Hardware-Based View-Independent Cell Projection

Manfred Weiler, Martin Kraus, and Thomas Ertl[*]
Visualization and Interactive Systems Group, Universität Stuttgart, Germany

Abstract

We present the first, view-independent cell projection algorithm for off-the-shelf programmable graphics hardware. Our implementation performs all computations for the projection and scan conversion of a set of tetrahedra on the graphics hardware and is therefore compatible with many of the hardware-accelerated optimizations for polygonal graphics, e.g. OpenGL vertex arrays and display lists. Apart from our actual implementation, we discuss potential improvements on future, more flexible graphics hardware and applications to interactive volume visualization of unstructured meshes.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Bitmap and framebuffer operations, Display algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture, Raytracing

Keywords: cell projection, pixel shading, programmable graphics hardware, ray tracing, tetrahedral meshes, unstructured meshes, volume rendering, volume visualization

1 Introduction

Without doubt there is a predominance of polygons in interactive, three-dimensional computer graphics. But even more importantly, the development of new mass-market graphics hardware is — apart from very few exceptions — driven by the needs of fast polygon-based rendering; thus, the performance gap between polygonal graphics and alternative approaches, e.g. volume graphics, is in fact widening despite recent progresses in volume graphics. This development is not only a serious challenge for the volume visualization community, but also offers great opportunities for technical advances in volume rendering — provided that the potential of new hardware features is successfully exploited for volume graphics.

Our ultimate goal are volumetric graphics primitives that are as well supported by graphics hardware as polygonal primitives are today. Given such primitives, rendering of complex volumes can be performed by decomposing them into tetrahedra and scan convert each cell in the same manner as complex surfaces are decomposed into single triangles for rasterization by the graphics hardware. In fact, it was demonstrated in [8] that hardware-supported scan conversion of tetrahedral primitives would dramatically accelerate unstructured volume rendering. The primary advantage of hardware-based rasterization of tetrahedral cells is the possibility to exploit graphics hardware optimizations, e.g. OpenGL display lists, vertex arrays, etc. More generally spoken, the amount of transferred data is reduced and higher transfer rates are achieved as the main CPU may send sets of tetrahedral primitives to the graphics board without any preprocessing — at the same rate as polygonal primitives are sent.

However, instead of proposing yet another hardware system for volume rendering, we present a tetrahedral cell projection algorithm that is suitable for off-the-shelf programmable graphics hardware, in particular nVidia's GeForce3 graphics chip. Our algorithm is "view-independent" in the sense that the very same per-pixel operations are performed independently of the viewing parameters, which is a crucial requirement for an implementation on today's programmable graphics hardware. Therefore, all the computations required for the projection and scan conversion of tetrahedral cells are performed on the graphics board and the only remaining task of the main CPU is to transfer the view-independent data specifying the tetrahedra to the graphics subsystem. In the case of a small number of tetrahedra, this data may actually be buffered on the graphics board, such that the rendering performance is not limited by the bandwidth of the graphics bus.

Thus, our approach promises to overcome several limitations of previously published, hardware-accelerated cell projection algorithms, which are briefly reviewed in Section 2. A detailed description of our algorithm is given in Section 3 before its implementation is presented in Section 4. Due to limitations of today's programmable graphics hardware, we were forced to design several workarounds, which are also described in detail as they are part of the current implementation. Based on integration of different optical models demonstrated in Section 5, we discuss the application of our approach to volume visualization of unstructured meshes in Section 6 and report performance results in Section 7. Future work, in particular with respect to future graphics hardware, is discussed in Section 8.

2 Previous Work

Cell projection is a well-known volume visualization technique for unstructured meshes. More precisely spoken, a scalar field is visualized, which is specified by scalar values at all vertices of a mesh. Of particular interest are tetrahedral meshes, since the scalar values may be linearly interpolated within tetrahedral cells. Usually, the scalar field is mapped to colors and opacities by transfer functions.

Early cell projection algorithms, e.g. by Max et al. [13], did not employ any graphics hardware but relied on a rasterization of the cells in software. This approach is still of interest, see for example Farias et al. [5], because of its flexibility and its suitability for parallelisation; however, hardware-accelerated algorithms turned out to be considerably faster under most circumstances. In the past, almost all of these algorithms were based on the Projected Tetrahedra (PT) algorithm, which was first published in [15].

[*]IfI, Universität Stuttgart, Breitwiesenstr. 20-22, 70565 Stuttgart, Germany. E-mail: {Manfred.Weiler | Martin.Kraus | Thomas.Ertl} @informatik.uni-stuttgart.de .

The algorithm by Shirley and Tuchman exploits hardware-accelerated triangle scan conversion by decomposing projected tetrahedra into triangles and rasterizing these triangles. The correct color and opacity are computed by ray integration along the view direction according to some optical model, e.g. the volume density model by Williams and Max [17, 12, 18]. While the volume rendering integral is evaluated correctly at the triangle vertices, the colors and opacities of all remaining pixels have to be interpolated linearly, which causes rather strong artifacts. Moreover, the triangles are semi-transparent; thus, they have to be rendered in the correct visibility order, i.e. the tetrahedral mesh has to be sorted.

Therefore, subsequent research was mainly focused on the problem of efficiently sorting unstructured meshes [13, 19, 1, 20], in particular non-convex meshes [16, 2] and cyclic meshes [9]. Considerably fewer publications, e.g. Stein et al. [16] and Röttger et al. [14], were concerned with a more accurate but still hardware-accerelated evaluation of the volume rendering integral.

Recently, a graphics hardware architecture was proposed by Wittenbrink [21] together with King et al. [8], which permits to rasterize and sort tetrahedral meshes in hardware. Unfortunately, this hardware architecture was not built yet.

3 Scan Conversion of Tetrahedra

As mentioned in the introduction, a hardware-based, view-independent scan conversion of tetrahedra would enable us to employ optimization techniques, such as OpenGL display lists or vertex arrays. These cannot be used in the PT algorithm, since there are several cases of the Shirley-Tuchman decomposition, of which one has to be selected depending on the view parameters.

In contrast to this, our approach moves all view-dependent computations to the graphics board and achieves view-independency by a rasterization technique similar to ray casting.

3.1 View-independent Cell Projection

In order to compute one ray for each pixel covered by the projected tetrahedron, we render its front faces with the scalar values of the volume specified as vertex colors. Note that, although we are only interested in the visible faces, view-dependent data is not required since we can assure that only front faces are rasterized by employing OpenGL backface culling.

As suggested in [14], we use a texture map to accurately evaluate the ray integral within the tetrahedron allowing arbitrary transfer functions. The variables required as texture coordinates are the scalar value s_f at the entry point, the scalar value s_b at the exit point, and the length l of the viewing ray within the cell; see Figure 1.

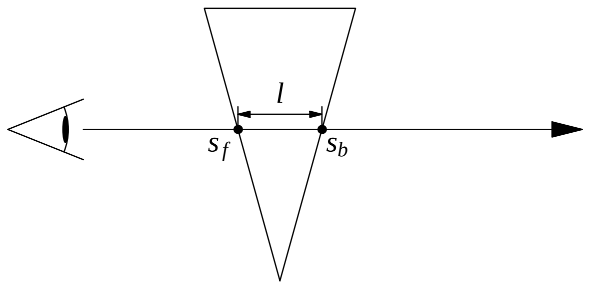

Figure 1: Intersecting a tetrahedral cell with a viewing ray. s_f and s_b are the scalar values on the front and back face respectively; l denotes the thickness of the cell for this ray.

As we set the vertex colors to the scalar values of the mesh, the linear interpolation of vertex colors performed by the graphics hardware provides the scalar value s_f on the entry (front) face per fragment. The only remaining effectively unknown parameter is the thickness l of the cell for the viewing ray, since the third parameter s_b can be easily computed from s_f and l using the gradient $\vec{g}$ of the scalar field within this tetrahedron and the normalized direction $\vec{d}$ of the viewing ray:

$$s_b = s_f + (\vec{g} \cdot \vec{d})l \; . \tag{1}$$

Thus, the remaining problem is to determine the correct per-fragment thickness of the cell l, which is equivalent to the problem of finding the distance at which a viewing ray exits the tetrahedron.

We will first describe the idea in general without considering limitations of current graphics hardware.

3.2 Computation of Exit Points

In this section we will assume, that any per-tetrahedron data is available at each fragment, in particular the face normals and the scalar field's gradient. Also, the coordinates of the entry point, which corresponds to the fragment, have to be known. The latter can easily be achieved by either explicitly defining the vertex positon as texture coordinates or using automatic texture coordinate generation with an appropriate parametrization.

The task of determining the length of the viewing ray inside a tetrahedron is quite similar to the problem of clipping a three-dimensional line against a convex polyhedron, which in this case is a tetrahedron. However, we are only interested in the exit point, as the entry point is already known. The two-dimensional analogue is depicted in Figure 2. Let $\vec{v}$ be the entry point corresponding to the fragment under consideration. The tetrahedron is bounded by four planes with each plane normal $\vec{n}_i$ perpendicular to face f_i.

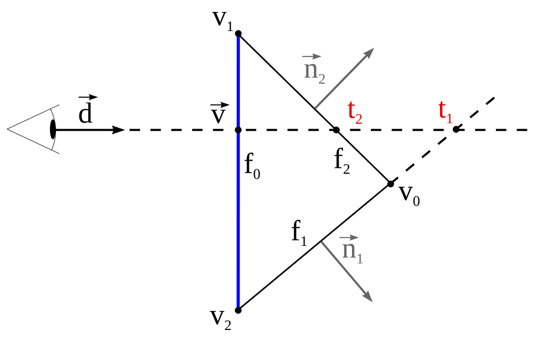

Figure 2: Illustration of per-fragment ray casting. We show the analogue situation in 2D. A viewing ray starting at the entry point $\vec{v}$, which corresponds to the fragment of the (blue) front face, is intersected with all faces of the cell.

We adapt the well-known idea of parametrical line clipping against convex polyhedra as proposed amongst others by Cyrus and Beck [3]: Test the line with every face of the polyhedron and compute line parameters for the intersection points. Using a classification of each intersection as potentially entering or potentially leaving, the exit point we are looking for is given by the leaving intersection with the smallest parameter value.

As the considered ray starts on a face, we only have to take into account the three remaining faces of the tetrahedron. We assume a parametric definition of the viewing ray:

$$\vec{r} = \vec{v} + t\vec{d} \; , \tag{2}$$

where in the case of perspective projection $\vec{d}$ is the normalized vector from the view point to the entry point corresponding to the fragment. For orthographic projection, $\vec{d}$ is the normalized view vector, which can be extracted from the camera parameters. With a_i denoting the constant term in the plane equation of face f_i, the ray parameter t_i for the intersection with face f_i can be computed as

$$t_i = \frac{(\vec{v} \cdot \vec{n}_i) - a_i}{(\vec{d} \cdot \vec{n}_i)} \,. \qquad (3)$$

The classification of intersections is slightly simpler than in the general line clipping algorithm. The reason is indicated in Figure 3: According to the direction of $\vec{n}_1$, face f_1 must be classified as potentially entering. However, as the ray starts at the correct entry point, which is the maximum of all potential entry points, the ray parameters for additional entering intersections are necessarily negative. All positive ray parameters $t_i > 0$ are potentially leaving and the minimum of all positive parameters corresponds to the actual exit point. Furthermore, as the face normals and the view direction are considered to be normalized, the minimum positive t_i is also the thickness l of the cell along the viewing ray, which is required for ray integration.

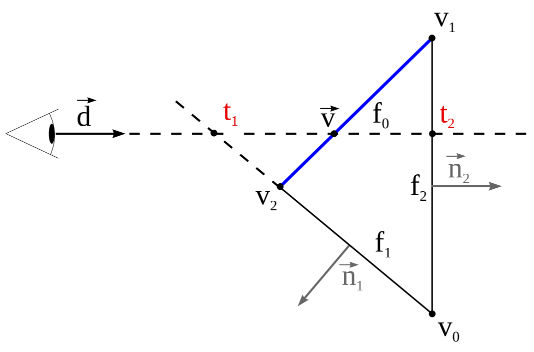

Figure 3: Potentially entering intersections can be eliminated by considering the sign of the corresponding ray parameter. Entering intersections besides the actual entry point correspond to negative ray parameters.

In the next section, we present our current implementation, which employs a combination of vertex programs and fragment operations. In Section 8, we will discuss improvements of the implementation for future hardware.

4 Implementation

Although the per-fragment computation presented in Section 3.2 is not very complex, it cannot be performed on a per-fragment basis on current graphics hardware. For example, nVidia's fragment pipeline of the GeForce3 chip series, consisting of texture shaders and register combiners [7], requires all texture lookups to take place before any arithmetical fragment operation. Our approach, however, requires a dependent texture lookup after several per-fragment computations. The ATI fragment shader extension allows to interleave arithmetical operations with texture lookups, but the currently available ATI Radeon R200 chip supports only eight such operations before the last texture lookup, which is insufficient for our purposes. However, by limiting us to orthographic projection, we can implement the same idea using mainly per-vertex operations, which can be performed view-independently by the graphics hardware using vertex programs (see [7]).

For othographic projections, the view direction is constant for every fragment; thus, the intersection parameters t_i vary linearly for all fragments of a certain front face. We can exploit this fact by computing the parameters only for the vertices of the front face and rely on the graphics hardware to interpolate the t_i parameters correctly per fragment.

4.1 Per-Fragment Computations

As the ray parameters are provided as linearly interpolated vertex attributes, the only remaining per-fragment operation is the selection of the positive minimum of the three ray parameters. We use texture coordinates to store the three t_i at each vertex since they must not be clamped to $[0,1]$, which is the case for color components. There are two methods for computing the minimum: A sequence of "conditional set" instructions can be used if included in the set of available fragment operations, as it is the case for the ATI Radeon R200 graphics chip.

In our current implementation for the nVidia Geforce3 graphics chip, however, we use a 3D texture map to determine the minimum of the three texture coordinates (r,s,t) with $0 \leq r,s,t \leq 1$. Unfortunately, when computing the t_i according to Equation 3 values greater than 1 are likely to occur. We avoid this by "normalizing" each tetrahedron such that the greatest possible thickness is 1.

This can be achieved by storing the maximum edge length of the tetrahedron in the homogeneous texture coordinate q. We use the homogeneous coordinate instead of dividing the texture coordinates by the maximum edge length, since the former allows a more accurate interpolation during rasterization, as the division by the homogeneous coordinate q is performed on the interpolated texture coordinates.

Although t_i may still be greater than 1 after the normalization, the correct minimum is guaranteed to be less than 1; thus, values greater than 1 may simply be clamped for the lookup using the OpenGL `CLAMP_TO_EDGE` texture environment. Clamping can also be employed to prevent negative values, which correspond to potentially entering intersections, from being considered by overwriting negative ray parameters by a large positive value.

Note that accurate shading depends heavily on the resolution of the minimum texture. This is especially an issue for stretched tetrahedra since, due to the normalization, they use only parts of the texture along the short edges. However, large textures impose a large memory overhead. We consider using a resolution of 128^3, which takes 4 MB for a luminance alpha texture map, a good compromise between image quality and memory requirements.

4.2 Per-Vertex Computations

As we have stated before, our goal is the view-independent scan conversion of tetrahedra. Therefore, the required per-vertex processing is not performed by the CPU but with the help of programmable vertex transformation as provided by the nVidia vertex program extension [7]. This extension replaces the standard OpenGL transform and lighting calculations by a vertex program, which may consist of a sequence of up to 128 floating-point vector instructions. The program is called for each vertex in order to transform the provided set of vertex attributes to output attributes, which are the input to the rasterization unit. An additional advantage of vertex programs is the relatively large set of floating-point 4-component registers to operate on and a large set of constant input registers to provide additional input required by the vertex program. This functional range allows us to define view-independent mesh primitives since all operations required for the cell projection can be performed by the vertex program discussed in the remainder of this section.

The crucial parts of the vertex program are the view transformation of the vertices and the computation of the ray parameters according to Equation 3. As the formulation of both is straight forward with the available set of vertex program instructions, we do not present particular code fragments here. However, we would like to mention that the OpenGL feedback mode serves well for testing vertex programs.

As explained in Section 3.2, we have to compute the ray parameters t_i of the intersection point of the ray with three faces of the tetrahedron. Conceptually, this would require the plane equations of all three faces as parameters for each vertex, since vertex programs cannot share any information between vertices. However, this would increase the number of vertex attributes significantly.

Fortunately, the number of parameters can be reduced by the following argument. Consider the intersection of a ray with a tetrahedron illustrated in Figure 4.

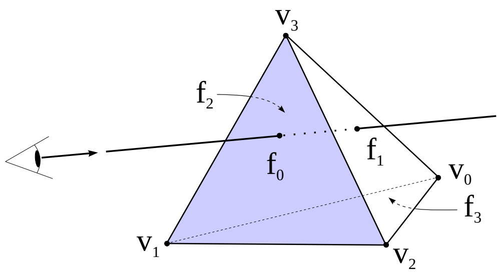

Figure 4: Nomenclature for tetrahedra.

Let the highlighted face f_0 of the tetrahedron opposite to v_0 be the front face to be rendered. In this case the potential exit faces are f_1, f_2, and f_3. Thus, the ray parameters t_1, t_2, and t_3 must be interpolated for every fragment of f_0 and therefore computed at the vertices v_1, v_2, and v_3. Note that for v_1 we already know the correct t_2 without any computation, since v_1 is part of face f_2; thus, t_2 must be 0. The same applies to t_3, which is also 0, as v_1 is part of f_3. In fact, we have to evaluate Equation 3 only for t_1; thus, we only require the plane equation of f_1 as vertex parameter for v_1. In general, we need the plane equation for f_i as vertex parameter for v_i.

Defining a plane equation as vertex parameter usually requires four float values, a normal vector with three components and a plane offset as additional parameter. Reducing this number of parameters is a worthy goal since every additional value that has to be transferred to the graphics adapter decreases the overall performance or reduces the maximum number of cells of an unstructured mesh buffered on the graphics chip. Therefore, it is worth mentioning that the same information can be provided using only three float values. As the plane normal has to be a unit vector, we can deduce the third component from the first and second using:

$$\vec{n} = (n_0, n_1, n_2) = (n_0, n_1, \sqrt{1 - n_0^2 - n_1^2}), \qquad \text{for } n_2 > 0. \tag{4}$$

Thus, we may compute the third component with the help of only a few vertex program instructions. However, Equation 4 only holds for positive n_2. Fortunately, $\vec{n}$ and $-\vec{n}$ leads to the same result of Equation 3; thus, we can handle this problem by simply negating the normal vector if the third component is less than 0.

Note that at each vertex only one ray parameter t_i is actually computed by the vertex program. However, all vertices of a face must agree on the ordering in which the three required ray parameters t_i are stored in the texture coordinates r, s, and t, such that they are consistently interpolated during rasterization. We achieve this by explicitly providing the index of the texture coordinate that should be used for storing t_i. An index of 0 denotes r, while s and t are denoted by 1 and 2, respectively. Using vertex program instructions we have found two different ways of storing a scalar in a certain component of an output vector. The corresponding code fragments are given in Figure 5 and Figure 6. See [7] for a detailed definition of vertex program syntax and semantics.

```
# c[8]  = {1, 2, 3, 0}
# c[9]  = {0, 1, 2, 0}
# R3.x  = computed ray parameter
SLT R1, v[TEX0].z, c[8];
SGE R2, v[TEX0].z, c[9];
MUL R0, R1, R2;

MUL R4, R3.x, R0;
ADD o[TEX0], R4;
```

Figure 5: Mapping the computed ray parameter t_i to a specific component of the texture coordinates with conditional set instructions “set on less than” (SLT) and “set on greater equal” (SGE). The component is specified as an index i ($0 \le i \le 2$) by the t-texture coordinate of the vertex input attributes (`v[TEX0].z`). Multiplying the output of a STL and a SGE operation with the defined constants results in a vector with 1 at position i and 0 elsewhere. We multiply this vector with the ray parameter and add it to the texture coordinates.

```
# c[18] = {1, 0, 0, 0}
# c[19] = {0, 1, 0, 0}
# c[20] = {0, 0, 1, 0}
# R1.x  = computed ray parameter

ARL A0.x, v[TEX0].z;
MUL R0, R1.x, c[A0.x + 18];
ADD o[TEX0], R0;
```

Figure 6: Mapping the computed ray parameter to a specific component of the texture coordinates using the “address register load” (ARL) instruction. The component is specified by the t-texture coordinate of the vertex input attributes (`v[TEX0].z`). The ARL instruction maps the index i to one of the vectors `c[18]`, `c[19]`, and `c[20]` provided as constant program attributes. We multiply this vector with the ray parameter and add it to the texture coordinates.

4.3 Edge Artifacts

Using the ray parameters computed at the vertex positions for texture coordinates as described in the previous section leads to artifacts at edges of the tetrahedron as can be observed on the left-hand side of Figure 7. Note that the thickness of the cell is mapped to intensity in the figure. The inset shows a dark seam along the edge representing a thickness of zero, which is obviously wrong.

The reason for these artifacts lies in the texture coordinates shown on the right-hand side of Figure 7. When rendering face f_1 the r coordinate represents the distance to face f_0. It is obvious that this distance is negative. In other words, the intersection of the viewing ray with face f_0 is a potentially entering intersection point and should not be considered for the exit point search. For most fragments on face f_1 this is the case, since the handling of negative ray parameters described in Section 4.2 will guarantee that t_0 does not influence the minimum search.

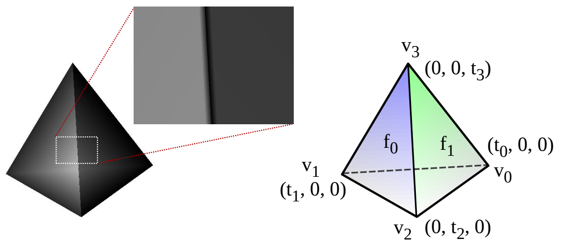

Figure 7: Identifying the minimum of the interpolated ray parameters with the thickness of the cell leads to artifacts along edges between visible faces. They result from assuming a zero thickness along the edge. In the presented visualization the thickness is mapped to intensity, thus the artifacts show up as clearly visible dark lines.

However, the interpolation weight for v_0 is zero along the edge from v_2 to v_3; therefore, we lose the classification stored in t_0. Thus, the intersection of the viewing ray with face f_0 is misinterpreted as entering point and the texture lookup for the minimal texture coordinate erroneously returns zero. The problem could be avoided, if a minimal interpolation weight could be specified. In this case, we would use a small offset for the interpolation and compensate this by biasing the texture coordinates before the lookup or by using a small bias in our minimum texture map.

Note that erroneously classified fragments are an additional reason, why our use of vertex programs is restricted to orthographic projections. We like to point out once more that these artifacts are solely introduced by the workaround of using per-vertex computations instead of per-fragment operations. With per-fragment operations, the classification can always be correctly evaluated resulting in the correct cell thickness.

However, we can also avoid the defects in a per-vertex manner. We only have to guarantee that at least one vertex on the edge has the correct classification of the second face sharing the edge. During rasterization this classification affects the whole edge. We utilize the counterclockwise definition of the faces to define a direction for every edge per face. Doing so, every vertex of the face can be considered as the start of one of the three edges. We supply the normal of the adjacent face as vertex attribute to the start vertex of each edge. Similarly to the handling of negative ray parameters described in Section 4.2, the vertex program checks whether the adjacent face is potentially entering and sets the corresponding texture coordinate to a large positive value instead of 0.

Considering face f_1 in the example depicted in Figure 7, v_3 is the start vertex of the edge from v_3 to v_2 and would set the r texture coordinate to some large positive value. The r texture coordinate of v_2 will not be changed for rendering face f_1. In face f_0 vertex v_2 is the start vertex of the edge from v_2 to v_3 and will modify its r component since face f_1, which corresponds to the ray parameter t_1 at v_1, is potentially entering for all fragments of face f_0.

Using this scheme, one endpoint of the edge is still being left "unclassified". However, in order to compute the full classification, each vertex of the face would need the normals of all three remaining faces, which would require even more per-vertex parameters. We deal image quality for transfer rate, since we have found this problem to be negligible. The remaining classification mistakes occur at singular positions, thus at most one pixel per tetrahedron is incorrectly colored and in most cases no pixel is influenced at all.

5 Optical Models and Ray Integration

Our approach of view-independent cell projection conceptually computes one ray segment of a viewing ray through a tetrahedron for each pixel. In order to determine the color and opacity contribution, we apply pre-integrated classification for cell projection [14, 4]. We use the scalar value on the front face s_f, the scalar value on the back face s_b, and the thickness l resulting from scan converting the tetrahedron as texture coordinates for a 3D texture map storing the values of the volume ray integral in dependency of l, s_f, and s_b.

The lookup in this texture map has to be performed based on the result of the lookup in the minimum texture described in Section 4.1. The setup for the 3D dependent texture lookup using the nVidia texture shader extension [7] is depicted in Figure 8. If the optical model only requires a 2D texture lookup a similar setup can be constructed by simply replacing stage 2 by a `DOT_PRODUCT_TEXTURE_2D_NV` operation.

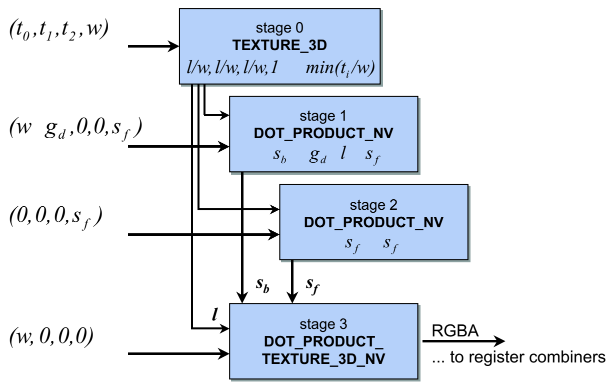

Figure 8: Texture shader setup for a 3D dependent lookup using the minimum ray parameter from the minimum texture.

The lookup in the minimum texture is performed with the computed ray parameters, which are "normalized" using the fourth texture coordinate. The scalar product in the second stage reconstructs the scalar value on the back face with the help of the gradient in view direction g_d and the scalar on the front face s_f. Note that we employ a luminance alpha texture map as minimum texture with the alpha channel set to 1 in order to allow for the addition of s_f. The third stage only supplies s_f as a texture coordinate and the fourth stage employs the "denormalized" ray parameter generated by multiplying the scaled length l/w with the scaling factor w together with s_f and s_b to look up the color and opacity according to each ray segment. We provide suitable texture coordinates for the second to fourth texture shader stage via the vertex program in order to combine all necessary parameters for the final lookup in stage 3.

5.1 Direct Volume Rendering

With the color and opacity contribution stored in a texture map, different shading techniques can be implemented. However, for unsorted cell projection only those optical models are possible, which lead to a commutative blend function, i.e. allow the compositing in arbitrary order. Among these models are maximum intensity projection and restrictions of the full volume density model to either the source term or the absorption term. For all these models, the texture lookup allows us to apply arbitrary transfer functions, which only affect the generation of the texture map.

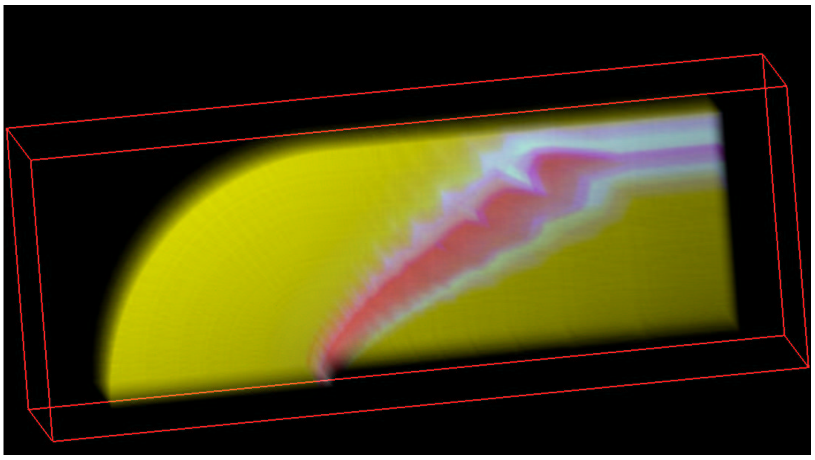

Figure 9: *The image shows a rendering of the bluntfin dataset decomposed into 225K tetrahedra. We demonstrate our view-independent cell projection method using emissive projection with an appropriate transfer function applied.*

For maximum intensity projection, for example, a 2D texture map contains the maximum of the transfer function in the interval $[min(s_f, s_b), max(s_f, s_b)]$ at position (s_f, s_b). Moreover, we have to use a black background and the maximum blending function of the OpenGL blend minmax extension.

A pure emissive rendering of the unstructured mesh, as demonstrated in Figure 9, simply adds the contribution of each tetrahedron. In this case a 2D texture map is sufficient, since the dependency of the color contribution on the length of the ray segment is linear and can be integrated by modulating the result of the shading lookup with the result of the minimum texture lookup. We use the register combiners extension [7] to multiply the result of texture stage 2 with the r coordinate of texture stage 0. The texture map employed in stage 2 contains the integral of the transfer function for the interval $[min(s_f, s_b), max(s_f, s_b)]$ at position (s_f, s_b).

In contrast to the previous mentioned models, a restriction to the absorption term of the volume density model requires a 3D texture map containing the integral of the absorption coefficient along the viewing ray segment.

All these optical models may be extended with intensity depthcuing [6] as demonstrated in Figure 11. Combining our approach with any cell sorting algorithm (see Section 6), we can as well simulate the original PT algorithm or integrate any published improvement, e.g. the appoaches of Stein et al. [16] or more recently of Röttger et al. [14] based on texture maps.

5.2 Isosurfaces

We are also able to render multiple flat shaded opaque isosurfaces without visibility ordering of the cells. A texture map as depicted in Figure 10 (see also [14]) is used to extract isosurface fragments from the rendered faces. The correct occlusion of isosurface fragments can be guaranteed even without sorting the cells of the unstructured mesh by a combination of the alpha test with the OpenGL z-test.

The basic idea is quite obvious. Whenever the viewing ray within a tetrahedron hits the isosurface, the inequalities $s_f < s_{\text{iso}}$ and $s_b > s_{\text{iso}}$ or vice versa hold. In these cases the texture map lookup results in the ambient material color of the isosurface. The texture map sets the alpha channel to 0 whenever no isosurface is present; thus, we can use the alpha test to render only fragments of the isosurface. As mentioned in [14], we have to slightly modify the texture map, effectively "thickening" the isosurfaces, in order to avoid gaps between isosurface patches in adjacent tetrahedra.

Figure 10b shows an example for a texture map for multiple isosurfaces. The "visibility ordering" is easy to understand: For $s_f < s_b$ we are looking along the gradient of the scalar field; thus, isosurfaces for smaller isovalues occlude those for greater isovalues and vice versa. Rendering multiple isosurfaces we can assign different material colors for each isovalue as depicted in Figure 12.

As our approach does not provide interpolated gradients we are limited to flat shaded isosurfaces. Therefore, the complete lighting may be calculated by the vertex program. We can render a virtually arbitrary number of directional lights (see Figure 13), as each light requires two entries in the vertex program's attribute vector — light direction in object coordinates and diffuse light color — and three vertex program operations.

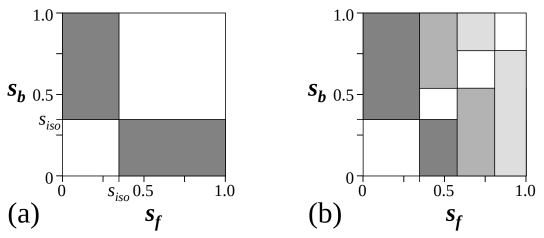

Figure 10: (a) A 2D texture map for rendering isosurfaces with view-independent cell projection. Texels corresponding to isosurface fragments are colored with the diffuse material color of the isosurface. The alpha channel is 0 in the absense of an isosurface. (b) A texture map for three opaque isosurfaces.

6 Cell Projection of Tetrahedral Meshes

Apart from the projection of single tetrahedra the rendering of tetrahedral meshes by cell projection includes more tasks, e.g. visibility sorting and the transfer of large amounts of tetrahedra, which will be discussed in this section.

Today, most of the volume renderers for unstructured meshes that are based on the projected tetrahedra algorithm cannot exploit the peek performance of modern graphics adapters [8]. The reason

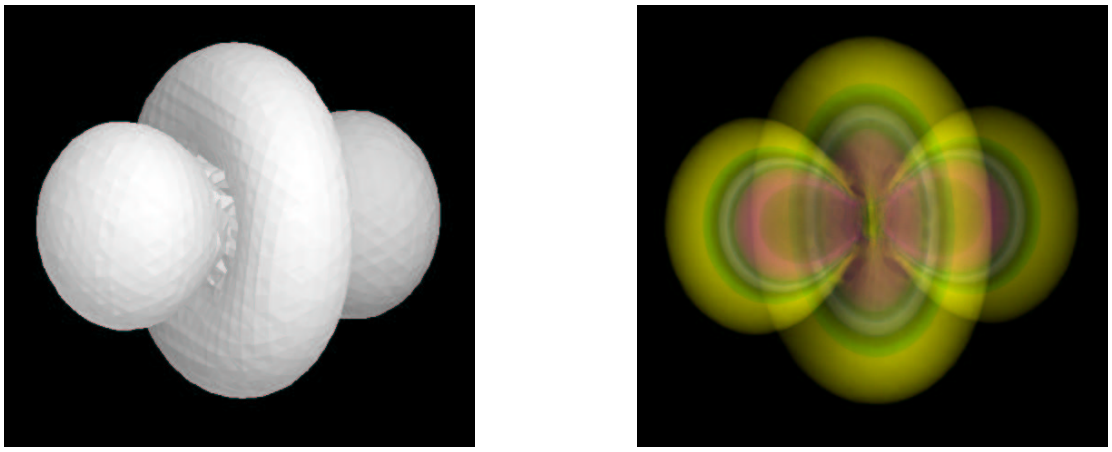

Figure 11: *In both images we show the same orbital-like artificial data set consisting of 150K tetrahedra. Different optical models are demonstrated. In the left image an isosurface is rendered, whereas for the right image a restriction of the full volume density model to the source term is used. We can apply arbitrary transfer functions to enhance features of the dataset. Intensity depth-cuing is used to enhance the perception of depth. Note the artifacts in the right image resulting from the limited frame buffer resolution of 8 bit.*

is twofold: Firstly, due to the overhead of element sorting, current algorithms are not able to supply ordered, decomposed PT triangles at a sufficient rate. A second limiting factor is the data transfer between the CPU and the graphics board, especially for PC graphics hardware. The peek performance of the AGP bus is only achieved if data is transferred in large blocks using burst transfer. Note that burst transfer has a great potential for improving the performance of volume renderers for unstructured meshes. According to [8], there is a possible speedup of factor 20. However, PT based algorithms produce a continous stream of triangles, which cannot be combined to blocks without significant overhead.

The motivation for our approach of view-independent cell projection is to overcome this bottleneck of data transfer between the CPU and the graphics board. With a view-independent description of the unstructured mesh, we are able to build a fixed set of data in a preprocessing step, which is reused in every frame. This allows many optimizations, e.g. caching or pre-compilation of the data for

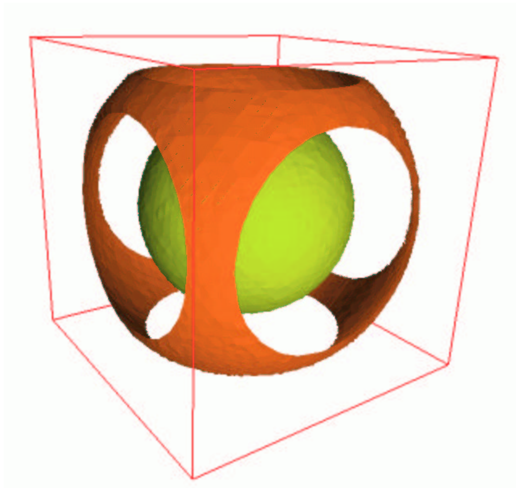

Figure 12: Individual material colors can be assigned to each isovalue when rendering multiple isosurfaces.

optimized transfer. In this context, standard OpenGL provides two mechanisms for optimization: display lists and vertex arrays [22].

The basic idea of display lists is to cache the OpenGL commands such that the same geometry can be defined once and rendered multiple times by simply executing the display list. Optimized performance can be achieved, since particular graphics hardware may store display lists in dedicated memory or may store the data in an optimized form that is more compatible with the graphics hardware or software.

Vertex arrays were designed to reduce the number of OpenGL function calls since storing all vertex related data in just a few arrays allows the programmer to specify a lot of graphical primitives with only one function call. Similarly to display lists, vertex arrays can also be cached or pre-compiled for more efficient rendering. Recent extensions, e.g. the vertex array range extension [7], might even allow to buffer vertex related data in local graphics memory or at least to store vertex data in a dedicated area of the client address space in order to enable the graphics hardware to pull the vertex data via Direct Memory Access (DMA) using burst transfer. Thus, the OpenGL client has to pass only vertex indices or the number of primitives to render.

Furthermore, vertex arrays allow for nonredundant processing of shared vertices. However, our current implementation requires individual vertex data for each face of the tetrahedra and, therefore, cannot benefit from optimized processing of shared vertices.

According to our measurements, display lists hardly improve the performance of our method on current PC graphics adapters based for example on the nVidia GeForce3 or ATI Radeon R200 chip. Therefore, our implementation employs vertex arrays. However, this choice has a few unfavorable implications. Rendering with vertex arrays allows no data per element but only per-vertex data; thus, some data replication is needed, e.g. to provide each vertex with the gradient of the scalar field that is constant per tetrahedron.

As switching to an interleaved vertex array does not significantly affect the performance of our implementation, we decided to use four arrays of vertex data: a vertex array, a color array, a normal array, and an array of texture coordinates. For each face of each tetrahedron we need individual vertex instances, i.e. instead of using a triangle strip of six vertices per tetrahedron, twelve vertices are required for the four triangular faces of each cell.

The per-vertex data necessary for our approach sums up to 15 float values. The vertex position is stored in the *x*, *y*, and *z* compo-

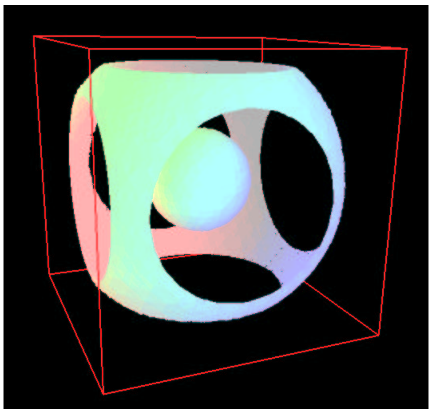
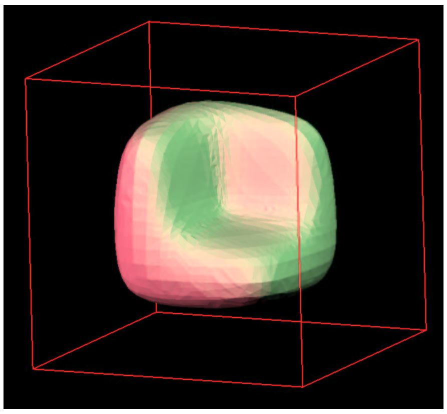

Figure 13: *Both images show flat shaded isosurfaces with multiple light sources using our view-independent cell projection. In the left image an analytical dataset of about 150K tetrahedra is rendered with two isosurfaces. Although perspective projection is used artifacts are hardly visible due to a comparatively small field-of-view. The right image shows the same method applied to a finite-element mesh (heat-sink) consisting of 120K tetrahedra using orthographic projection.*

nents of the vertex array, while the *w* component holds the constant factor of the plane equation for the opposite face, which is required for the computation of the ray parameter. Note that we can abuse this component for our purpose, since the vertex program can overwrite the homogeneous coordinate of the vertex with 1 before performing the view transformation. Two components of the opposite face normal are stored in *r* and *s* of the texture coordinates. The reconstruction of the third component of the normal vector is described in Section 4.2. The *t* texture coordinate holds the index of the component, the vertex will store the ray parameter in; *q* stores the maximal edge length of each element required for "normalizing" texture coordinates (see Section 4.1).

We use the four color components to specify the tetrahedon's gradient and the scalar value of the vertex. We do not have to take care of clamping, since all vertex data is provided as signed float values to the vertex program. Finally, each vertex is provided with the normal of one adjacent face in order to avoid edge artifacts (see Section 4.3).

We render a complete unstructured mesh by a single call of `glDrawArrays`. Thus, assuming that vertex data may reside in local graphics memory, our approach ideally limits the transfer between CPU and the graphics adapter to a few bytes per frame and the graphics chip can operate at full capacity without waiting for data to be delivered.

However, as all elements in the vertex arrays are defined in a fixed linear order, no visibility sorting can be applied, which limits our choice of optical models. Our view-independent projection method, though, also allows us to render in visibility order, using the same vertex array data. Instead of a single `glDrawArray` call, one call per tetrahedron may address the elements in sorted order by just transfering two indices per cell to the graphics hardware. Note that the sorting algorithm needs to provide only the sorted list of indices. In this sense, our approach is compatible with any published cell sorting algorithm.

7 Results

We have tested our implementation on two systems: The first was a Linux PC with an Athlon 1200 MHz processor and 512 MB RAM. However, the influence of the CPU is negligible since our approach moves most of the computation to the graphics processor. As graphics adapter a GeForce3 with 64 MB of local memory was used.

Table 1 shows the performance of the GeForce3 for different datasets presented in the paper. Sphere and heat-sink denote the datasets of Figure 13, whereas the orbital dataset is shown in Figure 11. Note that the performance is not affected by a particular optical model or transfer function, since different optical models or transfer functions are implemented by modifying the content of a 2D texture map only. Even the more expensive 3D texture lookup required for the absorption only optical model performs comparably which indicates that our implementation is not rasterization limited. We present the results for rendering without visibility sorting.

	no. cells	fps	tets/sec	MB/sec
Bluntfin	224 874	1.08	242 K	174
Heat-sink	121 668	2.01	244 K	175
Orbital	148 955	1.65	245 K	177
Sphere	148 955	1.65	245 K	176

Table 1: *Performance of our view-independent cell projection for different datasets using a GeForce3 graphics adapter.*

The Geforce3 chip performed better than the also tested GeForce3 Ti200 chip with the same amount of local memory. As expected, the achieved frame rates are linear in the number of cells. Table 1 shows that the number of tetrahedra processed per second is almost constant.

However, the average data transfer rate of about 175 MByte/sec, which can be computed with the product of vertices per element (12), number of vertex attributes (15), and the size of a float value, is significantly below the theoretical peek performance of 1000 MByte/sec of a 4×AGP connection.

This has two reasons: First, at least on the GeForce3, using our vertex programs obviously imposes a significant overhead compared to standard OpenGL transform and lighting. We were not able to transform more than 6.4 million vertices with our quite moderate vertex programs of about 30-50 instructions, depending on the optical model. Using standard OpenGL transform and lighting our test system achieved rates of up to 18 million vertices. Note, that a very simple vertex program, which only consists of the view-transformation, performs comparably.

However, the main bottleneck is not the vertex transformation unit or the rasterization but the data transfer. For datasets of relevant size we could not exploit the vertex array range extension, which we have found out to be 2-3 times faster than every other approach. This performance is due to the fact that vertex range arrays allow the vertex data to be cached in the local memory of the graphics adapter or in a memory area which can be accessed directly by the graphics adapter using DMA pull. However, vertex range arrays are currently limited to vertex data of less than 32 MB, even on graphics adapters with 128 MB of local memory, which would have restricted us to roughly 90K tetrahedra. Relying on ordinary vertex arrays the bottleneck of our implementation is the bandwidth for the main memory access. This is demonstrated by Table 2.

	no. cells	fps	tets/sec	MB/sec
Bluntfin	224 874	1.67	375 K	167
Heat-sink	121 668	3.02	367 K	163
Orbital	148 955	2.53	377 K	168
Sphere	148 955	2.53	377 K	168

Table 2: *Performance of our view-independent cell projection on a GeForce3 for different datasets using mainly short values instead of floats as vertex data.*

Here we have used short values, instead of floats for the required per-vertex data wherever possible. Thereby we were able to reduce the memory bandwidth requirements from 720 bytes per tetrahedron to 456 bytes per-tetrahedron. It has turned out that the effect of the more inaccurate data representation on the image quality is negligible. The table shows almost the same data transfer rates as Table 1 but the number of tetrahedra per second was increased correspondingly to the reduction of data.

Our second test system was a Linux PC with a GeForce4 Ti with 128 MB of local memory, an Athlon XP 1800+, and 512 MB RAM. On this graphics chip, we did not recognize a performance penalty for vertex program rendering, due to the second vertex pipline. Table 3 shows the performance of the system for the same datasets with a reduced amount of per-vertex data. The framerates are about 50% higher than on the GeForce3 system. However, as our implementation is memory bandwidth limited, the speedup mainly results from the faster memory access of the second test system rather than the faster graphics chip.

	no. cells	fps	tets/sec	MB/sec
Bluntfin	224 874	2.13	479 K	213
Heat-sink	121 668	3.81	464 K	207
Orbital	148 955	3.23	481 K	214
Sphere	148 955	3.23	481 K	214

Table 3: *Performance of our approach for different datasets using nVidia's GeForce4 chip.*

8 Future Work

The next generation of programmable graphics chips will probably implement DirectX 9 and in particular version 2.0 of the PixelShader language. According to Marshall in [11], this version will allow up to 32 texture lookups and up to 64 arithmetic operations, including the computation of reciprocals. Thus, we will be able to implement our algorithm with per-fragment operations only. This will not only remove the costly vertex program but also the restriction to orthographic projections. Furthermore, we will be able to employ texture lookups to access any mesh data, e.g. gradients of the scalar field or face normals, and thereby dramatically reduce the number of required vertex parameters. Moreover, this will allow us to reuse vertices of the mesh for multiple triangles; therefore, we will be able to project a tetrahedron by rendering a triangle strip of four triangles, i.e. with 6 vertices instead of 12 vertices in the current implementation. The reduced data will alleviate the memory bottleneck leading to a significant speedup.

A more flexible fragment pipeline will also allow us to implement several interesting extensions to our algorithm, e.g. clip planes for tetrahedra.

9 Conclusions

We have presented the first view-independent cell projection algorithm suitable for commercial off-the-shelf graphics hardware. Unfortunately, it turned out that today's graphics hardware is not flexible enough for an optimal implementation. Nonetheless, the view-independency allows us to exploit important optimization techniques, especially OpenGL vertex arrays. Therefore, our cell projection algorithm is likely to perform significantly better than the traditional Shirley-Tuchman projection on future graphics hardware.

Moreover, our algorithm is suitable for architectures supporting order-independent transparency as proposed by Wittenbrink [21] and it may be employed as part of a hardware implementation of the ray casting algorithm in unstructured meshes [10].

References

[1] P. Cignoni, C. Montani, D. Sarti, and R. Scopigno. On the optimization of projective volume rendering. In R. Scanteni, J. van Wijk, and P. Zanarini, editors, *Visualization in Scientific Computing '95*, pages 58–71. Springer-Verlag Wien, 1995.

[2] João Comba, James T. Klosowski, Nelson Max, Joseph S. B. Mitchell, Claudio T. Silva, and Peter L. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum (Proceedings of Eurographics '99)*, 18(3):369–376, 1999.

[3] M. Cyrus and J. Beck. Generalized two- and three-dimensional clipping. In *Computers and Graphics 3(1)*, pages 23–28, 1978.

[4] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '01*, Annual Conference Series, pages 9–16. Addison-Wesley Publishing Company, Inc., 2001.

[5] Ricardo Farias, Joseph S. B. Mitchell, and Claudio T. Silva. Zsweep: An efficient and exact projection algorithm for unstructured volume rendering. In Roger Crawfis and Danny Cohen-Or, editors, *Proceedings Volume Visualization and Graphics Symposium 2000*, pages 91–99. ACM Press, 2000.

[6] Foley, Van Dam, Feiner und Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 2 edition, 1990.

[7] Mark J. Kilgard, editor. *NVIDIA OpenGL Extension Specifications*. NVIDIA Corporation, 2001.

[8] Davis King, Craig M Wittenbrink, and Hans J. Wolters. An architecture for interactive tetrahedral volume rendering. In Klaus Mueller and Arie Kaufman, editors, *Volume Graphics 2001, Proceedings of the International Workshop on Volume Graphics 2001*, pages 163–180. Springer-Verlag, 2001.

[9] Martin Kraus and Thomas Ertl. Cell-projection of cyclic meshes. In Thomas Ertl, Kenneth Joy, and Amitabh Varshney, editors, *Proceedings IEEE Visualization 2001*, pages 215–222. ACM Press, 2001.

[10] Martin Kraus and Thomas Ertl. Implementing ray casting in tetrahedral meshes with programmable graphics hardware. Technical Report 1, Visualization and Interactive Systems Group at the University of Stuttgart, 2002.

[11] Brian Marshall. Directx graphics future, 2001. Presentation at the Microsoft DirectX Meltdown 2001, available at http://www.microsoft.com/mscorp/corpevents/meltdown2001/presentations.asp.

[12] N. Max. Optical models for direct volume rendering. In *IEEE Transactions on Visualization and Computer Graphics 1(2)*, pages 99–108, 1995.

[13] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. *ACM Computer Graphics (Proceedings of San Diego Workshop on Volume Visualization 1990)*, 24(5):27–33, 1990.

[14] Stefan Röttger, Martin Kraus, and Thomas Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In Thomas Ertl, Bernd Hamann, and Amitabh Varshney, editors, *Proceedings IEEE Visualization 2000*, pages 109–116. ACM Press, 2000.

[15] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. *ACM Computer Graphics (Proceedings of San Diego Workshop on Volume Visualization 1990)*, 24(5):63–70, 1990.

[16] Clifford M. Stein, Barry G. Becker, and Nelson L. Max. Sorting and hardware assisted rendering for volume visualization. In Arie Kaufman and Wolfgang Krueger, editors, *Proceedings 1994 Symposium on Volume Visualization*, pages 83–89. ACM Press, 1994.

[17] P. L. Williams and N. Max. A volume density optical model. In *ACM Computer Graphics (1992 Workshop on Volume Visualization)*, pages 61–68, 1992.

[18] P. L. Williams, N. L. Max, and C. M. Stein. A high accuracy volume renderer for unstructured data. In *IEEE Transactions on Visualization and Computer Graphics 4(1)*, pages 37–54, 1998.

[19] Peter L. Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, 1992.

[20] Craig M. Wittenbrink. Cellfast: Interactive unstructured volume rendering. In Craig M Wittenbrink, Amitabh Varshney, and Hans Hagen, editors, *IEEE Visualization 1999 Late Breaking Hot Topics*, pages 21–24, 1999.

[21] Craig M. Wittenbrink. R-buffer: A pointerless a-buffer hardware architecture. In *Proceedings Graphics Hardware 2001*, pages 73–80. ACM Press, 2001.

[22] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley, third edition, 1999.