CSE 332
Intro to Visualization

Visualizing Volumetric Data

Klaus Mueller

Computer Science Department
Stony Brook University
Volume Data Generation

Often obtained by scanning

- for example, X-ray CT
Volume Data – 2D Slice View
Which do you prefer: 2D or 3D
RAYCASTING CONCEPT

Image Plane

Eye

Data Set

Numerical Integration

Resampling
Sampling Along the Ray

Estimate sample values via interpolation
Sampling via Trilinear Interpolation

\[ f_v = f_1(1-p)(1-q)(1-r) + f_2(p)(1-q)(1-r) + f_3(p)(q)(1-r) + f_4(1-p)(q)(1-r) + f_5(1-p)(1-q)(r) + f_6(p)(1-q)(r) + f_7(p)(q)(r) + f_8(1-p)(q)(r) \]
Here is what it looks like in 2D for bi-linear interpolation.
We learned about RGB

There is one more channel – opacity (A)

- gives RGBA color
- opacity (A) = 1 – transparency (T)
- range [0.0 ... 1.0]

Opacity (A) multiplied by RGB creates a weighting effect
Opacity and Color Blending

\[ C_{mix} = C_{back} A_{back} (1 - A_{front}) + C_{front} A_{front} \]

\[ C_{mix} = C_R A_R (1 - A_B) + C_B A_B \]

\[ T_R = 0.00, \ A_R = 1.00 \]

\[ C = R \cdot 0.75 + B \cdot 0.25 \]

\[ T_B = 0.75 \]

\[ A_B = 0.25 \]
**Compositing – Merging the Samples**

### Back-to-front rendering

\[ C'_i = C_i A_i + (1 - A_i) C'_{i-1} \]

### Front-to-back rendering

\[ C'_i = C'_{i-1} + (1 - A'_{i-1}) C_i A_i \]
\[ A'_i = A'_{i-1} + (1 - A'_{i-1}) A_i \]

**A:** Opacity = 1 - Transparency = 1 - T

**C:** Color
Transfer Function

Determines what color & opacity a sample value should have

- input: an interpolated density value
- output: a color and opacity (RGBA)
**RAYCASTING SPECIFICS**

A point $P$ on a ray is given by:

$$P = \text{Eye} + t \cdot r_{i,j}$$

$t$: parametric variable

Spacing of pixels on image plane:

$$\Delta i = \frac{W}{N_i - 1} \quad \Delta j = \frac{H}{N_j - 1}$$

$N_i, N_j$: image dims. in pixels

A ray is specified by:

- eye position (Eye)
- screen pixel location $P_{i,j}$

$\rightarrow$ ray direction vector $(r_{i,j})$ of unit length

$$r_{i,j} = \frac{P_{i,j} - \text{Eye}}{|P_{i,j} - \text{Eye}|}$$

**Image-order projection:**

- scan the image row by row, column by column:

$$P_{i,j} = P_{0,0} + i \cdot v \cdot \Delta j + j \cdot u \cdot \Delta i$$

- $P_{i,j}$: Location of image pixel $(i, j)$ in world space

- $0 \leq i < N_i \quad 0 \leq j < N_j$

- $P_{0,0}$: image (=screen) origin in world space

- $u, v, n$: orthonormal image plane vectors ($n = v \times u$)
**Volume Rendering Modes**

**X-ray:**
rays sum volume contributions along their linear paths

**Iso-surface:**
rays look for the object surfaces, defined by a certain volume value

**Maximum Intensity Projection (MIP):**
a pixel value stores the largest volume value along its ray

**Full volume rendering:**
rays composite volume contributions along their linear paths
PRACTICAL IMPLEMENTATION

- 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
GPU Program

- 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 stepszie,
    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 * stepszie;
        // 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;
}
```
Why is front-to-back rendering better?

- early ray termination – terminate a ray when $A > 0.90$
- empty-space skipping – jump across empty space quickly
ISO-SURFACE RENDERING

• A closed surface separates ‘outside’ from ‘inside’ (Jordan theorem)
• In iso-surface rendering we say that all voxels with values > some threshold are ‘inside’, and the others are ‘outside’
• The boundary between ‘outside’ and ‘inside’ is the iso-surface
• All voxels near the iso-surface have a value close to the iso-threshold or iso-value
• Example:

  cross-section of a smooth sphere

  iso-boundary

  iso-value = 50
  will render a large sphere

  iso-value = 200
  will render a small sphere
Iso-Surface Rendering

iso-value = 30

iso-value = 80

iso-value = 200
To render an iso-surface we cast the rays as usual...

but we stop, once we have interpolated a value \( \text{iso-threshold} \)

We would like to illuminate (shade) the iso-surface based on its orientation to the light source.

Recall that we need a normal vector for shading.

The normal vector \( \mathbf{N} \) is the local gradient, normalized.
The gradient vector $\mathbf{g} = (g_x, g_y, g_z)^T$ at the sample position $(x, y, z)$ is usually computed via central-differencing (for example, $g_x$ is the volume density gradient in the $x$-direction):

\[
\begin{align*}
g_x &= \frac{f(x-1, y, z) - f(x+1, y, z)}{2} \\
g_y &= \frac{f(x, y-1, z) - f(x, y+1, z)}{2} \\
g_z &= \frac{f(x, y, z-1) - f(x, y, z+1)}{2}
\end{align*}
\]}
SHADING THE ISO-SURFACE

- The normal vector is the normalized gradient vector \( g \)
  \[ N = \frac{g}{|g|} \] (normal vector always has unit length)
- Once the normal vector has been calculated we shade the iso-surface at the sample point
- The color so obtained is then written to the pixel that is due to the ray

The color is calculated with the standard shading equation:
\[
C = C_{obj} (k_a I_A + k_d I_L N \cdot L) + k_s I_L (H \cdot N)^{ns}
\]

\( C_{obj} \) is obtained by indexing the color transfer function with the interpolated sample value
When hitting a surface set $A < 1.0$

- ray marches on
- inner structures can be seen
During Classification the user defines the "Look" of the data.

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

The user defines a Transferfunction.
Classification
Classification
Classification
Classification
Classification

Real-Time update of the transfer function necessary!!!
Classification
Transfer Functions: Multi-Dimensional

Boundaries in volume create arches in (value,gradient) domain [Kindlmann 98]

Arches guide placement of opacity to emphasize material interfaces [Kniss 01]
Transfer Functions: Multi-Dimensional

- Boundaries can be described in terms of:
  - maximum in 1st derivative
  - zero-crossing in 2nd derivative
- Semi-automatic classification possible in clean data
Transfer Functions: Multi-Dimensional

Dual-domain interaction:

[Kniss 01]

New Rendering

Changes to transfer function

Make features opaque by pointing at them

Actions in spatial domain

New transfer function
Multi-Dimensional Transfer Functions
Multi-Dimensional Transfer Functions
A single slider bar is most appreciated [Rezk-Salama Vis06]

Enables doctors to quickly fine-tune the transfer function for specific objects

- works since in CT usually only small deviations exist
- but these require complex interactions in the transfer function domain
Typical transfer function parameterization:

Datasets typically only deviate modestly from this

• but in complex ways
• meaning, lots of tweaking is required

[Rezk-Salama Vis06]
Parameter Mapping Approach (2)

We can learn these deviations by observing a few datasets

• encode the parameters into an N-D vector
• find the principal component of the vectors (the main Eigenvector)
• project all other vectors onto this Eigenvector
• the min and max then represent the min and max of the slider

[Rezk-Salama Vis06]