Real-Time Visualization

02: Real-Time Volume Graphics 1

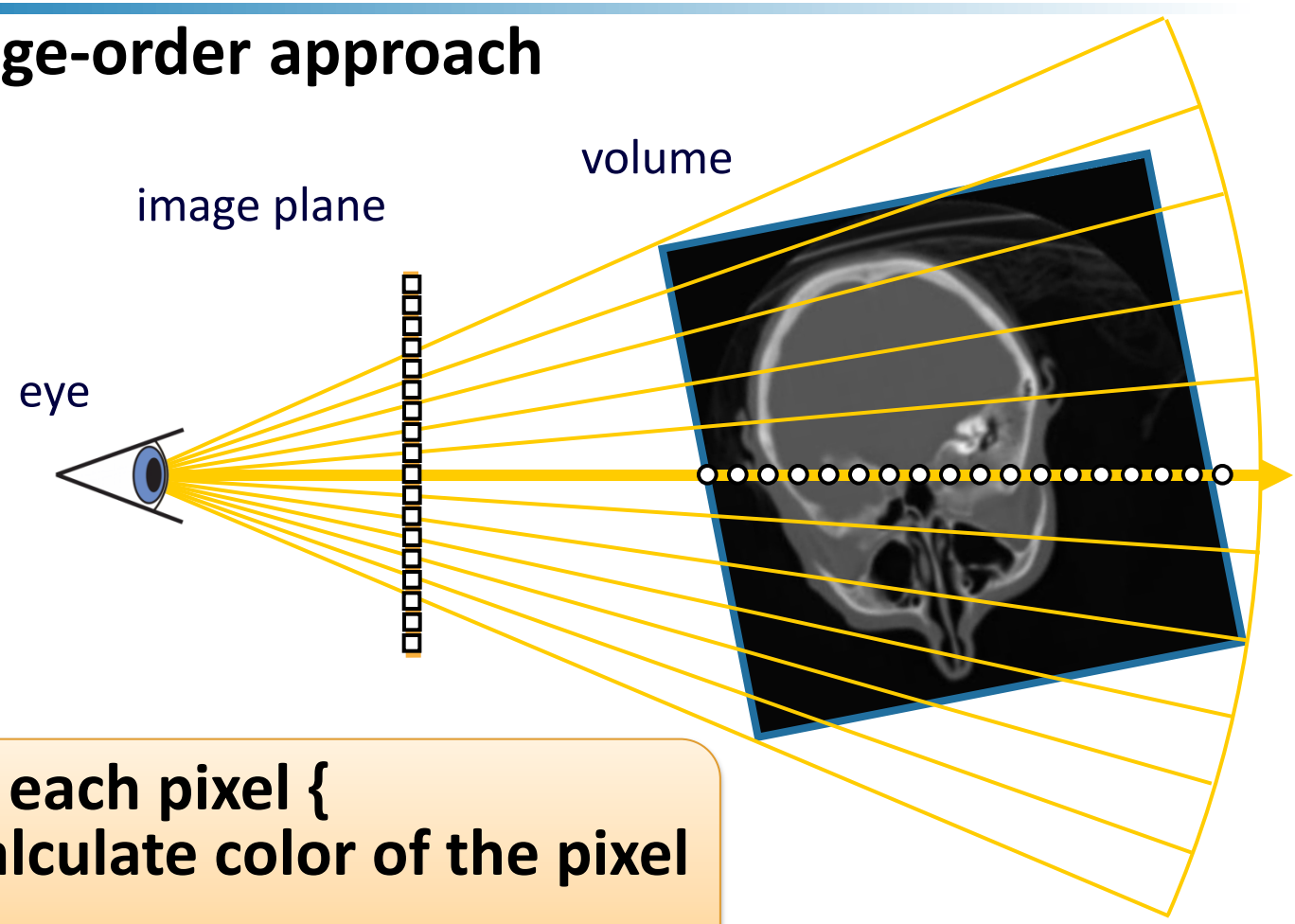
Stefan Bruckner

Lectures

1. **Introduction:** Volume rendering basics, GPU architecture, OpenGL, CUDA, case studies, … (03.05.)
2. **Real-Time Volume Graphics 1:** GPU ray-casting, optimizations, memory management, OpenGL vs. CUDA, … (10.05.)
3. **Real-Time Volume Graphics 2:** Advanced illumination, filtering, derivatives, advanced transfer functions, … (17.05.)
4. **Real-Time Computations on GPUs:** Fluid simulation, level-set deformation, … (24.05.)
5. **Volumetric Special Effects:** combining simulation and ray-casting, integration in game engine scenes, … (31.05.)
6. **Final event:** Project presentations (28.06.)

Volume Rendering (1)

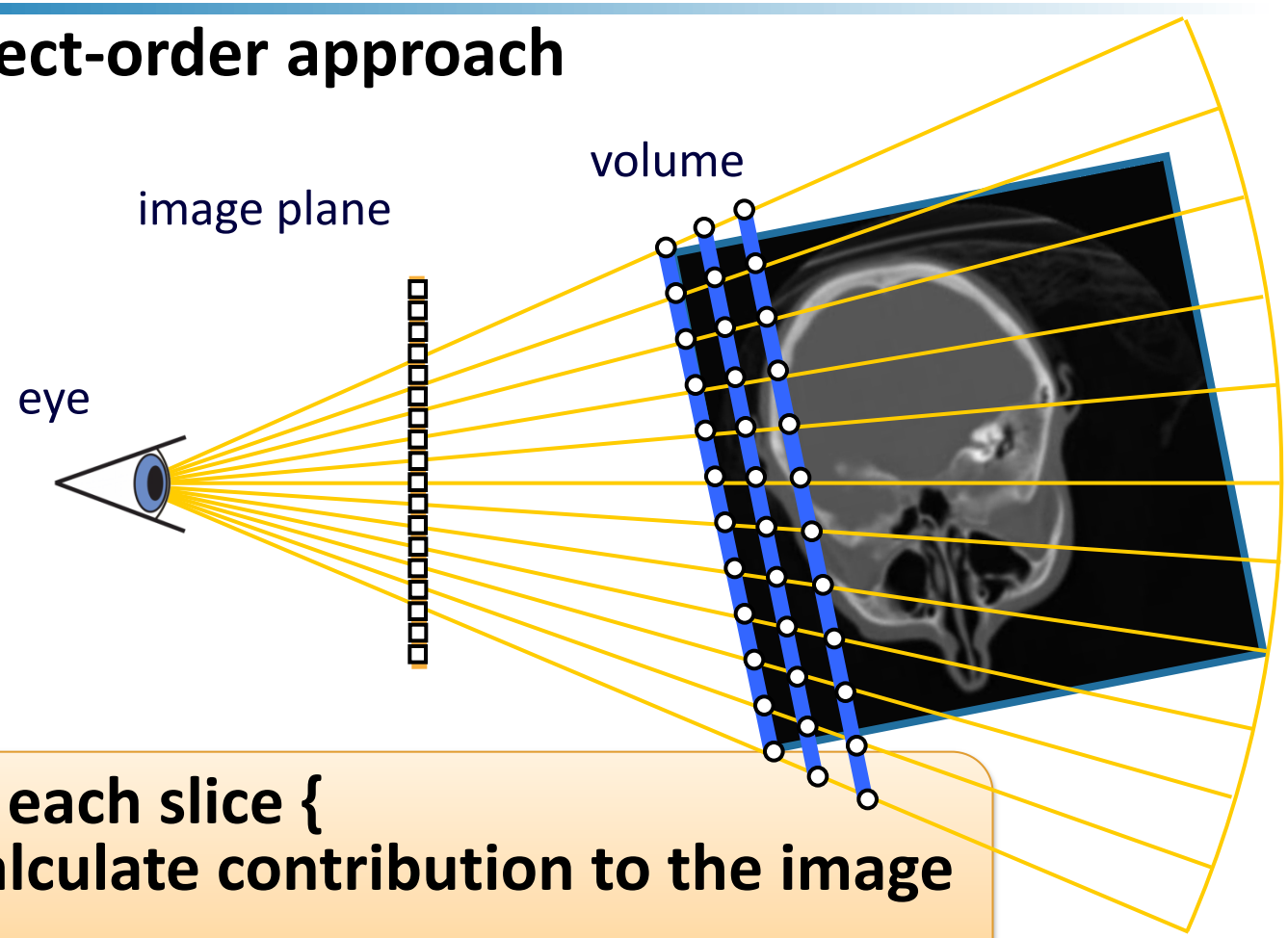
Image-order approach




```
For each pixel {
   calculate color of the pixel
}
```

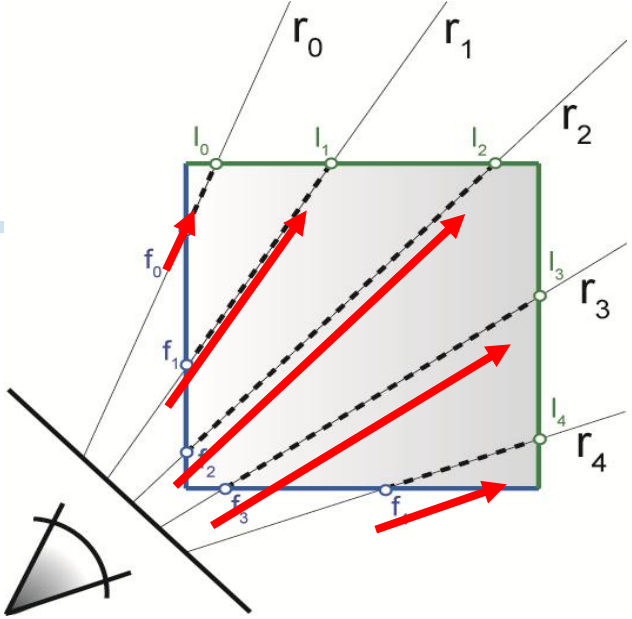
Volume Rendering (2)

Object-order approach

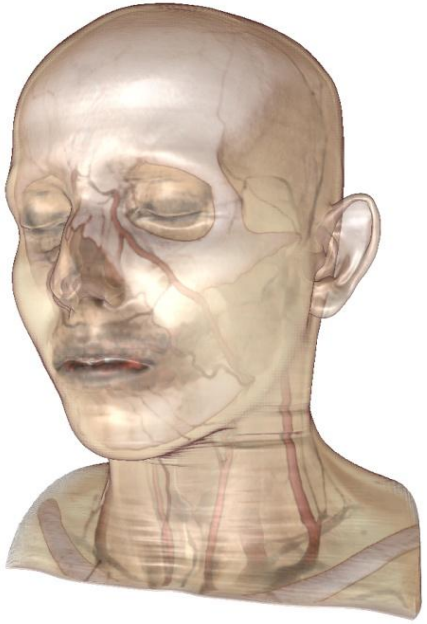



```
For each slice {
  calculate contribution to the image
}
```

Approaches: Ray-Casting

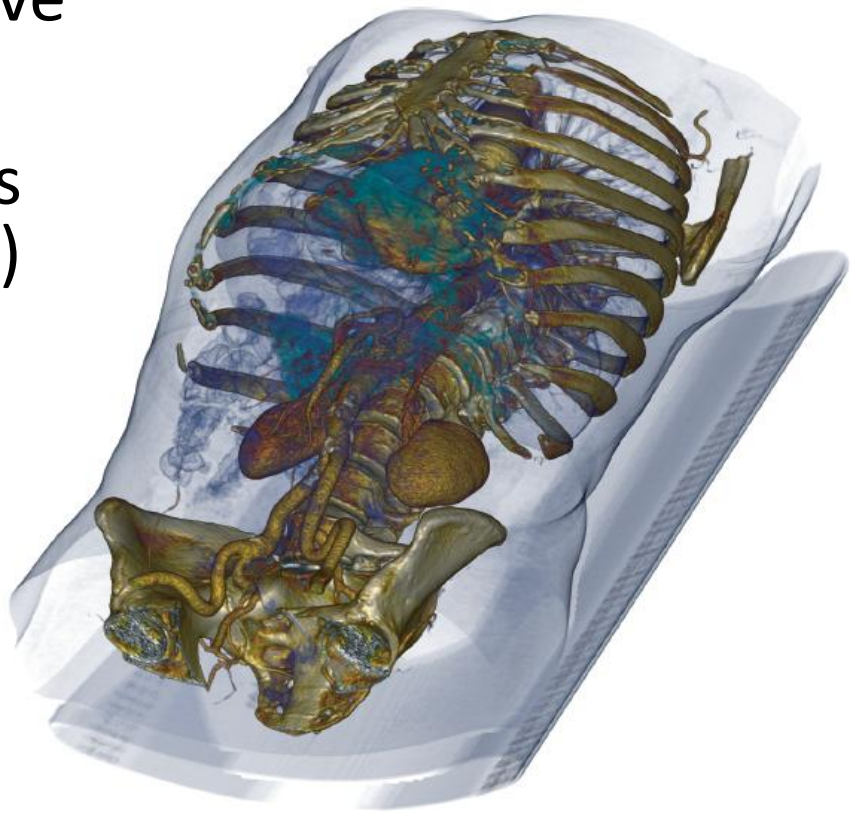


- "Natural" volume rendering method
- Image-order approach
  - Most common CPU approach
  - Well-supported by current GPUs!
- Standard optimizations
  - Early ray termination
  - Empty space skipping
- Many possibilities
  - Adaptive sampling
  - Realistic lighting



Approaches: CPU-Based Ray-Casting

- Most widely supported
- Needs multiple cores for interactive performance
- Easy to support large volume sizes (only dependent on CPU memory)
- No CPU-GPU bus transfers
- Interactive implementations usually orthogonal projection
- Exception: isosurface ray-casting (e.g., virtual endoscopy); no DVR



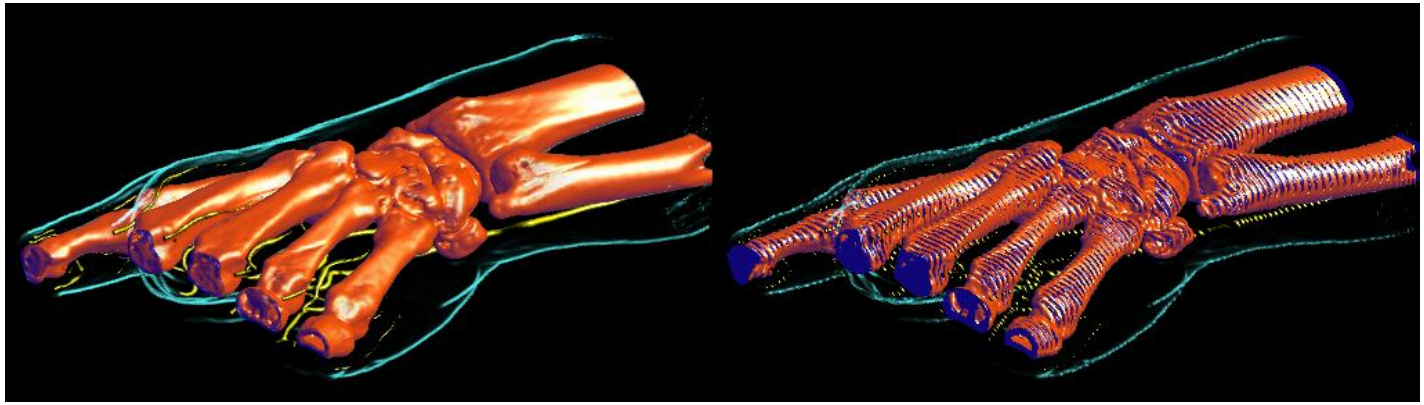
[Grimm et al., 2005]

Approaches: GPU-Based Ray-Casting

- Easy to implement; real-time!
- Very flexible (adaptive sampling, mixing of isosurface ray-casting and DVR ray-casting, ...)
- Exploits loops in fragment shader (Shader Model 3.0, ...)
- Early ray termination trivial
- Empty space skipping
- Perspective projection
- Complex lighting possible

Approaches: Texture Slicing

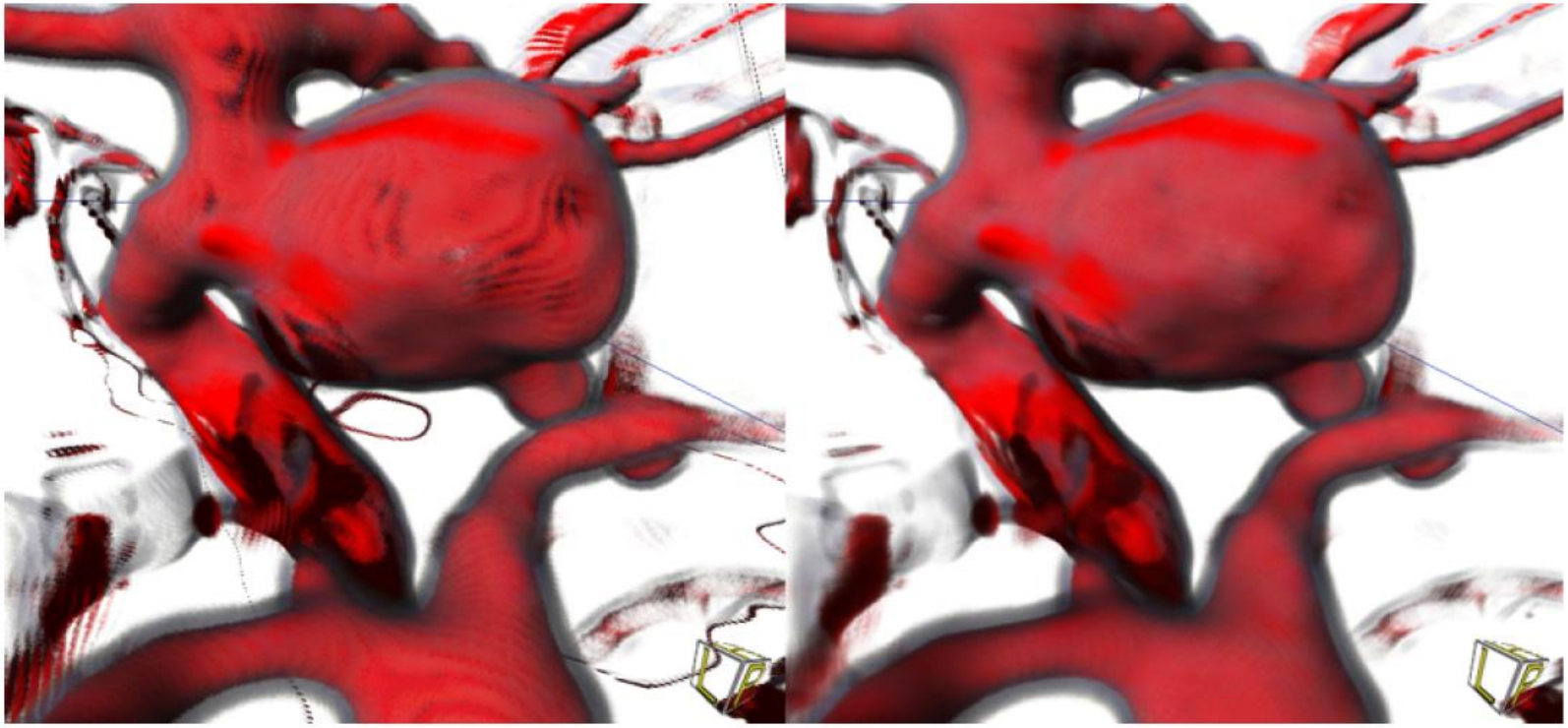
- Object-order approach
- Composite textured (viewport-aligned) slices / 3D texture
- Standard approach on pre-Shader Model 3.0 GPUs
- Not as flexible as ray-casting
- Optimizations possible, but complicated to implement (empty space skipping, early ray termination, ...)



Approaches: Splatting

- Example: neurovascular data
- Very sparse: few splats

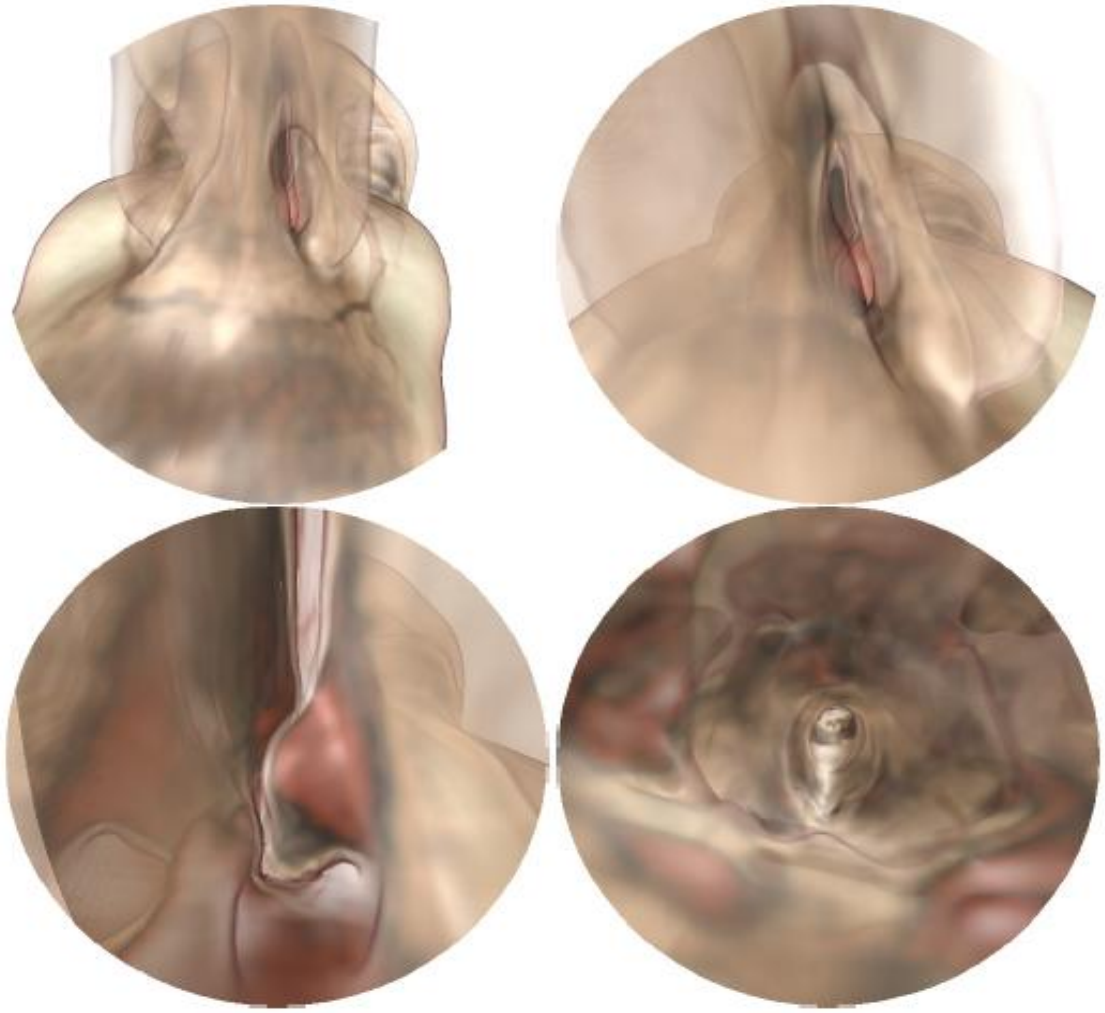
slicing (left) vs. splatting (right)



Where Is Correct Perspective Needed?

- Entering the volume
- Wide field of view

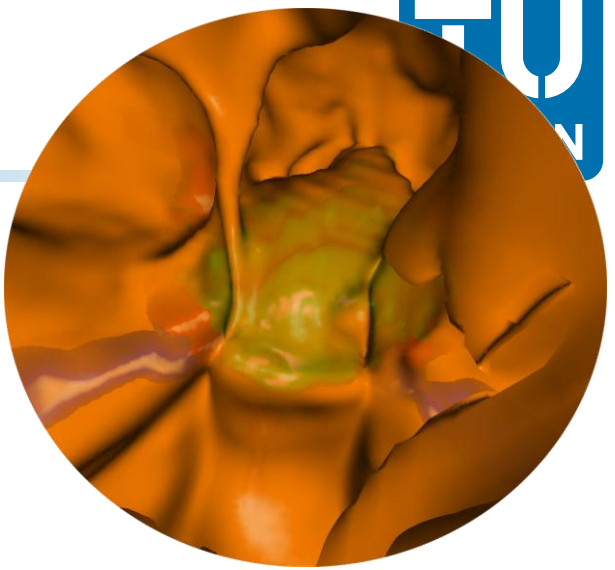
- Fly-throughs
- Virtual endoscopy
- Integration into perspective scenes, e.g., games



Virtual Endoscopy

Ray-casting

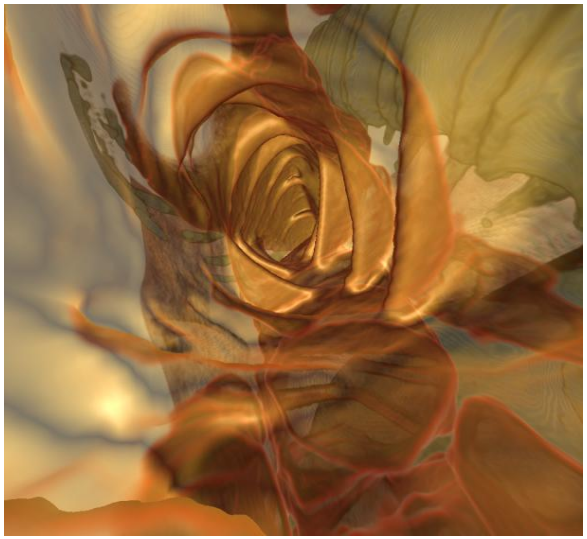
- Isosurface plus backround objects (segmentation required)
- Isosurface plus full DVR (no segmentation required)



[Neubauer et al., 2004]

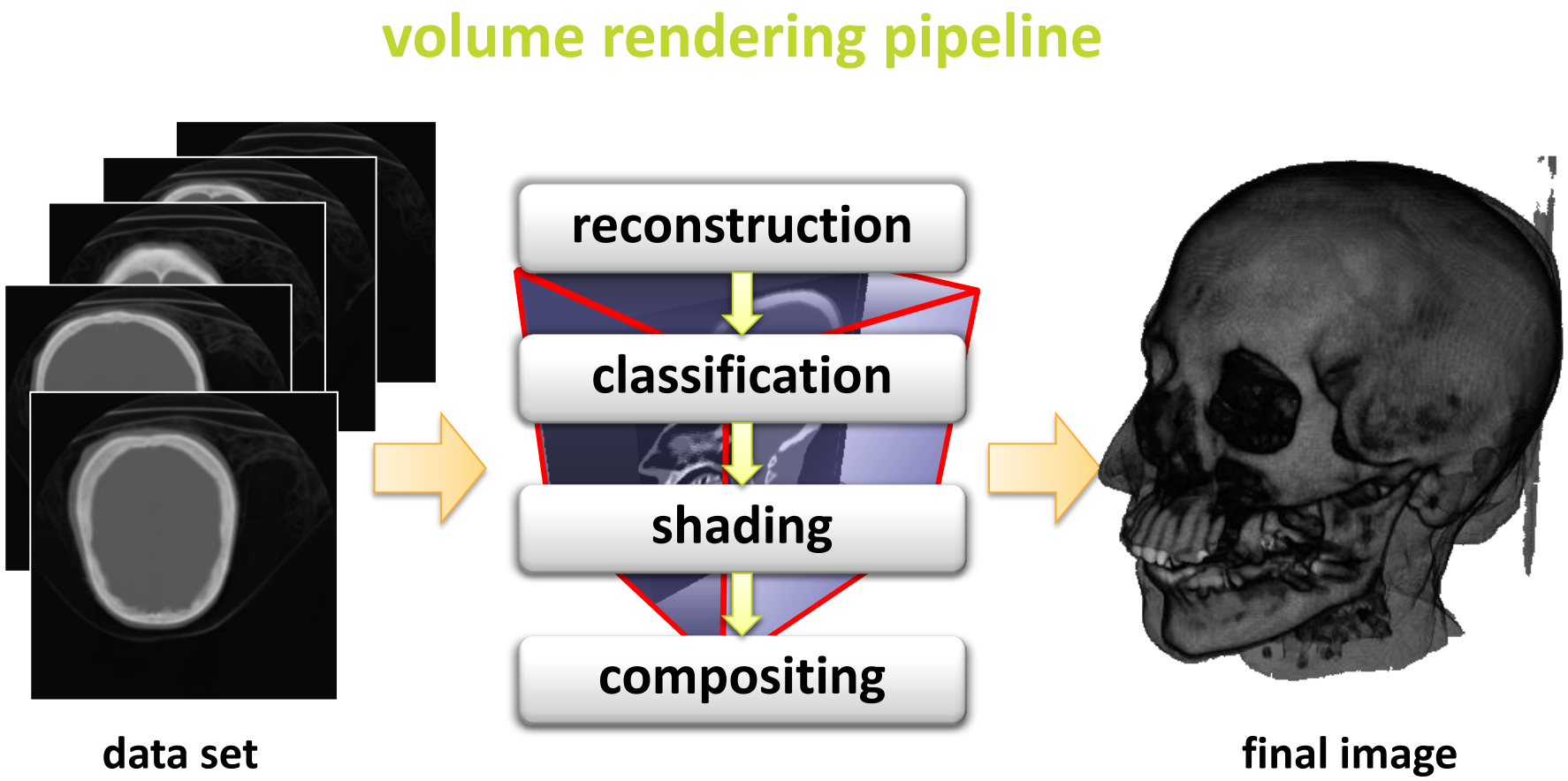
GPU-based ray-casting enables full real-time rendering, changing iso-value, transfer function, ...

Combine semi-transparent isosurface with DVR in single rendering pass



[Scharsach et al., 2006]

Volume Rendering Pipeline



Reconstruction (1)

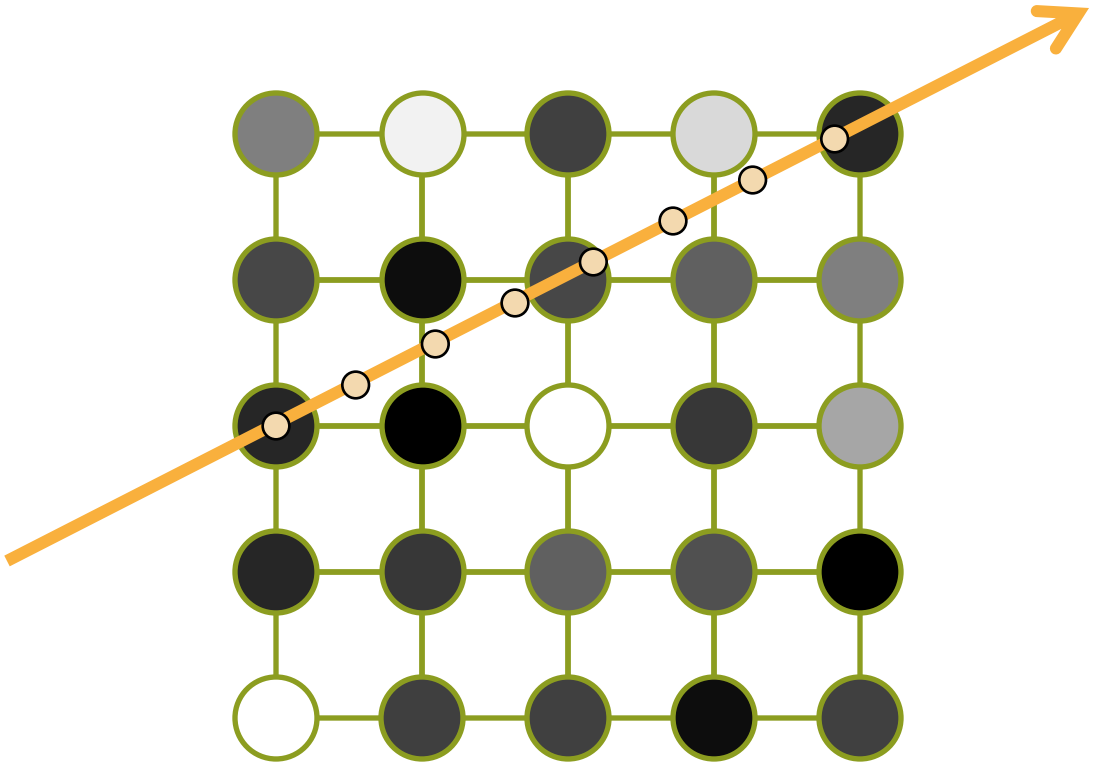
Usually volume data sets are given as a grid of discrete samples

- For rendering purposes, we want to treat them as continuous three-dimensional functions

We need to choose an appropriate reconstruction filter

- Requirements: high-quality reconstruction, but small performance overhead

Reconstruction (2)



Trilinear Interpolation

Simple extension of linear interpolation to three dimensions

Advantage: current GPUs automatically do trilinear interpolation of 3D textures

$$
\begin{aligned}
v_p = \; & v_{000}(1-x_p)(1-y_p)(1-z_p)+ \\
& v_{100}x_p(1-y_p)(1-z_p)+ \\
& v_{010}(1-x_p)y_p(1-z_p)+ \\
& v_{001}(1-x_p)(1-y_p)z_p+ \\
& v_{011}(1-x_p)y_pz_p+ \\
& v_{101}x_p(1-y_p)z_p+ \\
& v_{110}x_py_p(1-z_p)+ \\
& v_{111}x_py_pz_p
\end{aligned}
$$

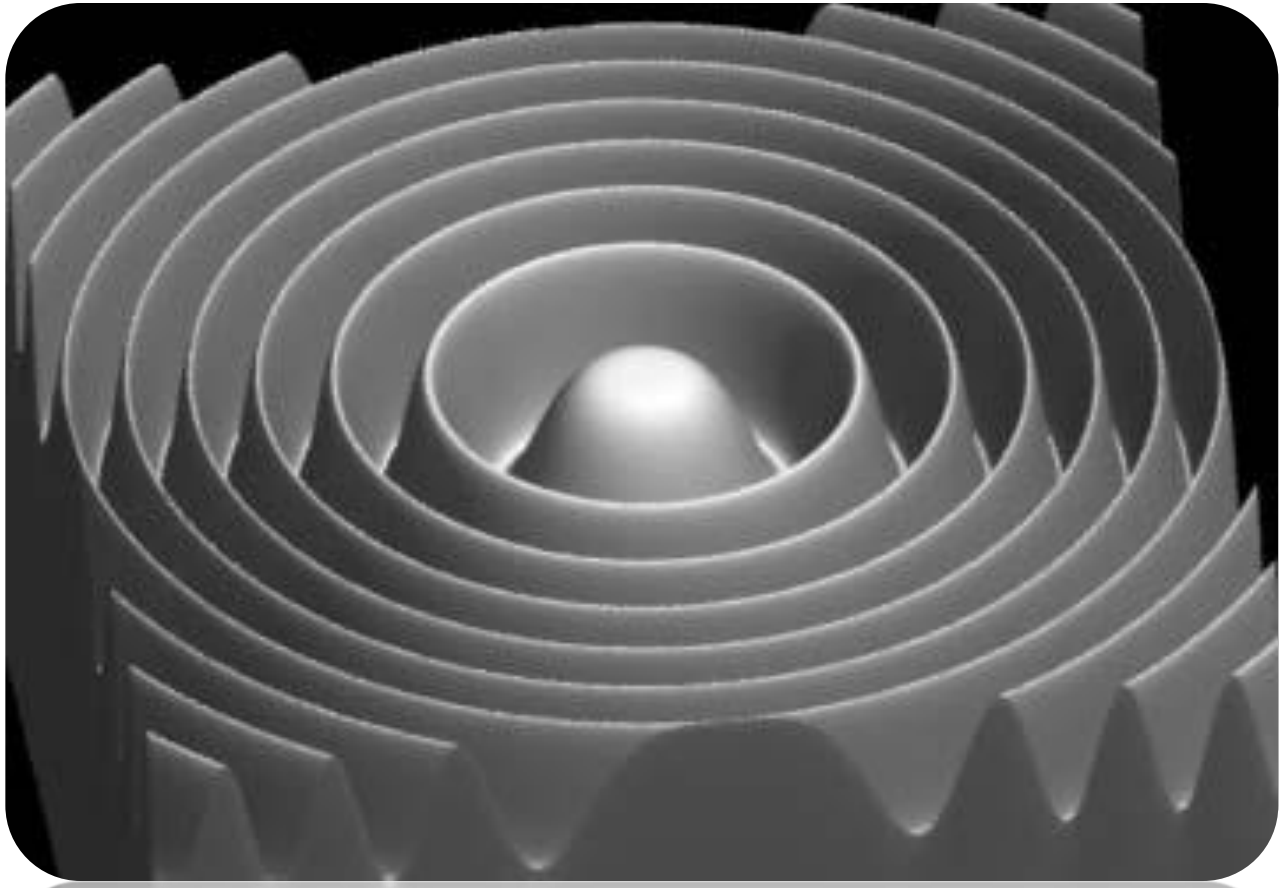
Other Reconstruction Filters

If very high quality is required, more complex reconstruction filters may be required

- Marschner-Lobb function is a common test signal to evaluate the quality of reconstruction filters [Marschner and Lobb 1994]
- The signal has a high amount of its energy near its Nyquist frequency
- Makes it a very demanding test for accurate reconstruction

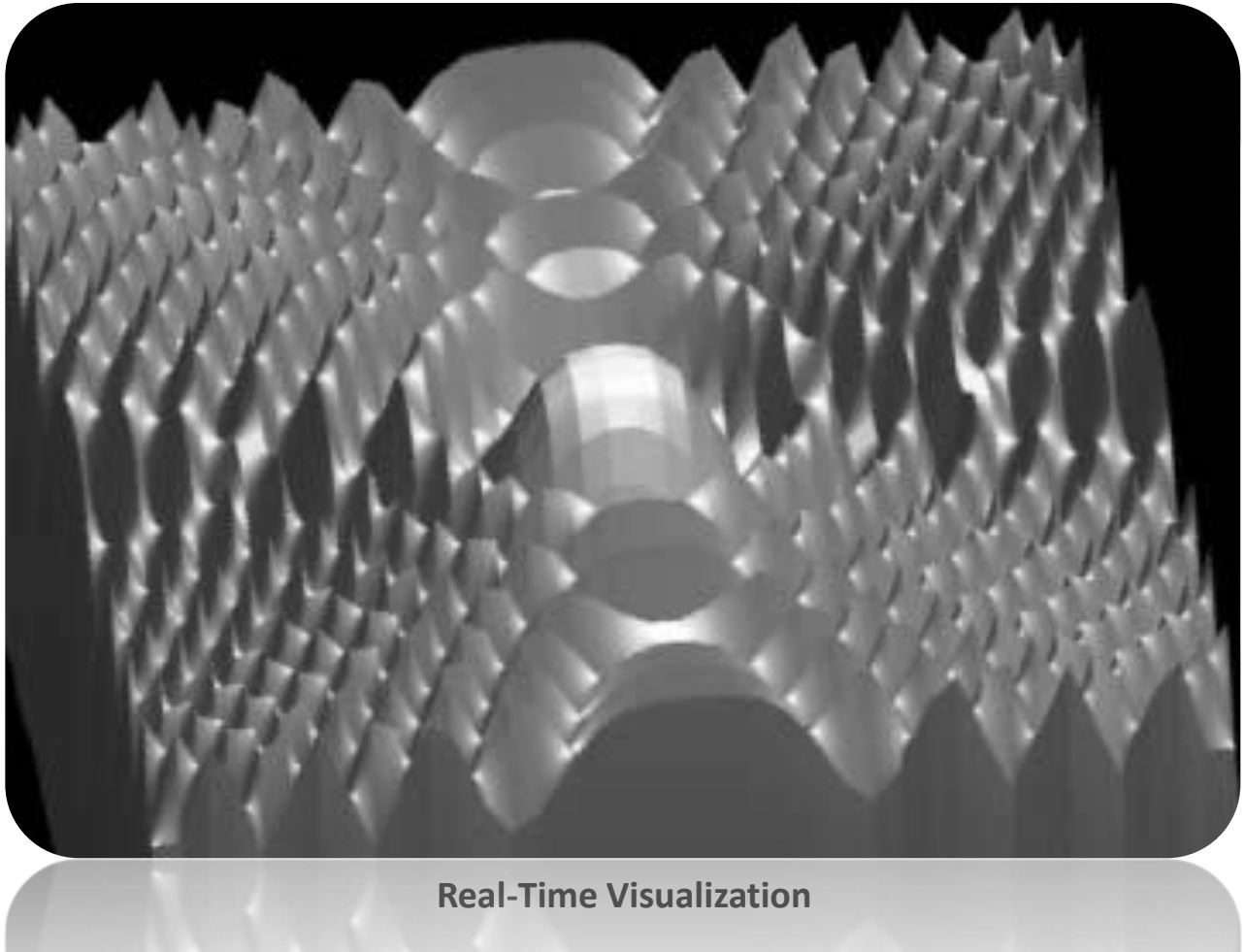
Comparison of Reconstruction Filters (1)

Marschner-Lobb test signal (analytically evaluated)



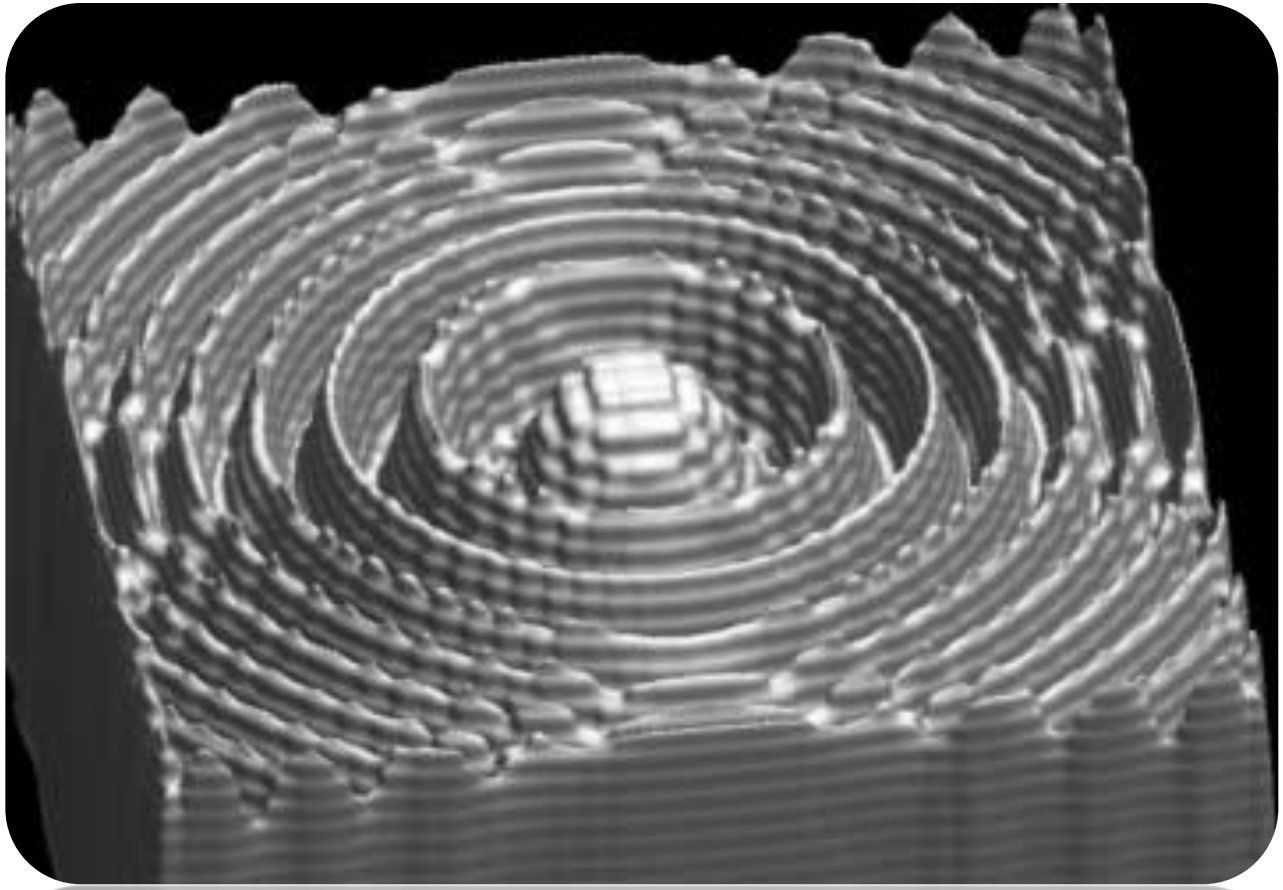
Comparison of Reconstruction Filters (2)

Trilinear reconstruction of Marschner-Lobb test signal



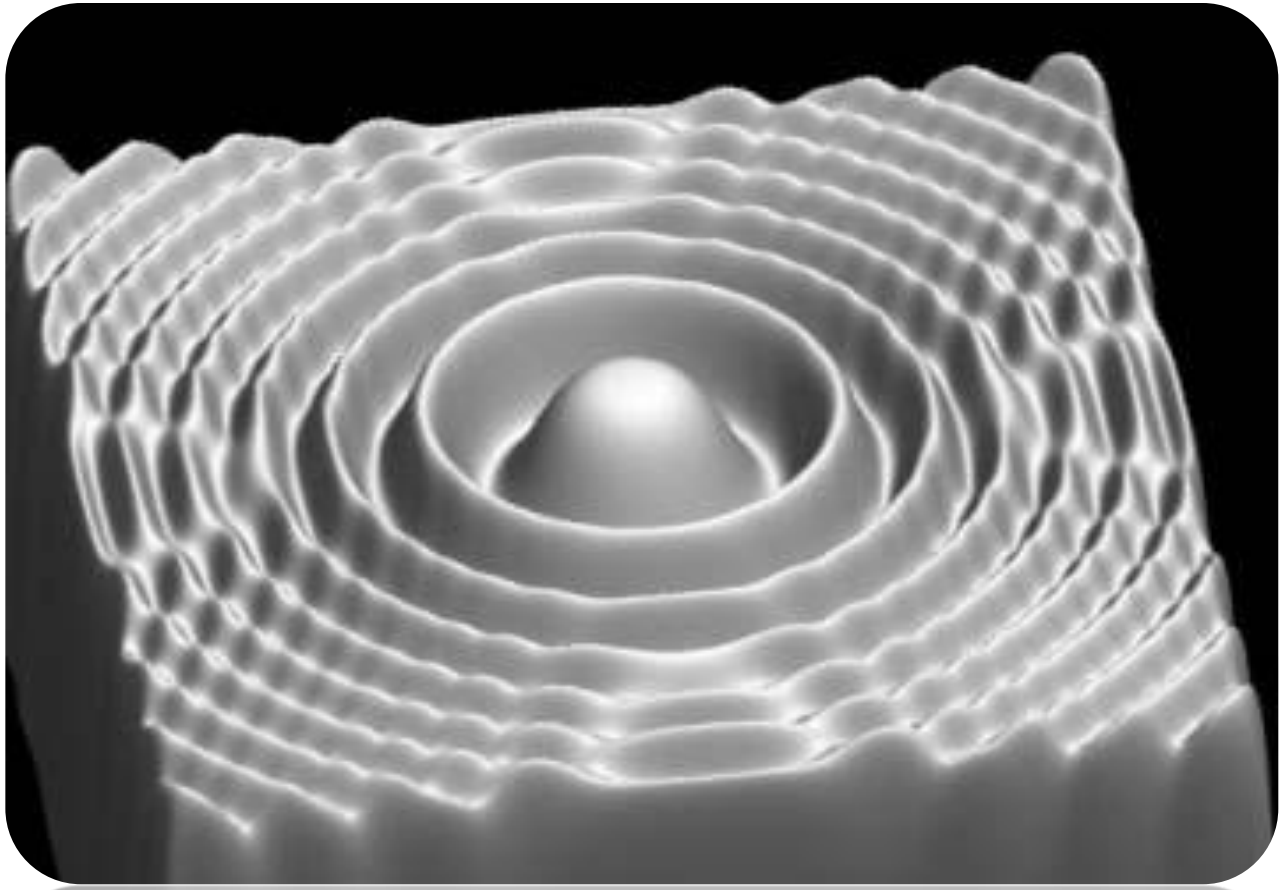
Comparison of Reconstruction Filters (3)

Cubic reconstruction of Marschner-Lobb test signal



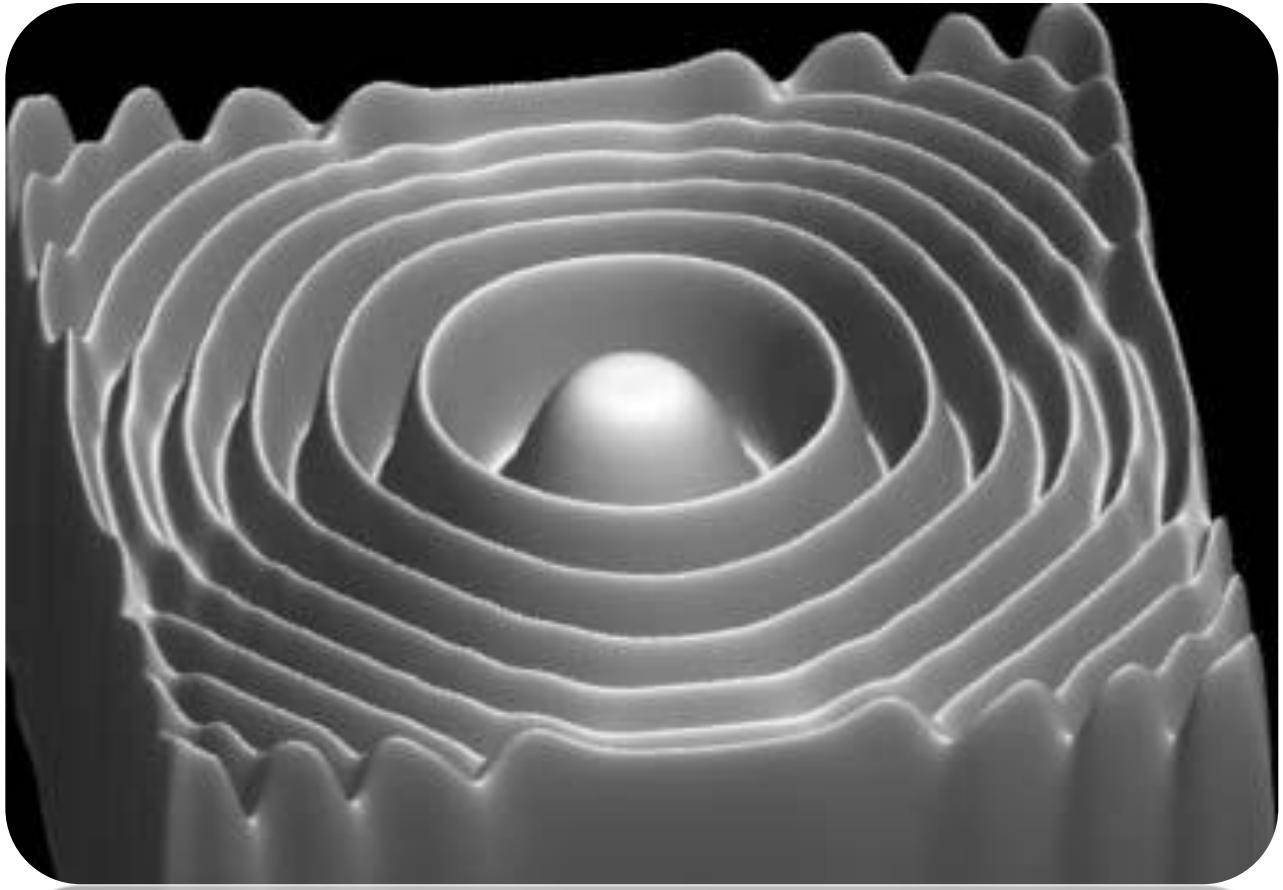
Comparison of Reconstruction Filters (4)

B-Spline reconstruction of Marschner-Lobb test signal



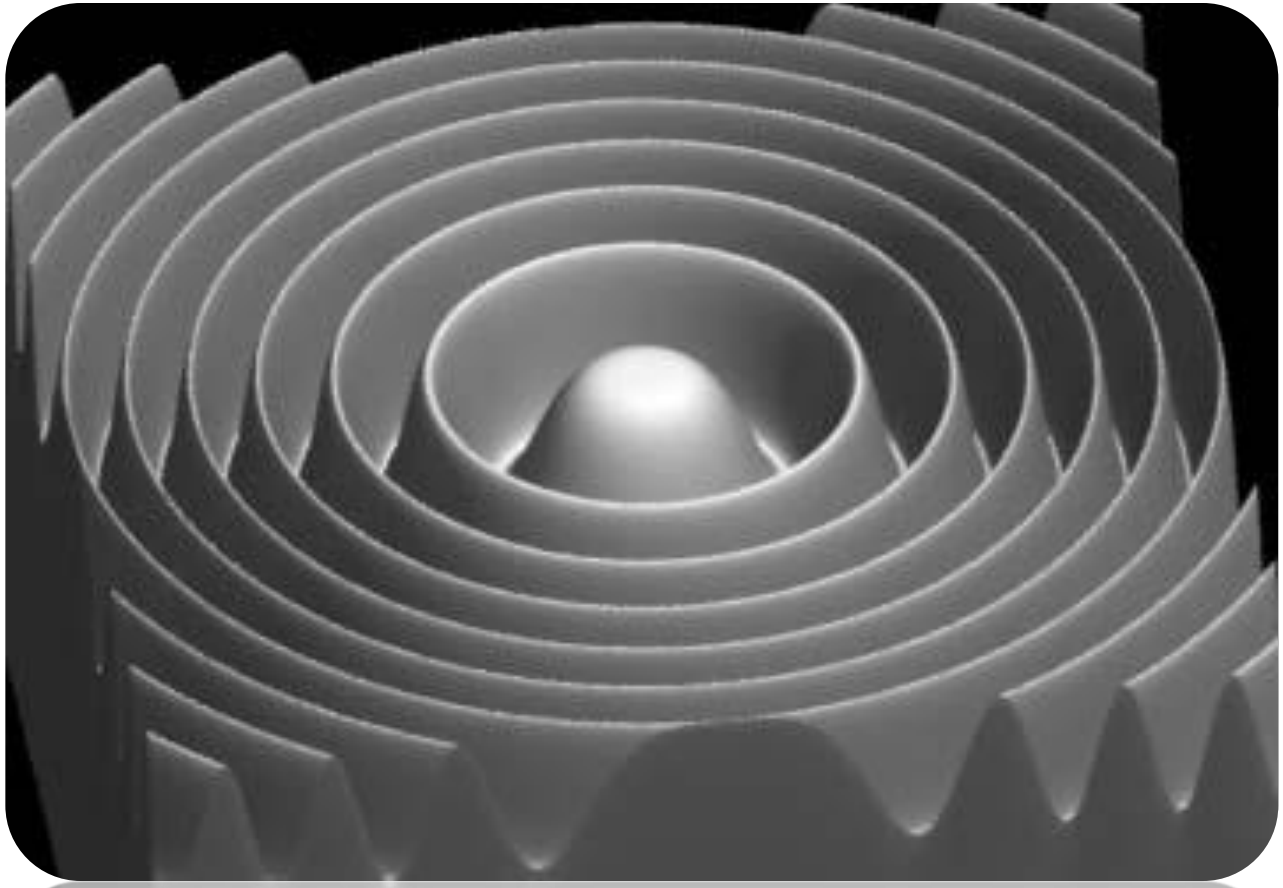
Comparison of Reconstruction Filters (5)

Windowed sinc reconstruction of Marschner-Lobb test signal



Comparison of Reconstruction Filters (6)

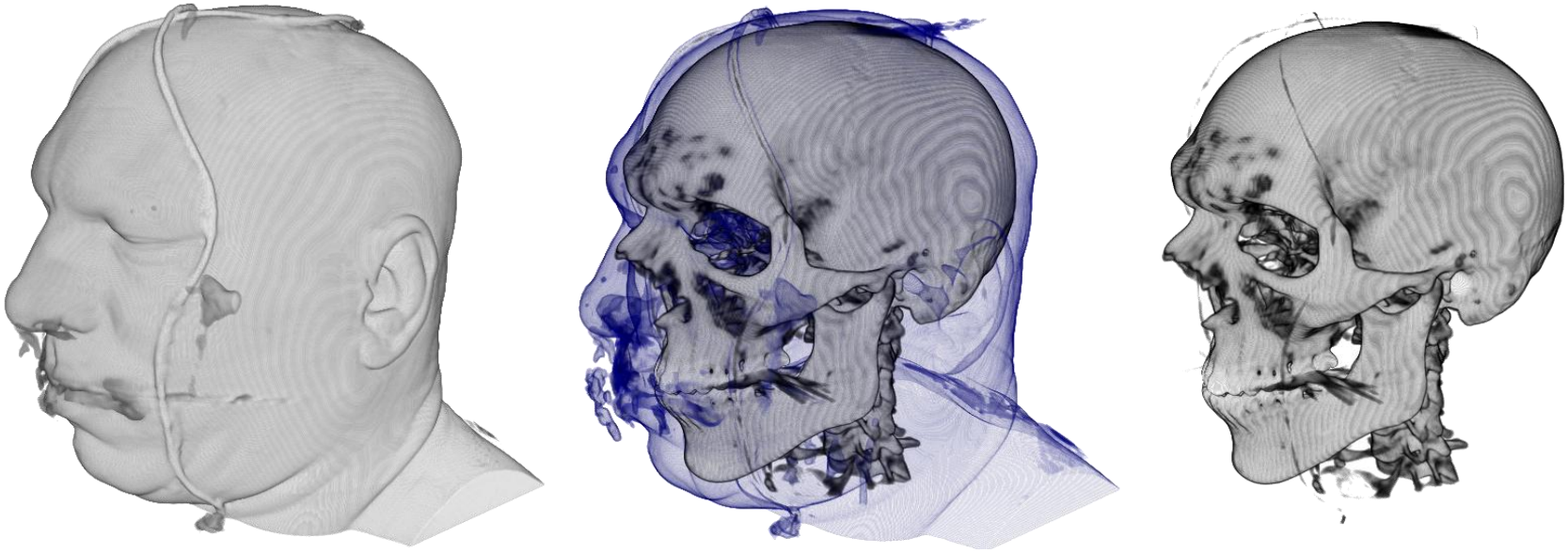
Marschner-Lobb test signal (analytically evaluated)



Classification (1)

During classification the user defines the appearence of the data

- Which parts are transparent?
- Which parts have which color?

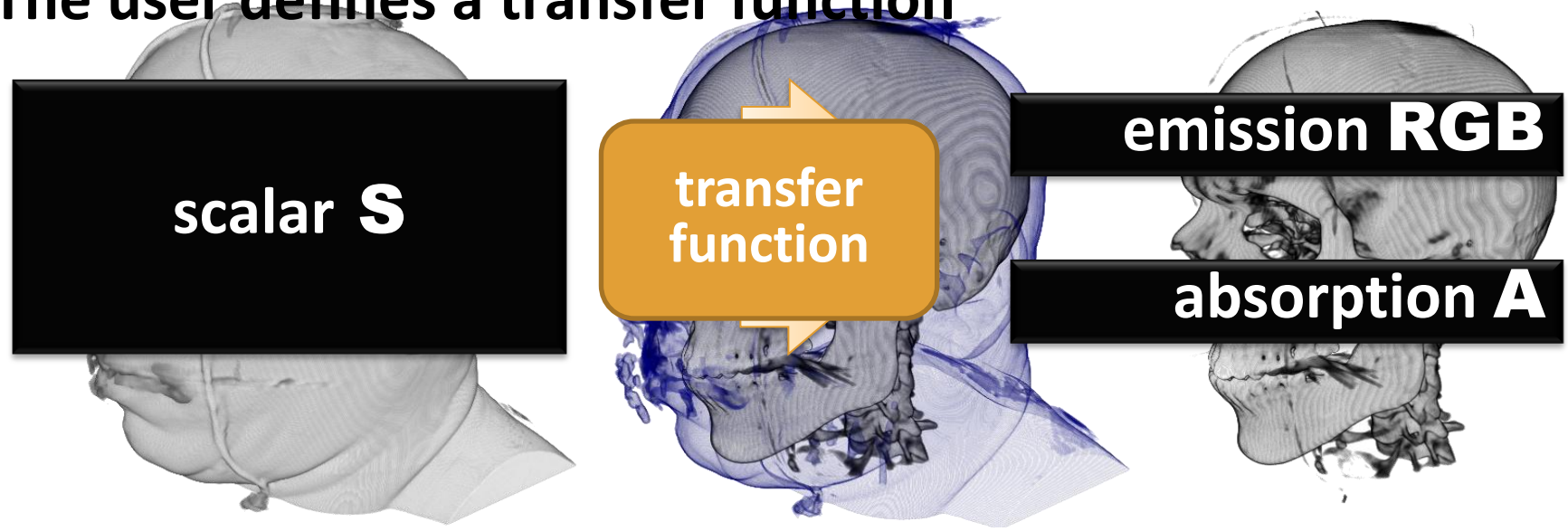


Classification (2)

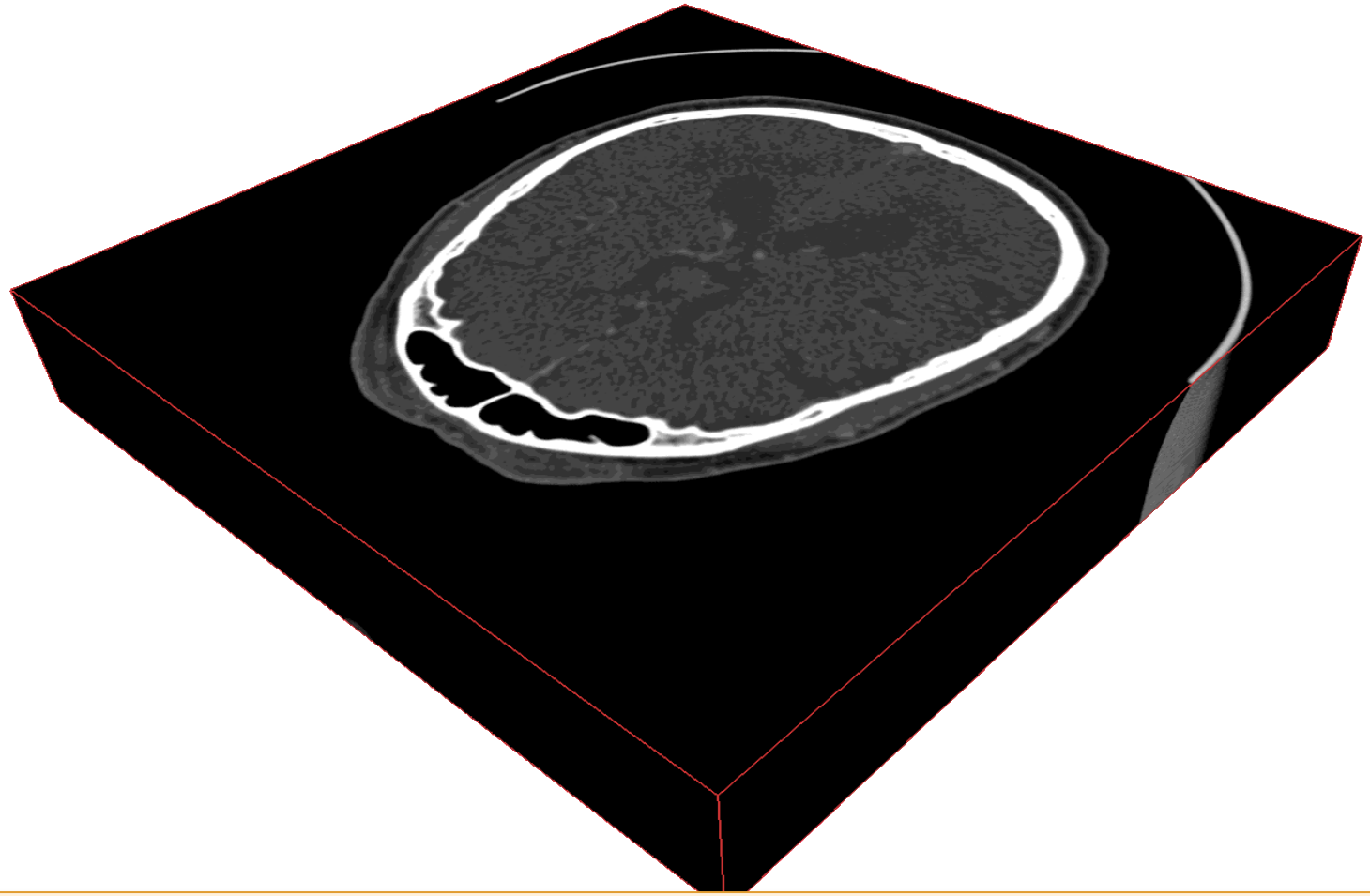
During classification the user defines the appearence of the data

- Which parts are transparent?
- Which parts have which color?

The user defines a transfer function

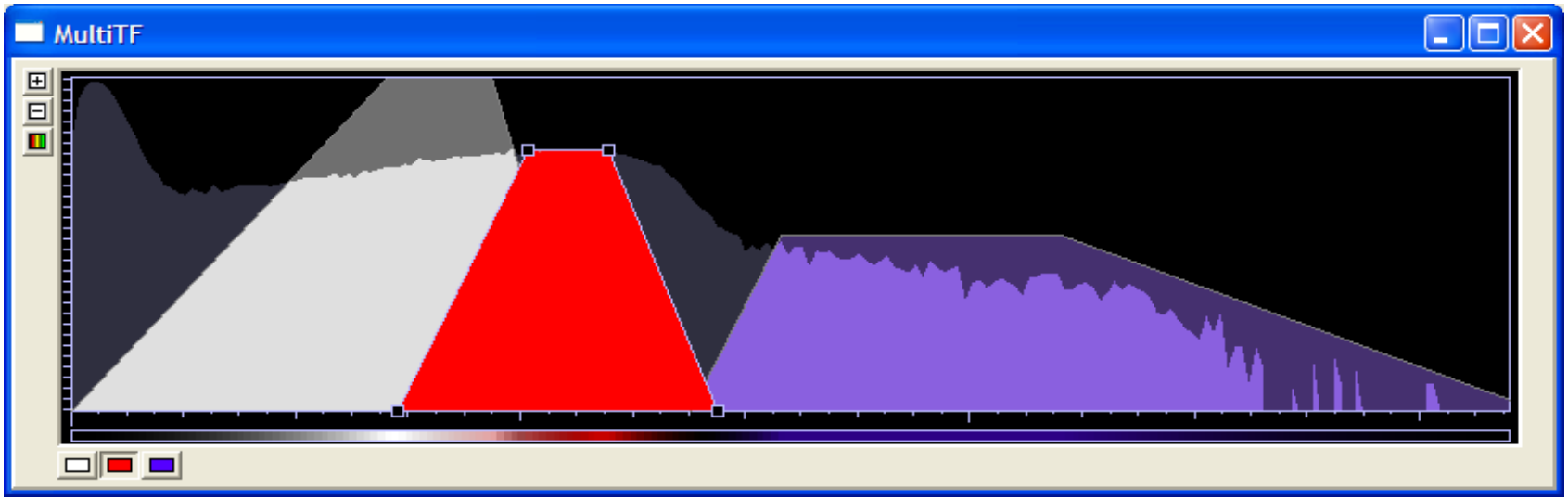


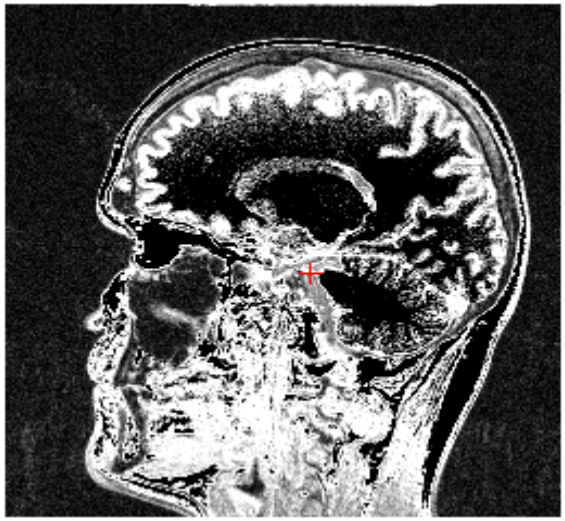
Transfer Functions (1)

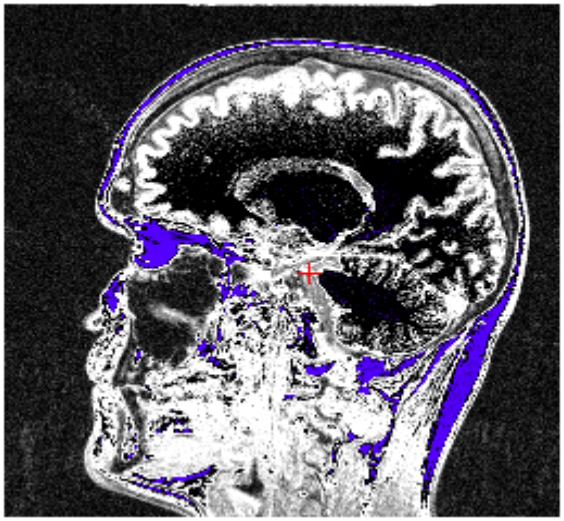


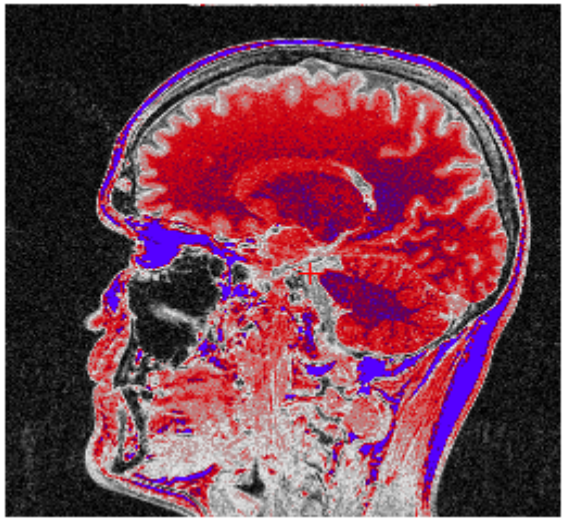
real-time update of the transfer function important

Transfer Functions (2)



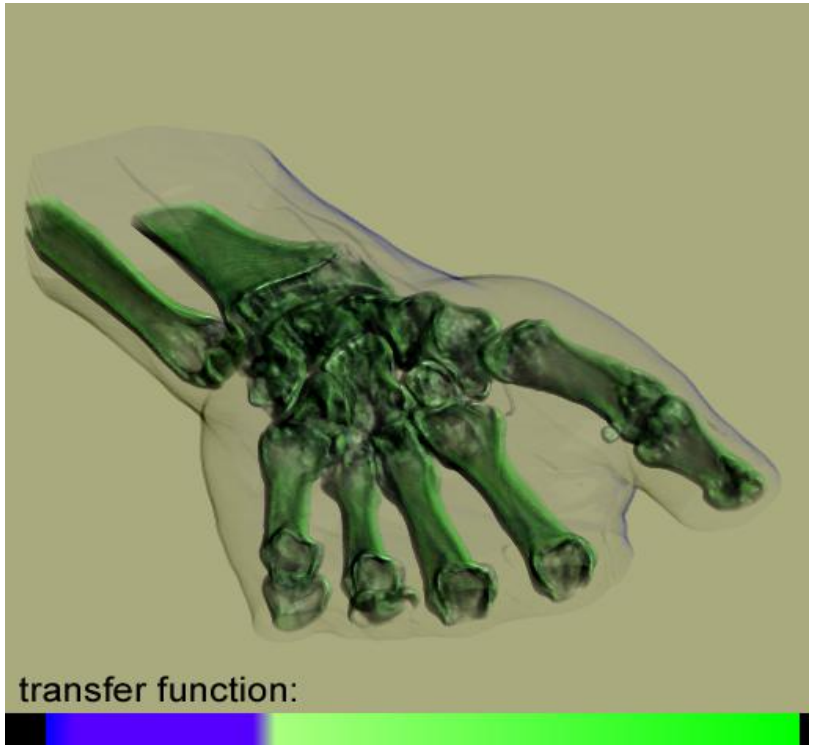


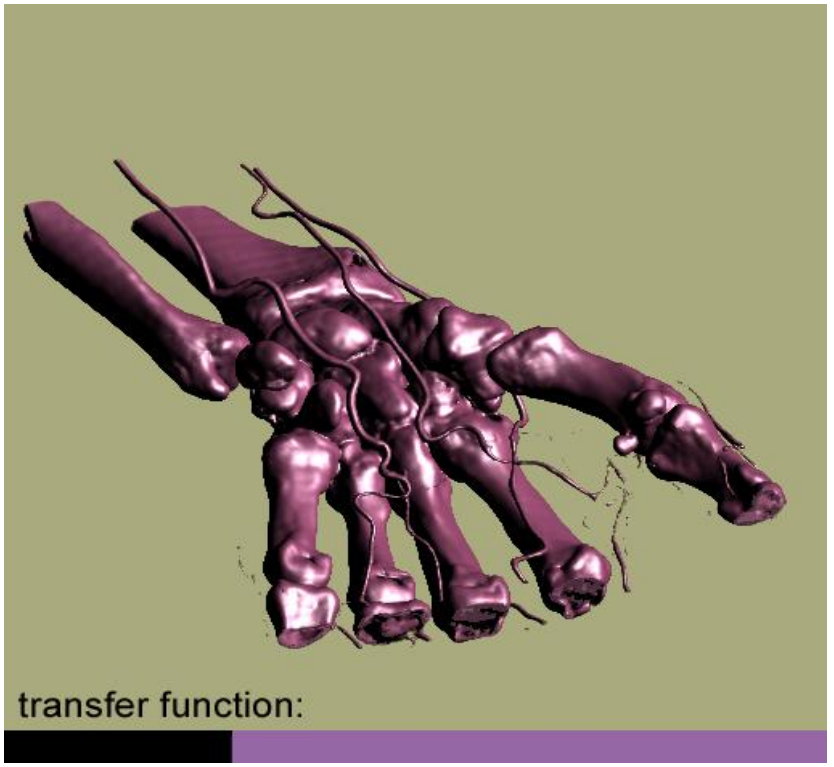




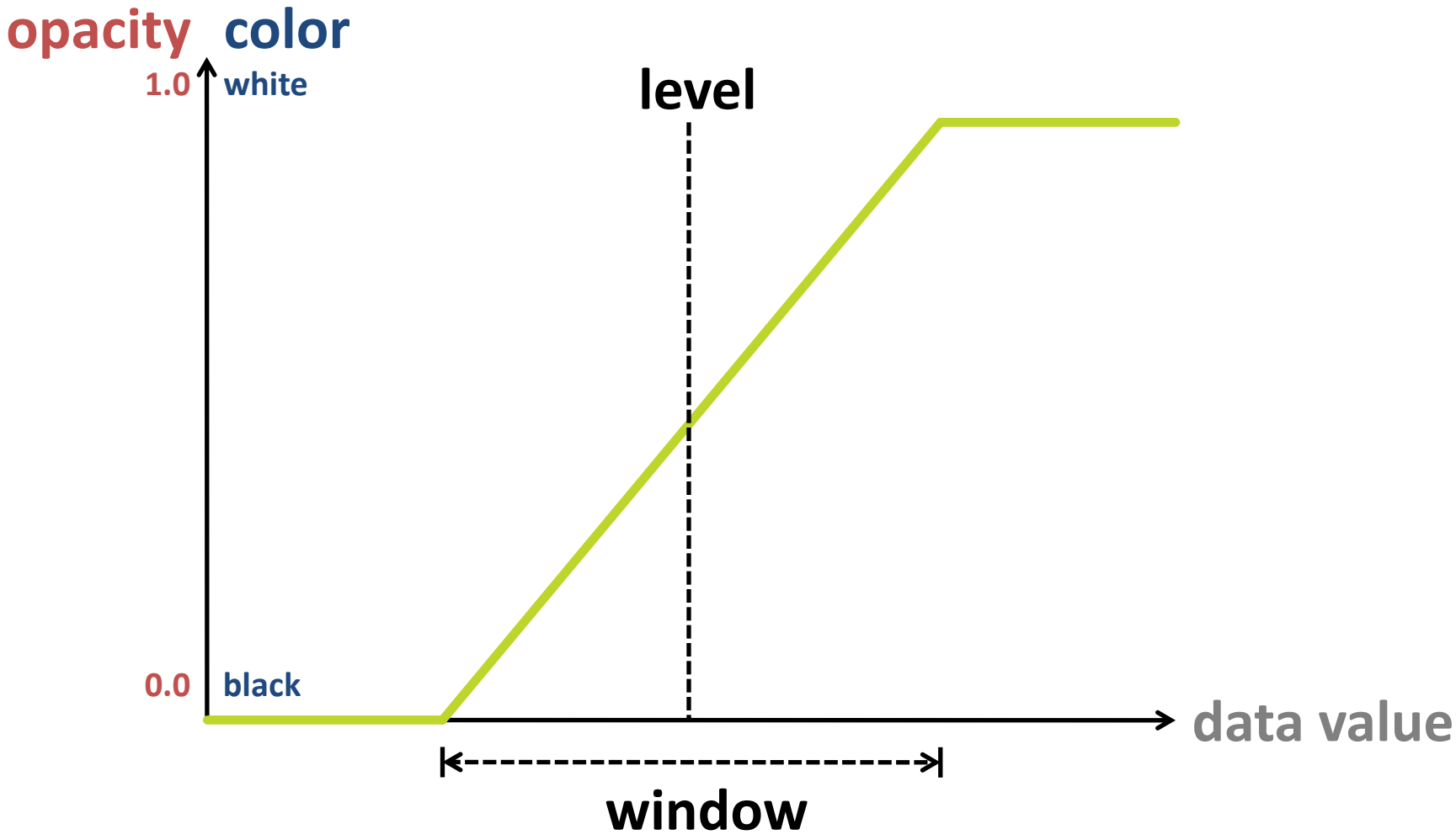
Transfer Functions

- Mapping of data attributes to optical properties
- Simplest: color table with opacity over data value





Window/Level



Classification Order (1)

Classification can occur before or after reconstruction

- Significant impact on quality

Pre-interpolative classification

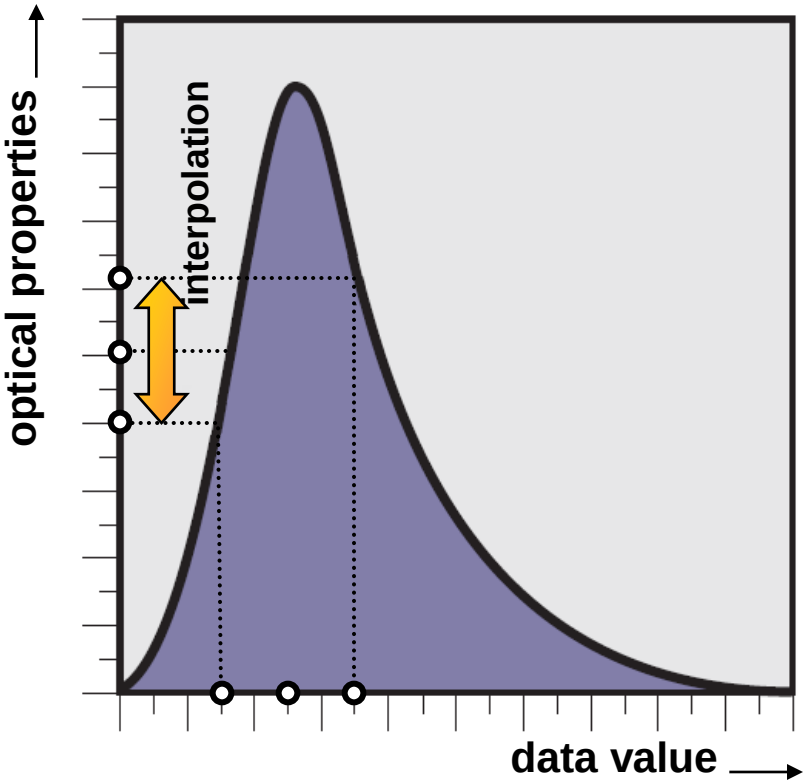
- Classify all data values and then interpolate between RGBA-tuples

Post-interpolative classification

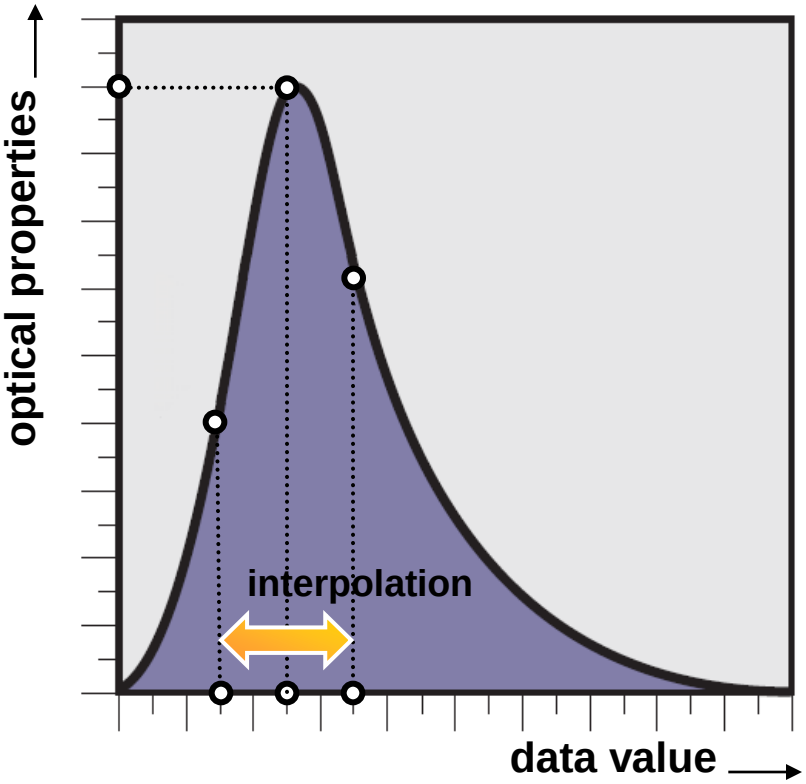
- Interpolate between scalar data values and then classify the result

Classification Order (2)

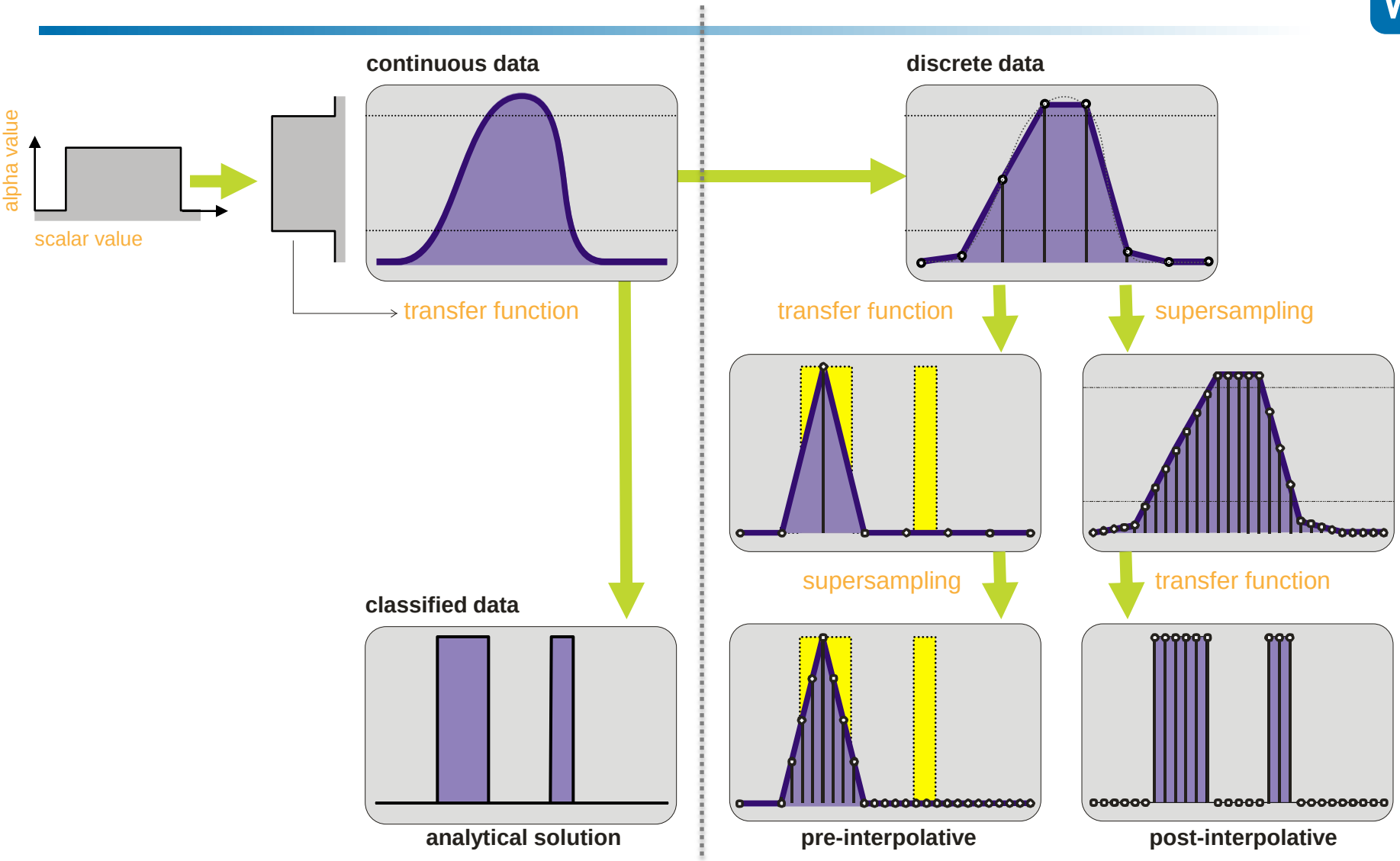
PRE-INTERPOLATIVE



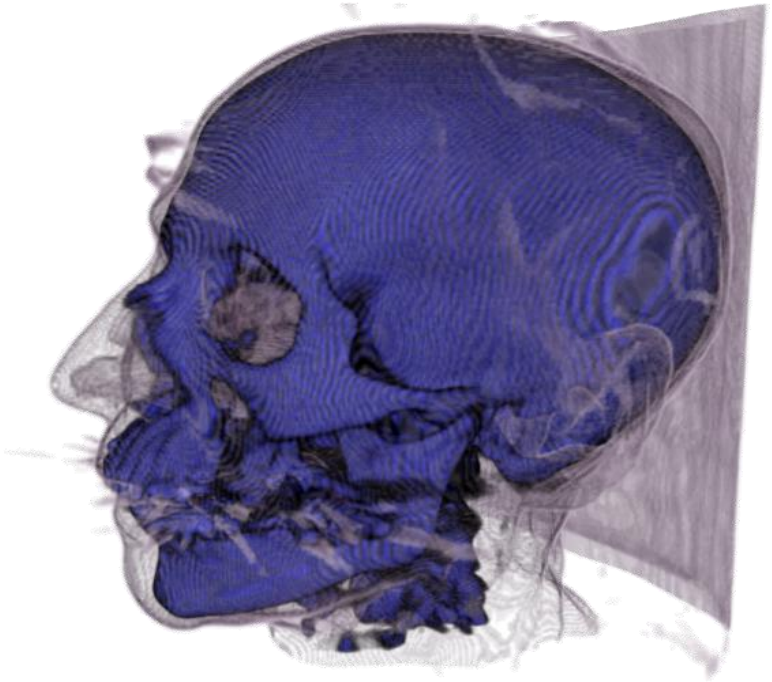
POST-INTERPOLATIVE



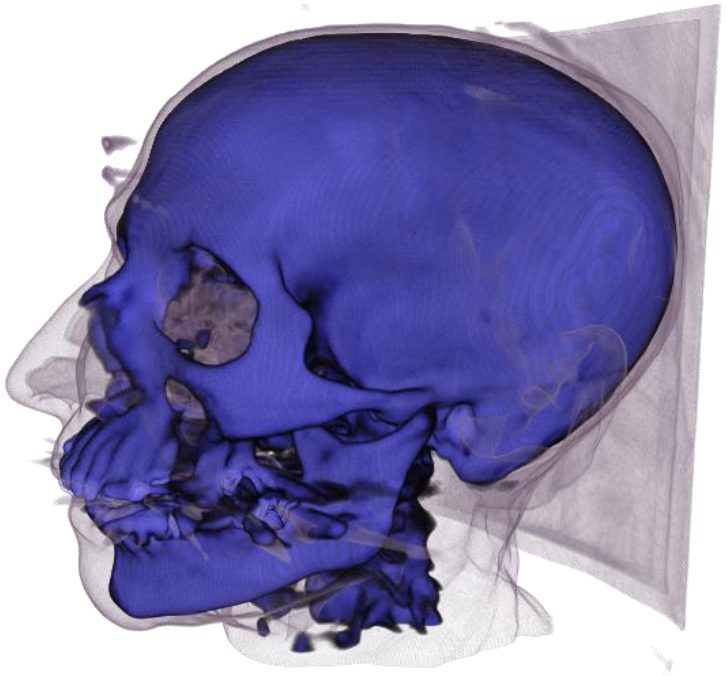
Classification Order (3)



Classification Order (4)



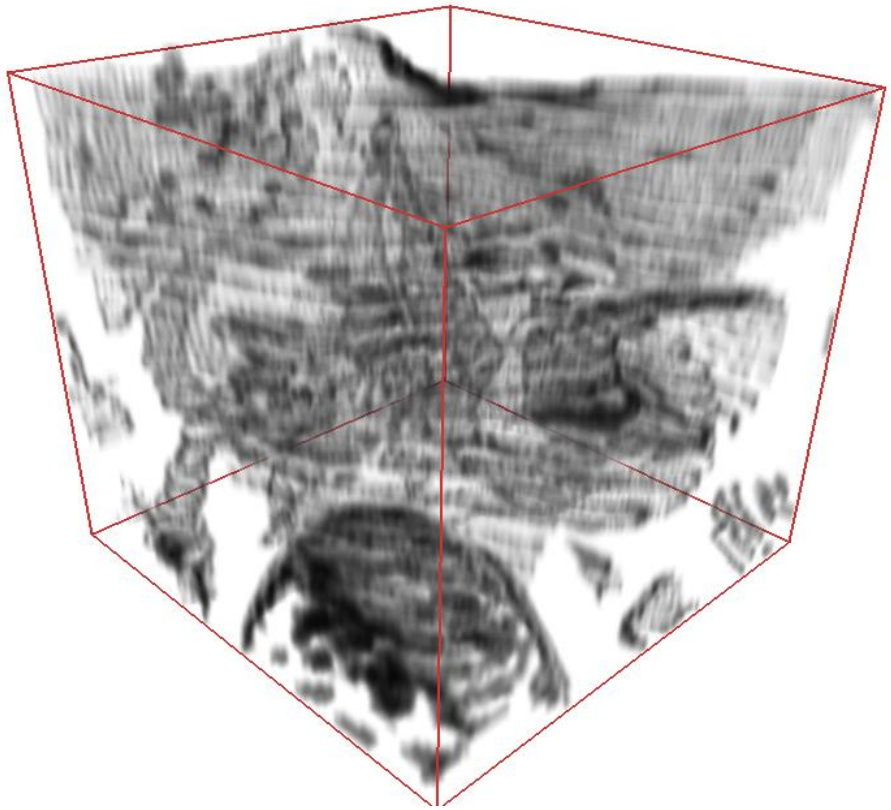
pre-interpolative



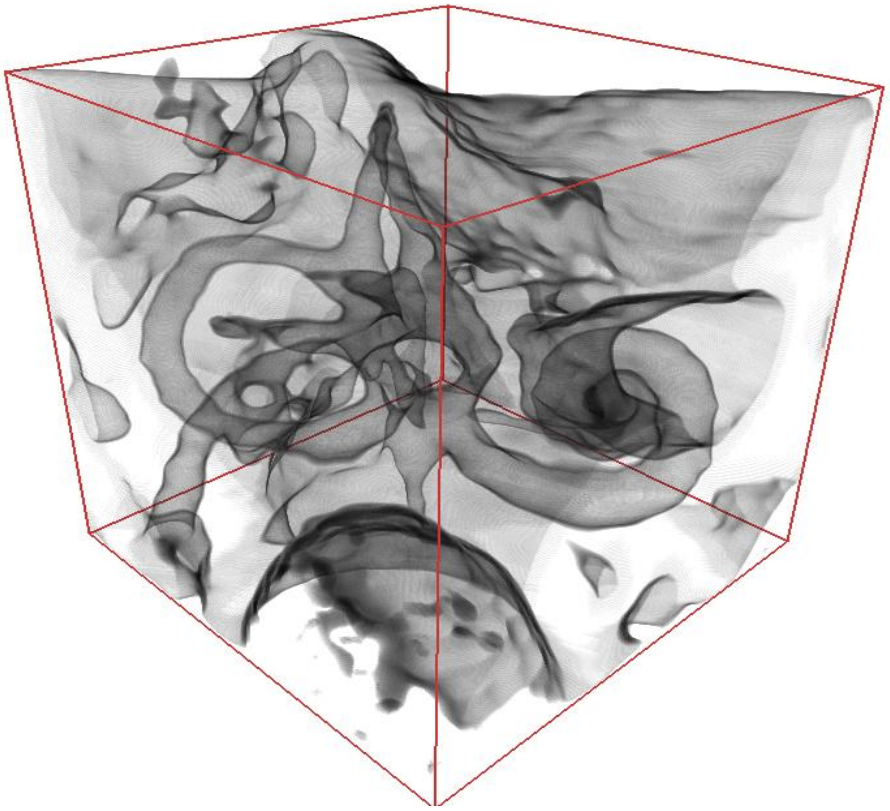
post-interpolative

same transfer function, resolution, and sampling rate

Classification Order (5)



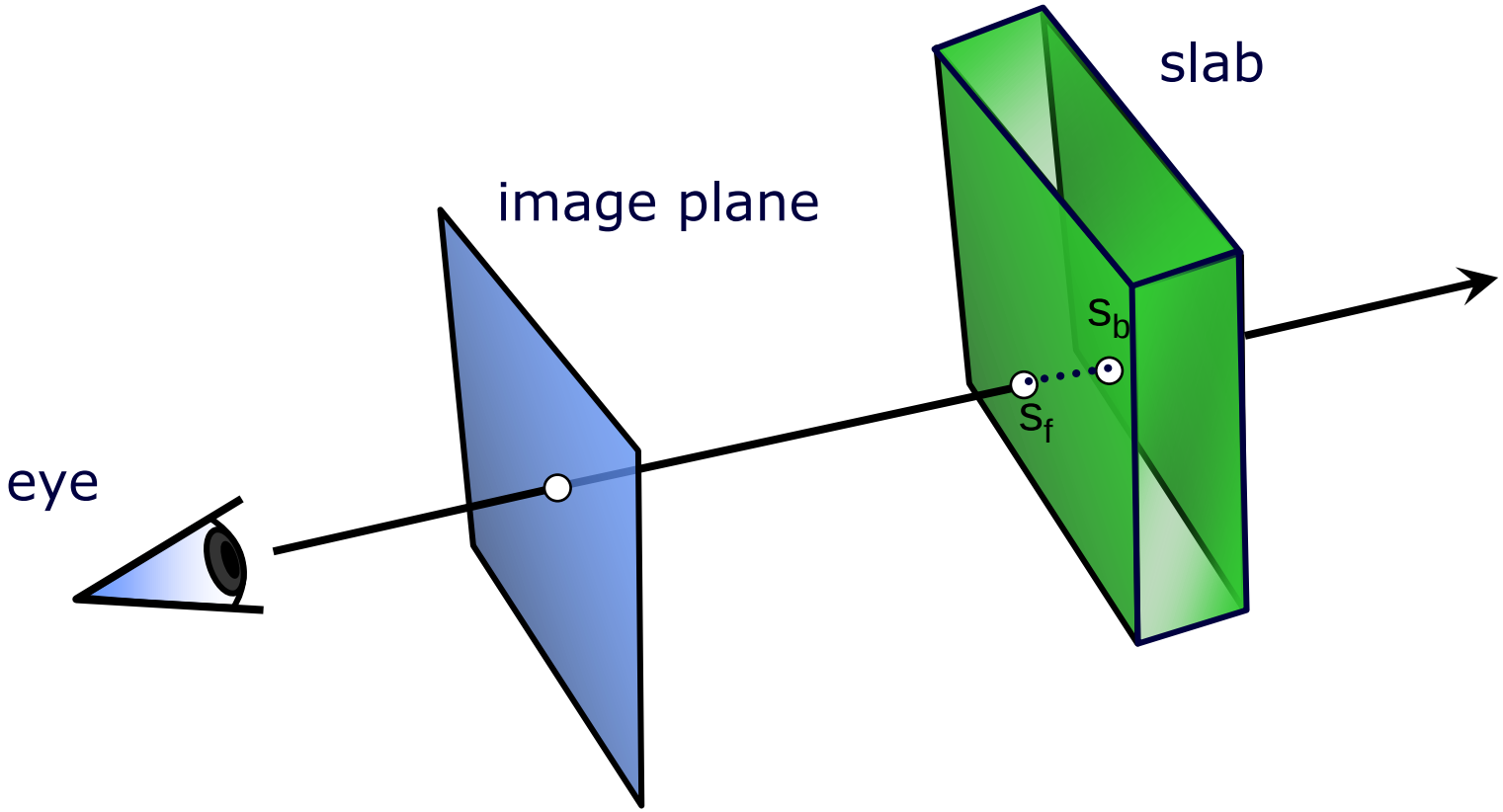
pre-interpolative



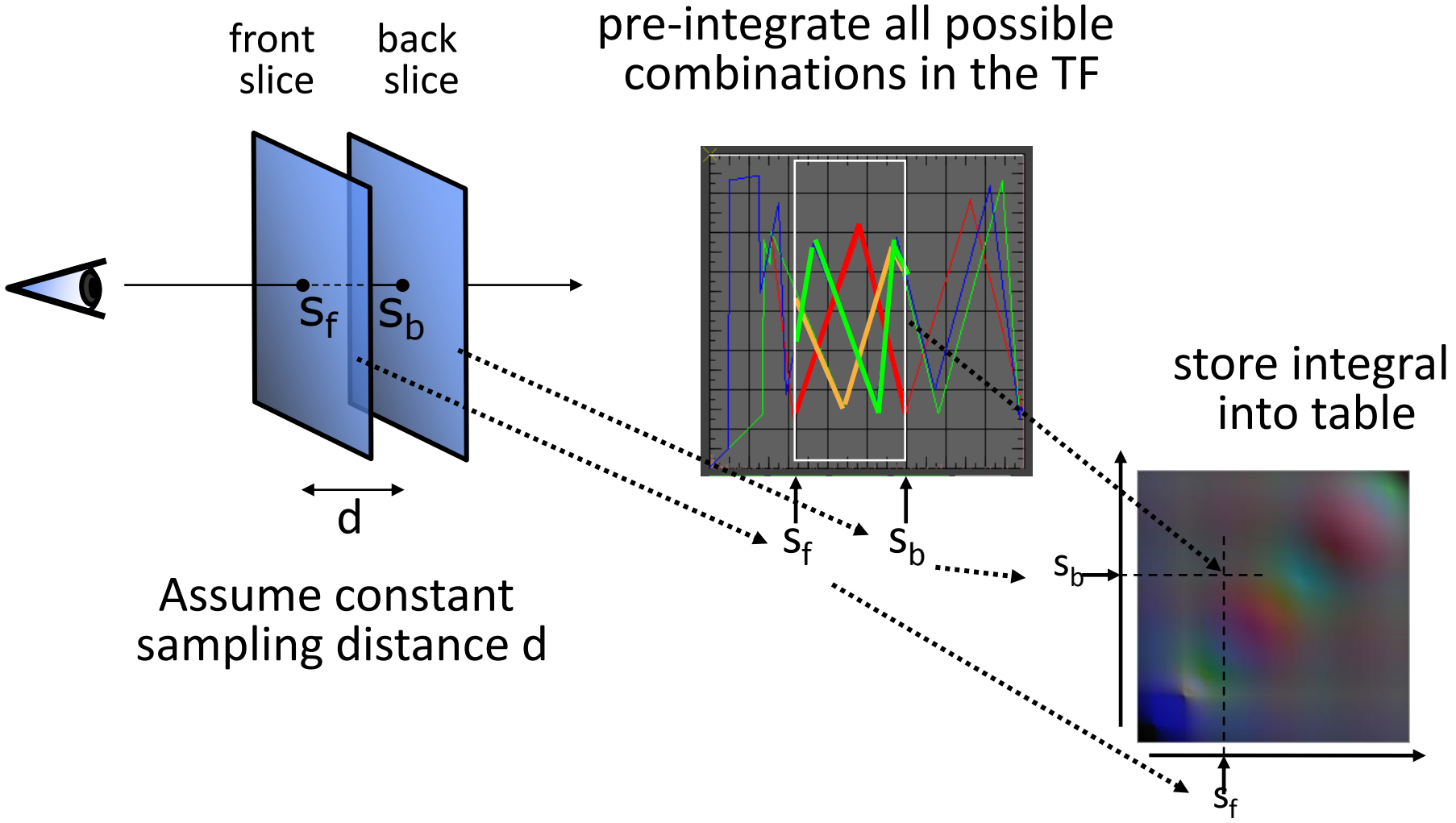
post-interpolative

same transfer function, resolution, and sampling rate

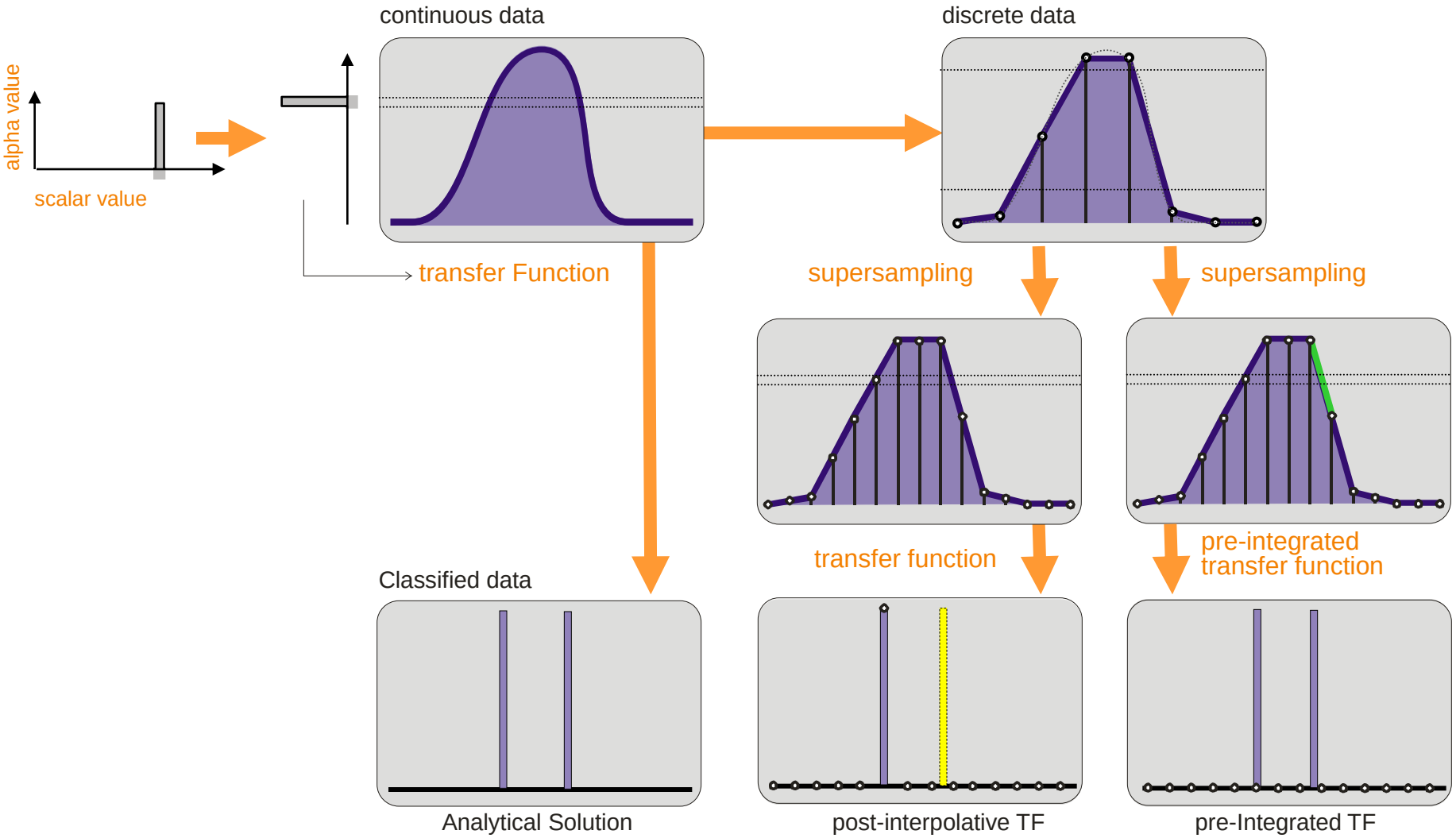
Pre-Integration (1)



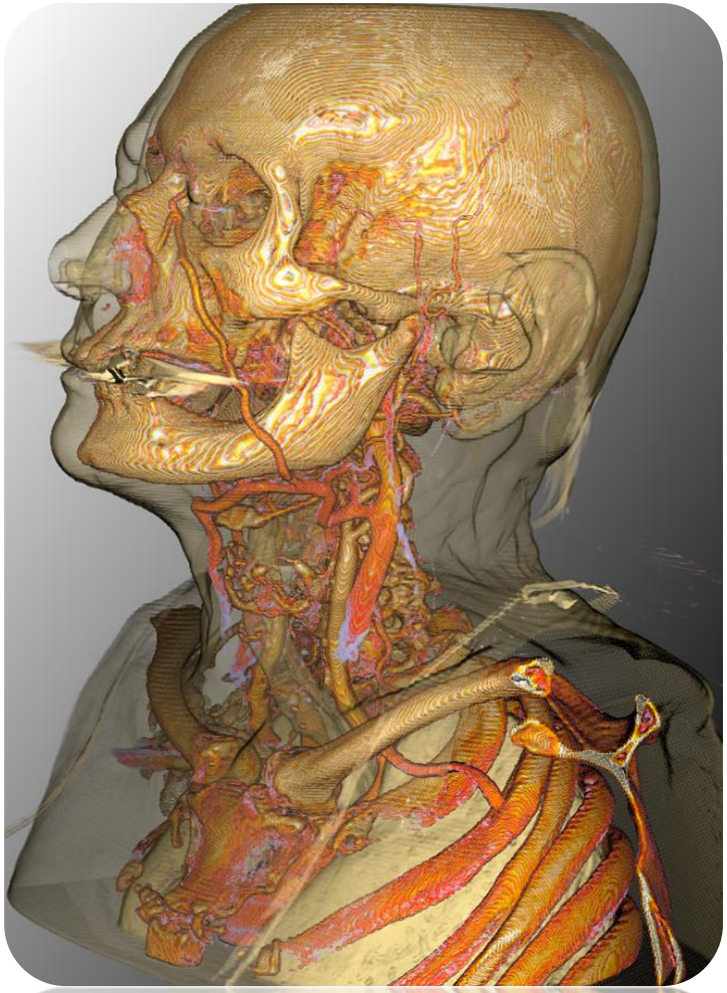
Pre-Integration (2)



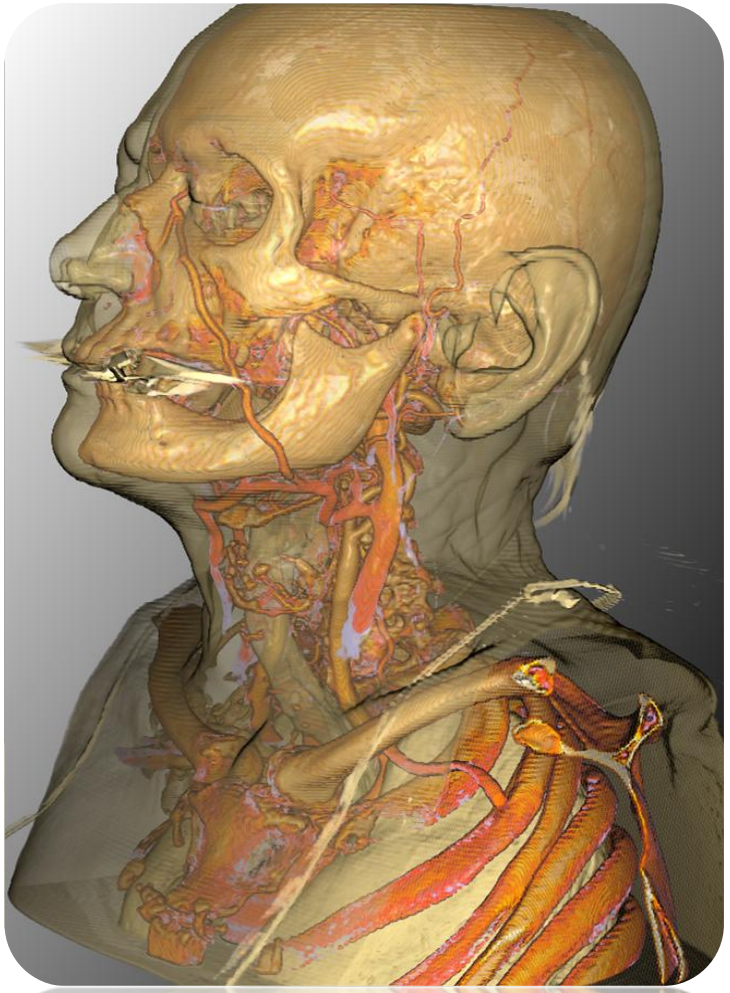
Pre-Integration (3)



Pre-Integration (4)



no pre-integration



pre-integration

Pre-Integration (5)

Shading more difficult to integrate, but possible

- Avoid high-dimensional lookup tables by decomposition
- [Lum et al. 2004], [Guetat et al. 2010]

Multi-dimensional transfer functions

- 2D transfer functions possible: [Kraus 2008]
- Higher dimensions difficult

Alternatives

- Adaptive sampling
- [Bergner et al. 2006]

Shading (1)

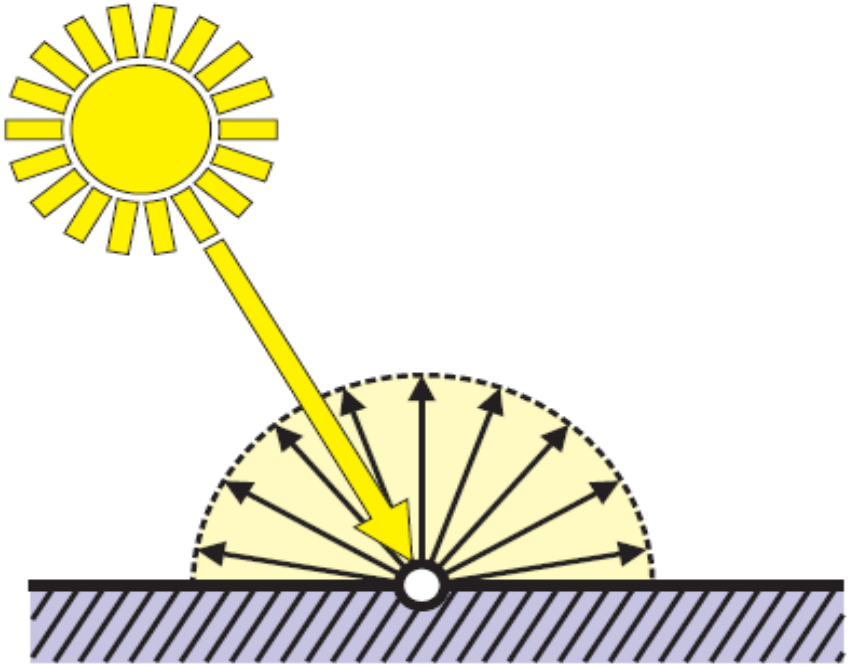
Make structures in volume data sets more realistic by applying an illumination model

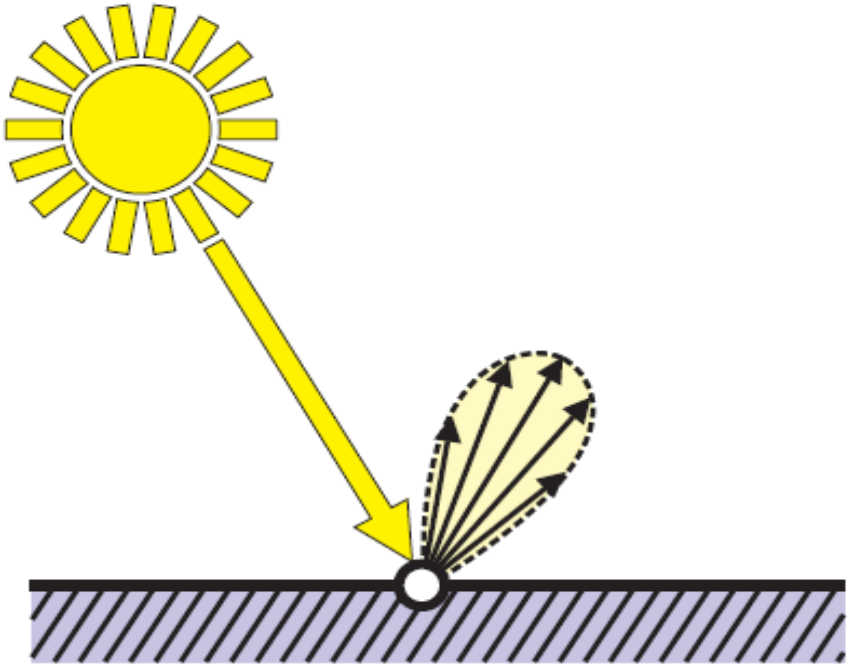
- Shade each sample in the volume like a surface
- Any model used in real-time surface graphics suitable
- Common choice: Blinn-Phong illumination model

Shading (2)

Local illumination, similar to surface lighting

- **Lambertian reflection**
  light is reflected equally in all directions
- **Specular reflection**
  light is reflected scattered around the direction of perfect reflection





Shading (3)

shaded volume rendering

unhaded volume rendering

Gradient Estimation (1)

Normalized gradient vector of the scalar field is used to substitute for the surface normal

The gradient vector is the first-order derivative of the scalar field

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f(\mathbf{x})}{\partial x} \\ \frac{\partial f(\mathbf{x})}{\partial y} \\ \frac{\partial f(\mathbf{x})}{\partial z} \end{pmatrix}$$

partial derivative in x-direction

partial derivative in y-direction

partial derivative in z-direction

Gradient Estimation (2)

We can estimate the gradient vector using finite differencing schemes, e.g. central differences:

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

Noisy data may require more complex estimation schemes

Gradient Magnitude

Magnitude of gradient vector can be used to measure the "surfaceness" of a point

- Strong changes → high gradient magnitude
- Homogenity → low gradient magnitude

Applications

- Use gradient magnitude to modulate opacity of sample
- Interpolate between unshaded and shaded sample color using gradient magnitude as weight

Compositing

DVR (Direct Volume Rendering)

- Physically-based
- Optical model for emission & absorption
- May require complex transfer function
- Visual cues due to accumulation and shading

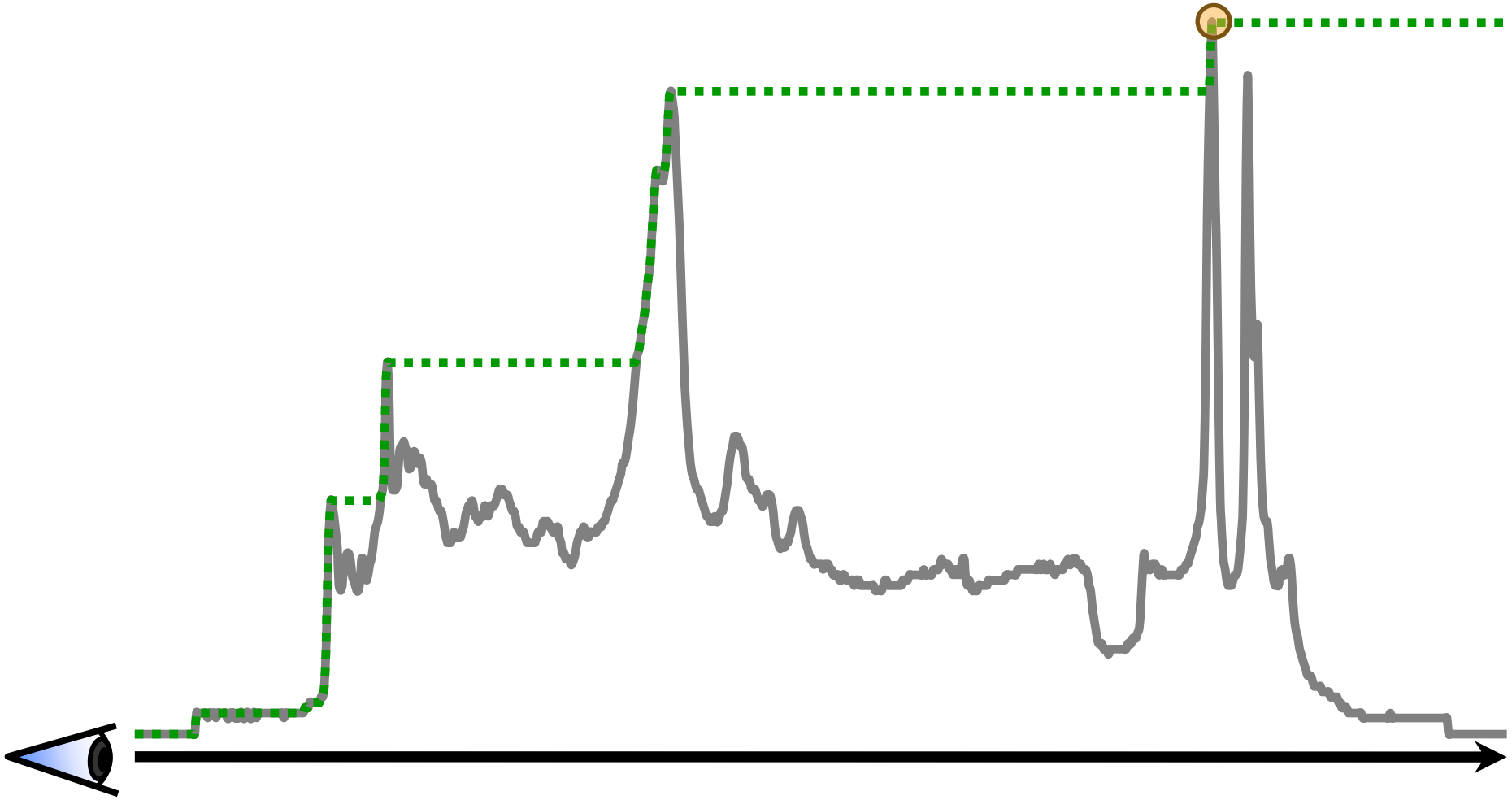
MIP (Maximum Intensity Projection)

- Practically-motivated
- Project maximum value along each viewing ray
- Suffices with window/level setting
- Spatial ambiguities caused by order-independency

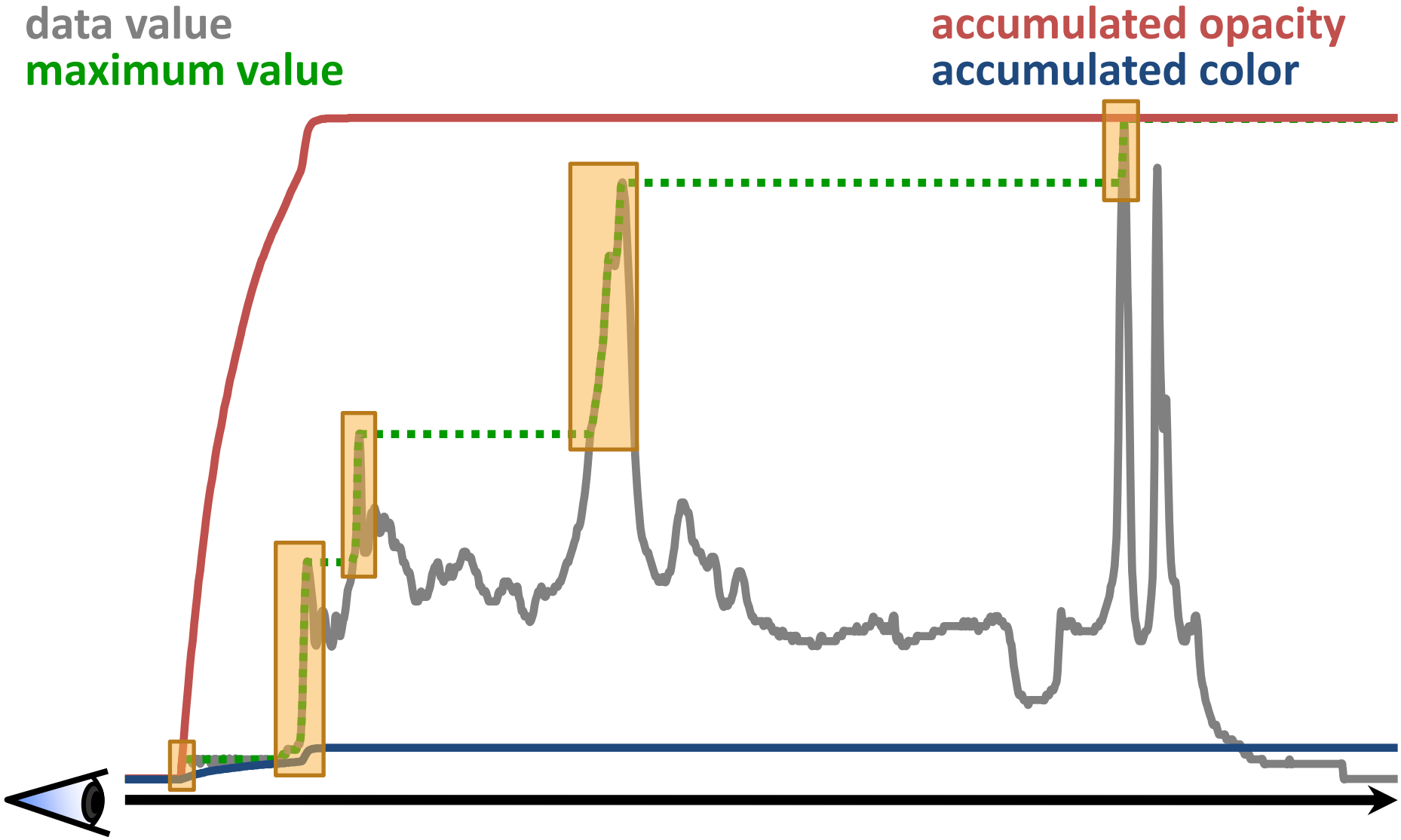
Maximum Intensity Projection

data value

maximum value



Direct Volume Rendering



Maximum Intensity Difference

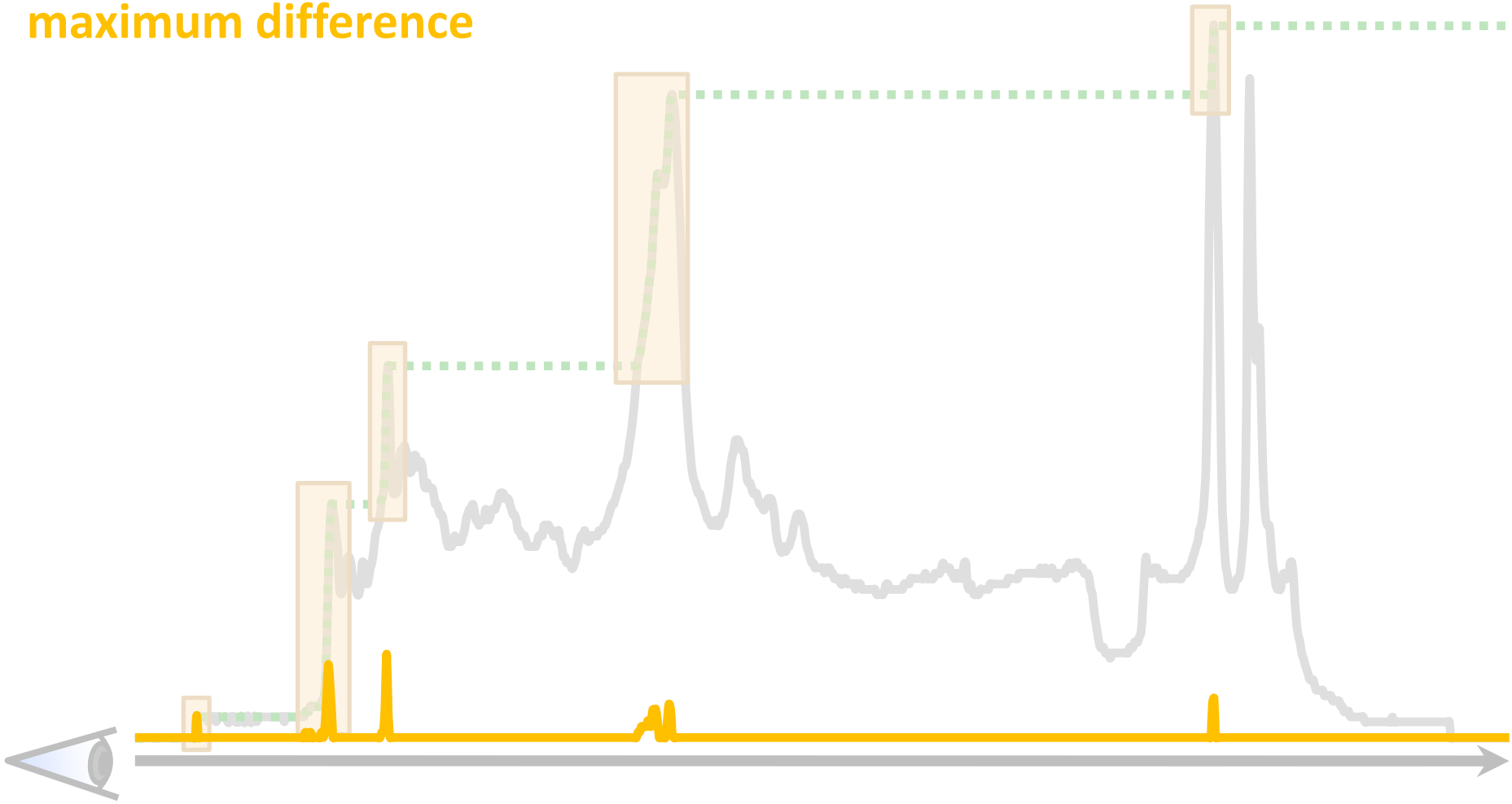
data value
maximum value

Difference between the data value at the i-th sample along a viewing ray and the current maximum

$$\delta_i = \begin{cases} f_i - f_{max_i} & \text{if } f_i > f_{max_i} \\ 0 & \text{otherwise} \end{cases}$$

Maximum Intensity Difference

data value
maximum value
maximum difference



Modified Compositing

Accumulated opacity A_i and color C_i at i-th sample along a viewing ray

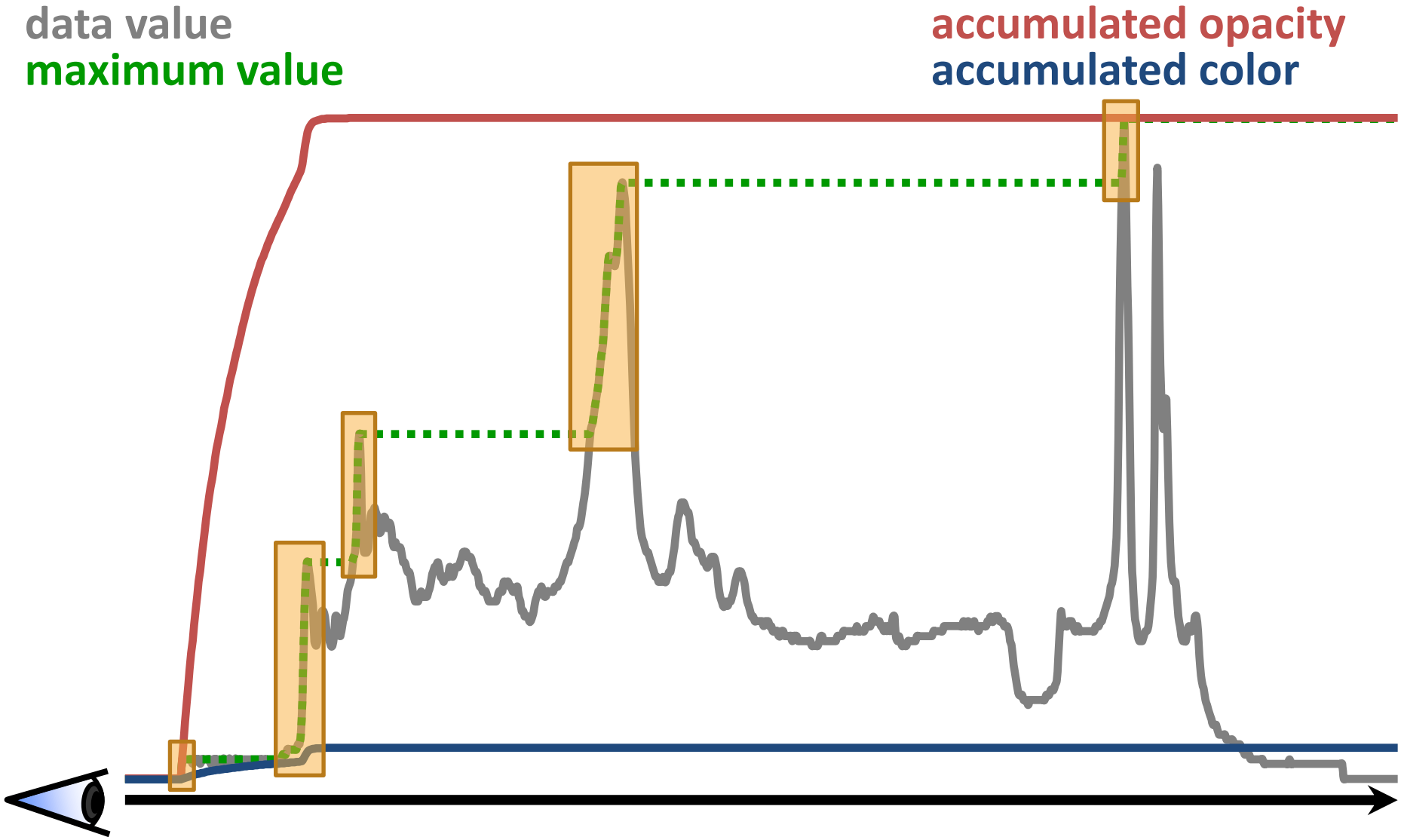
$$A_i = \beta_i A_{i-1} + (1 - \beta_i A_{i-1})\alpha_i$$

$$C_i = \beta_i C_{i-1} + (1 - \beta_i A_{i-1})\alpha_i c_i$$

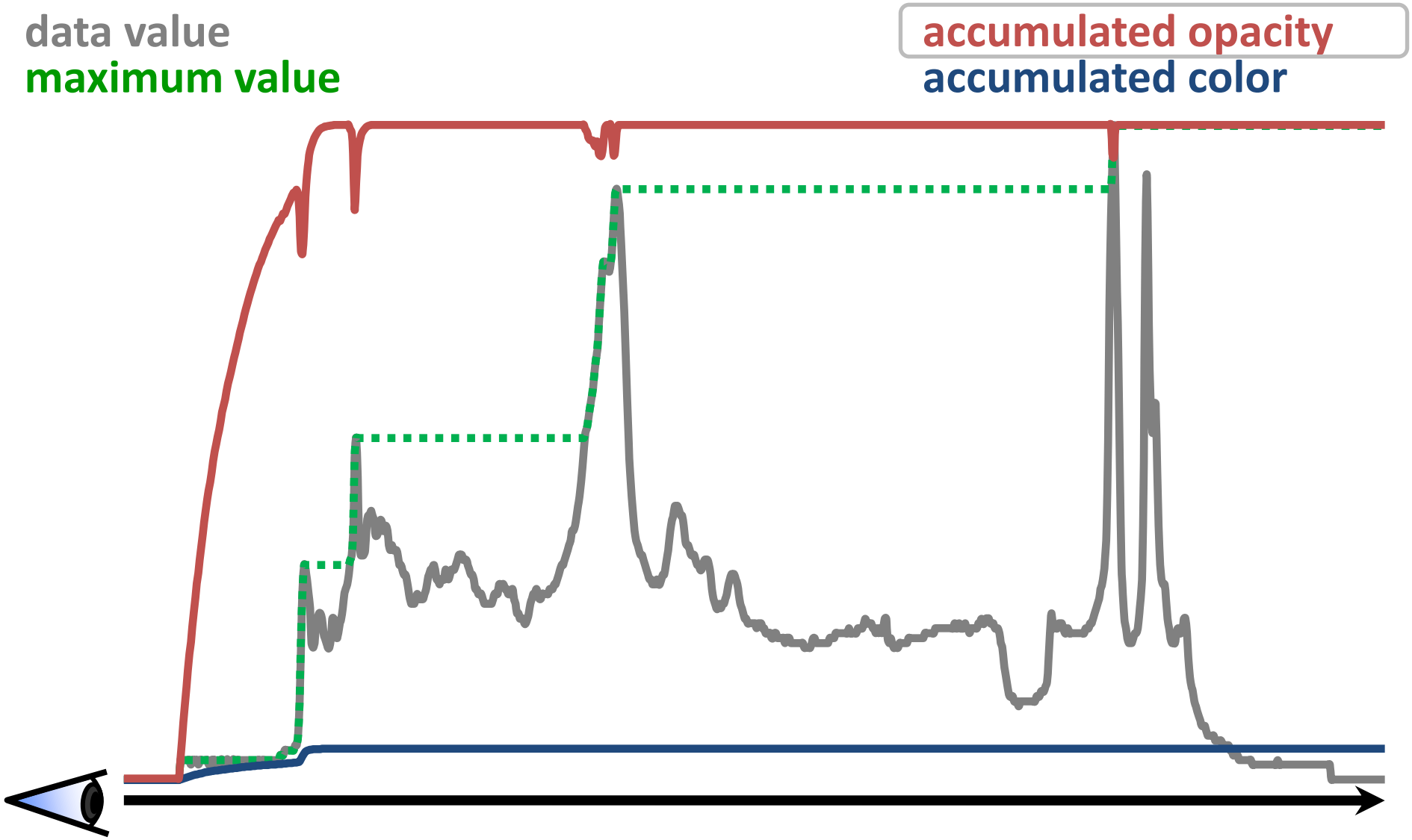
α_i opacity of the sample
c_i color of the sample

$$\beta_i = 1 - \delta_i$$

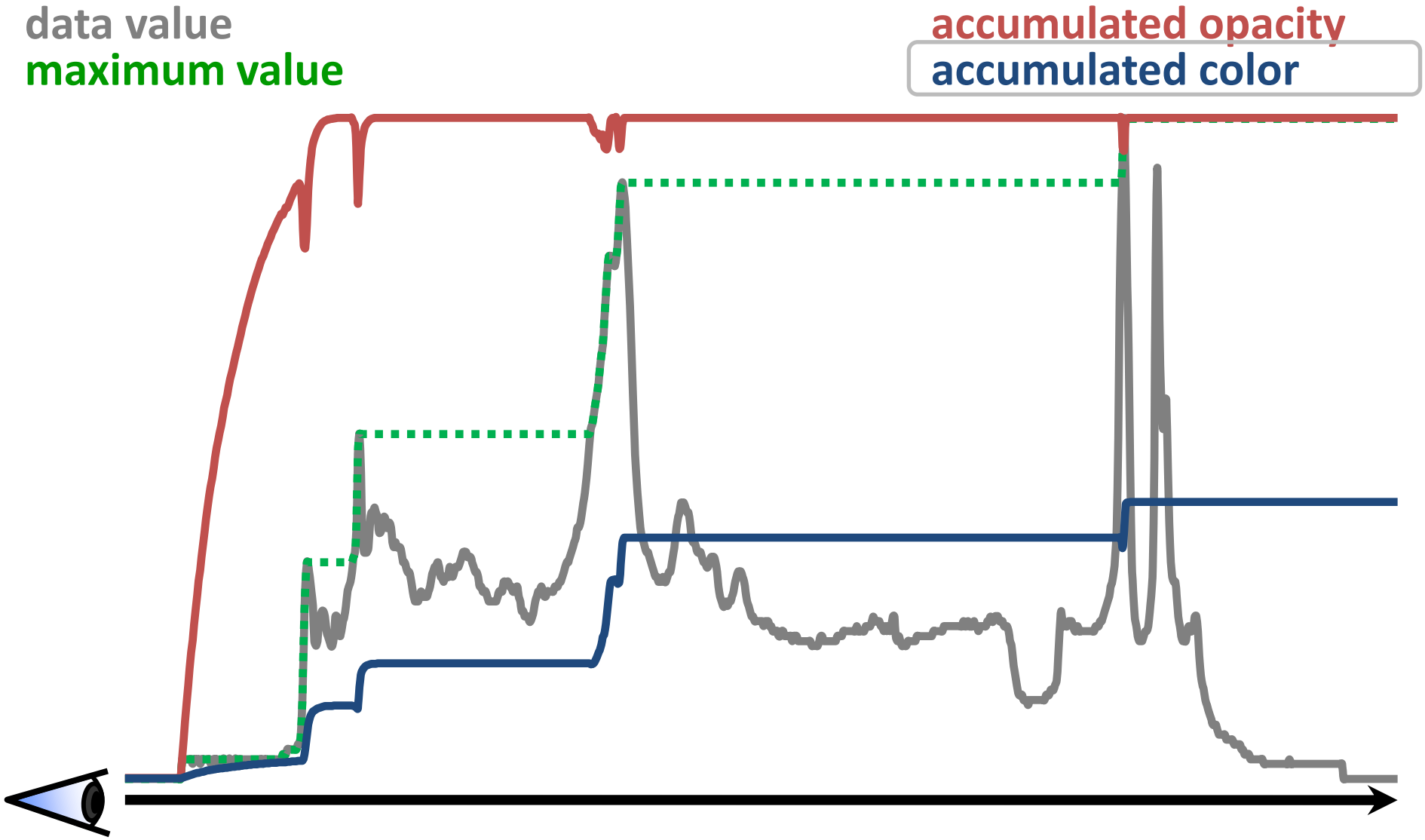
Direct Volume Rendering



Max. Intensity Difference Accumulation



Max. Intensity Difference Accumulation



Combining DVR, MIDA, and MIP (1)

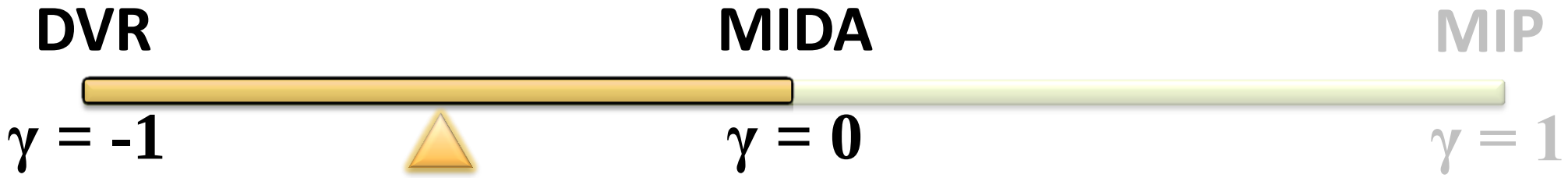
MIDA features characteristics of DVR as well as MIP

Use it as an intermediate step for a smooth transition

Ability to make a DVR image more MIP-like and vice versa

User interface: just replace “render mode” combobox with slider

Combining DVR, MIDA, and MIP (2)



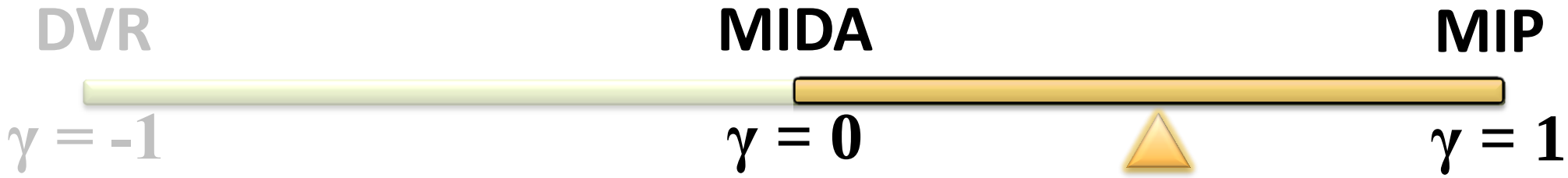
$$A_i = \beta_i A_{i-1} + (1 - \beta_i A_{i-1})\alpha_i$$

$$C_i = \beta_i C_{i-1} + (1 - \beta_i A_{i-1})\alpha_i c_i$$

α_i opacity of the sample
c_i color of the sample

$$\beta_i = 1 - \delta_i (1+\gamma)$$

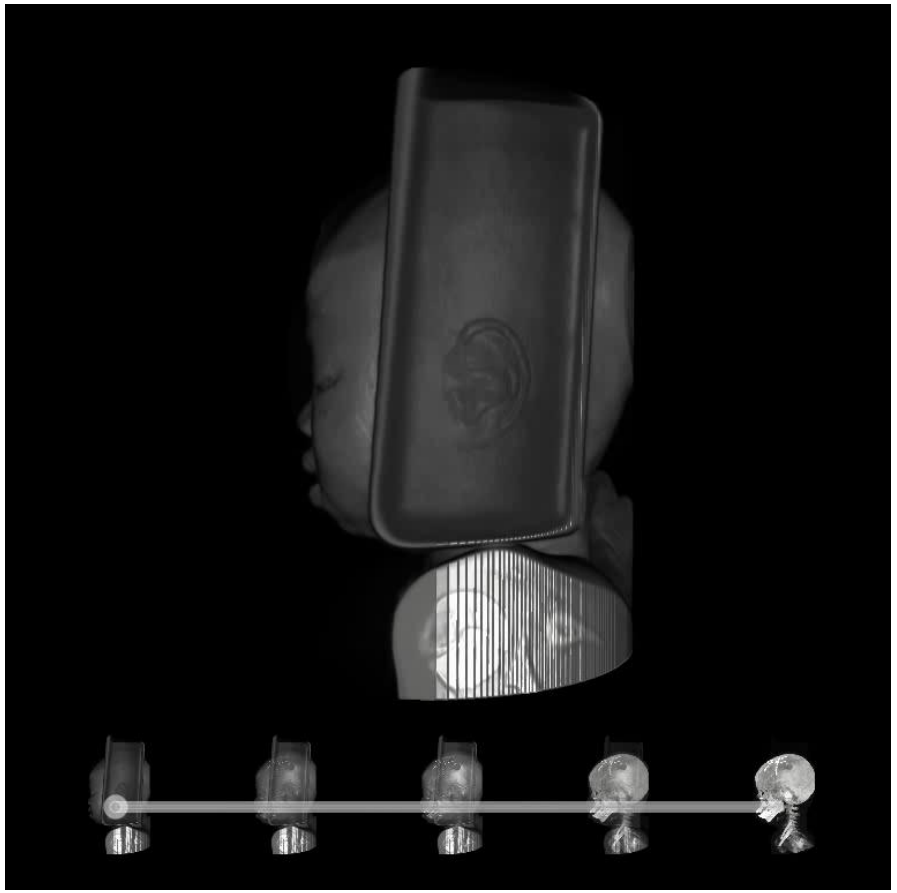
Combining DVR, MIDA, and MIP (3)

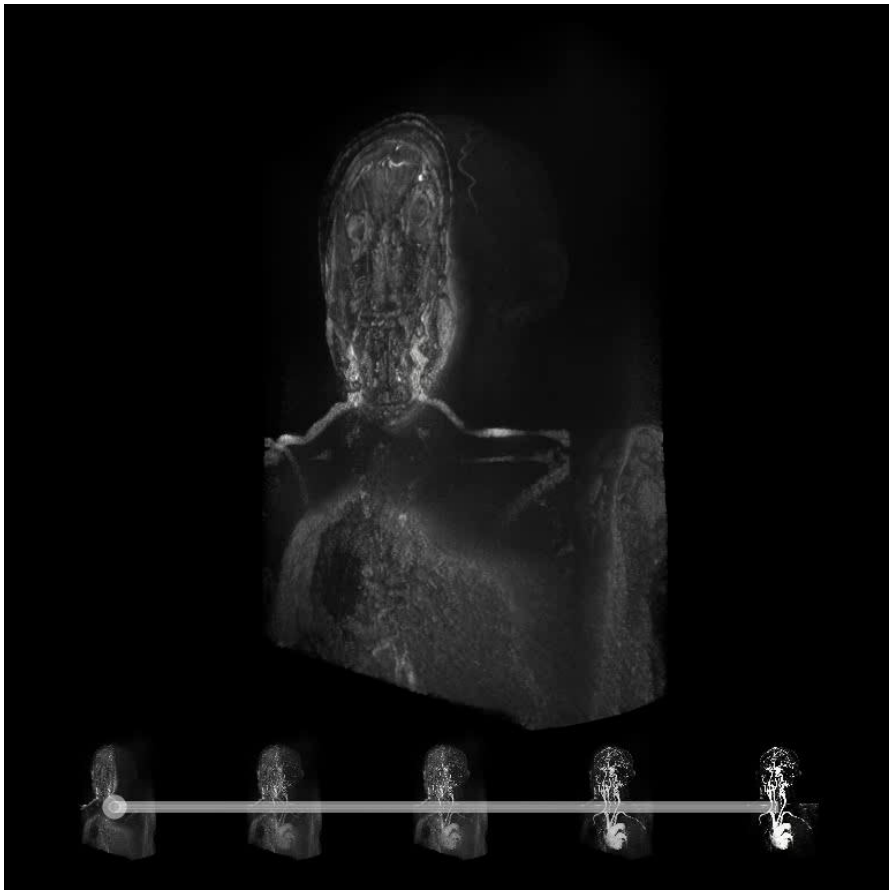


Could let β_i approach one only where the maximum changes as γ get closer to MIP

- Problem: non-graceful degradation if shading is used
- Solution: perform transition to MIP in image space by linearly interpolating between MIDA and MIP result colors

Example: DVR, MIDA, MIP





Coping With Volume Size (1)

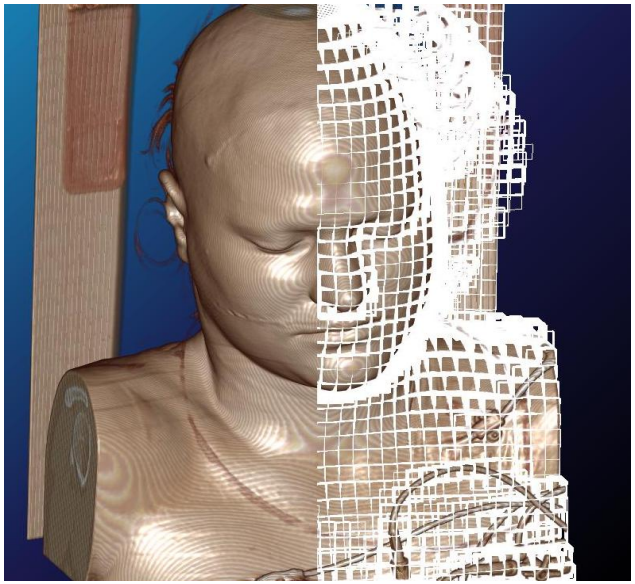
- Multi-gigabyte volumes
- Bricking
  - Hierarchical (e.g., octree) vs. flat/fixed number of layers
  - Multiresolution
  - Out-of-core rendering

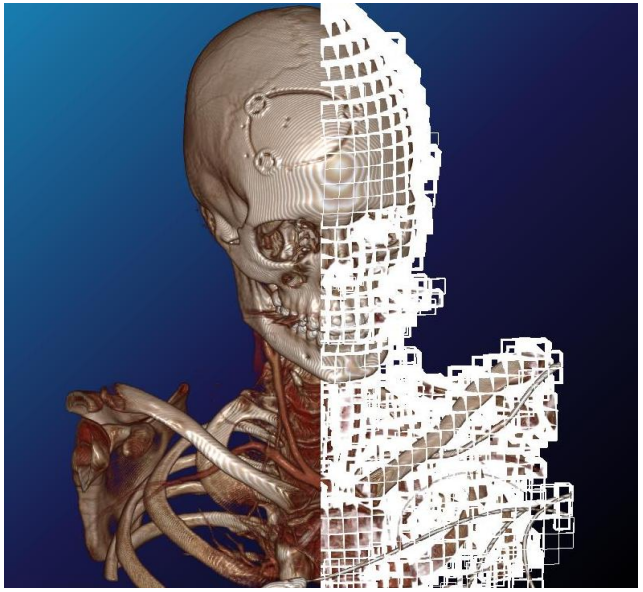
512x512x3396

[Ljung et al., 2006]

Coping With Volume Size (2)

- Cull bricks against TF (need min/max value per brick)
- Render each brick separately or all in single pass
- GPU filtering: duplicate neighbor voxels

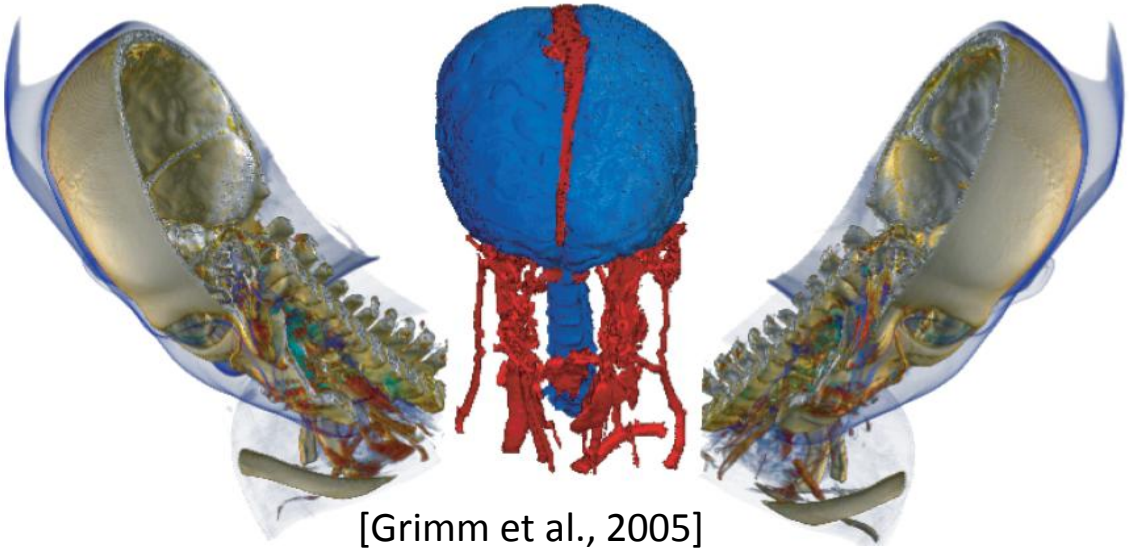




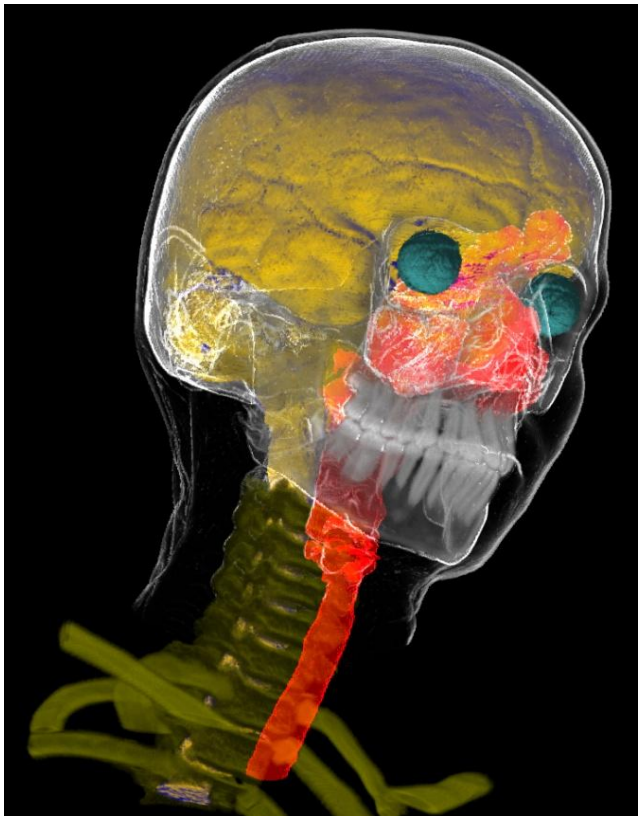
Courtesy Klaus Engel

Segmented Data

- Segmentation mask / object ID volume
- Per object:
  - Enabling/disabling, transfer function, rendering mode, clipping planes, ...
- Volume exploration (move objects, ...)

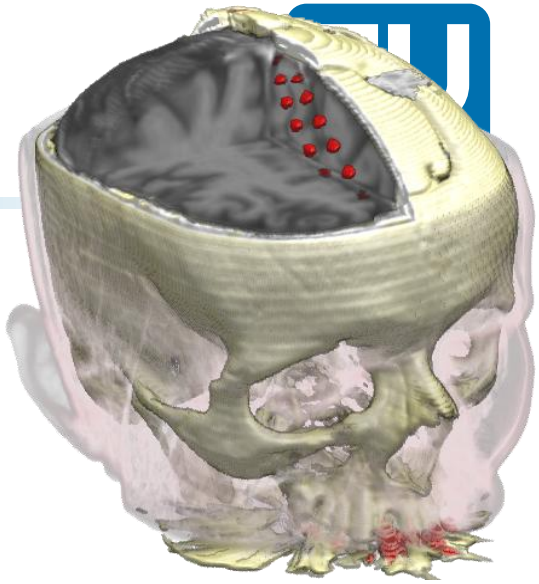


[Grimm et al., 2005]

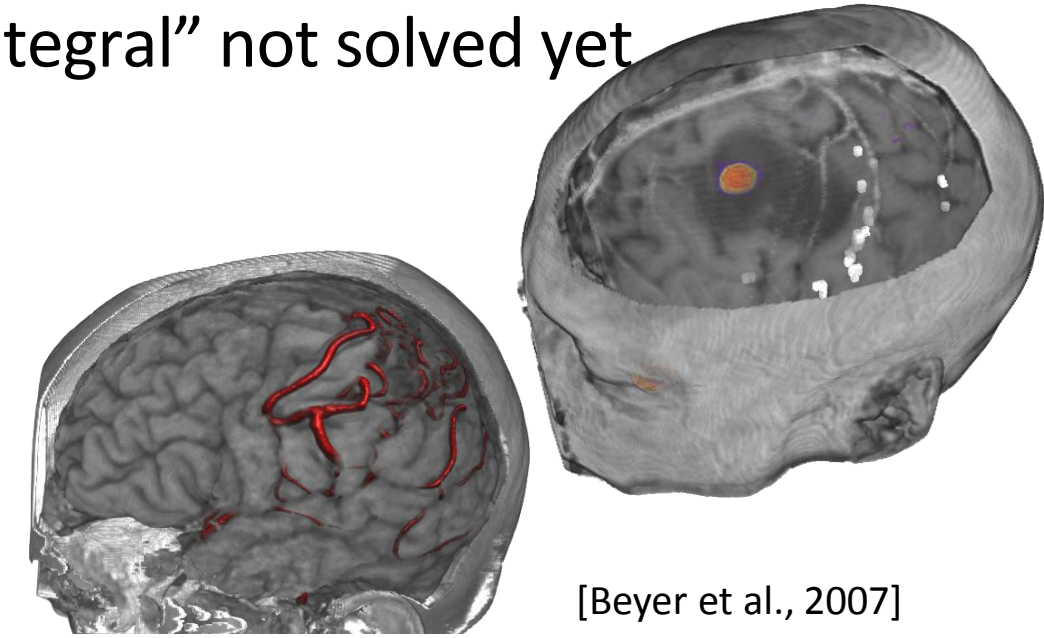


[Hadwiger et al., 2003]

Multi-Volume Rendering



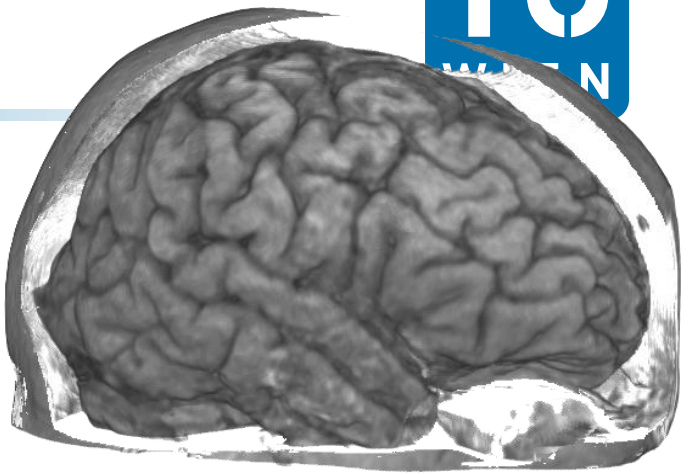
- Combine multiple modalities (CT, MR, fMR, PET, DSA, …)
- Per-volume transfer function
- Combine with segmentation info
- “Multi-volume rendering integral” not solved yet
- Simple approaches:
  - Blend multiple volumes
  - Switch transfer function according to segm. object
  - Switch depending on threshold (vessels from DSA, …)



[Beyer et al., 2007]

Opacity Peeling / Skull Peeling

- Peel away layers according to accumulated opacity
  - Example: show brain without segm.
  - Skull peeling: CT and MR combined: peeling with CT, rendering brain from MR

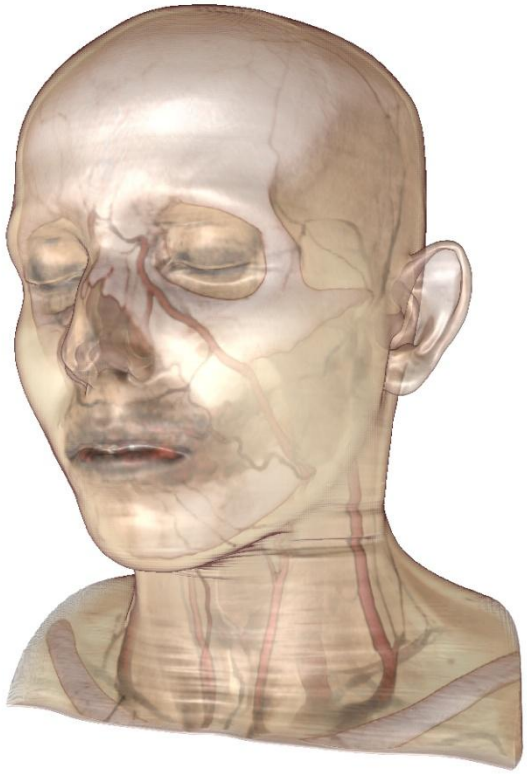


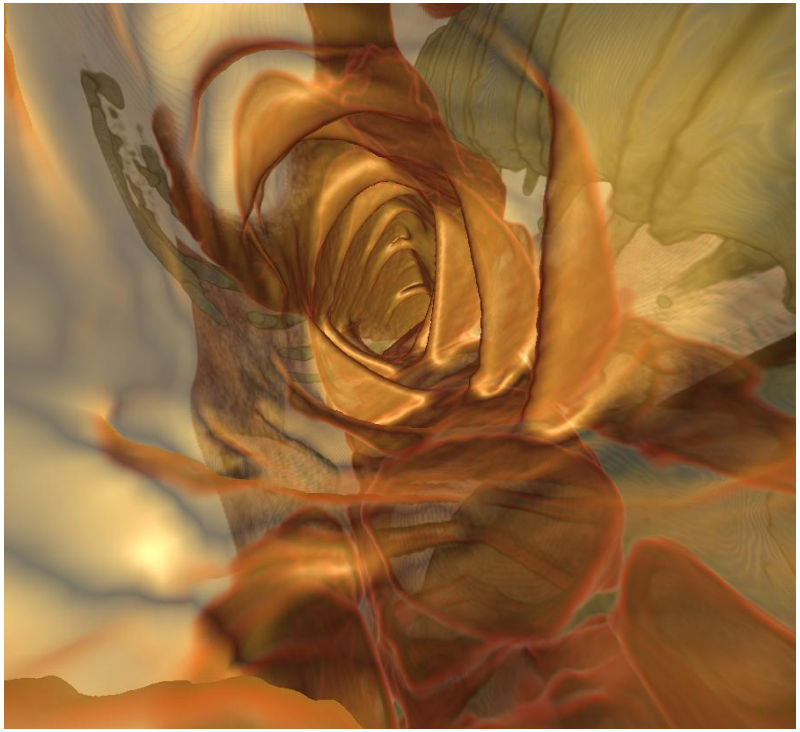
[Beyer et al., 2007]

Opacity layers 1 - 4

[Rezk-Salama et al., 2006]

GPU-Based Ray-Casting



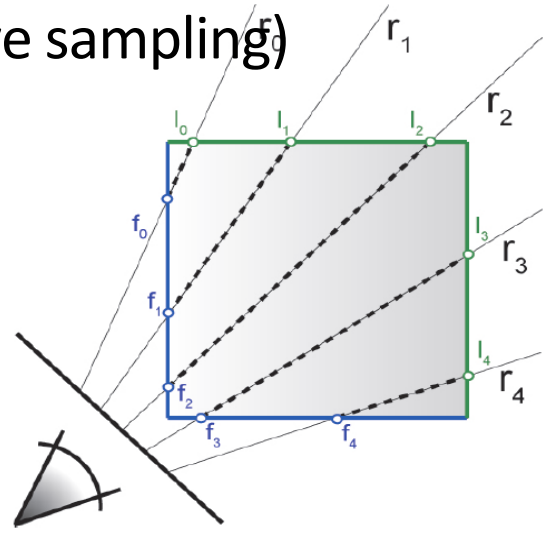


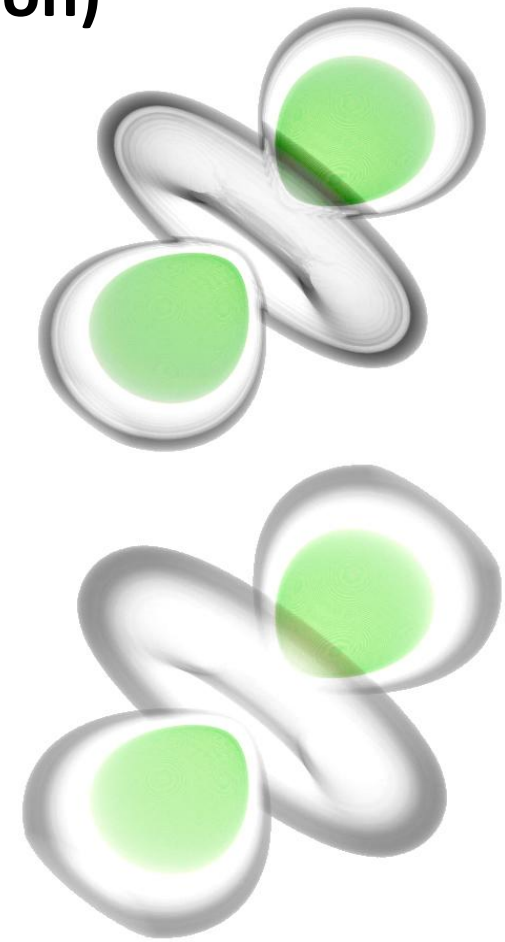
Why Ray-Casting on GPUs?

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





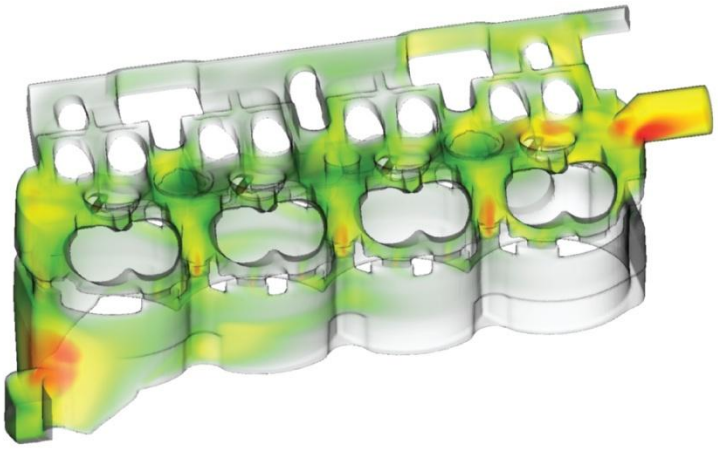
Recent GPU Ray-Casting Approaches

Rectilinear 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

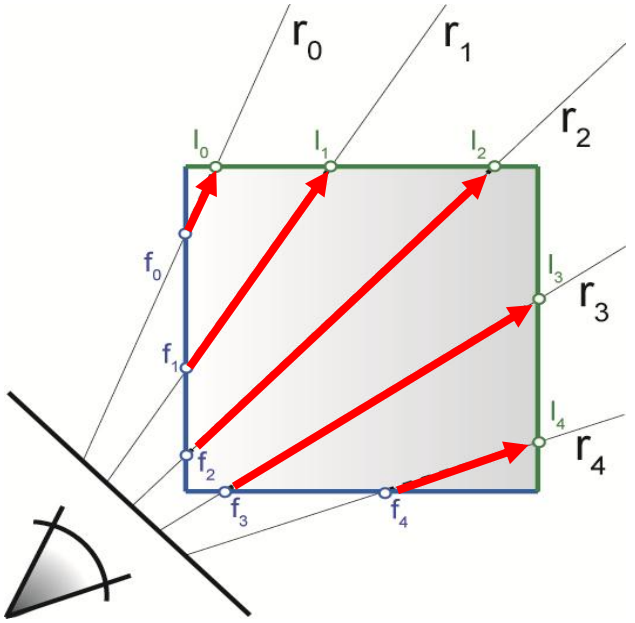
- [Weiler et al., 2002, 2003, 2004]
- [Bernardon et al., 2004]
- [Callahan et al., 2006]
- [Muigg et al., 2007]



Basic Ray Setup / Termination

Two main approaches:

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

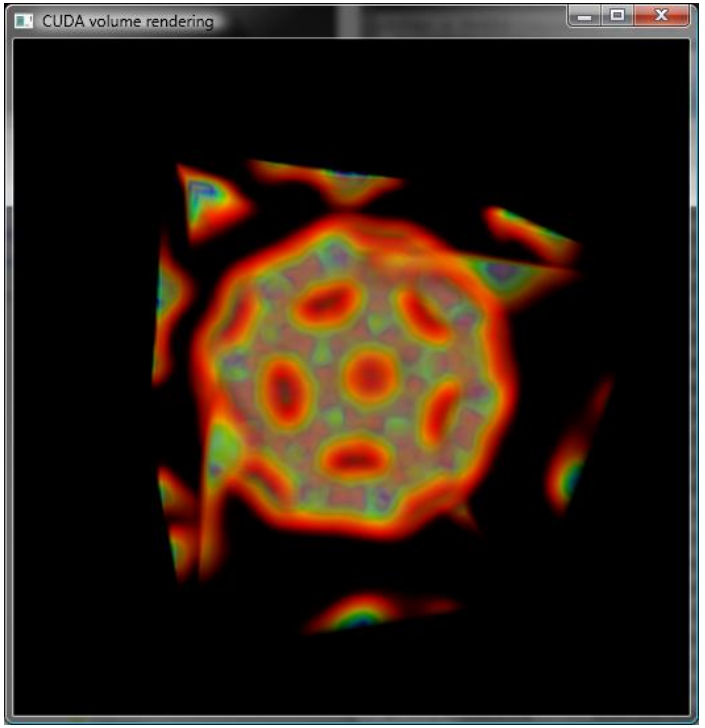


Some possibilities

- Ray start position and exit check
- Ray start position and exit position
- Ray start position and direction vector

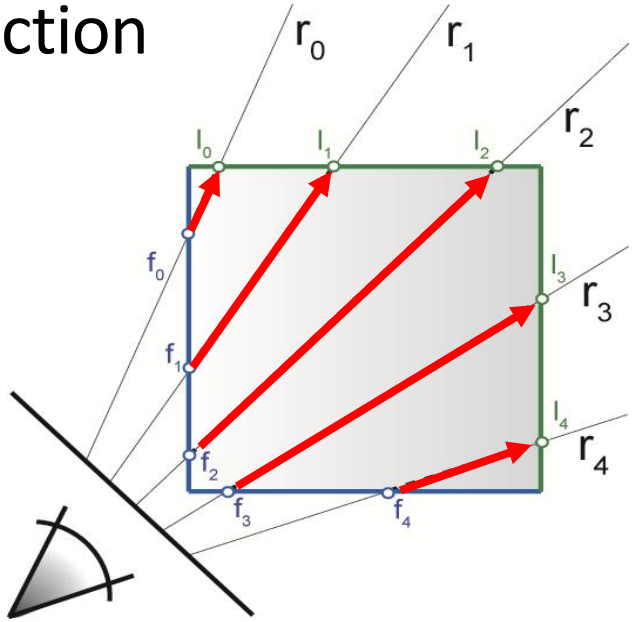
Single-Pass Ray Casting

- 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 (procedural ray setup)
- Alternative: image-based ray setup



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

CUDA Ray-Box Intersection Test

```
__device__ bool intersect(float & fNear, float & fFar, float3 vecOrigin, float3 vecDirection, float3 vecMinimumBounds, float3 vecMaximumBounds)
{
    // compute intersection of ray with all six bbox planes
    float3 fInverse = make_float3(1.0f) / vecDirection;
    float3 vecBottom = fInverse * (vecMinimumBounds - vecOrigin);
    float3 vecTop = fInverse * (vecMaximumBounds - vecOrigin);

    // re-order intersections to find smallest and largest on each axis
    float3 vecMinimum = fminf(vecTop, vecBottom);
    float3 vecMaximum = fmaxf(vecTop, vecBottom);

    // find the largest tmin and the smallest tmax
    float fNear = fmaxf(fmaxf(vecMinimum.x, vecMinimum.y), fmaxf(vecMinimum.x, vecMinimum.z));
    float fFar = fminf(fminf(vecMaximum.x, vecMaximum.y), fminf(vecMaximum.x, vecMaximum.z));

    return fFar > fNear;
};
```

http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm

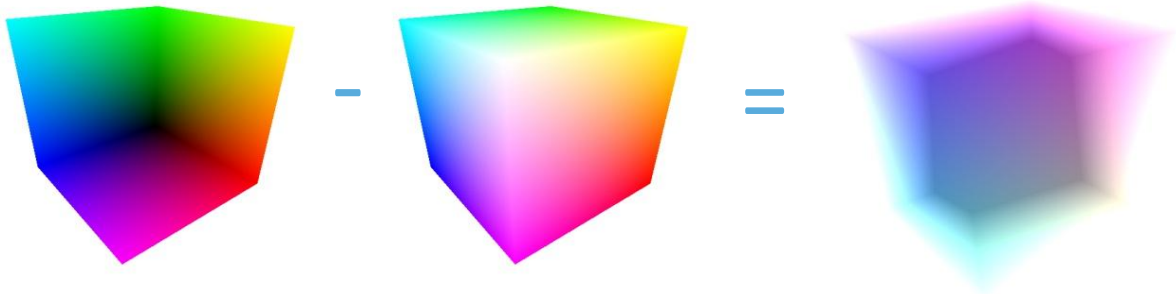
Fragment Shader

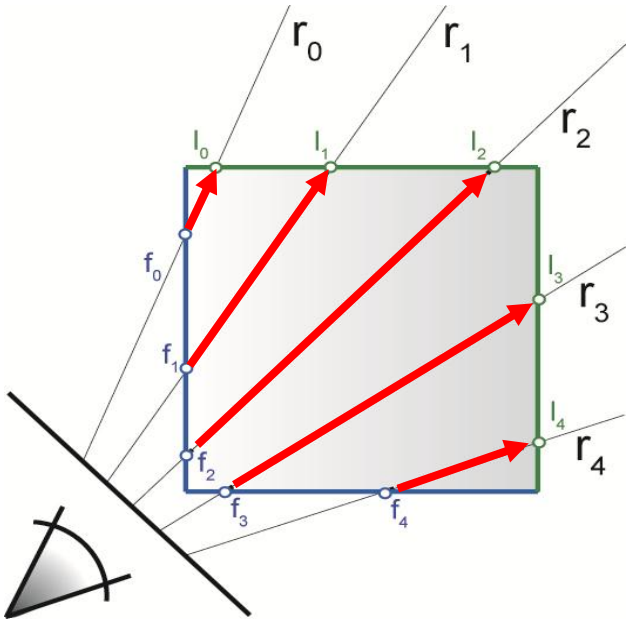
- Rasterize front faces of volume bounding box
- Texcoords are volume position in [0,1]
- Subtract camera position
- Repeatedly check for exit of bounding box

```
// 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;
}
```

Rasterization-Based Ray Setup

- Fragment == ray
- Need ray start pos, direction vector
- Procedural ray/box intersection;
  or rasterize bounding box





- Identical for orthogonal and perspective projection!

CUDA Kernel

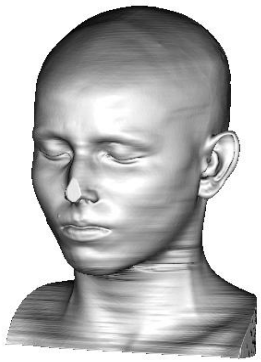
- Image-based ray setup
  - Ray start image
  - Direction image
- Ray-cast loop
  - Sample volume
  - Accumulate color and opacity
- Terminate
- Store output

```
__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_indx = ( __umul24( screencoord_y, cu_screensize.x ) + screencoord_x ) * 4;

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

    // ray-casting loop
    float4 color     = make_float4( 0.0f );
    float poscount   = 0.0f;
    for ( int i = 0; i < 8192; i++ ) {

        // next sample position in volume space
        float3 samplepos = dir_vec * poscount + startpos;
        poscount += cu_sampling_distance;

        // fetch density
        float tex_density = tex3D( cu_volume_texture, samplepos.x, samplepos.y, samplepos.z );

        // apply transfer function
        float4 col_classified = tex1D( cu_transfer_function_texture, tex_density );

        // compute (1-previous.a)*tf.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 ( ( poscount > dir_vec.w ) || ( color.w > 0.98f ) ) {
            break;
        }
    }

    // store output color and opacity
    d_output_buffer[ screencoord_indx ] = __saturatef( color );
}
```

Standard Ray-Casting Optimizations (1)

Early ray termination

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



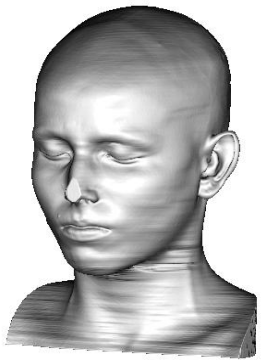
Several possibilities

- Older GPUs: multi-pass rendering with early-z test
- Shader model 3: break out of ray-casting loop
- Current GPUs: early loop exit works well

Standard Ray-Casting Optimizations (2)

Empty space skipping

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



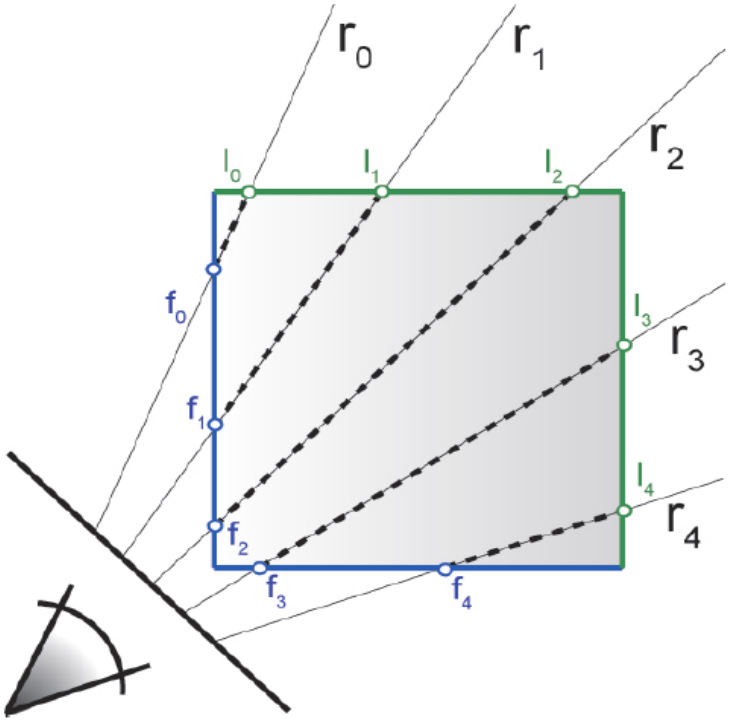
Several possibilities

- Per-sample check of opacity (expensive)
- Traverse hierarchy (e.g., octree) or regular grid

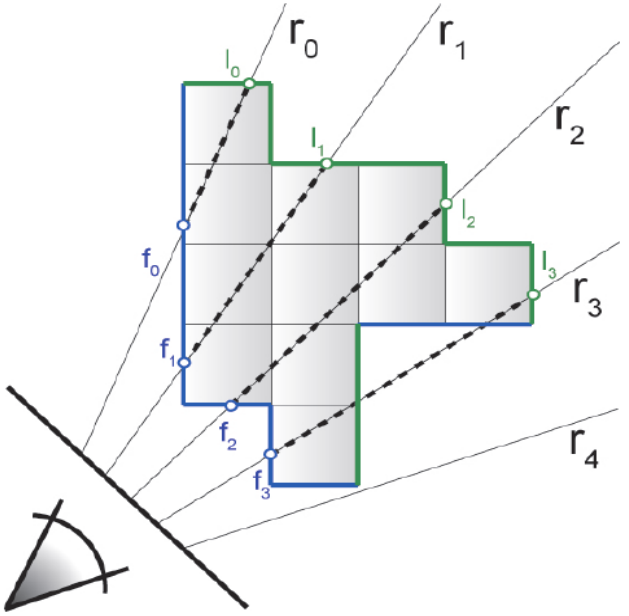
- These are image-order: what about object-order?

Object-Order Empty Space Skipping

- Modify initial rasterization step



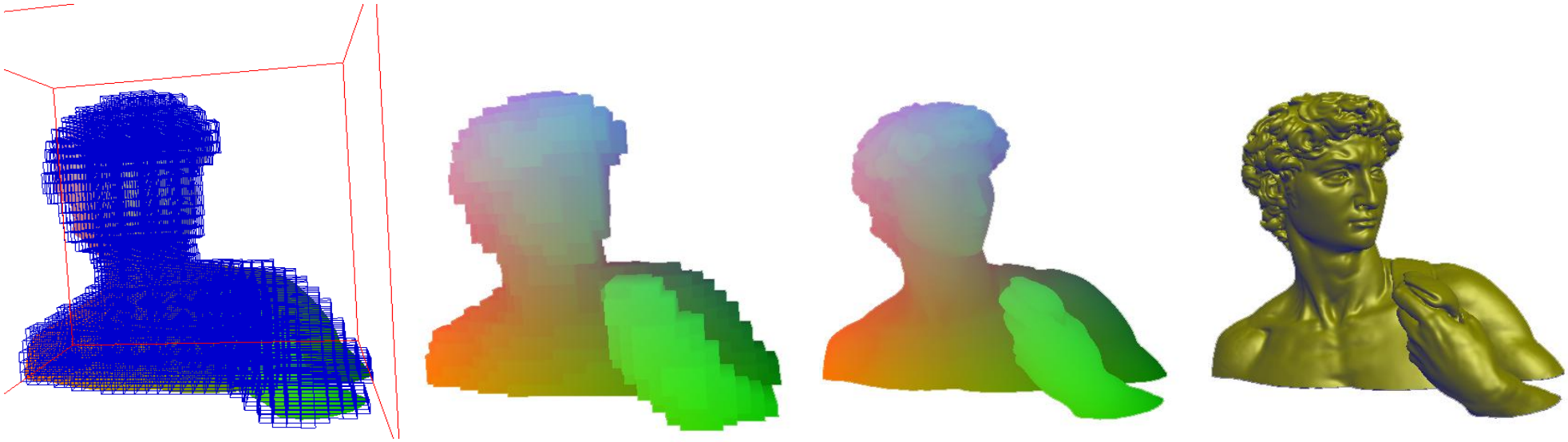
rasterize bounding box



rasterize “tight" bounding geometry

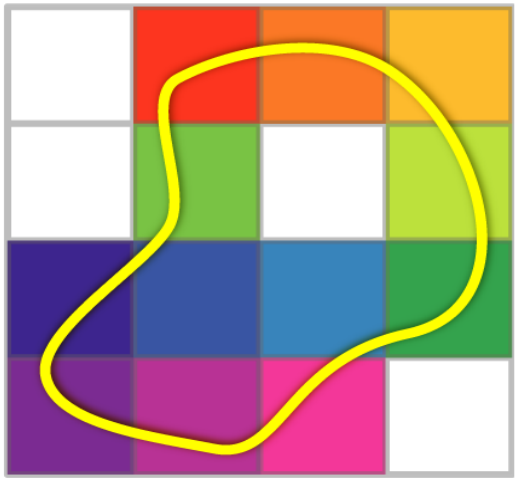
Object-Order Empty Space Skipping

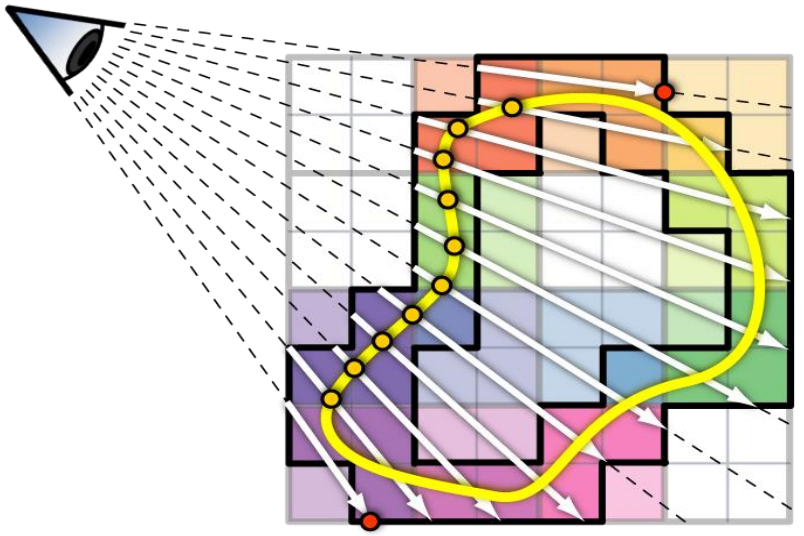
- Store min-max values of volume blocks
- Cull blocks against transfer function or iso value
- Rasterize front and back faces of active blocks



Object-Order Empty Space Skipping

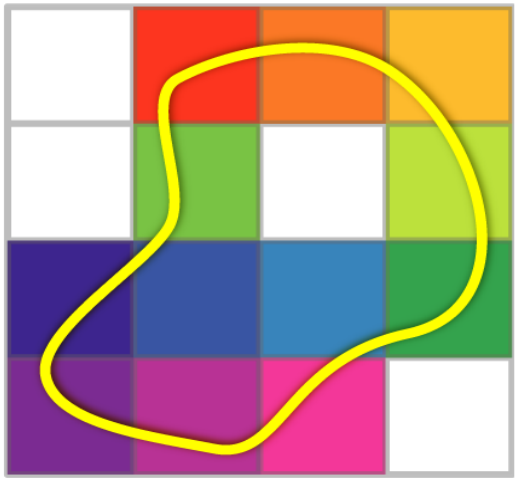
- Rasterize front and back faces of active min-max bricks
- Start rays on brick front faces
- Terminate when
  - Full opacity reached, or
  - Back face reached

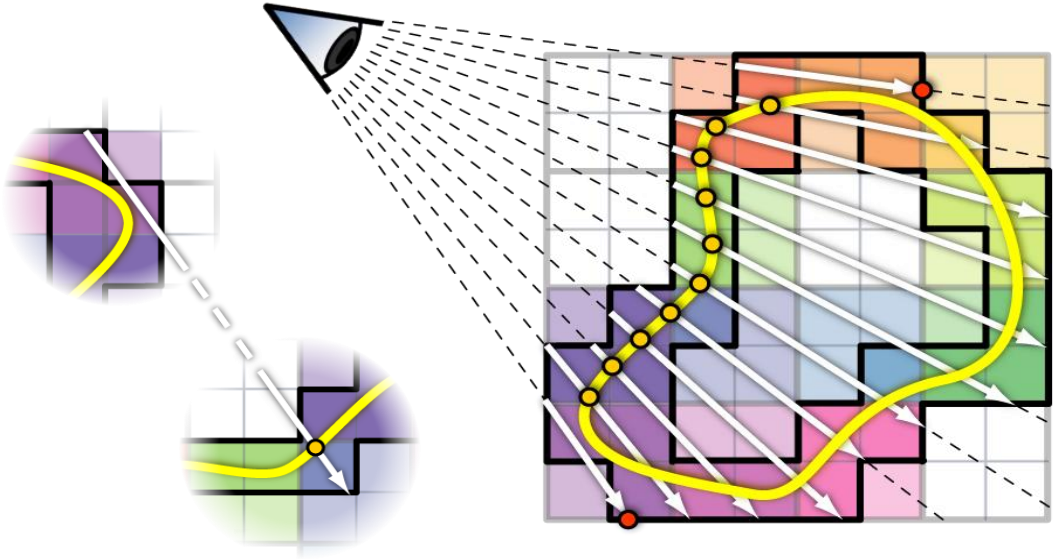




Object-Order Empty Space Skipping

- 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 skipped

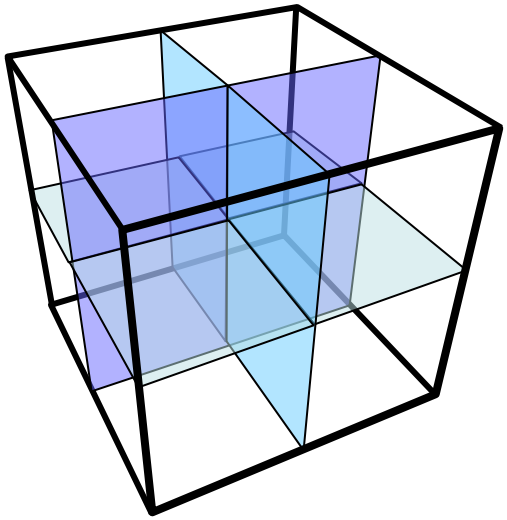




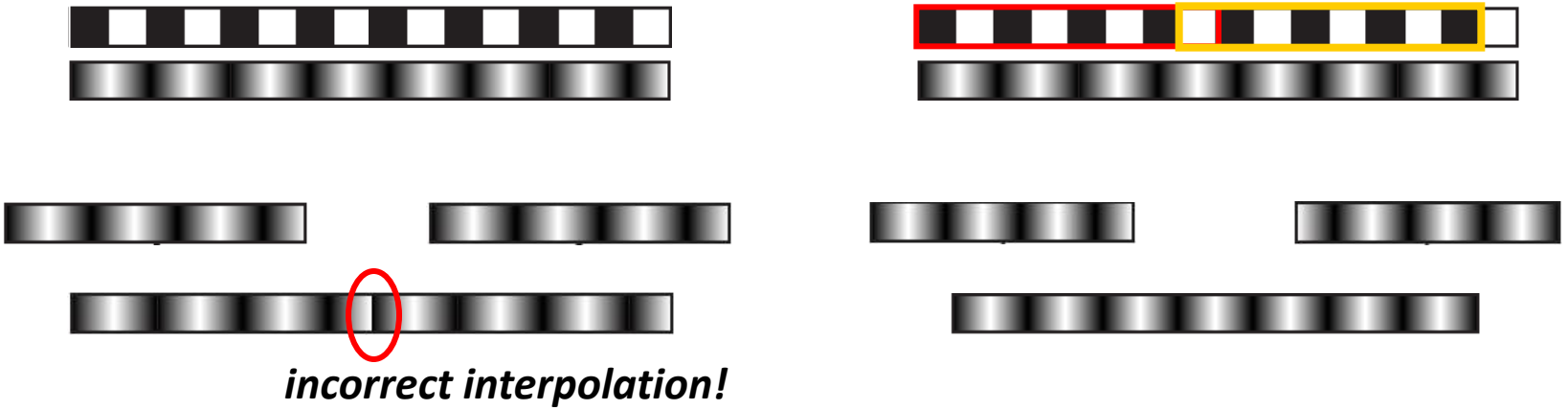
Memory Management

- What happens if data set is too large to fit into local GPU memory?

⇨ Divide data set into smaller chunks (bricks)

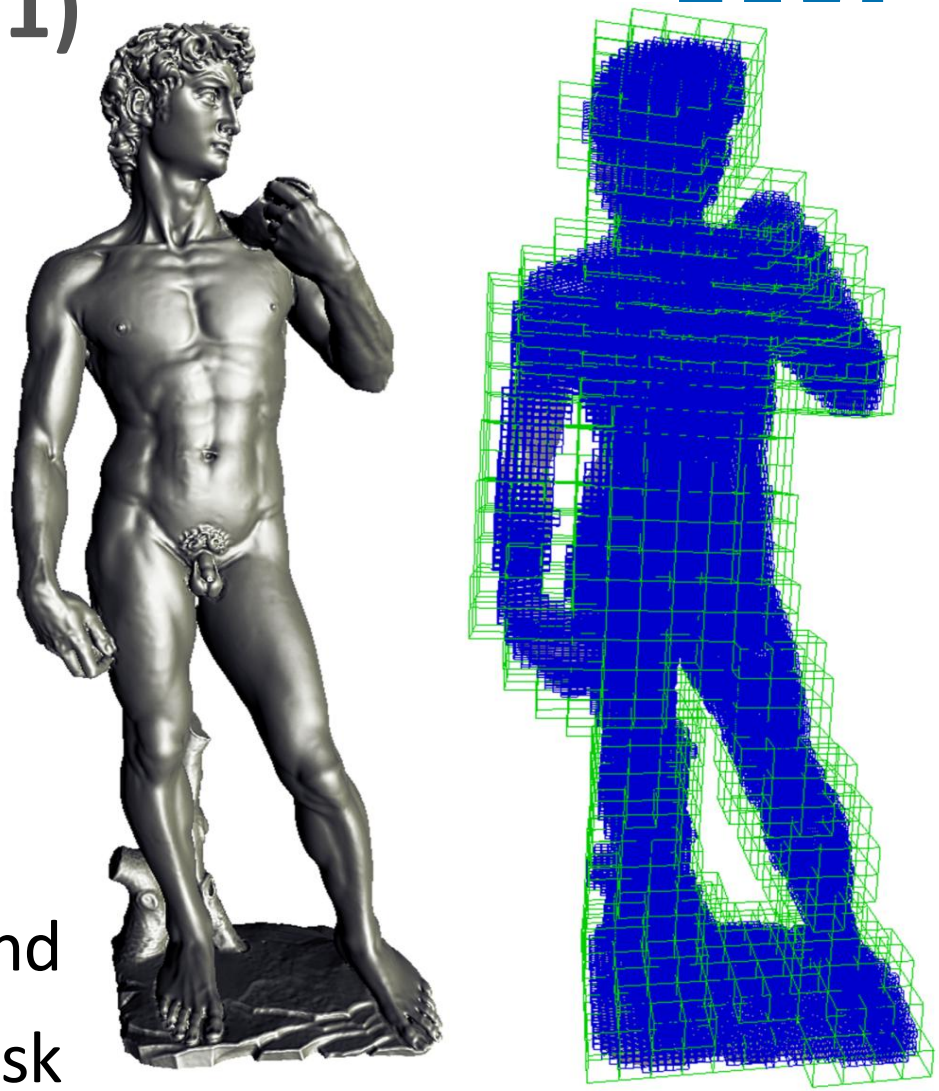


One plane of voxels must be duplicated for correct interpolation across brick boundaries



Simple Volume Bricking (1)

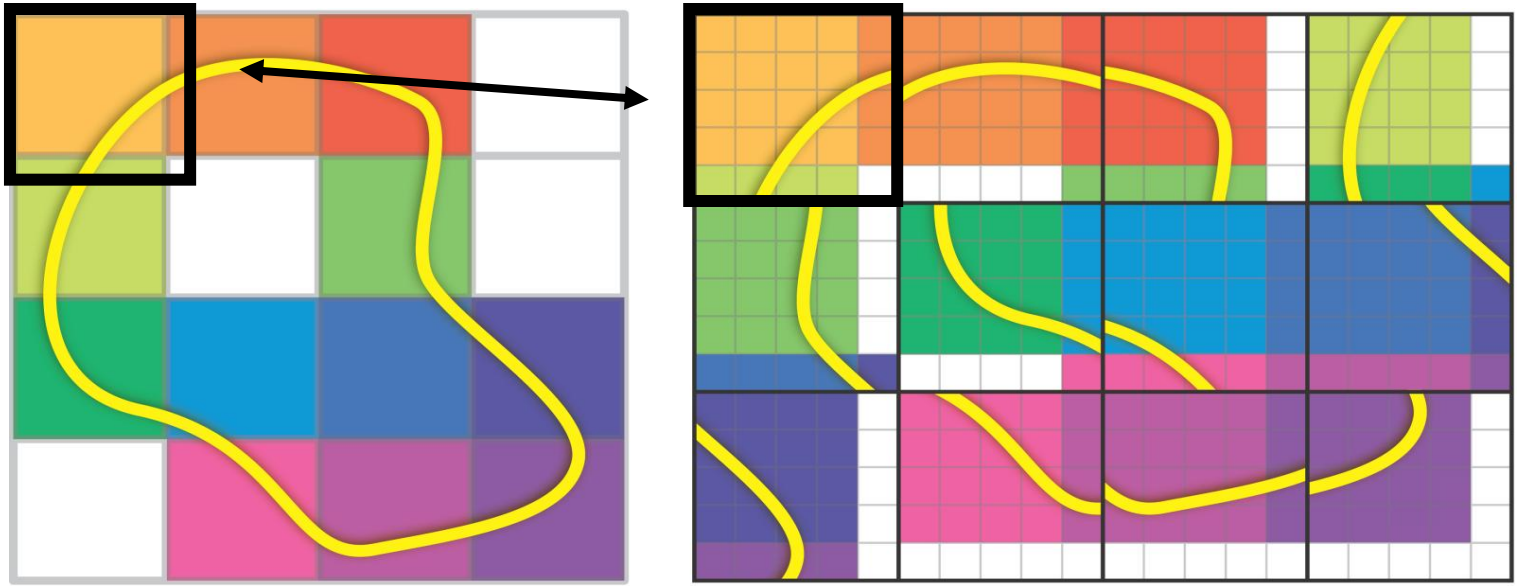
- Two subdivision levels
- Coarse bricks stored in cache texture (green)
- Finer blocks for object order empty space skipping (blue)
- Min-max of blocks/bricks
- Swap in/out bricks from cache texture on demand
- Out-of-core brick cache on disk



1536x576x352

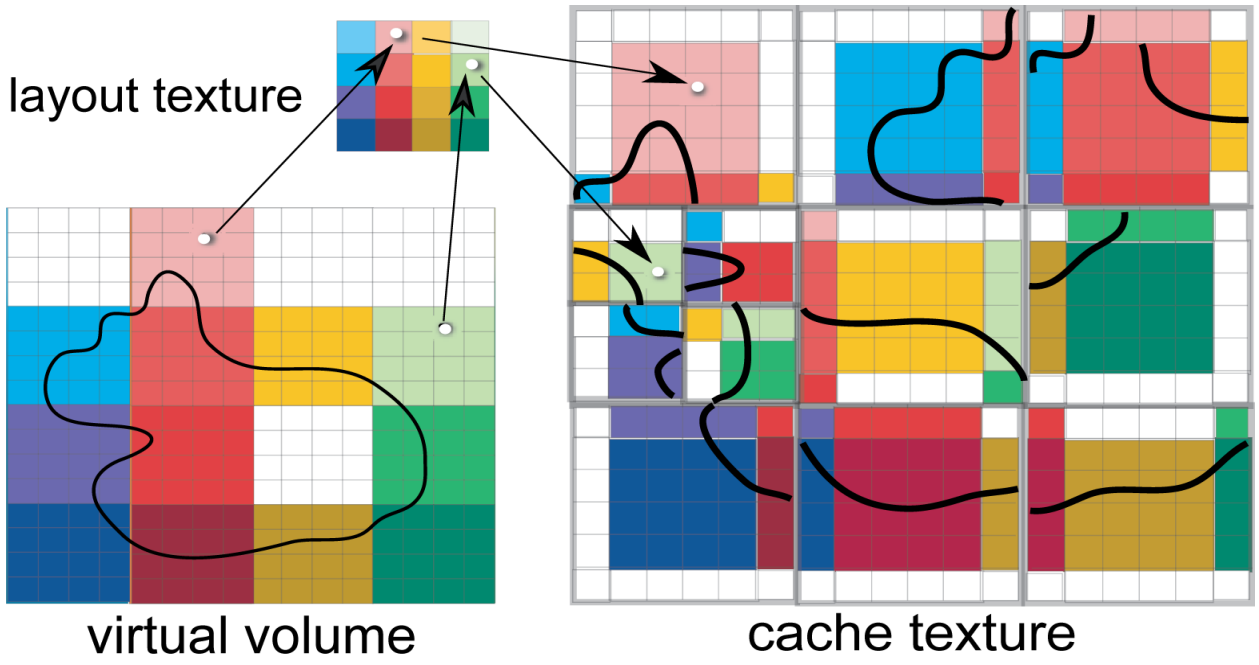
Simple Volume Bricking (2)

- Duplicate neighbor voxels for full-speed filtering
- Store n3 bricks as $(n+1)^3$
  - 10% overhead with 32^3 bricks



Simple Volume Bricking (3)

- Layout/index texture for address translation
- Supports multi-resolution rendering
- Map virtual volume coords to physical cache texture



Address Translation (1)

Conceptual approach

virtual volume position

scale factor according to resolution

$$\mathbf{x}'_{x,y,z} = (\mathbf{x}_{x,y,z} - \mathbf{brickPos}_{virtVol}) \cdot \mathbf{t}_w + \mathbf{t}_w + \mathbf{brickPos}_{brickCache}$$

final coordinate in cache texture

brick origin in virtual volume

brick position in cache

additional addition to account for duplicated border

Address Translation (2)

Translation in shader

output input from single lookup in layout texture

$$\mathbf{x}'_{x,y,z} = \mathbf{x}_{x,y,z} \cdot \mathbf{bscale}_{x,y,z} \cdot \mathbf{t}_w + \mathbf{t}_{x,y,z}$$

constant scaling factor

$$\mathbf{bscale}_{x,y,z} = \mathbf{vsize}_{x,y,z} / \mathbf{csize}_{x,y,z}$$

volume size

cache size

Address Translation (3)

Layout texture generation

index of brick in cache

cache texture size in texels

$$\mathbf{t}_{x,y,z} = \left(\mathbf{b}'_{x,y,z} \cdot \mathbf{bres}'_{x,y,z} - \mathbf{o}_{x,y,z} \right) / \mathbf{csize}_{x,y,z}$$

storage resolution of brick

original brick resolution

$$\mathbf{o}_{x,y,z} = \mathbf{b}_{x,y,z} \cdot \mathbf{bres}_{x,y,z} \cdot \mathbf{t}_w - \mathbf{t}_w$$

index of brick in volume

resolution scale

Thanks!

- Markus Hadwiger
- Christof Rezk-Salama
- Klaus Engel
- Henning Scharsach
- Johanna Beyer