and generalizes the relief mapping algorithm [Policarpo et al. 2005] for rendering height-field surfaces. Contrary to previous techniques, which are sensitive to the resolution used to perform a volumetric discretization of the space containing the non-height-field structures, the size of our representation grows with the depth complexity of the sampled structures.

The algorithm takes advantage of the vector processing capabilities of modern GPUs to check for the intersection of the viewing ray with up to four surface layers in parallel. Like its height-field counterpart, the described technique supports the rendering of self-shadowing, silhouettes, and interpenetrations. The represented structures can have complex topology with multiple connected components, and can cast and receive shadows into and from other objects of the scene. We have shown that the described technique has modest memory requirements when compared to previous approaches. The multilayer relief representations are easily acquired using ray casting, and the implementation of the technique is straightforward. The proposed technique seems an attractive alternative for rendering non-height-field structures and impostors in games. A shader for performing a basic rendering (*i.e.*, without self-shadowing, silhouettes or interpenetration) of non-height-field structures and relief impostors is listed in Appendix A.

Given the closed-surface assumption (see Section 3.1), surfaces with boundaries (*i.e.*, surfaces with holes) cannot be represented as a single-layer surface. In this case, two coincident layers need to be represented. This is similar to the common practice in computer graphics of using two polygons to represent a surface whose interior and exterior have different properties.

Acknowledgments

We would like to thank Alexandre MPB for modeling the non-height-field structures shown in Figures 1 and 13, and NVIDIA for donating the GeForce 7800 GT video card used in this work.

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.

CHEN, Y., TONG, X., WANG, J., LIN, S., GUO, B., AND SHUM, H.-Y. 2004. Shell texture functions. *ACM Transaction of Graphics - Proceedings of SIGGRAPH 2004 23*, 3 (August), 343–352.

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

FORSYTH, T. 2001. Impostors: Adding clutter. In *Game Programming Gems 2*, M. DeLoura, Ed. Charles River Media, 488–496.

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

MACIEL, P. W. C., AND SHIRLEY, P. 1995. Visual navigation of large environments using textured clusters. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, 95–102.

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

OLIVEIRA, M. M., AND POLICARPO, F. 2005. An efficient representation for surface details. Tech. Rep. RP-351, Federal University of Rio Grande do Sul - UFRGS, http://www.inf.ufrgs.br/∼oliveira/pubs_files/Oliveira_Policarpo_RP-351_Jan_2005.pdf, January.

OLIVEIRA, M. M., BISHOP, G., AND MCALLISTER, D. 2000. Relief texture mapping. In *Proceedings of SIGGRAPH 2000*, 359–368.

PENG, J., KRISTJANSSON, D., AND ZORIN, D. 2004. Interactive modeling of topologically complex geometric detail. *ACM Transaction of Graphics - Proceedings of SIGGRAPH 2004 23*, 3 (August), 635–643.

POLICARPO, F., OLIVEIRA, M. M., AND COMBA, J. 2005. Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of ACM Symposium on Interactive 3D Graphics and Games 2005*, ACM Press, 155–162.

PORUMBESCU, S., BUDGE, B., FENG, L., AND JOY, K. 2005. Shell maps. *ACM Transaction of Graphics - Proceedings of SIGGRAPH 2005 24*, 3 (July), 626–633.

SCHAUFLER, G. 1995. Dynamically generated impostors. In *GI Workshop on Modeling - Virtual Worlds - Distributed Graphics*, infix Verlag, D.W.Fellner, Ed., 129–135.

SHADE, J., GORTLER, S., WEI HE, L., AND SZELISKI, R. 1998. Layered depth images. In *Proceedings of SIGGRAPH 1998*, 231–242.

WANG, X., TONG, X., LIN, S., HU, S., GUO, B., AND SHUM, H.-Y. 2004. Generalized displacement maps. In *Eurographics Symposium on Rendering 2004*, 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 1983, Computer Graphics Proceedings*, 1–11.

Appendix A: Simple Shader for Relief Mapping of Non-Height-Field Details

```
void ray_intersect_rmqd_linear(
            in sampler2D quad_depth_map,
            inout float3 s, // texture position
            inout float3 ds) { // search vector
      const int linear_search_steps=10;
      ds/=linear_search_steps;
      for( int i=0;i<linear_search_steps-1;i++ ) {
            float4 t=tex2D(quad_depth_map,s.xy);
            float4 d=s.z-t; // compute distances to each layer
            d.xy*=d.zw; d.x*=d.y; // x=(x*z)*(y*w)
            if (d.x>0) s+=ds; // if ouside object move forward
      }
}
void ray_intersect_rmqd_binary(
            in sampler2D quad_depth_map,
            inout float3 s, // texture position
            inout float3 ds) {// search vector
      const int binary_search_steps=5;
      float3 ss=sign(ds.z);
      for( int i=0;i¡binary_search_steps;i++ ) {
            ds*=0.5; // half size at each step
            float4 t=tex2D(quad_depth_map,s.xy);
            float4 d=s.z-t; // compute distances to each layer
            d.xy*=d.zw; d.x*=d.y; // x=(x*z)*(y*w)
            if (d.x<0) { // if inside
                  ss=s; // store good return position
                  s-=2*ds; // move backward
            }
            s+=ds; // else move forward
      }
      s=ss;
}
float4 relief_map_quad_depth(
            float4 hpos : POSITION,
            float3 eye : TEXCOORD0,
            float3 light : TEXCOORD1,
            float2 texcoord : TEXCOORD2,
            uniform sampler2D quad_depth_map : TEXUNIT0,
            uniform sampler2D color_map : TEXUNIT1,
            uniform sampler2D normal_map_x : TEXUNIT2,
            uniform sampler2D normal_map_y : TEXUNIT3
            uniform float3 ambient,
            uniform float3 diffuse,
            uniform float4 specular,
            uniform float shine) : COLOR {
      float3 v=normalize (IN.eye); // view vector in tangent space
      float3 s=float3(IN.texcoord,0); // search start position
      // separate direction (front or back face)
      float dir=v.z; v.z=abs(v.z);
      // depth bias (1-(1-d)*(1-d))
      float d=depth*(2*v.z-v.z*v.z);
      // compute search vector
      v/=v.z;     v.xy*=d;     s.xy-=v.xy*0.5;
      // if viewing from backface
      if (dir<0) { s.z=0.996; v.z=-v.z; }
      // ray intersect quad depth map
      ray_intersect_rmqd_linear(quad_depth_map,s,v);
      ray_intersect_rmqd_binary(quad_depth_map,s,v);
      // discard if no intersection is found
      if (s.z>0.997) discard; if (s.z<0.003) discard;
      // get quad depth and color at intersection
      float4 t=tex2D(quad_depth_map,s.xy);
      float4 c=tex2D(color_map,s.xy);
      // get normal components X and Y
      float4 nx=tex2D(normal_map_x,s.xy);
      float4 ny=tex2D(normal_map_y,s.xy);
      // find min component of distances
      float4 z=abs(s.z-t);
      int m=0;    if (z.y<z.x)  m=1;
                  if (z.z<z[m]) m=2; if (z.w< z[m]) m=3;
      // get normal at min component layer
      float3 n; // normal vector
            n.x=nx[m]; n.y=1-ny[m];
            n.xy=n.xy*2-1; // expand to [-1,1] range
      n.z=sqrt(max(0,1.0-dot(n.xy,n.xy))); // normal z component
      if (m==1||m==3) n.z=-n.z; // invert normal Z if in back layer
      // compute light vector in view space
      float3 l=normalize(IN.light);
      // restore view direction z component
      v=normalize(IN.eye); v.z=-v.z;
      // compute diffuse and specular terms
      float ldotn=saturate(dot(l,n));
      float ndoth=saturate(dot(n,normalize(l-v)));
      // compute final color with lighting
      float4 finalcolor;
      finalcolor.xyz = c.xyz*ambient + ldotn*(c.xyz*diffuse +
                  c.w*specular.xyz*pow(ndoth,shine));
      finalcolor.w=1;
      return finalcolor;
}
```