Tutorial 7
Real-Time Volume Graphics

Klaus Engel
Markus Hadwiger
Christof Rezk Salama

Applications: Medicine

CT Human Brain:
Posterous Roman Project,
II National Library of Medicine, Maryland, USA

CT Angiography:
Dept. of Neuroradiology,
University of Erlangen, Germany

Applications: Geology

Deformed Plasticine Model,
Applied Geology,
University of Erlangen

Muschelkalk:
Palaeontology,
Virtual Reality Group,
University of Erlangen

Applications: Archeology

Hellenic Statue of Isis
3rd century B.C.
ARTIS, University of Erlangen-Nuremberg, Germany

Sotades Pygmaios Statue,
5th century B.C
ARTIS, University of Erlangen-Nuremberg, Germany

Applications:

Material Science,
Quality Control

Micro CT, Compound Material,
Applied Geology Department,
University of Erlangen

Biology

biological sample of the soil,
CT,
Virtual Reality Group,
University of Erlangen
Applications
Computational Science and Engineering

 REAL-TIME VOLUME GRAPHICS
Christof Rezk Salama
Computer Graphics and Multimedia Group, University of Siegen, Germany

Applications: Computer Science
Visualization of Pseudo Random Numbers

Outline
Data Set 3D Rendering Classification

Transfer Functions (TFs)
Map data value \( f \) to color and opacity

Physical Model of Radiative Transfer
Increase
true emission

Decrease
true absorption

Ray Integration
How do we determine the radiant energy along the ray?
Physical model: emission and absorption, no scattering

Extinction \( \tau \)
Absorption \( \kappa \)
Without absorption all the initial radiant energy would reach the point \( s \).

\[
I(s) = I(s_0) e^{-\kappa(s-s_0) - \tau(s-s_0)}
\]

\( s \)
viewing ray
\( s_0 \)
initial intensity
\( \kappa \)
absorption coefficient
\( \tau \)
extinction coefficient
Ray Integration

How do we determine the radiant energy along the ray?

Physical model: emission and absorption, no scattering

\[ I(s) = I(s_0) e^{-\int_{s_0}^{s} \tau(s') ds'} + \int_{s_0}^{s} q(s') e^{-\int_{s'}^{s} \tau(s'') ds''} ds' \]

One point along the viewing ray emits additional radiant energy.

Numerical Solution

Approximate integral by Riemann sum:

\[ \tau(0, t) \approx \tau_s(0, t) = \sum_{i=1}^{n} \frac{\Delta t}{2} n_i(\Delta t) \Delta t \]

Now we introduce opacity:

\[ (1 - A_t) - e^{-\tau_i(s)} \]

Ray Casting

Software Solution

Numerical Solution

Data Set

Numerical Integration

Resampling

High Computational Load

Numerical Solution

\[ q(t) = \sum_{i=1}^{n} c_i(1 - A_t) \]

\[ c_i(1 - A_t) \]

\[ c_{i+1}(1 - A_t) \]

Radiant energy absorbed at position i
**Summary**

- **Emission Absorption Model**
  
  \[ I(s) = I(s_0) e^{-\mu(s) s} + \int_{s_0}^s g(\tilde{s}) e^{-\mu(\tilde{s}) \tilde{s}} d\tilde{s} \]

- **Numerical Solutions**

  \[
  C_i^r = C_i^t + \sum_j \left[ (1 - A_{ij}^t) C_j^r \right] \\
  A_i^r = A_i^t + \sum_j \left[ (1 - A_{ij}^t) A_j^r \right]
  \]

**Real-Time Volume Graphics**

[03] GPU-Based Volume Rendering

**Volume Rendering**

*Image order approach:*

For each slice:
- calculate contribution to the image

For each pixel:
- calculate color of the pixel

*Object order approach:*

For each pixel:
- calculate color of the pixel

**Texture-based Approaches**

- No volumetric hardware-primitives!
- Proxy geometry (Polygonal Slices)
How does a texture work?

For each fragment:
- interpolate the texture coordinates (barycentric)
- Texture-Lookup: interpolate the texture color (bilinear)

2D Textures
- Draw the volume as a stack of 2D textures
- Decomposition into axis-aligned slices

Implementation

Fragment Program

```c
// single 2D texture sampling
float data[2][2][3], texWV; // textured color; uniform complex<float> alias = COLOR
{
float result = tex2DAlIase, texNV; 
return result;
}
```

Compositing
Compositing

**Maximum Intensity Projection**

- No emission/absorption
- Simply compute maximum value along a ray

### 2D Textures: Drawbacks

- Sampling rate is inconsistent
- Emission/absorption slightly incorrect
- Super-sampling on-the-fly impossible

### 3D Textures

- **3D Texture**: Volumetric Texture Object
- Trilinear Interpolation in Hardware
- Slices parallel to the image plane
- One large texture block in memory

### Resampling via 3D Textures

- Sampling rate is constant
- Supersampling by increasing the number of slices
Bricking:

What happens if data set is too large to fit into local video memory?
- Divide the data set into smaller chunks (bricks)

One plane of voxels must be duplicated to enable correct interpolation across brick boundaries.

Problem: Bus-Bandwidth
- Keep the bricks small enough!
- More than one brick must fit into video memory!
- Transfer and Rendering can be performed in parallel
- Increased CPU load for intersection calculation!
- Effective load balancing still very difficult!

Cube-Slice Intersection

Question: Can we compute this in a vertex program?

Vertex program:
- Input: 6 Vertices
- Output: 6 Vertices

- \( P_0 \): Intersection with red path
- \( P_2 \): Intersection with green path
- \( P_4 \): Intersection with blue path
- \( P_1 \): Intersection with dotted red edge or \( P_0 \)
- \( P_3 \): Intersection with dotted green edge or \( P_1 \)
- \( P_5 \): Intersection with dotted blue edge or \( P_2 \)

Back to 2D Textures

- Number of object aligned slices
- Visual artifacts due to bilinear interpolation

Utilize Multi-Textures (2 textures per polygon) to implement trilinear interpolation!
Implementation

//vertex program for compositing object aligned slices
// make multiplanar viewing possible
main:

texture0 = tex2D(texture0, texUV); // 1st pass: draw

//texstore = tex2D(texture0, texUV); // 2nd pass: draw

//texstore = tex2D(texture0, texUV); // 3rd pass: draw

//texstore = tex2D(texture0, texUV); // 4th pass: draw

2D Multi-Textures

- Sampling rate is constant
- Supersampling by increasing the number of slices

Advantages

- More efficient load balancing

  - Exploit the GPU and the available memory bandwidth in parallel
  - Transfer the smallest amount of information required to draw the slice image!
  - Significantly higher performance, although 3 copies of the data set in main memory

Summary

Rasterization Approaches for Direct Volume Rendering

2D Texture Based Approaches

- 3-float stacks of object aligned slices
- Visual artifacts due to bilinear interpolation only

3D Texture Based Approaches

- Non-object aligned slices
- Supersampling with trilinear interpolation
- Stacking bus transfer inefficient for large volumes

2D Texture Based Approaches

- 3 variable stacks of object aligned slices
- Supersampling with trilinear interpolation
- Higher performance for large volumes
Talk Outline

Why use ray-casting instead of slicing?
- Ray-casting of rectilinear (structured) grids
  - Basic approaches on GPUs
  - Basic acceleration methods
  - Object-order empty space skipping
  - Volume ray-casting
  - Endoscopic ray-casting

Why Ray-Casting on GPUs?
- Most GPU rendering is object-order (rasterization)
- Image-order is more “CPU-like”
  - Recent fragment shader advances
  - Simpler to implement
  - Very flexible (e.g., adaptive sampling)
  - Correct perspective projection
  - Can be implemented in single pass!
  - Native 32-bit compositing

Where Is Correct Perspective Needed?
- Entering the volume
- Wide field of view
- Fly-throughs
- Virtual endoscopy
- Integration into perspective scenes, e.g., games

Recent GPU Ray-Casting Approaches
- Rectilinear grids
  - [Krüger and Westermann, 2003]
  - [Rötger et al., 2003]
  - [Green, 2004] (NVIDIA SDK Example)
  - [Stegmaier et al., 2005]
  - [Scharsach et al., 2006]
- Unstructured (tetrahedral) grids
  - [Bernardon, 2004]

Single-Pass Ray-Casting
- Enabled by conditional loops in fragment shaders (Shader Model 3; e.g., GeForce 6800, ATI X1800)
- Substitute multiple passes and early-z testing by single loop and early loop exit
- No compositing buffer: full 32-bit precision!

NVIDIA example: compute ray intersections with bounding box, march along rays and composite

Basic Ray Setup / Termination
- Two main approaches:
  - Procedural ray-box intersection
    - [Rötger et al., 2003], [Green, 2004]
  - Rasterize bounding box
    - [Krüger and Westermann, 2003]
- Some possibilities
  - Ray start position and exit check
  - Ray start position and exit position
  - Ray start position and direction vector
Procedural Ray Setup/Termination

- Everything handled in the fragment shader
- Procedural ray / bounding box intersection
- Ray is given by camera position and volume entry position
- Exit criterion needed
- Pro: simple and self-contained
- Con: full load on the fragment shader

Fragment Shader

- Rasterize front faces of volume bounding box
- Texcoords are volume position in [0,1]
- Subtract camera position
- Repeatedly check for exit of bounding box

"Image-Based" Ray Setup/Termination

- Rasterize bounding box front faces and back faces
  [Krüger and Westermann, 2003]
- Ray start position: front faces
- Direction vector: back–front faces
- Independent of projection (orthogonal/perspective)

Standard Ray-Casting Optimizations (1)

Early ray termination

- Isosurfaces: stop when surface hit
- Direct volume rendering: stop when opacity >= threshold

Several possibilities

- Older GPUs: multi-pass rendering with early-z test
- Shader model 3: break out of ray-casting loop
- Current GPUs: early loop exit not optimal but good

Standard Ray-Casting Optimizations (2)

Empty space skipping

- Skip transparent samples
- Depends on transfer function
- Start casting close to first hit

Several possibilities

- Pre-sample check of opacity (expensive)
- Traverse hierarchy (e.g., octree) or regular grid
- These are image-order. What about object-order?

Object-Order Empty Space Skipping (1)

- Modify initial rasterization step
- Rasterize bounding box
- Rasterize "tight" bounding geometry
Object-Order Empty Space Skipping (2)
- Store min-max values of volume bricks
- Cull bricks against isovalue or transfer function
- Rasterize front and back faces of active bricks

Object-Order Empty Space Skipping (3)
- Rasterize front and back faces of active min-max bricks
- Start rays on brick front faces
- Terminate when
  - Full opacity reached, or
  - Back face reached

Isosurface Ray-Casting
- Isosurfaces/Level Sets
  - Scanned data
  - Distance fields
  - CSG operations
  - Level sets: surface editing, simulation, segmentation, ...

Intersection Refinement (1)
- Fixed number of bisection or binary search steps
- Virtually no impact on performance
- Refine already detected intersection
- Handle problems with small features / at silhouettes with adaptive sampling

Intersection Refinement (2)
- sampling rate 1/5 voxel (no adaptive sampling)
Intersection Refinement (3)

Deferred Isosurface Shading

- Shading is expensive
- Gradient computation; conditional execution not free
- Ray-casting step computes only intersection image

Enhancements (1)

- Build on image-based ray setup
- Allow viewpoint inside the volume
- Intersect polygonal geometry

Enhancements (2)

1. Starting position computation
   - Ray start position image
2. Ray length computation
   - Ray length image
3. Render polygonal geometry
   - Modified ray length image
4. Raycasting
   - Compositing buffer
5. Blending
   - Final image

Moving Into The Volume (1)

- Near clipping plane clips into front faces
- Fill in holes with near clipping plane
- Can use depth buffer [Scharsach et al., 2006]

Moving Into The Volume (2)

1. Rasterize near clipping plane
   - Disable depth buffer, enable color buffer
   - Rasterize entire near clipping plane
2. Rasterize nearest back faces
   - Enable depth buffer, disable color buffer
   - Rasterize nearest back faces of active bricks
3. Rasterize nearest front faces
   - Enable depth buffer, enable color buffer
   - Rasterize nearest front faces of active bricks
**Virtual Endoscopy**
- Viewpoint inside the volume with wide field of view
  - E.g.: virtual colonoscopy
- Hybrid isosurface rendering / direct volume rendering
  - E.g.: colon wall and structures behind

**Virtual Colonoscopy**
- First find isosurface; then continue with DVR

**Virtual Colonoscopy**
- First find isosurface; then continue with DVR

**Hybrid Ray-Casting (1)**
- Isosurface rendering
  - Find isosurface first
  - Semi-transparent shading provides surface information
- Additional unshaded DVR
  - Render volume behind the surface with unshaded DVR
  - Isosurface is starting position
  - Start with (1.0-iso_opacity)

**Hybrid Ray-Casting (2)**
- Hiding sampling artifacts (similar to interleaved sampling, [Heidrich and Keller, 2001])

**Conclusions**
- GPU ray-casting is an attractive alternative
- Very flexible and easy to implement
- Fragment shader conditionals are very powerful; performance pitfalls very likely to go away
- Mixing image-order and object-order well suited to GPUs (vertex and fragment processing?)
- Deferred shading allows complex filtering and shading at high frame rates
Thank You!

Acknowledgments
- Hunting Schersch, Christian Sigg, Daniel Weiskopf
- VRVis is funded by the Kplus program of the Austrian government

Tetrahedral Grids
- Traditional (rasterization): Projected Tetrahedra
- Ray casting: store mesh in textures

Ray/face intersection computations
- Pre-integration, (store current pos in texture)

Propagate from cell to cell

Tetrahedral Grids
- df

Real-Time Volume Graphics
[05] Transfer Functions

Classification
- During Classification the user defines the „Look“ of the data.
  - Which parts are transparent?
  - Which parts have which color?
Classification

During Classification the user defines the "Look" of the data.
- Which parts are transparent?
- Which parts have which color?
- The user defines a Transfer Function.

Classification

REAL-TIME VOLUME GRAPHICS
Christof Rezk Salama
Computer Graphics and Multimedia Group, University of Siegen, Germany
Eurographics 2006

Real-Time update of the transfer function necessary!!!

Classification

PRE-INTERPOLATIVE

POST-INTERPOLATIVE

Pre-Classification

Pre-Classification:

Color table is applied before interpolation. (pre-interpolative Transferfunction)

A color value is fetched from a table for each Voxel
A RGBA Value is determined for each Voxel

Possible Implementations

The naive Approach:

Save Emission and Absorption terms directly in the Texture.
Possible Implementations

**The naive Approach:**
Save Emission and Absorption terms directly in the Texture.

- Very high memory consumption
  - Main Memory (RGBA and scalar volumes)
  - Graphics Memory (RGBA volume)
- High Load on memory bus
  - RGBA Volume must be transferred.
- Upload necessary on TF change

Possible Implementations

**A better Approach:**
Apply color table during texture transfers from main memory to graphics card (standard OpenGL feature)

- High memory consumption
  - Main Memory (only scalar volume)
  - Graphics Memory (RGBA volume)
- Reduced load on memory bus
  - Only the scalar volume is transferred.
- Upload necessary on TF change

Possible Implementations

**The best approach:** Paletted Textures
Store the scalar volume together with the color table directly in graphics memory.

- Hardware-Support necessary!

Pre-Classification Summary

**Summary Pre-Classification**

- Application of the Transferfunction before Rasterization
- One RGBA Lookup for each Voxel
- Different Implementations:
  - Texture Transfer
  - Texture Color Tables (paletted textures)
- Simple and Efficient
- Good for coloring segmented data
Post-Classification

- Geometry Processing
- Rasterization
- Transfer Function

A color is fetched from the color table for each Fragment

Post-Classification

The color table is applied after interpolation (post-interpolative Transfer Function).

Texture 0 — Scalar field

\[ R = G = B = A = S \]

Texture 1 — Transfer Function [Emission RGB, Absorption A]

\[ RGBA = T(S) \]

CG Implementation

```cpp
//fragment program for post-classification
//using 3D textures
float4 main (float3 texUV : TEXCOORD0,
uniform sampler3D volume_texture,
uniform sampler1D transfer_function) :
COLOR
{
    float index = tex3D(volume_texture, texUV);
    float4 result = tex1D(transfer_function, index);
    return result;
}
```

Quality: Pre- vs. Post-Classification

- Comparison of image quality

<table>
<thead>
<tr>
<th>Pre-Classification</th>
<th>Post-Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same TF, same Resolution, same Sampling Rate</td>
<td></td>
</tr>
</tbody>
</table>

Pre- vs Post-Classification

- Functionality data
- Transfer Function
- Supersampling

Pre-interpolative TF

Analytical Solution

Post-interpolative TF
Post- vs Pre-Integrated Classification

Pre-Integrated Classification

Classification Artifacts / Pre-integration

Classification Artifacts / Pre-integration

Pre-integrate all possible combinations in the TF

Assume constant sampling distance $d$

Fast re-computation of the pre-integration table when transfer function changes

Use integral functions

When to use which Classification

- **Pre-Interpolative Classification**
  - If the graphics hardware does not support fragment shaders
  - For simple segmented volume data visualization

- **Post-Interpolative Classification**
  - If the transfer function is "smooth"
  - For good quality and good performance (especially when slicing)

- **Pre-Integrated Classification**
  - If the transfer function contains high frequencies
  - For best quality

**REAL-TIME VOLUME GRAPHICS**

Christof Rezk Salama
Computer Graphics and Multimedia Group, University of Siegen, Germany

Eurographics 2006

For good quality and good performance (especially when slicing)