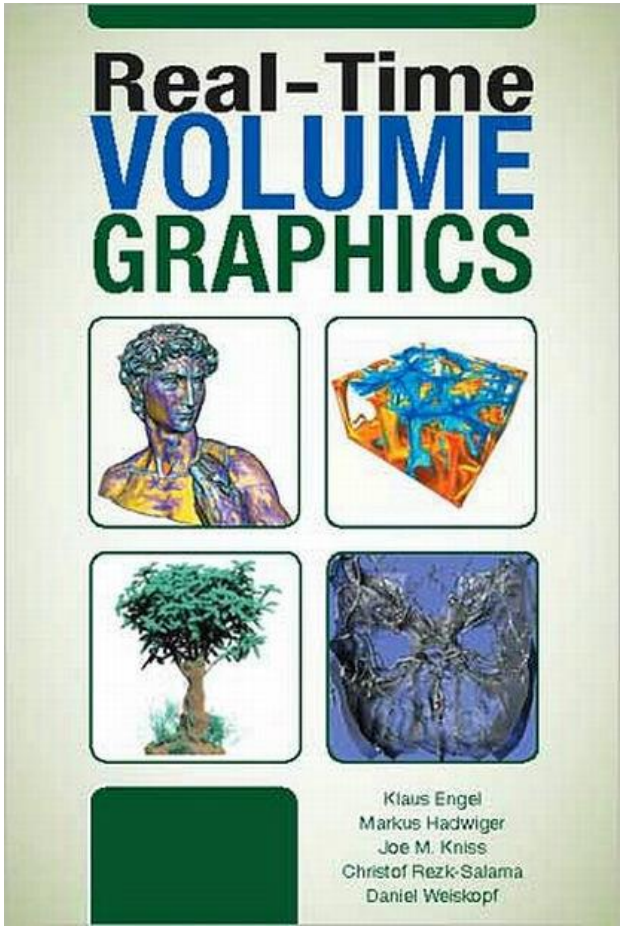
Real-time Graphics

10. GPU Volume Graphics

Martin Samuelčík

Book

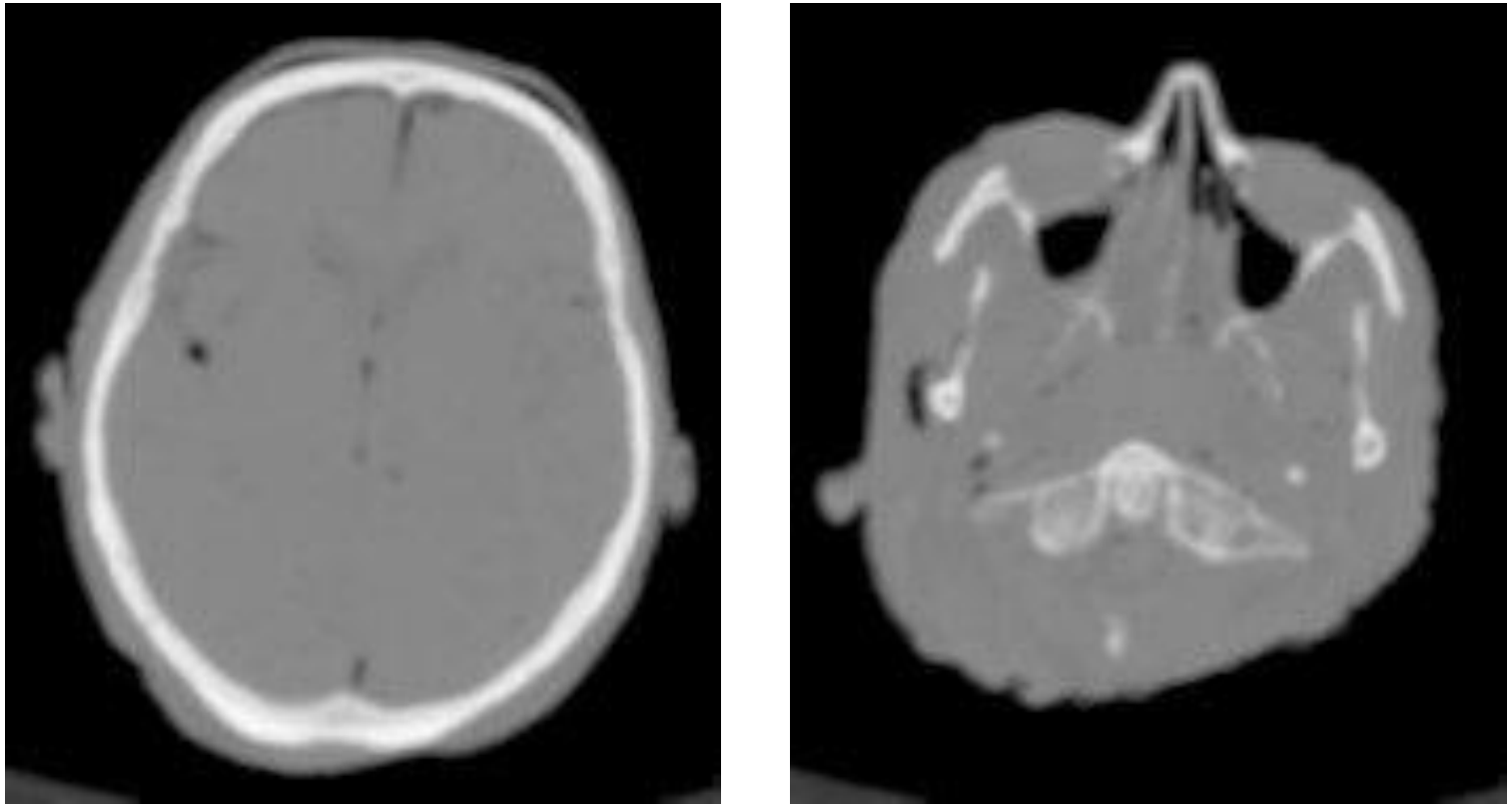
- Real-Time Volume Graphics
  - Klaus Engel
  - Markus Hadwiger
  - Joe M. Kniss
  - Christof Rezk-Salama
  - Daniel Weiskopf



Volume Data

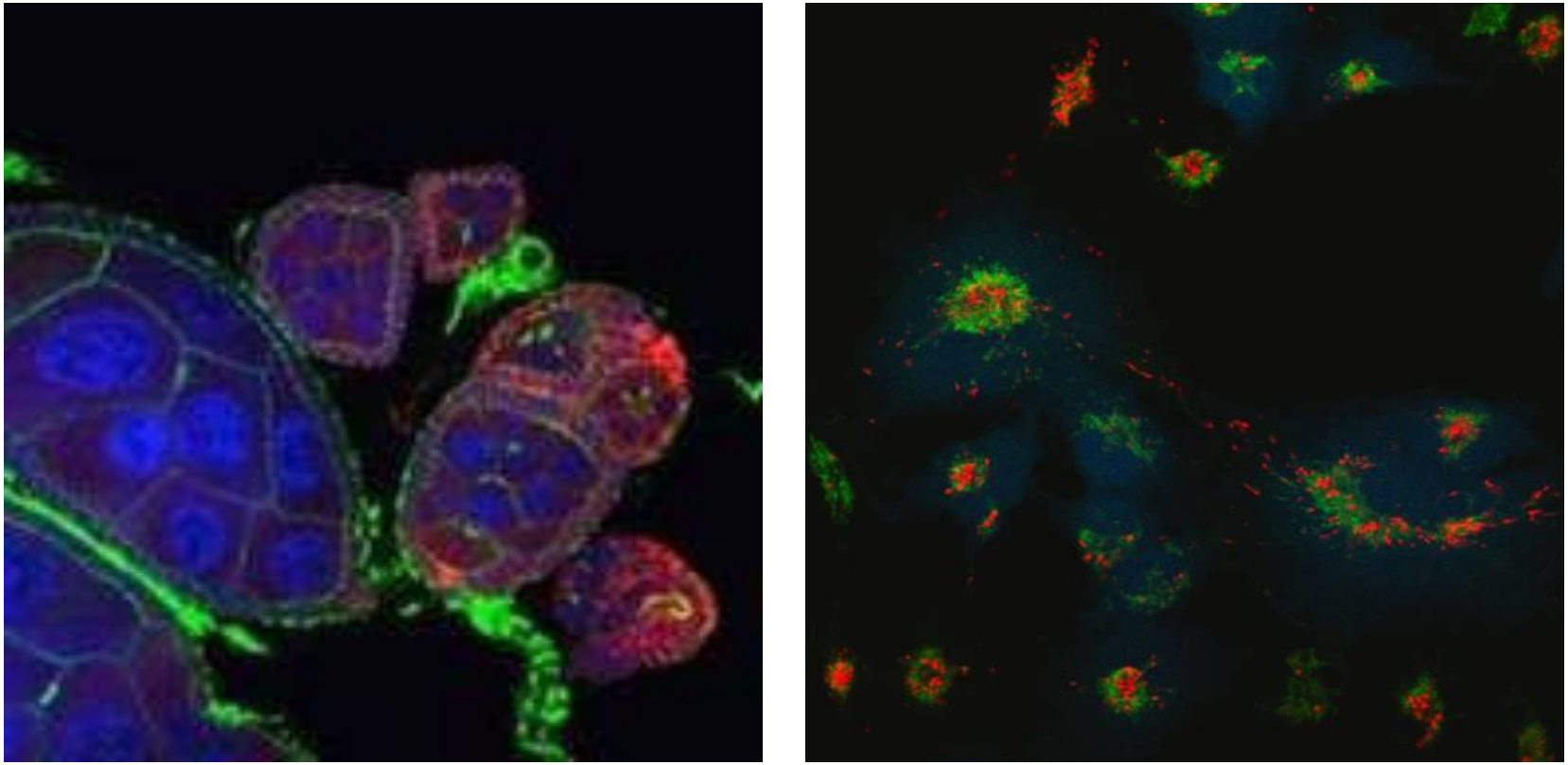
- Describes also inner structures of 3D objects
- Usage in medicine, geology, physics, …
- Measurement, simulations of tissues, bones, liquids, gases, fire, natural phenomena
- Discrete data
  - Volume elements - voxels
  - Set of voxels – organized in grids
    - Uniform, tetrahedral, …
  - Voxel – usually intensity value

Volume Data



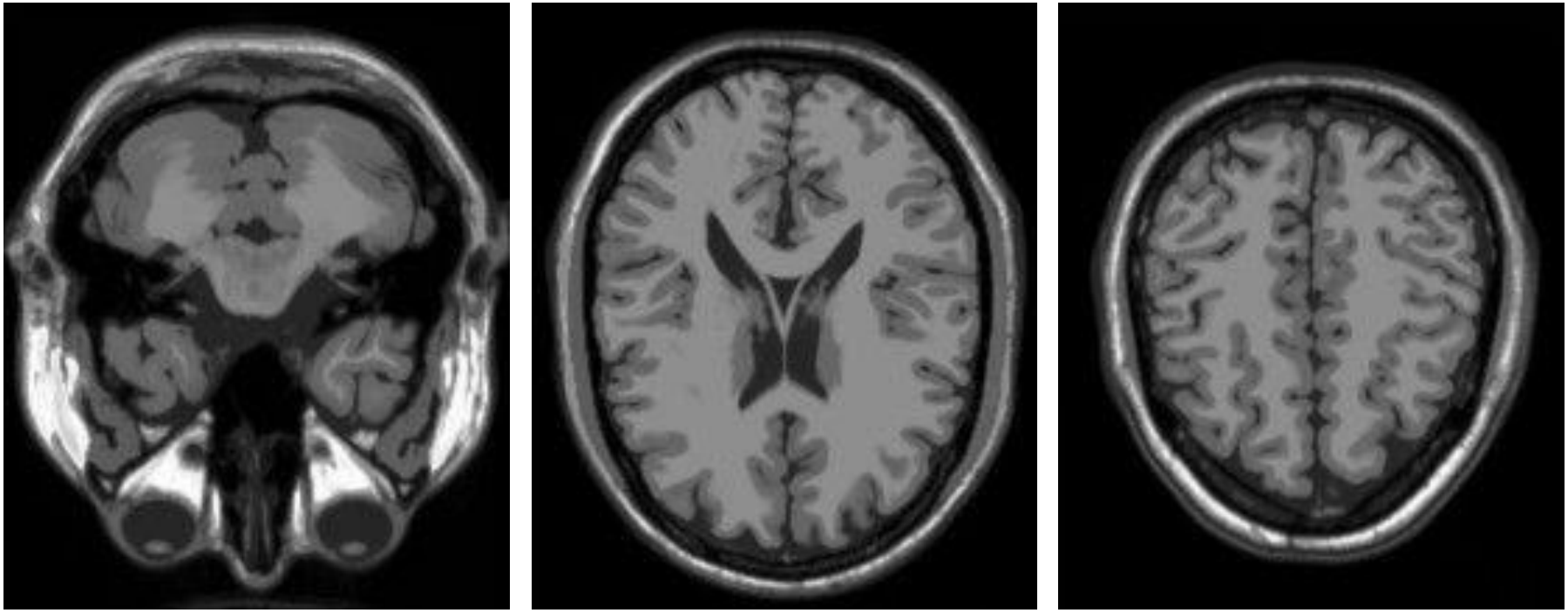
Computer Tomography dataset

Volume Data



Computer Tomography dataset

Volume Data

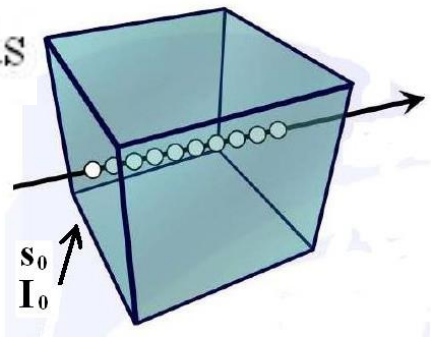


Magnetic Resonance Imaging dataset

Volume Rendering Integral

- Describes contribution of all volume values along viewing ray for final pixel color
- Based on physical model
- Emission, absorption, (in-out)scatter

$$I(D) = I_0 e^{-\int_{s_0}^{D} \kappa(t)dt} + \int_{s_0}^{D} q(s) e^{-\int_{s}^{D} \kappa(t)dt} ds$$



- Absorption $\kappa(t)$, emission $q(s)$
- Computational intensive
  - Numeric integration, Iterative computation

Iterative Computation

- Approximation of VRI by discretization
- Composition of discrete values along ray in some order (front-to-back, back-to-front, ..)

$$\Delta x = (D - s_0)/n \qquad \alpha_i = 1 - T_i \qquad T_i \approx e^{-\kappa(s_i)\Delta x} \qquad c_i \approx q(s_i)\Delta x$$

- Step n, front-to-back (i=n-1,..,0):

$$\hat{C}_i = \hat{C}_{i+1} + \hat{T}_{i+1} C_i\,, \qquad \hat{C}_n = C_n\,,$$
$$\hat{T}_i = \hat{T}_{i+1}(1 - \alpha_i)\,, \qquad \hat{T}_n = 1 - \alpha_n\,.$$

$$C_{\mathrm{dst}} \leftarrow C_{\mathrm{dst}} + (1 - \alpha_{\mathrm{dst}}) C_{\mathrm{src}}\,,$$
$$\alpha_{\mathrm{dst}} \leftarrow \alpha_{\mathrm{dst}} + (1 - \alpha_{\mathrm{dst}}) \alpha_{\mathrm{src}}\,.$$

- Step n, back-to-front (i=1,..,n):

$$\hat{C}_i = \hat{C}_{i-1}(1 - \alpha_i) + C_i\,, \qquad \hat{C}_0 = C_0\,,$$
$$\hat{T}_i = \hat{T}_{i-1}(1 - \alpha_i)\,, \qquad \hat{T}_0 = 1 - \alpha_0\,.$$

$$C_{\mathrm{dst}} \leftarrow (1 - \alpha_{\mathrm{src}}) C_{\mathrm{dst}} + C_{\mathrm{src}}\,.$$

Transfer Function

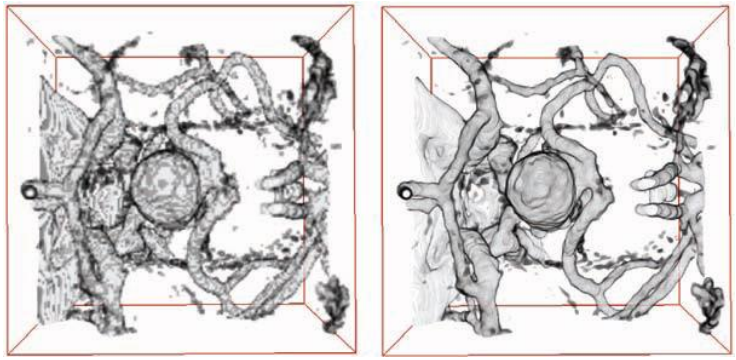
- Voxels holds scalar values
- How to map (transfer) values to emission, absorption, reflectivity, translucency, …
- Defining rendering parameters (color, transparency, …)
- Data-centric (contour spectrum, isosurfaces, curvature, boundary detection, LH histogram…)
- Image-centric (genetic alg., design galleries, …)
- Others (learning classifier, …)

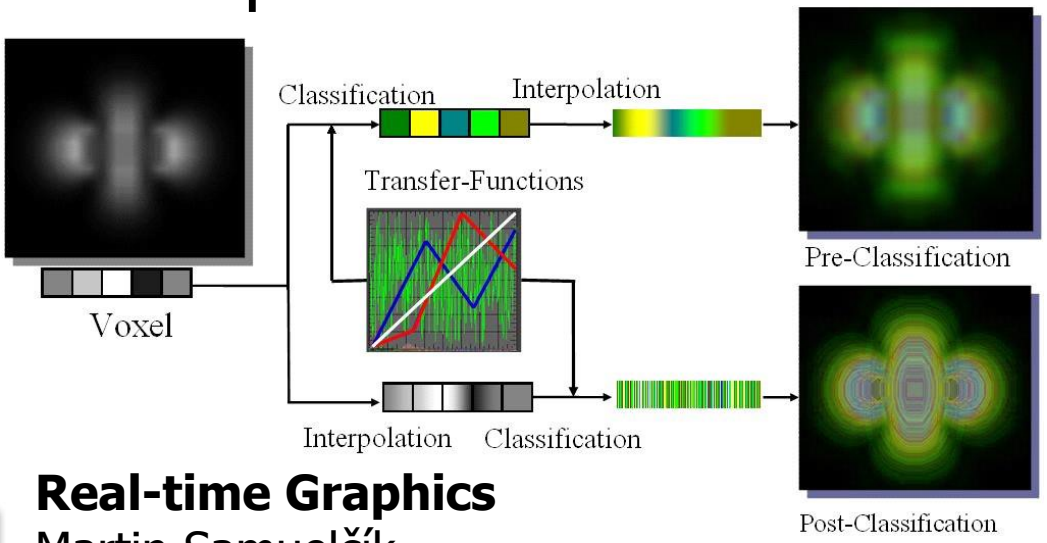
Transfer Function

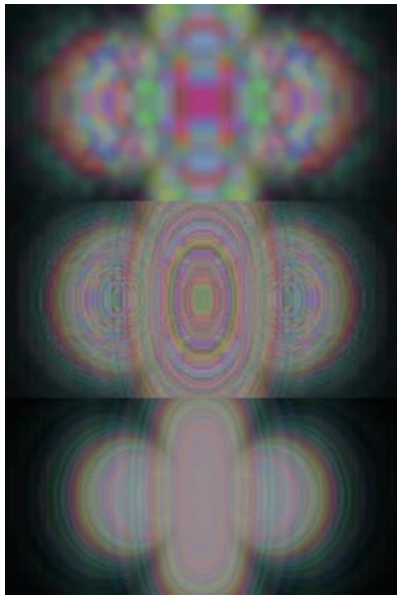
- Classification of voxels
- Discretization -> Look-up tables
- Application of transfer function
  - Pre-classification – transformation is applied to input textures, before interpolation
  - Post-classification – transformation is applied to interpolated value fetched from slice textures
  - Pre-integration – piece of volume integral is pre-integrated in 2D look-up table

Transfer Function

- Pre-classification
  - glPixelTransfer, glPixelMap
  - GL_EXT_paletted_texture
- Post-classification
  - Interpolation in 3D textures

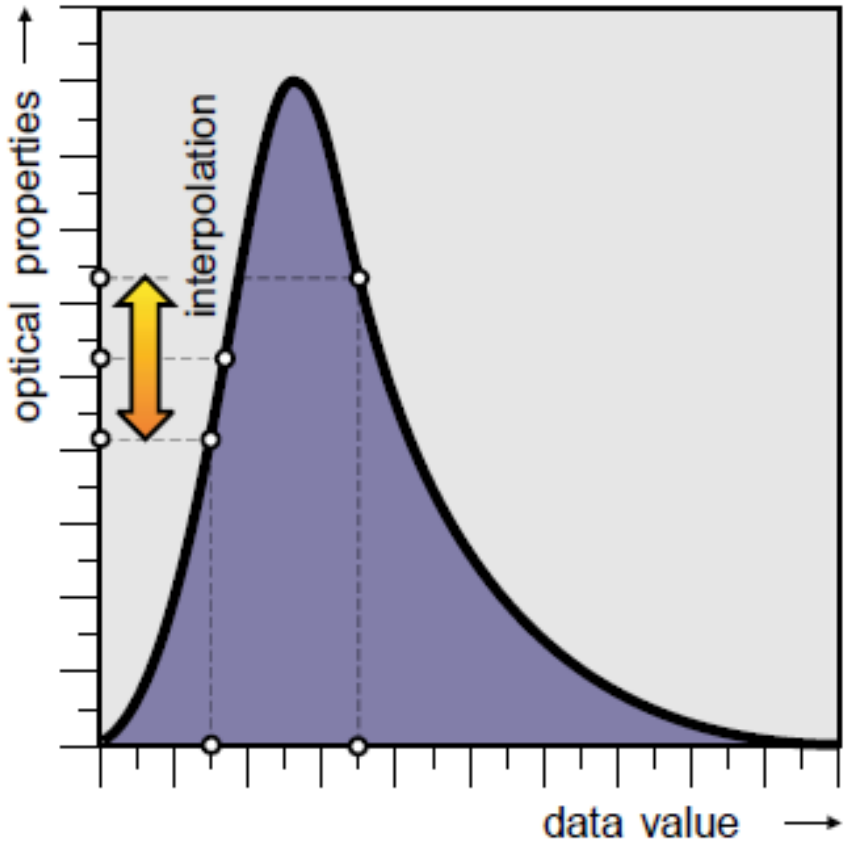




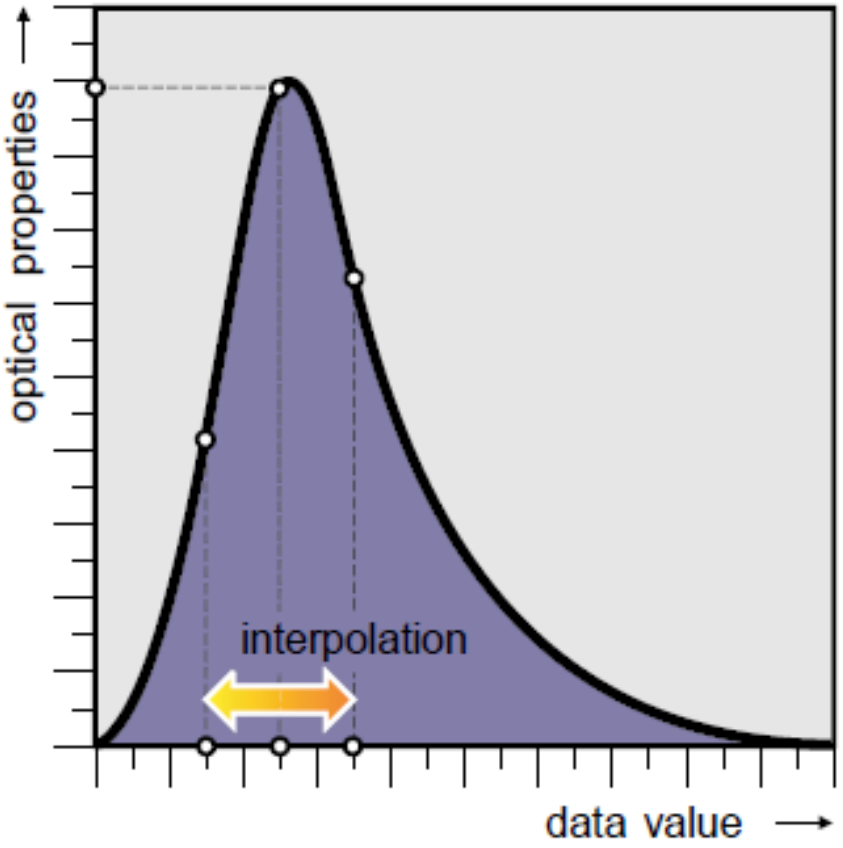


Transfer Function

Pre-classification



Post-classification

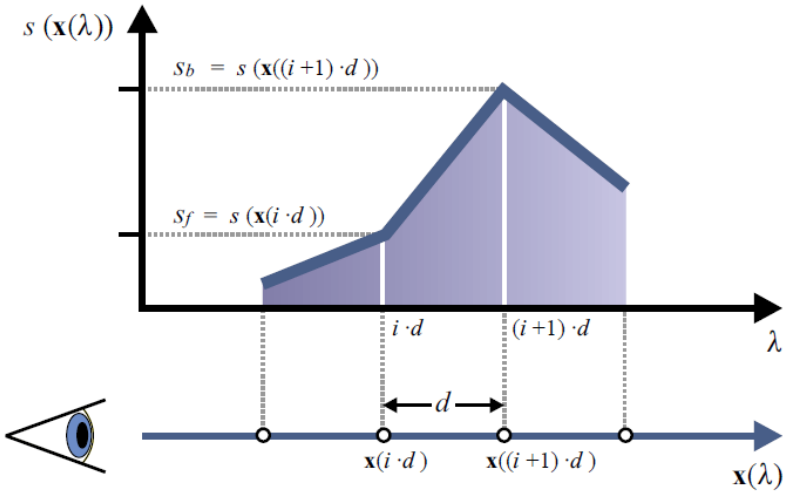


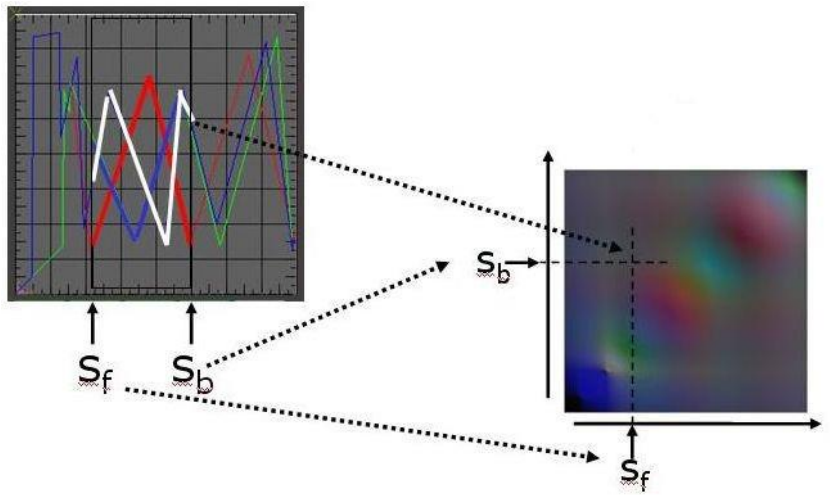
Transfer Function

- Pre-integration
  - Slab by slab rendering
  - Pre integration of all possible combinations
  - Look-up table - texture

$$\alpha_i = 1 - \exp\left(-\int_{i\,d}^{(i+1)d} \kappa\big(s(\mathrm{x}(\lambda))\big)\mathrm{d}\lambda\right)$$
$$\approx 1 - \exp\left(-\int_0^1 \kappa\big((1-\omega)s_f + \omega s_b\big)d\,\mathrm{d}\omega\right)$$

$$c_i \approx \int_0^1 q\big((1-\omega)s_f + \omega s_b\big) \times \exp\left(-\int_0^\omega \kappa\big((1-\omega')s_f + \omega' s_b\big)d\,\mathrm{d}\omega'\right)d\,\mathrm{d}\omega$$



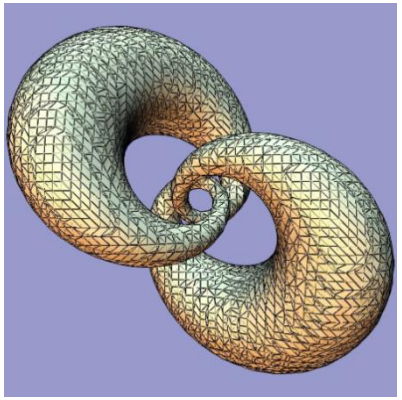


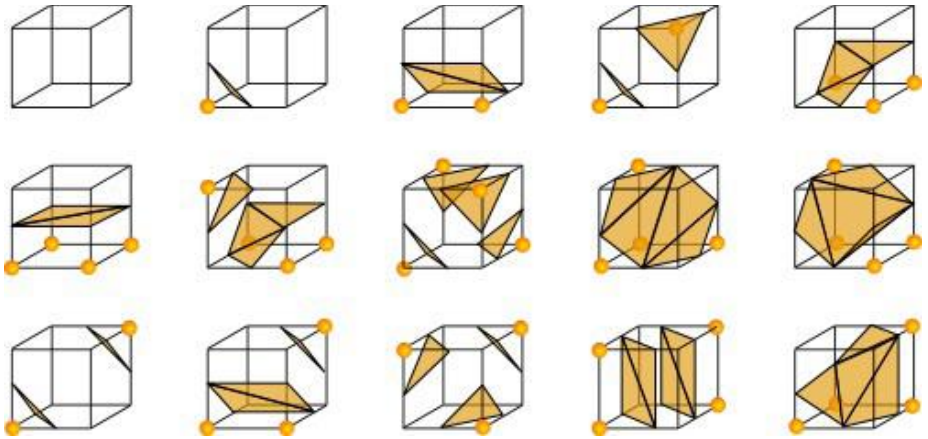
Volume Rendering

- Direct
  - Defining opacity and color of each voxel
  - Projection of volume from 3D to 2D
  - Rendering whole volume on screen, integrating voxels along viewing rays
- Indirect
  - Extraction of iso-surfaces for given iso-values in dataset
  - Rendering just surface as set of triangles
  - Contour detection

Marching Cubes

- Isosurface – set of points with same intensity value inside volume
- Extraction of isosurface from uniform grid
- For each grid cell, determine triangles inside cell by 8 corner values





GPU Marching Cubes

- Divide volume space into uniform cells
- Render cell as middle point
- Using geometry shader to generate triangles for each cell
- For each cell
  - Get values for corners of cell
  - Check if corners are inside or outside of iso-surface by subtracting by iso-value
  - Using triangle index texture, generate triangles from interpolated edge points

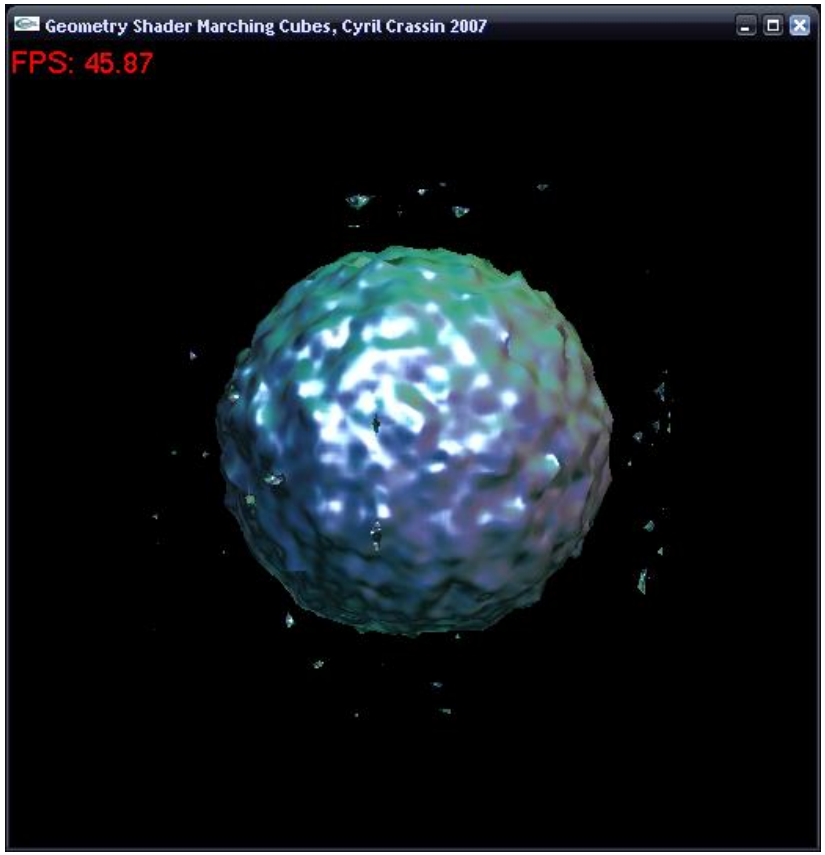
GLSL Marching Cubes

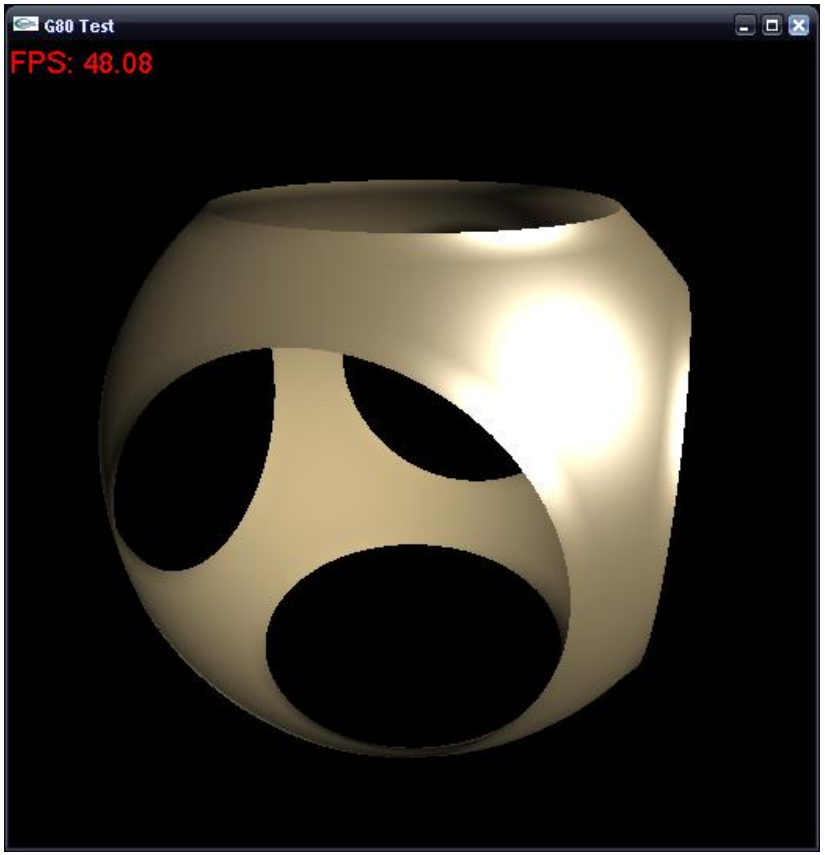
```
//GLSL version 1.20
#version 120
#extension GL_EXT_geometry_shader4 : enable
#extension GL_EXT_gpu_shader4 : enable

//Volume data field texture
uniform sampler3D dataFieldTex;
//Triangles table texture
uniform isampler2D triTableTex;
//Global iso level
uniform float isolevel;
//Marching cubes vertices decal
uniform vec3 vertDecals[8];
//Vertices position for fragment shader
varying vec4 position;

//Get vertex i position within current marching cube
vec3 cubePos(int i){
    return gl_PositionIn[0].xyz + vertDecals[i];
}

//Get vertex i value within current marching cube
float cubeVal(int i){
    return texture3D(dataFieldTex, (cubePos(i)+1.0f)/2.0f).a;
}

//Get triangle table value
int triTableValue(int i, int j){
    return texture2D(triTableTex, ivec2(j, i), 0).a;
}

//Compute interpolated vertex along an edge
vec3 vertexInterp(float isolevel, vec3 v0, float l0, vec3 v1, float l1){
    return mix(v0, v1, (isolevel-l0)/(l1-l0));
}
```

```
//Geometry Shader entry point
void main(void) {
    int cubeindex=0;
    float cubeVal0 = cubeVal(0);
    float cubeVal1 = cubeVal(1);
    float cubeVal2 = cubeVal(2);
    float cubeVal3 = cubeVal(3);
    float cubeVal4 = cubeVal(4);
    float cubeVal5 = cubeVal(5);
    float cubeVal6 = cubeVal(6);
    float cubeVal7 = cubeVal(7);

    //Determine the index into the triangle table
    cubeindex = int(cubeVal0 < isolevel);
    cubeindex += int(cubeVal1 < isolevel)*2;
    cubeindex += int(cubeVal2 < isolevel)*4;
    cubeindex += int(cubeVal3 < isolevel)*8;
    cubeindex += int(cubeVal4 < isolevel)*16;
    cubeindex += int(cubeVal5 < isolevel)*32;
    cubeindex += int(cubeVal6 < isolevel)*64;
    cubeindex += int(cubeVal7 < isolevel)*128;

    //Cube is entirely in/out of the surface
    if (cubeindex ==0 || cubeindex == 255)  return;

    vec3 vertlist[12];
    //Find the vertices where the surface intersects the cube
    vertlist[0] = vertexInterp(isolevel, cubePos(0), cubeVal0, cubePos(1), cubeVal1);
    vertlist[1] = vertexInterp(isolevel, cubePos(1), cubeVal1, cubePos(2), cubeVal2);
    vertlist[2] = vertexInterp(isolevel, cubePos(2), cubeVal2, cubePos(3), cubeVal3);
    vertlist[3] = vertexInterp(isolevel, cubePos(3), cubeVal3, cubePos(0), cubeVal0);
    vertlist[4] = vertexInterp(isolevel, cubePos(4), cubeVal4, cubePos(5), cubeVal5);
    vertlist[5] = vertexInterp(isolevel, cubePos(5), cubeVal5, cubePos(6), cubeVal6);
    vertlist[6] = vertexInterp(isolevel, cubePos(6), cubeVal6, cubePos(7), cubeVal7);
    vertlist[7] = vertexInterp(isolevel, cubePos(7), cubeVal7, cubePos(4), cubeVal4);
    vertlist[8] = vertexInterp(isolevel, cubePos(0), cubeVal0, cubePos(4), cubeVal4);
    vertlist[9] = vertexInterp(isolevel, cubePos(1), cubeVal1, cubePos(5), cubeVal5);
    vertlist[10] = vertexInterp(isolevel, cubePos(2), cubeVal2, cubePos(6), cubeVal6);
    vertlist[11] = vertexInterp(isolevel, cubePos(3), cubeVal3, cubePos(7), cubeVal7);
```

```
    // Create the triangle
    gl_FrontColor=vec4(cos(isolevel*5.0-0.5), sin(isolevel*5.0-0.5), 0.5, 1.0);
    int i=0;
    while(true){
        if(triTableValue(cubeindex, i)!=-1){
            //Generate first vertex of triangle//
            position= vec4(vertlist[triTableValue(cubeindex, i)], 1);
            gl_Position = gl_ModelViewProjectionMatrix* position;
            EmitVertex();
            //Generate second vertex of triangle//
            position= vec4(vertlist[triTableValue(cubeindex, i+1)], 1);
            gl_Position = gl_ModelViewProjectionMatrix* position;
            EmitVertex();
            //Generate last vertex of triangle//
            position= vec4(vertlist[triTableValue(cubeindex, i+2)], 1);
            gl_Position = gl_ModelViewProjectionMatrix* position;
            EmitVertex();
            //End triangle strip at firts triangle
            EndPrimitive();
        }else{
            break;
        }
         i=i+3;
    }
}
```

GLSL Marching Cubes





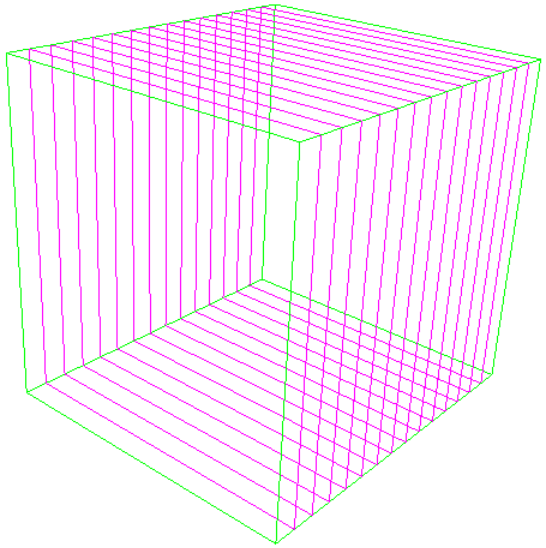
http://www.icare3d.org/blog_techno/gpu/opengl_geometry_shader_marching_cubes.html

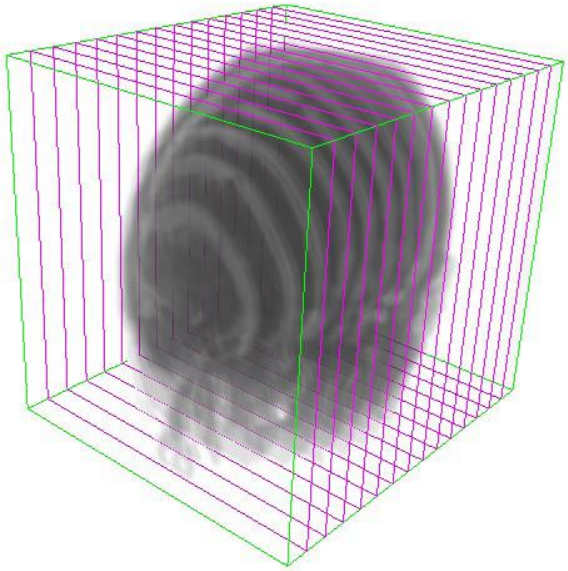
Direct Volume Rendering

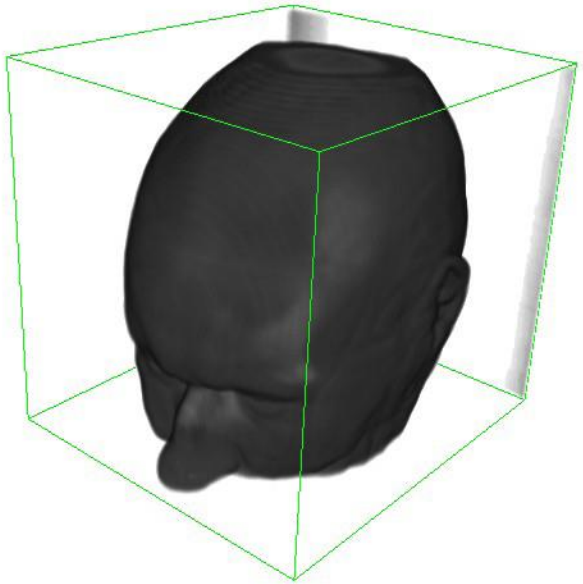
- Ray-casting
  - Rays are cast from eyes and VRI is accumulated along each viewing ray
- Texture slicing
  - Volume is sampled using object-aligned or image-aligned slices in some defined order
  - 2D textures
  - 3D textures
  - Multi-textures
- Shear-warp
  - Similar to texture slicing, slices are warped to better fit viewing direction

2D Textures

- Object aligned slices, rendered in front-to-back order
- Using alpha blending for composition

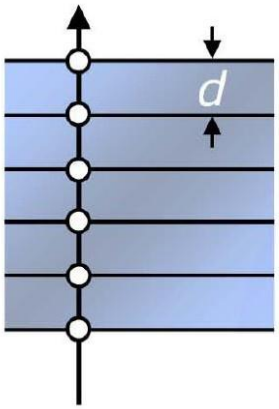


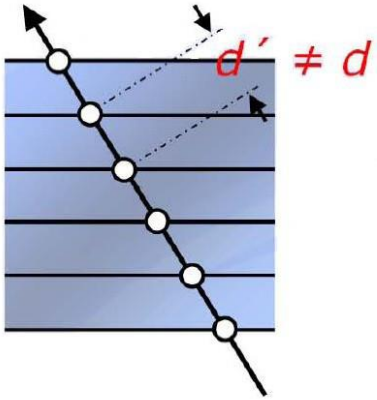


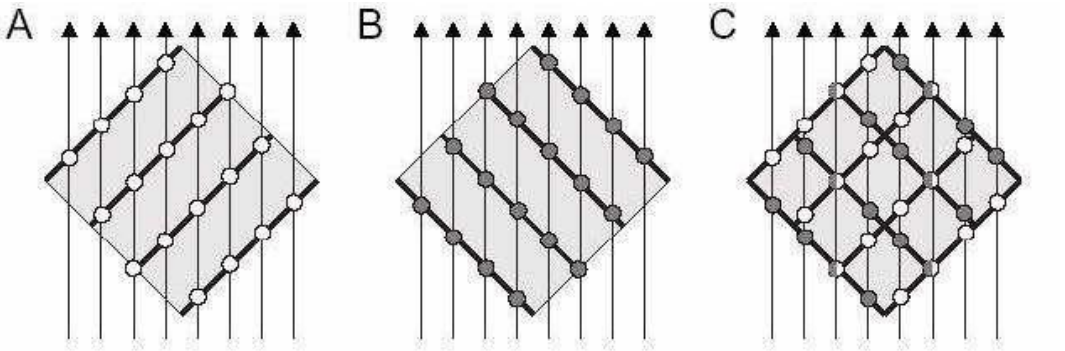


2D Textures

2D-Texture-Based Approach	
Pros	Cons
⊕ very high performance ⊕ high availability	⊖ high memory requirements ⊖ bilinear interpolation only ⊖ sampling artifacts ⊖ switching effects ⊖ inconsistent sampling rate

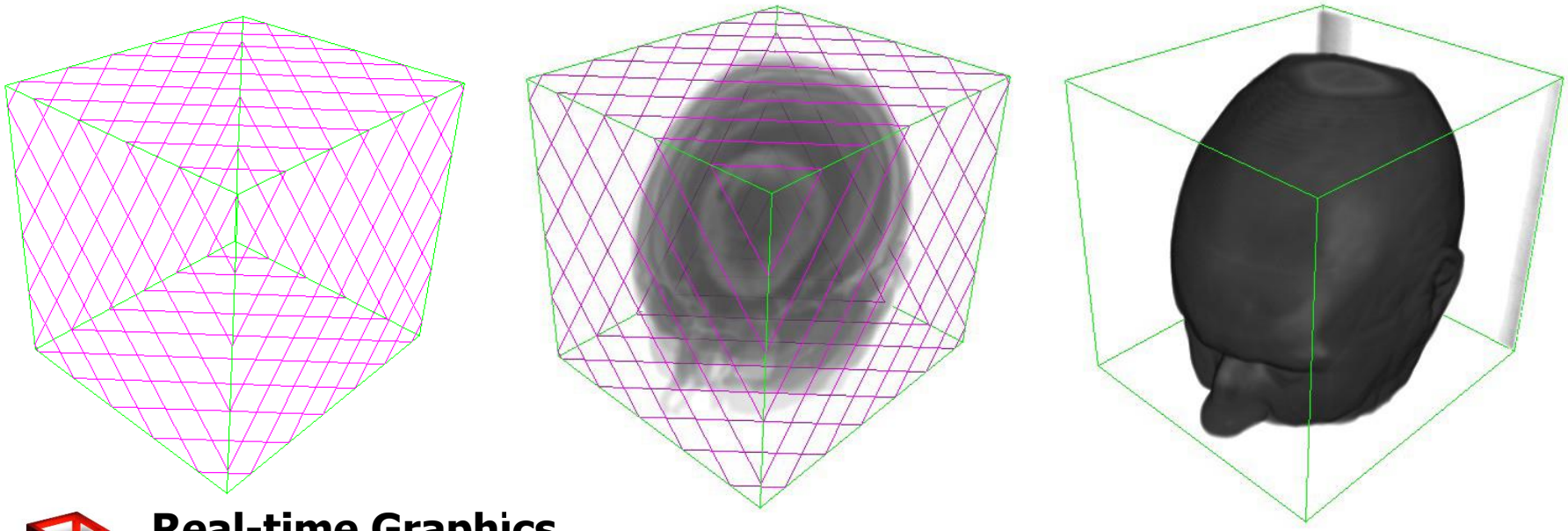






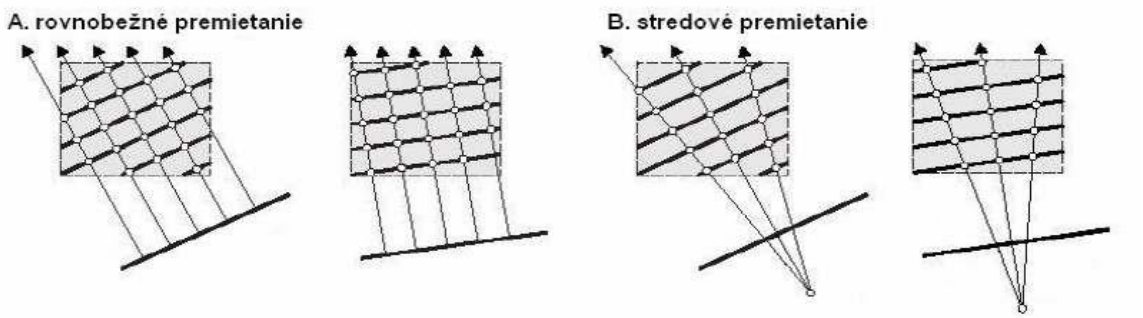
3D Textures

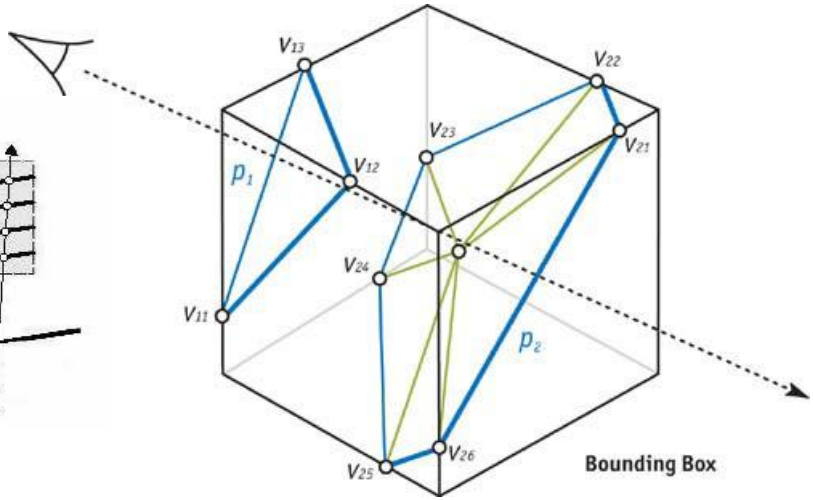
- View (image plane) aligned slices, rendered in front-to-back order
- Using alpha blending for composition
- GL_EXT_texture3D



3D Textures

3D-Texture-Based Approach	
Pros	Cons
⊕ high performance ⊕ trilinear interpolation	⊖ availability still limited ⊖ inefficient memory management ⊖ inconsistent sampling rate for perspective projection





3D Textures GLSL

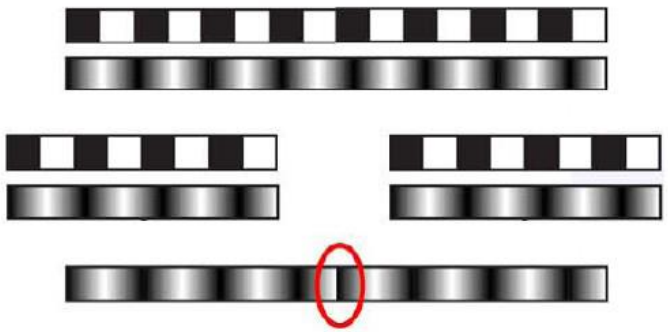
http://www.visualizationlibrary.com/

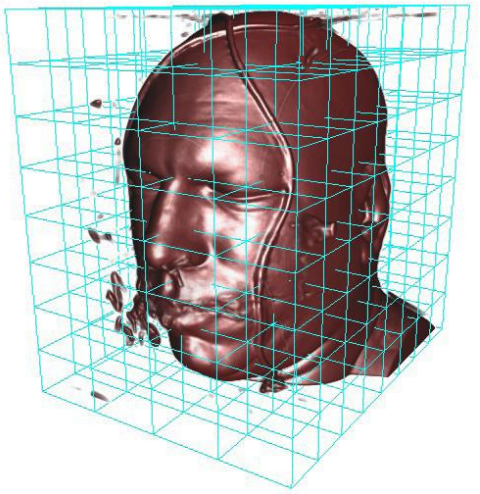
```
// FRAGMENT SHADER FOR VOLUME RENDERING USING 3D TEXTURES
varying vec3 frag_position; // in object space
uniform sampler3D volume_texunit;
uniform sampler3D gradient_texunit;
uniform sampler1D trfunc_texunit;
uniform float trfunc_delta;
uniform vec3 light_position[4]; // light positions in object space
uniform bool light_enable[4]; // light enable flags
uniform vec3 eye_position; // camera position in object space
uniform float val_threshold;
uniform vec3 gradient_delta; // for on-the-fly gradient computation
uniform bool precomputed_gradient;

// computes a simplified lighting equation
vec3 blinn_phong(vec3 N, vec3 V, vec3 L, int light)
{
        // material properties
        vec3 Ka = vec3(1.0, 1.0, 1.0);
        vec3 Kd = vec3(1.0, 1.0, 1.0);
        vec3 Ks = vec3(1.0, 1.0, 1.0);
        float shininess = 50.0; // diffuse coefficient
        float diff_coeff = max(dot(L,N),0.0);
        // specular coefficient
        vec3 H = normalize(L+V);
        float spec_coeff = pow(max(dot(H,N), 0.0), shininess);
        if (diff_coeff <= 0.0) spec_coeff = 0.0;
        // final lighting model
        return Ka * gl_LightSource[light].ambient.rgb +
               Kd * gl_LightSource[light].diffuse.rgb * diff_coeff +
               Ks * gl_LightSource[light].specular.rgb * spec_coeff;
}
```

```
void main(void) {
  // sample the LUMINANCE value
  float val = texture3D(volume_texunit, gl_TexCoord[0].xyz ).r;
  // all the pixels whose val is less than val_threshold are discarded
  if (val < val_threshold) discard;
  // sample the transfer function, post-classification
  float clamped_val = trfunc_delta+(1.0-2.0*trfunc_delta)*val;
  vec4 color = texture1D(trfunc_texunit, clamped_val);
  vec3 color_tmp;
  // compute the gradient and lighting only if the pixel is visible "enough"
  if (color.a > LIGHTING_ALPHA_THRESHOLD) {
    vec3 N;
    if (precomputed_gradient) {
      // retrieve pre-computed gradient
      N = normalize( (texture3D(gradient_texunit, gl_TexCoord[0].xyz).xyz - vec3(0.5,0.5,0.5))*2.0 );
    }
    else {
      // on-the-fly gradient computation: slower but requires less memory (no gradient texture required).
      vec3 sample1, sample2;
      sample1.x = texture3D(volume_texunit, gl_TexCoord[0].xyz-vec3(gradient_delta.x,0.0,0.0) ).r;
      sample2.x = texture3D(volume_texunit, gl_TexCoord[0].xyz+vec3(gradient_delta.x,0.0,0.0) ).r;
      sample1.y = texture3D(volume_texunit, gl_TexCoord[0].xyz-vec3(0.0,gradient_delta.y,0.0) ).r;
      sample2.y = texture3D(volume_texunit, gl_TexCoord[0].xyz+vec3(0.0,gradient_delta.y,0.0) ).r;
      sample1.z = texture3D(volume_texunit, gl_TexCoord[0].xyz-vec3(0.0,0.0,gradient_delta.z) ).r;
      sample2.z = texture3D(volume_texunit, gl_TexCoord[0].xyz+vec3(0.0,0.0,gradient_delta.z) ).r;
      N = normalize( sample1 - sample2 );
    }
    vec3 V = normalize(eye_position - frag_position);
    for(int i=0; i<4; ++i) {
      if (light_enable[i]) {
        vec3 L = normalize(light_position[i] - frag_position);
        color_tmp.rgb += color.rgb * blinn_phong(N,V,L,i);
        }
    }
  }
  else
    color_tmp = color.rgb;
  gl_FragColor = vec4(color_tmp,color.a);
}
```

Bricking

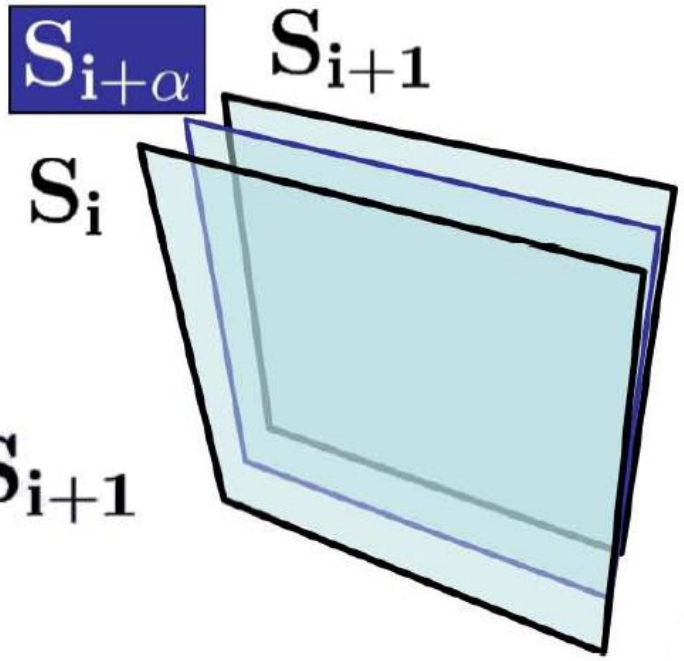
- 3D texture harder to fit into memory - data set is split into smaller chunks -bricks
- Problem:
  - interpolation on boundaries – voxel(s) must be duplicated
  - bus-bandwidth – several bricks must fit into memory





Multi-textures

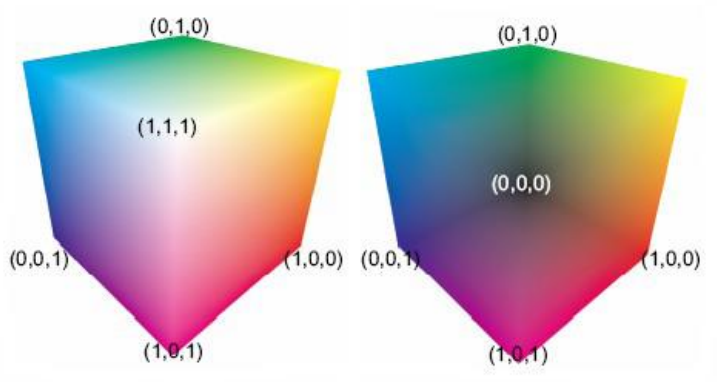
- Multi-textures (2 textures per polygon) to implement trilinear interpolation
- Blending of two adjacent slices
- Constant sampling rate
- GL_ARB_multitexture



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

Ray-casting

- Sent one ray per screen pixel
- Ray determination
  - Render front-faces of cube
  - Render back-faces of cube
  - Substract to get direction
  - Front-faces determine starting position
  - Back-faces determine exit position
- For each ray, traverse volume along ray by some small steps and compute VRI



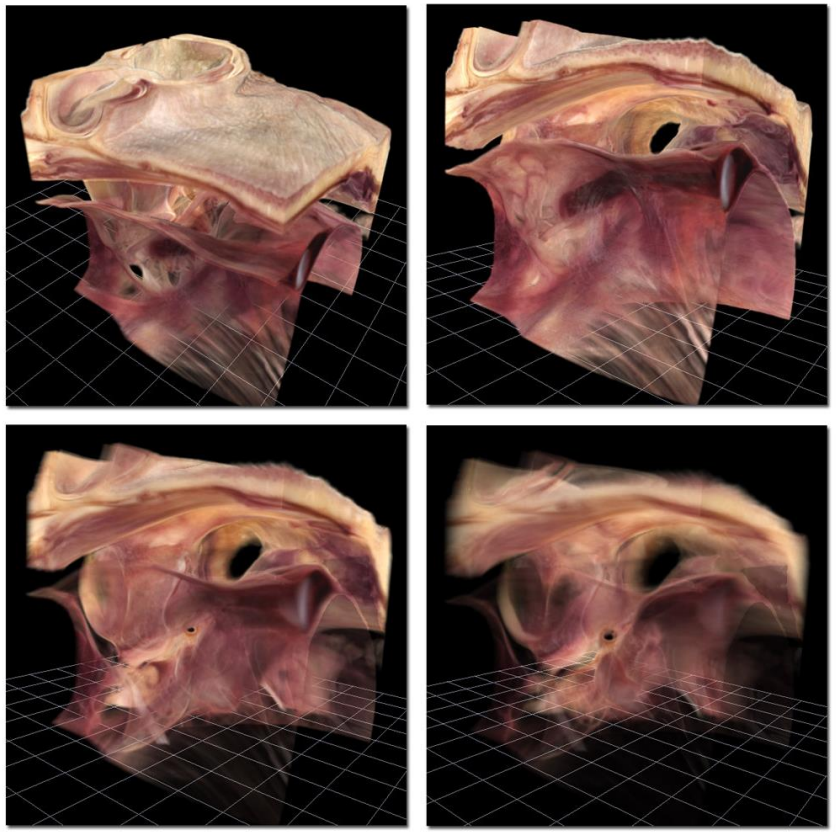
Ray-casting Cg

```
// Define interface between the application and the vertex program
struct app_vertex
 {
   float4 Position       : POSITION;
   float4 TexCoord       : TEXCOORD1;
   float4 Color          : COLOR0;

};

// Define the interface between the vertex- and the fragment programs
struct vertex_fragment
 {
   float4 Position       : POSITION; // For the rasterizer
   float4 TexCoord       : TEXCOORD0;
   float4 Color          : TEXCOORD1;
   float4 Pos            : TEXCOORD2;
};

struct fragment_out
{
  float4 Color           : COLOR0;
};

// Raycasting vertex program implementation
vertex_fragment vertex_main( app_vertex IN )
{
  vertex_fragment OUT;
  // Get OpenGL state matrices
  float4x4 ModelView = glstate.matrix.modelview[0];
  float4x4 ModelViewProj = glstate.matrix.mvp;
  // Transform vertex
  OUT.Position = mul( ModelViewProj, IN.Position );
  OUT.Pos =  mul( ModelViewProj, IN.Position );
  OUT.TexCoord = IN.TexCoord;
  OUT.Color = IN.Color;
  return OUT;
}
```

```
// Raycasting fragment program implementation
fragment_out fragment_main( vertex_fragment IN,
                            uniform sampler2D tex,
                            uniform sampler3D volume_tex,
                            uniform float stepsize )
{
  fragment_out OUT;
  float2 texc = ((IN.Pos.xy / IN.Pos.w) + 1) / 2; // find the right place to lookup in the backside buffer
  float4 start = IN.TexCoord; // the start position of the ray is stored in the texturecoordinate
  float4 back_position  = tex2D(tex, texc);
  float3 dir = float3(0,0,0);
  dir.x = back_position.x - start.x;
  dir.y = back_position.y - start.y;
  dir.z = back_position.z - start.z;
  float len = length(dir.xyz); // the length from front to back is calculated and used to terminate the ray
  float3 norm_dir = normalize(dir);
  float delta = stepsize;
  float3 delta_dir = norm_dir * delta;
  float delta_dir_len = length(delta_dir);
  float3 vec = start;
  float4 col_acc = float4(0,0,0,0);
  float alpha_acc = 0;
  float length_acc = 0;
  float4 color_sample;
  float alpha_sample;

  for(int i = 0; i < 450; i++)
  {
     color_sample = tex3D(volume_tex,vec);
     alpha_sample = color_sample.a * stepsize;
     col_acc   += (1.0 - alpha_acc) * color_sample * alpha_sample * 3;
     alpha_acc += alpha_sample;
     vec += delta_dir;
     length_acc += delta_dir_len;
     if(length_acc >= len || alpha_acc > 1.0) break; // terminate if opacity > 1 or the ray is outside the volume
  }
  OUT.Color =  col_acc;
  return OUT;
}
```

Ray-casting

http://www.daimi.au.dk/~trier/?page_id=98



Local Illumination

- Traditional local illumination models
- Normal = gradient

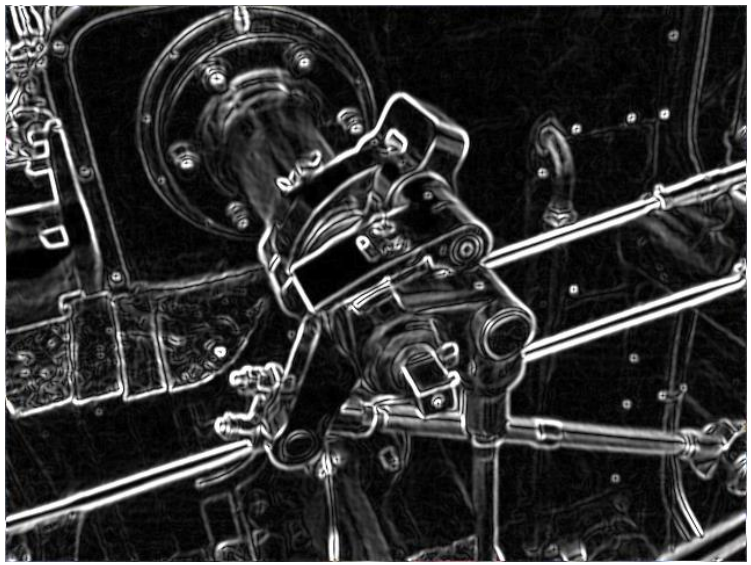
$$\mathbf{n}(\mathbf{x}) = \frac{\nabla f(\mathbf{x})}{\|\nabla f(\mathbf{x})\|}, \qquad \text{if } \|\nabla f(\mathbf{x})\| \neq 0 \qquad \nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f(\mathbf{x})}{\partial x} \\ \frac{\partial f(\mathbf{x})}{\partial y} \\ \frac{\partial f(\mathbf{x})}{\partial z} \end{pmatrix}$$

- Gradient estimation
  - On the fly, pre-processing
  - Central differences
  - Sobel operator
  - …

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

Sobel operator

$$\nabla \mathbf{f} = \begin{bmatrix} G_x & G_y & G_z \end{bmatrix}^T = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} \end{bmatrix}^T$$



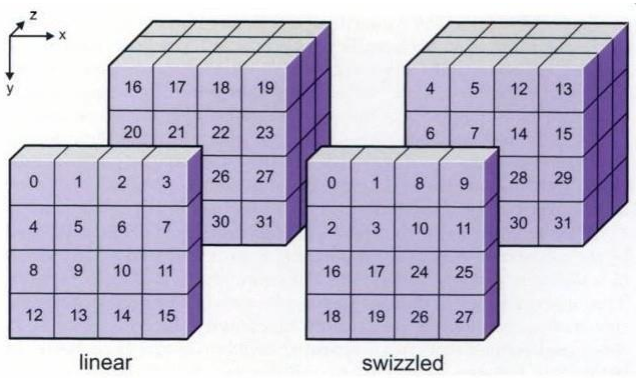
$$G_x : \begin{bmatrix} -2 & -4 & -2 \\ 0 & 0 & 0 \\ 2 & 4 & 2 \end{bmatrix}, \begin{bmatrix} -4 & -8 & -4 \\ 0 & 0 & 0 \\ 4 & 8 & 4 \end{bmatrix}, \begin{bmatrix} -2 & -4 & -2 \\ 0 & 0 & 0 \\ 2 & 4 & 2 \end{bmatrix}$$

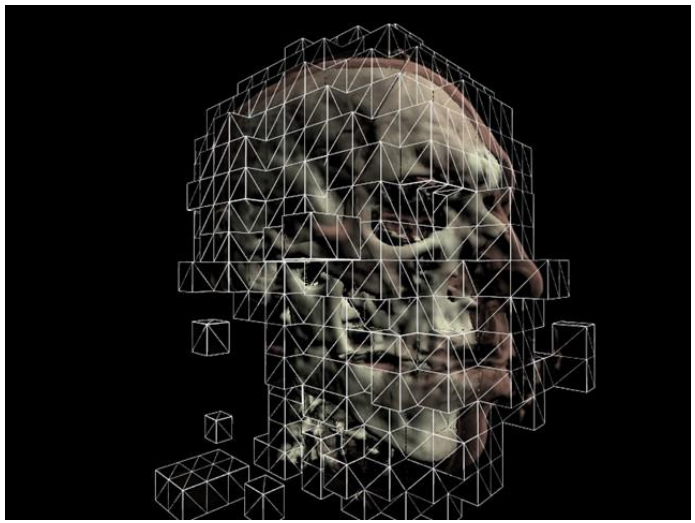
$$G_y : \begin{bmatrix} -2 & 0 & 2 \\ -4 & 0 & 4 \\ -2 & 0 & 2 \end{bmatrix}, \begin{bmatrix} -4 & 0 & 4 \\ -8 & 0 & 8 \\ -4 & 0 & 4 \end{bmatrix}, \begin{bmatrix} -2 & 0 & 2 \\ -4 & 0 & 4 \\ -2 & 0 & 2 \end{bmatrix}$$

$$G_z : \begin{bmatrix} -2 & -4 & -2 \\ -4 & -8 & -4 \\ -2 & -4 & -2 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 2 & 4 & 2 \\ 4 & 8 & 4 \\ 2 & 4 & 2 \end{bmatrix}$$

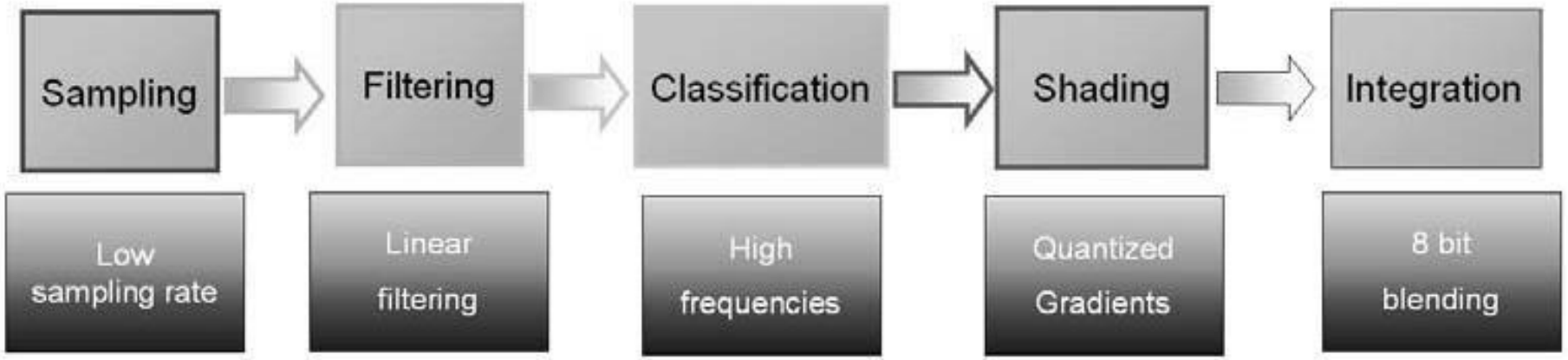
Speed-up Techniques

- Memory management (swizzling, mipmapping)
- Asynchronous data upload
- Bi / Tri-linear filtering
- Empty-space leaping
- Occlusion culling
- Early ray termination
- Image downscaling



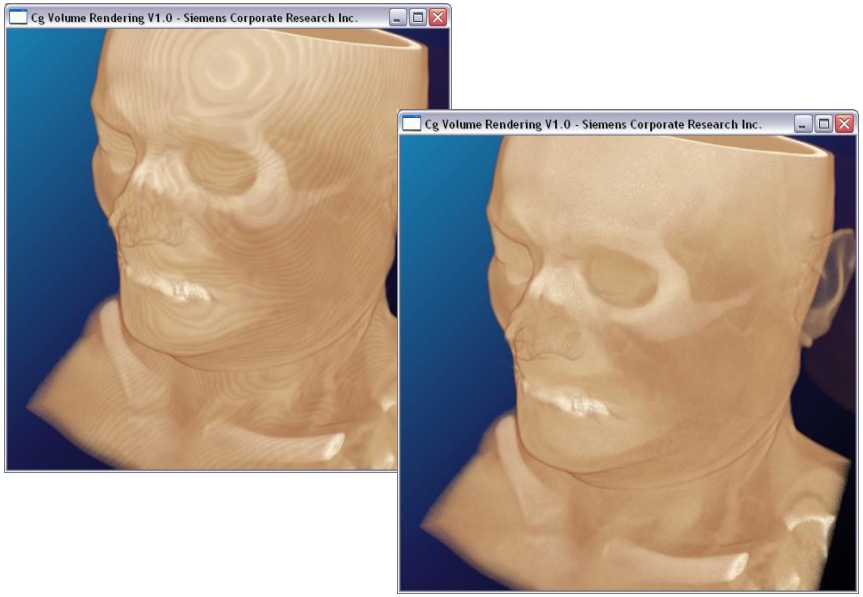


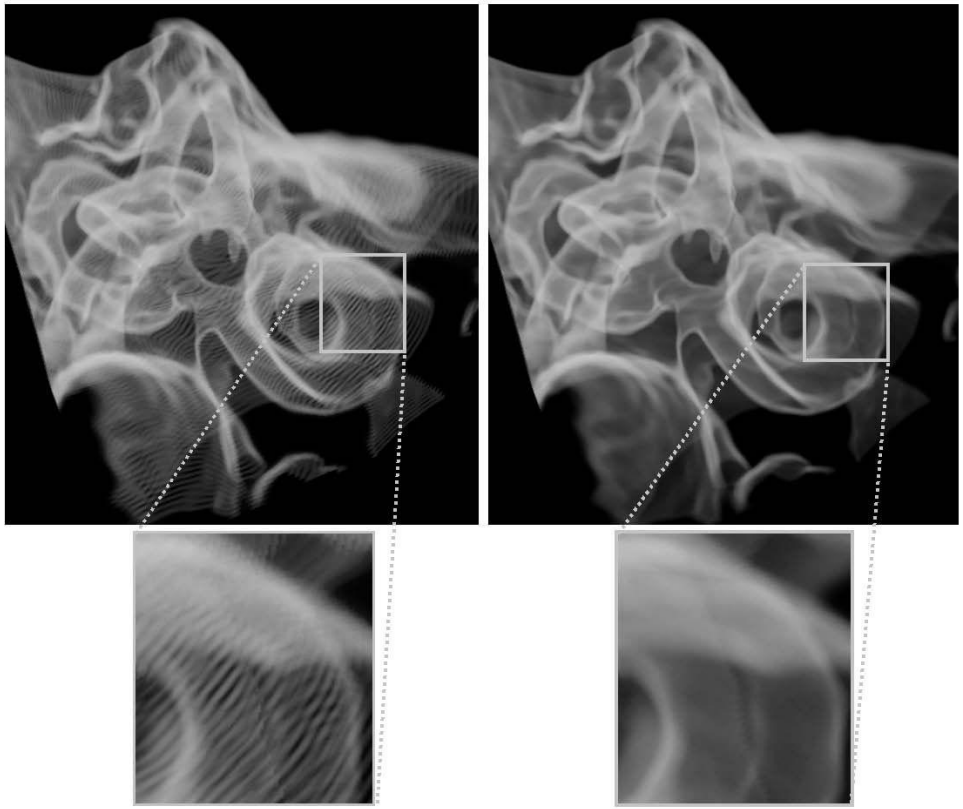
Improving Quality



Sampling Artifacts

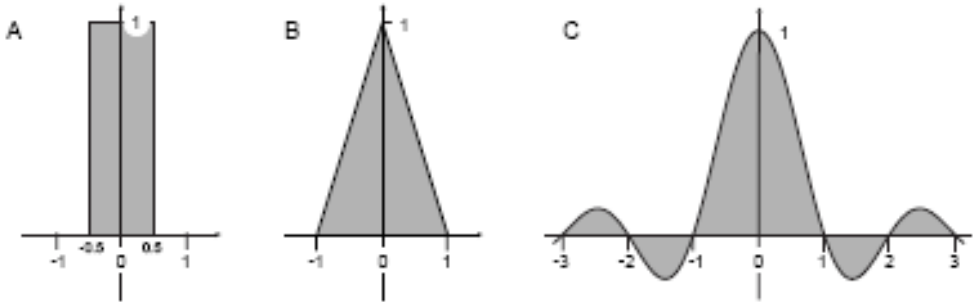
- Adaptive sampling
- Opacity correction
- Stochastic jittering

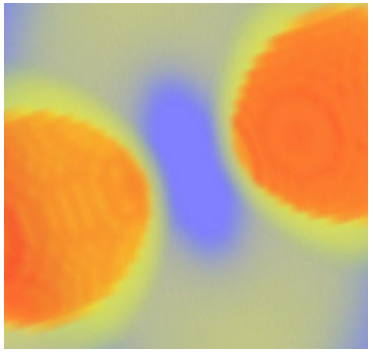


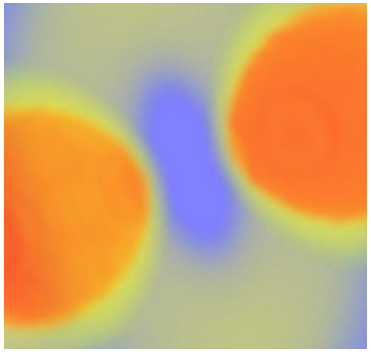


Filtering Artifacts

- Discrete data → continuous signal
  - Internal filtering precision
  - Linear filter
- HQ filtering
  - Convolution (bspline, gauss, sinc, …)
  - Bricking (more overlap samples)

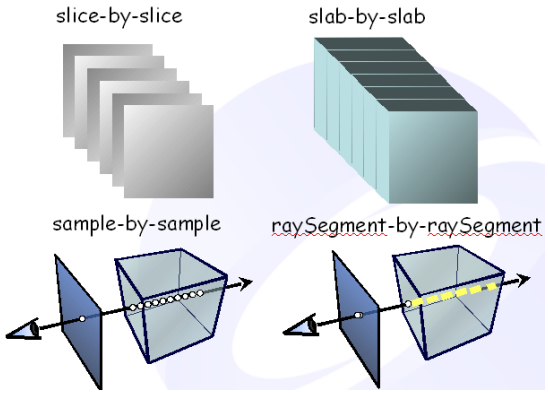


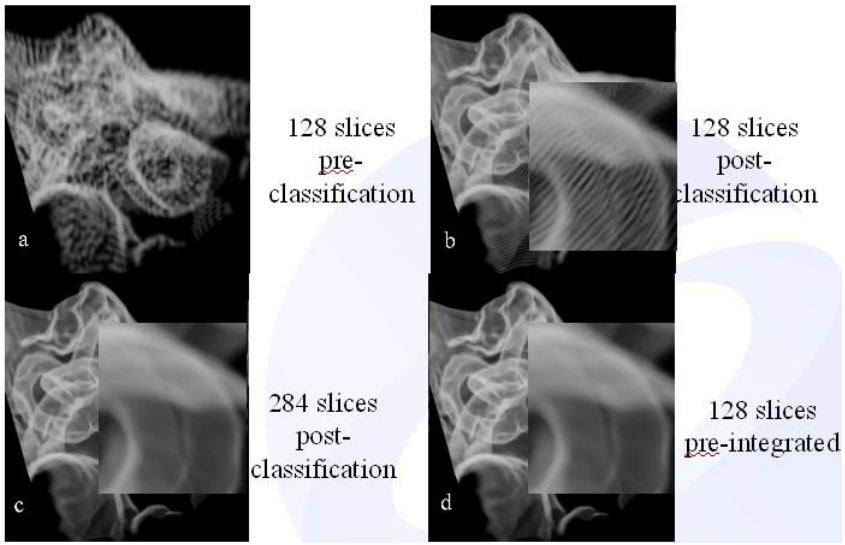




Classification Artifacts

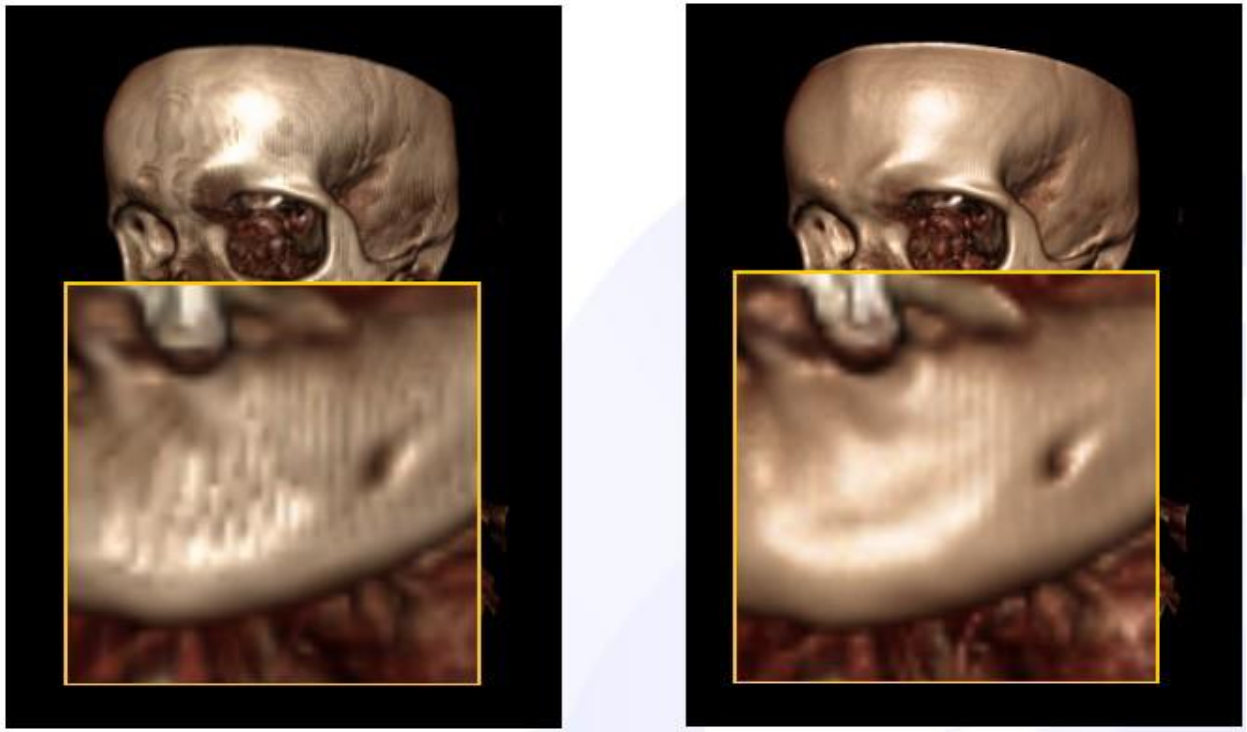
- High frequency in the Transfer Function → increase sampling rate
- Pre-integrated classification
  - Pre-integration for TF
  - Integration for the scalar field
  - Still not „correct”





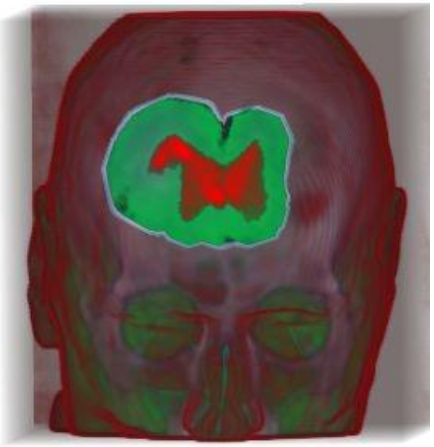
Shading Artifacts

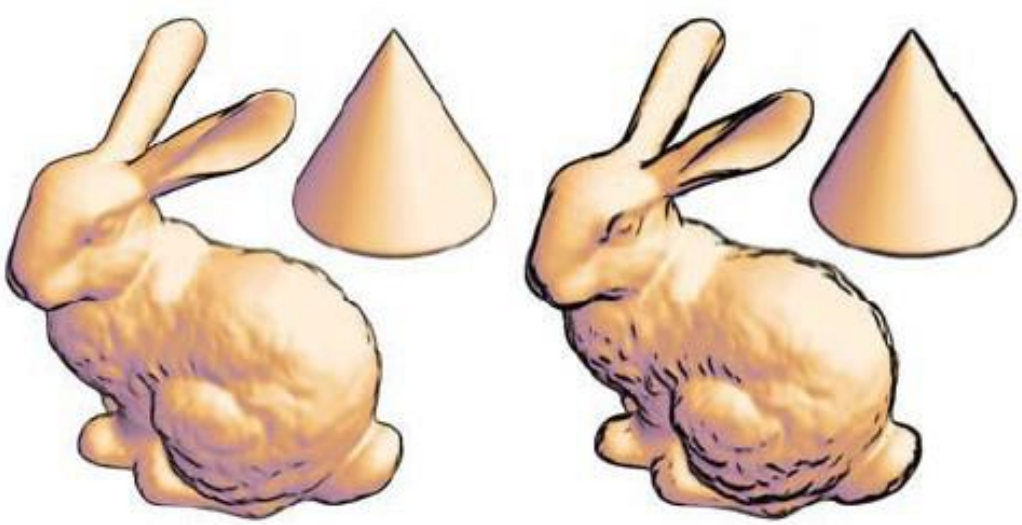
- Limited gradient precision
- Pre-computation (16/32bit, Sobel operator)
- On the fly



Other Techniques

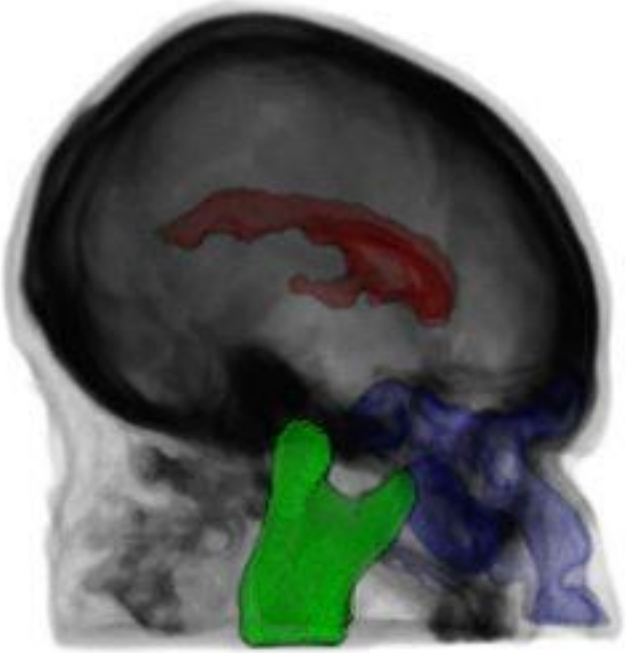
- Focus & context, peeling
- Non photorealistic rendering

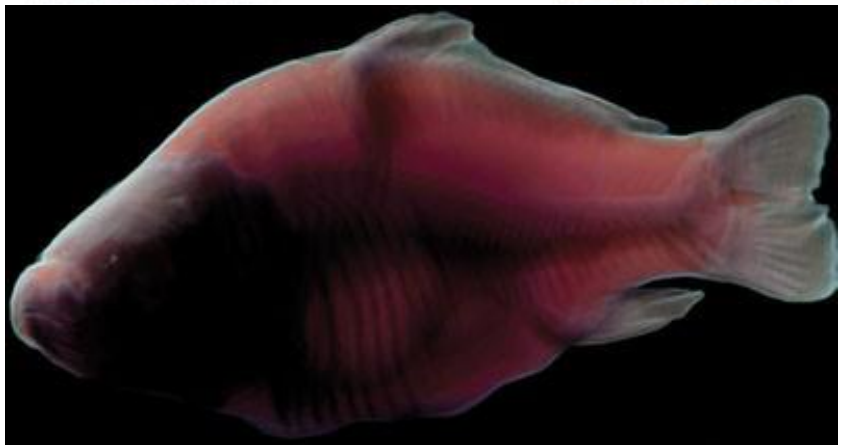




Other Techniques

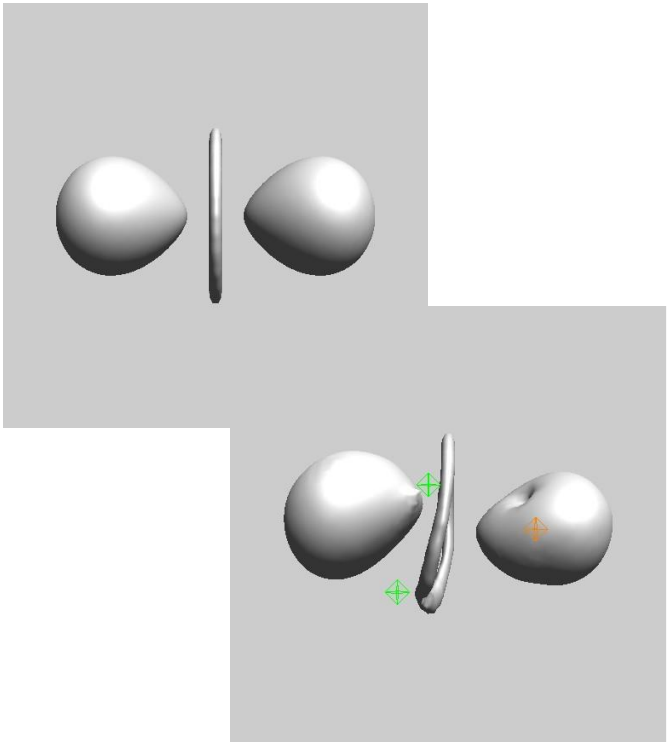
- Segmentation
- Global lightning





Other Techniques

- Effects
- Deformations



(a)

(b)

Questions?