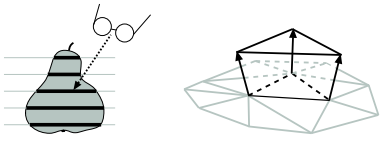
Interactive Semi-Transparent Volumetric Textures

Hendrik P. A. Lensch Katja Daubert Hans-Peter Seidel

Max-Planck-Institut für Informatik*

Abstract

Volumetric textures are often used to increase the visual complexity of an object without increasing the polygon count. Although it is much more efficient in terms of memory to store only the volume close to the surface and to determine the overall shape by a triangle mesh, rendering is much more complicated compared to a single volume. We present a new rendering method for volumetric textures which allows highest quality at interactive rates even for semi-transparent volumes. The method is based on 3D texture mapping where hundreds of planes orthogonal to the viewing direction are rendered back to front slicing the 3D surface volume. This way we are able to correctly display semi-transparent objects and generate precise silhouettes. The core problem is to calculate the intersection of prisms formed by extruding the triangles of the mesh along their normals and the rendering planes. We present two solutions, a hybrid and a purely hardware-based approach.



(a) (b)

Figure 1: *(a) Rendering artifacts can occur if the planes aren't rendered orthogonally to the viewing direction. (b) Prism formed by extruding one triangle of a mesh along its normals.*

1 Introduction

Modeling objects for computer graphics applications always is a compromise between trying to get enough detail to make the object look realistic, while keeping the polygon count low, so that it will render at sufficiently high rates. A well known and often employed method is to have a fairly coarse mesh of the objects surface, and then add realistic detail using some texture-related techniques. In this paper we will focus on techniques in which these textures are volumes, so called volumetric textures [11]. These volumes are applied, usually repetitively, to the surface, with the surface normal vectors controlling the direction of the third texture dimension, giving the surface a certain thickness.

Software-based techniques for visualizing volumetric textures have been known for a long time but are usually too slow for interactive display. Recently, Meyer et al. [16] introduced a hardware-based method for interactively rendering volumetric textures by slicing the volume over each facet. However, since the slicing direction is not necessarily orthogonal to the viewing direction, artifacts can occur at grazing angles (Fig. 1(a)). Furthermore, rendering semi-transparent volumes with this method would require a costly depth sorting of the faces of the base geometry to correctly account for transparency.

In this work we propose an alternative algorithm which assumes the surface geometry to be a triangle mesh. Extruding a triangle along its three normals results in a prism, as shown in Fig. 1(b). We now generate planes in the whole range of the surface volume, from back to front, orthogonal to the viewing direction, and slice each plane with each volume prism. As we always generate planes orthogonal to the viewing direction artifacts are avoided.We can obtain high quality images at interactive rates. The presented algorithm can correctly handle semi-transparent volumetric textures without sorting primitives beforehand.

The main contributions of this paper are two algorithms to efficiently compute the intersection of planes and prisms: a hybrid one, only partly implemented using graphics hardware (Section 5) and a second designed to be mapped fully onto hardware (Section 6).

*{lensch,daubert,hpseidel}@mpi-sb.mpg.de, Stuhlsatzenhausweg 85, D–66123 Saarbrücken.

2 Related Work

The initial idea of applying volumetric textures to a simple object in order to render surface detail consisting of complex geometry has been proposed by Kajiya and Kay [11] who rendered fur on a teddy bear. Perlin and Hoffert [18] extended the approach and modified the surface structure by three-dimensional texture functions, so called hyper textures (see also [23]). Volumetric textures have been successfully applied in the areas of tree and landscape modeling [17, 4], or to improve the appearance of synthesized textiles [9]. A survey on volumetric textures can be found in [7]. Although volumetric textures can replace very complex surface geometry by a simple volume the rendering effort is not necessarily decreased. Most of these techniques use a purely software-based approach for rendering.

Volume rendering has been an active area of research in the last two decades. Software- and hardware-based techniques have been proposed, e.g ray casting [10, 15], splatting [21] or forward projection [8, 22].

The classical approach for hardware-based volume rendering using 3D texture mapping [3, 1] renders several slices through the volume from back to front integrating the pixel intensity. These slices are generated by simple polygons to which the 3D texture is applied. We will basically follow this approach to display volumetric textures.

Different approaches exist to choose the orientation of the textured polygons slicing the volume. One technique precomputes three different sets of slices, each perpendicular to one of the major axes of the volume (see [13, 19, 6]). According to the current viewing direction the best set is selected and displayed. The orientation of the slicing may flip when changing the viewpoint.

Better quality can be achieved when aligning the texture slices in such a way that they are always perpendicular to the viewing direction (screen-space aligned). For each frame the position of the slices and the texture coordinates have to be recomputed by intersecting the volume by a number of planes. We will apply the same technique and compute the slicing on the fly for each single prism constructed from a triangle in the base mesh. For a complete, coherent review on volume visualization techniques see [2].

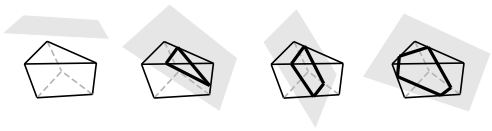


Figure 2: *Plane-prism intersections can result in triangles, quadrilaterals or pentagons.*

Meyer et al. [16] combined a hardware-based volume rendering approach with volumetric textures using three sets of orthogonal slices. In order to minimize artifacts at grazing viewing angles, Meyer et al. introduce certain criteria to control the number of slices needed, depending on the viewing direction and a maximal "depth" the user is allowed to see between the slices of a volume. A significant drawback of this method is, that it explicitly can not handle semi-transparent volumes, as this would require sorting the facets from back to front for each view. As we globally generate planes from back to front and then render the intersections with the prisms building the volume, our method is capable of correctly displaying semi-transparencies, as demonstrated e.g. in Fig. 7.

Lengyel et al. [14] render fur using volumetric textures, by displaying the object in concentric shells from the body outwards, similar to the approach by Meyer et al. In order to deal with artifacts near grazing viewing angles, "fin" polygons are placed orthogonal to the surface in silhouette regions and textured. This approach however breaks down for regular structures, and volumes with larger transparent regions.

3 Prisms and Planes

The input required for the algorithms presented in the next sections is a 3D volume texture as well as a 2D surface description, including surface normals. We will restrict ourselves to triangle meshes, which can easily be constructed from other meshes by tessellation.

For each triangle in the base mesh the three normals at the vertices span a prism. The thickness of the volume over a triangle can

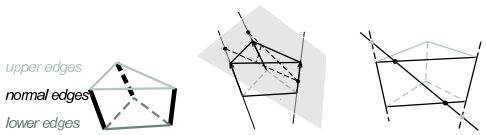
be varied by assigning different lengths to the normals. If a surface triangle's normals vary strongly, self-intersecting prisms might occur, leading to invalid results during rendering. Degenerated cases need to be excluded in the construction phase. By assigning texture coordinates to the six vertices of the resulting prism we map the 3D volume data set into the prism.

As we will need to refer to the prism's edges later on, we will introduce the following names: The three edges belonging to the original mesh triangle will be called *lower edges*, the three edges corresponding to the normals we will refer to as *normal edges*, and the three edges connecting the normal's endpoints to a new triangle are the *upper edges* (see Fig. 3(a)).

Volume rendering is done by generating planes from back to front and intersecting them with all prisms. To obtain the highest quality we will always orient these slices perpendicular to the viewing direction. We determine the location of the last and first plane using the bounding volume of all the prisms.

The main problem we have to solve for rendering is to find the intersection of the current slice with the prism. A first step to compute the intersection polygon is to classify the intersection based on the intersection of the plane with prolonged normal edges (Fig. 3(b)). The plane intersects each normal edge either above (case a), within (case b) or below the prism (case c). Based on this classification we obtain 27 different cases how a plane can intersect a prism. Due to symmetries we can reduce them to four basic cases, shown in Fig. 2: no intersection, intersections resulting in a triangle, in a quadrilateral, or a pentagon. Furthermore, the classification of the normal edges into a, b, c also determines which of the nine edges of the prism will be intersected.

This information will be used in the following slicing algorithms. The first one is implemented in software using the hardware just for rendering, followed by a hybrid approach where the classification is done in software while the actual intersection is performed within a vertex program. In Section 6 the entire plane/prism intersection is done on the graphics board.



(a) (b) (c)

Figure 3: *(a) Nomenclature of the prism's edges. (b) We classify the normal edges by the position of the intersection point of the plane with the normal edge (here: above (a), within (b), above (a)). (c) The software algorithm considers two classified normal edges at once. The classification (here a and c) decides which of the prism edges to intersect (see Section 4).*

4 Software Slicing

In the first implementation, given a plane and a prism, we compute the intersecting polygon in software and render it using OpenGL. Assuming the prism's normal edges are classified as explained above, what remains is to find the intersections of the plane with all nine edges. In addition, these intersections have to be in the correct order to define a polygon.

Instead of intersecting the nine edges separately we consider the intersections of the plane with the quadrilateral spanned by two neighboring normal edges at a time (Fig. 3(c)). The classification directly determines which of the four edges will be intersected. For instance, if we know that the first normal edge is classified as a (above), and the next as c (below), the plane must intersect the upper and the lower edge connecting both normal edges, as depicted in Fig. 3(c). To avoid considering intersection points twice, we decide that only intersections with the first normal edge will be drawn for this quadrilateral.

Visiting the quadrilaterals in the same order as the edges in the original triangle mesh will ensure that the resulting polygon is correctly oriented and the vertices are issued in the correct order.

The algorithm for slicing a surface volume with a number of planes using the idea explained above is given in Fig. 4. A number of slices through the complete object are rendered from back to front. For each plane intersection tests are performed with all prisms. Bounding spheres are used to detect trivial

```
for each plane // (from back to front)
  classify normal edges;
  for each prism // (each corresp. to mesh facet)
    if (!trivial reject)
      glBegin( GL_POLYGON )
      for each pair of normal edges [i, i+1]
        look at classifications (c[i], c[i+1])
        if(a,a) ; // do nothing, plane is above
        if(a,b) isect(upper);
        if(a,c){isect(upper);isect(lower);}
        if(b,a){isect(normal);isect(upper);}
        if(b,b) isect(normal);
        if(b,c){isect(normal);isect(lower);}
        if(c,a){isect(lower);isect(upper);}
        if(c,b) isect(lower);
        if(c,c) ; // do nothing, plane is below
      end for;
      glEnd();
    end if
  end for;
end for;
```

Figure 4: *Software algorithm.*

cases where the prism is not intersected at all. Otherwise, the edge intersections are computed based on the classification. The `isect` subroutine computes the intersection point with the given edge, interpolates texture coordinates, and issues the corresponding `glVertex` and `glTexCoord` commands.

5 Hybrid Algorithm

To make this algorithm more efficient, we will now map parts of the `isect` routine onto hardware. In the following we will rely heavily on vertex programs, a feature available on newer graphics cards. They consist of assembler style code working on 4-vector registers which are called for every vertex before rasterization. The input consists of attributes for each vertex, as well as global program parameters. Vertex programs control a vertex's color, texture coordinates, position, normal etc. [24].

As just mentioned, a vertex program is called for each vertex. Unfortunately there is no way to decide, inside a vertex program, that this vertex should not be rendered. As a consequence we have to know in advance how many vertices we want to render, which makes it impossible, at a first glance, to put the classification of the normal edges into a vertex program.

However, the intersection of a plane and a line can easily be computed with a vertex program. The plane parameters (normal and point on plane) are set as program parameters. The line's beginning and end point are passed to the vertex program as vertex attributes. The vertex program computes the intersection and sets the position of the output vertex correspondingly. Additionally, we can pass along other vertex attributes like the texture coordinates corresponding to both points and have them interpolated to set the texture coordinate of the output vertex.

To use this vertex program with our software algorithm explained above, a fake vertex is set up for each of the prism's nine edges and stored in a vertex array. This means, for each prism edge we generate vertex attributes (position and texture coordinates) for the starting and end point. The display routine still looks like Fig. 4 and computes the classification of every normal edge in software. However, the `isect` routine is changed to now call `glVertex` for the fake vertex corresponding to the prism edge determined by the classification.

In our implementation we do not generate all nine edges (fake vertices) for each prism, instead we make use of the fact that edges are shared by neighboring prisms. This drastically reduces the amount of data stored in main memory. (For the torus mesh in Fig. 8 we need 243 kB of memory for the attributes, reduced to only 109 kB when sharing edges.)

6 Hardware Algorithm

As already mentioned, the main problem with writing a vertex program for slicing planes and prisms lies in the differing number of vertices the resulting polygon may have. While the presented hybrid algorithm decides which of the prism's lines to intersect based on a software classification step, we will now present an algorithm which is fully implemented as a vertex program. Although this algorithm currently is not faster than the hybrid one, we think that it will be superior in the near future since the performance of graphics boards currently is increasing faster than processor speed. After explaining the strategy, we will show how to map it to a vertex program.

6.1 Strategy

The key idea of this method is to render the fixed number of six vertices per prism, two corresponding to each quadrilateral spanned by two normal edges as in Fig. 3, or to put it another way two corresponding to each normal edge. The vertices are named $v0_l$, $v0_r$, $v1_l$, $v1_r$, $v2_l$, $v2_r$, indicating the number of the normal and the quadrilateral (left/right).

In order to figure out where to place each vertex we assign five edges to each vertex: the corresponding normal edge and the adjacent two upper and two lower edges. We will call one set of corresponding upper and lower edges the *primary edges* and the other set the *secondary edges*. Fig. 10 visualizes the normal and primary edges that are assigned to the six vertices by different colors. Unfortunately, this setup prevents us from sharing data between neighboring prisms since primary and secondary edges will be different for every primitive.

The position of each vertex will be set to the intersection of the plane with one of the five assigned edges. The strategy used to intersect edges and to choose positions is listed in Fig. 5: First we try to intersect with the normal edge. If no intersection can be found, we intersect the two primary edges and choose the intersection closer to the normal edge. If still no intersection occurs we intersect the secondary edges and again choose the intersection closer to the normal edge. If none of the three cases hold, the plane does not intersect the prism at all. In this case all vertex positions will be set to somewhere outside the scene.

As we render six vertices, but the resulting intersection polygons have at most five vertices, several vertices will be mapped to the same positions as their neighbors. All six vertices will be displayed by the graphics board which does not affect the rendering quality. Fig. 6 demonstrates which vertices take care of rendering which corner of the triangle, quadrilateral and pentagon depicted in Fig. 2.

If the vertices are rendered in the correct order, i.e. first the left then the right vertex corresponding to each corner ($v0_l$, $v0_r$, $v1_l$, $v1_r$, $v2_l$, $v2_r$) using the above strategy we obtain a correctly oriented intersection polygon.

```
// — case 1: normal edge —
λ = intersect(normal);
if ( λ ∈ [0...1] ){
  interpolate(normal, λ); return;}

// — case 2: primary edges —
λu = intersect(upper primary);
if ( λu ∉ (0...1] ) λu = 2.0;
λl = intersect(lower primary);
if ( λl ∉ (0...1] ) λl = 2.0;
if ( λu < λl and λu ∈ (0...1] ){
  interpolate(upper primary, λu);return;}
if ( λl < λu and λl ∈ (0...1] ){
  interpolate(lower primary, λl);return;}

// — case 3: secondary edges —
λu = intersect(upper secondary);
if ( λu ∉ (0...1] ) λu = 2.0;
λl = intersect(lower secondary);
if ( λl ∉ (0...1] ) λl = 2.0;
if ( λu < λl and λu ∈ (0...1] ){
  interpolate(upper secondary, λu);return;}
if ( λl < λu and λl ∈ (0...1] ){
  interpolate(lower secondary, λl);return;}

setVertexToNirvana();
```

Figure 5: *Order in which each vertex tries to intersect the assigned edges. Setting invalid values to 2.0 avoids accidently selecting an out-of-range value with the "<"-operator.* `interpolate` *interpolates a vertex position and a texture coordinate from the end points of the corresponding edge using the given* λ.

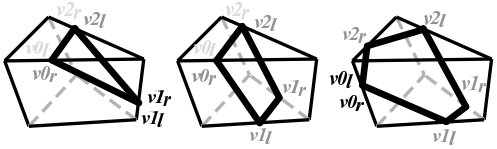


Figure 6: *Color of vertices corresponds to applicable case – black: normal edge (case 1), dark grey: primary edge (case 2), light grey: secondary edge (case 3).*

6.2 Implementation

In this section we will explain the important steps when implementing the algorithm using vertex programs. The vertex program is setup to provide each vertex with six points marking the beginning and end point of the edges, and the six corresponding texture coordinates. The plane parameter and normal, as well as a few constants are passed as program parame-

ters.

The most critical point when coding the algorithm in a vertex program is that there are no statements to control the program flow. Vertex programs are designed to be able to run in parallel for all vertices at once and therefore are linear in their execution. All code in a vertex program is executed. This means that if we implement the different cases shown in Fig. 5 we will have to compute all cases and then take care that only the results of the correct case are finally chosen.

The structure of our vertex program can be split into two parts. First we compute all five intersections or λ-values (one for the normal case, two for the primary edge case, two for the secondary edge case), which is easy to code. In the second part we select the correct case, based on the previously computed values. This part is more complicated, as we have to somehow emulate the if-statements, e.g using instructions which set a register differently, depending on the value of another register. `MIN` / `MAX` assign the component-wise minimum / maximum of two source vectors to a destination register, and `SLT` (set on less than)/ `SGE` (set on greater than or equal to) perform a component-wise assignment of either 0.0 or 1.0 into the destination register depending on two source registers.

We handle the selection of the correct case using six registers. Five of these registers, which we will call *validity registers*, each correspond to one λ-value and will be set to one if the corresponding λ-value is in the correct range, to zero otherwise. For instance:

```
SGE tmp1, λ, 0.0; // tmp1 = 1 if λ ≥ 0.0
SGE tmp2, 1.0, λ; // tmp2 = 1 if 1 ≥ λ
MUL valid, tmp1, tmp2; // combine
```

In the primary case and secondary case, the validity registers also control which of λ_{upper} and λ_{lower} to choose (the smaller value in the correct range, or none if both are out of range). At the end the final λ-value λ_{res} is computed as a weighted sum of all λ-values, with the validity registers as weights.

Before actually computing the weighted sum, we have to make sure that only one λ is selected. Here we have to respect the order given by Fig. 5, the normal case is preferred to the primary case, and this case again is preferred to the secondary case. We use a sixth register `sel` for this task. It is initially set to one. The weighted sum then is computed step by step. After each addition the selection register is updated. It will be zero after the first valid λ has been encountered:

```
λ_res = λ_res + sel * valid_i * λ_i
sel = (1 - valid_i) * sel
```

Analogous to selecting λ, we use the weighted sum with the same weights to select the correct points and texture coordinates between which to interpolate.

Note, that some of the computed λ-values could be infinity, which leads to invalid results when multiplying with zeros (in the selection or validity registers). Therefore we upper bound all computed λ-values to 2.0, using the `MIN`-statement.

7 Results and Discussion

We implemented both the hybrid approach and the pure hardware algorithm on two PCs, one with a GeForce3, one with a GeForce4 graphics card, both with an AMD Athlon 1GHz processor. We tested both algorithms for the semi-transparent data set shown in the middle of Fig. 8 on different surfaces: the distorted torus seen in the same figure which consists of 576 triangles, and the terrain from Fig. 7 which has 3200 triangles. The results for both algorithms for a varying number of planes can be seen in Table 1.

We can render arbitrary volumetric textures at high interactive rates using the hybrid solution. The rendering times of this method currently even exceed those of the hardware solution, which we explain with a better load balancing between the CPU and the graphics card, as the CPU computes the vertex classification in this algorithm, whereas all decisions are left the vertex program in the pure hardware solution. The amount of computation time needed for the software classification in relation to the complete rendering time is about 22% for the torus scene, and about 46% for the terrain scene.

		Torus		Terrain	
planes	board	hyb.	hw	hyb.	hw
250	GF3	38	10/1.6	14	6/0.3
	GF4	38	25/4	13	12/0.7
500	GF3	20	5/0.8	7	3/0.1
	GF4	20	13/2	7	6/0.4
1000	GF3	12	3/0.4	4	1.4/0.1
	GF4	12	6/1	4	3/0.2

Table 1: *Comparison of rendering times (in fps) for different algorithms. Second number in hardware column gives rates without trivial reject. Image resolution: 512 × 512.*

Comparing the rendering times for the two different graphics cards we observe that the rendering times for the hybrid solution are fairly identical, whereas the hardware solution already computes considerably faster on a GeForce4, nearly achieving the rates of the hybrid solution. We attribute this to the fact that vertex programs execute more efficiently on a GeForce4, and are confident that for future graphics boards the hardware solution will overtake the hybrid solution since the performance of graphics boards is currently increasing faster than the performance of processors.

We tested two different implementations of the hardware algorithm, one using a display list to render the whole scene, i.e. computing the intersections for one plane, and another using vertex arrays, stored in AGP-memory. We obtained identical rendering times for both implementations, from which we conclude that the bottleneck for the hardware algorithm is the execution time for the vertex programs (hardware algorithm's vertex program: 107 instructions, hybrid algorithm: 28 instructions). To store the vertex attributes for the torus scene we need 243 kB for the hybrid approach and 486 kB for the hardware approach.

Adding a trivial reject test based on bounding spheres to the hardware implementation, 86% of the triangles are rejected for the torus scene and even 95% for the terrain scene. The resulting speed-ups (torus scene: 6×, terrain scene: 14–15×) are due to the smaller number of vertex programs being executed.

The presented algorithms can either be used to render the volume as is (Fig. 8 (top)), in which the semi-transparent egg sitting in the fully opaque torus was simply mapped onto the geometry), or combined with pixel shading (in Fig. 8 (bottom), 9, 11, and 12 we applied simple Phong lighting).

Fig. 9 and 12 show volumes without semi-transparencies with 128×128×128 voxels. The car park was sliced with 1500 planes to get subtle details, whereas 200 planes fully suffice for the chain data-set. The volume for the outdoor scene in Fig. 7 consists of a mixture of fully opaque (trees, floor, flowers) and semi-transparent voxels (ground fog, smoke). The image was rendered with 1000 slices. Notice how the nearly transparent ground fog only becomes visible at grazing angles. The volcano-scene in Fig. 11, which also consists of a partly semi-transparent volume for the smoke rings, was sliced with 4500 planes and greatly profits from a simple per-pixel lighting algorithm.

7.1 Per-Primitive Programs

Even though the bottleneck for the hardware rendering algorithm doesn't seem to be the data transfer from and to the graphics card, a considerable amount of data could be saved if there were a per-primitive program. This program could be given all the data for one primitive (position and texture coordinates for six vertices), instead of passing each vertex all attributes like we currently are forced to do in the hardware algorithm. (In our case that would lead to a data reduction to 1/6). The vertex program would also be simpler to code: the λ-values could be computed for all nine edges, then, depending on the different values, we would select the order in which to render the intersections.

If a per-primitive program could decide how many vertices to render, or just not to render a vertex, we could avoid scenarios like ours, where we are forced to place several vertices at the same position or to project vertices outside the scene if they needn't be drawn.

8 Conclusion and Future Work

This paper presents a new method for hardware accelerated rendering of volumetric textures applied to triangle meshes. We propose a hybrid (software/hardware) and a pure

hardware-based algorithm to efficiently perform plane/prism intersections. Using the hybrid algorithm to render volumetric textures we achieve high interactive frame rates on current graphics hardware. Although the pure hardware algorithm performs slower at present we expect it to overtake the hybrid algorithm on future graphics platforms offering more efficient execution of vertex programs.

The presented method is the first to correctly handle semi-transparent textures in hardware at interactive rates. Semi-transparent volume textures require rendering slices through the complete object from back to front as it is done with our technique. Rendering a stack of slices per prism at once [16] would require a careful sorting of the prism with respect to their distance to the viewer which is costly.

Arbitrary materials like fur [14], textiles [9, 5], etc. can be rendered on a 2D surface in hardware using our method. Another possible application would be to render displacements in hardware as proposed by Kautz et al. [13], using our method to generate the intersection polygons and thereby removing artifacts due to non-orthogonal viewing directions.

As future work, we would like to combine the volumetric rendering with more complex shading models, e.g. anisotropic ones [12]. The design of rendering planes from back to front should make it possible to apply other, more sophisticated rendering methods than direct volume rendering, for instance iso-surface extraction, LIC, and others listed in [20].

References

[1] K. Akeley. Realityengine graphics. In *Proc. of SIGGRAPH 93*, pages 109–116, 1993.

[2] K. Brodlie and J. Wood. Recent advances in volume visualization. *Computer Graphics Forum*, 20(2):125–148, 2001.

[3] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *IEEE Volume Visualization Symp.*, pages 91–98, 1994.

[4] N. Chiba, K. Muraoka, A. Doi, and J. Hosokawa. Rendering of forest scenery using 3d textures. *The Journal of Visualization and Computer Animation*, 8(4):191–199, 1997.

[5] K. Daubert and H.-P. Seidel. Hardware-based Volumetric Knit-Wear. In *Proc. of Eurographics 2002*, 2002.

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

[7] J. M. Dischler and D. Ghazanfarpour. A survey of 3d texturing. *Computers & Graphics*, 25(1):135–151, February 2001.

[8] G. Frieder, D. Gordon, and R. Reynolds. Back-to-front display of voxel-based objects. *IEEE Computer Graphics and Applications*, 5(1):52–59, 1985.

[9] E. Gröller, R. T. Rau, and W. Straßer. Modeling textiles as three dimensional textures. In *Eurographics Rendering Workshop 1996*, pages 205–214, 1996.

[10] L. Tuy H. Tuy. Direct 2D display of 3D objects. *IEEE Computer Graphics and Applications*, 4(10):29–33, 1984.

[11] J. T. Kajiya and T. L. Kay. Rendering Fur With Three Dimensional Textures. In *Proc. of SIGGRAPH 89*, 1989.

[12] J. Kautz and H.-P. Seidel. Towards Interactive Bump Mapping with Anisotropic Shift-Variant BRDFs. In *Proc. EG/SIGGRAPH Workshop on Graphics Hardware*, pages 51–58, 2000.

[13] J. Kautz and H.-P. Seidel. Hardware Accelerated Displacement Mapping for Image Based Rendering. In *Proc. of Graphics Interface 2001*, pages 61–70, 2001.

[14] J. Lengyel, E. Praun, A. Finkelstein, and H. Hoppe. Real-Time Fur over Arbitrary Surfaces. In *Symposium on Interactive 3D Graphics*, pages 227–232, 2001.

[15] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[16] A. Meyer and F. Neyret. Interactive Volumetric Textures. In *Proc. of Eurographics Workshop on Rendering*, pages 157–168, 1998.

[17] F. Neyret. Synthesizing verdant landscapes using volumetric textures. In *Eurographics Rendering Workshop 1996*, pages 215–224, June 1996.

[18] K. Perlin and E. M. Hoffert. Hypertexture. In *Proc. of SIGGRAPH 89*, pages 253–262, July 1989.

[19] G. Schaufler. Per-Object Image Warping with Layered Impostors. In *Proc. of Eurographics Rendering Workshop*, pages 145–156, 1998.

[20] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proc. of SIGGRAPH 98*, pages 169–178, July 1998.

[21] L. Westover. Footprint evaluation for volume rendering. In *Proc. of SIGGRAPH 90*, pages 367–376, 1990.

[22] J. Wilhelms and A. van Gelder. A coherent projection approach for direct volume rendering. In *Proc. of SIGGRAPH 91*, pages 275–284, 1991.

[23] S. P. Worley and J. C. Hart. Hyper-rendering of hyper-textured surfaces. In *Implicit Surfaces*, pages 99–104, 1996.

[24] C. Wynn. OpenGL Vertex Programming on Future-Generation CPUs. Slide collection, available from www.nvidia.com.

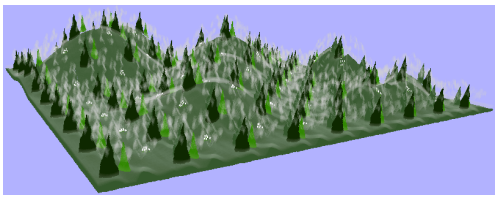


Figure 7: *Volumetric texture with semi-transparent parts (smoke, ground fog) applied to terrain mesh.*

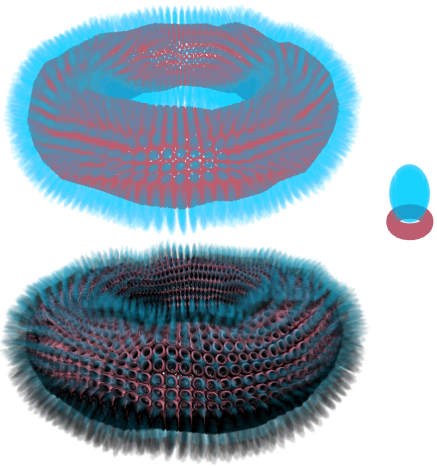


Figure 8: *Volume data set (*$128 \times 128 \times 128$*), consisting of an opaque torus supporting the semi-transparent egg (right). Top: volume rendered as is. Bottom: volume rendered in combination with a simple per-pixel lighting algorithm.*

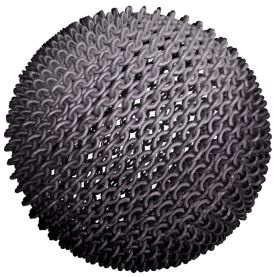


Figure 9: *The volume consists of 2 chain-links and was rendered with 200 planes. We use per-pixel lighting. Notice the precise silhouettes.*

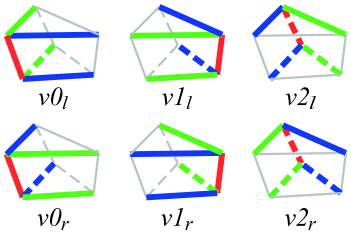


Figure 10: *For the pure hardware algorithm we construct the six fake vertices* $v0_l, v0_r, v1_l, v1_r, v2_l, v2_r$*. Each of them is assigned a normal edge (red), two primary edges (green) and two secondary edges (blue). The* l *or* r *subscripts define in which direction (left/right) the primary edges are oriented.*

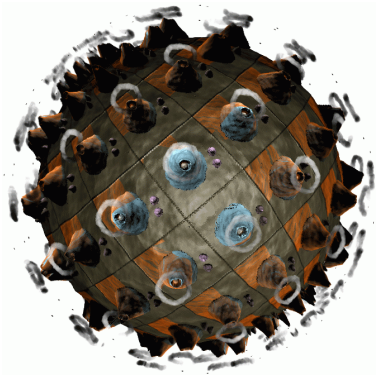


Figure 11: *Volume consisting of opaque (volcano, floor), semi-transparent (smoke), and fully transparent parts. The volume is lit using a simple per-pixel lighting algorithm. (volume resolution:* $128 \times 128 \times 128$*)*

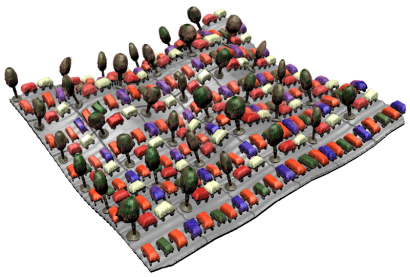


Figure 12: *Volume (*$128 \times 128 \times 128$*) consisting of several cars, a ground plane and some trees, assembled to a car-park scene (rendered with 1500 planes), using a simple lighting algorithm.*