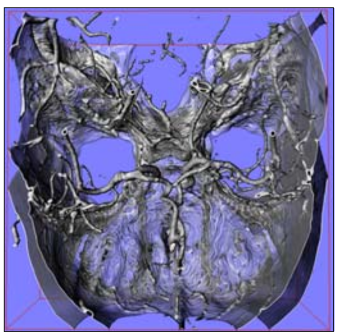
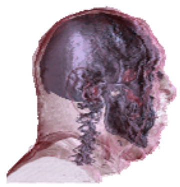
Tutorial 7
Real-Time Volume Graphics

Klaus Engel
Markus Hadwiger
Christof Rezk Salama

Real-Time Volume Graphics
[01] Introduction and Theory

Appliations: Medicine



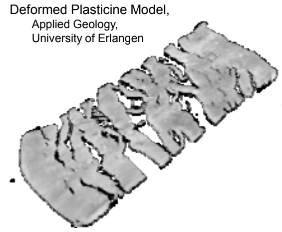


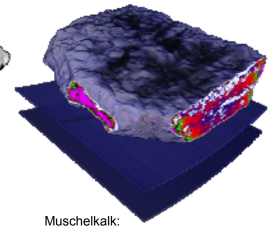
CT Human Head:
Visible Human Project,
US National Library of Medicine, Maryland, USA

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

Applications: Geology

Deformed Plasticine Model,
Applied Geology,
University of Erlangen





Muschelkalk:
Paläontologie,
Virtual Reality Group,
University of Erlangen

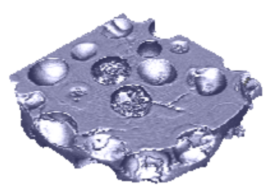
Applications: Archeology

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

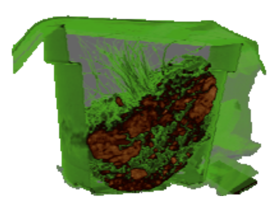
Applications:

Material Science, Quality Control



Micro CT, Compound Material,
Material Science Department, University of Erlangen

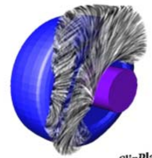
Biology

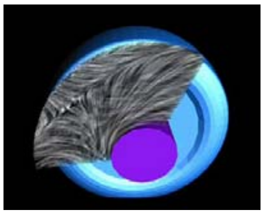


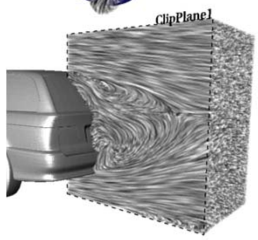
biological sample of the soil, CT,
Virtual Reality Group,
University if Erlangen

Applications

Computational
Science and Engineering

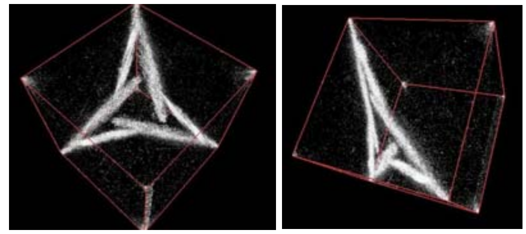






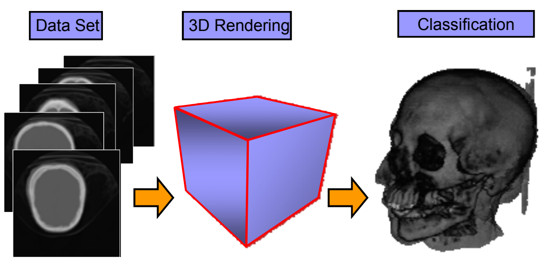
Applications: Computer Science

- Visualization of Pseudo Random Numbers



Entropy of Pseudo Random Numbers,
Dan Kaminsky, Doxpara Research, USA, www.doxpara.com

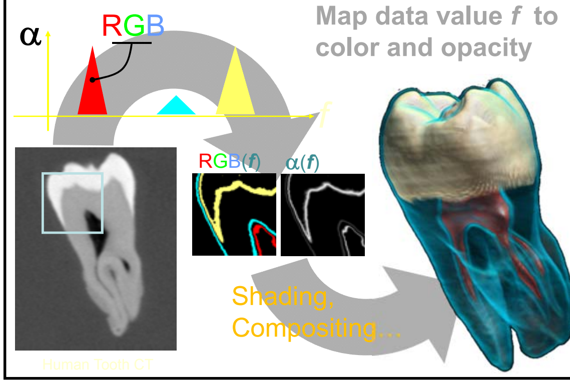
Outline



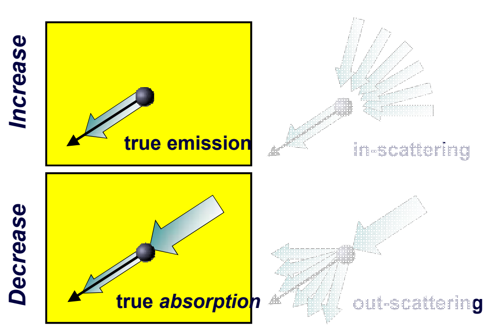
- in real-time on commodity graphics hardware

Transfer Functions (TFs)

Map data value *f* to color and opacity



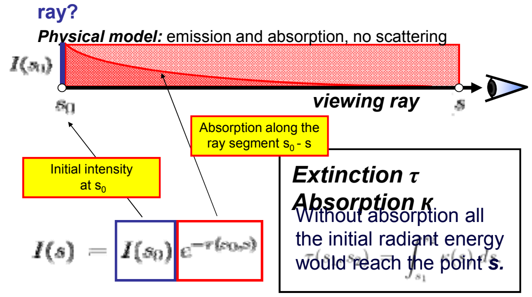
Physical Model of Radiative Transfer



Ray Integration

How do we determine the radiant energy along the ray?

Physical model: emission and absorption, no scattering



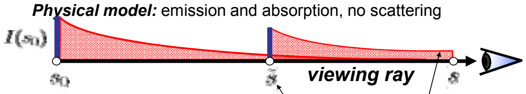
$$I(s) = I(s_0)\, e^{-\tau(s_0,s)}$$

Extinction τ
Absorption κ
Without absorption all the initial radiant energy would reach the point ***s.***

Ray Integration

How do we determine the radiant energy along the ray?

Physical model: emission and absorption, no scattering

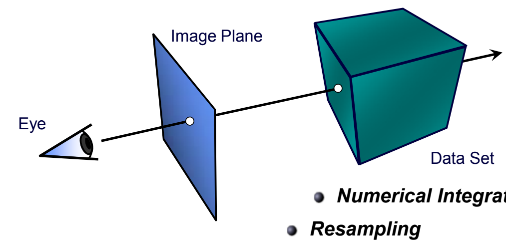


One point $\tilde s$ along the viewing ray emits additional radiant energy.

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

Ray Casting

- Software Solution



- ***Numerical Integration***
- ***Resampling***
- ***High Computational Load***

Numerical Solution

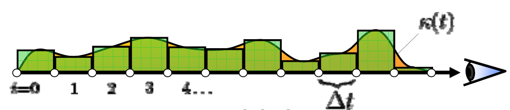
Extinction: $\tau(0,t) \approx \tilde\tau(0,t) = \sum_{i=0}^{\lfloor t/\Delta t\rfloor} \kappa(i\cdot\Delta t)\,\Delta t$

Approximate Integral by Riemann sum:

$$e^{-\tilde\tau(0,t)} = e^{-\sum_{i=0}^{\lfloor t/\Delta t\rfloor}\kappa(i\cdot\Delta t)\,\Delta t}$$

$$\tau(0,t) \approx \sum_{i=0}^{\lfloor t/\Delta t\rfloor} \kappa(i\cdot\Delta t)\,\Delta t$$

Numerical Solution



$$\tau(0,t) \approx \tilde\tau(0,t) = \sum_{i=0}^{\lfloor t/\Delta t\rfloor} \kappa(i\cdot\Delta t)\,\Delta t$$

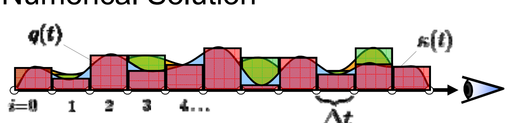
$$e^{-\tilde\tau(0,t)} = \prod_{i=0}^{\lfloor t/\Delta t\rfloor} e^{-\kappa(i\cdot\Delta t)\,\Delta t}$$

Now we introduce opacity:

$$(1 - A_i) = e^{-\kappa(i\cdot\Delta t)\,\Delta t}$$

$$A_i = 1 - e^{-\kappa(i\cdot\Delta t)\,\Delta t}$$

Numerical Solution



$$\tau(0,t) \approx \tilde\tau(0,t) = \sum_{i=0}^{\lfloor t/\Delta t\rfloor} \kappa(i\cdot\Delta t)\,\Delta t$$

$$e^{-\tilde\tau(0,t)} = \prod_{i=0}^{\lfloor t/\Delta t\rfloor} (1 - A_i)$$

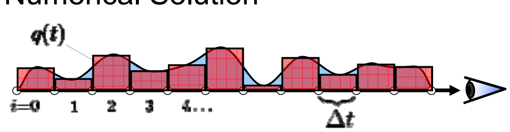
$$q(t) \approx C_i = q(i\cdot\Delta t)\,\Delta t$$

Now we introduce opacity:

$$1 - A_i = e^{-\kappa(i\cdot\Delta t)\,\Delta t}$$

$$\tilde C = \sum_{i=0}^{\lfloor T/\Delta t\rfloor} C_i\, e^{-\tilde\tau(0,t)}$$

Numerical Solution



$$e^{-\tilde\tau(0,t)} = \prod_{i=0}^{\lfloor t/\Delta t\rfloor} (1 - A_i)$$

$$q(t) \approx C_i = q(i\cdot\Delta t)\,\Delta t$$

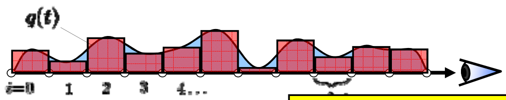
can be computed recursively:

$$\tilde C = \sum_{i=0}^{\lfloor T/\Delta t\rfloor} C_i \prod_{j=0}^{i-1} (1 - A_j)$$

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

Radiant energy observed at position *i* | Radiant energy emitted at position *i* | Absorption at position *i* | Radiant energy observed at position *i–1*

Numerical Solution



Back-to-front compositing

$$\tilde{C} = \sum_{i=0}^{\lfloor T/\Delta t \rfloor} C_i$$

Early Ray Termination:

Stop the calculation when

$$A'_i \approx 1$$

can be computed recursively

Front-to-back compositing

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

Summary

- Emission Absorption Model

true emission true *absorption*

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

- Numerical Solutions

Back-to-front iteration

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

Front-to-back iteration

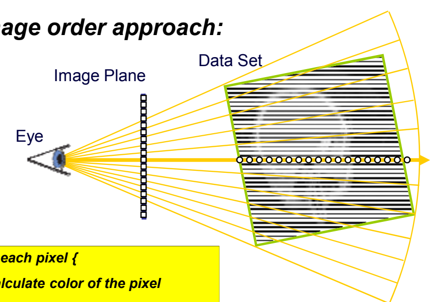
$$C'_i = C'_{i+1} + (1 - A'_{i+1}) C_i$$
$$A'_i = A'_{i+1} + (1 - A'_{i+1}) A_i$$

Real-Time Volume Graphics

[03] GPU-Based Volume Rendering

Volume Rendering

Image order approach:

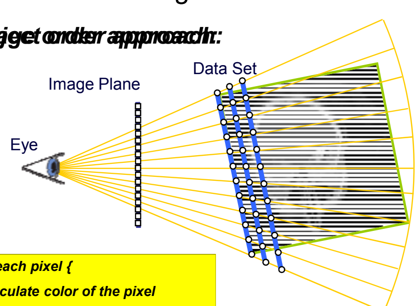


For each pixel {
calculate color of the pixel
}

Volume Rendering

Object order approach:

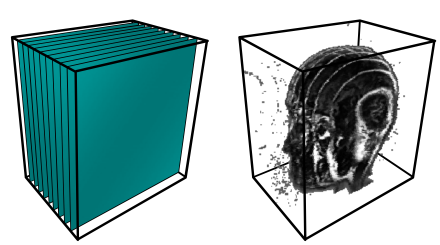
Image order approach:



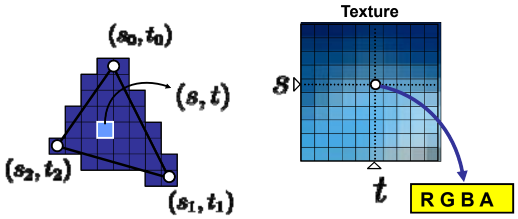
For each pixel {
calculate color of the pixel
}

Texture-based Approaches

- No volumetric hardware-primitives!
- Proxy geometry (Polygonal Slices)



How does a texture work?

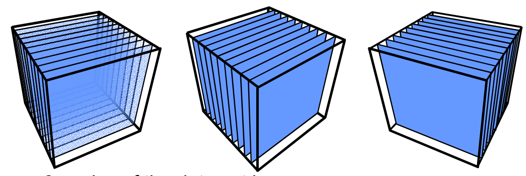


For each fragment: interpolate the texture coordinates **(barycentric)**

Texture-Lookup: interpolate the texture color **(bilinear)**

2D Textures

- Draw the volume as a stack of 2D textures ***Bilinear Interpolation in Hardware***
  - Decompostition into axis-aligned slices



- 3 copies of the data set in memory

Implementation

```
GLfloat pModelViewMatrix[16];

//get the current modelview matrix
glGetFloatv(GL_MODELVIEW_MATRIX, pModelViewMatrix);

//rotate the initial viewing direction
GLfloat pViewVector[4] = 0.0f, 0.0f, -1.0f, 0.0f;
MatVecMultiply(pModelViewMatrix, pViewVector);

//find the maximal vector component
int nMax = FindAbsMaximum(pViewVector);

switch (nMax) {
case X:
   if(pViewVector[X] > 0.0f) {
      DrawSliceStack_PositiveX();
   } else {
      DrawSliceStack_NegativeX();
   }
   break;
case Y:
   if(pViewVector[Y] > 0.0f) {
```

REAL-TIME VOLUME GRAPHICS Eurographics 2006

Implementation

```
//draw slices perpendicular to x-axis
//in back-to-front order
void DrawSliceStack_NegativeX() {

   double dXPos = -1.0;
   double dXStep = 2.0/double(XDIM);

   for(int slice = 0; slice < XDIM; ++slice) {
      //select the texture image corresponding to the slice
      glBindTexture(GL_TEXTURE_2D, textureNamesStackX[slice]);

      //draw the slice polygon
      glBegin(GL_QUADS);
         glTexCoord2d(0.0, 0.0); glVertex3d(dXPos,-1.0,-1.0);
         glTexCoord2d(0.0, 1.0); glVertex3d(dXPos,-1.0, 1.0);
         glTexCoord2d(1.0, 1.0); glVertex3d(dXPos, 1.0, 1.0);
         glTexCoord2d(1.0, 0.0); glVertex3d(dXPos, 1.0,-1.0);
      glEnd();

      dXPos += dXStep;
   }
}
```

Fragment Program

```
// simple 2D texture sampling
float4 main (half2 texUV : TEXCOORD0,
             uniform sampler2D slice) :  COLOR
{
   float4 result = tex2D(slice, texUV);
   return result;
}
```

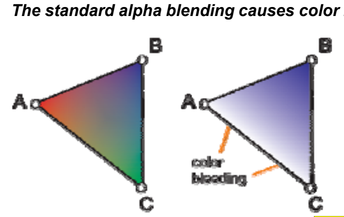
We assume here that the RGBA texture already contains emission/absoption coefficients.

Transfer functions are discussed later

Compositing

```
//alpha blending for colors pre-multiplied with opacity
glEnable(GL_BLEND);
glAlphaFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
```

The standard alpha blending causes color bleeding!



Vertex A: RGBA = (1,0,0,0)
Vertex B: RGBA = (0,0,1,1)
Vertex C: RGBA = (0,1,0,0)

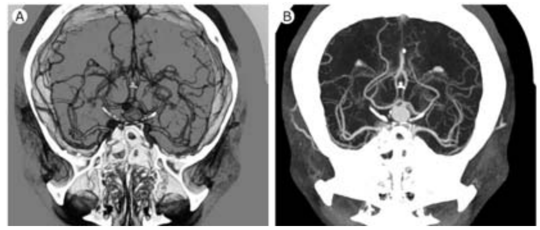
Solution: Associated Colors:
RGB values must be pre-multiplied by opacity A!

Compositing

- ***Maximum Intensity Projection***
  No emission/absorption
  Simply compute maximum value along a ray

```
#ifdef GL_EXT_blend_minmax
   //enable alpha blending
   glEnable(GL_BLEND);

   //enable maximum selection
   glBlendEquationEXT(GL_MAX_EXT);

   //setup arguments for the blending equation
   glBlendFunc(GL_SRC_COLOR, GL_DST_COLOR);
#endif
```

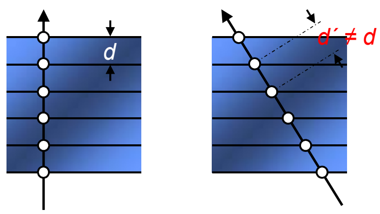
Compositing



Emission/Absorption ***Maximum Intensity Projection***

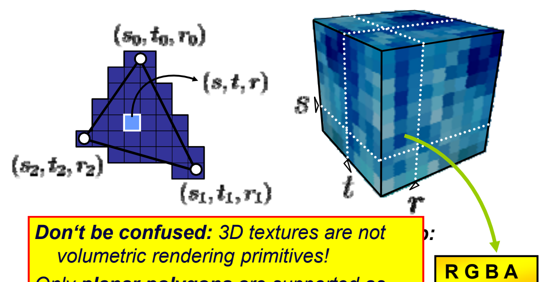
2D Textures: Drawbacks

- Sampling rate is inconsistent



- Emission/absorption slightly incorrect
- ***Super-sampling on-the-fly impossible***

3D Textures

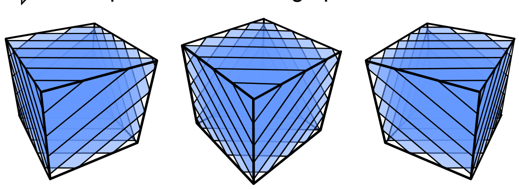


Don't be confused: *3D textures are not volumetric rendering primitives!*
*Only **planar polygons** are supported as rendering primitives.*

3D Textures

3D Texture: Volumetric Texture Object

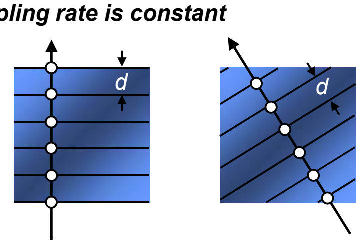
- Trilinear Interpolation in Hardware
- Slices parallel to the image plane



- One large texture block in memory

Resampling via 3D Textures

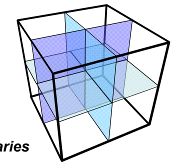
- ***Sampling rate is constant***



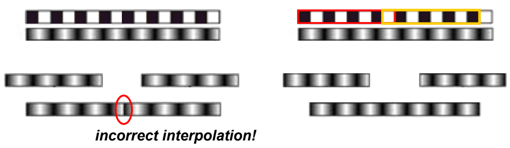
- Supersampling by increasing the number of slices

Bricking

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

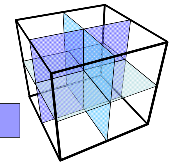


One plane of voxels must be duplicated to enable correct interpolation across brick boundaries



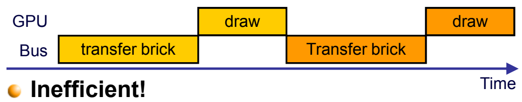
Bricking

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



Problem: **Bus-Bandwidth**

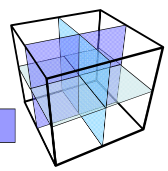
- Unbalanced Load for GPU und Memory Bus



- **Inefficient!**

Bricking

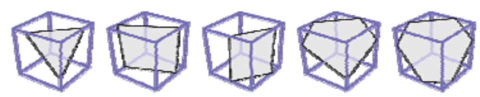
- What happens if data set is too large to fit into local video memory?
- ⇨ Divide the data set into smaller chunks (bricks)



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!***

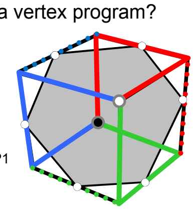
Cube-Slice Intersection



Question: Can we compute this in a vertex program?

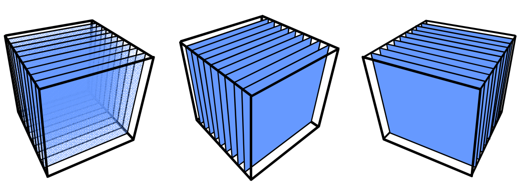
Vertex program:
Input: 6 Vertices
Output: 6 Vertices

- P0: Intersection with red path
- P1: Intersection with dotted red edge or P0
- P2: Intersection with green path
- P3: Intersection with dotted green edge or P1
- P4: Intersection with blue path
- P5: Intersection with dotted blue edge or P2



Back to 2D Textures

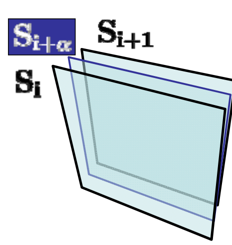
- ***~~fixed~~ number of object aligned slices***
- ***visual artifacts due to bilinear interpolation***



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

2D Multi-Textures

Axis-Aligned Slices

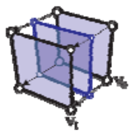


- Bilinear Interpolation by 2D Texture Unit
- Blending of two adjacent slice images

$$S_{i+\alpha} = (1-\alpha)S_i + \alpha \cdot S_{i+1}$$

- Trilinear Interpolation

Implementation



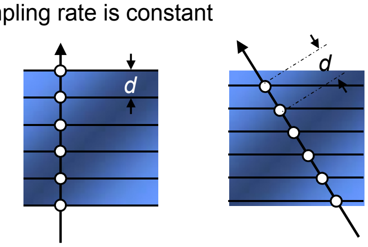
```
//vertex program for computing object aligned slices
void main( float4 Vertex0   :  POSITION,
           float4 Vertex1   :  TEXCOORD0,
           half2  TexCoord0 :  TEXCOORD1,

           uniform float    slicePos,
           uniform float4x4 matModelViewProj,

           out float4 VertexOut   :  POSITION,
           out half3  TexCoordOut :  TEXCOORD0)
{
  //interpolate between the two positions
  float4 Vertex = lerp(Vertex0, Vertex1, slicePos);

  //transform vertex into screen space
  VertexOut = mul(matModelViewProj, Vertex);

  //compute the correct 3D texture coordinate
  TexCoordOut = half3(TexCoord.xy, slicePos);

  return;
}
```

Implementation

```
//fragment program for trilinear interpolation
//using 2D multi-textures
float4 main (half3 texUV : TEXCOORD0,
             uniform sampler2D texture0,
             uniform sampler2D texture1 ) :  COLOR
{
   //two bilinear texture fetches
   float4 tex0 = tex2D(texture0, texUV.xy);
   float4 tex1 = tex2D(texture1, texUV.xy);

   //additional linear interpolation
   float4 result = lerp(tex0,tex1,texUV.z);

   return result;
}
```

2D Multi-Textures

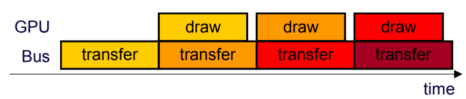
- Sampling rate is constant



- 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!
- **Significanly higher performance**, although 3 copies of the data set in main memory

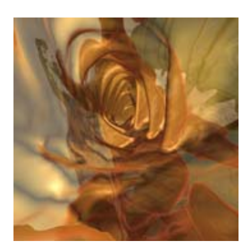
Summary

Rasterization Approaches for Direct Volume Rendering

- ***2D Texture Based Approaches***
  - 3 fixed stacks of object aligned slices
  - Visual artifacts due to bilinear interpolation only
  - No supersampling
- ***3D Texture Based Approaches***
  - Viewport aligned slices
  - Supersampling with trilinear interpolation
  - Bricking: Bus transfer inefficient for large volumes
- ***2D Texture Based Approaches***
  - 3 variable stacks of object aligned slices
  - Supersampling with Trilinear interpolation
  - Higher performance for larger volumes

Real-Time Volume Graphics

[04] GPU-Based Ray-Casting

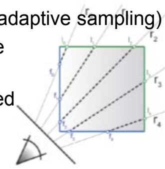


Talk Outline

- 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
  - Isosurface ray-casting
  - Endoscopic ray-casting

Why Ray-Casting on GPUs?

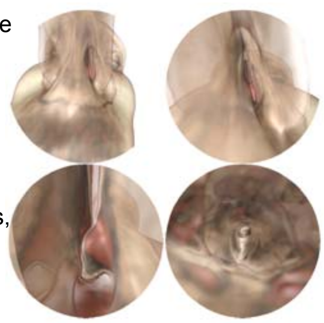
- Most GPU rendering is object-order (rasterization)
- Image-order is more "CPU-like"
  - Recent fragment shader advances
  - Simpler to implement
  - Very flexible (e.g., adaptive sampling)
  - Correct perspective projection
  - Can be implemented in single pass!
  - Native 32-bit compositing



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]
  - [Röttger et al., 2003]
  - [Green, 2004] (NVIDIA SDK Example)
  - [Stegmaier et al., 2005]
  - [Scharsach et al., 2006]

- Unstructured (tetrahedral) grids
  - [Weiler et al., 2002, 2003, 2004]
  - [Bernardon, 2004]

Single-Pass Ray-Casting

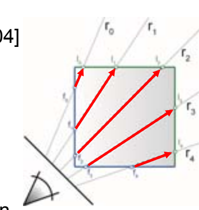
- Enabled by conditional loops in fragment shaders (Shader Model 3; e.g., Geforce 6800, ATI X1800)
- Substitute multiple passes and early-z testing by single loop and early loop exit
- No compositing buffer: full 32-bit precision!

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

Basic Ray Setup / Termination

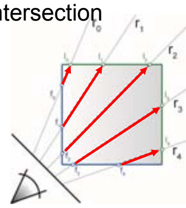
- Two main approaches:
  - Procedural ray/box intersection [Röttger et al., 2003], [Green, 2004]
  - Rasterize bounding box [Krüger and Westermann, 2003]

- 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
- 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

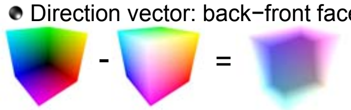
Fragment Shader

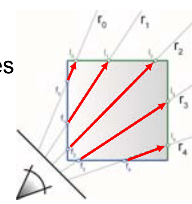
- Rasterize front faces of volume bounding box
- Texcoords are volume position in [0,1]
- Subtract camera position
- Repeatedly check for exit of bounding box

```
// 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/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)

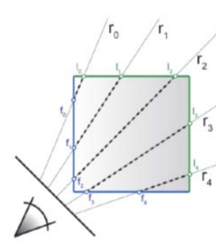
Empty space skipping

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

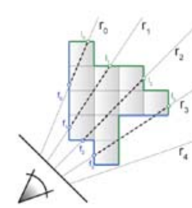
- Several possibilities
  - Per-sample check of opacity (expensive)
  - Traverse hierarchy (e.g., octree) or regular grid
  - These are image-order: what about object-order?

Object-Order Empty Space Skipping (1)

- Modify initial rasterization step



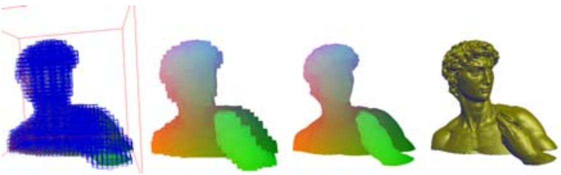
rasterize bounding box



rasterize “tight" bounding geometry

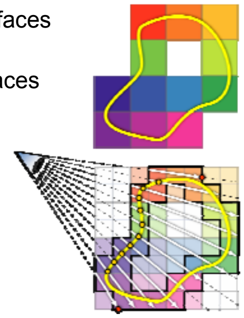
Object-Order Empty Space Skipping (2)

- Store min-max values of volume bricks
- Cull bricks against isovalue or transfer function
- Rasterize front and back faces of active bricks



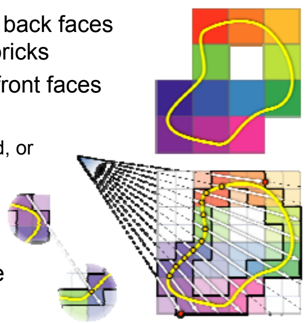
Object-Order Empty Space Skipping (3)

- 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



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
- Not all empty space is skipped

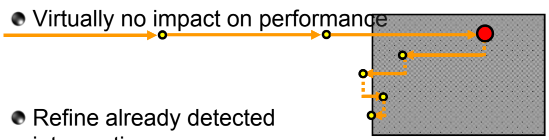


Isosurface Ray-Casting

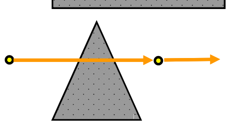
- Isosurfaces/Level Sets
  - scanned data
  - distance fields
  - CSG operations
  - level sets: surface editing, simulation, segmentation, …

Intersection Refinement (1)

- Fixed number of bisection or binary search steps
- Virtually no impact on performance

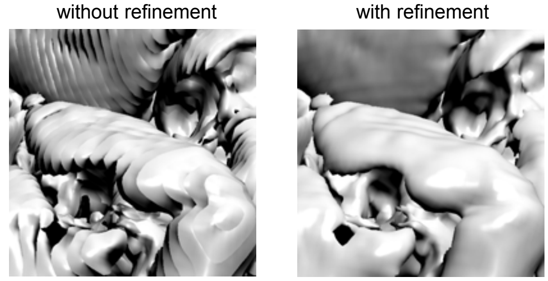


- Refine already detected intersection
- Handle problems with small features / at silhouettes with adaptive sampling



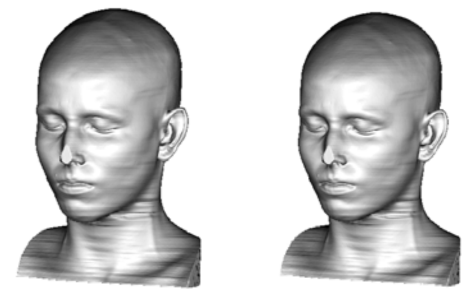
Intersection Refinement (2)

without refinement | with refinement



sampling rate 1/5 voxel (no adaptive sampling)

Intersection Refinement (3)



Sampling distance 1.0, 24 fps — Sampling distance 5.0, 66 fps

Deferred Isosurface Shading

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

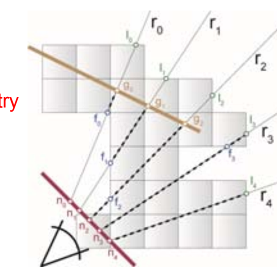
Enhancements (1)

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

- Intersect polygonal geometry

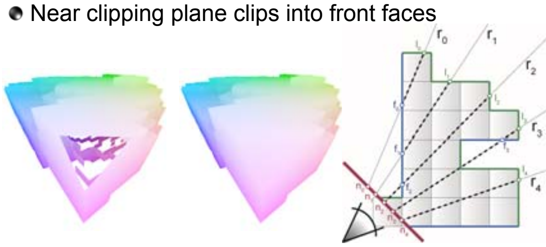
Enhancements (2)

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

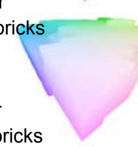


- Fill in holes with near clipping plane
- Can use depth buffer [Scharsach et al., 2006]

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



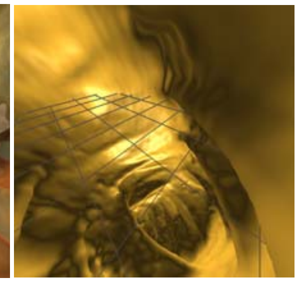
3. Rasterize nearest front faces
   - Enable depth buffer, enable color buffer
   - Rasterize *nearest front faces* of active bricks

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

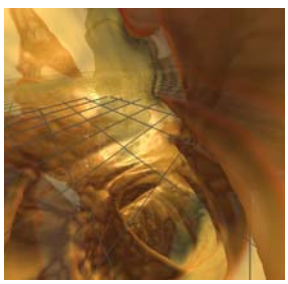
Virtual Colonoscopy

- First find isosurface; then continue with DVR



Virtual Colonoscopy

- First find isosurface; then continue with DVR

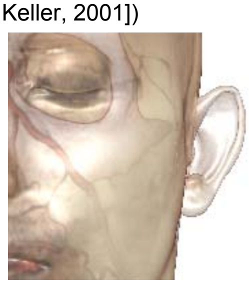


Hybrid Ray-Casting (1)

- 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

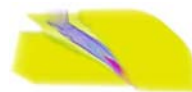
- GPU ray-casting is an attractive alternative
- Very flexible and easy to implement
- Fragment shader conditionals are very powerful; performance pitfalls very likely to go away
- Mixing image-order and object-order well suited to GPUs (vertex and fragment processing!)
- Deferred shading allows complex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

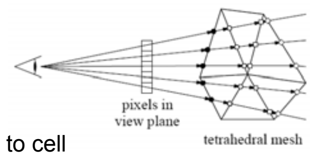
Thank You!

Acknowledgments
- Henning Scharsach, 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



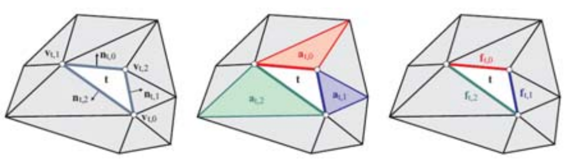


[Weiler et al.]

- Propagate from cell to cell
- Ray/face intersection computations
- Pre-integration; (store current pos in texture)

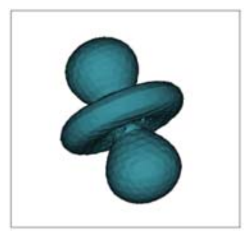
Tetrahedral Grids

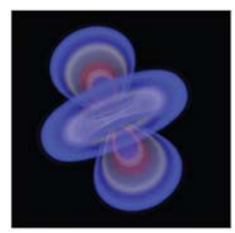
- fd



Tetrahedral Grids

- dfd



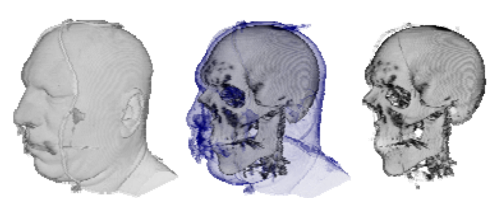


Real-Time Volume Graphics

[05] Transfer Functions

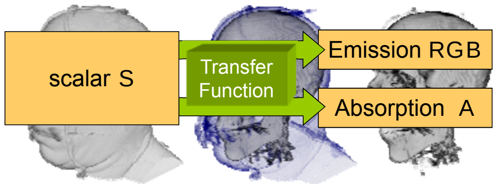
Classification

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

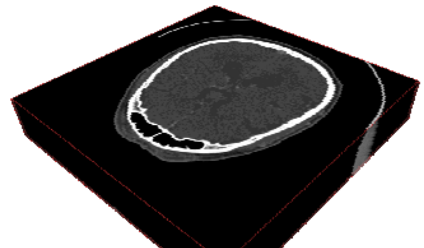


Classification

- 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.***

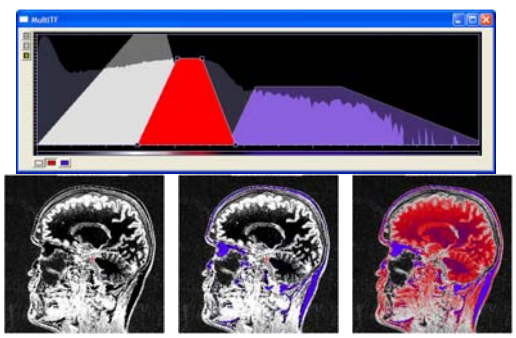


Classification

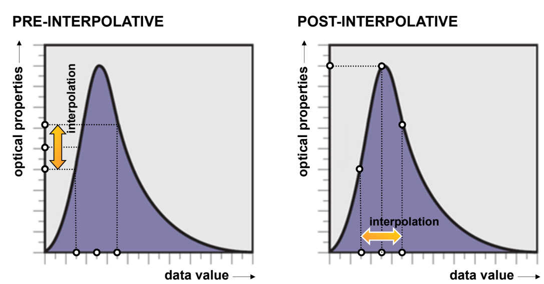


Real-Time update of the transfer function necessary!!!

Classification



Pre- vs Post-Interpolative Classification



Pre-Classification

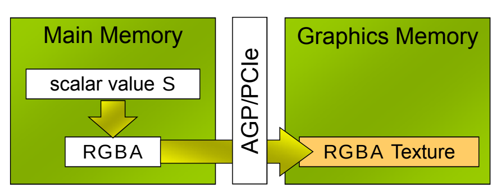
- Pre-Classification:

  Color table is applied before interpolation.
  (<u>pre-interpolative Transferfunction</u>)

- A color value is fetched from a table **<u>for each Voxel</u>**
- A RGBA Value is determined **<u>for each Voxel</u>**

Possible Implementations

- **<u>The *naive* Approach:</u>**
  Save Emission and Absorption terms directly in the Texture.

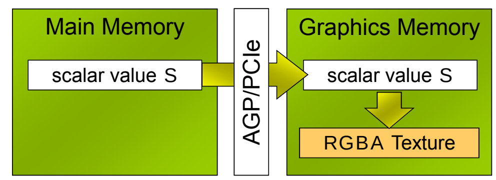


Possible Implementations

- **The *naive* Approach:**
  Save Emission and Absorption terms directly in the Texture.
- Very high memory consumption
  - Main Memory (RGBA und 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)

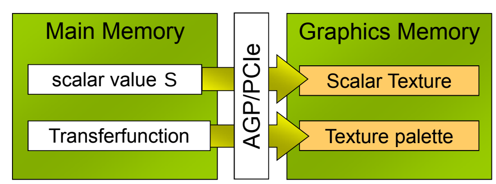


Possible Implementations

- **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!



Possible Implementations

- **The best approach:** Paletted Textures
  Store the scalar volume together with the color table directly in graphics memory.
- Hardware-Support necessary!
- Low memory consumption
  - Main Memory (scalar volume can be deleted!)
  - Graphics Memory (scalar volume + TF)
- Low load on memory bus
  - Scalar volume must be transferred only once!
- **Only the color table must be re-uploaded on TF change**

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

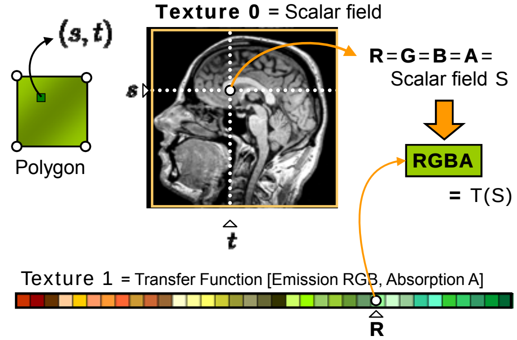
Post-Classification

- Post-Classification:

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

- A color is fetched from the color table ***for each Fragment***

Post-Classification

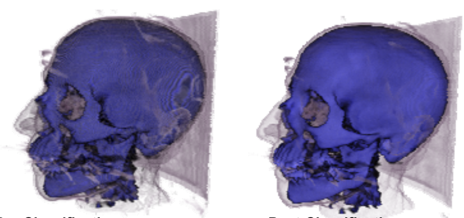


CG Implementation

```
//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

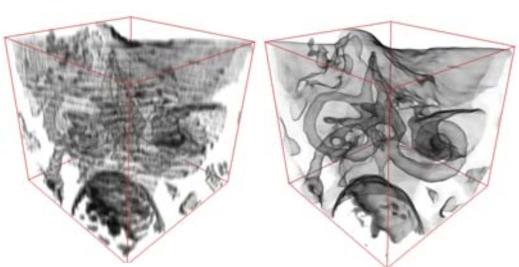
- Comparison of image quality



Pre-Classification **Post-Classification**

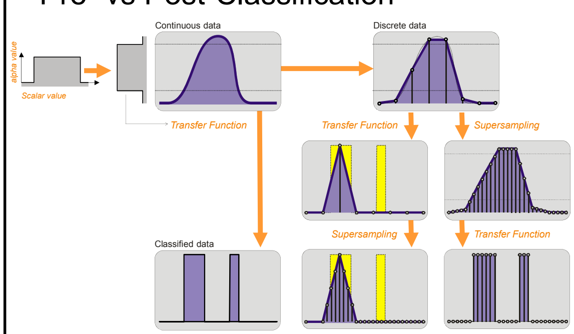
Same TF, same Resolution, same Sampling Rate

Quality

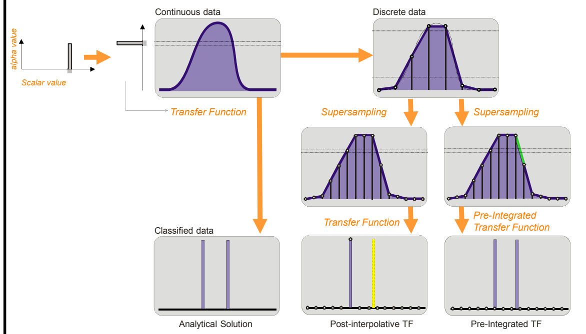


Pre-Classification **Post-Classification**

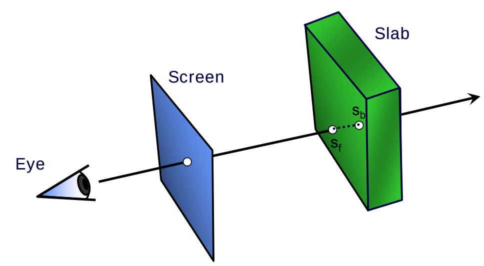
Pre- vs Post-Classification



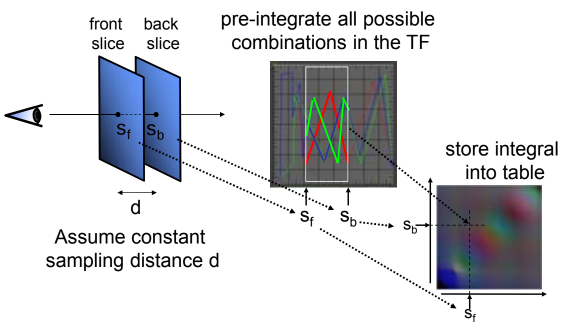
Post- vs Pre-Integrated Classification



Classification Artifacts / Pre-integration



Pre-Integrated Classification



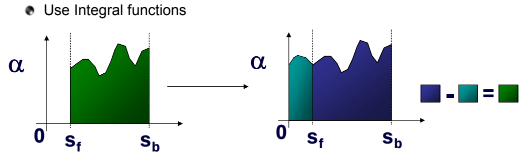
Classification Artifacts / Pre-integration

Cg Fragment Program

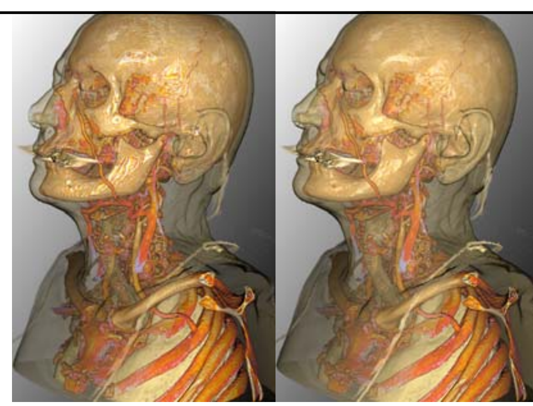
```
struct v2f_simple {
    float4 Hposition : POSITION;
    float3 TexCoord0 : TEXCOORD0;
    float3 TexCoord1 : TEXCOORD1;
float4 Color0 : COLOR0;
};

float4 main(v2f_simple IN,
            uniform sampler3D Volume,
            uniform sampler2D TransferFunction,
            uniform sampler2D PreIntegrationTable) : COLOR
{
    float4 lookup;
    //sample front scalar
    lookup.x = tex3D(Volume, IN.TexCoord0.xyz).x;
    //sample back scalar
    lookup.y = tex3D(Volume, IN.TexCoord1.xyz).x;

    //lookup and return pre-integrated value
    return tex2D(PreIntegrationTable, lookup.yx);
}
```

Pre-Integrated Classification

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



  - Hardware-Accelerated Computation: Roettger, Ertl. **A Two-Step Approach for Interactive Pre-Integrated Volume Rendering of Unstructured Grids**. In *Proc. VolVis '02*



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