Eurographics Symposium on Rendering (2007)
Jan Kautz and Sumanta Pattanaik (Editors)

Interactive Smooth and Curved Shell Mapping

Stefan Jeschke[†] Stephan Mantler[‡] Michael Wimmer[†]

[†]Vienna University of Technology [‡]VRVis Research Center

Abstract

Shell mapping is a technique to represent three-dimensional surface details. This is achieved by extruding the triangles of an existing mesh along their normals, and mapping a 3D function (e.g., a 3D texture) into the resulting prisms. Unfortunately, such a mapping is nonlinear. Previous approaches perform a piece-wise linear approximation by subdividing the prisms into tetrahedrons. However, such an approximation often leads to severe artifacts. In this paper we present a correct (i.e., smooth) mapping that does not rely on a decomposition into tetrahedrons. We present an efficient GPU ray casting algorithm which provides correct parallax, self-occlusion, and silhouettes, at the cost of longer rendering times. The new formulation also allows modeling shells with smooth curvatures using Coons patches within the prisms. Tangent continuity between adjacent prisms is guaranteed, while the mapping itself remains local, i.e. every curved prism content is modeled at runtime in the GPU without the need for any precomputation. This allows instantly replacing animated triangular meshes with prism-based shells.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Picture/Image Generation]: Display algorithms; I.3.7 [Three-Dimensional Graphics and Realism]: Color, shading, shadowing, and texture;

1. Introduction

Image-based representations have been used for a long time to simulate surface detail. Common examples are texture maps [BN76] for color, bump maps [Bli78] for enhanced shading, and displacement maps [CCC87] for fine-scale height field-like details. *Volume textures* by Kajiya and Kay [KK89] and more recently *shell maps* by Porumbescu et al. [PBFJ05]) add arbitrary small-scale surface detail to a triangulated object. The detail, called *texture function*, can be a sampled height field, a 3D texture, an analytical 3D function, or, in the case of shell maps [PBFJ05], even meshes. The idea is to construct a volume from a surface mesh by applying an extrusion along the vertex normals to obtain prisms. The texture function is then mapped into these prisms. This makes it possible to interactively render three-dimensional surfaces like bricks, fur, cloth, webbing, facades, ornaments, etc. but also complete 3D textural objects, with correct parallax, occlusions, silhouettes and shadows.

However, there is as yet no formulation that allows a *smooth* mapping between the surface detail definition space and its world-space representation. While some offline approaches provide a reasonable approximation through recursive subdivision [SSS00] for the special case of displacement mapping, recent approaches [PBFJ05, HEGD04] resort to piece-wise linear mappings induced by subdivision of the world-space prisms into tetrahedrons. These mappings lead to objectionable artifacts at the subdivision borders similar to the artifacts induced by non-perspective correct texture mapping (see Figure 1, left).

In this paper, we present a new formulation and implementation of a smooth mapping between the texture function and world space. Texture functions can be applied to geometry with high local curvature, and even very thick surfaces can be displayed without reducing image quality (see Figure 1, middle). The new formulation allows for the definition of smooth curved shells based on Coons patches (Figure 1, right). Our implementation of the mapping relies on ray casting on graphics hardware and provides interactive frame rates.

The main contributions in this paper are:

- A high-quality shell-mapping algorithm that avoids artifacts due to piece-wise linear approximations. This algo-

[†] {jeschke|wimmer}@cg.tuwien.ac.at
[‡] step@vrvis.at

© The Eurographics Association 2007.

delivered by EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

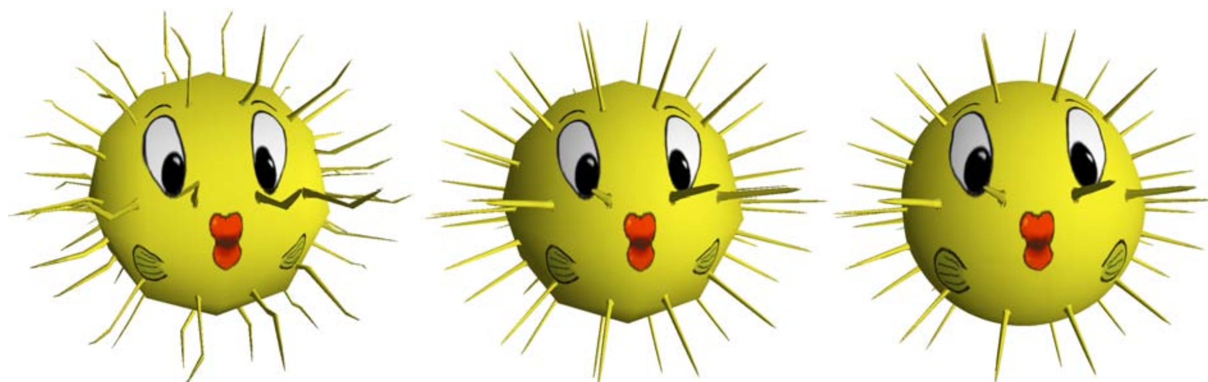

Figure 1: *Shell mapped blowfish. Left: previous shell mapping approach based on tetrahedra. Middle: smooth shell mapping. Right: curved shell mapping. Note that the mesh is the same in all three images.*

rithm gives pixel-correct results for the intended mapping, runs completely on graphics hardware and is interactive.

- Curved shell maps, an extension to smooth shell maps that provides tangent-continuity at the prism borders. This is achieved using a Coons-patch formulation that has previously been introduced for curved displacement mapping in offline rendering. Curved shell maps are slower than smooth shell maps, but still interactive.

The shell mapping methods do not require any preprocessing so that the mesh and the extrusion (i.e., normals) can be modified interactively. This fact makes smooth and curved shell maps a powerful modeling primitive. Both methods can simply replace a standard texture map without any other modifications to the base mesh.

2. Related Work

There are numerous methods for representing volumetric surface detail, including parallax mapping, displacement mapping, relief maps, and image slice-based methods. Rendering surface detail can in general be done by *forward mapping* polygonal representations or *backward mapping* texture-based representations using ray casting. In the following, we will focus the discussion on methods based on ray casting, as they are most closely related to the new approach.

Kajiya and Kay [KK89] first used ray casting through 3D texture maps to display fur. They define a shell volume over a surface and use mappings to transfer from texture space to shell space. Rays that intersect the shell are transformed to texture space and traced through the 3D texture. Neyret [Ney98] used a similar algorithm and rendered various types of objects using volumetric textures. In contrast, the *hypertexture* introduced by Perlin and Hoffert [PH89] renders functions instead of sampled data.

Several approaches [PHL91, HS98, PH96] are based on ray tracing displacement maps, which is computationally expensive and can still not be implemented on graphics hardware. Smits et al. [SSS00] already pointed out the problem of curved rays in texture space (see Section 4), but their algorithm did not address it.

View-dependent visibility information along rays can also be precomputed, as was demonstrated by Wang et al. for displacement maps [WWT*03] and arbitrary volumetric data [WTL*04]. The method provides impressive results (including indirect illumination, shadowing, and correct silhouettes), runs in real time on a GPU, but needs large amounts of memory for the sampled five-dimensional function. In addition, the approximated mesh curvature can lead to texture distortions and only works for static 3D textures.

Recently, performing ray casting on graphics hardware became popular. Policarpo et al. [POC05] presented *relief texture mapping*, which starts with rendering a textured mesh. For every rendered pixel, the entry point and view direction of the corresponding ray is transformed to texture space, which contains a height field. The ray is linearly traversed until it hits the height field, and a binary search is used to increase the intersection accuracy. In practice, the method shows visually pleasing results with correctly handled self-occlusions and real-time performance, and it is simple to implement. However, as for simple texture mapping, the object silhouette remains flat, and systematic artifacts are introduced between adjacent triangles.

In subsequent work, Oliveira and Policarpo [OP05] tried to correctly render silhouettes by locally approximating the object surface with a piecewise quadric representation at the fragment shader level. These quadrics are used as reference surfaces for ray traversal so that a ray can leave the mesh, resulting in better approximated object silhouettes. Unfortunately, the quadrics work only sufficiently correct for the direct neighborhood of the ray entry point. Figure 2 shows an example where this approximation is incorrect

© The Eurographics Association 2007.

for a torus, resulting in large texture distortions and even holes in the representation. In addition, interactively deforming the mesh is not straightforward because the quadrics must be precalculated from the mesh geometry. Most re-

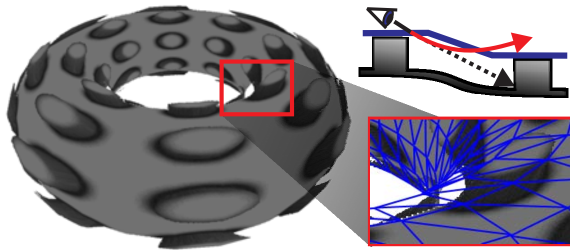

Figure 2: *Texture distortions and holes due to wrong surface curvature approximation.*

cently, Policarpo [PO06] presented a variation of his method that renders multiple image layers instead of height fields. However, since the work is based on the previous methods [POC05, OP05], it shares the same problems.

Hirche et al. [HEGD04] form prisms by extruding the base mesh. In contrast to this paper, every prism is decomposed into three tetrahedrons, with adjacent prism sides sharing the same triangulation in order to ensure a continuous volume. For each pixel of a rendered tetrahedron, the two intersections of the viewing ray with the tetrahedron are computed, transformed into texture space, and linear sampling is applied between them. The result is a linear bijective mapping between tetrahedrons and texture space which is an approximation of the "smooth" mapping described in this paper. Unfortunately, as Porumbescu et al. [PBFJ05] pointed out, the piecewise linear approximation systematically results in objectionable artifacts. These are visible as "bucklings" along the tetrahedron borders, as can be observed in Figure 3 as well as Figure 1. In addition, tetrahedrons can be inverted at long thin triangles, resulting in empty shell space. By rendering the prism geometry directly, our algorithm can avoid all such systematic errors, generate fewer primitives and consequently has less overdraw.

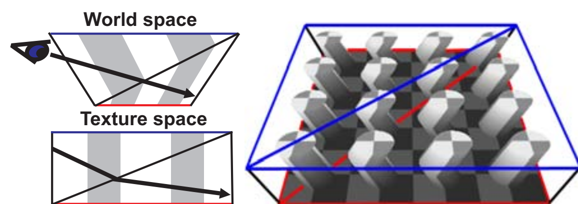

Figure 3: *Bucklings in a height field function due to a linear approximation of the prisms using tetrahedrons.*

Dufort et al. [DLP05] showed how the algorithm by Hirche et al. can be used for semi-transparent surfaces by a front-to-back composition of the tetrahedra. Porumbescu et al. [PBFJ05] introduced *shell maps*, which use the same tetrahedron-based (i.e., piece-wise linear) mapping as Hirche [Por06]. They show both forward mapping of geometry and backward mapping using a software ray tracer. However, since their mapping is still based on tetrahedrons, it shares the problems of Hirche et al. [HEGD04]. Note that for forward mapping of geometry-based surfaces, *mesh quilting* [ZHW*06] can greatly reduce distortions.

Recently, Baboud and Decoret [BD06] rendered whole objects on the GPU as a collection of images with depth. Orthogonal and perspective projections are supported, which are linear mappings between world space and texture space in contrast to shell mapping.

3. Shell Definition

In this section, we give a definition of the spaces involved in shell mapping, and describe the "correct" mapping function which is the basis for smooth shells. The term "correct" means here that the shell is defined along the interpolated vertex normals, which is the most intuitive and common definition. We will then extend the smooth mapping formulation to curved shell maps which provide tangent continuity between prisms.

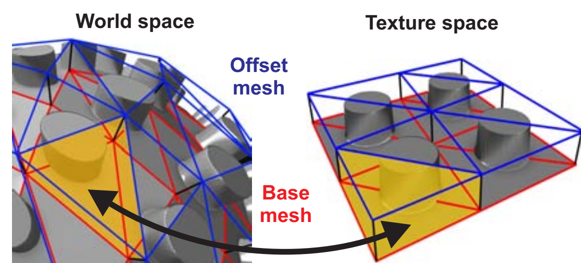

Figure 4: *Shell definition: world-space prisms correspond to prisms in texture space.*

Given a triangle mesh called *base mesh* (see Figure 4, left), a *shell* is constructed by extruding this mesh to an *offset mesh* along its normals at every vertex. The shell is then the union of all prisms formed by the base mesh triangles together with their offset mesh counterparts [WWT*03, HEGD04, PBFJ05]. Every prism has a corresponding prism in texture space (see Figure 4, right). Intersections between prisms pose no problem to the rendering algorithm and can be allowed similar to [PBFJ05].

3.1. Smooth Shell Mapping

Given a shell construction as above, the question is how to define a function Φ to map between a texture space prism and its corresponding world space prism. This will then be used to map the texture function to a shell. Previous approaches based the mapping on a tetrahedalization, basically

© The Eurographics Association 2007.

a piece-wise linear approximation without C^1 continuity at tetrahedron boundaries, leading to image artifacts. In contrast, we define a *smooth* mapping that is C^1 continuous within every prism, and C^0 continuous between prisms. The mapping is based on a displacement mapping function proposed by Smits et al. [SSS00], who used it to displace microtriangles of a mesh according to analytic functions. However, the mapping could also be seen as a straightforward application of Phong shading, as it is based on normal vector interpolation.

Let (u,v,w) be the coordinates of a texture-space point $\mathbf{p}_t$ with respect to the current texture-space prism (note that mapping from texture space to the current texture space prism is a trivial affine mapping). Let $\mathbf{p}_0,\mathbf{p}_1,\mathbf{p}_2$ be the world-space coordinates of the base triangle, and $\mathbf{n}_0,\mathbf{n}_1,\mathbf{n}_2$ the corresponding normal vectors. With $t = 1-u-v$, the mapping is given by interpreting (t,u,v) as the barycentric coordinates of a world-space point on the base triangle and extruding this point by w along the interpolated normal vector:

$$\mathbf{p} = \Phi(\mathbf{p}_t) = t\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2 + w(t\mathbf{n}_0 + u\mathbf{n}_1 + v\mathbf{n}_2) \quad (1)$$

For the texture-space prism given by $u,v,t \geq 0, 0 \leq w \leq 1$, this mapping gives exactly the prism in world space that was defined above (see Figure 5), which means that we have created a valid mapping. In general, this prism is twisted: The world-space prism is bounded by bilinear patches given by $u=0, v=0$ and $t=0$ respectively.

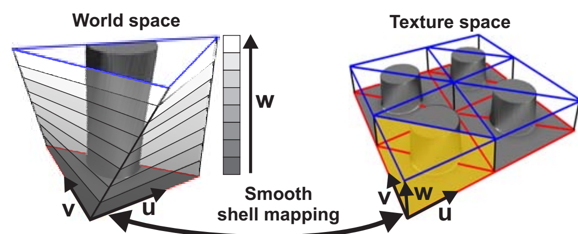

Figure 5: *World space and texture space correspondence for smooth shell mapping.*

Starting from a texture function $f(\mathbf{p}_t)$ in texture space, the *shell* itself can now be defined by mapping the texture function to world space using Φ:

$$f'(\mathbf{p}) = f(\Phi^{-1})(\mathbf{p}))$$

For this to work, Φ has to be a bijective mapping, which is the case as long as the normal vectors do not define a concave or even self-intersecting prism. As Φ is a smooth mapping, also the surface detail defined in the texture function will appear smooth in the prism as shown in Figure 5. However, note that Φ is not in general an easy function to invert. The solution requires finding the appropriate root of a third-order polynomial, which is possible but expensive.

Barycentric correspondence. In order to solve the problem of inverting Φ, we make the following observation which we call *barycentric correspondence*: for a fixed w, the mapping defines a triangle $T(w)$ within the prism given by the vertices $T(w) = \{\mathbf{p}_0 + w\mathbf{n}_0, \mathbf{p}_1 + w\mathbf{n}_1, \mathbf{p}_2 + w\mathbf{n}_2)\}$. Now the coordinates (u,v) of any point mapped into T, $\mathbf{p} = \Phi(u,v,w) \in T$ correspond *exactly* to the barycentric coordinates of $\mathbf{p}$ with respect to T. Conversely this means if we know the barycentric coordinates (u,v) of a point $\mathbf{p} \in T(w)$, we can conclude that $\Phi^{-1}(\mathbf{p}) = (u,v,w)$.

Therefore, for any given w, we can intersect a world-space ray with $T(w)$ and map the intersection point back to texture space just by calculating its barycentric coordinates. This correspondence is the basis for the rendering algorithm described in Section 4.

3.2. Curved Shell Mapping

Here we extend the smooth mapping to a *curved* mapping so that the resulting surface maintains tangent continuity at the prism boundaries. It is of essential importance for the rendering algorithm that the barycentric correspondence is also valid for the curved shells, so they must be defined over the interpolated vertex normals. This excludes Bezier surfaces and most other curved surface definitions. Instead, to the best of our knowledge, the only technique fulfilling this requirement is a construction based on Coons patches, previously used for displacing polygon vertices [SSS00]. Another advantage of Coons patches is that their construction is completely local to the base triangle, which makes them useful for dynamic mesh deformations without any precomputations. We briefly repeat the patch construction here (for details of the lengthy derivation see Smits et al. [SSM00]), and then describe how these patches can be integrated in our shell mapping framework.

We start with a triangle $T = \{P_1,P_2,P_3\}$ together with the corresponding normals and a point P at barycentric coordinates $(1-u-v,u,v)$ (see Figure 6, left). The Coons surface defines a displacement of P in direction of its interpolated normal vector. This displacement is constructed using Hermite interpolation as follows: starting at vertex P_1, curves are constructed to P_2 and P_3 by applying Hermite interpolation with the vertex normals. On each of the two curves, we construct a point and corresponding normal at the location defined by $1-u-v$. Between these two points we apply another Hermite interpolation at position $(1-u-v,u,v)$ (see Figure 6), resulting in a displacement value for P.

These steps are repeated for vertices P_2 and P_3, respectively. Finally, the three obtained displacement values are blended using either Boolean sums or again Hermite interpolation. Smits et al. [SSM00] use the resulting displacement $Coons_T(1-u-v,u,v)$ to modulate P along its normal.

Coons patch construction for shells. In order to define curved shell maps, for any given w we construct a

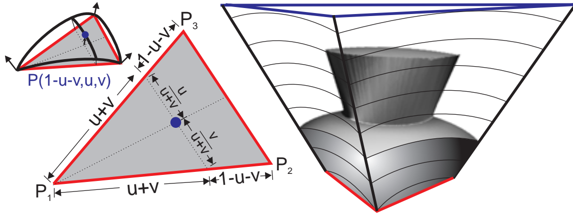

Figure 6: *Construction of a curved shell map in world space using Coons patches.*

coons patch for the corresponding world-space prism triangle $T(w)$. A straightforward application of the Coons patch to the triangle would have the problem that the constructed surface may protrude from the world-space prism. For curved shell maps, we therefore compress the texture function in height, giving a new height value $w' = w/2 + 1/4$. Note that this compression is heuristic, and does not guarantee that the shell stays within the world-space prism for extreme curvatures. In Section 7 we will show a general way how this problem can be solved given sufficiently fast hardware. We also denote the interpolated normal $n(u,v) = (1-u-v)\mathbf{n_0} + u\mathbf{n_1} + v\mathbf{n_2}$. Then we define the curved shell mapping function Φ_c as:

$$\Phi_c(u,v,w) = \Phi(u,v,w') + \frac{n(u,v)}{|n(u,v)|} Coons_{T(w')}(1-u-v,u,v)$$

Note that the Coons patch is not simply a distortion of texture space, since its definition depends on the world-space prism. The resulting family of curved surfaces defines a smoothly curved volume inside the prism (Figure 6), and also maintains tangent continuity to adjacent prisms at the prism boundaries.

Barycentric correspondence for curved shells. As for smooth shells, we face the problem of inverting Φ_c. Due to the construction the barycentric correspondence holds in a similar way as for smooth shells: for a certain height $\bar{w}$, the coordinates (u,v) of any point mapped to $T(\bar{w}, \mathbf{p}) = \Phi_c(u,v,w)$, correspond *exactly* to the barycentric coordinates (u,v) of $\mathbf{p}$ with respect to $T(\bar{w})$.

There is, however, a notable difference to the smooth case: from $\bar{w}$ we cannot immediately determine the texture space height value w that generated the point $\mathbf{p}$ so that it lies on $T(\bar{w})$. An exact solution to this problem consists in solving the equation $\Phi_c(u,v,w) - \Phi(u,v,\bar{w}) = 0$ for w. In practice, we determine w through linear interpolation between the bounding Coons patches at $w = 0$ and $w = 1$:

$$w = \frac{|\mathbf{p} - \Phi_c(u,v,0)|}{|\Phi_c(u,v,1) - \Phi_c(u,v,0)|} \qquad (2)$$

It can easily be shown that Equation 2 provides exact results for prism boundaries. While we have not found an analytic bound for the error towards the prism center, we have verified it for all our test models and the introduced error (manifesting itself in tangent continuity violations) never exceeded $\frac{1}{100}$ of the shell thickness, making it invisible in practice.

In conclusion,

$$\Phi_c^{-1}(\mathbf{p}) = (u,v,w),$$

where (u,v) is determined as for smooth shells, and w is determined by Equation 2.

4. Rendering Shells

Φ^{-1} is a non-linear mapping between prism and texture space (Section 3.1). Consequently, marching a ray through world space and evaluating the corresponding texture space samples is not feasible in graphics hardware, as Φ^{-1} involves finding the roots of a third-order polynomial at each step. Even if the analytic inverse mapping or any approximation (for instance, using polynomials) were used, ray marching acceleration techniques (see Section 4.4) could not be applied because they march the ray at defined distances in texture space. This would only be possible for the special case that Φ^{-1} is a linear mapping. Unfortunately, interactive framerates and a high image quality are hardly possible without such acceleration methods. On the other hand, adopting a coarse piece-wise linear approximation as in previous approaches leads to a non-exact solution and systematic artifacts, as demonstrated in Section 2. Instead, we march the ray *stepwise linearly* in texture space and *correct* the marching direction so as to not exceed a given error (e.g., one pixel).

The first step is to choose a geometric structure to represent shells on graphics hardware (Section 4.1). For each rasterized fragment of the geometric structure, a pixel shader will carry out ray casting operations into the shell. In doing so, first the ray vs. prism entry and exit points are computed (Section 4.2). Afterwards, the ray is sampled between these two points while keeping the introduced error below one pixel (Section 4.3). Section 4.4 shows how to keep the number of samples low for fast processing without reducing image quality. Finally, Section 4.5 shows how curved shells are sampled.

4.1. Choosing a Rendering Primitive

The primitive to be displayed has to cover every pixel of the shell on the screen. As Hirche et al. [HEGD04] already pointed out, simply using the base or the offset mesh is not sufficient to capture object silhouettes. Instead, the entire volume between the base mesh and the offset mesh must be rasterized by rendering every individual prism defined in Section 3. Every prism should be enclosed as tightly as

© The Eurographics Association 2007.

possible to avoid unnecessary fragment computations. Consequently, we triangulate the prisms directly.

The prism sides need special attention: the triangulation has to form the *convex hull* of the prism. This can be obtained by forming a "hypothetical" triangle that connects three arbitrary vertices of one side. If the fourth vertex lies behind the corresponding plane, the triangulation is convex. Otherwise, the other vertex combination has to be used for this side. This is repeated for all three prism sides. Please note that this triangulation does not introduce any error in the actual rendering result as will become clear in Section 4.3.

4.2. Ray Segment Calculation

For each rasterized fragment, we need to cast a ray through the prism, transform it to texture space and evaluate the texture function. For this, we have to compute the entry and exit point in world space and afterwards transform them to texture space. Note that for our sampling algorithm (see Section 4.3) a point slightly *outside* the prism is sufficient instead of the *exact* intersection. Consequently, the current world space position generated by the rasterizer can already be taken as the entry point.

The exit point is calculated by intersecting the ray with all prism triangles. From the resulting points the one with the largest distance to the viewer is identified. Further, the barycentric coordinates of the resulting points with respect to the triangles make it easy to determine whether the intersection points lie inside the triangles (considered a "valid" intersection) or not (discarded from further consideration). Note that while there exist correct ray vs. bilinear patch intersections for the prism sides (for instance by Ramsey et al. [RPH04]), the described approximate solution is faster to compute and sufficiently correct.

Now the obtained world space entry and exit points are mapped to texture space. As discussed in Section 3.1, Φ can be inverted for any given value w. We therefore need to find the value w for the entry and exit points. This is done using a bisectional search: Starting from $w = 0$ for a point $\mathbf{p}$, a triangle $T(w) = \{\mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3}\}$ is constructed in world space at the current height and its normal vector $\mathbf{n}$ is computed. In each iteration, the sign of the dot product $dot(\mathbf{n}, \mathbf{p} - \mathbf{p_1})$ determines the search direction. In practice, we have found 12 iterations to provide a sufficiently correct solution, also for shells with very large thickness. The (u, v) coordinates can then directly be calculated with the obtained w components by using the barycentric correspondence. Finally, the obtained (u, v, w) coordinates of the entry and exit point are passed to the actual ray sampling step. Again, note that the ray segment starting and ending slightly outside the actual prism does not introduce any errors to the rendering result as will become clear in the following section.

4.3. Corrected Ray Path Sampling

With the approximate ray vs. prism entry and exit points in texture space, we sample along this ray segment in order to compute possible surface intersections. Note that a ray in world space forms a *curve* in texture space, as can be observed in Figure 7. In this figure, the ray even leaves and

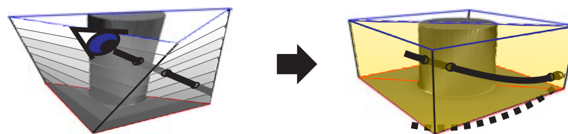

Figure 7: *Transforming a viewing ray from world space (left) to texture space (right) typically forms a curve. The footprint of the curve emphasises the curve progression.*

re-enters the prism. Figure 8 shows what happens if we just linearly sample in texture space: the texture in world space is bent in the opposite direction to the way the ray is curved in texture space.

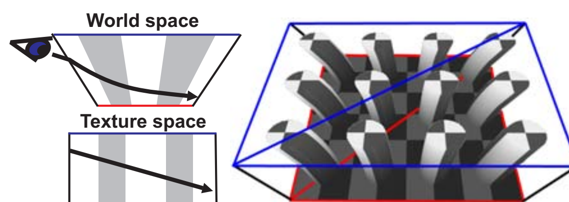

Figure 8: *Linearly approximating the ray in texture space results in a bent output. The bending is view dependent.*

The idea of our approach is to linearly approximate the curved ray in texture space, but *adjust* towards the correct world-space sampling position if the introduced error exceeds some image-space error. For now we assume a fixed distance between two subsequent samples. Section 4.4 will lift this restriction.

We start at the (approximate) ray entry point and sample linearly towards the exit point (see Figure 9, top left). In order to guarantee a decent image quality, the next sample position is transformed back to world space by evaluating Φ for the current position, see Figure 9, top right. The visual error is calculated as the angle between the view vector and a vector from the eye to the sample point. If this error exceeds a user-defined limit (for instance, one pixel angle) the sampling position must be recalculated. In this case, a triangle is constructed in world space at the current w position and the viewing ray vs. triangle intersection (the yellow point in Figure 9) defines the new sampling direction after being transformed back into texture space. This procedure is repeated with a sampling point in the new direction until the error is low enough (Figure 9, bottom). This approximation converges quickly, and in practice 4 iterations are usually

© The Eurographics Association 2007.

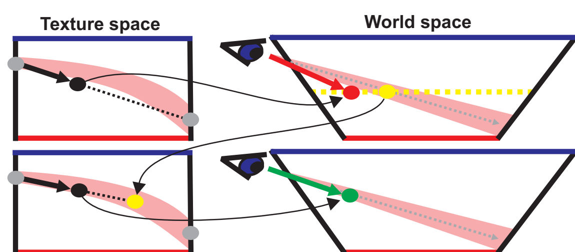

Figure 9: *Sampling correction: the sample point is projected into world space and checked for validity (top row). If invalid, we redirect to a closer valid point (bottom row).*

sufficient. Afterwards, we test whether the new sampling position is inside the prism using barycentric coordinates. This is necessary because we might have started outside the actual prism (see Section 4.2) or the ray has already left the prism but might re-enter (see Figure 7). Finally, sampling is continued towards the exit point and the whole process is repeated until an intersection occurs or the exit point is passed.

Figure 10 shows a possible sampling path through the texture space (left) and the obtained result (right). A comparison to Figure 3 and Figure 8 shows the improved image quality compared to coarse linear approximations.

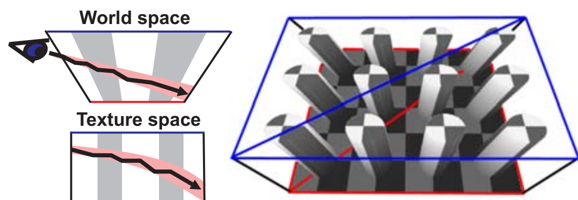

Figure 10: *Corrected sampling. Because the ray always stays within one pixel difference to the correct ray (pink area), no visible artifacts appear.*

It is important to note that for viewing rays almost aligned with a triangle, the ray intersection-based correction cannot work. However, it turns out that in this case a correction is not needed at all because linear sampling is already correct for such rays. This important property ensures that the algorithm works for all possible configurations and does not require special-case treatment. Another interesting property is that due to the image-space error metric, the number of corrections required decreases with increasing distance to the viewer, thus naturally reducing the computational time for distant scene parts.

The actual sample evaluation depends on the actual texture function. For height fields, the current w component of the ray is tested against the lookup value. If it is lower, an intersection is found. For 3D textures we simply have to test if the texture at the current position is non-empty.

Note that in case of an intersection, an exact z-buffer value must be stored in order to provide correct visibility. The fragment depth value cannot be used here because neighboring prisms might alternately intersect the viewing ray if it enters, leaves and re-enters, as shown in Figure 7. The correct z-value is computed by transforming the intersection point to world space, computing the distance to the eye and multiplying that value with the camera projection matrix.

4.4. Efficient Sampling and Filtering

So far we have assumed a constant sampling distance. However, the sampling should be adjusted in order not to miss any surface detail while at the same time providing fast sampling with only few steps. In order to reconcile these two demands, we chose *distance maps* [Don, KJZV92], because they can be applied to height fields as well as to 3D textures (another possible techniques include [KRS05,RSP05]). They require preprocessing for the texture function, but this does not impact the possibility to interactively modify the base and offset meshes.

In a preprocess, the texture space is partitioned using a regular 3D grid of cells called the *distance map*. For each such cell, the distance to the closest surface point is calculated and stored. Conceptually this defines the radius of an empty sphere around each cell (see Figure 11, left). Cells below the surface have a radius of 0. At runtime the radii are used as distance to the next sampling position as described in Section 4.3, because no surface can be missed in between. This process is repeated until a radius of 0 is reached (indicating an intersection), or the exit point of the ray has been passed (discarding the pixel). Figure 11 gives an example for this process.

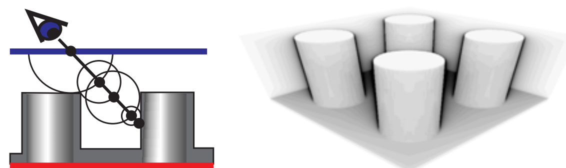

Figure 11: *Left: sphere tracing algorithm. Right: example with the number of steps visualized: from white=0 steps to black=40 steps.*

An important issue is a good reconstruction filter of the height field in order to avoid artifacts. We adopt a commonly used strategy: after an intersection point has been found, a bisectional search with the linearly filtered *original* texture provides much better quality with only little additional effort. Four search steps have been found to be sufficient.

4.5. Curved Shell Rendering

The display algorithm for curved shells is similar to the one for smooth shells, i.e., the geometric representation (Sec-

© The Eurographics Association 2007.

tion 4.1), the ray segment computation (Section 4.2) and the corrected ray sampling (Section 4.3) are identical. The main difference is the transformation of the sampling position before a sample value is looked up by computing the inverse mapping (2) in Section 3.2.

In order to still accelerate rendering with distance mapping as described in Section 4.4, we adapted the method to also work for curved shells. Because the empty spheres are compressed and curved by the space defined by the Coons patches (Figure 12 (left)), we scale each sphere by the thickness of the curved surface with respect to the thickness of the curved shell, resulting in the dotted sphere in Figure 12 (bottom, left). Although the effectiveness of the distance map acceleration is reduced, it is still more efficient than using a constant step size. Also note that while this heuristic is a non-conservative approximation and can theoretically lead to artifacts in regions where the curvature of the shell is higher than the curvature of the downscaled sphere, in pratice we never experienced such artifacts.

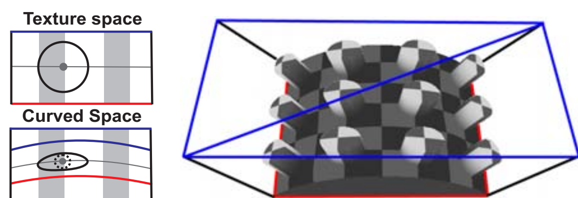

Figure 12: *Curved shell mapping (left) with an example (right).*

5. Implementation and Results

Given a base mesh with corresponding normals and texture coordinates, the application performs the extrusion and prism triangulation in a preprocessing step. At runtime the whole prism geometry (i.e. base mesh vertices and normals) and texture coordinates are passed through the vertex shader to the pixel shader. The latter performs the ray segment calculation (see Section 4.2) and the sampling step (Sections 4.3 to 4.5) in a single rendering pass. We implemented the described algorithm as a Shader Model 3.0 HLSL shader under DirectX 9. The machine was an Intel Pentium4 running at 3.2GHz with two NVIDIA Geforce 8800 GTX graphics boards running in SLI mode.

Concerning image quality, Figures 1 and 13 show the difference between tetrahedron-based, smooth, and curved shell mapping. Note the correct mapping along the surface offset for smooth shell mapping, and the absence of image artifacts. The curved shell mapping variant keeps this property and additionally smoothes the surface with C^1 continuity (Figures 1 and 13, right).

The additional images that come with this paper (additional electronic material) show numerous examples for mapping sampled data (although shells are not limited to this) onto different objects. For scene illumination we used Normal Mapping and Phong shading, or environment reflection mapping (Figure 14). The technique works equally well for very close and distant view points, and the screen-space error metric ensures that even very large offsets do not lead to any artifacts. Note that the filtering described in Section 4.4 lets the surfaces and silhouettes appear smooth, although all textures have a resolution of only 128 by 128 (by 128 for 3D textures) texels. Because the correct depth buffer value is stored, shadow mapping is also naturally applicable for smooth and curved shells.

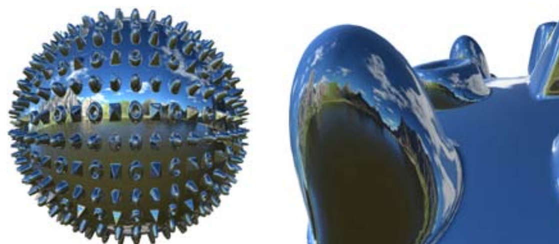

Figure 14: *Smooth shell map with environment mapping.*

Concerning rendering speed, we deal with fairly simple geometry and the vertex shader only passes data, so we are clearly pixel shader limited. Table 1 shows the achieved frame rates for the well known tiger model (see the additional images) mapped with a height field. We include frame rates for just the ray segment generation (see Section 4.2), and the smooth, the curved, and the tetrahedron-based shell mapping. In order to give an impression how the methods perform, we modulated some of the most performance-dominating parameters.

Technique / Test	Ray segment	Smooth shells	Curved shells	Tetra shells
Reference	115	15	5	43
Screen 640x480	135	18	5.9	51
Screen 1280x1024	55	7.6	3.4	33
Half thickness	139	19	7.5	53
Double thickness	83	10	3.1	32
8 times tiling	115	8.5	2.8	20
$\frac{1}{8}$th tiling	115	21	5.5	79

Table 1: *Frame rates for the smooth, curved and tetrahedron-based shell maps for the tiger model.*

The test in the row marked "Reference" was performed at 800x600 pixels screen resolution. As can be expected, all methods are sensitive to the number of actually rasterized fragments. Changing the shell thickness has a similarly high

© The Eurographics Association 2007.

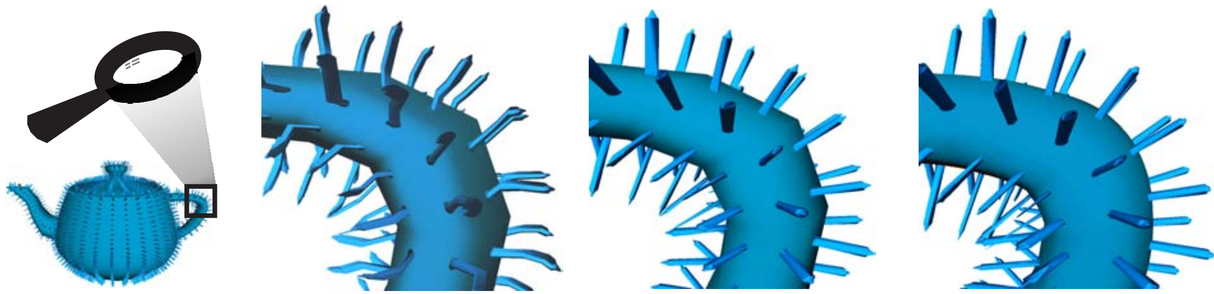

Figure 13: *Tetrahedra-based (left), smooth (middle), and curved (right) shell mapping. The mesh is the same in all three images.*

influence on the performance as changing the screen resolution. In contrast, the texture tiling influences the ray casting performance less than linearly. Smooth shell mapping is between 2.3 and 4.3 times slower than previous tetrahedron-based methods, which is the price for the high quality of the output images. The curved version is another 2.2 to 3.8 times slower than the smooth shell mapping because of the Coons patch evaluation and the increased sampling rate. However, note that the representation quality is also significantly higher.

It is interesting to note that the performance influence of the chosen error angle is negligible: relaxing it from one pixel hardly influences the frame rate, but lowers the image quality by introducing distortions. Consequently, it is not useful for trading rendering speed against image quality—the shell thickness has much more influence.

6. Discussion

The two presented shell-mapping techniques ensure high image quality for a given base mesh with normals. One of the big advantages of the new technique is its flexibility: because the mapping is local (i.e. only the vertex coordinates, normals and texture coordinates are required), any existing mesh can instantly be extended to a shell without the need for any preprocessing. We also used the geometry shader available in DirectX 10 to generate the prisms, which makes the technique completely transparent to the CPU. Animations are also naturally possible, i.e., deforming the base and/or the offset mesh, changing the mesh parametrization, or animating the texture function itself, as can also be seen in the video provided with this paper. In addition, when rendering the *back facing* polygons of a prism, all required fragments are generated even if the viewer is placed *inside* a prism. Only the ray segment calculation (Section 4.2) must be modified accordingly.

These desirable properties come at the cost of slower rendering than previous approaches because the pixel shader is more complex. It depends on the application if the image quality justifies the reduced rendering speed. For the curved shells, we also tried to subdivide the base mesh in the DirectX 10 geometry shader using the Coons patch approach and then applied the (faster) smooth shell mapping in order to get approximated curved shells. Unfortunately, the geometry shader performance was too low to be efficient, thus postponing this option for future drivers and/or hardware. Note that the two techniques can also be applied in a selective way, for instance, in mesh regions with varying local curvature, or by switching between smooth, curved and tetrahedron-based shells at certain distances, thus providing different levels of detail.

Rendering semi-transparent data with smooth and curved shells is not straightforward. Primitive sorting as in Dufort et al. [DLP05] is not sufficient here, because a ray may alternately intersect two prisms (see Figure 7). Depth peeling [Eve99] would be possible but this is rather slow due to the significant overdraw.

7. Conclusions and Future Work

This paper has presented smooth and curved shell mapping, two techniques for mapping volumetric data to arbitrary meshes. The main contribution compared to existing approaches is a *correct* smooth and optionally curved mapping along the interpolated surface normal. Exactly the same results are rendered as would be obtained by inverting the mapping Φ within the given pixel tolerance. In particular, our mapping does not exhibit any of the artifacts shown by previous methods, nor does it introduce new ones. It can even handle meshes where tetrahedra-based approaches fail (see Section 2). This allows us to apply textures to geometry with high local curvature and very thick surfaces with high image quality. Furthermore, in contrast to a hypothetical solution based on inverting Φ, our algorithm allows using rendering acceleration techniques for interactive rendering. We believe that this correct mapping will facilitate using volumetrically mapped surfaces in a more general way, also because it can instantly replace texture-mapped triangles without the need for any preprocessing.

An interesting question for future work is how to effectively define the prism height for curved shells (see Sec-

© The Eurographics Association 2007.

tion 3.2) so that it is tightly enclosed by the base and offset mesh. This will reduce the amount of fragments that do actually not contribute to the output image. Tesselating each base and offset triangle with respect to the curved surface in the geometry shader is too slow at the moment, but might become efficient in the future.

Another problem to be solved in the future are texture distortions due to the piecewise linear mapping along the surface. Because such distortions are more distracting with shell mapping than with simple texture mapping, one of the numerous surface parameterization techniques for reducing them should be applied. It might also be possible to non-linearly distort the textures at runtime, just as we did with the height in curved shell maps.

8. Acknowledgements

This work was supported by the Austrian Science Fund (FWF) contract nos. P17260 and P17261-N04.

References

[BD06] BABOUD L., DÉCORET X.: Rendering geometry with relief textures. In *Graphics Interface'06 Conference Proceedings* (2006), pp. 195–201.

[Bli78] BLINN J. F.: Simulation of wrinkled surfaces. In *SIGGRAPH '78 Conference Proceedings* (1978), pp. 286–292.

[BN76] BLINN J. F., NEWELL M. E.: Texture and reflection in computer generated images. *Communications of ACM 19*, 10 (1976), 542–547.

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The reyes image rendering architecture. In *SIGGRAPH '87 Conference Proceedings* (1987), pp. 95–102.

[DLP05] DUFORT J.-F., LEBLANC L., POULIN P.: Interactive rendering of meso-structure surface details using semi-transparent 3d textures. In *Proceedings of Vision, Modeling, and Visualization 2005* (Nov. 2005), pp. 399–406.

[Don] DONNELLY W.: *GPU Gems 2*. ch. Per-Pixel Displacement Mapping With Distance Functions, pp. 123–136.

[Eve99] EVERITT C.: Interactive order-independent transparency, NVIDIA Corp., White Paper, 1999.

[HEGD04] HIRCHE J., EHLERT A., GUTHE S., DOGGETT M.: Hardware accelerated per-pixel displacement mapping. In *Proceedings of Graphics Interface* (2004), pp. 153–158.

[HS98] HEIDRICH W., SEIDEL H.-P.: Ray-tracing procedural displacement shaders. In *Proceedings of Graphics Interface* (1998), pp. 8–16.

[KJZV92] KAREL J. ZUIDERVELD A. H. J. K., VIERGEVER M. A.: Acceleration of ray-casting using 3d distance transforms. In *Visualization in Biomedical Computing II, Proc. SPIE 1808* (1992), pp. 324–335.

[KK89] KAJIYA J. T., KAY T. L.: Rendering fur with three dimensional textures. In *SIGGRAPH '89 Conference Proceedings* (1989), pp. 271–280.

[KRS05] KOLB A., REZK-SALAMA: Efficient empty space skipping for per-pixel displacement maps. In *Proceedings of Vision, Modeling and Visualization* (2005).

[Ney98] NEYRET F.: Modeling, animating, and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics 4*, 1 (1998), 55–70.

[OP05] OLIVEIRA M. M., POLICARPO F.: *An efficient representation for surface details*. Tech. Rep. RP-351, Federal University of Rio Grande do Sul - UFRGS, 2005.

[PBFJ05] PORUMBESCU S. D., BUDGE B., FENG L., JOY K. I.: Shell maps. *ACM Transactions on Graphics 24*, 3 (2005), 626–633.

[PH89] PERLIN K., HOFFERT E. M.: Hypertexture. In *SIGGRAPH '89 Conference Proceedings* (1989), pp. 253–262.

[PH96] PHARR M., HANRAHAN P.: Geometry caching for ray-tracing displacement maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques* (1996), pp. 31–40.

[PHL91] PATTERSON J., HOGGAR S., LOGIE J.: Inverse displacement mapping. *Computer Graphics Forum 10*, 2 (1991), 129–139.

[PO06] POLICARPO F., OLIVEIRA M. M.: Relief mapping of non-height-field surface details. In *Proceedings of I3D* (2006), pp. 55–62.

[POC05] POLICARPO F., OLIVEIRA M. M., COMBA J. L. D.: Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of I3D* (2005), pp. 155–162.

[Por06] PORUMBESCU S. D.: Personal communication, 2006.

[RPH04] RAMSEY S. D., POTTER K., HANSEN C.: Ray bilinear patch intersections. *Journal of Graphics Tools 9*, 3 (2004), 41–47.

[RSP05] RISSER E. A., SHAH M. A., PATTANAIK S.: *Interval Mapping*. Tech. rep., School of Engineering and Computer Science, University of Central Florida, 2005.

[SSM00] SMITS B., SHIRLEY P., M.STARK: *Direct ray tracing of smoothed and displacement mapped triangles*. Tech. Rep. UUCS-00-008, Computer Science Department, University of Utah, 2000.

[SSS00] SMITS B. E., SHIRLEY P., STARK M. M.: Direct ray tracing of displacement mapped triangles. In *Proceedings of the Eurographics Workshop on Rendering Techniques* (2000), pp. 307–318.

[WTL*04] WANG X., TONG X., LIN S., HU S., GUO B., SHUM H.-Y.: Generalized displacement maps. In *Proceedings of the Eurographics Symposium on Rendering* (2004), pp. 227–233.

[WWT*03] WANG L., WANG X., TONG X., LIN S., HU S., GUO B., SHUM H.-Y.: View-dependent displacement mapping. *ACM Transactions on Graphics 22*, 3 (2003), 334–339.

[ZHW*06] ZHOU K., HUANG X., WANG X., TONG Y., DESBRUN M., GUO B., SHUM H.-Y.: Mesh quilting for geometric texture synthesis. *ACM Transactions on Graphics 25*, 3 (2006), 690–697.

© The Eurographics Association 2007.