

Texture-Encoded Tetrahedral Strips

Manfred Weiler*
University of Stuttgart

Paula N. Mallón†
University of Santiago de Compostela

Martin Kraus‡
Purdue University

Thomas Ertl*
University of Stuttgart

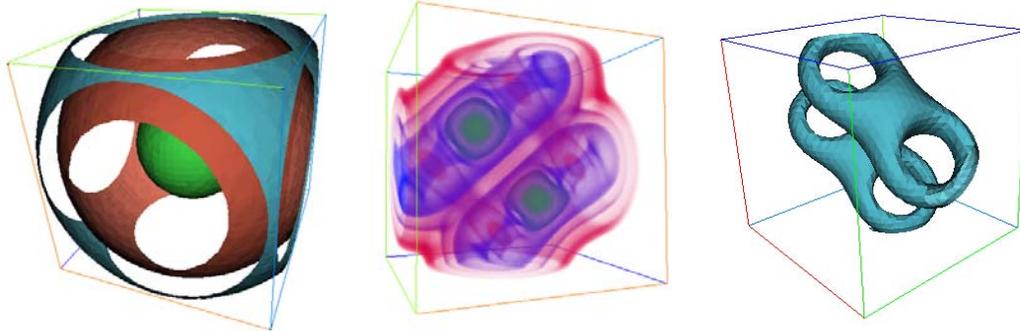


Figure 1: Volume visualizations of tetrahedral meshes which have been encoded in a compact texture representation based on tetrahedral strips and are stored in the texture memory of the graphics adapter. The renderings have been computed with a ray casting algorithm for programmable graphics hardware adapted for this mesh representation.

ABSTRACT

The use of triangle strips is a common method to compactly store and efficiently render large polygonal meshes. The advantages of triangle stripification also apply to tetrahedral meshes; therefore, tetrahedral strips are an attractive data structure for storing and volume rendering tetrahedral meshes as noted in several publications. However, tetrahedral strips are still not supported by current graphics hardware.

In this paper, we present the first system to take advantage of tetrahedral strips in off-the-shelf graphics hardware. This is achieved by encoding tetrahedral strips in texture maps and rendering them with the help of a ray casting algorithm running solely on the graphics chip. Our data structure supports sequential and generalized tetrahedral strips by including a small amount of adjacency information, which allows us to access all face neighbors in constant time.

Utilizing these texture-encoded tetrahedral strips, our enhanced graphics-hardware-based volume ray casting algorithm for tetrahedral meshes is capable of handling large data sets. Additional improvements presented in this paper include support for multiple ray traversal steps in one rendering pass and the intrinsic support for non-convex meshes using a rendering technique similar to depth peeling.

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, Texture, Raytracing

*e-mail: {weiler|ertl}@vis.uni-stuttgart.de

†e-mail: paulanm@dec.usc.es

‡e-mail: kraus@purdue.edu

Keywords: tetrahedral strips, ray casting, pixel shading, programmable graphics hardware, cell projection, tetrahedral meshes, unstructured meshes, volume visualization, pre-integrated volume rendering

1 INTRODUCTION

Polygonal strips and in particular triangle strips are one of the most popular representations for surface objects. The main reason is their compactness, which is achieved by encoding t triangles with $t + 2$ vertex indices instead of the $3t$ vertex indices required by a triangle list. Due to the rapidly growing demand for more complex objects in interactive graphics applications, these low storage requirements are of increasing importance.

Graphics hardware can benefit from supporting polygonal strips in several ways: The data transfer bottleneck between the CPU and the GPU, which has turned out to be one of the most limiting factors of the current hardware architecture, is widened. The capacity for storing objects as vertex arrays in the local memory of the graphics card is increased. Last but not least, repeated processing of vertices can be reduced since the repetitious pattern of strips permits effective caching strategies. Therefore, a wide support for polygonal strips is provided by graphics adapters and graphics APIs.

The advantages of triangle strips generalize directly to the volumetric case. Therefore, tetrahedral strips are an attractive representation of unstructured meshes; in particular for graphics hardware with support for scan conversion of tetrahedral primitives. According to [9] this kind of hardware architecture could dramatically accelerate unstructured volume rendering. Unfortunately, it has not been built yet; thus, it was not possible to efficiently use tetrahedral strips for hardware-supported unstructured volume rendering so far. However, modern graphics adapters provide a flexible programmable graphics pipeline offering the possibility to extend the functionality of the graphics card beyond the standard pipeline and allowing for advanced shading and visualization algorithms. In this work we introduce the first system to exploit tetrahedral strips based on programmable graphics hardware.

The key idea is to extend the classical surface strips by a small amount of neighbor information which is not only required for rendering the tetrahedra in correct visibility order but also allows for various pre- and post-processing steps requiring the connectivity between tetrahedra. In other words, our data structure is universal and can store sequential as well as general strips.

By encoding tetrahedral strips in textures and transferring them to the local memory of the graphics adapter, we can perform hardware-based ray casting directly with the texture-encoded strips. Our solution is similar to [16], but overcomes their huge memory overhead with the compact strip representation allowing for unstructured data sets of significantly larger sizes.

Before presenting our strip data structure in Section 3, we discuss previous work in Section 2. Section 4 describes the generation of tetrahedral strips, which is crucial for a compact representation of a tetrahedral mesh in texture memory. Implementation issues of our ray casting approach for the tetrahedral strips are discussed in Section 5, while results are presented in Section 6.

2 RELATED WORK

Representations based on triangular or polygonal strips are well known and, therefore, widely supported by graphics hardware, graphics APIs like OpenGL and DirectX, and graphics libraries. Since the effectiveness of this data structure for encoding a large polygonal mesh heavily depends on the stripification, research has been focused on optimal algorithms for this purpose. An optimal stripification for the rendering covers the given mesh with as few strips and as few vertex replications as possible [4].

In general, the triangle strip encoding—posed as a problem of converting a given triangle mesh into the minimal set of triangle strips covering the mesh—is NP-complete. Thus, in the literature several proposals exist that rely on heuristics to find sub-optimal solutions.

One core idea is to build the sequence of triangles by extending an existing strip always choosing the adjacent triangle with the least number of neighbors as next triangle. This should minimize the number of strips with length one. It has first been presented in [1] and is now commonly known as the SGI greedy heuristic to mesh. The STRIPE [4] system supports arbitrary polygonal meshes applying a global and a local approach. The global approach (“patchification”), tries to find large rectangular regions consisting only of quadrilaterals, which are triangulated sequentially along each row or column. For the remaining polygons the strips are created on the fly during the triangulation applying several heuristics. The produced triangle strips do not take into account hardware optimizations.

One of the most efficient stripification systems particularly optimized for performance is the Fast Triangle Strip Generator (FTSG) [17], which creates triangle strips based on the construction of a spanning tree in the dual graph of the triangulation. It achieves an encoding quality comparable with or better than that of STRIPE. The algorithm presented in [13] is designed for triangulated irregular networks (TIN). It works by creating a spanning tree of the dual graph, and then traversing the tree in a modified depth-first fashion.

Many of these algorithms are nowadays available as tools, e.g., [1] or the nVidia NVTriStrip tools [2], which can generate strips from arbitrary geometry, stitch together strips using degenerate triangles, and optimize them with respect to vertex buffering and post T&L vertex caches.

Moreover, the need for more compact representations has motivated research on mesh compression. Some compression algorithms [15, 8, 11] work by encoding the processing sequence of triangles, using an additional command string. Roughly speaking this can be considered as a general triangle strip, where the con-

nection of the next vertex with the previously extracted triangles is not fixed. All those methods lack arbitrary access to individual triangles. Some of these methods have been extended to be used in tetrahedral meshes [7, 14], however they have not been proposed for directly rendering the data sets. In fact, it is always necessary to reconstruct the complete 3D object previous to rendering.

One of the few publications about the problem of generating tetrahedral strips and tetrahedral fans is [9]. King et al. propose a hardware architecture for tetrahedral meshes encoded as strings and fans, which are rendered with the projected tetrahedra algorithm [12]. Ray casting for off-the-shelf programmable graphics hardware has been introduced recently [16] based on the ray propagation approach published in [6]. This approach suffers from a significant memory overhead since excessive data replication is required to meet the limited fragment shader capabilities of the ATI 9700 graphics card, which was targeted by their implementation.

3 TETRAHEDRAL STRIPS

In order to compactly store the data of a tetrahedral mesh in a way that allows for direct rendering by the GPU, we need to extend the classical tetrahedral strips, which consist only of the indices of vertices, with information about the connectivity between the tetrahedra. The overhead introduced by this data should be minimized in order to maintain the advantages of tetrahedral strips. Moreover, access to the four neighbors of each tetrahedron should be provided in constant time for efficient processing and rendering. In this section, we present our scheme for storing the neighbor information for each tetrahedron in the strips. The presentation is based on the usual convention that the i -th face is opposite to vertex v_i ($i \in \{0, 1, 2, 3\}$).

We introduce the structure with the example in Figure 2(a), which shows three adjacent tetrahedral strips defined by eleven vertices. The tables in Figure 2(b) next to the image of the strip show the organization of the strips as different lists, containing the indices of the vertices and the neighbors related to them. Specifically, four different lists (one per face) are required for the neighbor information. They are denoted by N_0 , N_1 , N_2 , and N_3 respectively. We place the neighbor information for each tetrahedron below the first vertex index in the strip; thus, for tetrahedron $\{4,3,8,6\}$, which is marked with bold lines, the neighbors are shown below vertex 4.

An entry in the neighbor list consists of two indices: the index of the strip of the neighbor tetrahedron and the position of its first vertex within the strip. For clarity, the index positions are shown on the top. Giving an example, the neighbor of our tetrahedron $\{4,3,8,6\}$ at face 0 is $\{3,8,6,7\}$ and is encoded as $(2,3)$ since 3 is the third vertex of strip 2. Tetrahedron $\{4,3,8,6\}$ itself has the index $(2,2)$. Note that we reference strips and tetrahedra beginning with 1 in order to avoid special handling for non-existing neighbors. They can simply be represented by $(0,0)$.

In order to describe how the same information can be stored more compactly, we first consider the special case of sequential strips, without swap operations introduced by replicated vertices. An example of a sequential strip is strip 2 in Figure 2(b). For the neighbor information we can distinguish three different cases:

Face neighbors in the same strip: Every tetrahedron in a sequential strip, except for the first and the last tetrahedron, has at least two neighbors in the same strip. The neighbor at face 0 (N_0) is the next tetrahedron in the strip, and the neighbor at face 3 (N_3) is the previous tetrahedron; e.g., for $\{4,3,8,6\}$ the N_0 neighbor is labeled with $(2,3)$, and the N_3 neighbor is denoted by $(2,1)$. We call these neighbors “implicit neighbors” since the connectivity is determined by the ordering of tetrahedra in a strip.

Face neighbors linking to a different strip: The adjacent tetrahedron associated with face 1 (N_1) and face 2 (N_2), e.g., $(1,3)$

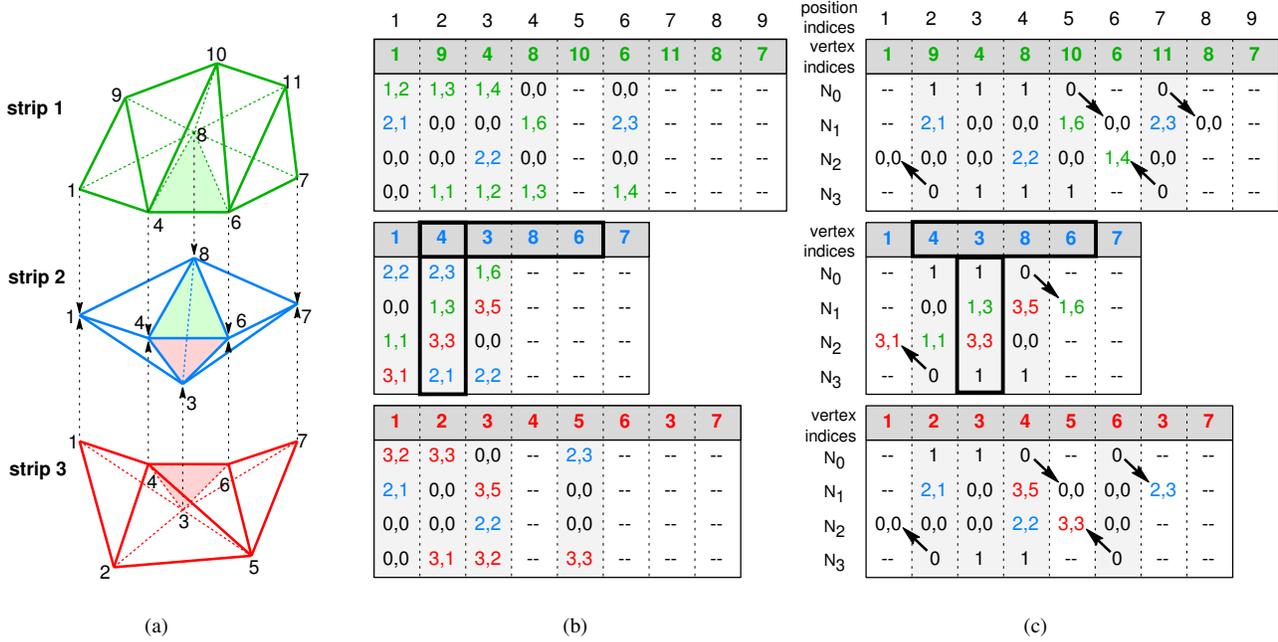


Figure 2: Example for generalized tetrahedral strips. The mesh is shown in (a), followed by the layout for the adjacency information (b), and our improved layout (c).

and (3,3), usually is a tetrahedron in a different strip. Therefore, they must still be stored as explicit indices and are called “explicit neighbors.”

Linking the ends of the strip: The first tetrahedron in the strip may be connected to a different strip by means of the N_3 neighborhood. In our example, $\{1,4,3,8\}$ in strip 2 has the N_3 neighbor (3,1). Similarly, the last tetrahedron in the strip may be linked to a different strip by means of N_0 . (See (1,6) in tetrahedron $\{3,8,6,7\}$ of strip 2.)

Taking advantage of the implicit neighbors, the structure presented in Figure 2(b) can be condensed as depicted in Figure 2(c). Note that we shift the neighbor information by one slot and place it below the second vertex of each tetrahedron. For example tetrahedron $\{4,3,8,6\}$ in strip 2 has its neighbors below vertex 3 in this scheme.

For most tetrahedra, only the N_1 and N_2 neighbors are encoded explicitly, while N_0 and N_3 are replaced by two flags indicating whether the current tetrahedron has implicit (value 1) or explicit (value 0) N_0 and N_3 neighbors. For value 0, it is still necessary to encode the neighbor explicitly. Fortunately, in sequential strips this happens only at the first and the last tetrahedron of the strip. In order to consider these explicit N_0 and N_3 neighbors without any overhead in our structure, we utilize the otherwise unused N_1 and N_2 entries of the next and previous tetrahedron respectively. Since we shifted neighbor information by one slot, we have a spare slot at the beginning of the strip and two spare slots at the end of the strip. Thus, we can store the explicit N_0 neighbor in the N_1 neighbor of the next slot and the explicit N_3 neighbor in the N_2 neighbor of the previous slot as indicated by the arrows in Figure 2(c).

In generalized strips, vertex replication is allowed in order to change the face along which the strip is continued. This enables the construction of longer strips. For example in order to continue strip 3 in Figure 2(a) after $\{3,4,5,6\}$ with $\{5,6,3,7\}$, vertex 3 has to be replicated. Consequently, $\{3,4,5,6\}$ is encoded twice in the same strip: as $\{3,4,5,6\}$ and $\{4,5,6,3\}$. However, the neighbor in-

formation is only considered for the first one, below vertex 3 (see Figure 2(b)) and the second one is called “virtual tetrahedron.”

Because of these virtual tetrahedra, N_0 and N_3 neighbors now become explicit even for a tetrahedron within the strip. Luckily, we do not need the N_1 and N_2 neighbor of virtual tetrahedra, which are automatically used for storing the now explicit N_0 and N_3 by the scheme already described. As an example, the N_3 neighbor for tetrahedron $\{5,6,3,7\}$ is encoded at N_2 below vertex 5.

4 STRIP GENERATION

The size of a tetrahedral mesh encoded as tetrahedral strip heavily depends on the quality of the stripification algorithm. Despite the extensive literature on the stripification of polygonal meshes, there seems to be hardly any work on the stripification of tetrahedral meshes except from [9]. They propose a greedy algorithm, using four different heuristics in order to determine the next tetrahedron for the strip.

Our approach is also based on a greedy algorithm that particularly tries to avoid isolated tetrahedra since they generate an overhead of three indices. It exploits the same data structure for creating sequential and generalized strips: For each tetrahedron we store the adjacent tetrahedra by their faces along with a flag indicating whether a tetrahedron has already been visited by the stripification. Each strip is first constructed in one direction until no following tetrahedron can be found; after that construction is continued backward starting from the beginning of the strip.

We employ various heuristics in order to determine the next tetrahedron in the current strip and the first tetrahedron for a new strip. In order to select the starting tetrahedra for the next strip, we employ four queues ($queue_i$, $i \in [1,4]$) which store tetrahedra with i unvisited neighbors. Starting points are always selected from the non-empty queue with the smallest index, which has turned out as the strategy leading to fewest isolated tetrahedra. Moreover, we found that filling the queues only with neighbors of already visited tetrahedra leads to more correlated strips, significantly increasing

the average strip length. Only if no unvisited neighbor exists, an arbitrary unvisited tetrahedron is selected.

In order to find the longest strip starting from a certain tetrahedron, we consider all possible starting combinations. Thus, up to 24 different vertex permutations are analyzed in case of four unvisited neighbors. Since for a tetrahedron with only one unvisited neighbor, the permutation can be freely chosen, in this case we instead consider all permutation of the neighbor. For further increase of the strip length, we can consider not only one starting point but a set of starting points. The stripification is executed in parallel on all starting points, selecting the longest resulting strip.

During the generation of each strip, we apply different heuristics in order to decide which tetrahedron to add next:

Sequential strips: Choose the next tetrahedron that generates a sequential strip.

Generalized strips: Choose the tetrahedron with the fewest unvisited neighbors and in case of ambiguities select the tetrahedron which introduces fewest replicated vertices.

We also experimented with a hybrid strategy that primarily selects the next tetrahedron in sequential order and follows the strategy for general strips in case the former is not possible. However, the results were always inferior to the generalized strips strategy.

5 RAYCASTING TETRAHEDRAL STRIPS

With the tetrahedral strip data structure presented in Section 3 we now have a compact representation which can be used to efficiently transfer tetrahedral meshes of reasonable size to the graphics card and apply rendering algorithms that run completely on the GPU without CPU interference. We adopt a ray casting algorithm for fragment shading hardware first presented in [16]. The basic idea is a ray propagation approach similar to [6] which is evaluated by the graphics processing unit. Starting from its first intersection with the mesh, each view ray is propagated front to back from cell to cell until the whole mesh has been traversed. A traversal step includes the computation of the exit point for the current cell, the determination of the scalar value at the exit point, the computation of the ray integral within the current cell, the accumulation of the color and opacity contribution, and the determination of the next cell from neighbor information.

The traversal is implemented in multiple passes. In each traversal pass, one polygon is rasterized that covers the projected area of the mesh’s boundary. Since each pixel of the polygon represents one view ray, a fragment program can compute the necessary operations per view ray based on the mesh data accessed from texture maps. Intermediate information about the traversal state—current cell index, position of the last intersection, scalar value at the last intersection, and color/opacity accumulated so far—is exchanged between passes via hardware-accelerated p-buffers, which are bound as texture maps for the next traversal pass.

Unfortunately, implementing this approach with our tetrahedral strip data structure generates more restrictive requirements for the graphics hardware. In particular, additional fragment program operations are needed for retrieving the required mesh information and computing properties which are not included in the tetrahedral strip data structure. Since this cannot be mapped to the 64 instruction slots of the ATI Radeon 9700 chip family employed in [16] without applying multi-passing for each traversal step, our implementation was performed on an nVidia GeForceFX graphics adapter. At the time of the implementation this was the only available hardware supporting long fragment programs with up to 1024 instructions in one pass.

5.1 Mesh Data Encoding

Several two-dimensional floating-point texture maps are utilized to store the tetrahedral mesh in the local memory of the graphics card at full precision. Since there is no power-of-two restriction for the size of floating-point textures on the nVidia card, we can adapt the size of all textures to the requirements of the data set, making the most effective use of the available texture memory. In particular, different sizes are possible for the textures storing vertices, normals, connectivity, etc. Table 1 provides an overview of the required textures. Note that in all mesh textures we initialize the texel at position (0, 0) with zero in order to avoid special handling for tetrahedra with no neighbors and to allow for ray termination, which is handled by an active cell of “0”.

The minimal texture set for the tetrahedral strip consists of an RGBA texture for the vertex list and an RGB texture map for the connectivity encoded in the strip. The vertex list texture stores the coordinates \mathbf{v}_k and the scalar value s_k of each vertex k in sequential order according to their indices.

The RGB connectivity texture stores one-to-one the encoded strings of the tetrahedra strips that were generated as described in Section 4. Each texel in this texture represents one tetrahedron including virtual tetrahedra generated for the general strips. The three overhead vertices for each strip are also represented by texels in this map. One entry in this texture contains $v_{t,0}$, which is the 16 bit two-component texture coordinate of the first vertex of the corresponding tetrahedron pointing into the vertex list texture. Additionally there are two sets of 16 bit texture coordinates $N_{t,1}$ and $N_{t,2}$ into the connectivity texture referring to the entries of the N_1 and N_2 neighbors respectively. The information whether the tetrahedron has implicit N_0 and N_3 neighbors is stored as the most significant bit of the u and v component of N_1 respectively and has to be removed prior to writing the index of the next cell into the traversal buffer. Accessing and removing this information from the texture coordinate can efficiently be achieved by only one fragment program instruction each. Note that the GeForceFX only supports two-dimensional texture maps with sizes up to 4096^2 ; thus, only 12 bits of each texture coordinate are actually required, leaving four spare bits per component for encoding supplementary information. Even extending the texture dimension to 2^{15} , which is sufficient for one billion tetrahedra, would still allow us to use one bit for additional data.

Our strip generation typically creates a large number of strips with varying lengths. In order to access consecutive entries of the strip by applying a signed offset to the u component of the texture coordinate pointing to the first vertex, the strips are laid out along the rows of the texture map, prohibiting wrapping within the strip.

Table 1: Summary of the textures described in the main text.

mesh data	tex. coords		texture data			
	u	v	r	g	b	α
tetstrip	t		$v_{t,0}$	$N_{t,1}$	$N_{t,2}$	—
vertices		k		\mathbf{v}_k		s_k
normal idx (opt.)	t		$n_{t,0}$	$n_{t,1}$	$n_{t,2}$	$n_{t,3}$
face normals (opt.)		j		\mathbf{n}_j		o_j
scalar data (opt.)		t		\mathbf{g}_t		\hat{g}_t

traversal data	channel	texture data (bits)			
		0-7	8-15	16-23	24-31
current cell	R		t		
intersection	G	λ		$s(\mathbf{e} + \lambda \mathbf{r})$	
acc. color	B		r		g
acc. color/opacity	A		b		α

To avoid unnecessary unused space in the connectivity texture, we apply a simple but effective greedy layout algorithm: We process all strips sorted by descending length, starting with the longest, and assign it to the texture line with the smallest but sufficient number of free slots. For fast computation of this slot, we utilize a free space list with one entry for each texture row. This list is sorted by ascending free slots after the placement of each strip. Texels in the texture map are filled in left-to-right order.

The connectivity texture and the vertex list texture provide all mesh data required for the ray casting computations. The plane equation for the tetrahedral faces, used in the computation of the exit points, can be computed from the vertex positions. The scalar value at the exit point can be interpolated from the scalar values at the vertices using barycentric interpolation where the interpolation weights are given as normalized distances from the opposite face:

$$s(\mathbf{x}) = \sum_{i=0}^3 w_i s_i \quad \text{with} \quad w_i = \frac{\mathbf{n}_i \cdot (\mathbf{x} - \mathbf{v}_{3-i})}{\mathbf{n}_i \cdot (\mathbf{v}_i - \mathbf{v}_{3-i})} \quad (1)$$

However, since long fragment programs result in reduced frame rates, we trade space for performance by optionally defining three additional texture maps if enough texture memory is available. The computation of the plane equation for the tetrahedral faces is replaced by the lookup in a floating-point texture storing the normal components \mathbf{n}_j and the offset o_j of the plane equation in RGBA. In order to allow for sharing faces between adjacent tetrahedra we use an indirect addressing scheme. A normal index map with the same layout as the connectivity texture stores for each tetrahedron the four texture coordinates $n_{t,0..3}$ for the face parameters in the face texture. In each of these texture coordinates the most significant bit of the u coordinate indicates whether the normal vector has to be flipped in order to achieve an outward facing normal. The interpolation of the scalar value at the exit point can be accelerated exploiting Equation (2),

$$s(\mathbf{x}) = \mathbf{g}_t \cdot (\mathbf{x} - \mathbf{x}_0) + s(\mathbf{x}_0) = \mathbf{g}_t \cdot \mathbf{x} - \mathbf{g}_t \cdot \mathbf{x}_0 + s(\mathbf{x}_0) = \mathbf{g}_t \cdot \mathbf{x} + \hat{g}_t \quad (2)$$

which states that the interpolation for an arbitrary point inside the tetrahedron can be computed by a dot product between the coordinates of that point and the gradient \mathbf{g}_t of the tetrahedron plus a constant \hat{g}_t . We optionally store these parameters in a gradient texture map with the same layout as the connectivity texture. The gradients are also employed to perform the lighting when ray casting isosurfaces.

5.2 Ray Traversal

As mentioned before, the traversal of the view rays through the tetrahedral mesh is implemented as a multi-pass rendering of a quad with the same pixel coverage as the projection of the mesh, applying an appropriate fragment program. After each traversal pass intermediate results are written into a floating-point p-buffer which can be loaded as a texture map for the consecutive passes. Usually we have to apply ping-pong rendering with two p-buffers since concurrent read/write operations are not allowed according to the specification of the `WGL_draw_buffer` extension. However, we can avoid the implied overhead of continuous texture bind operations by binding the current render target simultaneously as texture map, which is an undocumented feature of the GeForceFX card. For comparison, a variant of our system without this workaround has also been implemented using the front and back color buffer of a floating-point p-buffer as ping-pong targets. This is necessary since the employed ray termination mechanism requires to share depth values between the rendering targets, which in OpenGL can only be achieved in this way. In DirectX9, color surfaces and depth surfaces are created and bound independently.

Based on this traversal data structure, the GPU-based multi-pass algorithm is performed as presented in Figure 3. We start with an initialization rendering pass that determines the first hit of each view ray with the boundary of the mesh. This is achieved by rendering the list of boundary triangles with the index of the corresponding tetrahedron specified as additional texture coordinate. The mesh coordinates are also specified as texture coordinates, resulting in the interpolation of the actual intersection point position for each pixel.

In the structure of the traversal buffer presented in Table 1, we had to deal with the restriction of the GeForceFX, which only supports a single 4-component 32 bit floating-point output target. However, seven parameters are required for the traversal algorithm consisting of the texture coordinate t pointing to the current cell, the ray parameter λ representing the position of the last intersection, the corresponding scalar value and the accumulated color/opacity as RGBA value. In order to fit those into 128 bits, we employ the 16 bit floating-point format defined by the `NV_fragment_program` extension that also provides the necessary operations for packing and unpacking two 16 bit floating-point or unsigned short values into one 32 bit float. Note that we exploit the same operations for unpacking the two unsigned short components of the texture coordinates encoded in the mesh textures. However, this requires a careful assembly of the connectivity texture map since the GeForceFX uses different endianness than the CPU. In an implementation for the GeForce6 chip series we could have avoided the limited precision for the traversal parameters by exploiting multiple render targets. However, even with the 16 bit precision we never experienced visual artifacts during our tests.

The current cell index and intersection parameters are written into the traversal p-buffer. Based on these values, several traversal passes are performed for computing the ray traversal. The traversal stops if no more pixels are set during the rendering. This is detected by an occlusion query (`NV_occlusion_query`) eventually issued with the traversal rendering. In order to avoid the rasterization of fragments for completed view rays special termination passes are used which modify the depth value of fragments with a current cell index of "0"; thus, fragments are potentially discarded by an early depth-test.

```

// Phase I - Traversal
for (max number of peels)
{
    activeViewRayCount = drawFirstHit();

    if (activeViewRayCount == 0)
        break;

    while (activeViewRayCount > threshold)
    {
        pass++;

        if (pass % terminationPassFrequency == 0)
            drawTermination();

        if (pass % occlusionPassFrequency == 0)
            activeViewRayCount = drawTraversalWithOcc();
        else
            drawTraversal();
    }
}

// Phase II - Final pass into frame buffer
drawFinalPass();

```

Figure 3: Traversal algorithm for the GPU-based ray casting.

5.3 Non-Convex Tetrahedral Meshes

As illustrated in Figure 4 the ray propagation from cell to cell is only correct for convex tetrahedral meshes. In non-convex meshes, view rays that have already left the mesh may re-enter the mesh. This situation cannot be handled by the GPU-based ray caster presented so far. In [16] this has been addressed by a convexification process that fills the area between the boundary of the mesh and its convex hull with virtual tetrahedra, which require special handling by the fragment program. Moreover, the computation of this convexification is an expensive pre-process that might generate many additional tetrahedra.

Our implementation provides intrinsic support for non-convex tetrahedral meshes without the need for computationally expensive preprocessing by applying a technique similar to depth peeling [5]. The basic idea is to perform several traversal cycles as described in Section 5.2. In the first cycle, we traverse all view rays, starting from the first intersection with the mesh boundary, which correspond to the red rays in Figure 4. The following cycles restart the traversal from the second (blue), third (green), etc. intersection of the view rays with the mesh boundary. Note that since we always use the same traversal buffer for each cycle without clearing, the accumulated color/opacity from the previous cycle is correctly blended with the new cells.

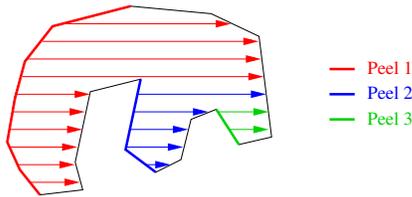


Figure 4: With our GPU-based ray casting implementation view rays are only traversed until they first leave the mesh, which is a problem for non-convex tetrahedral meshes. Extracting the second, third, etc. layer of front faces by depth peeling allows us to find reentries of view rays.

In order to restart the ray traversal, the extraction of the second, third, etc. layer of boundary faces is required. This is achieved with a two-way depth-test implemented in a fragment program. With enabled back face culling only front-facing triangles from the list of boundary faces are extracted, and only the primitives closest to the viewer pass the regular `GL_LESS` depth-test. In addition to that, a fragment program compares the depth value of all fragments with the depth values of the previous layer of boundary faces provided by a texture map. Fragments with a depth value less or equal than the depth value in the texture map are discarded, effectively “peeling-off” the previous layer of boundary polygons. Enhancing the ray caster with respect to this technique only requires to read back the depth values after the first-hit rendering, and providing them in a texture map to the first-hit fragment program extended by the additional depth-test (see Figure 5 for an example). Note that a similar idea has been developed parallel to ours in [3]. They report that a bias for the depth test may be required in order to prevent self-occlusion. However, according to our experiences this is not necessary for the GeForceFX.

5.4 Ray Casting Fragment Program

We implemented the fragment programs for the first-hit and the traversal computations with nVidia’s high-level shading language Cg [10]. With the instruction set of Cg the implementation is straightforward. We will therefore not present particular code here. We refer to [16] for more detailed information.

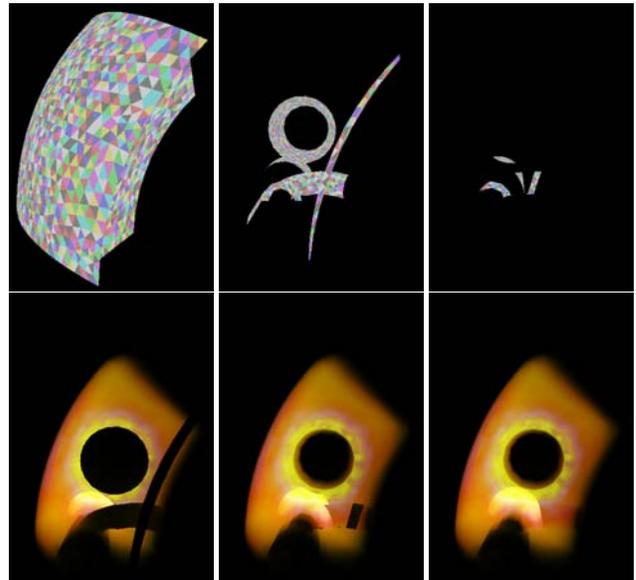


Figure 5: Volume rendering of the Super Phoenix (spx) data set employing our depth peeling technique. The data set is courtesy of Bruno Notrosso (Electricité de France).

Fragment program sizes are influenced by the optical model and range from 176 to 213 instructions for the space-optimized implementation and from 152 to 167 with the performance-optimized implementation. The more expensive isosurface program mainly results from the additional lighting computation requiring a lookup in the gradient texture, the normalization of the gradient, and the dot product with the view ray direction. The overhead for the on-the-fly computation of the gradient imposes an even larger overhead.

Note that the 1024 instructions supported by the GeForceFX chip enabled us to perform multiple traversal steps in the same fragment program by implementing the traversal in a static loop, which can be unrolled by the Cg compiler. Using these long fragment programs has the advantage of reducing the expensive read/write operations to and from the traversal buffer, therefore, resulting in better performance. Exploiting the full instruction limit of the graphics adapter, we were able to perform 5-6 traversal steps per pass in the case of the fragment programs for optimal data set size and one more traversal step for the program set with the minimized instruction numbers.

6 RESULTS

We tested our implementation on an Intel Pentium 4 2.80 GHz processor with 2 GB memory, and an nVidia GeForce 6800 GT graphics board running at 350 MHz with a memory clock rate of 1000 MHz. Our stripification code was tested on various data sets with sizes ranging from 1715 to 148995 tetrahedra.

6.1 Encoding

Table 2 shows our results for generating sequential and general strips. Note that in the table only one line is presented for the Sphere and the E11 data set since they share the same original connectivity as both correspond to a uniform 32^3 grid which has been decomposed into five tetrahedra per cell. In the sequential case our stripification algorithm shows encoding results with average strip length around 4.3 tetrahedra. The number of isolated tetrahedra ranges from 6% to 8%. However, if we select the starting points

Table 2: Results for the sequential (Seq) and generalized (Gen) stripification.

data set	n_tetra	n_ver	n_strips		av_length		max_length		% isolated		% compression	
			Seq	Gen	Seq	Gen	Seq	Gen	Seq	Gen	Seq	Gen
Cube	1715	512	436	205	3.93	8.36	34	143	6.06	3.32	66.10	59.87
Fighter	70125	13832	16169	7274	4.34	9.64	281	2249	7.56	3.15	63.44	59.20
Spx	103488	37320	24009	11161	4.31	9.27	281	2235	8.06	3.38	63.60	59.46
Heatsink	121668	23576	27855	12503	4.37	9.73	198	2505	7.86	2.79	63.26	56.53
Sphere/Ell	148955	32768	34319	15201	4.34	9.80	154	1037	7.95	2.84	63.42	57.58

randomly, the average strip length decreases by 1% and the number of generated strips by 3%. The overall compression rate varied between 63% and 66%. According to Equation (3) we will never get below $3/8 \approx 37\%$ the size of the original connectivity information:

$$\frac{3(t+3n)}{8t} = \frac{3}{8} + \frac{9}{8} \frac{t}{n} \quad (3)$$

Our sequential stripification scheme for a mesh with t tetrahedra decomposed into n strips requires $3(t+3n)$ indices; one vertex and two neighbors per tetrahedron and three vertices overhead for each strip. In contrast, $(4+4)t$ indices for vertices and neighbors are required when storing the tetrahedra in a plain list. For an improvement with respect to a tetrahedra list the average strip length t/n has to be larger than $9/5 = 1.8$, which was easily achieved for the tested data sets.

Results for encoding general strips are also presented in Table 2. They were generated considering the 24 different permutations of the starting tetrahedra. In this case there is almost no difference in the results if we consider the permutations over the first neighbor of the starting tetrahedron. This is because with generalized strips all the combinations are tested during the generation of the strip. Almost all the starting tetrahedra were selected from the *queue*₁. In this case, the average length of the strip is around 9.4 tetrahedra per strip, not including virtual tetrahedra, and the longest consists of 2505 tetrahedra. However, the number of isolated tetrahedra is always less than 3.4% of the input data, leading to an overall compression between 56% and 60%.

Probably the most important advantage of our proposed data structure is the low memory profile, which allows us to store large data sets in the local memory of the graphics card. A maximum supported data set size of roughly 500.000 tetrahedra was the most severe limitation of the previous solution presented in [16]. With respect to the maximum size of a mesh that can now be stored on the graphics card we provide the memory consumption C of a tetrahedral mesh encoded in our strip data structure:

$$C = C_t n_i + C_v n_v$$

$$n_i = n_t + 3n_s + n_r = \bar{l}_s n_s$$

C_t	size of an index in the tetstrip texture (= 96 bit)
C_v	size of a vertex in the vertex list texture (= 128 bit)
n_i	number of indices in all strips
n_v, n_t	number of vertices/tetrahedra in the data set
n_s	number of strips
n_r	number of vertex replications
\bar{l}_s	average number of vertices in a strip

According to this formula we can estimate the upper bound for a data set that can be processed: In an ideal encoding we only have one sequential strip ($n_s = 1$) without replicated vertices ($n_r = 0$). Provided the reasonable assumption ($n_v \approx \frac{1}{6} n_t$) this gives a total

memory consumption ($C_t + \frac{1}{6} C_v$) ≈ 117 bit per tetrahedron, neglecting the constant term $3C_t$, which can be justified for a large number of tetrahedra. We compare this to the 256 MByte of memory available on the graphics card, of which at least 12 MByte are consumed by a 1280×1024 pixel framebuffer with 32 color bits and 24 depth bits, a 400×400 pixel p-buffer for the traversal with 128 bit floating-point color and 24 bit depth, and a two-dimensional pre-integration texture. Under these assumptions even a commodity off-the-shelf graphics adapter is capable of dealing with 17.5 million tetrahedra. Upcoming graphics card generations aim at 512 MByte as standard texture memory size, bringing 35.8 million tetrahedra into reach.

6.2 Rendering

Table 3 presents the timings that our GPU-based ray caster achieved for the images presented in this paper acquired for a 400×400 viewport. Sphere, Ell (volume), and Ell (iso) correspond to the left, middle, and right image of Figure 1 respectively. Spx denotes the data set in Figure 5 and was rendered with four depth peels. Timings were recorded during an animation where the data set was rotated about the x- and the y-axes with different angular velocity. The table contains the average, minimum, and maximum framerate for the animation. For all test cases interactive framerates of several frames per second could be achieved.

Comparing the space-optimized and the performance-optimized fragment program version, we experience almost twice the framerate for the performance-optimized implementation due to a significant reduction of arithmetic instructions. Note that the number of texture sampling instructions is almost identical since we can avoid the lookup of the vertex coordinates in the performance-optimized implementation. However, this performance increase comes along with significant memory overhead (≈ 608 bit per tetrahedron instead of 117 bit per tetrahedron). We consider this acceptable since it still represents an improvement compared to the 1280 bit per tetrahedron reported in [16] and allows for the handling of over 3 million tetrahedra. We also experimented with different variations using only normal textures or only gradient textures. The performance was slightly faster than with the space-optimized program, but the increased number of texture lookup operations almost out-

Table 3: Framerates per second for the GPU-based ray caster employing either the space-optimized or the performance-optimized fragment program.

data set	passes	space-opt.			perf.-opt.		
		avg	min	max	avg	min	max
Sphere	108	4.0	3.5	4.7	6.4	4.4	7.5
Ell (volume)	129	2.7	2.2	3.1	4.3	2.9	5.1
Ell (iso)	128	2.9	2.2	3.4	4.8	3.2	5.7
Spx	346	2.7	2.2	3.2	4.0	3.3	4.7

weigh the saved arithmetic instructions.

Unfortunately, despite our optimization efforts documented in Section 5, the performance of our ray casting implementation finally was not quite as we could expect given the results from [16] for the ATI Radeon 9700 card. The GeForce 6800 should be significantly faster than the last generation ATI chip. According to our analysis this is mainly caused by the fact that the early depth-test is significantly less effective on the nVidia card, which causes our algorithm to continuously perform ray computations for pixels that are already fully computed. This analysis is based on two observations: First, employing texture kill instructions in the fragment program instead of employing special ray termination passes we could improve the framerate by up to 14%, although the texture kill instruction is known for disabling the early depth-test. Second, splitting the screen-sized rectangle used for traversal into several tiles as suggested in [3], performing the ray termination test for each tile individually, and completely avoiding the rendering of tiles containing only finished rays leads to significantly higher framerates. Theoretically, if the early depth-test prohibits further computation on finished rays, the same or even lower performance should be expected for tile-based rendering. The timings in Table 3, therefore, were acquired with the viewport split into 16×16 tiles, which we experienced as optimal with respect to the overhead introduced for the tile handling.

Texture bind operations turned out not to be the bottleneck for our implementation, even if they are not avoided by simultaneously binding the current render target as texture (see Section 5.2). The difference was less than 2% and also the variation for performing multiple traversal steps per rendering pass was below 8%. However, using multiple traversal steps per rendering pass has an effect on the accumulated opacity, which triggers the early ray termination. Every time the opacity is written to the p-buffer it is quantized to 16 bit whereas within multiple traversal steps in a single rendering pass the opacity is always computed and stored with 32 bit precision. This might lead to a slightly different number of overall traversal step. However, a visual effect could not be noticed.

7 CONCLUSION

We have presented the first system to exploit tetrahedral strips as volumetric primitives in off-the-shelf graphics hardware. It is based on a new data structure which extends the classical tetrahedral strips by compact neighbor information indispensable not only for volume rendering of unstructured meshes but a great variety of filtering and mapping algorithms. Exploiting this data structure, our approach encodes the strips into textures and renders them with the help of a ray casting algorithm running solely on the graphics chip. In contrast to previous approaches, the low memory requirements of our texture encoded tetrahedral strips allow for the handling of data sets with several millions of tetrahedra, intrinsically supporting non-convex meshes. Future work, concentrating on exploiting texture paging, might even result in support for data sets exceeding the physical memory of the graphics card.

Universality is an important advantage of our strip data structure in the sense that it supports sequential as well as general strips. It can therefore be used with virtually any stripification algorithm. However, only few approaches for the generation of tetrahedral strips exist and are mainly based on heuristics; thus, there is still need for more research.

With respect to performance considerations, our approach seems to be hampered by limitations of current graphics hardware, which should be removed with upcoming graphics card generations. Moreover, we expect our approach to fully benefit from the advanced performance of future graphics card generations, since the data transfer bottleneck between the CPU and the GPU is completely removed.

ACKNOWLEDGEMENT

This work was partially supported by the Ministry of Science and Technology of Spain under contract MCYT-FEDER TIC2001-3694-C02 and by the Secretaría Xeral I+D of Galicia (Spain) under contract PGIDIT03TIC10502PR.

REFERENCES

- [1] K. Akeley, P. Haeberli, and D. Burns. The tomesh.c program. Technical report, Silicon Graphics, 1990. available from SGI Developer's Toolbox CD.
- [2] C. Beeson and J. Demer. Nvtristrip, library version. Software available via Internet web site, <http://developer.nvidia.com/>, January 2002.
- [3] Fabio F. Bernardon, Christian A. Pagot, Joao L. D. Comba, and Claudio T. Silva. GPU-based Tiled Ray Casting using Depth Peeling. Technical report, SCI Institute, University of Utah, 2004.
- [4] F. Evans, S. S. Skiena, and A. Varshney. Optimizing Triangle Strips for Fast Rendering. *Visualization*, pages 319–326, 1996.
- [5] Cass Everitt. Interactive Order-Independent Transparency. Technical report, nVidia whitepaper available from www.nvidia.com/object/Interactive_Order_Transparency.html, 2001.
- [6] Michael P. Garrity. Raytracing Irregular Volume Data. In *Proceedings of the 1990 Workshop on Volume Visualization*, pages 35–40. ACM Press, 1990.
- [7] S. Gumhold, S. Guthe, and W. Straßer. Tetrahedral Mesh Compression with the Cut-Border Machine. pages 51–58, 1999.
- [8] S. Gumhold and W. Straßer. Real Time Compression of Triangle Mesh Connectivity. In *SIGGRAPH'98 Conference Proceedings*, pages 133–140, 1998.
- [9] Davis King, Craig M Wittenbrink, and Hans J. Wolters. An Architecture for Interactive Tetrahedral Volume Rendering. In Klaus Mueller and Arie Kaufman, editors, *Proc. of the Intl. Workshop on Volume Graphics 2001*, pages 163–180. Springer-Verlag, 2001.
- [10] nVidia. Cg language specification, 2002. Cg Language Specification, available at <http://developer.nvidia.com/cg>.
- [11] J. Rossignac. Edgebreaker: Connectivity Compression for Triangle Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, 1999.
- [12] P. Shirley and A. Tuchman. Polygonal Approximation to Direct Scalar Volume Rendering. In *Proceedings San Diego Workshop on Volume Visualization, Computer Graphics*, volume 24, pages 63–70, 1990.
- [13] B. Speckmann and J. Snoeyink. Easy triangle for TIN terrain models. In *Canadian Conference on Computational Geometry*, pages 239–244, 1997.
- [14] A. Szymczak and J. Rossignac. Grow & Fold: Compression of Tetrahedral Meshes. In *Proc. ACM Symp. on Solid Modeling and Applications*, pages 54–64, 1999.
- [15] C. Touma and C. Gotsman. Triangle Mesh Compression. *GI98 Conference Proceedings*, pages 26–34, 1998.
- [16] Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proc. IEEE Visualization '03*, pages 333–340. IEEE, 2003.
- [17] X. Xiang, M. Held, and J. S. B. Mitchell. Fast and effective stripification of polygonal surface models. In *ACM Sympos. Interactive 3D Graphics*, pages 71–78, 1999.