Real-Time Shell Space Rendering of Volumetric Geometry

Nico Ritsche[*]

Abstract

This work introduces a new technique for real-time rendering of arbitrary volumetric geometry into a polygonal mesh's shell space. The shell space is a layer of variable thickness on top or below the polygonal mesh. The technique computes view ray shell geometry intersections in a pixel shader. In addition to arbitrary volumetric shell geometry, represented as volume textures, it can handle also the less general case of height-field shell geometry. To minimize the number of ray tracing steps, a 3D distance map is used for skipping empty space in the shell volume. The shell volume is defined by a pre-computed tetrahedra mesh. The tetrahedra subdivide prisms extruded from the polygonal base mesh. A vertex shader computes tetrahedron face plane distances for generating the per-pixel tetrahedron thickness using non-linear interpolation. The technique includes local lighting and overcomes shortcomings of previous shell space rendering approaches such as high storage requirements and involved per-vertex computations [Wang et al. 2004] or low shell geometry depth complexity and mapping distortions [Policarpo and Oliveira 2006]. Additionally, rendering artifacts for shallow view angles common to many related techniques are reduced. Furthermore, methods for generating a geometry volume texture and the corresponding distance map from a polygonal mesh are presented.

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

Keywords: real-time rendering, shell space, surface geometry, volume texture, 3D distance map, graphics hardware, displacement mapping, shading and lighting

1 Introduction

Rendering detailed surfaces in real-time has been, and still is, one of the major goals in computer graphics. So far, highly detailed surfaces are predominantly rendered using triangle based polygonal meshes. However, the limits of triangle based rendering become apparent when it comes to rendering highly detailed scenes, containing millions of triangles. Even the latest graphics cards fail in rendering such scenes with decent frame rates - the bottleneck is the huge triangle processing and transfer cost.

To reduce the amount of triangles of a detailed scene, Culling and Level of Detail (LOD) techniques have been introduced (see for instance Akenine-Möller and Haines [2002]). While these can be very effective, many of them consume significant resources themselves.

An alternative to the purely triangle based real-time rendering can be found when looking at the current trend in graphics hardware towards highly programmable and high-performance vertex and pixel shading units. Recent commodity graphics hardware is capable of rendering detailed surfaces in real-time using only coarse triangle meshes combined with finer scaled geometry stored either as a height-field, in a volume texture or in some form of layered data structure. Using a simplified form of ray tracing, this finer scaled geometry can then be rendered into the space either directly above or below the coarse triangle mesh. In this work, this layer is referred to as 'shell space' according to Porumbescu et al. [2005]. The geometry mapped into the shell space is called 'shell geometry'. The ray tracing is either executed in a pixel shader in real-time or as a pre-process, where the result is stored as a high dimensional function and then queried at render time in a pixel shader. Techniques combining traditional triangle based rendering with shell space ray tracing in this way, balance the rendering pipeline by lessening the geometry transfer at the expense of consuming more fill rate.

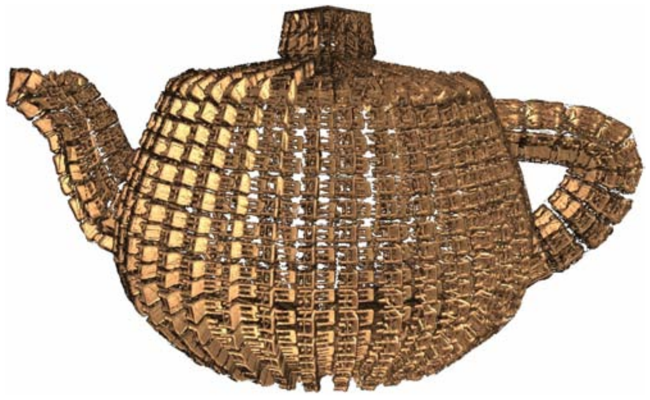


Figure 1: Chair shell geometry applied to a teapot base mesh under local illumination, rendered with the technique described in this work.

However, currently most shell space ray tracing based real-time rendering techniques are limited to height-field shell geometry. Only three techniques so far are capable of handling arbitrary shell geometry [Wang et al. 2004; Dufort et al. 2005; Policarpo and Oliveira 2006]. Wang et al.'s technique uses a pre-computed *generalized displacement map* to query shell geometry intersections at render time, whereas Dufort et al. and Policarpo and Oliveira compute view-ray shell geometry intersections in real-time in a pixel shader. While these techniques produce impressive results in many cases, they have some significant limitations. *Generalized Displacement Mapping (GDM)* has high shell geometry storage requirements because it does the ray tracing in a pre-process and stores the result as a five dimensional function. Moreover, *GDM* requires complex geometric computations per vertex. Dufort et al.'s technique achieves only highly interactive performance in most cases, due to its focus on semi-transparent shell geometry, necessitating a costly per frame tetrahedra sorting. Finally, Policarpo and Oliveira's approach is only practical for shell geometry with low depth complexity, suffers from texture space distortion for highly curved meshes and is prone to rendering artifacts for shallow view angles.

In order to overcome these weaknesses, this work introduces a new technique for rendering arbitrary volumetric as well as height-field geometry into the shell space of a polygonal mesh. The technique

[*]e-mail: nico.ritsche@3dgraphicsresearch.com

GRAPHITE 2006, Kuala Lumpur, Malaysia, November 29 – December 02, 2006.
© 2006 ACM 1-59593-564-9/06/0011 $5.00

shares some similarities with Dufort et al.'s approach but focuses on rendering opaque volume texture shell geometry with real-time instead of only interactive frame rates. To avoid mapping distortions introduced by most previous approaches, the polygonal base mesh's shell space is sub-divided into tetrahedra. The tetrahedra front faces are rendered with z-buffering enabled. Using a tetrahedra mesh, computing complex per vertex ray-prism intersections as in *Generalized Displacement Mapping* can be avoided. Instead, only less complicated view ray tetrahedron face plane intersections are computed in a vertex shader. Using these intersections, the per pixel tetrahedron thickness is computed by non-linear interpolation. A pixel shader computes view ray shell geometry intersections inside the tetrahedron, using the tetrahdron thickness, utilizing a 3D distance map [Donnelly 2005] to efficiently skip empty space. In contrast to Dufort et al.'s approach, the traffic between vertex and pixel shader (and thus the interpolation work) is minimized by only sending the texture space intersection distances and view direction, instead of sending both the intersection positions and the tetrahedron ray entry positions in object as well as in texture space. Moreover, no per vertex object to texture space transformation matrix is required. In comparison to linear sampling approaches, the new technique generates smoother renderings and requires less ray sampling steps. Furthermore, it reduces the typical under-sampling artifacts for shallow view angles common to the majority of existing real-time ray tracing based approaches. See figures 1, 10 and 11 for images rendered with the new technique.

In summary, the contributions of this work are as follows:

- A new real-time technique based on pixel shader ray tracing for rendering arbitrary volumetric geometry into a polygonal mesh's shell space, utilizing a 3D distance map to skip empty space.
- A method for generating a geometry volume texture from a polygonal mesh.
- A method for creating a 3D distance map for a geometry volume texture.

2 Related Work

Techniques closely related to this work can be found in the emerging field of shell space rendering based on real-time pixel shader ray tracing. In *Relief Mapping*, shell geometry is represented using a height-field together with a normal map for lighting [Policarpo et al. 2005]. In the rendering process, for each pixel on a relief mapped polygon, a view ray is traced - assuming the surface is locally planar - starting from the position where it hits the polygonal surface, until the intersection with the relief is found. Firstly, the ray is sampled linearly, then, when the intersection with the height-field was found, a binary search follows to prevent missing fine surface features. Finally, the pixel is shaded using the color and normal at the intersection's texture space xy-position. Additionally, self-shadowing of height-map geometry is computed by tracing a ray from the intersection to the light source(s) using a similar search strategy. An extended version of the technique supports height-field silhouettes for closed relief mapped polygonal meshes by locally fitting the height-field to quadrics, approximating the curvature of the underlying object [Oliveira and Policarpo 2005]. The most recent incarnation of relief mapping is capable of rendering non-height-field shell geometry [Policarpo and Oliveira 2006]. This is achieved by using several layers of relief textures, storing the depth values of the shell geometry's boundaries in the direction of the base mesh's normal and the according shell geometry normals. In addition to rendering the shell geometry, the technique can at the same time apply a texture to the underlying polygonal surface. An advantage of the technique are its low shell geometry storage requirements, although it is only practical for surface geometry with low depth complexity.

An extended version of *Parallax Occlusion Mapping (POM)* [Brawley and Tatarchuk 2005] also applies height-field shell geometry to a polygonal mesh [Tatarchuk 2005]. Following a linear ray sampling, the extended *POM* computes the exact intersection between the ray segment defined by the hit position from the linear search and the last ray sampling position outside the height-field geometry and the linear segment which locally approximates the height-field between those ray positions. As a measure to improve the accuracy for shallow view angles, the sampling rate is made a linear function of the angle between the view ray and the polygonal surface normal. The greater this angle, the more densely the view ray is sampled. The technique incorporates smooth shell geometry self-shadowing using a visibility sample filtering technique and a LOD scheme blending between *POM* (close distances) and Bump Mapping (distances further away).

Cone Step Mapping (CSM) [Dummer 2006] performs significantly better than *Relief Mapping* and *POM* while producing similar quality results. It uses top-opened cones as empty space bounding volumes, placed on top of each height-field surface position, stored in an extra 2D texture called cone step map. For each cone, the cone radius is maximized, ensuring no geometry is inside the cone. During ray tracing, the step size is computed as the distance from the current view direction to the boundary of the cone queried from the cone step map. This way, the number of ray tracing steps is minimized. *CSM* incorporates height-field self-shadows.

Donnelly [2005] uses spheres instead of cones as empty space bounding volumes to minimize the number of ray tracing steps. This makes his technique applicable for ray tracing arbitrary shell geometry. However, a downside of this generality is that spheres do not include directional information like cones do.

Hirche et al. [2004] render height-field geometry into a mesh's shell space which is defined by a tetrahedra mesh, generated from extruded prisms. The tetrahedra are rendered using a modified Projected Tetrahedra (PT) algorithm [Shirley and Tuchman 1990]. To render a tetrahedron, it is decomposed into triangles on the CPU, depending on the viewer position. The resulting triangles are then rendered, attributed with their secondary (or back side) vertex generated in the decomposition step. A pixel shader performs a linear sampling of the viewing ray between the interpolated texture coordinates of the primary (front) vertex and the interpolated secondary vertex, using four fixed steps. The ray height-field intersection is computed by intersecting the line segment between the first sampling position where the ray is inside the height-field and the previous sampling position with the linearly approximated height-field surface between these two positions.

Dufort et al. [2005] render shell geometry represented as volume textures with interactive frame rates. It supports semi-transparency and is thus capable of rendering a wide range of volumetric effects like glow, fog, or volumetric shadows. A mip-map based shell geometry filtering scheme and shadow mapping for inter-object and shell geometry shadowing is included. Just like in Hirche et al.'s approach the base mesh's shell space is defined by a tetrahedra mesh. The rendering steps are as follows: Firstly, the tetrahedra are sorted in the CPU according to their depth to ensure they are rendered correctly using alpha blending. Next, for each tetrahedron vertex, intersections between the view ray and the tetrahedron's faces are computed. These intersections are then interpolated to obtain the tetrahedron exit position per fragment. Finally, a pixel shader traces the view ray with a fixed step size between the tetrahedron entry and exit position, accumulating the volume texture's voxel's color contributions. A bottleneck of the technique is the expensive per-frame depth sorting of tetrahedra.

A challenge mainly for pixel shader ray tracing based techniques which do not partition the shell space using extra polygonal elements (like e.g. prisms or tetrahedra) poses the fact that for rays nearly tangential to the polygonal surface, the depth of the shell space to be ray traced dramatically increases. To avoid rendering artifacts in form of 'jaggies' due to missed shell geometry features (linear sampling) or chewing gum like distortions due to early ray tracing stops (distance map sampling), some techniques increase the number of ray sampling steps for shallow view angles. However, a high number of sampling steps mostly means a big performance hit, so a compromise between rendering quality and performance has to be found. Furthermore, a problem of the techniques which do not add extra polygonal elements to the base mesh, are mapping distortions for meshes with high curvature variation, due to the assumption of a locally planar base mesh. Most of these techniques also fail in rendering correct shell geometry silhouettes and cannot handle open surfaces (basic *Relief Mapping*, *POM*, *CSM*, *Displacement Mapping with Distance Functions*).

View-dependent Displacement Mapping (VDM) [Wang et al. 2003] and *Generalized Displacement Mapping (GDM)* [Wang et al. 2004] do the shell space ray tracing not at render time in a pixel shader but in a pre-process and store the result as a high dimensional function queried at render time. This way, the ray tracing can be much more accurate than (online) pixel shader ray tracing because there is no limited time frame. Thus sampling artifacts can be completely avoided in case the offline ray tracing is accurate enough. However, a disadvantage is the immense storage space required for the resulting high dimensional function. In *VDM*, this pre-computed function is called *view-dependent displacement map (vdm)*. It is a 5D function storing the distance between a position on the base surface and the intersection with the shell geometry stored as a height-field, for each possible viewing direction. This distance is also dependent on the curvature of the underlying surface. If no height-field intersection occurs, the *vdm* is set to -1 to represent a silhouette ray. During rendering the *vdm* is queried to determine the intersection with the height-field. *View-dependent Displacement Mapping* cannot handle open surfaces and produces mapping distortions for highly curved base surfaces, due to the locally planar base mesh assumption. *Generalized Displacement Mapping* generalizes the *VDM* approach for arbitrary volumetric shell geometry. The *gdm* is defined for each position inside the shell geometry volume and contains the distance to the closest solid volume for each view direction. Just like the *vdm*, the *gdm* is a 5D function, however, in contrast to the *vdm* it is not only defined over the polygonal surface, but over the entire shell space, defined by prisms extruded from the base mesh. Outside facing prism faces are rendered and view ray prism intersections are computed in a vertex shader. The texture space view ray necessary for the *gdm* lookup is computed as the view ray segment inside the prism. Shell geometry self-shadowing is achieved by doing further *gdm* lookups in the direction of each light source. *Generalized Displacement Mapping* includes local illumination as well as global illumination based on bi-scale radiance transfer [Sloan et al. 2003]. In comparison to *VDM*, *GDM* has the advantage of handling arbitrary shell geometry. Also, *GDM* produces far less mapping distortions than *VDM* and open surfaces are rendered correctly, due to its prism approach. However, rendering a *gdm* is significantly slower than rendering a *vdm*, mainly due to the costly ray-prism intersection computations involved in *GDM*.

The remainder of this paper is organized as follows. Section 3 describes the new shell space rendering algorithm, consisting of tetrahedra mesh and distance map generation in pre-processes as well as per frame shell geometry intersection computations. Section 4 presents a method for generating a geometry volume texture from a polygonal mesh. In section 5 results of the new rendering technique are given which are discussed in section 6 and finally, section 7 summarizes the presented approach and points out directions for future work.

3 Shell Space Rendering of Volumetric Geometry

This section describes the new real-time algorithm for rendering volume texture as well as height-field shell geometry into the shell space of a polygonal mesh. The technique is similar to Dufort et al.'s approach [2005] but focuses on rendering opaque shell geometry with real time instead of semi-transparent shell geometry with only interactive frame rates.

3.1 Algorithm Overview

In a pre-process, a tetrahedra mesh covering the polygonal mesh's shell space is generated by sub-dividing prisms extruded from the base mesh. In another pre-process, a 3D distance map for the shell geometry is generated. At render time the shell geometry mapped into the polygonal mesh's shell space is intersected with a view ray for each fragment covered by an outward facing tetrahedron face in a pixel shader. If the ray intersects shell geometry, the fragment is shaded, otherwise it is discarded. Using z-buffering it is ensured that only the visible parts of the shell geometry are rendered, making a tetrahedra sorting unnecessary. Each view ray needs only to be traced between its corresponding tetrahedron entry and exit position. To find out the length of this ray segment, the fragment's tetrahedron thickness in view direction is computed. This is achieved by first intersecting the view ray at each tetrahedron vertex with the planes spanned by the tetrahedron faces in a vertex shader. Next, the distances from the vertex to the intersections are sent to a pixel shader. For each two vertices, these distances are linear functions between the view ray intersections with a specific plane and *not* between the two vertices. Thus to correctly interpolate these distances, non-linear interpolation has to be used (see [Kraus et al. 2004]). In the pixel shader the smallest interpolated intersection distance greater zero is chosen. This distance is the per fragment tetrahedron thickness in view direction.

3.2 Tetrahedra Mesh Generation

The shell space of the polygonal base mesh is defined by extruding its vertices along their normals. The amount of extrusion defines the shell volume's thickness. Depending on the mesh's geometry, the offset has to vary over the surface to avoid self-intersecting shell space regions (see figure 2, bottom left). Generating a non self-intersecting shell space for a complex polygonal mesh is beyond the scope of this work, but for many meshes visually pleasing results can be achieved using a uniform offset (see for instance [Peng 2004] for the generation of non self-intersecting shell spaces). The vertices of the base mesh triangles and the extruded vertices define prisms. Each of these prisms is subdivided into three tetrahedra. This way, a mesh of connected tetrahedra is defined over the complete shell space (see figure 2, top and bottom right). It has to be ensured that the tetrahedra are aligned face to face, i.e. no edges should cross, otherwise linear interpolation of shell space texture coordinates yields inconsistent results. For the generation of a consistent tetrahedra mesh see for instance [Porumbescu et al. 2005].

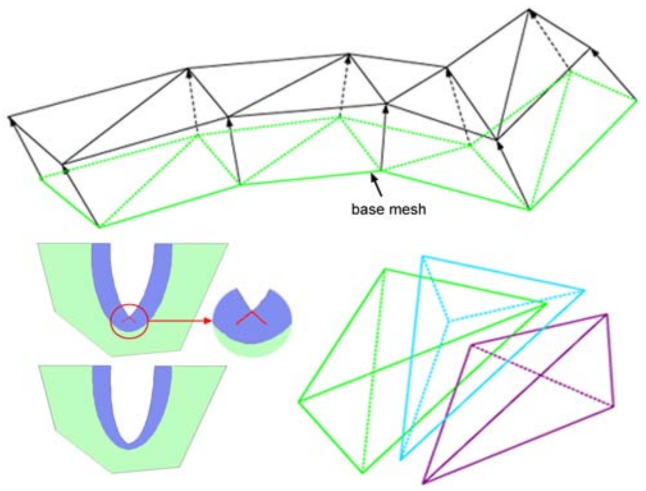


Figure 2: Top: The vertices of a base mesh are extruded in normal direction. The resulting prism mesh defines the base mesh's shell volume. Bottom right: Each extruded prism is sub-divided into three tetrahedra. Bottom left: Self-intersecting (top) and non self-intersecting (bottom) shell space (dark area).

3.3 Distance Map Generation

As another pre-process, a 3D distance texture containing the distance to the closest surface point for each position in texture space is created. This texture is used by the shell space rendering algorithm to skip empty space when ray tracing view rays (see section 3.5). For height-fields the technique described in [Donnelly 2005] can be used to compute a distance map. For volume texture shell geometry the distance map is computed from an initial polygonal representation of the geometry stored in the volume texture. This polygonal representation is also used to create the geometry volume texture itself, by discretising the mesh using a regular 3D raster (see section 4). The same raster is used to create the distance texture. For each voxel in the raster, the distance of the voxel centre to the closest surface point of the polygonal mesh is computed directly. This is achieved by first locating the triangle closest to the voxel (-center) and then locating the triangle's surface position closest to the voxel. Computing a distance map this way produces very accurate results (see figure 3). However, it is a very time consuming computation for large distance maps and complex shell geometry meshes. While the 64x64x32 distance maps shown in figure 3 took only about a minute to compute on a current PC (each), the computation of a 256x256x128 distance map for a highly tesselated shell geometry mesh takes several hours.

3.4 Tetrahedron Face Plane Intersection

At render time a vertex shader computes the distances from the vertex to the planes spanned by the tetrahedron faces. These distances are needed to generate the per fragment tetrahedron thickness. For the planes spanned by the tetrahedron faces where the vertex is part of the intersection is already known: the vertex position itself. Obviously, the distances from these planes to the vertex are all zero and do not need to be computed. Thus the only plane which remains to be intersected is the plane spanned by the tetrahedron face where the vertex is not part of. The vertices of this tetrahedron face are supplied as per vertex attributes both in texture and in object space. In addition, an index for identifying the plane to be intersected is

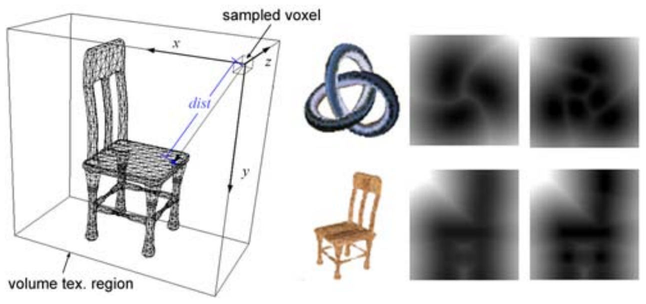


Figure 3: Left: Distance computation for an example voxel. *dist* is the distance from the depicted voxel to the closest position on the polygonal mesh surface. Right: A few z-slices of 64x64x32 distance maps generated for a torus knot (top) and a chair (bottom) geometry volume texture. The images at the center show renderings of the corresponding geometry volume textures, created with the technique described in this work.

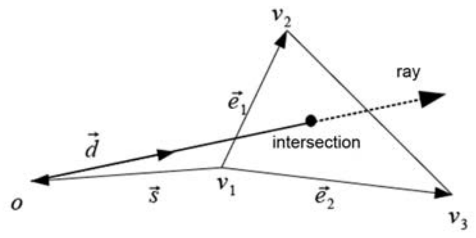
supplied. The intersection is computed using a common ray triangle intersection formula [Akenine-Möller and Haines 2002]:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{\vec{q}\cdot\vec{e}_1}\begin{pmatrix} \vec{q}\cdot\vec{e}_2 \\ \vec{p}\cdot\vec{s} \\ \vec{q}\cdot\vec{d} \end{pmatrix} \tag{1}$$

$$\begin{aligned} \vec{p} &= \vec{d}\times\vec{e}_2 \\ \vec{q} &= \vec{s}\times\vec{e}_1 \\ \vec{s} &= \vec{o}-\vec{v}_1 \end{aligned} \tag{2}$$

$$\text{ray: } R(t) = \vec{o}+t\vec{d} \tag{3}$$

$$\text{intersection: } \vec{o}+t\vec{d} = \vec{v}_1+u\vec{e}_1+v\vec{e}_2 \tag{4}$$



The intersection computation is done in object space. Additionally, the texture space intersection position is computed using the barycentric intersection coordinates u and v. This can be done because u and v specify positions relative to the tetrahedron face spanning the intersected plane. The texture space intersection position is used to compute the texture space view direction, either by subtracting the intersection position from the vertex's texture coordinate in case the intersection lies in negative view direction ($t < 0$), or by subtracting the texture coordinate from the intersection position in case the intersection lies in positive view direction ($t > 0$). The plane distances are sent to the pixel shader using non-linear interpolation, while standard linear interpolation is used for the view direction. The non-linear interpolation is achieved by separately passing the distances to the intersected plane divided by the intersection position's corresponding clip space position w and the non-linear interpolation term $1/w$ to the pixel shader. In the pixel shader the distance is then reconstructed by dividing it by the linearly interpolated non-linear interpolation term $1/w$ (see figure 4). Details about non-linear interpolation can be found in [Kraus et al. 2004].

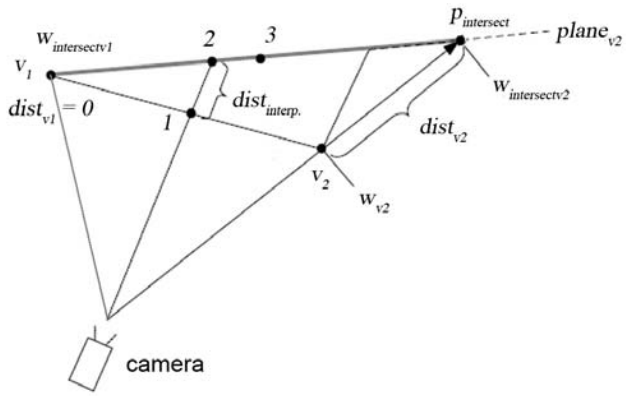


Figure 4: 2D view of a tetrahedron. Linear interpolation of the intersection distances to $plane_{v2}$ ($dist_{v1}$ and $dist_{v2}$) between the vertices v_1 and v_2 would produce an incorrect result. This is because the intersection distance is a linear function between v_1 and $p_{intersect}$ and *not* between v_1 and v_2. As an example, at position 1 (the centre position between v_1 and v_2) linear interpolation would yield the interpolated distance at position 3 and not the correct distance at position 2.

3.5 Shell Geometry Intersection

3.5.1 Overview

For each rendered fragment a view ray is intersected with the shell geometry inside the corresponding tetrahedron, and the fragment is shaded if an intersection was found (see figure 5, top). The following pseudo code roughly illustrates the process for volume texture shell geometry:

```
reconstruct tetrahedron face plane distances;
tetra_thickness = smallest face plane dist. > 0;
rayPosition = tc_in;
travelledDist = 0;

for i = 0 to maxSearchSteps
{
  dist = texLookUp(distance map, rayPosition);
  travelledDist = travelledDist + dist;
  if ( travelledDist > tetra_thickness )
    discard fragment;
  else
    step = dist * viewDir;

  rayPosition = rayPosition + step;
}

shade fragment;
```

3.5.2 Details

The distances to the planes spanned by the tetrahedron faces are reconstructed per fragment from the linearly interpolated intersection distances (which were divided by the corresponding clip space w coordinate in the vertex shader) and the linearly interpolated non-linear interpolation term $1/w$. The smallest positive intersection distance is the tetrahedron thickness at the fragment's position in view direction (see [Weiler et al. 2002], section 3.2). After the thickness was chosen, all necessary quantities to start the ray tracing are given. The 3D texture coordinate of the fragment, which was attributed per vertex and then conventionally linearly interpolated, is the texture space tetrahedron entry position (tc_{in}) of the view ray. This position plus the tetrahedron thickness times the normalized texture space view direction computed per vertex is the view ray's tetrahedron exit position (tc_{out}). The view ray segment between these two positions has to be searched for the first shell geometry intersection (see figure 5, top). Next, the view direction

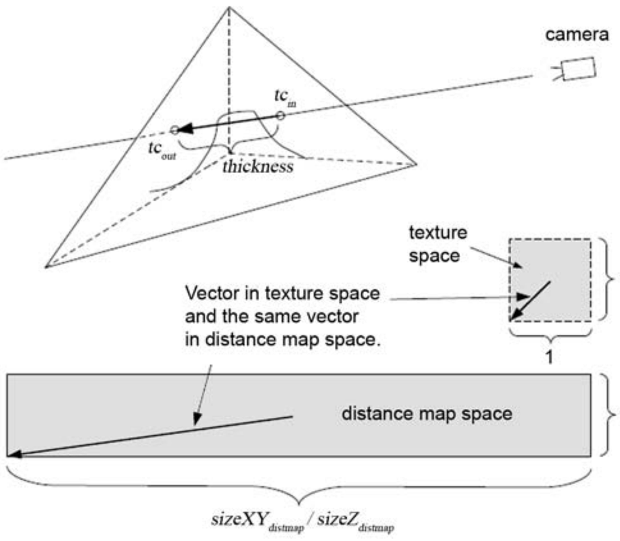

camera
tc_{in}
tc_{out}
thickness
texture space
Vector in texture space and the same vector in distance map space.
1
1
distance map space
1
$sizeXY_{distmap} / sizeZ_{distmap}$

Figure 5: Top: Shell volume Ray Tracing inside a tetrahedron. The view ray is traced between the entering position tc_{in} and the exit position tc_{out}. Bottom: 2D views of texture space and distance map space, which is the space where the distance values in the distance map are defined in.

in distance map space - which is the space where the distances in the distance map are measured in - is computed. For this, the xy-component of the texture space vector from the ray entry to the ray exit position is multiplied by the ratio of the distance map's xy-edge length to its edge length in the depth dimension. The z-component is left unaltered because both the distance map's z-depth and the shell space's texture space thickness is 1 (see figure 5, bottom). Before normalizing the distance map view direction, its length, which is the tetrahedron thickness in distance map space, is computed. Next, the intersection search loop is entered. Ideally, the number of loop-iterations should depend on the tetrahedron thickness which defines the length of the traced ray segment. Unfortunately, it turned out to be inefficient to use a dynamic loop count with current graphics hardware.[1] For volume texture shell geometry a fixed number of loop-iterations turned out to produce best results. However, for height-field shell geometry better results were achieved with a variable number of loop-iterations, dependent on the tetrahedron's orientation relative to the view direction. The more shallow the angle between the view ray and the tetrahedron face, the more loop-iterations are required to sample the shell geometry with

[1]This work was implemented using a NVIDIA graphics card. It is said that ATI graphics cards have better dynamic branching capabilities due to smaller pixel block sizes. Thus, there might be a chance that a dynamic loop count is more efficient on ATI cards.

enough accuracy. Thus, for height-field shell geometry the tetrahedra are grouped according to their orientation in a pre-process, and each tetrahedra group is rendered separately with an appropriate number of ray sampling iterations. In each iteration the distance map is queried at the current texture space view ray position, starting with the tetrahedron entry position (tc_{in}). The result of the query is the distance which can be traveled safely without intersecting any shell geometry (see figure 6, right). Next, the step vector is computed by multiplying the distance map distance with the normalized distance map view vector. The step vector is converted back to texture space and then added to the current ray position. If the total traveled distance map distance exceeds the distance map tetrahedron thickness, the pixel is discarded. If the intersection is found before the last iteration of the ray tracing loop, the current ray position remains unaltered because the distance map queries yield zero distances in the remaining iterations. Finally, to receive the shell geometry color at the intersection the texture space intersection position is used to either query the geometry volume texture or the 2D height-field texture representing the shell space geometry (in case of height-field shell geometry only the uv-components of the intersection position are used). To find out if the intersection position received from the ray tracing is inside or outside solid volume, the opacity value stored in the alpha channel of the geometry volume texture is used. Local lighting is achieved using the shell geometry normal queried from a 3D (geometry volume texture) or 2D (height-field) normal map. The normals are transformed to local tangent space using the base mesh's tangent frame. The tangent frame is either interpolated at the view ray's tetrahedron entry position, or - for more accurate lighting - fetched from a tangent frame texture generated in a pre-process. Using a tangent frame texture requires the base mesh's texture coordinates to define a bijective mapping.

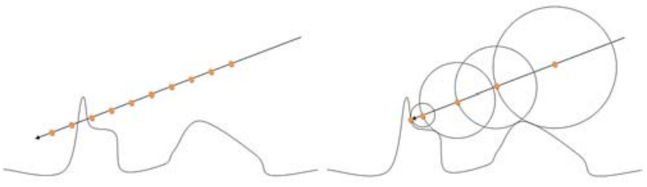


Figure 6: Left: Ray sampling approaches with equally sized steps can miss fine features. Right: Ray tracing using a 3D distance map. The spheres visualize the maximum distances found in the distance map at the sampling positions. These distances can safely be traveled without hitting any surface geometry. Using the distances from the distance map as ray tracing step sizes ensures that no surface geometry features are missed.

4 Geometry Volume Texture Generation

This section describes a method for creating a geometry volume texture from a polygonal mesh as a 'by-product' of the distance map generation. The method is practically for free in case a distance map is created as well. As already mentioned in section 3.3 about distance map generation, the geometry volume texture is created using a 3D raster. This raster is overlaid over the polygonal mesh. A voxel in the raster represents a voxel in the volume texture. During distance map creation the vector to the closest surface position is computed for each voxel (see section 3.3). To find out if a voxel is inside solid volume, this vector is normalized and the dot product with the normal at the closest surface position is computed. If the result is positive the voxel is inside solid volume (see figure 7). In this case it is set to the color at the closest surface position and its alpha value is set to one. (Ideally the dot product is either 1 or -1. However, in practice it deviates from that values, due to precision issues.)

Alternatively, a geometry volume texture can be created by sampling the polygonal mesh's triangles in the resolution of the 3D texture and setting the voxels of the sampling positions to the corresponding surface color. This method is more efficient then the volume sampling approach presented above, however it doesn't fill the interior of the object. This possibly causes the rendering algorithm to produce holes in the shell geometry. To alleviate this, the shell geometry boundaries can be thickened by slightly offsetting the mesh in negative normal direction and then repeating the sampling process. However, this in turn can lead to self-intersections which distort the shape of the geometry. See figure 8 for some sample layers of geometry volume textures.

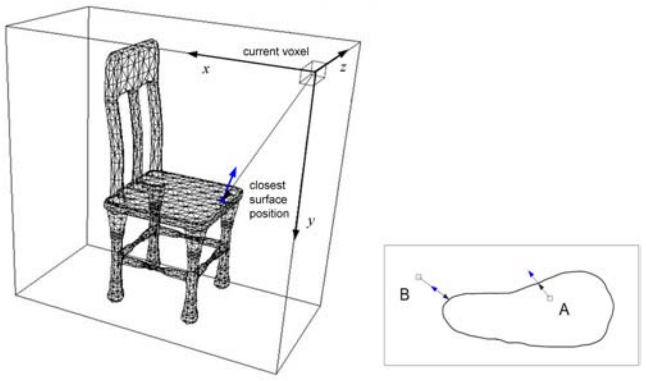


Figure 7: A 3D (left) and a 2D (right) example for the geometry volume texture generation. A voxel is inside solid volume in case the vector from the voxel to the closest mesh surface location and the surface normal at that location point in the same direction (A). Otherwise the voxel is outside solid volume (B).

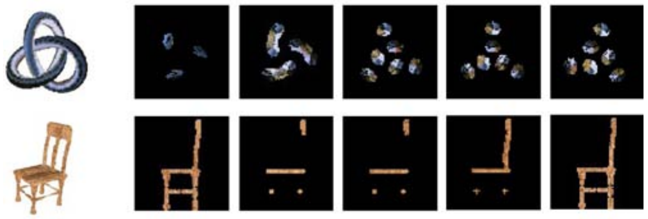


Figure 8: Top: A few layers of a torus knot (top) and a chair (bottom) geometry volume textures.

5 Results

The technique was implemented in C++ using DirectX9 with its high level shading language HLSL. Figure 1 as well as figures 10 and 11 at the end of this paper show results, rendered on an AMD Athlon64 with 2,4 GHz, 1 GB ram and a GeForce 7800 GTX graphics card. The screen/window resolution varied from 800x600 up to 1280x1024. The size of the geometry volume textures used in the volume texture shell geometry renderings is 64x64x32. For these renderings the view rays were sampled with a maximum of 4 to 10 steps per tetrahedron, regardless of the view ray's orientation relative to the base surface. The size of the height-fields used in the height-field shell geometry renderings is 256x256. For these renderings the maximum ray tracing search steps used for a tetrahedron vary from 4 for view directions perpendicular to the base surface, to 9 for view directions tangential to the surface. Back facing tetrahedra were culled (see table 1 for details).

Base Geometry (figure)	Shell Geometry	lit	#tetrahedra	#pixels covered	≈fps	#ray sampl. steps per tetra.
Cylinder (10 B)	Torus (64x64x32)	•	630	149338	83	10
Sphere (10 A)	Torus Knot (64x64x32)	-	1650	373689	35	4
Teapot (10 E)	Torus (64x64x32)	-	3072	465628	33	6
Plane (10 F)	Torus Knot (64x64x32)	-	6	200428	100	8
Teapot (11 A)	Stone Wall (256x256)	-	3072	356509	50	4/9
Cylinder (11 B)	Brick Wall (256x256)	•	630	102521	100	4/9
Plane (10 D)	Chair (64x64x32)	-	6	40988	190	10
Torus (11 C)	Relief (256x256)	•	1080	862200	40	4/9

Table 1: Comparison of selected renderings. Note that the number of pixels covered is not a measure of the total number of fragments processed. The number of processed fragments depends on the depth complexity of the object. The depth complexity in turn is determined by the average number of overlying tetrahedron front faces.

6 Discussion

The performance of the technique mainly depends on the number of fragments processed by the pixel shader. The number of processed fragments in turn is dependent on the number and size of front facing tetrahedra faces rendered. The processing time of a single fragment mainly depends on the number of maximum search steps per tetrahedron and to a minor degree on the kind of shell geometry used. The less voxels the shell geometry occupies, the more fragments are likely to be discarded early. Also, the less complex the shell geometry, the less view ray sampling steps are required to produce accurate results. These theoretical observations are supported by the rendering results data listed in table 1. However, for instance considering the sphere rendered in figure 10 A was rendered with ≈35 fps, while the teapot in figure 10 E was rendered with ≈33 fps might be striking at first glance. The performance of the two renderings is almost equal, even though in the teapot rendering ≈86% more tetrahedra are rendered, ≈25% more pixels are covered and 2 more ray sampling steps per tetrahedron are used. Considering these differences, the performance discrepancy should be greater than 2 fps. However, the reason for the teapot being rendered with almost the same performance as the sphere can possibly be found taking into account the shell geometry used. The torus knot shell geometry used in the sphere rendering is more complex than the torus used in the teapot rendering. Thus, while for many fragments a few 3D texture fetches at different view ray locations are necessary to reach the torus knot surface, the surface of the torus is reached after only one ray sampling iteration in most cases, meaning the complete empty space between the polygonal surface and the shell geometry surface being skipped using only one distance map texture fetch. This means, in the remaining 5 ray sampling iterations the texture is fetched at the same location again and again, yielding a zero distance. This in turn means, the final 4 texture fetches most likely benefit from the texture cache in current graphics hardware. To the author of this work, this seems the most probable explaination for the low performance difference between the renderings 10 A and 10 E. However, other factors, like details in the graphics hardware used, might play a role as well.

Comparing the technique with linear view ray sampling approaches, no additional texture fetches for the 3D distance map are necessary for volume texture shell geometry. This is because the geometry volume texture fetch needed for linear sampling is simply replaced by a distance map texture fetch.

A problem of the technique shared with most (if not all) tetrahedra mesh based shell space rendering approaches is the linear interpolation of the 3D texture coordinates over the tetrahedron triangles, leading to texture space distortion (see figure 12, bottom right). The finer the base mesh is tesselated, i.e. the smaller the tetrahedra, the less severe is the distortion. However, a finer tesselation also means more tetrahedra are generated. A solution to the problem is to interpolate the texture coordinates with non-linear interpolation. The idea was successfully implemented partly in the course of this work. The full implementation remains a task for future work. The effect of the texture space distortion is most prominent near the positions where the tetrahedra faces slice the shell geometry when rendering close-up views (see figure 12, top right).

When looked at an object from further away, thin lines appear for some views near the tetrahedra mesh edges, fine geometric structures in geometry volume textures are missed for some view angles, don't have precise shape or change when viewed from different view angles, and shell geometry seen from far away is distorted or vanishes (see figure 12, bottom left). However, it is not yet clear if these problems are solely caused by the texture space distortion. Inaccurate tetrahedron thickness computation might also be involved.

Aliasing poses a problem for a high view ray sampling resolution (see figure 12, top left). A smoothing or filtering scheme possibly inspired by the one used by Peng [2004] could be employed to alleviate this. In addition, increasing the resolution of the shell geometry and distance texture could lessen the problem.

The technique is much less prone to under-sampling artifacts for shallow view angles than most other real-time ray tracing based shell space rendering techniques. This is due to the distance map approach and a high view ray sampling resolution for flat view angles. Nevertheless, these under-sampling artifacts can appear in case the base mesh is only coarsely tesselated. Increasing the sampling rate for shallow view angles removes these artifacts at the cost of rendering performance.

A problem appearing solely when using height-field shell geometry are rendering artifacts, for view angles nearly tangential to the base surface. It is still unclear what causes these artifacts. However, again the texture space distortion and/or imprecise tetrahedron thickness computations near the tetrahedra face planes are likely to be the cause.

The technique compares favorable with multi layer relief mapping [Policarpo and Oliveira 2006] (see figure 9). However, the overall image quality is slightly better in the rendering produced with multi layer relief mapping, due to the minor rendering artifacts produced by the shell space rendering algorithm.

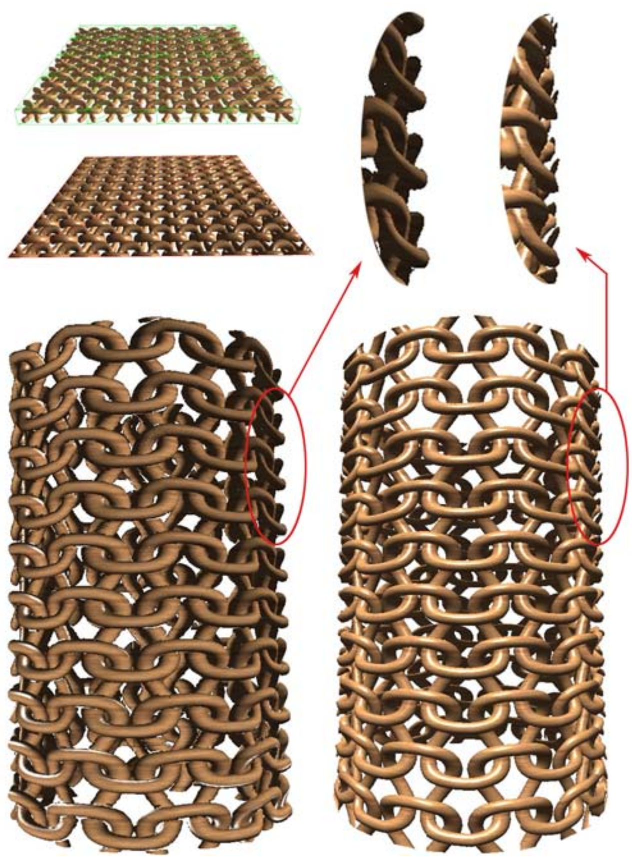


Figure 9: Comparison of multi layer relief mapping by Policarpo and Oliveira [2006] (here 'MLRM') and the shell space rendering technique described in this work ('RSSR'). The magnified areas show that MLRM (right) produces distortions in some silhouette areas whereas RSSR (left) produces correct results. In addition, RSSR outperforms MLRM in this scene configuration (53 fps vs. 30 fps). However, the performance of RSSR is dependent on the kind of shell geometry used, hence the result cannot easily be transferred to other scenarios. In the top left a comparison of a plane rendered with RSSR and a plane rendered with MLRM is shown. The result indicates that RSSR (top) has a clear advantage over MLRM (bottom) because it correctly handles the open surface. (Weave pattern courtesy of Fabio Policarpo.)

7 Conclusion and Future Work

This work introduces a new algorithm for rendering arbitrary non-polygonal geometry into the shell space of a polygonal mesh. The shell geometry is stored as a geometry volume texture. This means no constraints are put on the kind of shell geometry used. For meshes with tiled shell geometry the required texture memory is very low - the shell geometry used for the renderings presented in figures 1, 10 and 11 needs only 1MB texture memory, including the corresponding distance map. Furthermore, the technique supports height-field shell geometry and includes local illumination. Depending on the shell geometry, rendering resolution, the shape and resolution of the base mesh, the technique achieves real-time (25+) to highly real-time frame rates (100+). The use of a 3D distance map for ray sampling reduces the number of shell geometry intersection search steps and yields more precise intersections and smoother silhouettes compared to most linear ray sampling approaches [Policarpo et al. 2005; Tatarchuk 2005; Dummer 2006]. In comparison to all techniques which do not add polygons to the underlying polygonal mesh, the technique introduced in this work correctly handles open surfaces and reduces mapping distortions. Moreover, it overcomes further issues of previous approaches for rendering arbitrary volumetric geometry like limited shell geometry depth complexity [Policarpo and Oliveira 2006], poor performance [Dufort et al. 2005] or high storage requirements and involved per-vertex computations [Wang et al. 2004]. Furthermore under-sampling artifacts for shallow view angles common to most real-time ray tracing based shell space rendering techniques are reduced.

Problems of the technique are texture space distortions most prominent in close-up views due to linear texture coordinate interpolation, aliasing due to the lack of a proper filtering or anti-aliasing scheme and minor rendering artifacts at tetrahedron edges. When using high resolution geometry volume textures, limited texture memory may become an issue.

The main task for future work should be fixing the technique's problems, i.e. removing the texture space distortion and rendering artifacts and adding a filtering/anti-aliasing scheme. Next, further interesting directions for future work are outlined:

- A Level of Detail (LOD) technique which adjusts the ray tracing accuracy for different viewing distances could be incorporated.
- A shadowing technique capable of rendering shadows cast by shell geometry onto itself as well as on other objects could be included.
- A more sophisticated lighting technique including additional global lighting effects like inter-reflections could be added. Precomputed Radiance Transfer (PRT) could be an option for such a lighting technique.
- The pixel shader ray tracing could be sped up. This could be achieved for instance by employing a different (or additional) empty space skipping technique.
- The number of view-ray tracing steps could be made dependent on the tetrahedron thickness to achieve the same sampling accuracy for each view ray segment, utilizing future graphics hardware's (hopefully) improved branching efficiency.
- Support for semi-transparent shell-geometry could be included - preferably without significantly slowing down the algorithm.
- It could be experimented with higher resolution geometry volume textures and distance maps.
- Cone Step Mapping could be incorporated to improve rendering quality and performance for height-field shell geometry.
- A geometry volume texture modeler for modeling and sculpting shell mapped geometry in real-time could be developed.

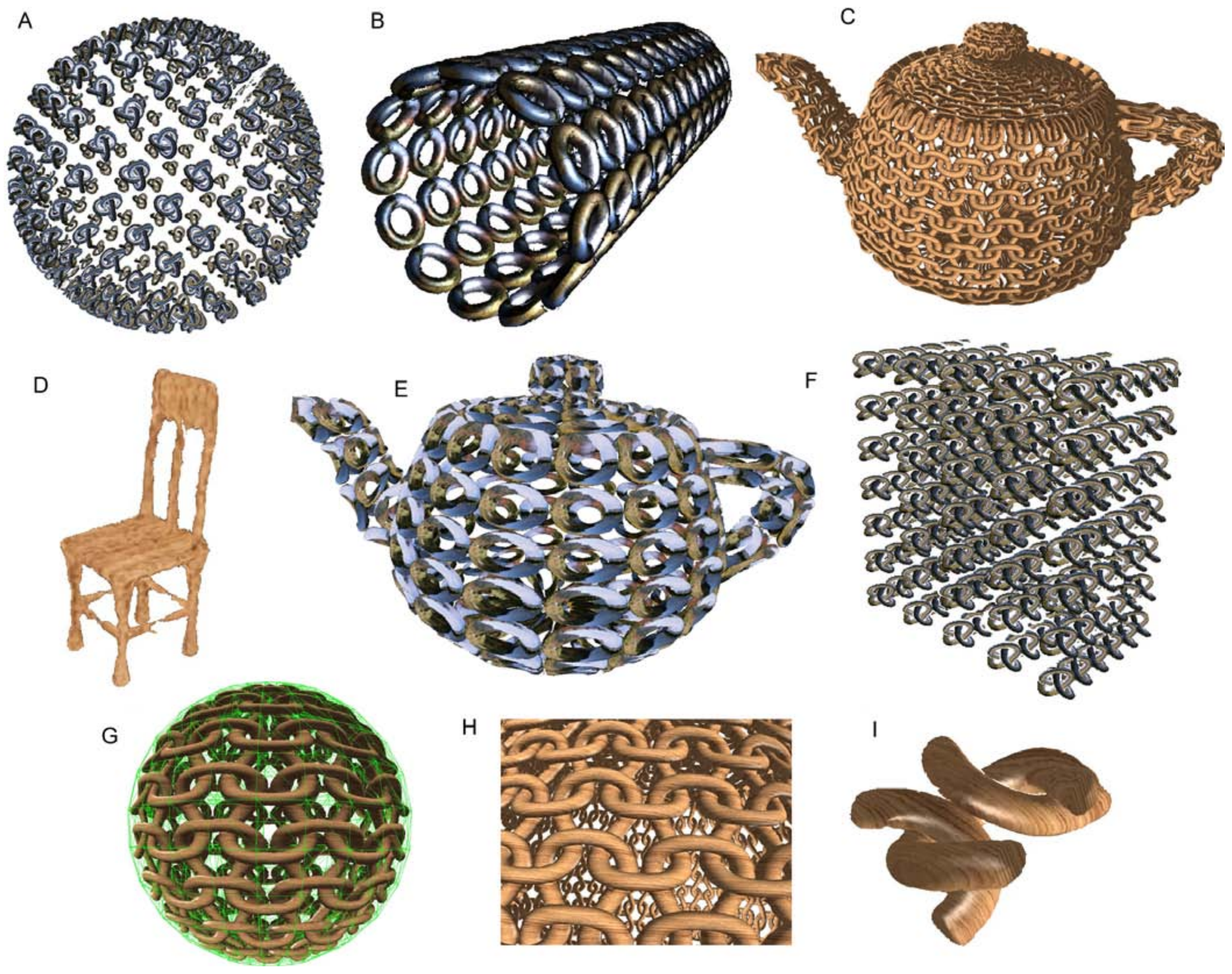


Figure 10: Polygonal meshes with volume texture shell geometry. A: Sphere with torus knot shell geometry (≈35 fps). B: Illuminated cylinder with torus shell geometry (≈83 fps). C: Illuminated teapot with weave shell geometry (≈30 fps). D: Quad base mesh with chair shell geometry (≈190 fps). E: Teapot with torus shell geometry (≈33 fps). F: Quad base mesh with highly tiled torus knot shell geometry (≈100 fps). G: Weave shell geometry sphere with superimposed tetrahedra mesh. H: Weave teapot close-up. I: Single weave tile close-up. (Weave pattern in C, G, H and I courtesy of Fabio Policarpo.)

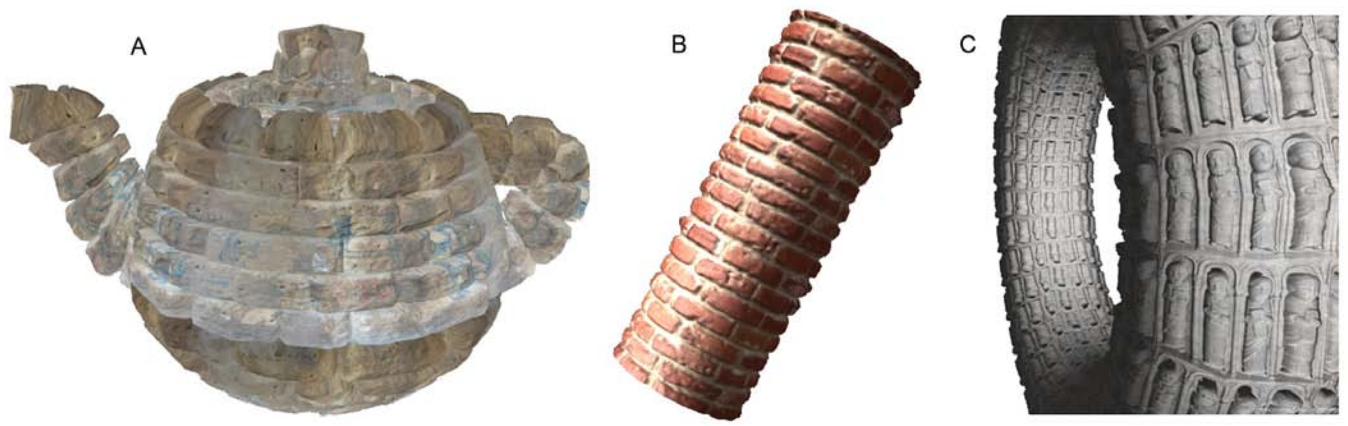


Figure 11: A: Teapot with height-field shell geometry (≈50 fps). B: Cylinder with bricks height-field shell geometry and local lighting (≈100 fps). C: Torus with relief height-field shell geometry and local lighting (≈25 fps).

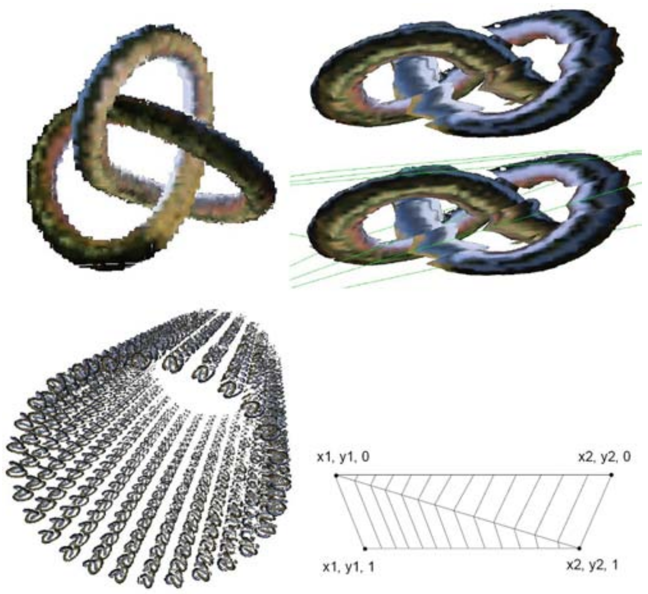


Figure 12: Top left: Aliasing occurs for high ray sampling resolutions. Top right: Distortions appearing in a in close-up view. The superimposed shell mesh reveals the distortions occur close to a tetrahedron slicing plane. Bottom left: Shell geometry vanishes or becomes distorted when seen from far away. Bottom right: Side view of a prism extruded from the base mesh. The two triangles are side faces of the two visible tetrahedra. Conventionally interpolating the texture space over these triangles produces differently oriented texture space z-axes as indicated by the line patterns inside the triangles. This results in severe texture space distortions.

8 Acknowledgement

Thanks to Angela Brennecke and Martin Spindler for valuable suggestions about how to improve this paper.

References

AKENINE-MÖLLER, T., AND HAINES, E. 2002. *Real-Time Rendering*, 2nd ed. A. K. Peters. http://www.realtimerendering.com.

BRAWLEY, Z., AND TATARCHUK, N. 2005. *Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing*. ShaderX^3. Charles River Media, ch. 2.5.

DONNELLY, W. 2005. *Per-Pixel Displacement Mapping with Distance Functions*. GPU Gems 2, Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley Professional, ch. 8, 123–136.

DUFORT, J.-F., LEBLANC, L., AND POULIN, P. 2005. Interactive rendering of meso-structure surface details using semi-transparent 3d textures. In *Proc. Vision, Modeling, and Visualization 2005*.

DUMMER, J., 2006. Cone step mapping: An iterative ray-heightfield intersection algorithm. http://www.mganin.com/lonesock/ConeStepMapping.pdf.

HIRCHE, J., EHLERT, A., GUTHE, S., AND DOGGETT, M. 2004. Hardware accelerated per-pixel displacement mapping. In *Proceedings of the 2004 Conference on Graphics Interface*, Canadian Human-Computer Communications Society, vol. 62 of *ACM International Conference Proceeding Series*, 153–158.

KRAUS, M., QIAO, W., AND EBERT, D. S. 2004. Projecting tetrahedra without rendering artifacts. In *IEEE Visualization Proceedings of the conference on Visualization '04*, 370–375.

OLIVEIRA, M. M., AND POLICARPO, F. 2005. An efficient representation for surface details (rp-351). Tech. rep., Universidade Federal do Rio Grande do Sul (UFRGS), Instituto de Informatica.

PENG, J. 2004. *Thick Surfaces: Interactive Modeling of Topologically Complex Geometric Detail.* PhD thesis, New York University.

POLICARPO, F., AND OLIVEIRA, M. M. 2006. Relief mapping of non-height-field surface details. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games (to appear).*

POLICARPO, F., OLIVEIRA, M. M., AND COMBA, J. L. D. 2005. Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, 155–162.

PORUMBESCU, S. D., BUDGE, B., FENG, L., AND JOY, K. I. 2005. Shell maps. In *Proceedings of ACM SIGGRAPH 2005*, vol. 24 of *ACM Transactions on Graphics*, 626–633.

SHIRLEY, P., AND TUCHMAN, A. 1990. A polygonal approximation to direct scalar volume rendering. In *Proceedings of the 1990 workshop on Volume visualization*, ACM Press, 63–70.

SLOAN, P.-P., LIU, X., SHUM, H.-Y., AND SNYDER, J. 2003. Bi-scale radiance transfer. In *Proceedings of ACM SIGGRAPH 2003*, vol. 22 of *ACM Transactions on Graphics (TOG)*, 370–375.

TATARCHUK, N., 2005. Practical dynamic parallax occlusion mapping. SIGGRAPH 2005 Sketch.

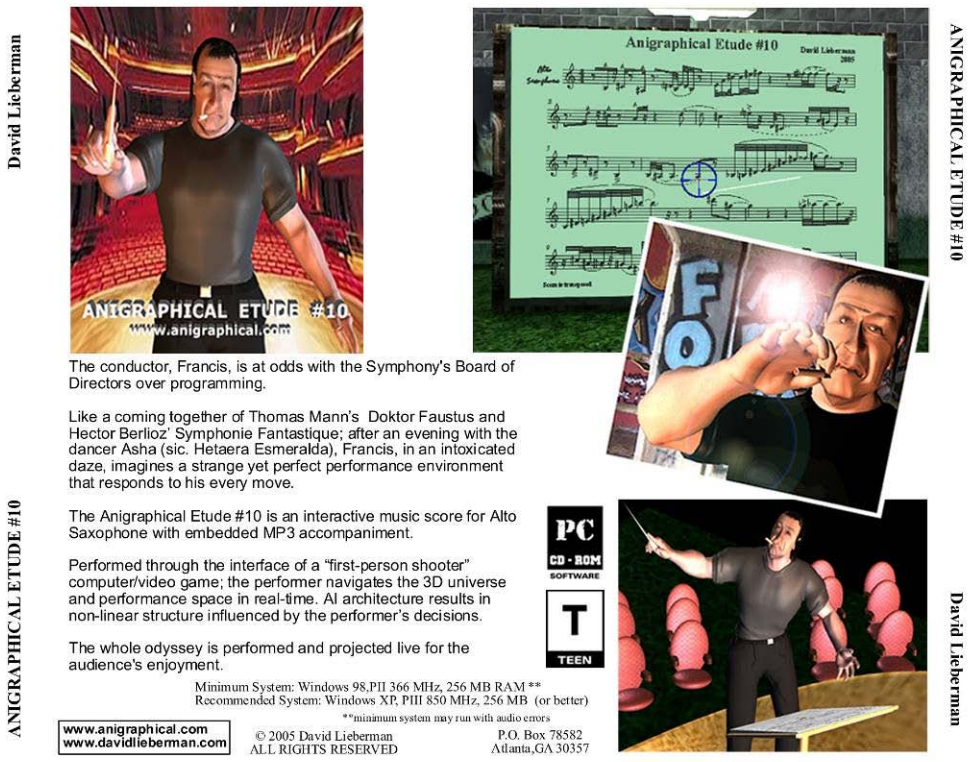
WANG, L., WANG, X., TONG, X., LIN, S., HU, S., GUO, B., AND SHUM, H.-Y. 2003. View-dependent displacement mapping. In *Proceedings of ACM SIGGRAPH 2003*, vol. 22 of *ACM Transactions on Graphics (TOG)*, 334–339.

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

WEILER, M., KRAUS, M., AND ERTL, T. 2002. Hardware-based view-independent cell projection. In *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, 13–22.

Game Enhanced Music Manuscript

David Lieberman



The conductor, Francis, is at odds with the Symphony's Board of Directors over programming.

Like a coming together of Thomas Mann's Doktor Faustus and Hector Berlioz' Symphonie Fantastique; after an evening with the dancer Asha (sic. Hetaera Esmeralda), Francis, in an intoxicated daze, imagines a strange yet perfect performance environment that responds to his every move.

The Anigraphical Etude #10 is an interactive music score for Alto Saxophone with embedded MP3 accompaniment.

Performed through the interface of a "first-person shooter" computer/video game; the performer navigates the 3D universe and performance space in real-time. AI architecture results in non-linear structure influenced by the performer's decisions.

The whole odyssey is performed and projected live for the audience's enjoyment.

Minimum System: Windows 98,PII 366 MHz, 256 MB RAM **
Recommended System: Windows XP, PIII 850 MHz, 256 MB (or better)
**minimum system may run with audio errors

www.anigraphical.com
www.davidlieberman.com

© 2005 David Lieberman
ALL RIGHTS RESERVED

P.O. Box 78582
Atlanta,GA 30357

Real-Time Shell Space Rendering of Volumetric Geometry

Nico Ritsche