Reading Assignment #8 (until Nov. 5)

Read (required):

• Real-Time Volume Graphics, remainder of Chapter 4 (Sec. 4.5-)

• Real-Time Volume Graphics, parts of Chapter 10:
  Secs. 10.1, 10.2, 10.3, 10.6
**Image order approach:**

*For each pixel*

*calculate color of the pixel*
**Object order approach:**

For each slice {
    calculate contribution to the image
}
Basic Volume Rendering Summary

Volume rendering integral for *Emission Absorption* model

\[
I(s) = I(s_0) e^{\tau(s_0, s)} + \int_{s_0}^{s} q(\tilde{s}) e^{\tau(s_0, \tilde{s})} d\tilde{s}
\]

Numerical solutions: **Back-to-front** vs. **Front-to-back compositing**

\[
C_i' = C_i + (1 - A_i) C_{i-1}' \\
C_i' = C_{i+1}' + (1 - A_{i+1}') C_i \\
A_i' = A_{i+1}' + (1 - A_{i+1}') A_i
\]

Approaches: **Image order** vs. **Object order**
Ray-Casting vs. Isosurface Ray-Casting

Ray-Casting

- Ray setup
- Loop over ray
  - Sample scalar field
  - Classification
  - Shading
  - Compositing

Isosurface Ray-Casting

- Ray setup
- Loop over ray
  - Sample scalar field
  - if value >= isoValue (i.e., first hit)
    - break out of the loop
  - [Refine first hit location] (optional)
  - Shading
  - (Compositing not needed)
GPU-Based Ray-Casting
Ray-Casting

• “Natural” volume rendering method
• Image-order approach
  – Most common CPU approach
  – Well-supported by current GPUs!
• Standard optimizations
  – Early ray termination
  – Empty space skipping (empty space leaping)
• Many possibilities
  – Adaptive sampling
  – Realistic lighting
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 a single rendering pass!
  • Native 32-bit float compositing
Recent GPU Ray-Casting Approaches

Regular grids

- [Krüger and Westermann, 2003], [Röttger et al., 2003]
- [Green, 2004] (in NVIDIA SDK)
- [Stegmaier et al., 2005]
- [Scharsach et al., 2006]
- [Gobbetti et al., 2008]

Unstructured (tetrahedral, ...) grids

- [Bernardon et al., 2004]
- [Callahan et al., 2006]
- [Muigg et al., 2007]
Single-Pass Ray Casting

- Enabled by conditional loops in fragment shaders (Shader Model 3.0 and higher / NVIDIA CUDA)
- Substitute multiple passes and early-z testing by single loop and early loop exit
- Volume rendering example in NVIDIA CUDA SDK
Implementation

Ray setup
Loop over ray
  Sample scalar field
  Classification
  Shading
  Compositing
Implementation

Ray setup

Loop over ray

Sample scalar field
Classification
Shading
Compositing
Ray Setup

Two main approaches:

- Procedural ray/box intersection
  [Röttger 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 / CUDA kernel
• 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 computational load per-pixel/fragment
Rasterization-Based Ray Setup

- Fragment == ray
- Need ray start pos, direction vector
- Rasterize bounding box

- Identical for orthogonal and perspective projection!
Implementation

Ray setup

Loop over ray

Sample scalar field

Classification

Shading

Compositing
Classification – Transfer Functions

During Classification the user defines the “look“ of the data.

- Which parts are transparent?
- Which parts have what color?
During Classification the user defines the "look" of the data.

• Which parts are transparent?
• Which parts have what color?

The user defines a transfer function.
Transfer Function Application

texture = scalar field

\( (s, t) \)

resampling

scalar value \( S \)

\( T(S) \)

RGBA

transfer function texture = [Emission RGB, Absorption A]
// Cg 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;
}

Windowing Transfer Function

Map input scalar range to output intensity range

• Select scalar range of interest
• Adjust contrast
Implementation

Ray setup

Loop over ray

- Sample scalar field
- Classification
- **Shading**
- Compositing
Gradient Reconstruction

Central differences

- Cheap and quality often sufficient (2+2+2 neighbors in 3D)

Discrete convolution filters on grid

- Image processing filters; e.g. Sobel (3x3x3 neighbors)

Continuous convolution filters

- Derived continuous reconstruction filters
- E.g., the cubic B-spline and its derivatives (4x4x4 neighbors)
On-the-fly Gradient Estimation

\[ \nabla f(x, y, z) \approx \frac{1}{2h} \begin{pmatrix} f(x + h, y, z) - f(x - h, y, z) \\ f(x, y + h, z) - f(x, y - h, z) \\ f(x, y, z + h) - f(x, y, z - h) \end{pmatrix} \]

```c
float3 sample1, sample2;
// six texture samples for the gradient
sample1.x = tex3D(texture, uvw-half3(Delta, 0.0, 0.0)).x;
sample2.x = tex3D(texture, uvw-half3(Delta, 0.0, 0.0)).x;
sample1.y = tex3D(texture, uvw-half3(0.0, Delta, 0.0)).x;
sample2.y = tex3D(texture, uvw-half3(0.0, Delta, 0.0)).x;
sample1.z = tex3D(texture, uvw-half3(0.0, 0.0, Delta)).x;
sample2.z = tex3D(texture, uvw-half3(0.0, 0.0, Delta)).x;
// central difference and normalization
float3 N = normalize(sample2-sample1);
```
Pre-compute gradients at grid points with any method
Store normalized gradient directions in RGB texture

Sample gradient texture in fragment shader: interpolation
Re-normalize after fetch!

\[
\begin{pmatrix}
  n_X \\
  n_Y \\
  n_Z
\end{pmatrix}
\rightarrow
\begin{pmatrix}
  R \\
  G \\
  B
\end{pmatrix}
\]

RGB gradient texture  \(\text{lerp of texture filter}\)  renormalize!
On-The-Fly Gradients

Reduce texture memory consumption!

Central differences before and after linear interpolation of values at grid points yield the same results

*Caveat:* texture filter precision

Filter kernel methods are expensive, but:

Tri-cubic B-spline kernels can be used in real-time (e.g., GPU Gems 2 Chapter “Fast Third-Order Filtering”)
Ray setup

Loop over ray

Sample scalar field
Classification
Shading

Compositing

\[
\begin{align*}
C_i'' & = C_{i+1}'' + (1 - A_{i+1}') C_i' \\
A_i' & = A_{i+1}' + (1 - A_{i+1}') A_i
\end{align*}
\]
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

```cpp
// Cg fragment shader code for single-pass ray casting
float4 main(VS_OUTPUT IN, float4 TexCoord0 : TEXCOORD0,
uniform sampler3D SamplerDataVolume,
uniform sampler1D SamplerTransferFunction,
uniform float3 camera,
uniform float stepsize,
uniform float3 volExtentMin,
uniform float3 volExtentMax
): COLOR
{
    float4 value;
    float scalar;
    // Initialize accumulated color and opacity
    float4 dst = float4(0,0,0,0);
    // Determine volume entry position
    float3 position = TexCoord0.xyz;
    // Compute ray direction
    float3 direction = TexCoord0.xyz - camera;
    direction = normalize(direction);
    // Loop for ray traversal
    for (int i = 0; i < 200; i++) // Some large number
    {
        // Data access to scalar value in 3D volume texture
        value = tex3D(SamplerDataVolume, position);
        scalar = value.a;
        // Apply transfer function
        float4 src = tex1D(SamplerTransferFunction, scalar);
        // Front-to-back compositing
        dst = (1.0-dst.a) * src + dst;
        // Advance ray position along ray direction
        position = position + direction * stepsize;
        // Ray termination: Test if outside volume ...
        float3 temp1 = sign(position - volExtentMin);
        float3 temp2 = sign(volExtentMax - position);
        float inside = dot(temp1, temp2);
        // ... and exit loop
        if (inside < 3.0)
            break;
    }
    return dst;
}
```
• Image-based ray setup
  – Ray start image
  – Direction image

• Ray-cast loop
  – Sample volume
  – Accumulate color and opacity

• Terminate
• Store output

CUDA Kernel

```c
__global__
void RayCastCUDAKernel( float *d_output_buffer, float *d_startpos_buffer, float *d_direction_buffer )
{
    // output pixel coordinates
dword screencoord_x = _umul24( blockIdx.x, blockDim.x ) + threadIdx.x;
dword screencoord_y = _umul24( blockIdx.y, blockDim.y ) + threadIdx.y;

    // target pixel [RGBA-tuple] index
dword screencoord_index = ( _umul24( screencoord_y, cu_screensize.x ) + screencoord_x ) * 4;

    // get direction vector and ray start
    float4 dir_vec = d_direction_buffer[ screencoord_index ];
    float4 startpos = d_startpos_buffer[ screencoord_index ];

    // ray-casting loop
    float4 color = make_float4( 0.0f );
    float pscout = 0.0f;
    for ( int i = 0; i < 64; i++ )
    {
        // next sample position in volume space
        float3 samplepos = dir_vec * pscout + startpos;
        pscout += cu_sampling_distance;

        // fetch density
        float tex_density = tex3d( cu_volume_texture, samplepos.x, samplepos.y, samplepos.z );

        // apply transfer function
        float4 col_classified = tex4d( cu_transfer_function_texture, tex_density );

        // compute |1-preAlpha|*col.a
        float prev_alpha = -color.w * col_classified.w + col_classified.w;

        // composite color and alpha
        color.xyz = prev_alpha * col_classified.xyz + color.xyz;
        color.w += prev_alpha;

        // break if ray terminates (behind exit position or alpha threshold reached)
        if ( ( pscout > dir_vec.x ) || ( color.w > 0.9f ) ) { break; }
    }

    // store output color and opacity
    d_output_buffer[ screencoord_index ] = _saturatef( color );
}
```
Isosurface Ray-Casting

Isosurfaces/Level Sets

• Scanned data
• Distance fields
• CSG operations
Opaque isosurfaces:
only one sample contributes per ray/pixel

Discard all samples except first hit on isosurface / object boundary

\[ f(x) \geq f_{iso} \]

\[ f(x) < f_{iso} \]

Threshold transfer function / alpha test
First hit ray casting
Implementation - Isosurface Ray-Casting

Ray setup
Loop over ray
  Sample scalar field
  Classification
    if sample is opaque (i.e., first hit)
      break out of the loop
  Refine first hit location
Shading
(Compositing not needed)
Thank you.

Thanks for material

• Helwig Hauser
• Eduard Gröller
• Daniel Weiskopf
• Torsten Möller
• Ronny Peikert
• Philipp Muigg
• Christof Rezk-Salama