CS 519: Scientific Visualization

Volume Visualization

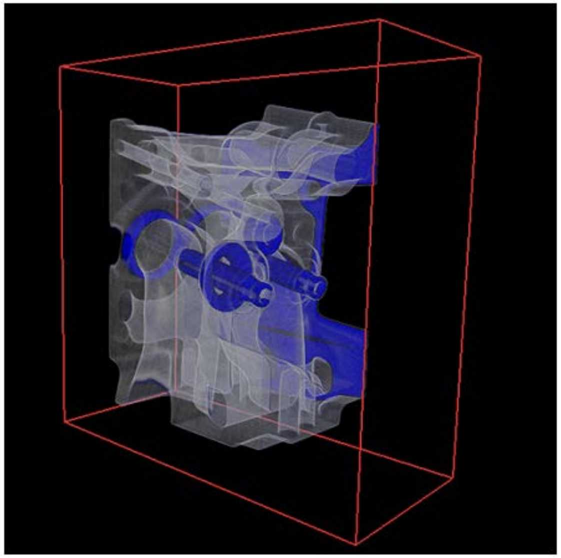
Eric Shaffer

Some slides adapted Alexandru Telea, *Data Visualization Principles and Practice*

Some slides adapted from Travis Gorkin, *Volume Rendering using Graphics Hardware*

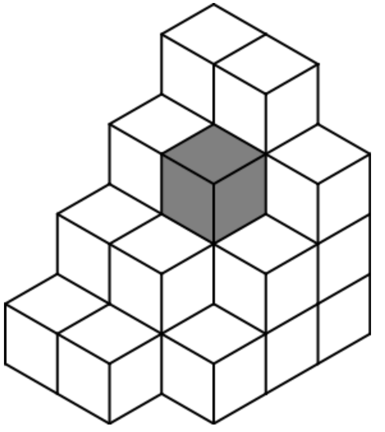
Volume Rendering Definition

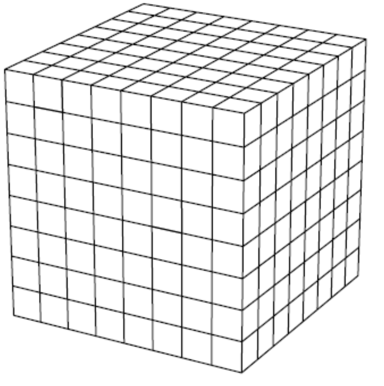
- Generate 2D projection of 3D data set
- Visualization of medical and scientific data
- Rendering natural effects - fluids, smoke, fire
- Direct Volume Rendering (DVR)
  - Done without extracting any surface geometry



Volumetric Data

- 3D Data Set
  - Discretely sampled on regular grid in 3D space
  - 3D array of samples
- Voxel – volume element
  - One or more constant data values
    - Scalars – density, temperature, opacity
    - Vectors – color, normal, gradient
  - Spatial coordinates determined by position in the grid





8x8x8 resolution orthogonal (uniform) voxel grid

Volume Visualization

1. **Motivation**
   - how to see through 3D scalar volumes?

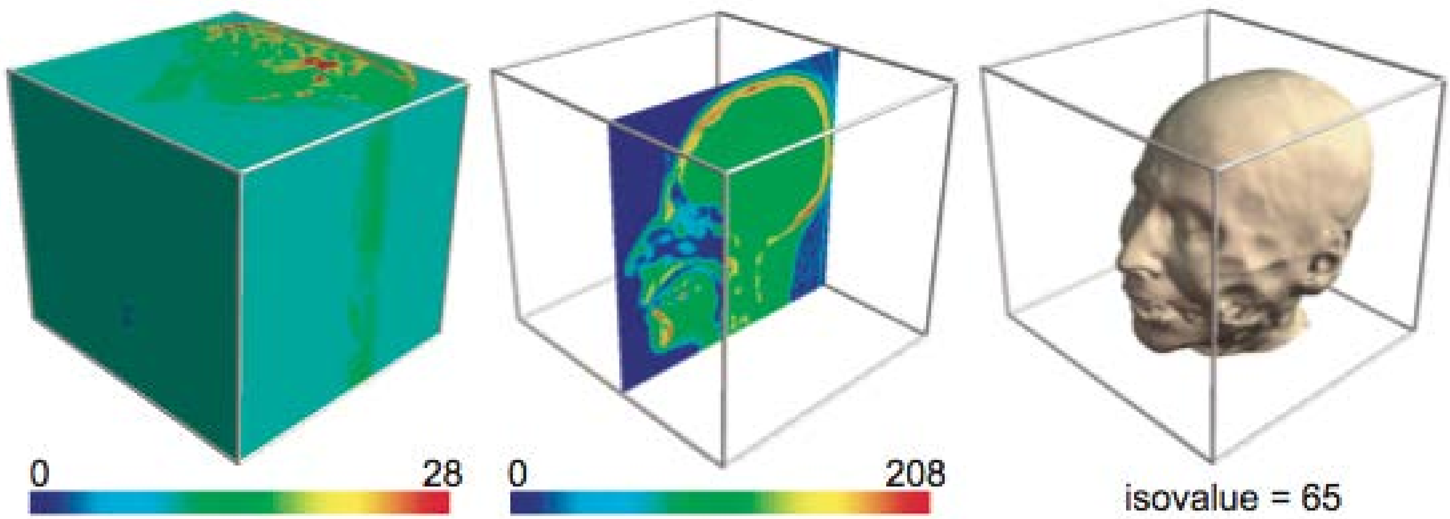
2. **Methods and techniques**
   - ray function (MIP, average intensity, distance to value, isosurface)
   - classification
   - compositing
   - volumetric shading

3. **Advanced points**
   - sampling and interpolation
   - classification and interpolation order
   - performance issues

Volume Visualization

Scalar volume $s : \mathbf{R}^3 \rightarrow \mathbf{R}$

How to visualize this?



direct color mapping

- see only outer surface

slicing

- all details on slice
- no info outside slice

contouring

- all details on contour
- no info outside contour

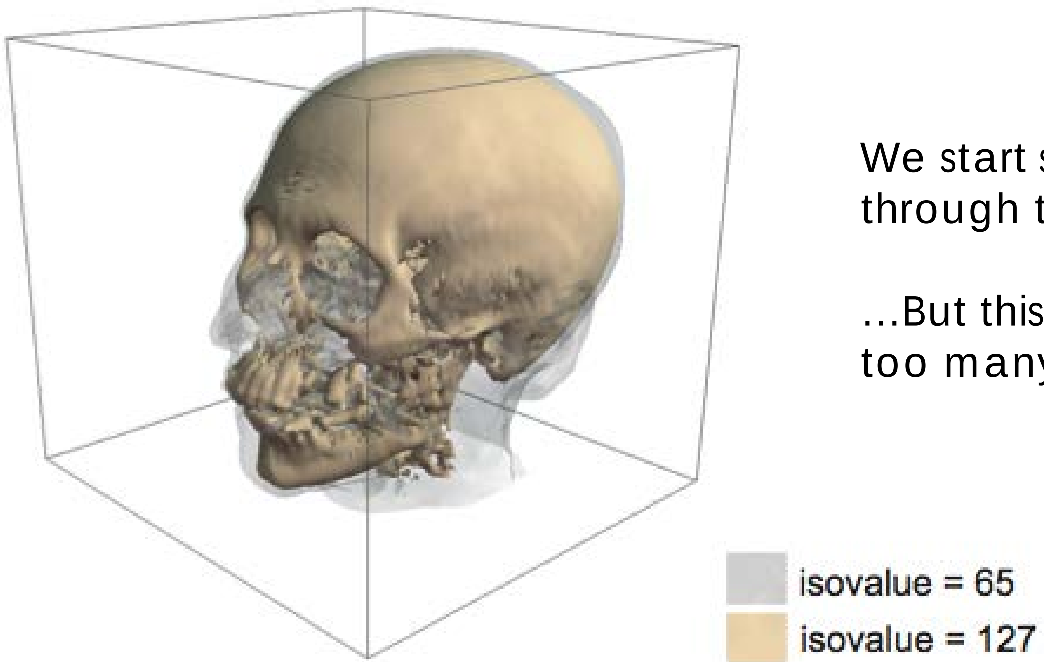
How to visualize this so we see through the volume

Idea

- use known techniques (slices and contours)
- use transparency

First try

- draw several contours C_i for several values s_i
- transparency α_i proportional to scalar value s

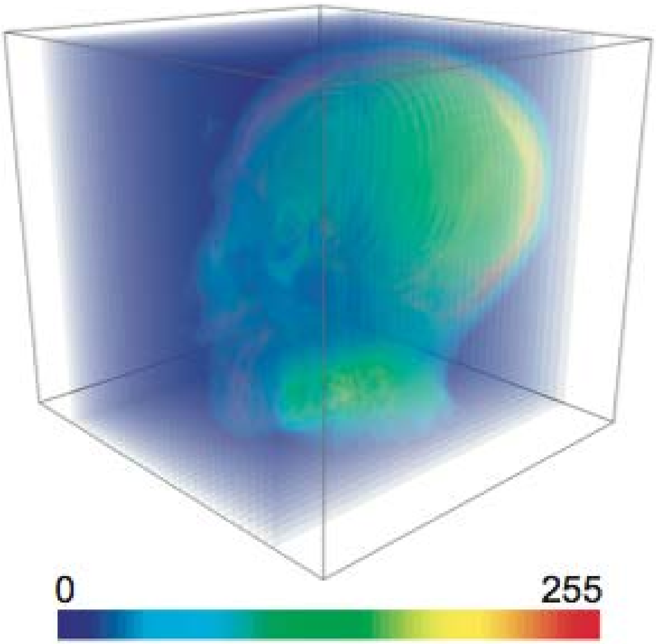


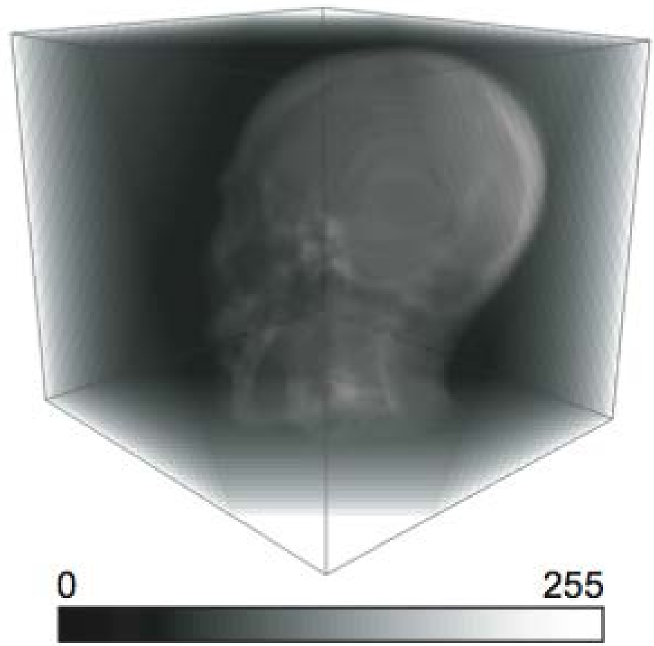
We start seeing a little bit through the volume…

…But this won't work for too many contours

Second try

- draw several parallel slices S_i
- transparency α_i inversely proportional to number of slices $\alpha_i = \dfrac{1}{\|S\|}$





axis-aligned slices

- not OK if we view volume across slicing direction

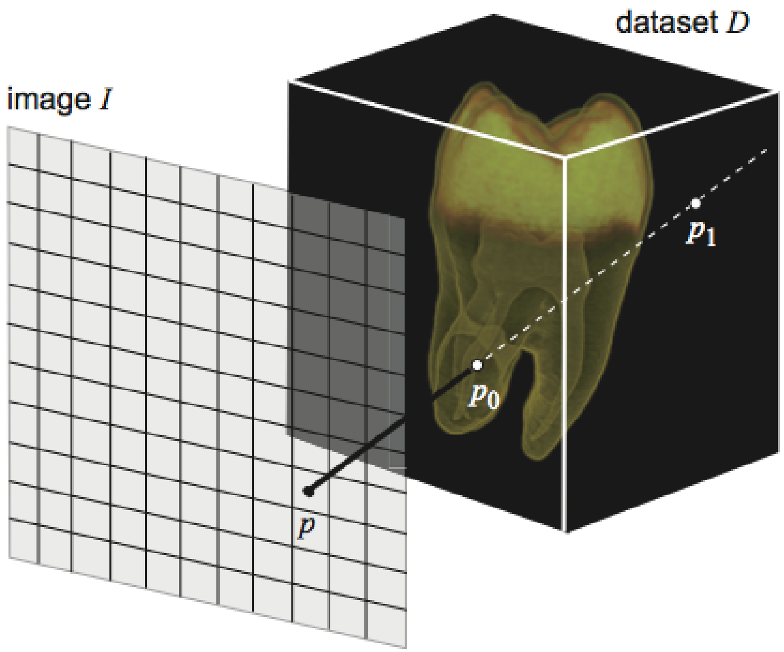
view direction-aligned slices

- any viewing direction OK
- must reslice when changing viewpoint

Volume Rendering Basics

Main idea

- consider a scalar signal $s : D \to \mathbf{R}$ to be drawn on the screen image I
- for each pixel $p \in I$
  - construct a ray $\mathbf{r}$ orthogonal to I passing through p
  - compute intersection points p_0 and p_1 of $\mathbf{r}$ with D
  - express $I(p)$ as function of s along $\mathbf{r}$ between p_0 and p_1



1. Parameterize ray

$$p(t) = (1-t)p_0 + tp_1, \quad t \in [0,1]$$

1. Compute pixel color

$$I(p) = f(F(s(t)), \qquad t \in [0,1]$$

ray function

transfer function

Volume Rendering

Define a ray function $F : \{s(t) \mid t \in [0,1]\} \to \mathbf{R}$

all scalar values along ray

a single resulting scalar value

The ray function 'aggregates' all scalar values along a ray

Next, define a transfer function $f : \mathbf{R} \to [0,1]^4$

a single scalar value

an RGBA color

- same concept as color mapping (see Module 2)

Idea

- ray function: says how to combine all scalar values along a ray into a single value
- transfer function: says how to map a single scalar value to a color
- The process of computing all rays for an image I is called ray casting

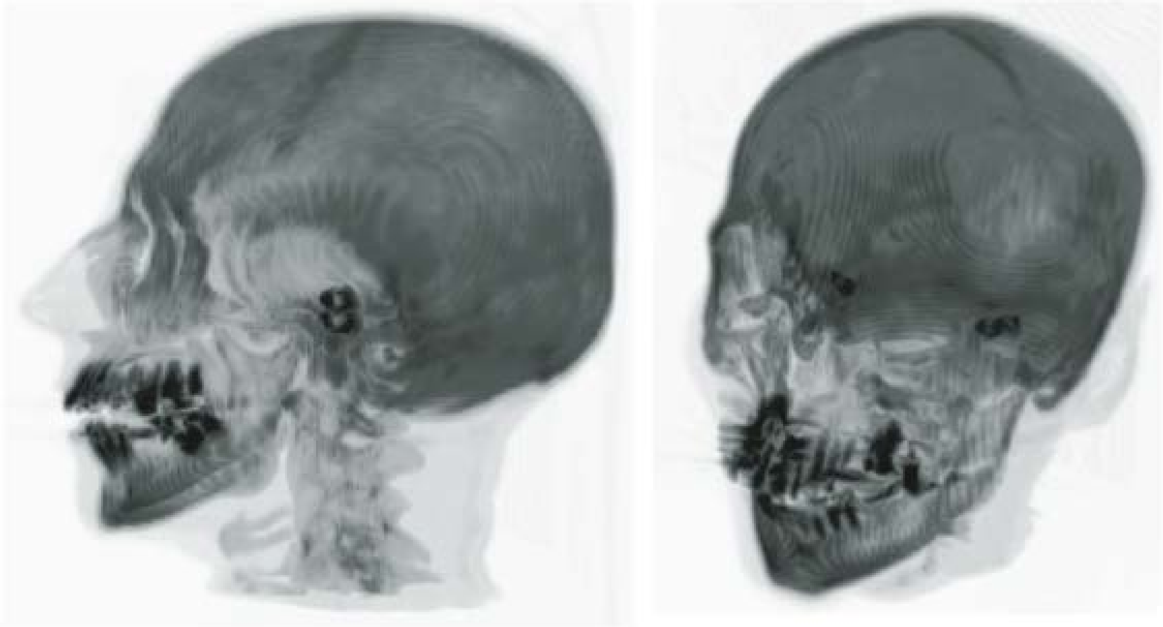
Maximum Intensity Projection (MIP)

First example of ray function

- find maximum scalar along ray, then apply transfer function to its value

$$I(p) = f(\max_{t \in [0,T]} s(t))$$

- useful to emphasize high-value points in the volume



Example
MIP of human head CT

- white = low density (air)
- black = high density (bone)

OK, but gives no depth cues

Average Intensity Projection

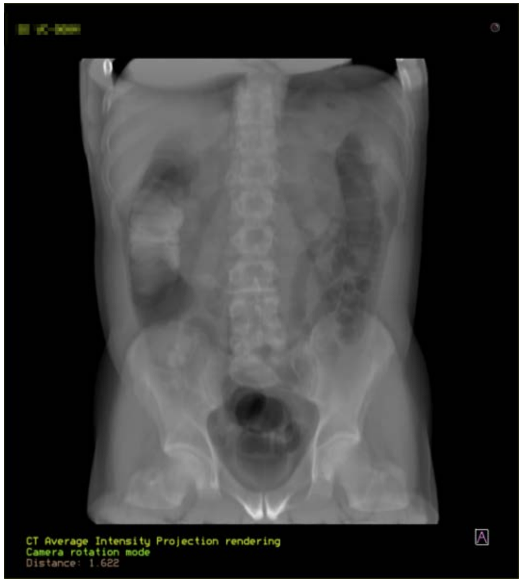
Second example of ray function

- compute average scalar along ray, then map it to color

$$I(p) = f\left(\frac{\int_{t=0}^{T} s(t)dt}{T}\right)$$

- useful to emphasize average tissue type (e.g. density in a CT scan)

maximum intensity projection



average intensity projection

Example
Human torso CT

- black = low density (air)
- white = high density (bone)

Average intensity projection is equivalent to an X-ray

Distance to Value Function

Third example of ray function

- compute distance along ray until a specific scalar value σ

$$I(p) = f\left(\min_{t \in [0,T], s(t) \geq \sigma} t\right)$$

- useful to emphasize depth where some specific tissue is located

distance to value
20

distance to value
50

Example
Human head CT

- black = low distance
- white = high distance

Isosurface Function

Fourth example of ray function

- compute whether a given isovalue σ exists along ray

$$I(p) = \begin{cases} f(\sigma), & \exists t \in [0,T], s(t) = \sigma \\ I_0, & \text{otherwise.} \end{cases}$$

- produces same result as marching cubes, but with a higher accuracy

isosurface (marching cubes)

isosurface (software ray casting)

isosurface (hardware ray casting)

Composite Function

Fifth example of ray function

- compute a color at each point along the ray (apply transfer function *first*)
- blend (compose) all colors to get the final pixel color (ray function=alpha blending)

$$I(p) = F(\{f(s(t)) \mid t \in [0,1]\})$$

transfer function (applied to all pixels along ray)

ray function (blends all colors produced by transfer function along ray)

Idea

- transfer function: controls color+transparency of all material types
- ray function: blends together all material colors+transparencies along ray
- most powerful (but most computationally expensive) ray function
- allows huge range of effects (depending on type of transfer function)
- designing 'good' transfer functions is however non-trivial

Compositing Color Samples

- Over operator – back-to-front order

$$\hat{C}_i = C_i + (1 - A_i)\hat{C}_{i+1}$$

$$\hat{A}_i = A_i + (1 - A_i)\hat{A}_{i+1}$$

- Under operator – front-to-back order

$$\hat{C}_i = (1 - \hat{A}_{i-1})C_i + \hat{C}_{i-1}$$

$$\hat{A}_i = (1 - \hat{A}_{i-1})A_i + \hat{A}_{i-1}$$

Volumetric Shading

Shading

- is required if we compute e.g. isosurfaces
- but can also be useful for composite ray function

Method

- instead of simply using the colors $I(t) = f(s(t))$
- composite the shaded colors (see Chapter 2, Phong lighting)

$$I(t) = c_{\text{amb}} + c_{\text{diff}}(t)\max(-\mathbf{L}\cdot\mathbf{n}(t), 0) + c_{\text{spec}}(t)\max(-\mathbf{r}\cdot\mathbf{v}, 0)^{\alpha}$$

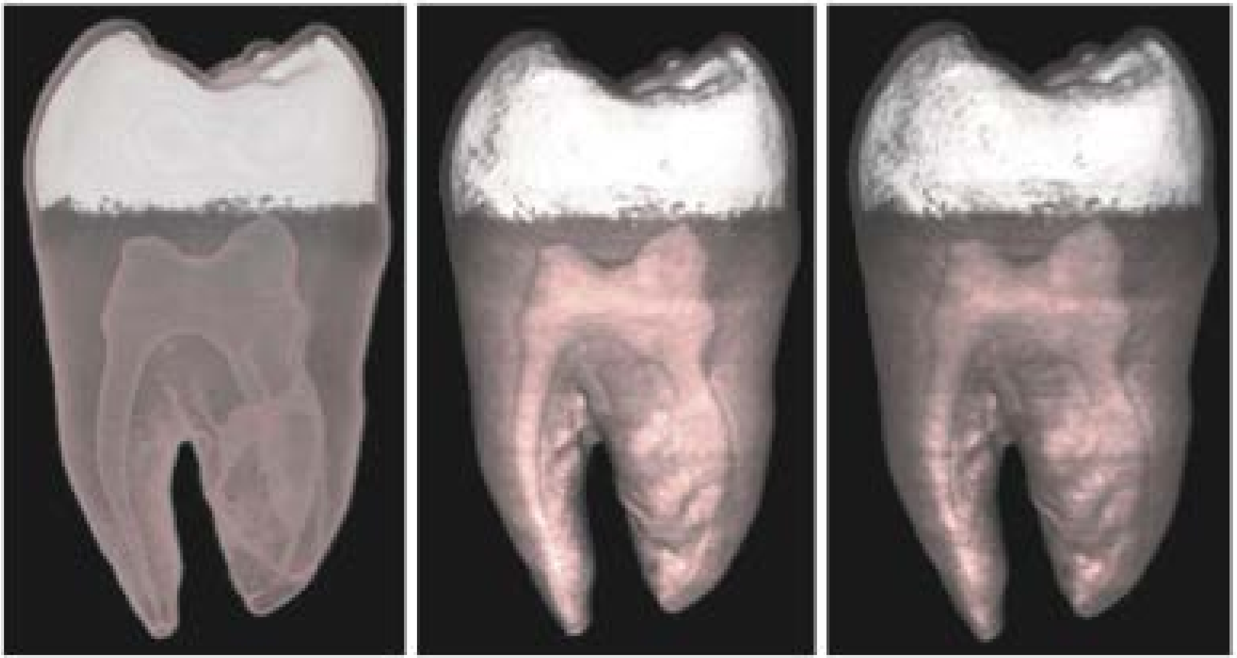
How to implement

- lighting coefficients c and light vector $\mathbf{L}$: user sets them as desired
- surface normal $\mathbf{n}$: compute from gradient of scalar value

(we did the same for isosurfaces, see Module 3)

Volumetric Shading

Results



no shading | diffuse lighting | diffuse and specular lighting

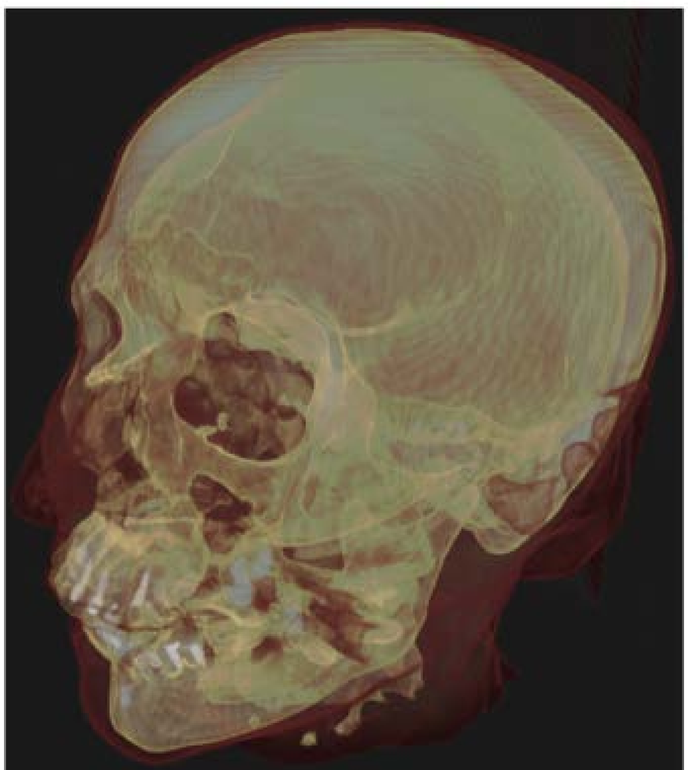
Shading

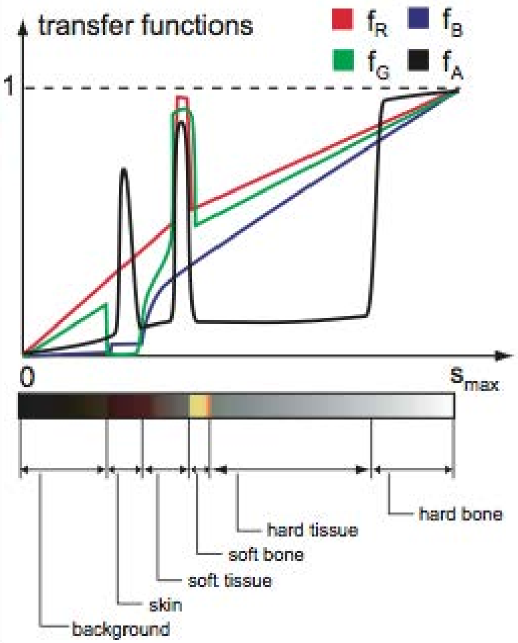
- gives very good cues of depth and shape structure
- is quite cheap and simple to compute

Transfer Functions

Extremely powerful modeling tool (mainly when using composite ray function)

- design four functions f_R, f_G, f_B, f_A
- use color and transparency to emphasize desired material properties (e.g. tissue type)
- use any ray function described so far



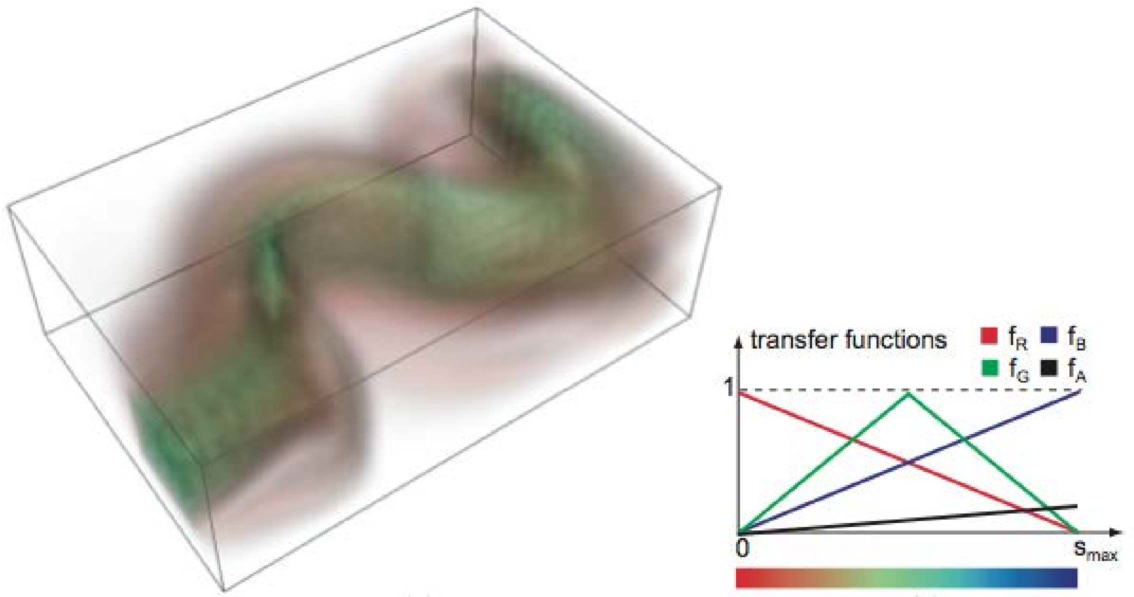


Example
Human head CT

- emphasize bone
- show also muscles

Volume Visualization of Vector Fields

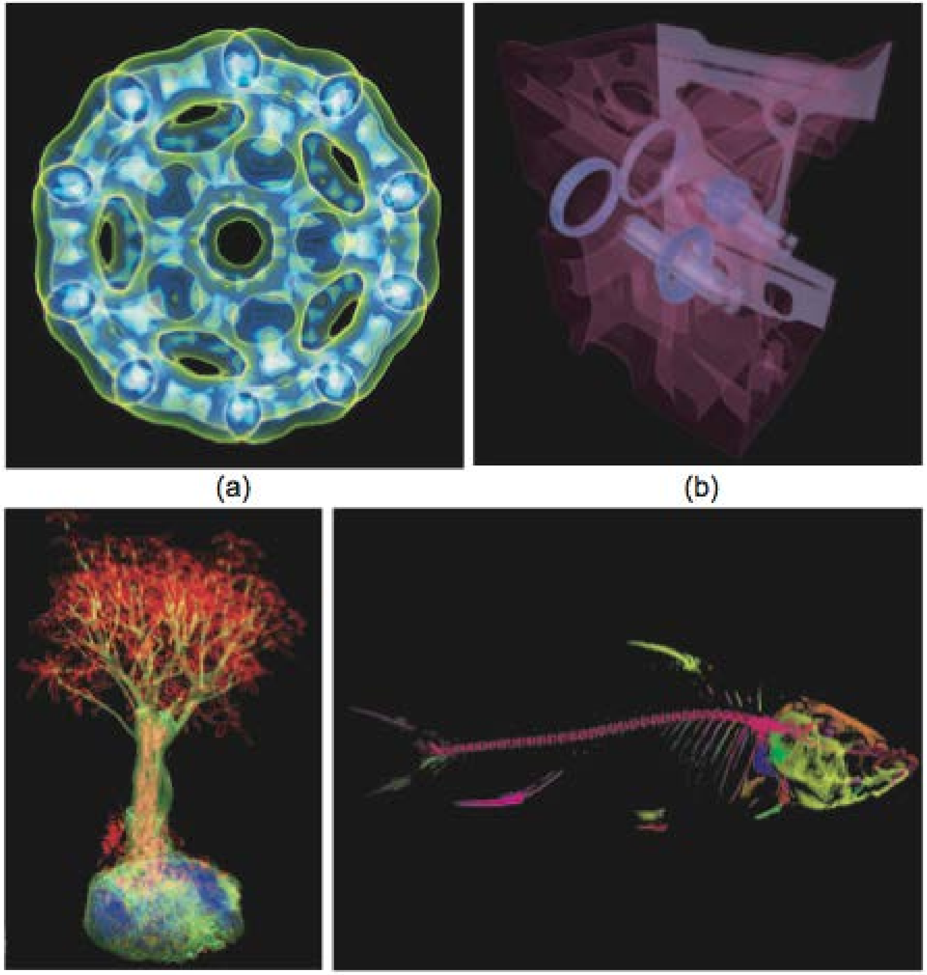
Volume rendering can be used to visualize also **vector datasets**



Volume rendering of fluid flow vector field magnitude

- red = slow flow
- green = more rapid flow
- blue = fastest flow

Transfer Function Examples



a) electron density
b) car engine part
c) bonsai tree (scanned)
d) fish

Transfer Function Example

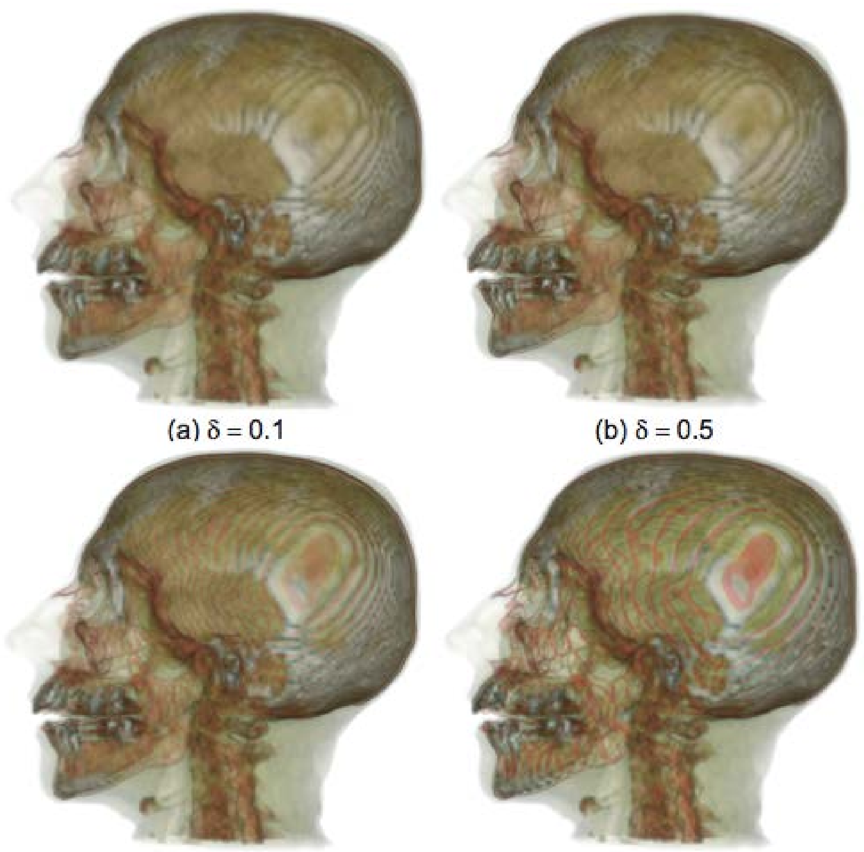
Volume rendering of human MRI dataset

- shading: mimics natural lighting
- backdrop added for extra effect

Implementation Issues

Sampling density

- recall the ray parameterization $q(t) = (1-t)q_0 + tq_1, \quad t \in [0,1]$
- we need to sample along the ray (e.g. integrate, compute min/max, etc)
- how small should we take the sampling step $\delta = dt$?

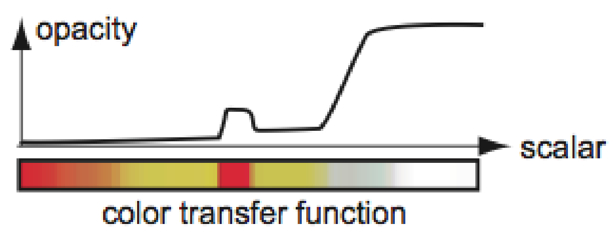


Human head CT, four different δ values

- smaller δ: more accuracy
- too small δ: slow rendering

Practical guideline

- δ should never exceed a voxel size (otherwise we skip voxels while traversing the ray…)



Volume Visualization Summary

Volume visualization (book Chapter 10)

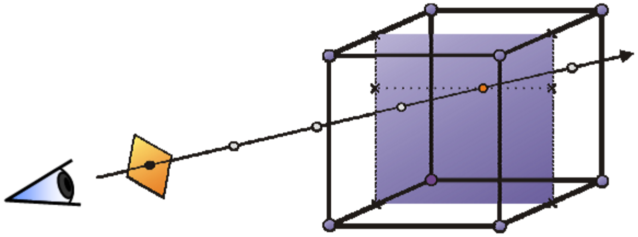
- Extends classical scalar visualization to ‘see through’ 3D volumes
  - ray functions and transfer functions
- Evaluation
  - produces highly realistic, easy to interpret images
  - requires quite some computational power
  - can be easily accelerated using GPUs (e.g. pixel shaders, CUDA)
  - good transfer function design: critical, application-dependent, hard

Volume Rendering in More Depth

We take a closer look at how two fundamental problems are addressed

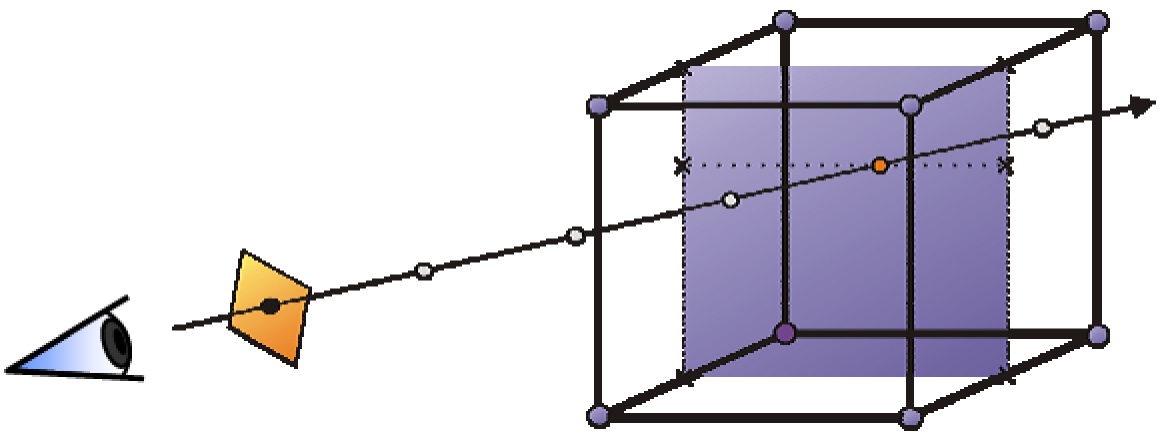
- How can we render at interactive speeds?
  - Image-order techniques (i.e. ray-casting)
  - Object-order techniques (using rasterization)
  - What optimizations can be made for each approach?

- How can we generate high-quality images?
  - What mathematical model do we use to render?
  - How does the model work with the approaches listed above?



A Volume Rendering Optical Model

- Light interacts with volume contents through:
  - Absorption
  - Emission
  - Scattering
- Sample volume along viewing rays
- Accumulate optical properties



Transfer Function

- Maps voxel data values to optical properties

Voxel Data
- Density
- Temperature

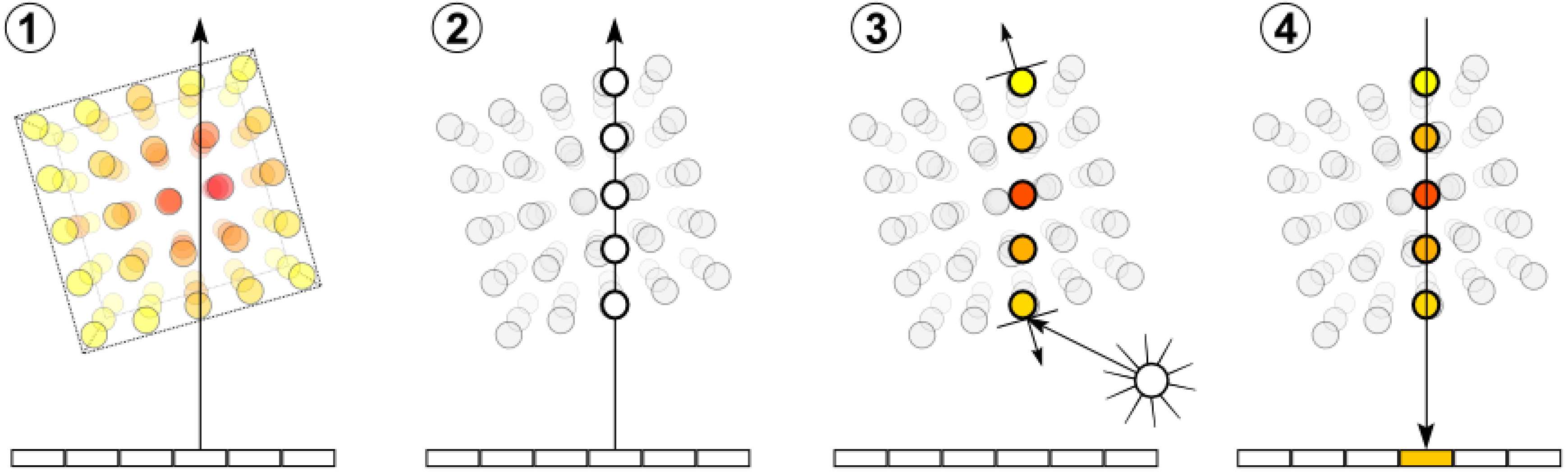
→

Optical Properties
- Color
- Opacity

- Glorified color maps
- Emphasize or classify features of interest in the data
- Piecewise linear functions, Look-up tables, 1D, 2D
- GPU – simple shader functions, texture lookup tables

Image Order Technique: Volume Ray Marching

1. Raycast – once per pixel
2. Sample – uniform intervals along ray
3. Interpolate – trilinear interpolate, apply transfer function
4. Accumulate – integrate optical properties



Ray Marching Accumulation Equations

- Accumulation = Integral
- Color

$$\overline{C} = \int_0^\infty \overline{C}_i T_i \, ds$$

Transmissivity = 1 - Opacity

$$T = 1 - A$$

Total Color = Accumulation (Sampled Colors x Sampled Transmissivities)

Ray Marching Accumulation Equations

- Discrete Versions
- Accumulation = Sum
- Color

$$\overline{C} = \sum_{i=1}^{n} \overline{C}_i T_i$$

Transmissivity = 1 - Opacity

$$T = 1 - A$$

- Opacity

$$A = 1 - \prod_{j=1}^{n} (1 - A_j)$$

$$\boxed{C = \sum_{i=1}^{n} C_i \prod_{j=1}^{i-1} (1 - A_j)}$$

Compositing Color Samples

- Over operator – back-to-front order

$$\hat{C}_i = C_i + (1 - A_i)\hat{C}_{i+1}$$

$$\hat{A}_i = A_i + (1 - A_i)\hat{A}_{i+1}$$

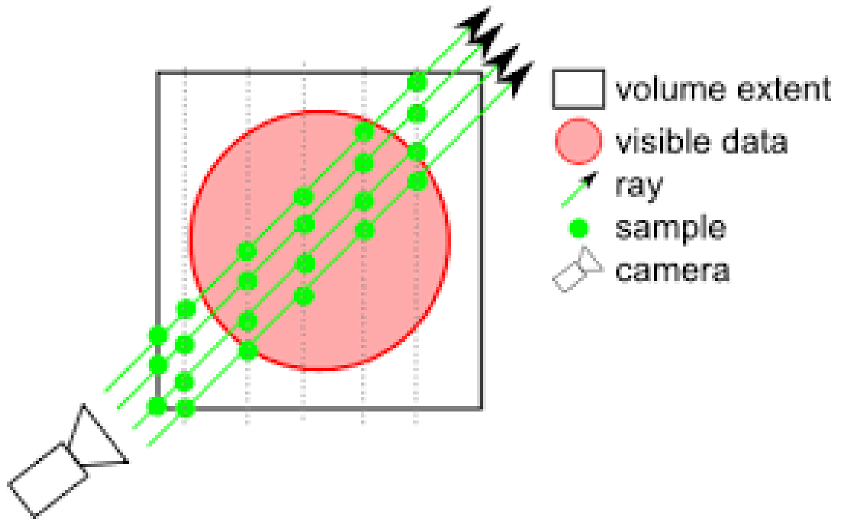
- Under operator – front-to-back order

$$\hat{C}_i = (1 - \hat{A}_{i-1})C_i + \hat{C}_{i-1}$$

$$\hat{A}_i = (1 - \hat{A}_{i-1})A_i + \hat{A}_{i-1}$$

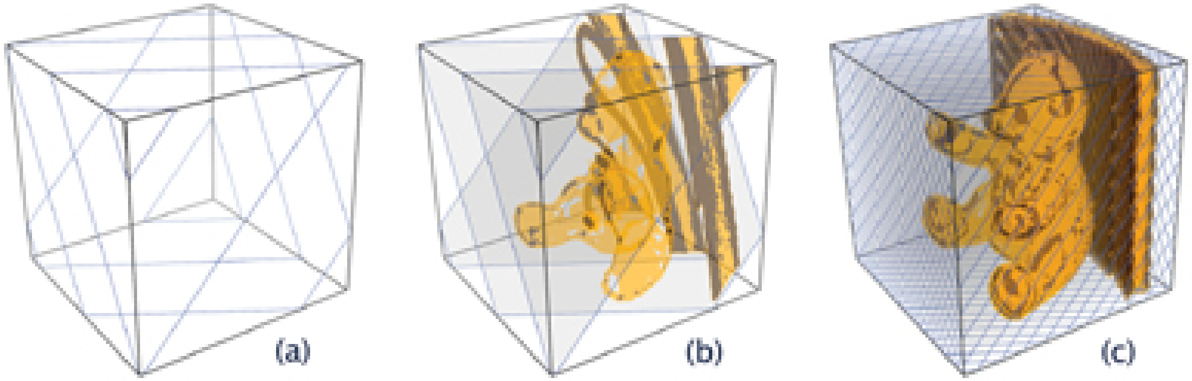
CPU Based Volume Rendering

- Raycast and raymarch for each pixel in scene
  - Camera (eye) location: x_C
  - For Each Pixel
    - Look Direction: $\hat{n}$
    - Cast Ray Along: $x_C + \hat{n}s$
    - Accumulate Color Along Line
  - Sequential or coarse-grained parallel process
    - Minutes or Hours per frame
  - Optimizations
    - Space Partitioning
    - Early Ray Termination

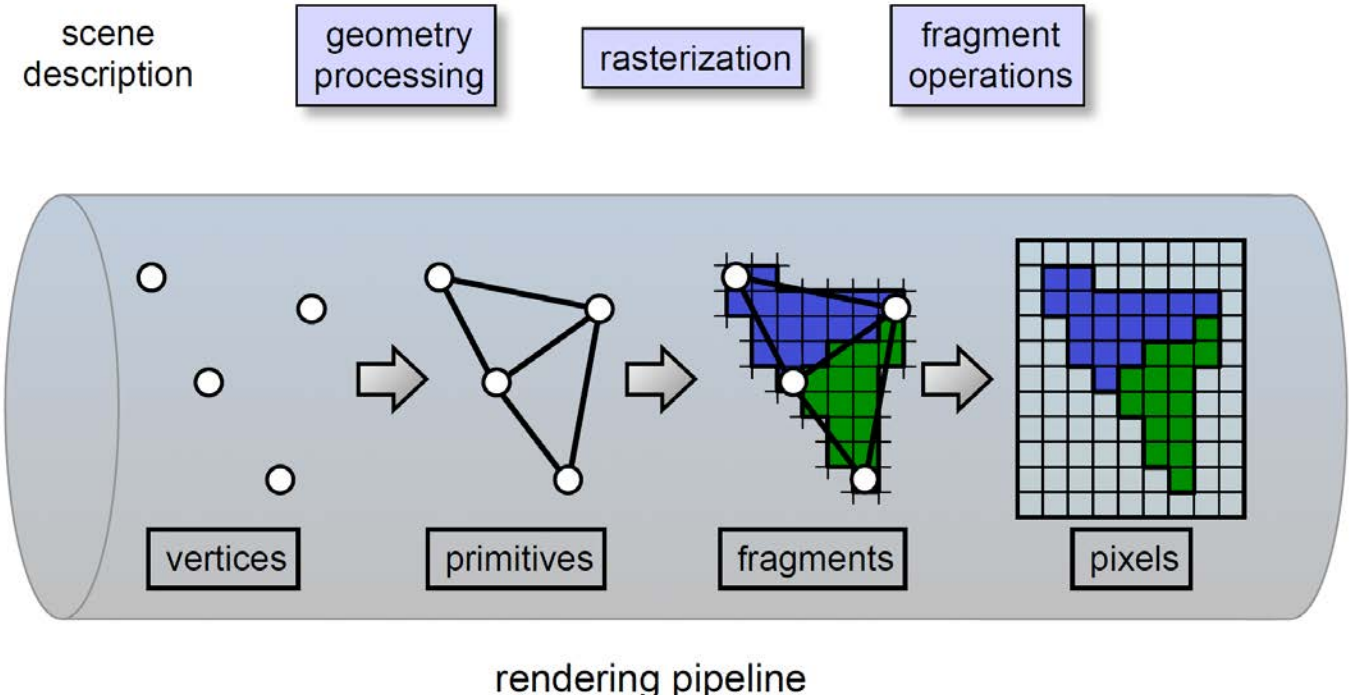


Using Rasterization for Volume Rendering

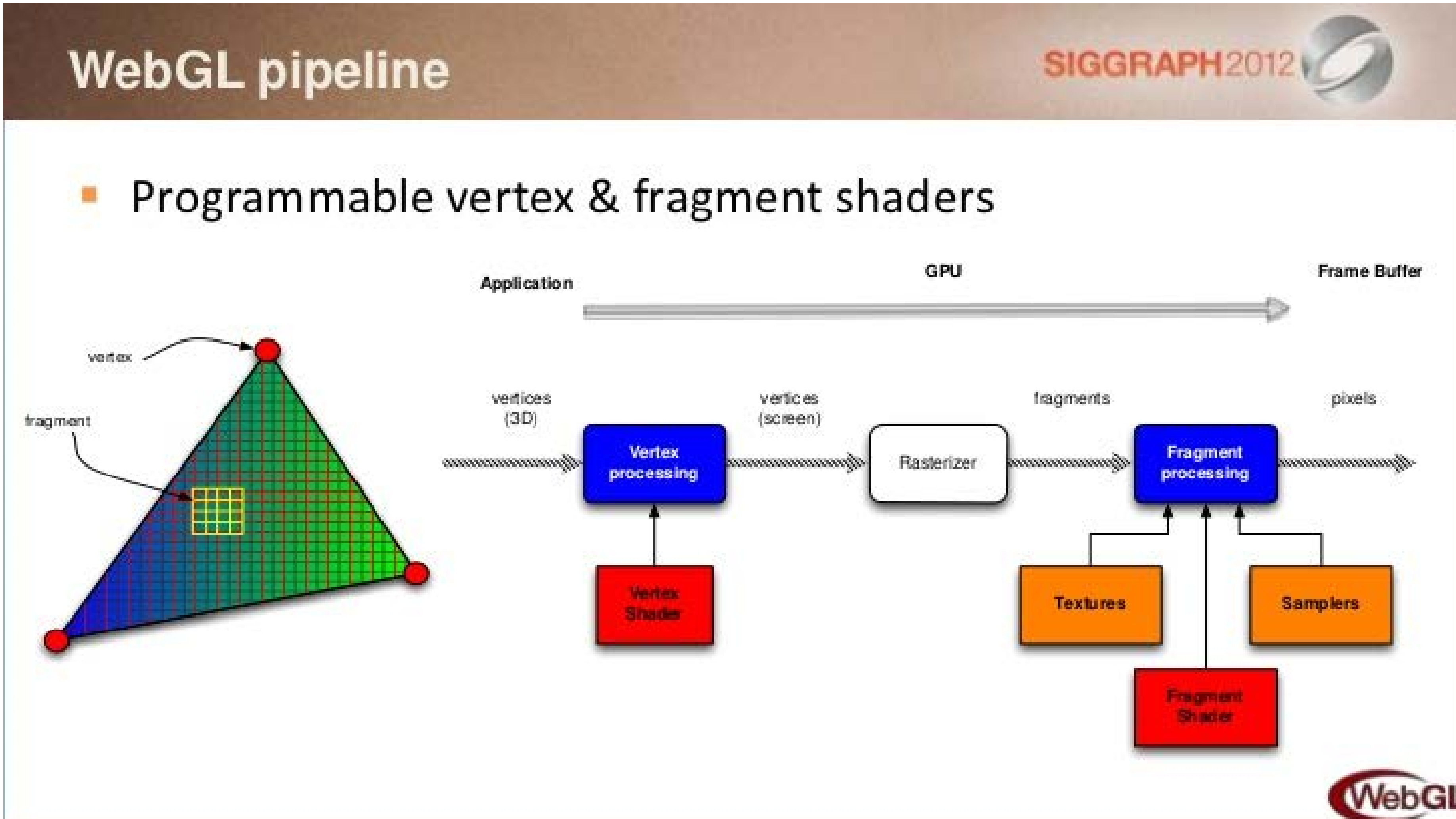
- How can we speed things up?
- Modern rasterization engines are designed for interactive speed
  - OpenGL, D3D, Vulkan
- It's possible to use that technology for volume rendering
  - We can render polygons and composite them
    - Each polygon will be a slice through the volume
  - So is this object-order or image-order rendering?



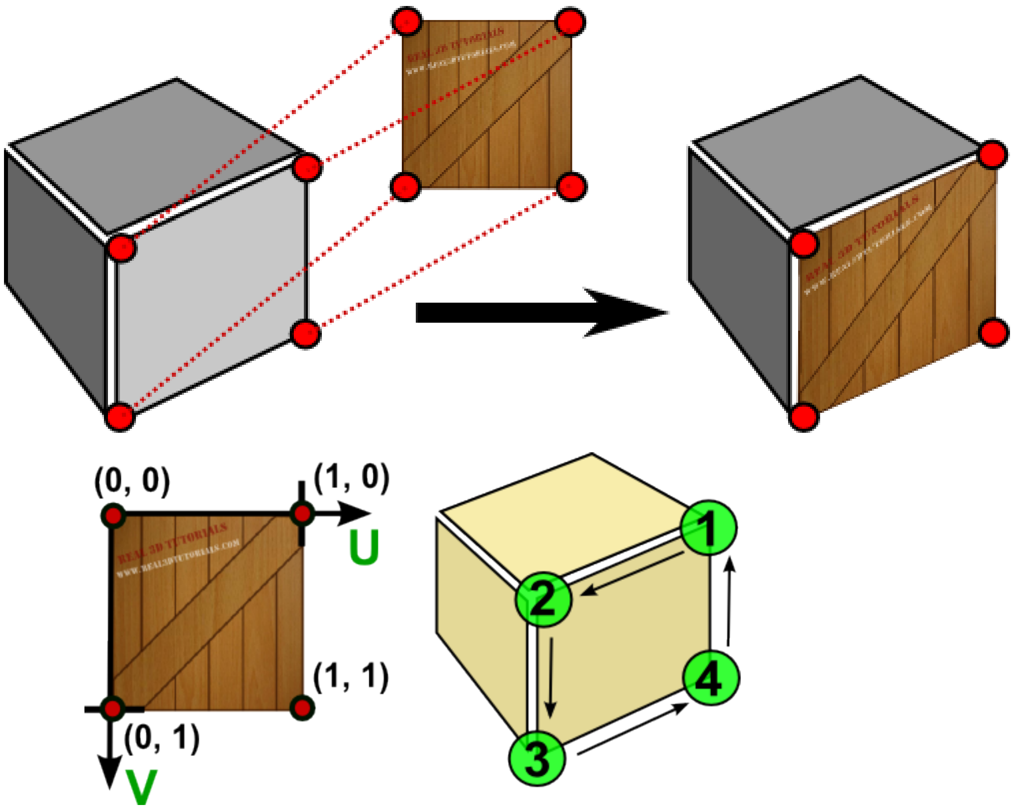
A Modern Rasterization Engine in Two Slides



A Modern Rasterization Engine in Two Slides



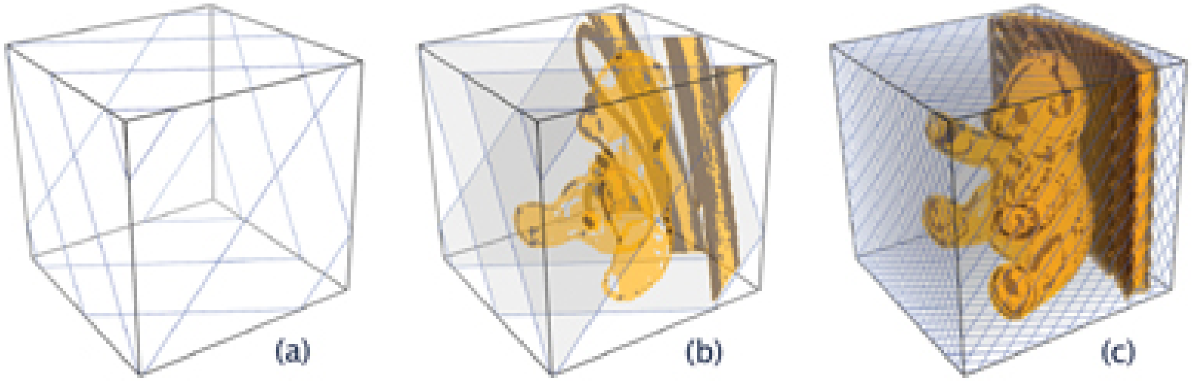
How 2D Texture Mapping Works in One Slide



- Each fragment (pixel) generated by polygon is colored using a 2D image as a lookup table
- Some APIs support 3D textures…similar techniques used to map colors onto polygons

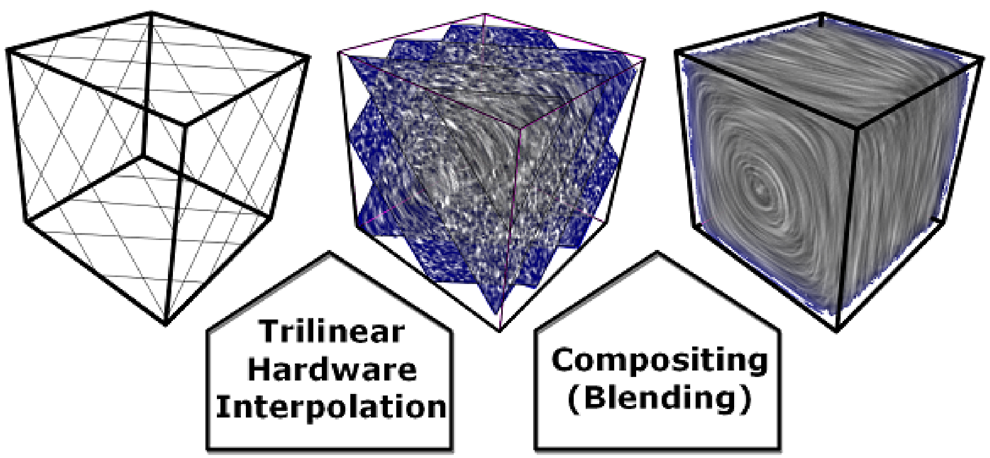
Slice-Based Volume Rendering (SBVR)

- No volumetric primitive in graphics API
- Proxy geometry - polygon primitives as slices through volume
- Texture polygons with volumetric data
- Draw slices in sorted order – back-to-front
- Use fragment shader to perform compositing (blending)



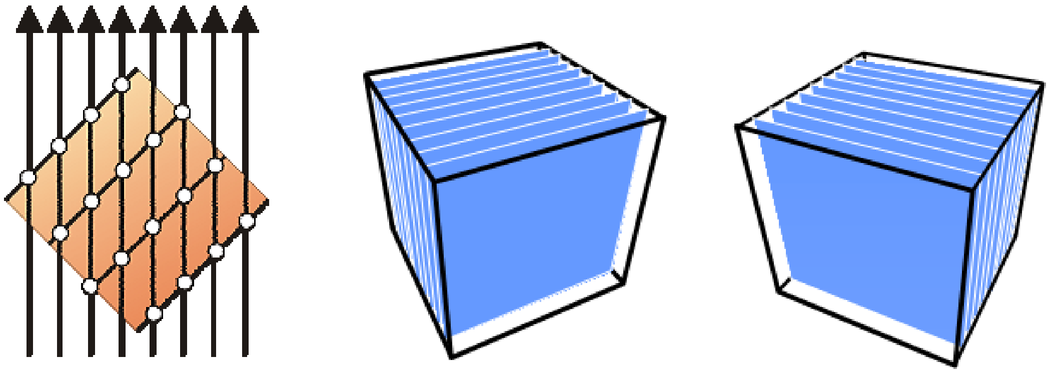
Volumetric Data

- Voxel data sent to GPU memory as
  - Stack of 2D textures
  - 3D texture
- Leverage graphics pipeline

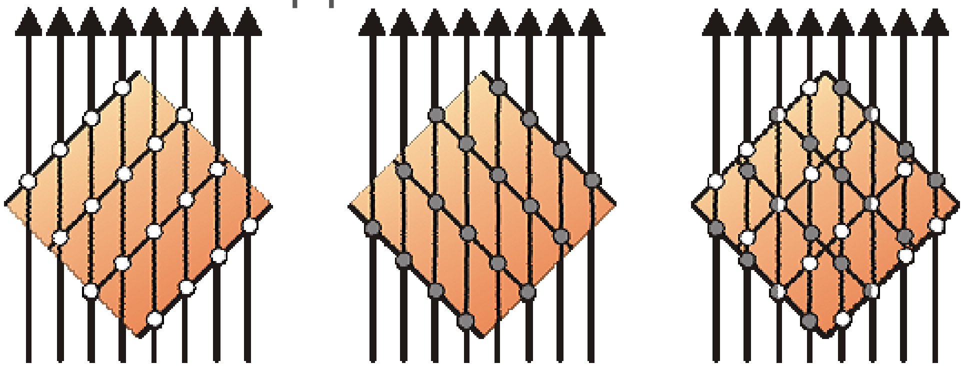


Proxy Geometry

- Object-Aligned Slices
  - Fast and simple
  - Three stacks of 2D textures – x, y, z principle directions



  - Texture stack swapped based on closest to viewpoint

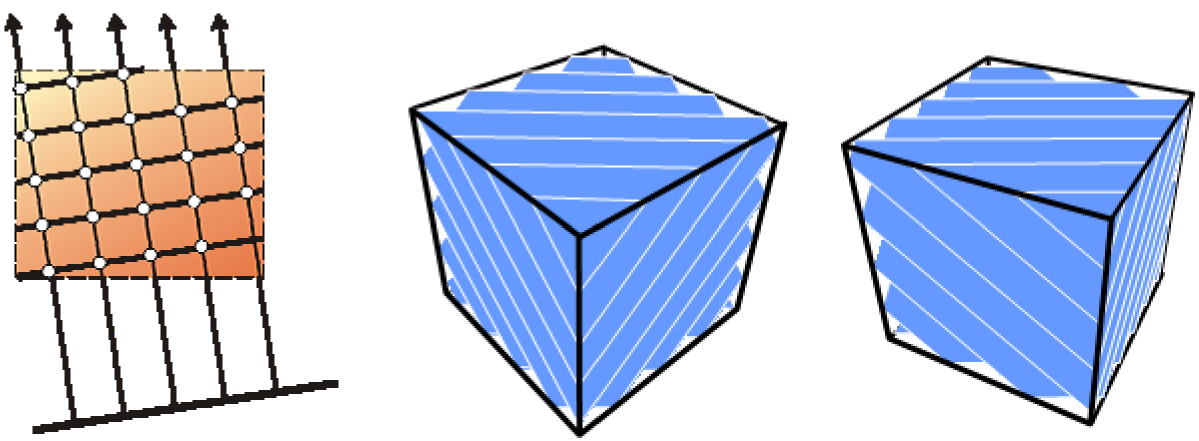


Proxy Geometry

- Issues with Object-Aligned Slices
  - 3x memory consumption
    - Data replicated along 3 principle directions
  - Change in viewpoint results in stack swap
    - Image popping artifacts
    - Lag while downloading new textures
  - Sampling distance changes with viewpoint
    - Intensity variations as camera moves

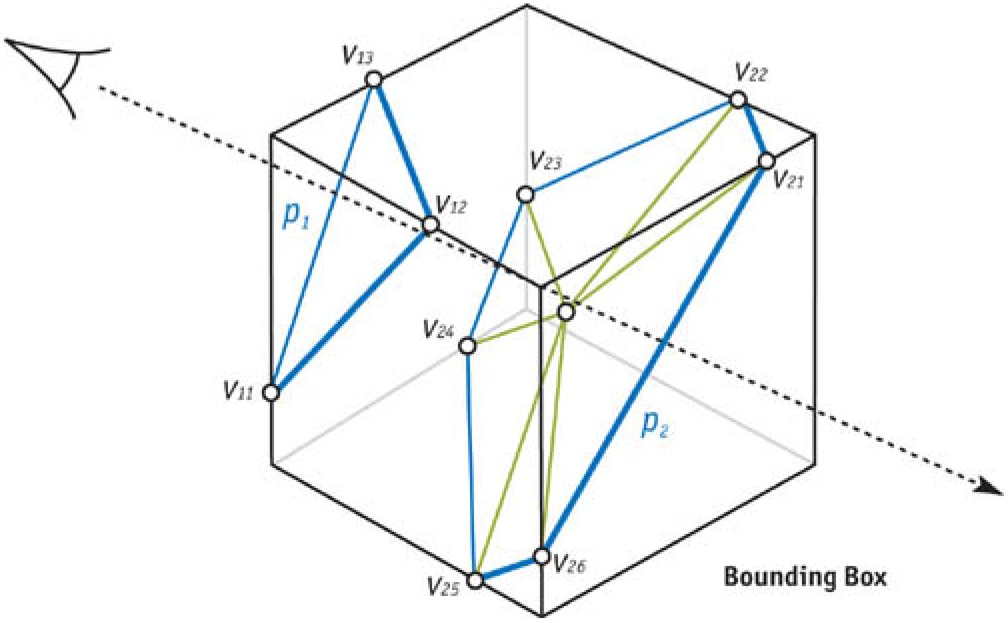
Proxy Geometry

- View-Aligned Slices
  - Slower, but more memory efficient
  - Consistent sampling distance

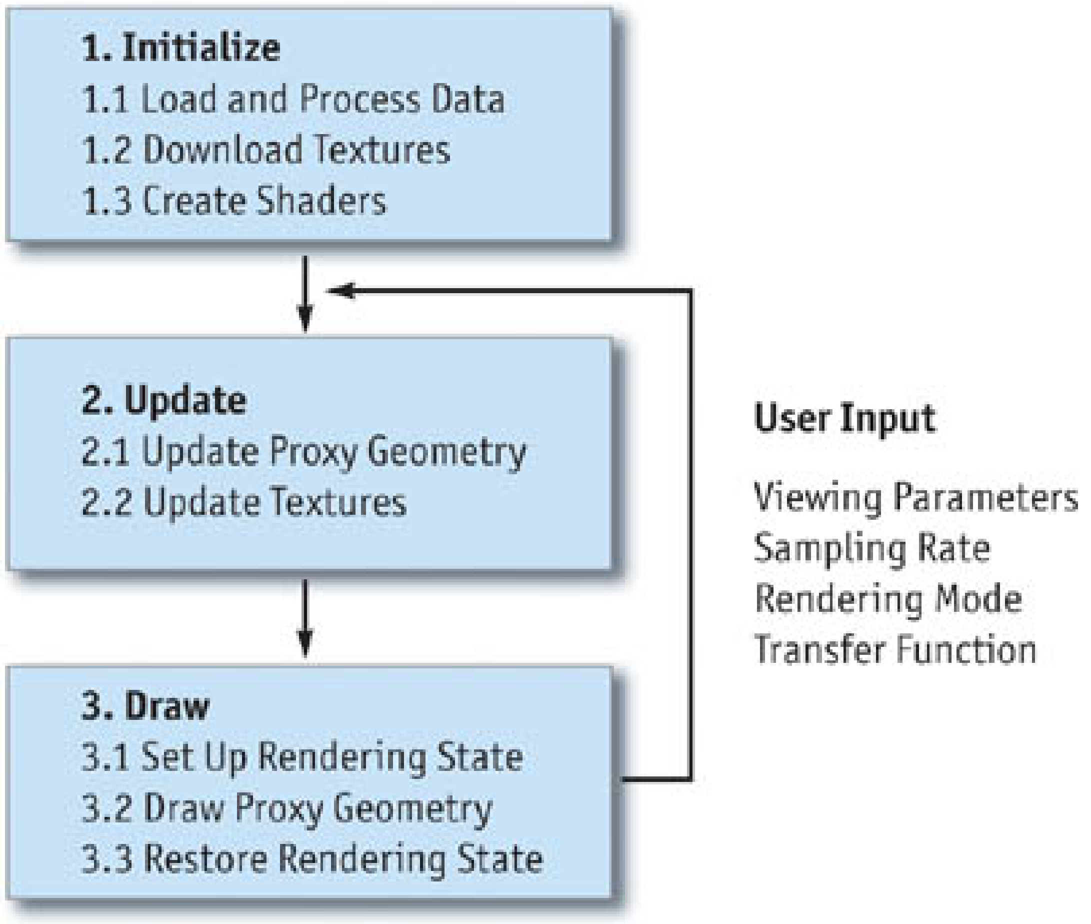


Proxy Geometry

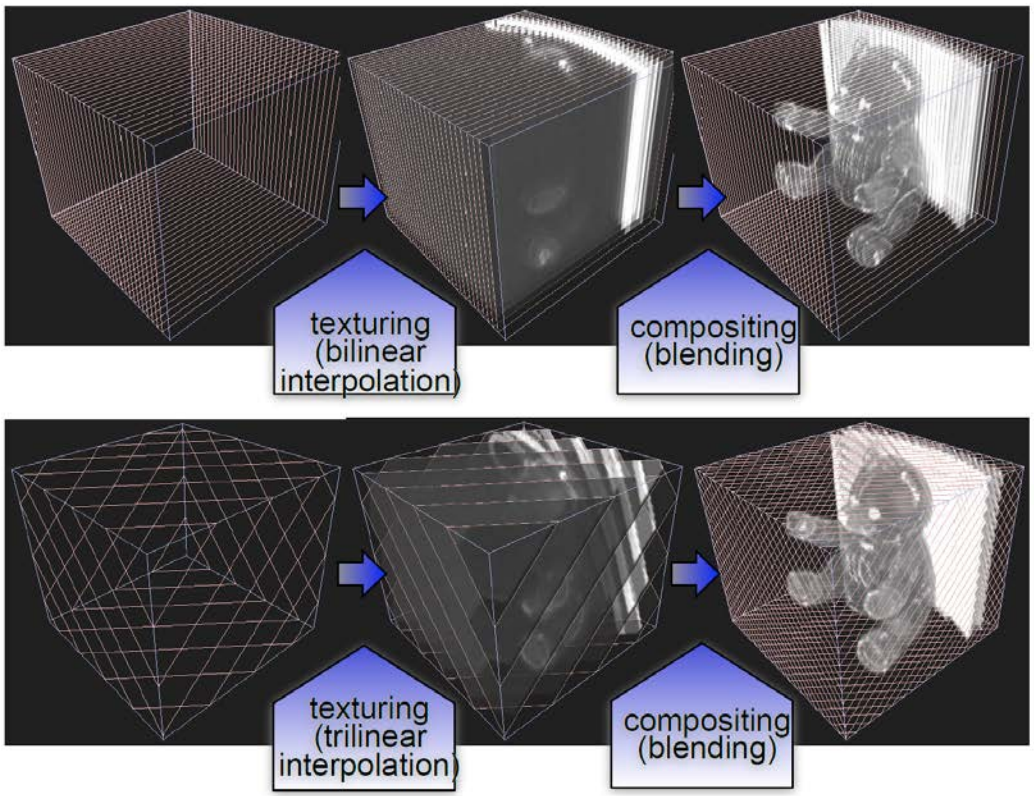
- View-Aligned Slices Algorithm
  - Intersect slicing planes with bounding box
  - Sort resulting vertices in (counter)clockwise order
  - Construct polygon primitive from centroid as triangle fan



Sliced-Based Volume Rendering Steps



Slice-based Volume Rendering



2D textures
axis-aligned

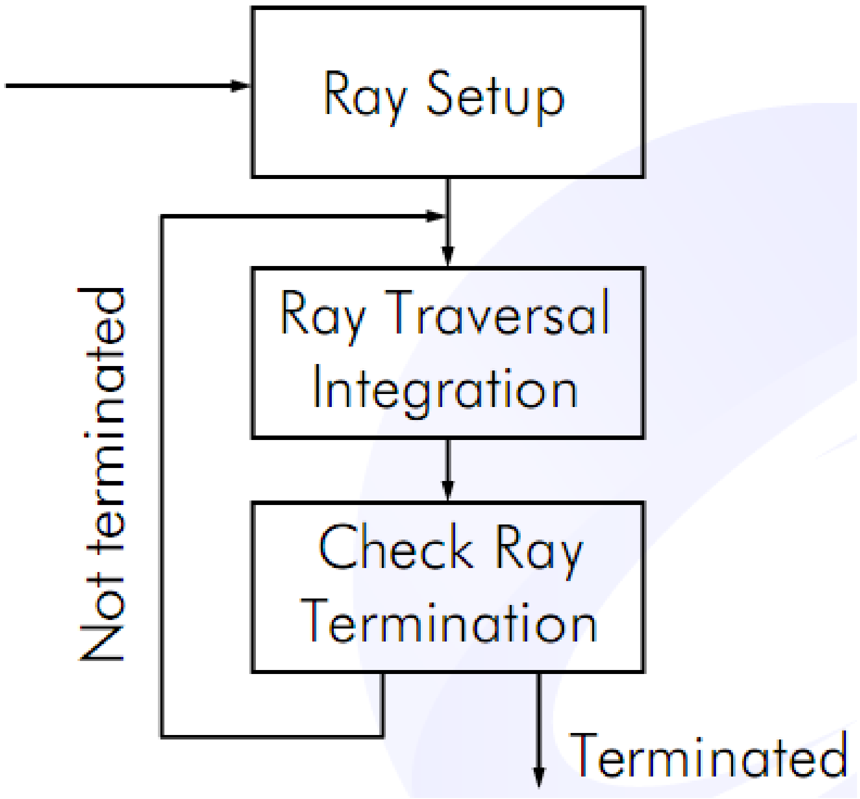
3D texture
view-aligned

Volume Raycasting on GPU

- Alternative: Use the GPU to cast rays through volume
  - Image or Object order?
- Raymarching implemented in fragment shader
  - Cast rays of through volume
  - Accumulate color and opacity
  - Terminate when opacity reaches threshold

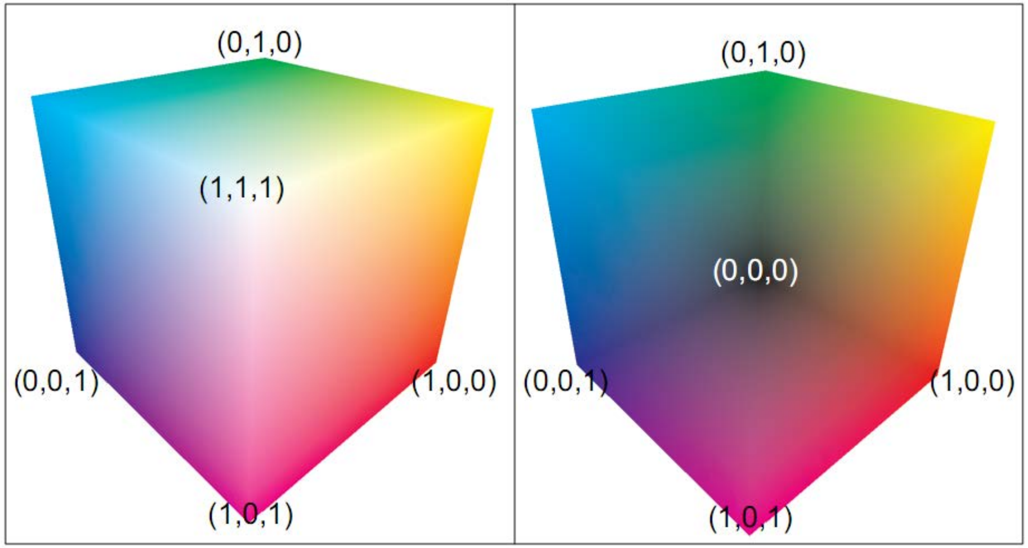
Volume Raycasting on GPU

- Multi-pass algorithm
- Initial passes
  - Precompute ray directions and lengths
- Additional passes
  - Perform raymarching in parallel for each pixel
  - Split up full raymarch to check for early termination



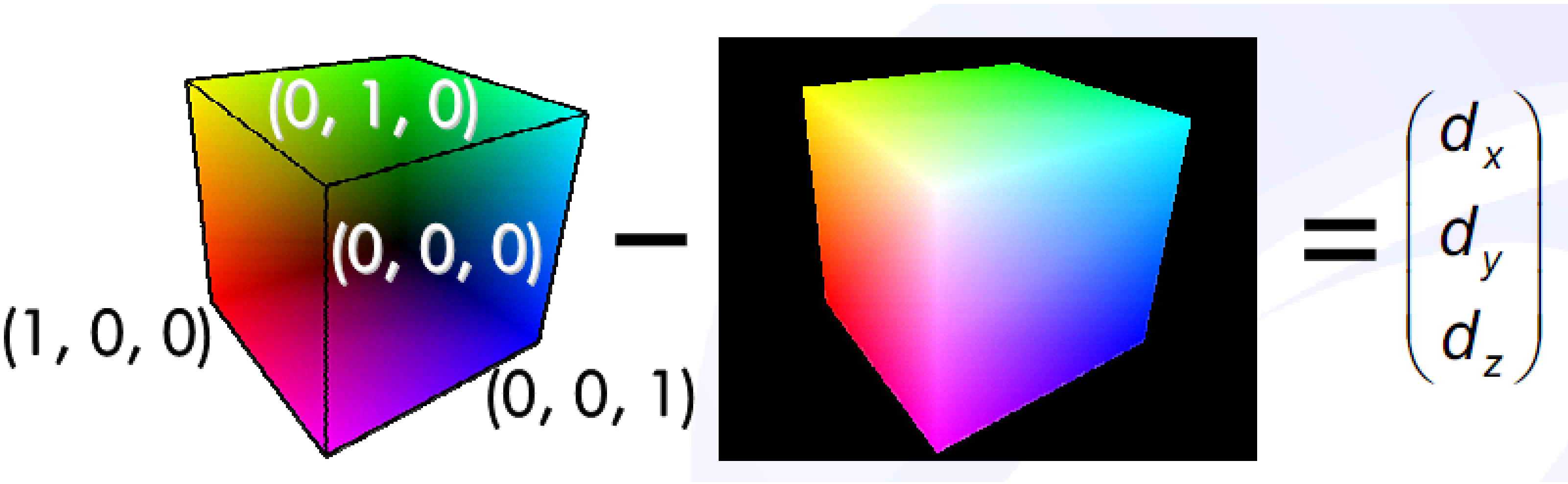
Step 1: Ray Direction Computation

- Ray direction computed for each pixel
- Stored in 2D texture for use in later steps
- Pass 1: Front faces of volume bounding box
- Pass 2: Back faces of volume bounding box
- Vertex color components encode object-space principle directions



Step 1: Ray Direction Computation

- Subtraction blend two textures
- Store normalized direction – RGB components
- Store length – Alpha component



$$- \quad = \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}$$

Fragment Shader Raymarching

- DIR[x][y] – ray direction texture
  - 2D RGBA values
- P – per-vertex float3 positions, front of volume bounding box
  - Interpolated for fragment shader by graphics pipeline
- s – constant step size
  - Float value
- d – total raymarched distance, s x #steps
  - Float value

- Parametric Ray Equation

$$r = P + d \cdot DIR[x][y]$$

  - r – 3D texture coordinates used to sample voxel data

Fragment Shader Raymarching

- Ray traversal procedure split into multiple passes
  - M steps along ray for each pass
  - Allows for early ray termination, optimization
- Optical properties accumulated along M steps
  - Simple compositing/blending operations
  - Color and alpha(opacity)
- Accumulation result for M steps blended into 2D result texture
  - Stores overall accumulated values between multiple passes
- Intermediate Pass – checks for early termination
  - Compare opacity to threshold
  - Check for ray leaving bounding volume

Improving Image Quality

- Local illumination using Blinn-Phong illumination model
- Volumetric shadows

Local Illumination

- Blinn-Phong Shading Model

$$I = k_a + I_L k_d (\hat{l} \cdot \boxed{\hat{n}}) + I_L k_s (\hat{h} \cdot \boxed{\hat{n}})^N$$

Resulting = Ambient + Diffuse + Specular

- Requires surface normal vector
  - Whats the normal vector of a voxel?

Local Illumination

- Blinn-Phong Shading Model

$$I = k_a + I_L k_d (\hat{l} \cdot \boxed{\hat{n}}) + I_L k_s (\hat{h} \cdot \boxed{\hat{n}})^N$$

Resulting = Ambient + Diffuse + Specular

- Requires surface normal vector
  - Whats the normal vector of a voxel? **Gradient**
  - Central differences between neighboring voxels

$$grad(I) = \nabla I = \frac{(right - left)}{2x}, \frac{(top - bottom)}{2x}, \frac{(front - back)}{2x}$$

Local Illumination

- Compute on-the-fly within fragment shader
  - Requires 6 texture fetches per calculation
- Precalculate on host and store in voxel data
  - Requires 4x texture memory
  - Pack into 3D RGBA texture to send to GPU

Voxel Data

- X Gradient
- Y Gradient
- Z Gradient
- Value

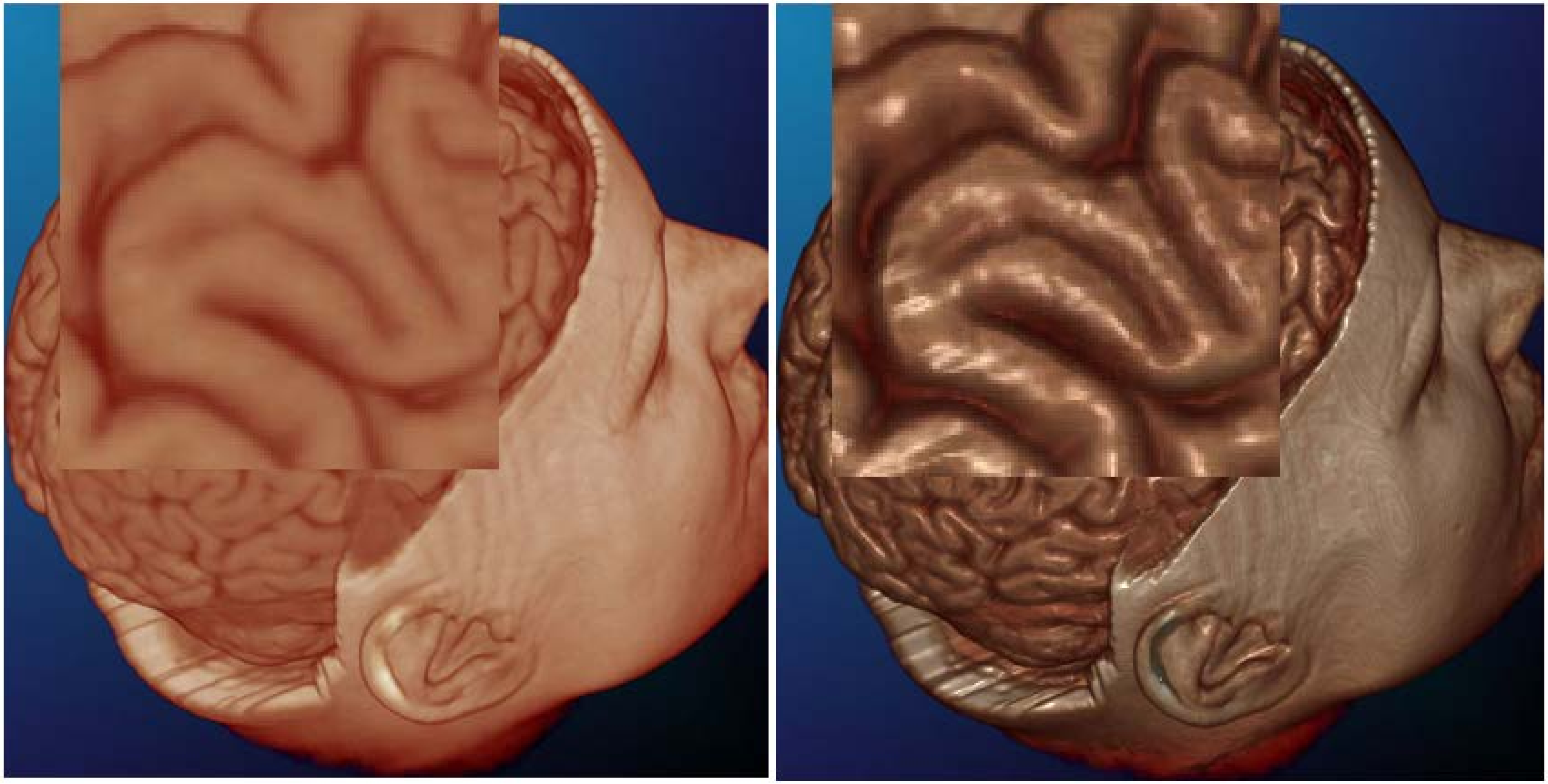
→

3D Texture

- R
- G
- B
- A

Local Illumination

- Improve perception of depth
- Amplify surface structure



Volumetric Shadows

- Light attenuated as passes through volume
- ‘Deeper’ samples receive less illumination
- Second raymarch from sample point to light source
  - Accumulate illumination from sample’s point of view
  - Same accumulation equations
- Precomputed Light Transmissivity
  - Precalculate illumination for each voxel center
  - Trilinearly interpolate at render time
  - View independent, scene/light source dependent

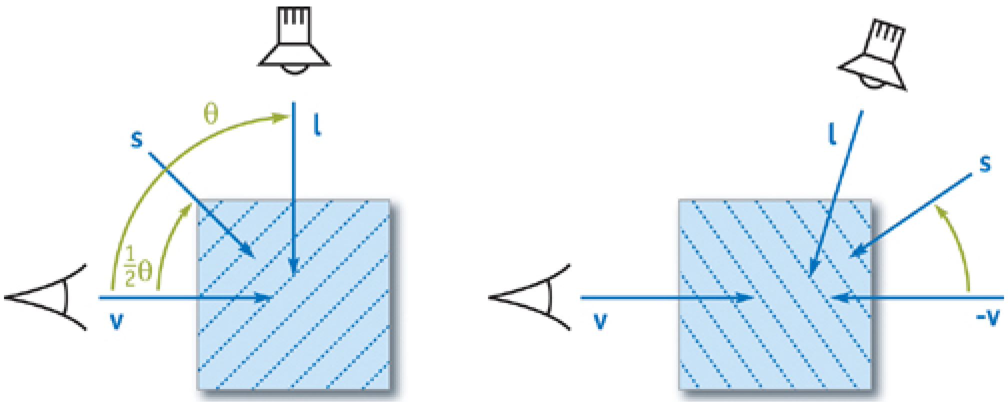
Volumetric Shadows on GPU

- Light attenuated from light's point of view
- CPU – Precomputed Light Transfer
  - Secondary raymarch from sample to light source
- GPU
  - Two-pass algorithm
  - Modify proxy geometry slicing
  - Render from both the eye and the light's POV
    - Two different frame buffers

Two Pass Volume Rendering with Shadows

- Slice axis set half-way between view and light directions
  - Allows each slice to be rendered from eye and light POV
- Render order for light – front-to-back
- Render order for eye – (a) front-to-back

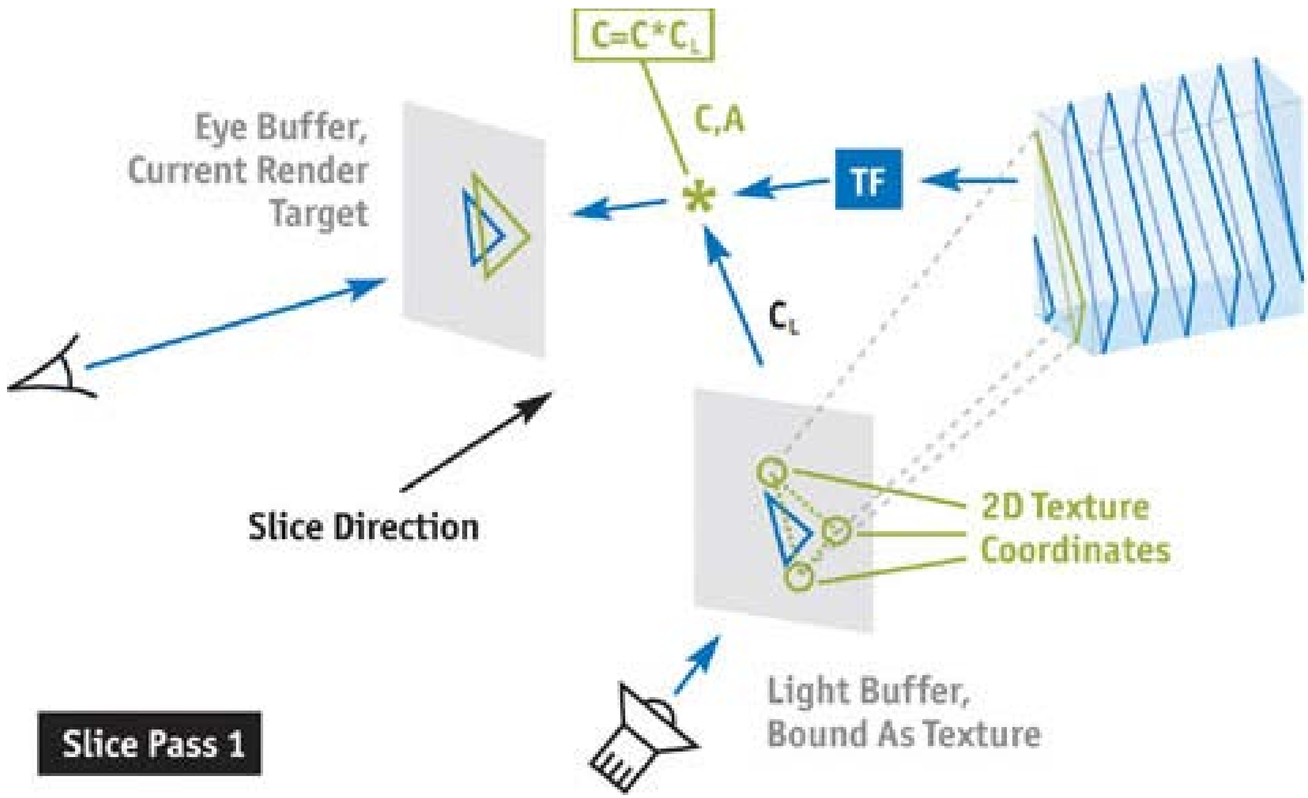
  (b) back-to-front



(a) (b)

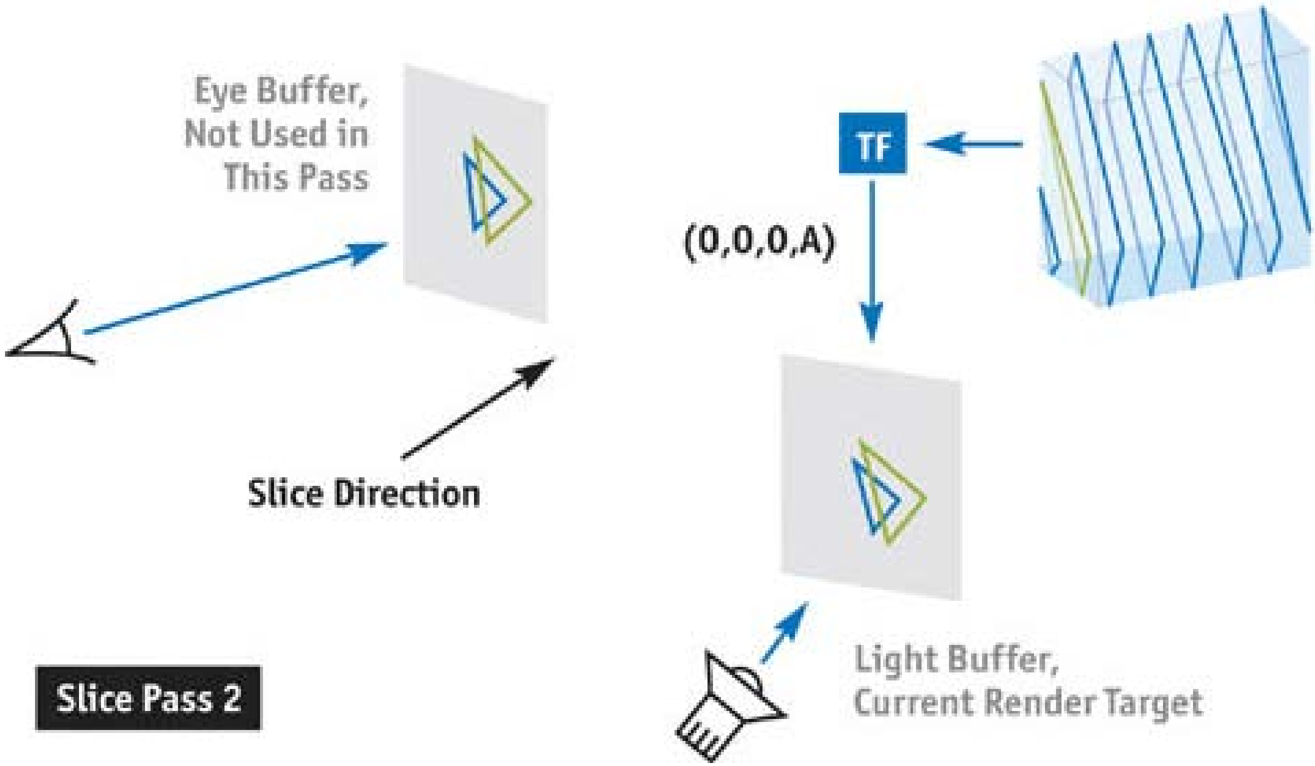
First Pass

- Render from eye
- Fragment shader
  - Look up light color from light buffer bound as texture
  - Multiply material color * light color



Second pass

- Render from light
- Fragment shader
  - Only blend alpha values – light transmissivity



Volumetric Shadows

(a) (b)

Volume Rendering in CUDA

- 3D Slicer – www.slicer.org
- Open source software for visualization and image analysis
- Funded by NIH, medical imaging, MRI data
- Currently integrating CUDA volume rendering into Slicer 3.2