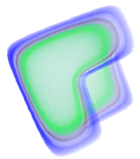
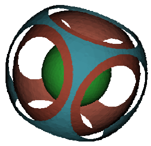


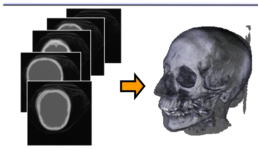


Hardware-Based Volume Ray Casting

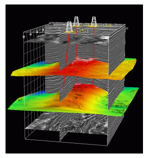
Manfred Weiler

Institute of Visualization and Interactive Systems
University of Stuttgart

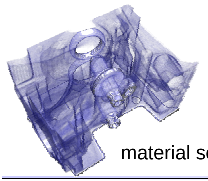
Motivation – Volume Data



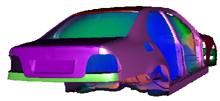
medical imaging



geology



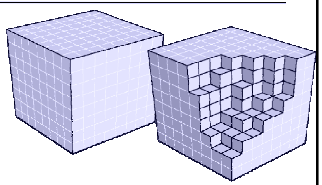
material science

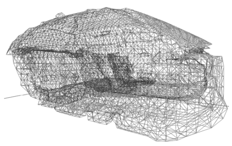


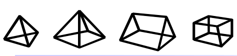
automotive engineering

Grid Types

- Uniform rectininear grid (voxels)
- Reconstruction with trilinear interpolation

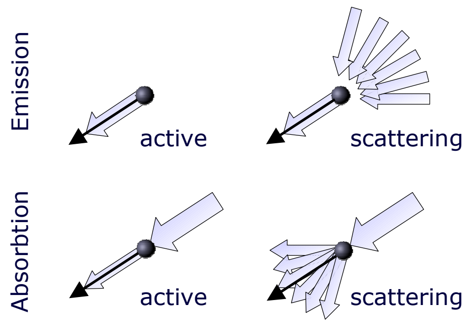






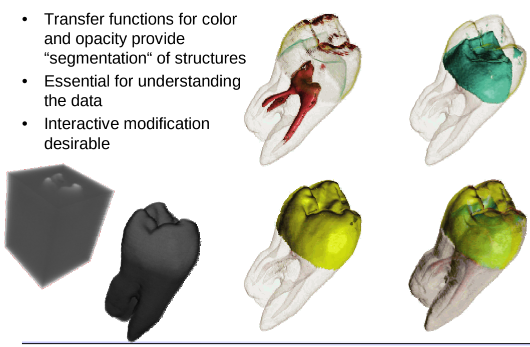
- Unstructured grid
- Decomposed into tetrahedra
- Reconstruction with linear interpolation

Physical Model of Radiative Transfer



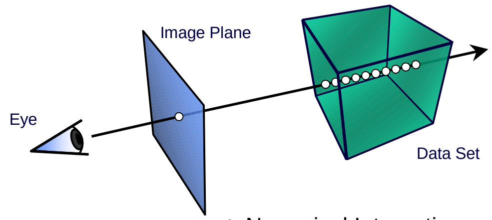
Transfer Function

- Transfer functions for color and opacity provide “segmentation“ of structures
- Essential for understanding the data
- Interactive modification desirable



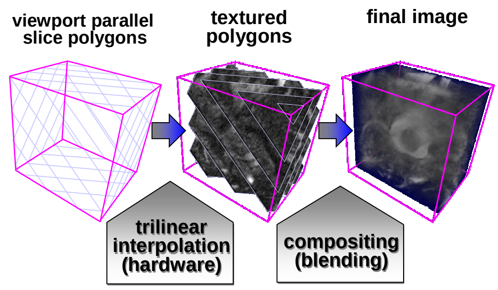
Ray Casting in Regular Meshes

Ray Casting



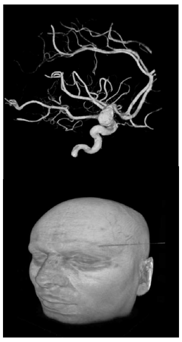
- Numerical Integration
- Resampling
- ⇨ High Computational Load

Texture-Based Volume Rendering



Problems

- Fragment processing overhead
  - Lookup and trilinear interpolation
  - Lighting computation, blending etc.
- Typically:
  - Emphasize boundaries
  - Select material values
- Only about 0.2% and 4% of all fragments contribute to the final image



Hardware-Based Ray Casting

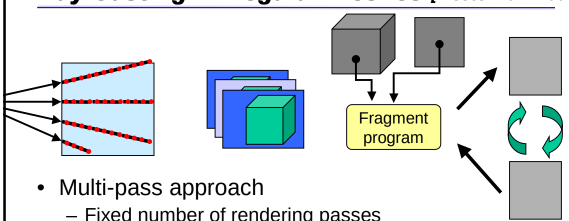
- Possible optimizations
  - Early ray termination
  - Empty space leaping
  - Adaptive sampling
- Hardly applicable to texture-based volume rendering
- Combine dynamic sampling and hardware acceleration

$\Rightarrow$ Ray casting in graphics hardware

Hardware Mapping

- Idea:
  - Parallel ray casting
  - Parallel traversal of all view rays
  - Fragment program for computing ray integration and ray traversal
- Ideally:
  - Render ONE screen-sized rectangle
  - Complete volume integral in ONE large fragment program
- Problems:
  - Currently no dynamic loops
  - Insufficient number of fragment operations
  - Only fragment kill – no real program abort
  - Radeon 9800: unlimited, dynamic?
- DirectX9 implementation for Radeon 9700

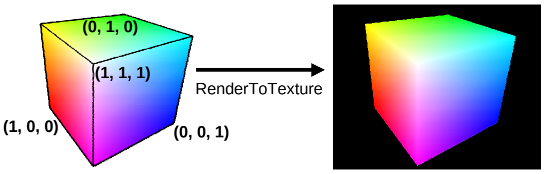
Ray Casting in Regular Meshes [Westermann2003]



- Multi-pass approach
  - Fixed number of rendering passes
  - Constant number of steps per pass
  - 2D textures for accumulating color and opacity
  - Access volume data from 3D texture map
- Additional pass for ray termination

Ray Setup

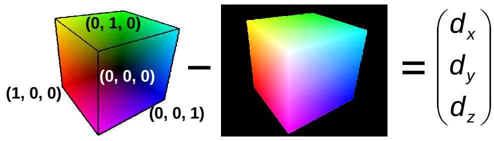
- Pass 1: Entry point determination
  - Render front faces of volume bounding box
  - Assign local texture coordinates as face colors



  - Store result in 2D texture (TEX0)
  - Fragment color corresponds to texture space coordinates of first intersection

Ray Setup (cont.)

- Pass 2: Ray direction determination
  - Render back faces of volume bounding box
  - Assign local texture coordinates as face colors
  - Issue raster position with each vertex `(texcoord0)`



- Fragment shader computes 2D texture (DIR)
  - Ray direction: normalize(COL – TEX0) ⇒ RGB
  - Length of ray segment ⇒ alpha

Ray Traversal

- Pass 3 to N:
  - Render front faces of volume bounding box
  - Issue raster position with each vertex `(texcoord0)`
  - Local texture coordinates of each vertex `(texcoord1)`
  - Global counter as constant color `(cnt)`
  - Ray increment as second constant color `(delta)`
  - Perform M traversal steps per pass:

```
// initial ray position
r = texcoord1 + cnt * DIR(texcoord0);
// increment ray position
r = r + delta;
```


Ray Integration

- Inner loop:
  - Lookup scalar value at position `r` from 3D texture
  - Accumulate color and opacity in register

$$C_{dst} = C_{dst} + (1-\alpha_{dst})\alpha_{src} C_{src}$$
$$\alpha_{dst} = \alpha_{dst} + (1-\alpha_{dst})\alpha_{src}$$

- Outer loop:
  - Read color and opacity from 2D texture (RES)
  - Blend with locally accumulated color/opacity
  - Store result back in RES
  - Write opacity to 1 if ray already left the volume

Ray Termination

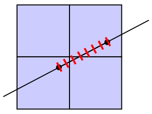
- Intermediate pass:
  - Render front faces of volume bounding box
  - Issue raster position with each vertex
  - Read accumulated opacity
  - Write z_{far} if opacity exceeds threshold z_{near} else
  - Exploit early z-test (GL_GREATER)
  - Blend with zero opacity to preserve color

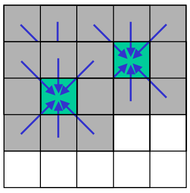
Isosurface Ray Casting

- Inner loop:
  - Perform ray traversal per-pass back-to-front
  - Store current ray position in register if scalar value greater than threshold
  - Potential overwrite with every new sample point
  - Back-to-front traversal results in first hit closest to the viewer
- Outer loop:
  - Check if register has been altered
  - Perform surface lighting with normals from 3D gradient texture
  - Assign opacity of 1 to terminate ray

Empty Space Skipping

- Determine empty space prior to each traversal pass
- Based on a data structure encoding empty space
  - Regular grid – cells of 8^3 voxels
  - Emptiness for whole ray segment required
  - Minimum and maximum per block of 3x3x3 cells
  - Stored as a 3D texture map





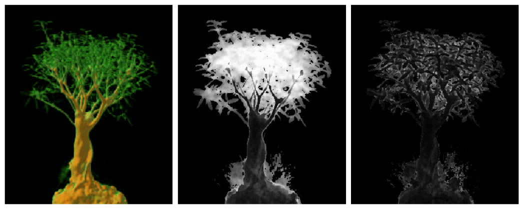
Empty Space Skipping (cont.)

- 2D texture map identifies empty regions
  - Parameterized with minimum/maximum values
  - With respect to current transfer function
  - CPU-computed
  - Updated on transfer function modification
- Modify intermediate pass
  - Before each traversal pass
  - Sample coarse grid at current ray position
  - Determine visibility by a dependent texture lookup
  - Lock fragments by writing z_{far} to z-buffer

Adaptive Sampling [Röttger2003]

- Importance volume
  - Store appropriate step length per voxel
  - Based on second derivative of volume data
  - With respect to transfer function
- Replace fixed step length by lookup in importance volume
- Requires dynamic ray termination
  - E.g. DirectX 9 occlusion query
- Subsumes space leaping

Results

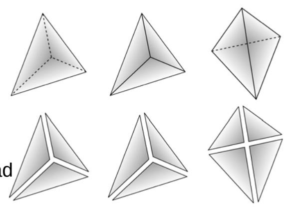


- Effect of early ray termination depends on transfer function

Ray Casting in Tetrahedral Meshes

Motivation

- Volume rendering for unstructured grids most commonly performed using Shirley Tuchman Projected Tetrahedra
- Limited benefit from rapid development of graphics hardware
- View dependent
  - Computational overhead
  - Memory bandwidth
  - Graphics bus bandwidth



Motivation

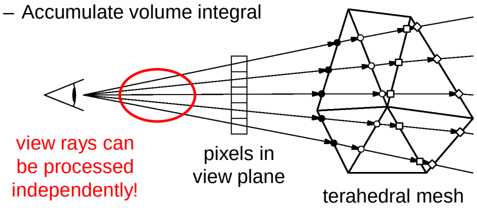
- Goal
  - Remove the memory bottleneck
  - Graphics hardware runs at full capacity

⇒ Build an algorithm that completely runs on the graphics hardware

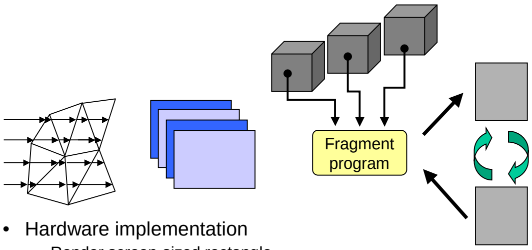
- Requirements for suitable hardware algorithms
  - Parallel implementation straightforward
  - No random access memory writes

Ray Propagation Approach

- Software implementation [Garrity1990]
  - Traverse rays front to back
  - Stop at intersected cell faces
  - Compute color and opacity for current ray segment
  - Accumulate volume integral

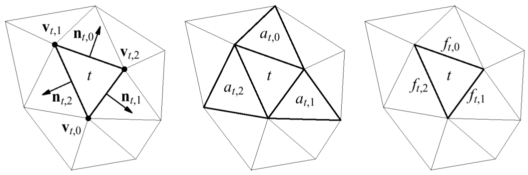


Hardware-Based Ray Propagation



- Hardware implementation
  - Render screen-sized rectangle
  - One rendering pass per propagation step
  - Intermediate information communicated via 2D RGBA textures
  - Mesh data accessed from 3D texture maps

Mesh Data



- Vertices, normals, scalars, neighbor data
- Stored with full precision as 32 bit float texture
- Requires four 3D texture maps
- Indices encoded in two components

Traversal Data

- Current cell, intersection point, accumulated color
- Floating point 2D textures
- Size equals viewport size
- Addressed with raster position

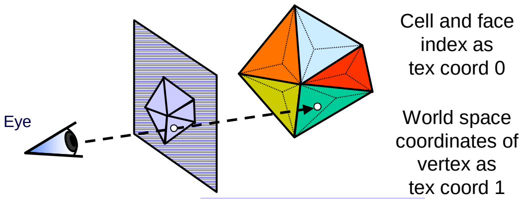
data in texture	texture coord.		texture data			
	u	v	r	g	b	a
current cell	raster pos		t		j	
intersection point	raster pos				λ	s(**e**+λ**r**)
color, opacity	raster pos		r	g	b	a

Multi-Pass Algorithm

1. initialization (1 pass)
   determine first hit
2. while still within the mesh (n passes)
   (a) compute exit point for current cell
   (b) determine scalar value at exit point
   (c) compute ray integral within current cell
   (d) blend to color buffer
   (e) proceed to adjacent cell through exit point

First Hit

- Requires current cell, intersection face and point
- Render boundary faces
  - Extract visible faces with back face culling



$$\lambda = \sqrt{dot(\vec{t}_1 - \vec{e}, \vec{t}_1 - \vec{e})}$$

Scalar Value

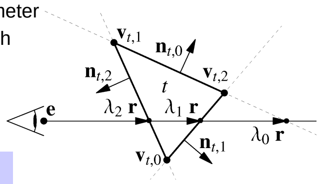
- No full linear interpolation from vertex scalars
- Reduce and fragment program operations

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

- Only two operations
  - Dot product and sum
- Same amount of data
  - Four floats per tetrahedron

Exit Point Computation

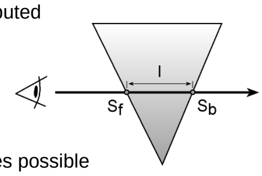
- Compute ray intersection with tetrahedra faces
  - Exclude entering face (only 3 tests)
  - Only non-visible faces ($\mathbf{r} \bullet \mathbf{n}_{t,i} > 0$)
  - Minimum ray parameter
  - Straight forward with fragment program operations



$$\mathbf{r}(\lambda) = \mathbf{e} + \lambda \mathbf{r}$$
$$\lambda_i = \frac{(\mathbf{v} - \mathbf{e}) \bullet \mathbf{n}_{t,i}}{\mathbf{r} \bullet \mathbf{n}_{t,i}}; \quad \mathbf{v} := \mathbf{v}_{t,3-i}$$

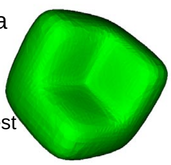
Ray Integration

- Perform shading via pre-integrated classification [Röttger2000]
- Observation: Ray integral only depends on S_f, S_b, and l
- Store numerically pre-computed ray integral in a 3D texture.

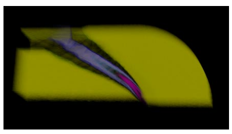


- Different shading techniques possible
  - Emission, absorption, isosurfaces, MIP
  - Density-emitter
- Arbitrary transfer functions possible
  - Affect only generation of texture map

Cell Traversal

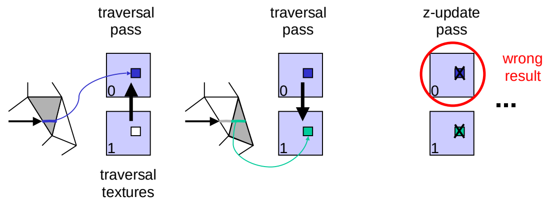


- Current cell index in traversal data
- Updated from neighbor cell $a_{t,i}$
- Boundary cells with index –1
  - ⇒ Identifying λ_{new} and λ_{old} or early z-test
- Non-convex meshes
  - ⇒ Imaginary cells for handling reentries
- Early ray termination
  - Same mechanism



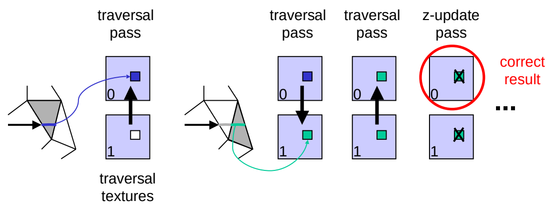
Terminating Finished Rays

- Stop fragment processing
- Exploits early z-test (GL_LESS)
- Write z_{near} if current cell equals –1
- Special z-update passes (eventually)



Terminating Finished Rays cont.

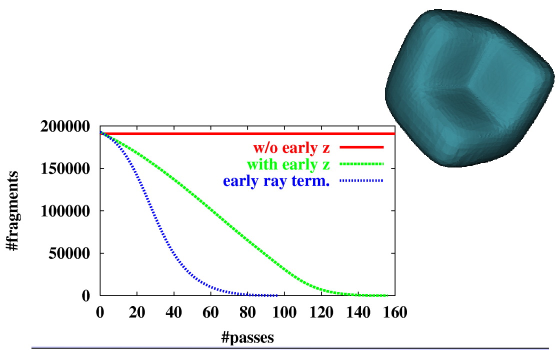
- Z-updates only after even traversal passes
- Final result always in texture set 0
- Copy color in traversal pass (current cell = –1)



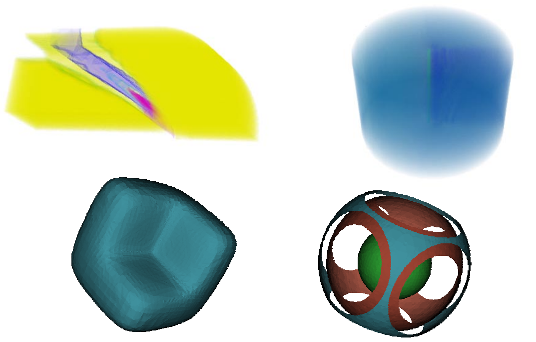
Terminating Rendering

- Exploit DirectX 9 occlusion query
- Render until no more pixels are set
- Do not wait for asynchronous delivery
- Overhead of additional rendering passes neglectable

Results



Results



Summary

- Volume ray casting
  - For regular meshes
  - For tetrahedral meshes
- Exploits features of programmable graphics hardware
- Benefits from reduced number of fragment operations
  - Early ray termination
  - Space leaping
  - Adaptive sampling
- Rasterization more complex
- Where is break even point?

References

[Garrity1990] M. P. Garrity. Raytracing Irregular Volume Data. In Proceedings of the 1990 Workshop on *Volume Visualization, pages 35-40. ACM Press, 1990.*

[Purcell2002] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. In Proceedings of ACM SIGGRAPH 2002, volume 21, pages 703-721, 2002.

[Röttger2000] S. Röttger, M. Kraus, and T. Ertl. Hardware-Accelerated Volume and Isosurface Rendering Based On Cell-Projection. In Proceedings IEEE Visualization 2000, pages 109-116. ACM Press, 2000.

[Röttger2003] S. Röttger, S. Guthe, D. Weiskopf, T. Ertl. Smart Hardware-Accelerated Volume Rendering, In Proceedings of EG/IEEE TCVG Symposium on Visualization *VisSym '03* (to appear), 2003

[Westermann2003] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In Proceedings IEEE Visualization 2003 (to appear), 2003.

[Weiler2003] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. In Proceedings IEEE Visualization 2003 (to appear), 2003.