Real-Time Volume Graphics

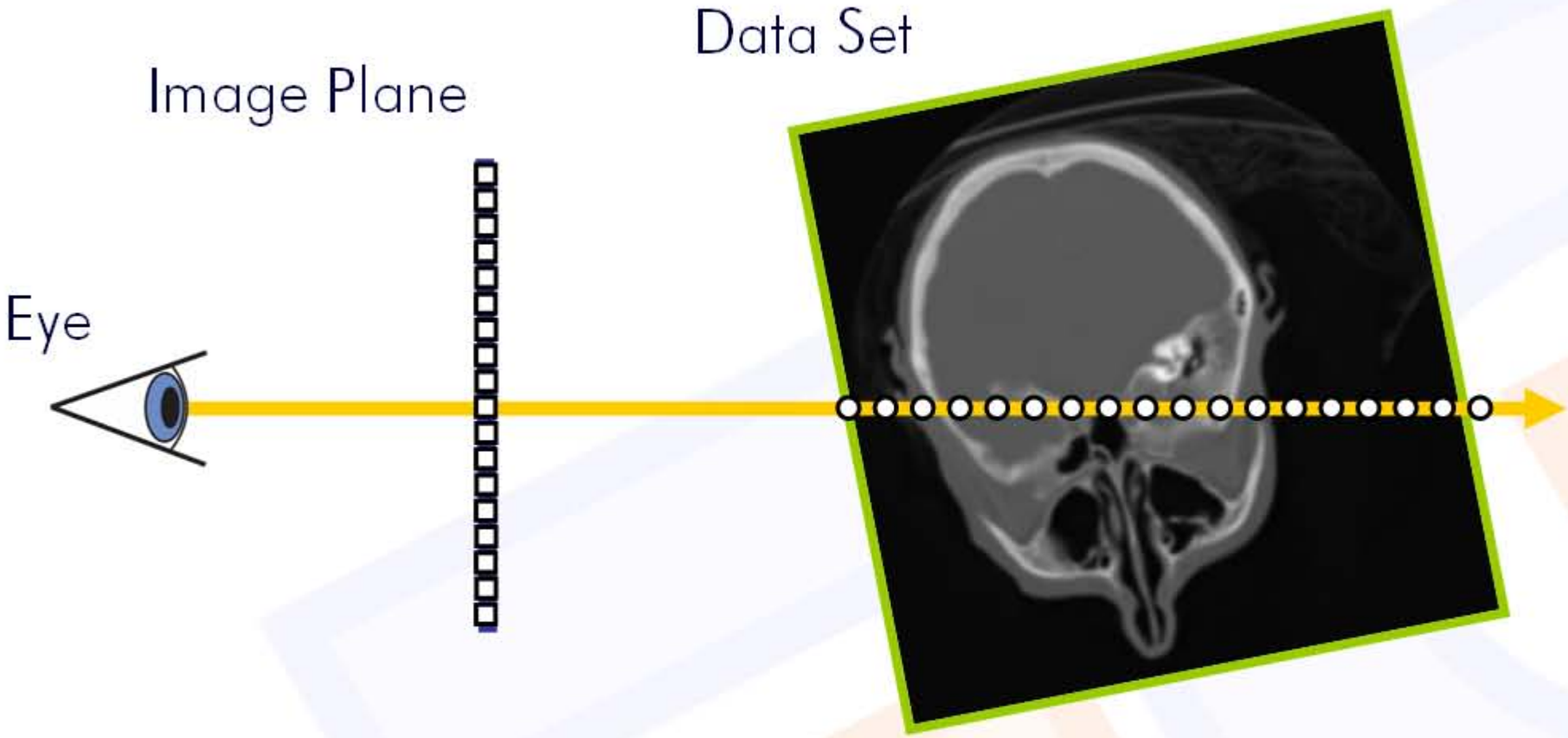
[03] GPU-Based Volume Rendering

Volume Rendering

Image order approach:

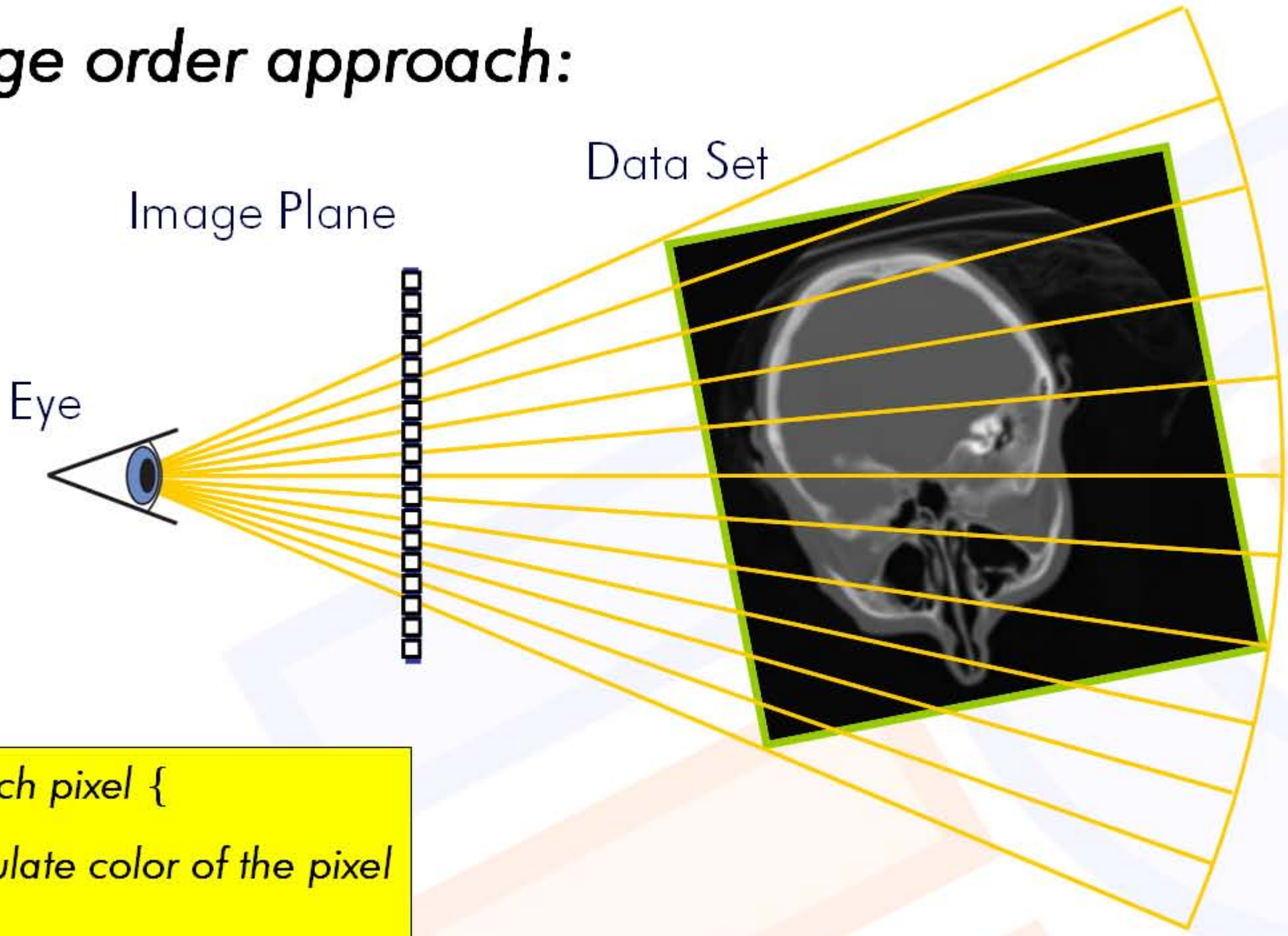
Volume Rendering

Image order approach:



Volume Rendering

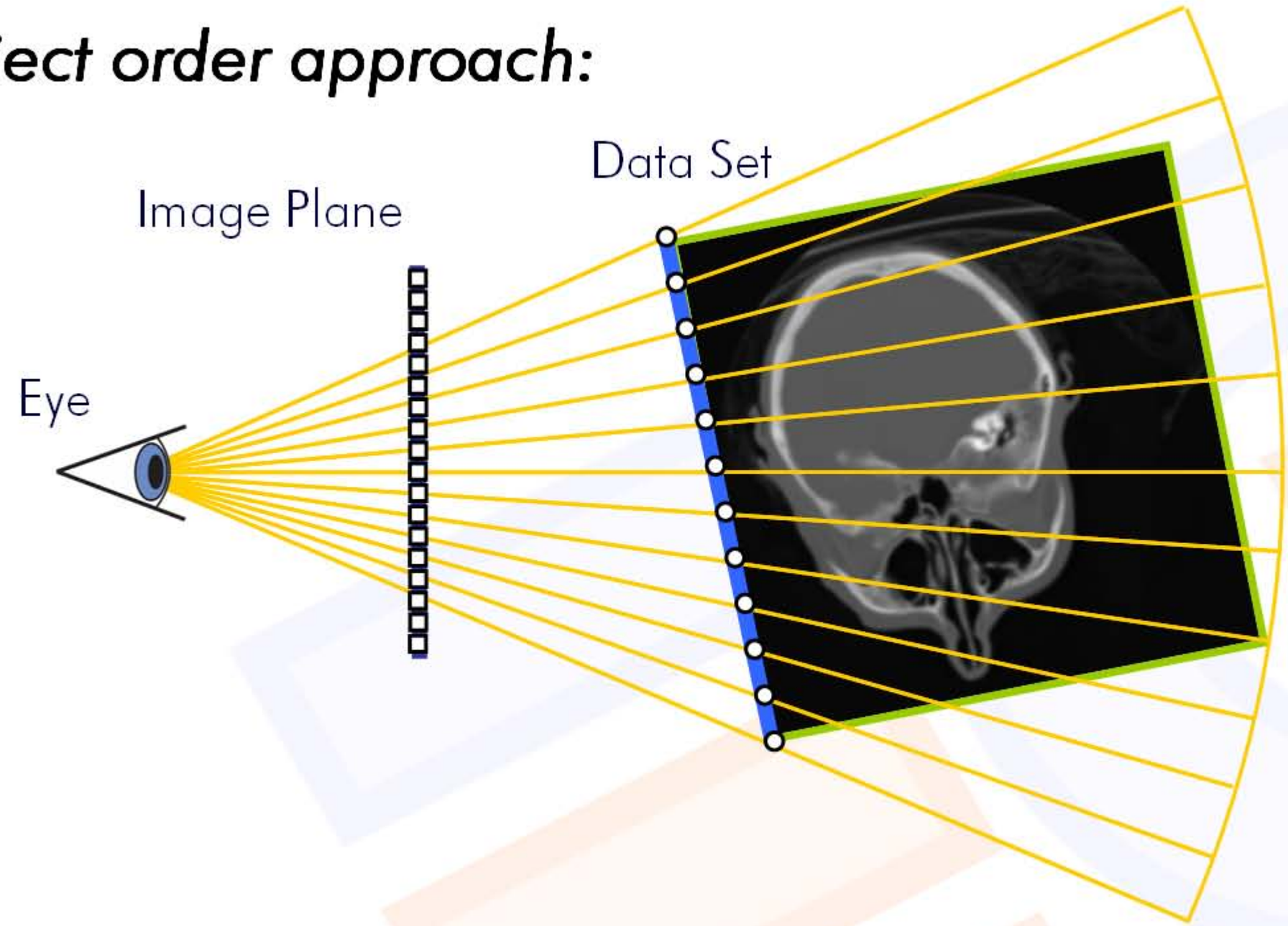
Image order approach:




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

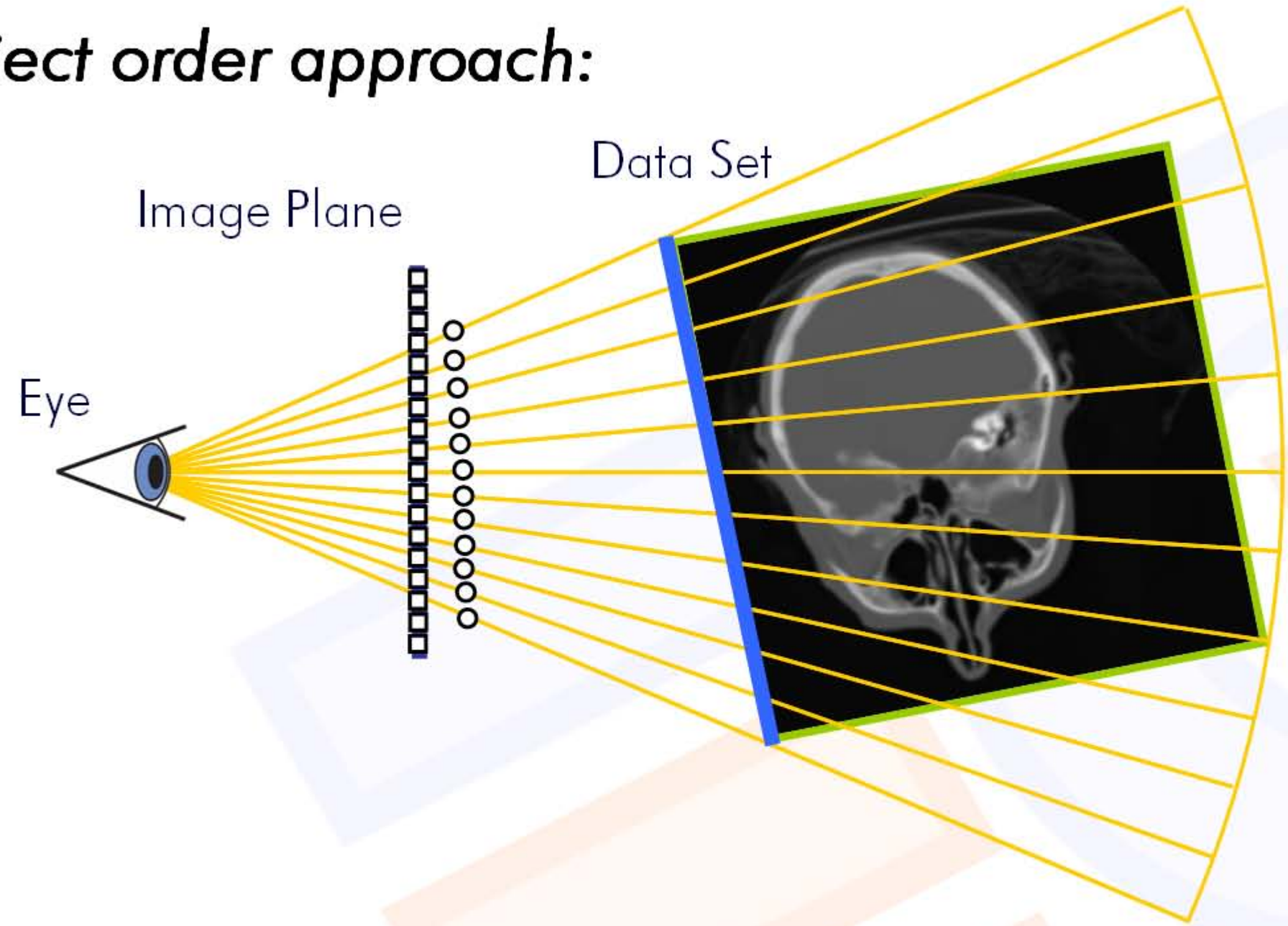
Volume Rendering

Object order approach:



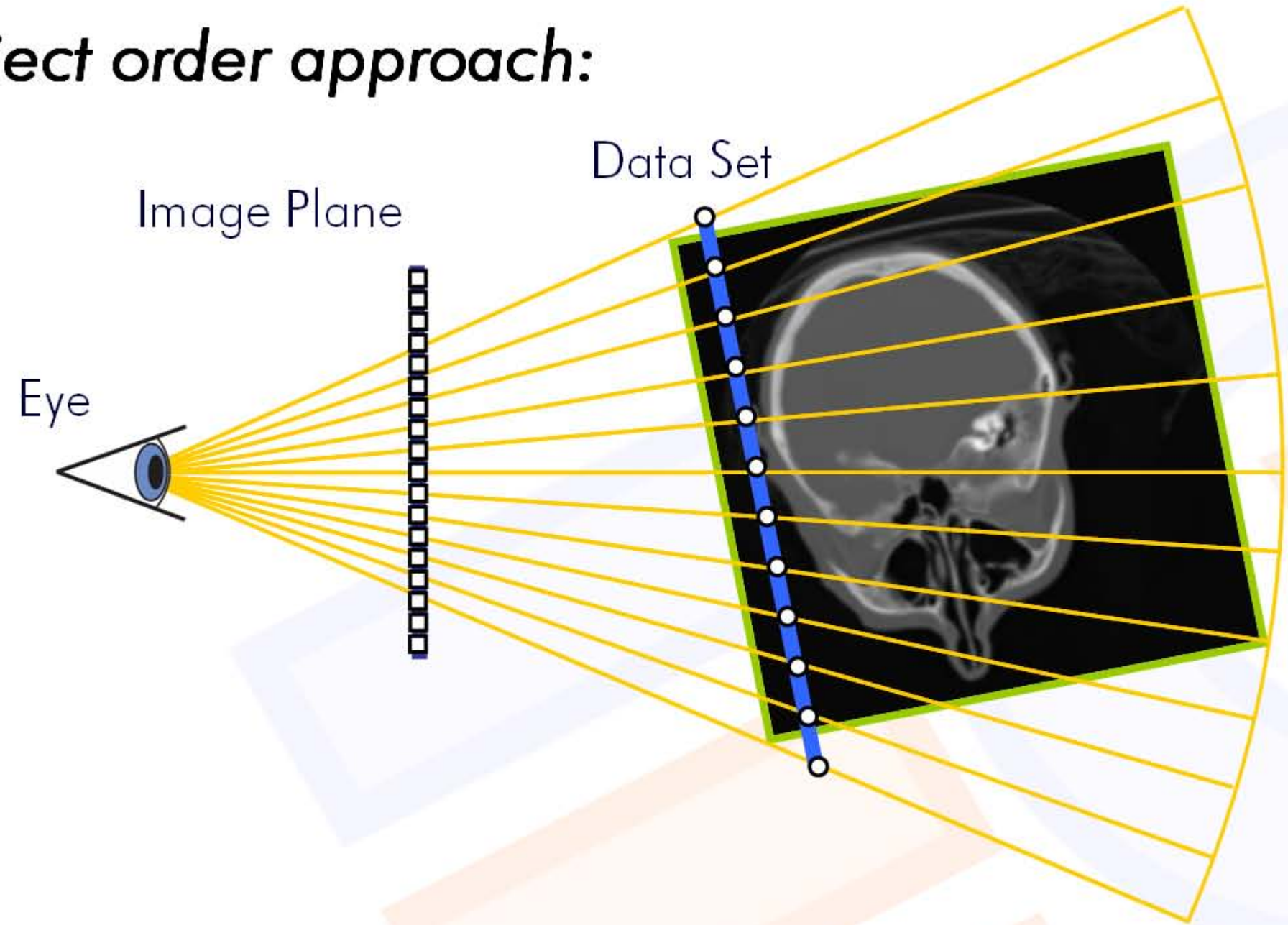
Volume Rendering

Object order approach:



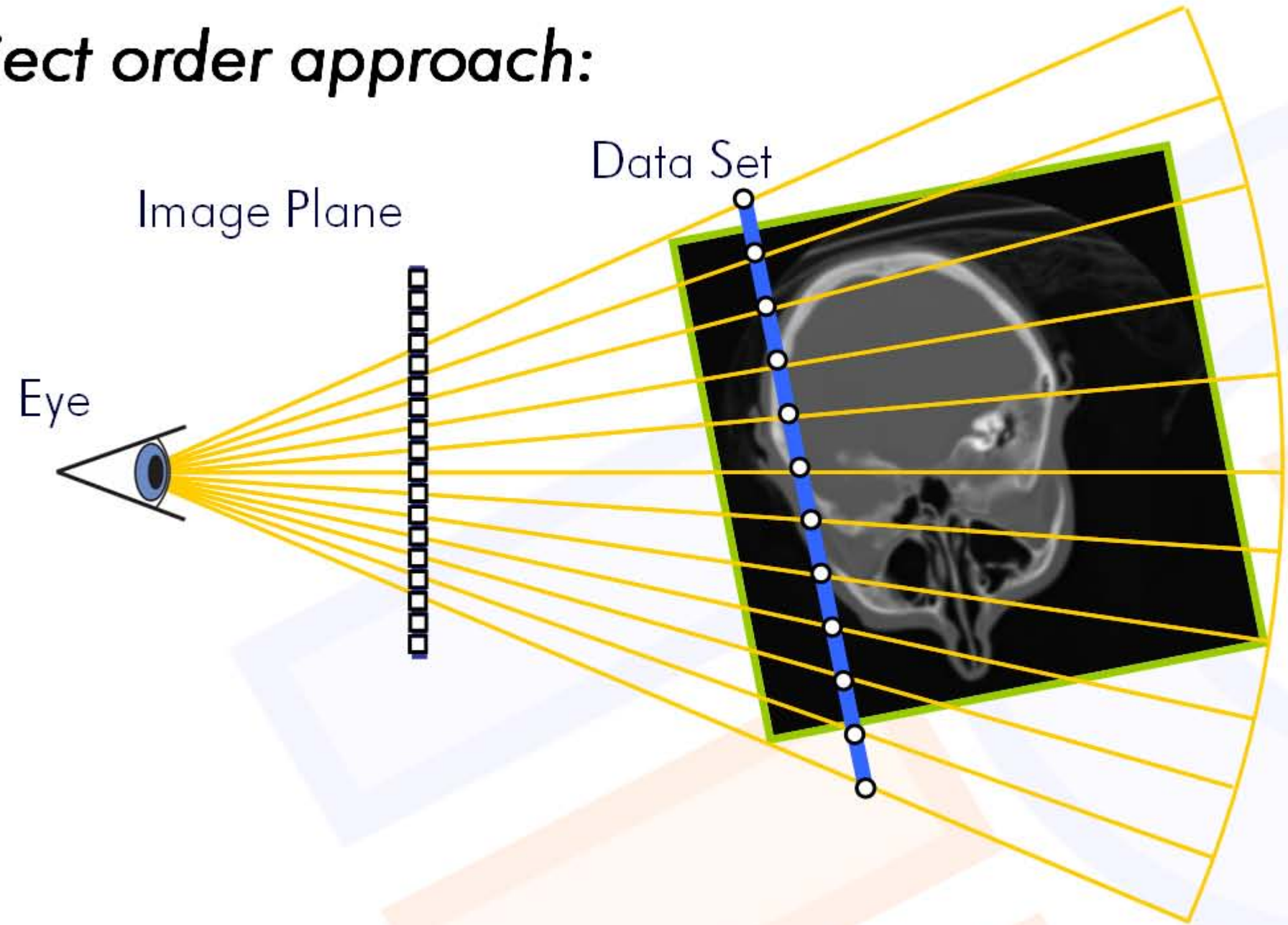
Volume Rendering

Object order approach:



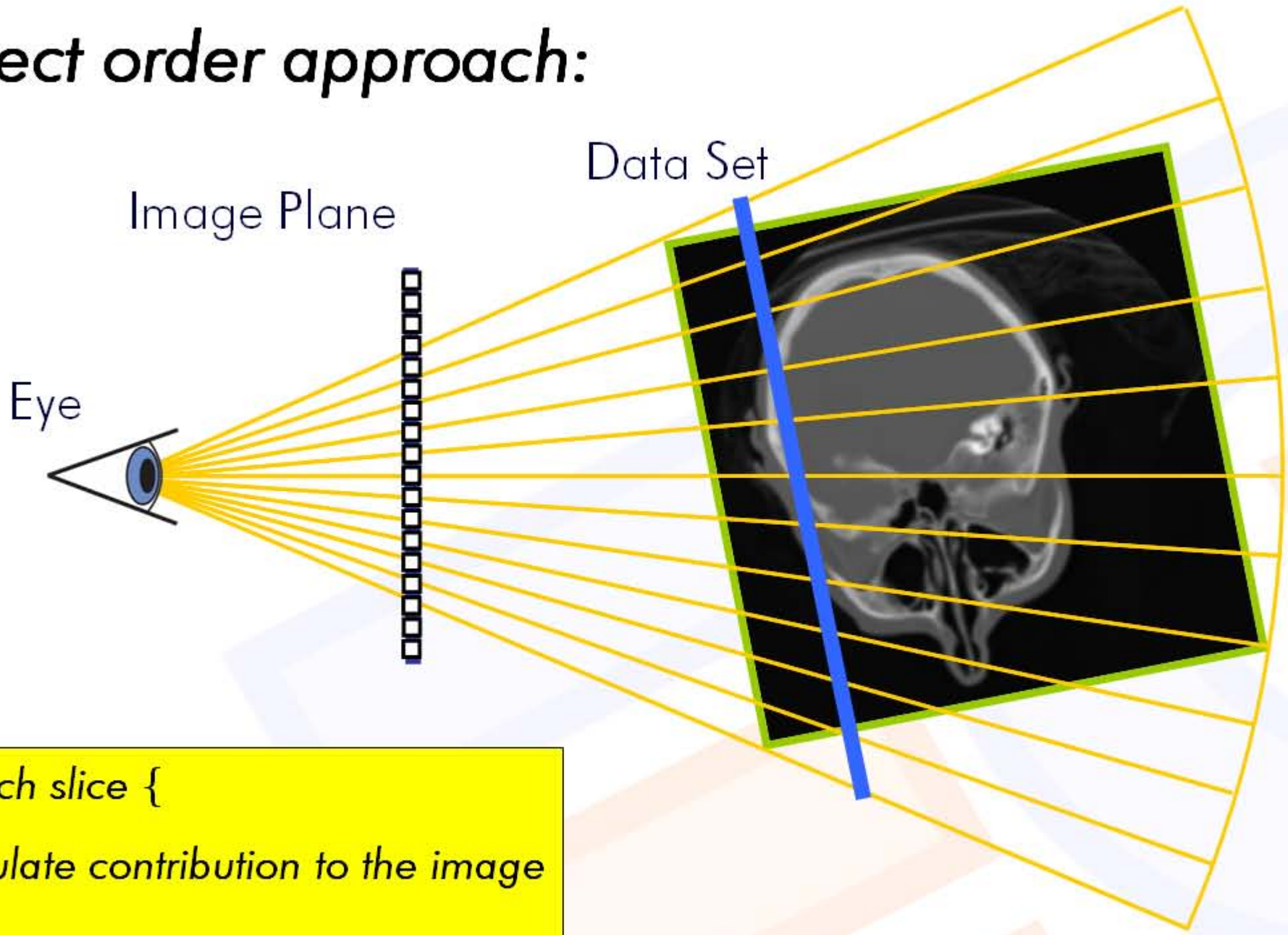
Volume Rendering

Object order approach:



Volume Rendering

Object order approach:

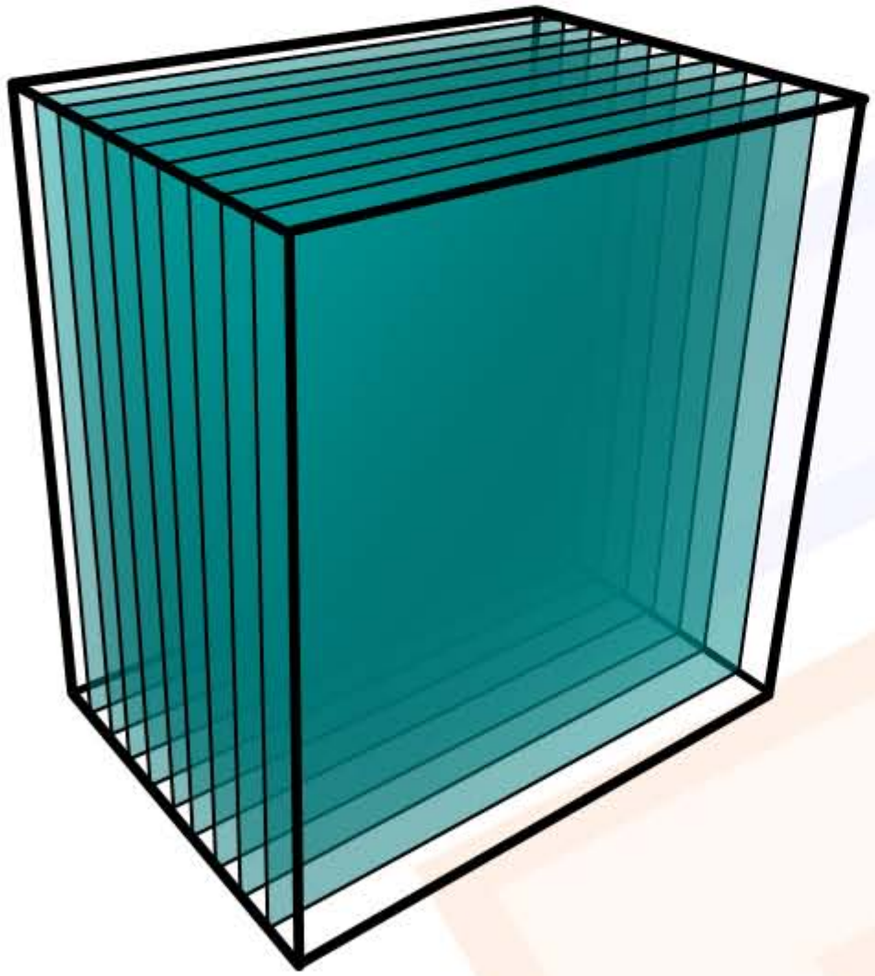


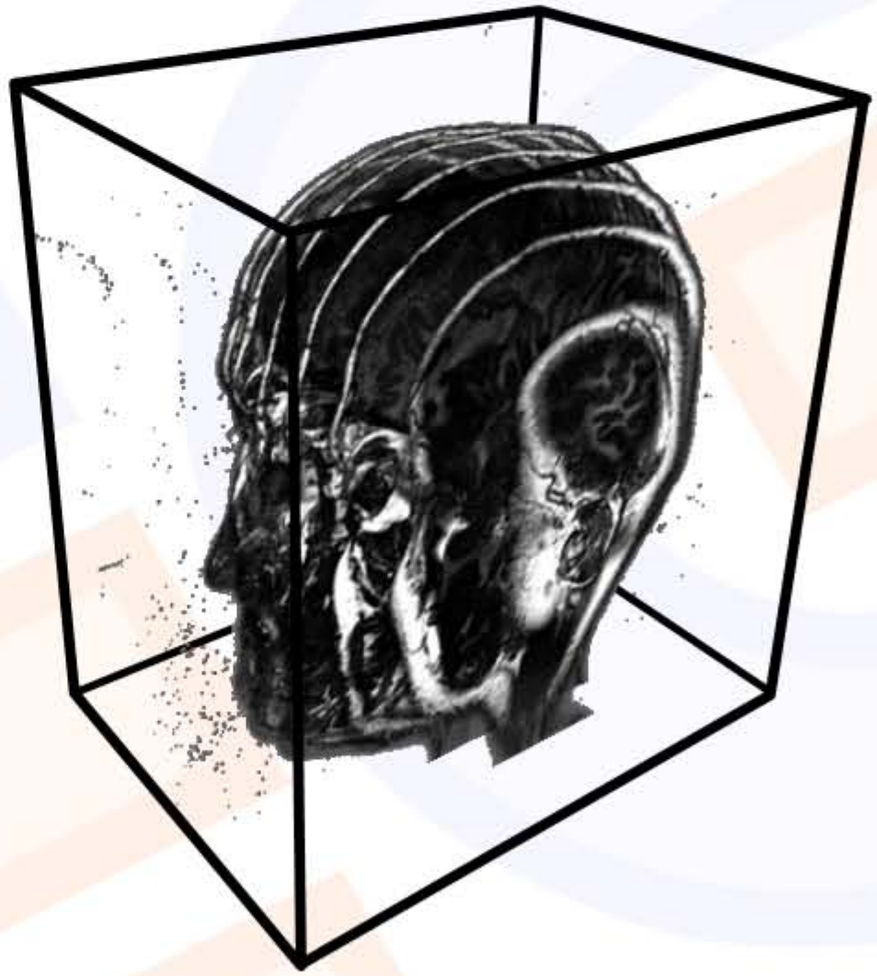

```
For each slice {
    calculate contribution to the image
}
```

Texture-based Approaches

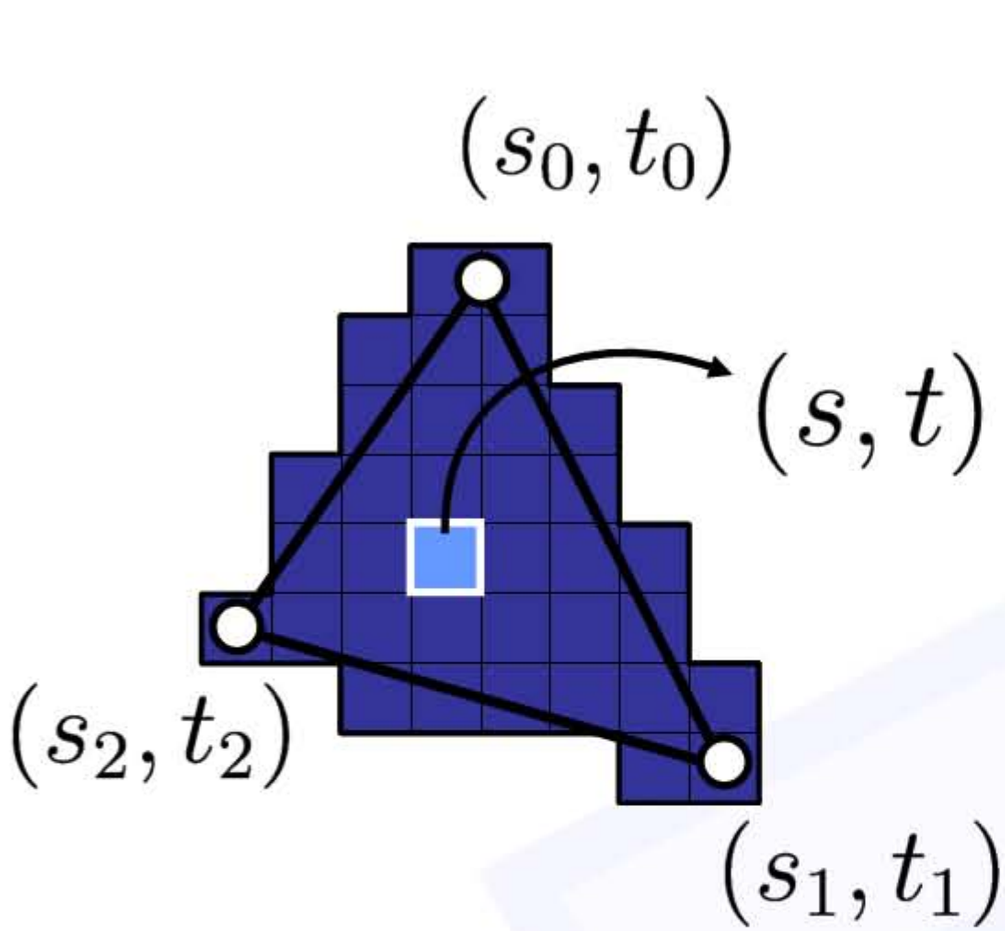
- No volumetric hardware-primitives!

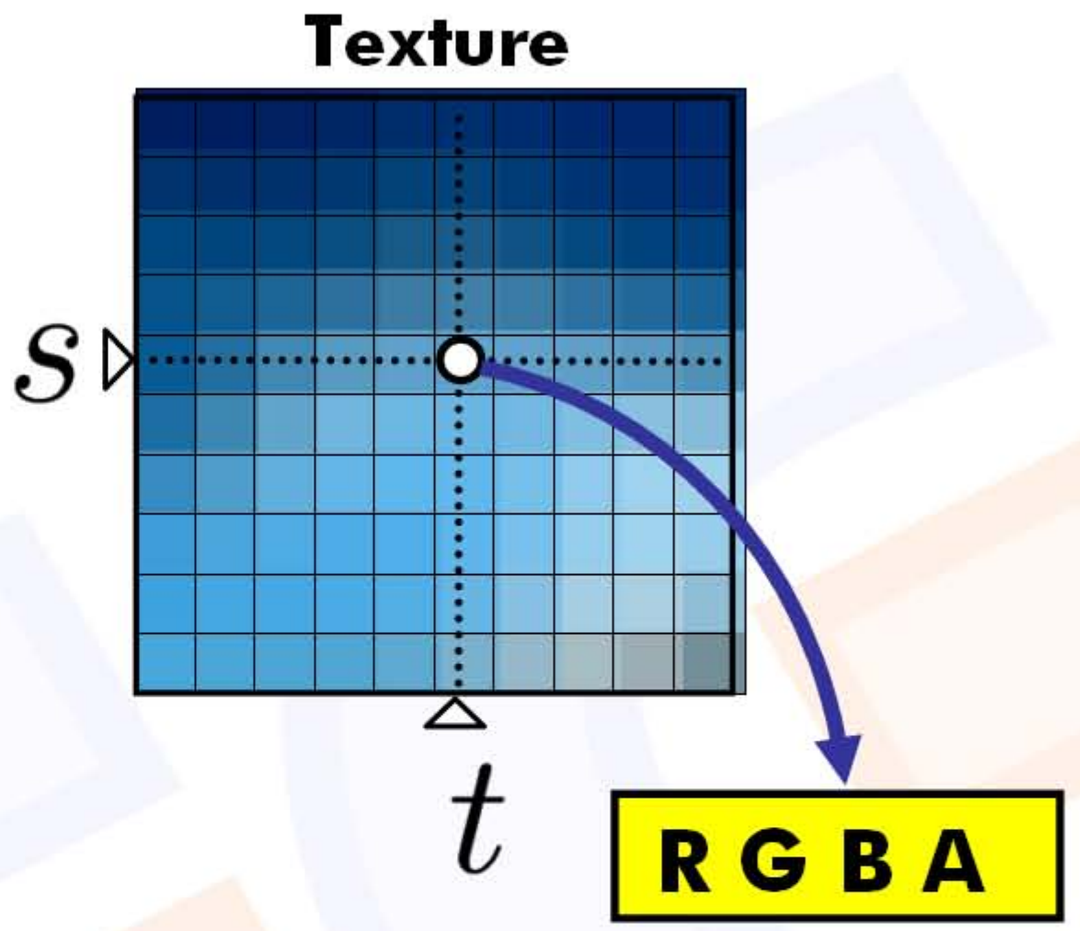
➡ Proxy geometry (Polygonal Slices)





How does a texture work?





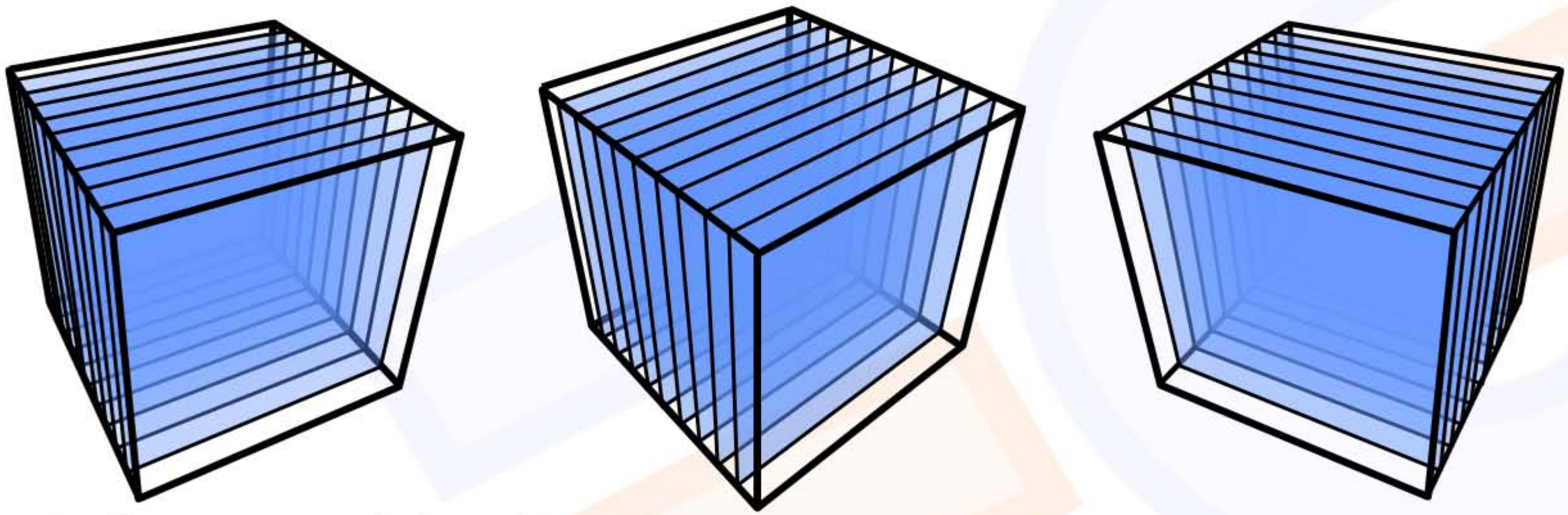
For each fragment:
interpolate the
texture coordinates
(barycentric)

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);
```

Implementation

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

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;
}
```

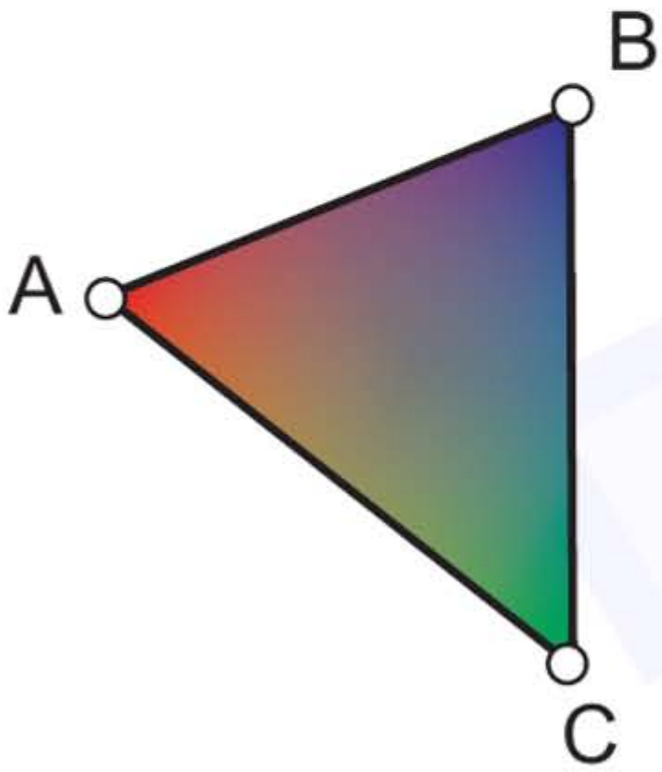
We assume here that the RGBA texture already contains emission/absoption coefficients.

Transfer functions are discussed later

Compositing

```
//standard alpha blending setup
glEnable(GL_BLEND);
glAlphaFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

The standard alpha blending causes color bleeding!



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

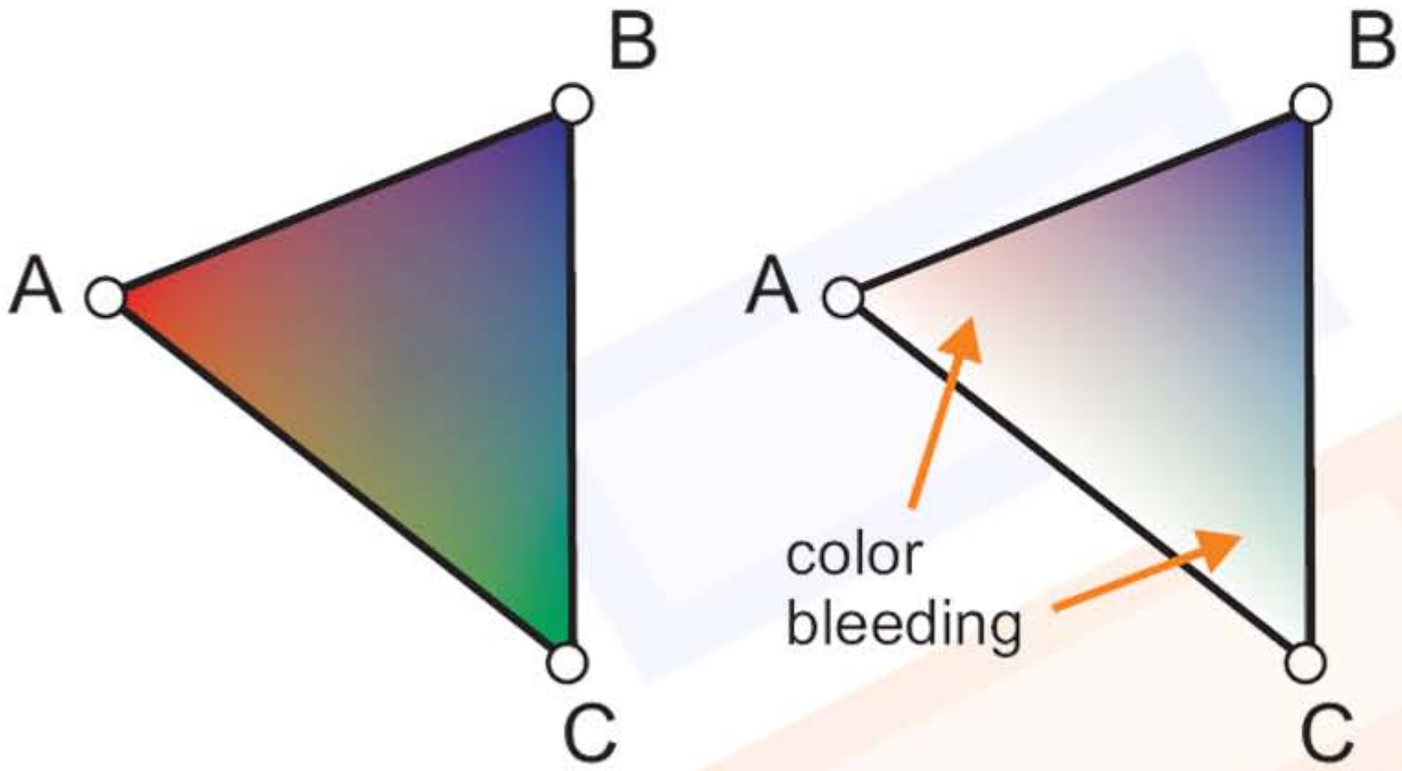
Vertex B: RGBA = (0,0,1,1)

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

Compositing

```
//standard alpha blending setup
glEnable(GL_BLEND);
glAlphaFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

The standard alpha blending causes color bleeding!



Vertex A: RGBA = (1,0,0,**0**)

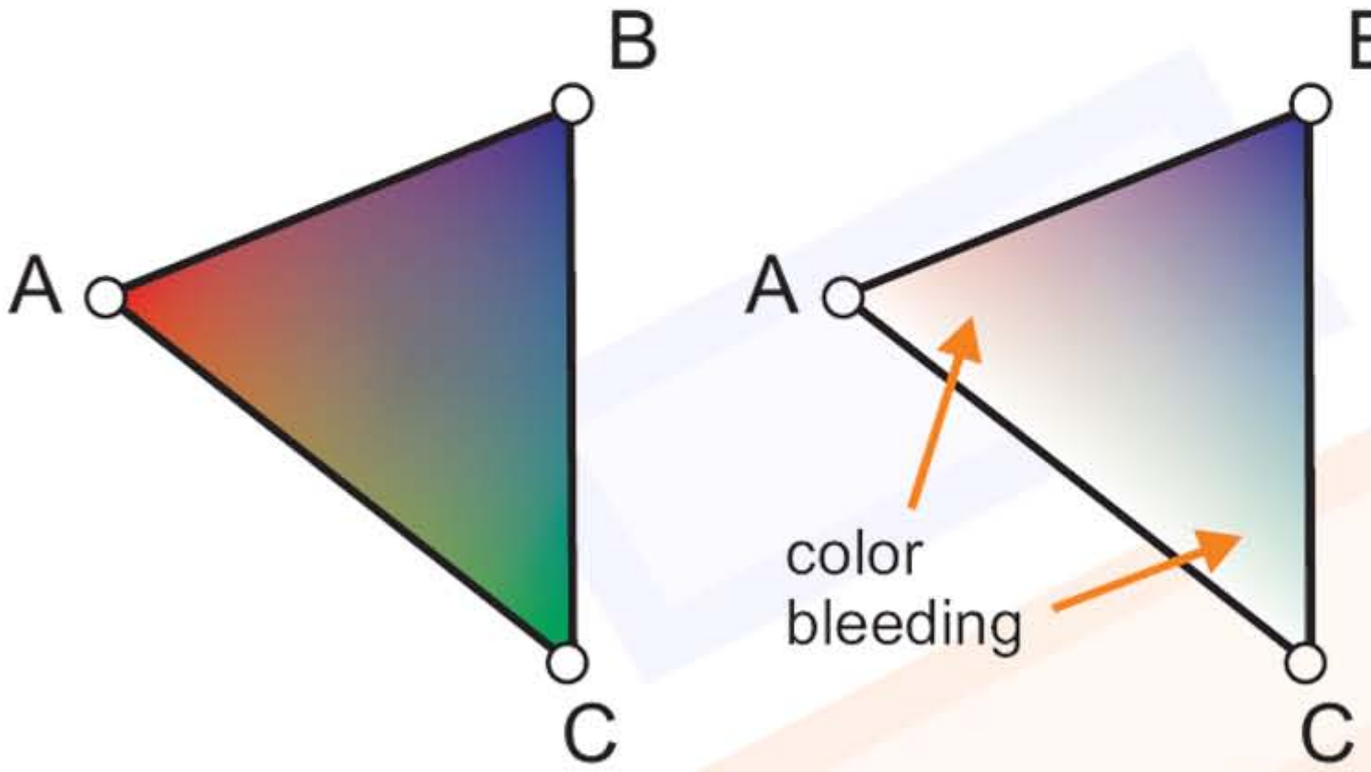
Vertex B: RGBA = (0,0,1,1)

Vertex C: RGBA = (0,1,0,**0**)

Compositing

```
//standard alpha blending setup
glEnable(GL_BLEND);
glAlphaFunc(GL_SRC_ALPHA, 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)

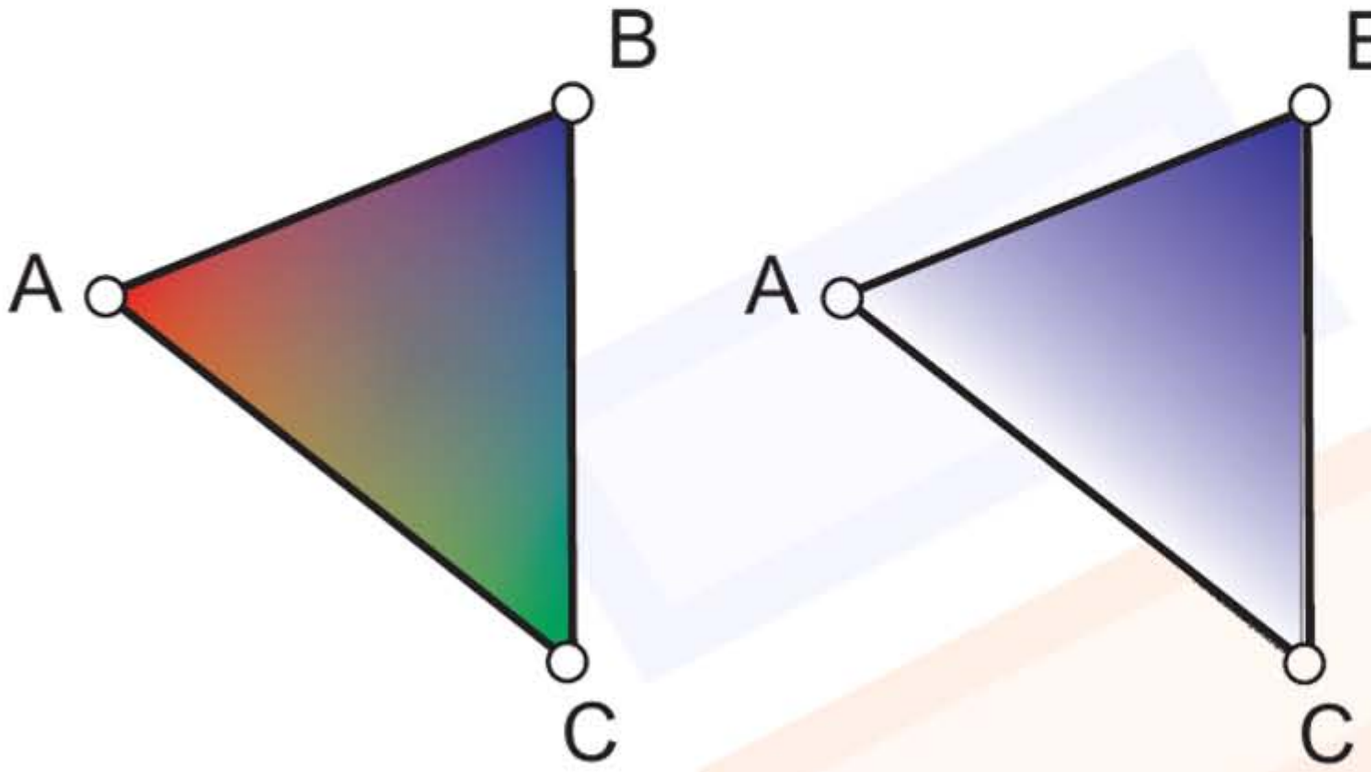
Solution: Associated Colors:

RGB values must be pre-multiplied by opacity A!

Compositing

```
//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 = (0,0,0,**0**)

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

Vertex C: RGBA = (0,0,0,**0**)

Solution: Associated Colors:

RGB values must be pre-multiplied by opacity A!

Compositing

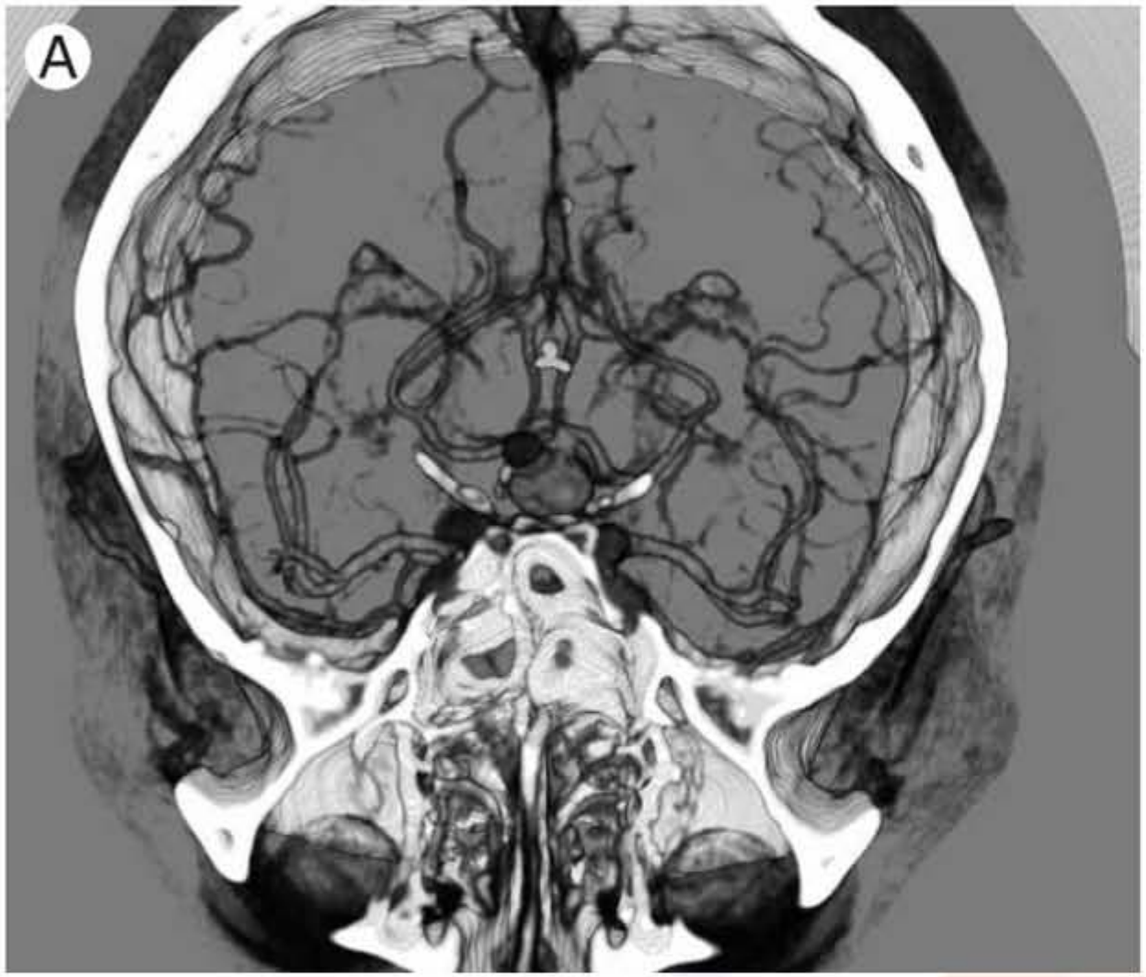
- *Maximum Intensity Projection*

  No emission/absorption

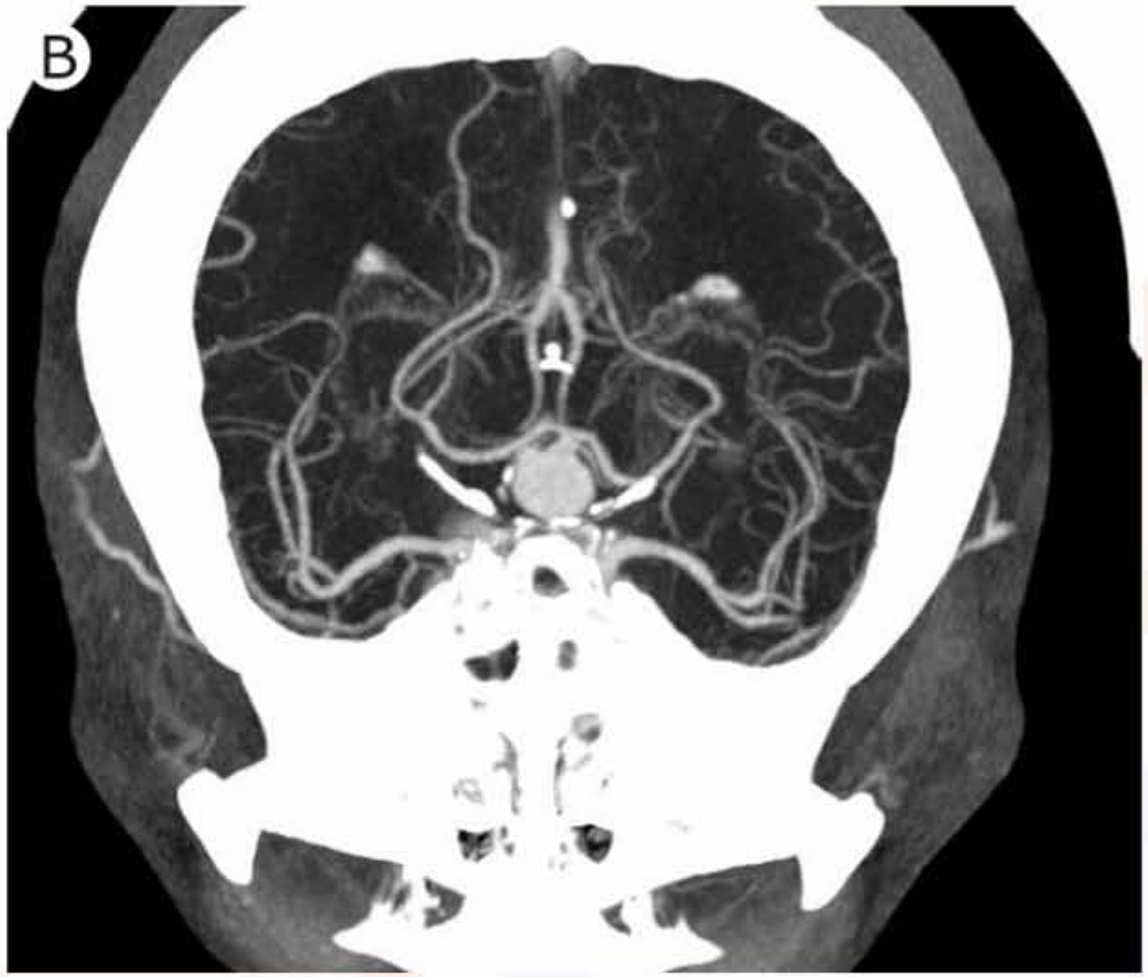
  Simply compute maximum value along a ray

```
#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



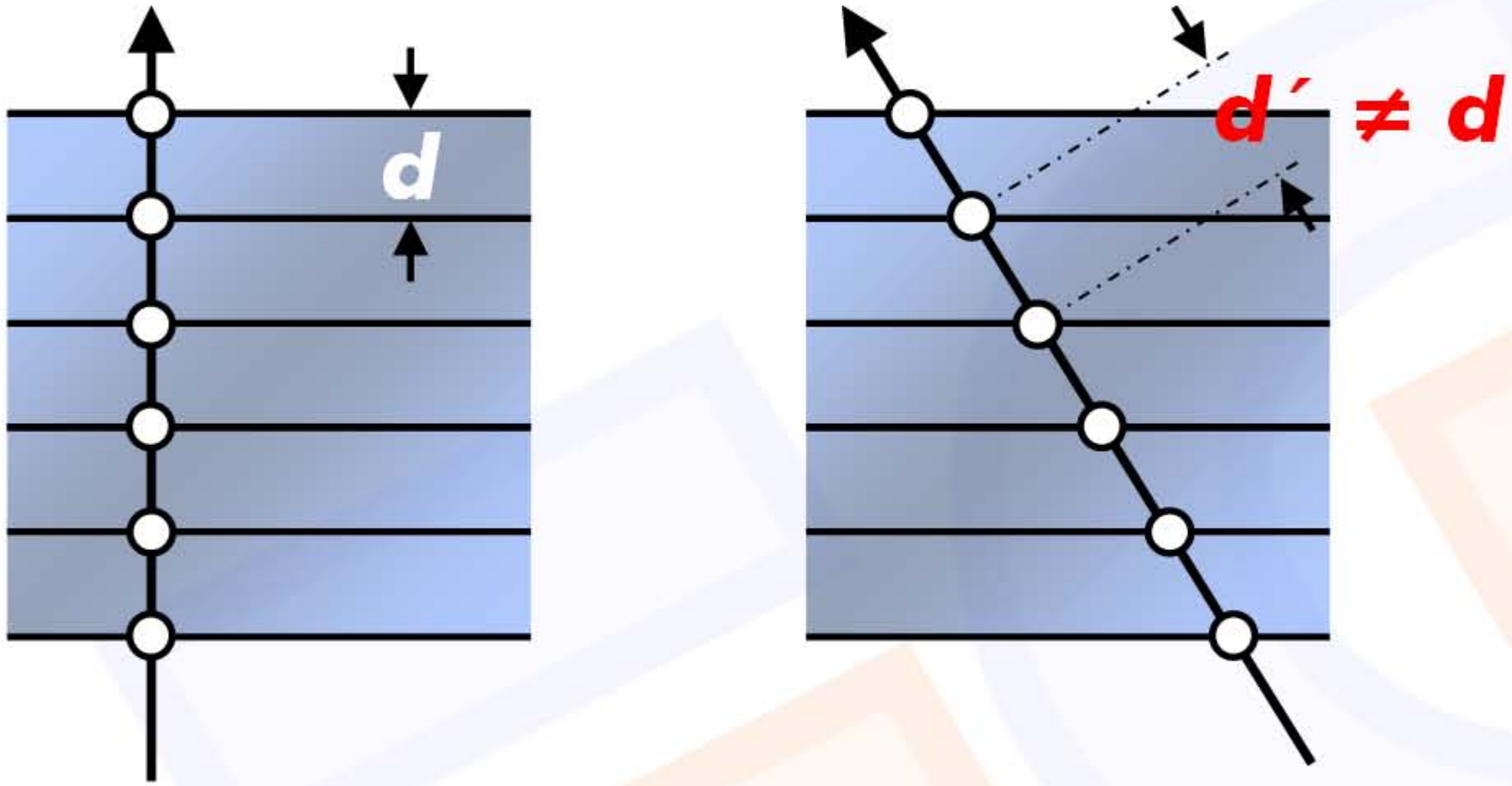
Emission/Absorption



Maximum Intensity Projection

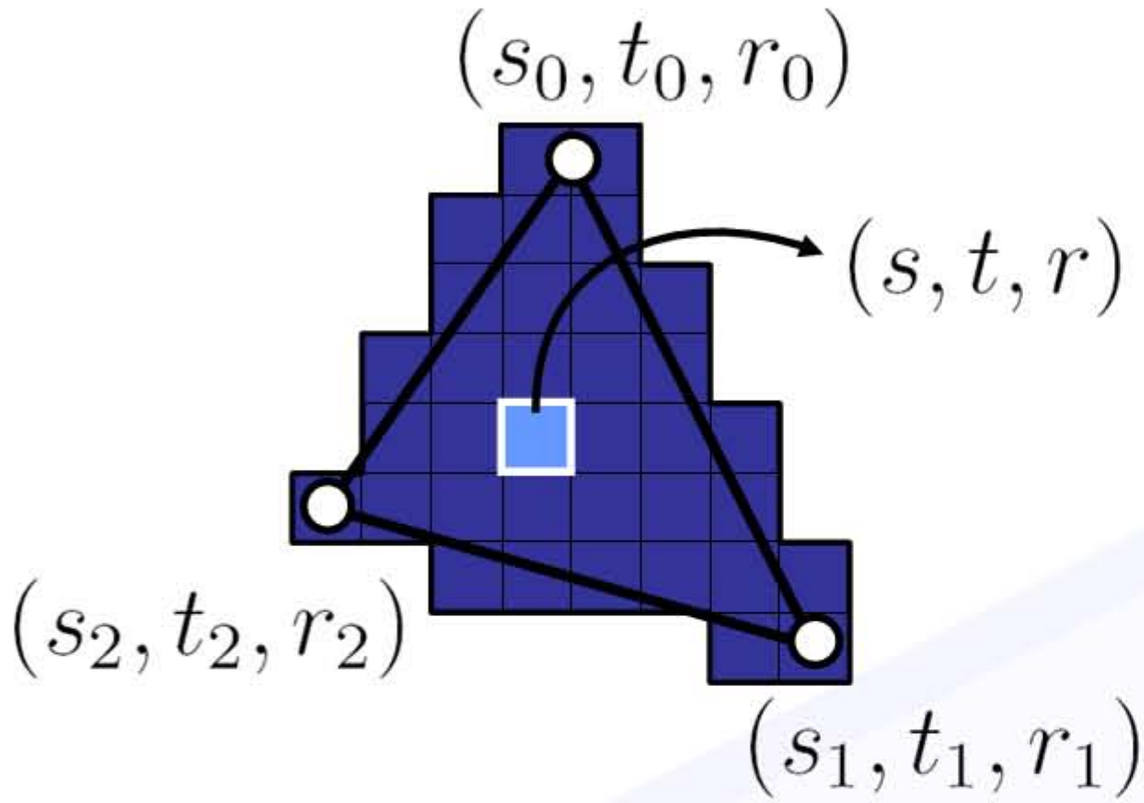
2D Textures: Drawbacks

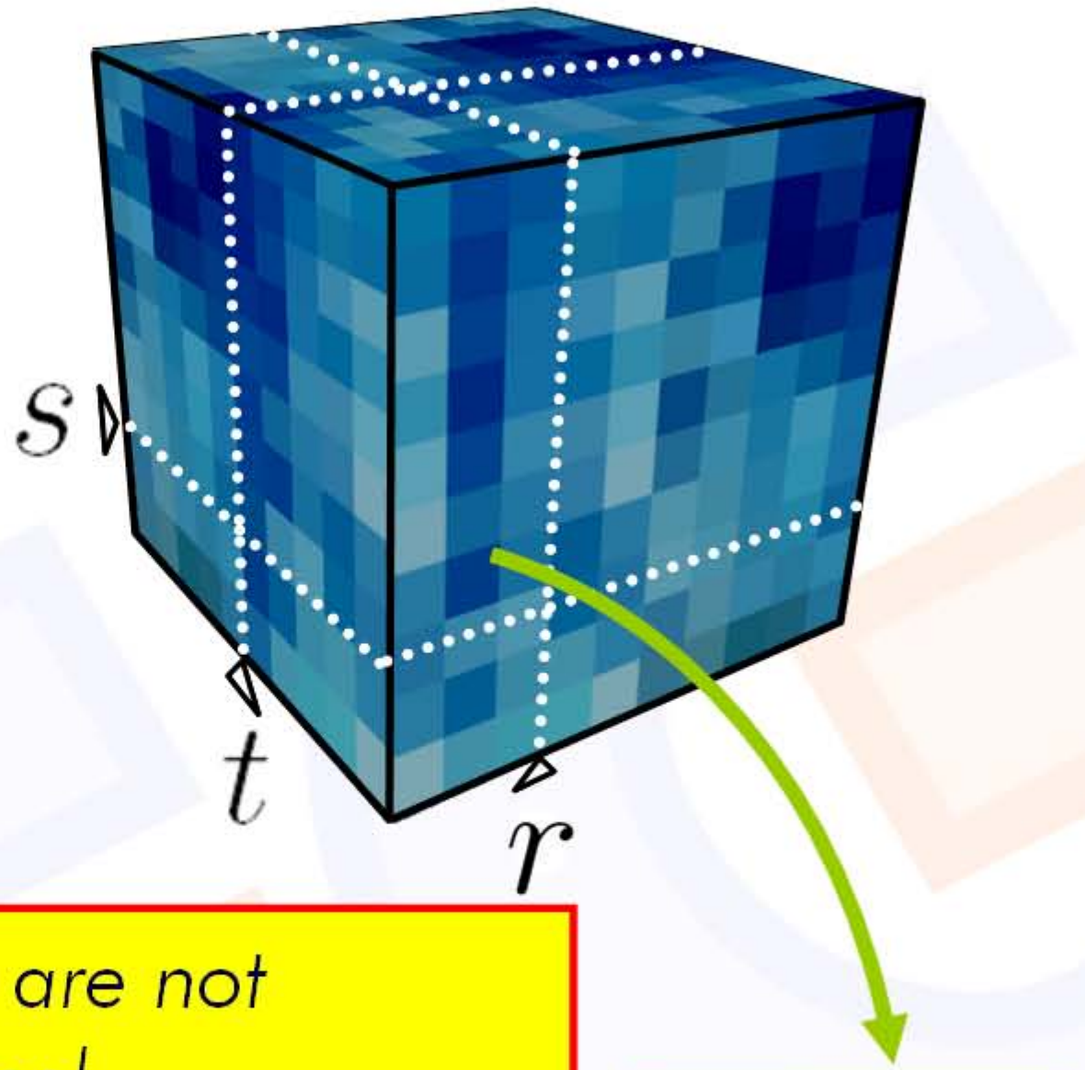
- Sampling rate is inconsistent



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

3D Textures





Don't be confused: 3D textures are not volumetric rendering primitives!

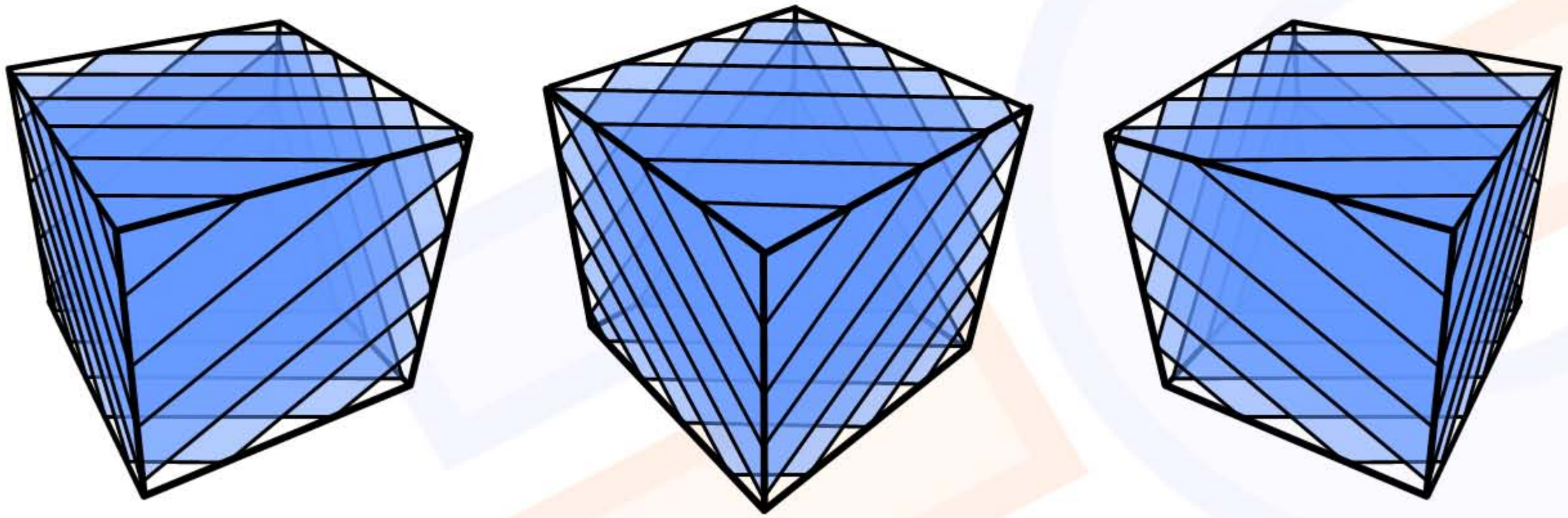
*Only **planar polygons** are supported as rendering primitives.*

3D Textures

3D Texture: Volumetric Texture Object

- Trilinear Interpolation in Hardware

⇨ Slices parallel to the image plane

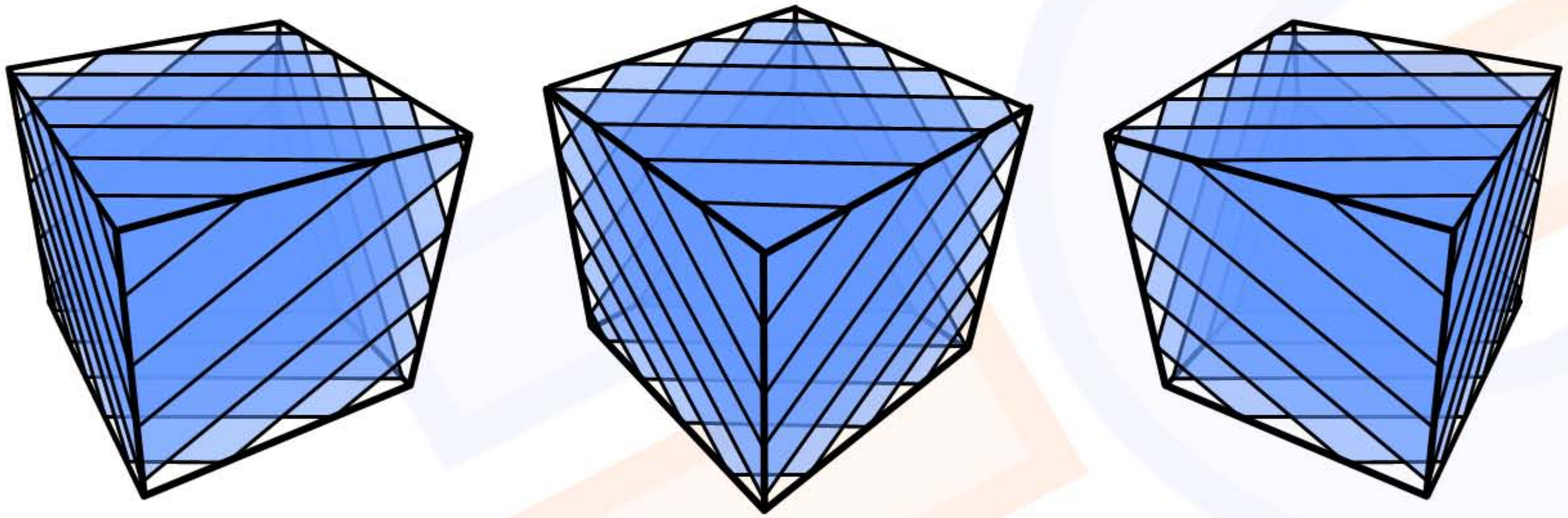


3D Textures

3D Texture: Volumetric Texture Object

- Trilinear Interpolation in Hardware

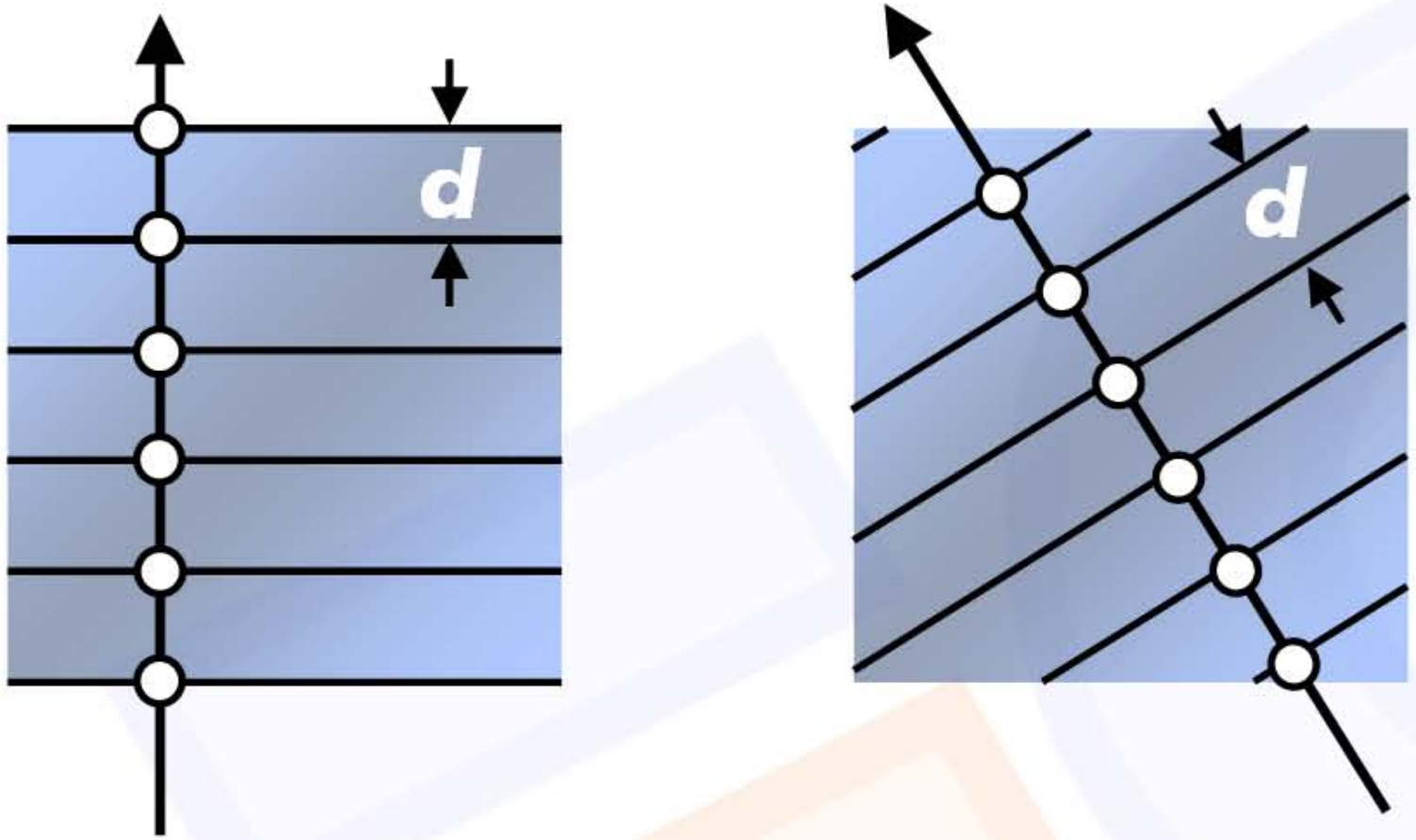
⇨ Slices parallel to the image plane



- One large texture block in memory

Resampling via 3D Textures

- *Sampling rate is constant*

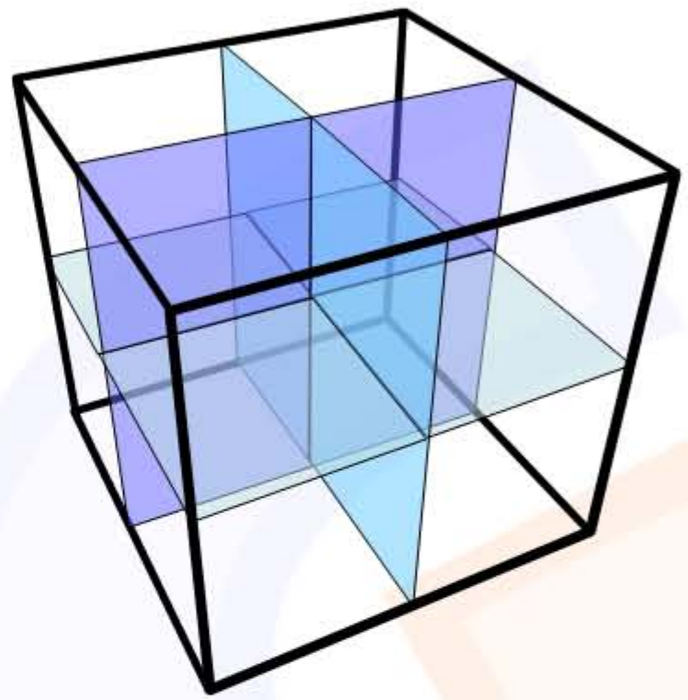


- Supersampling by increasing the number of slices

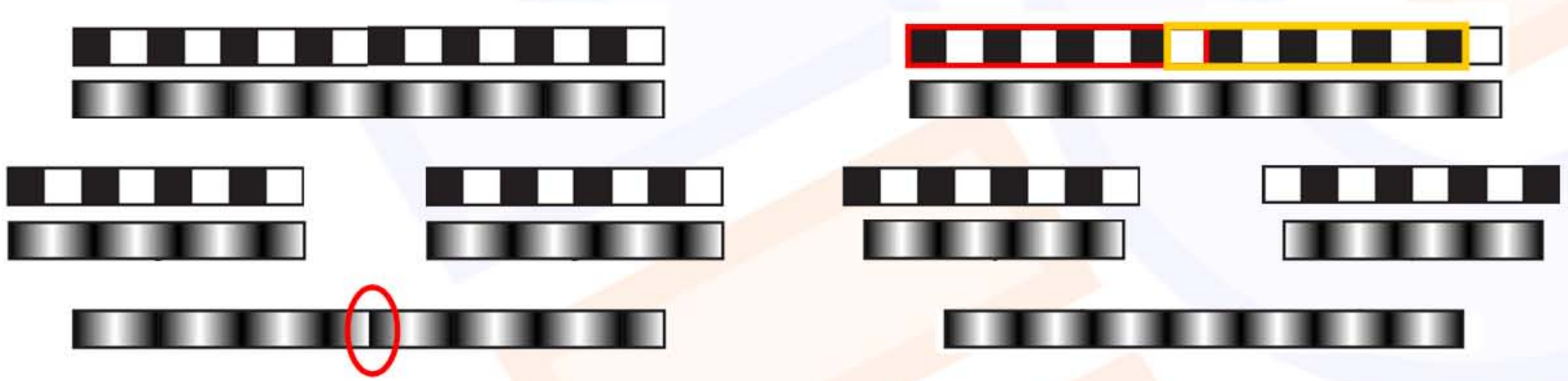
Bricking

- 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

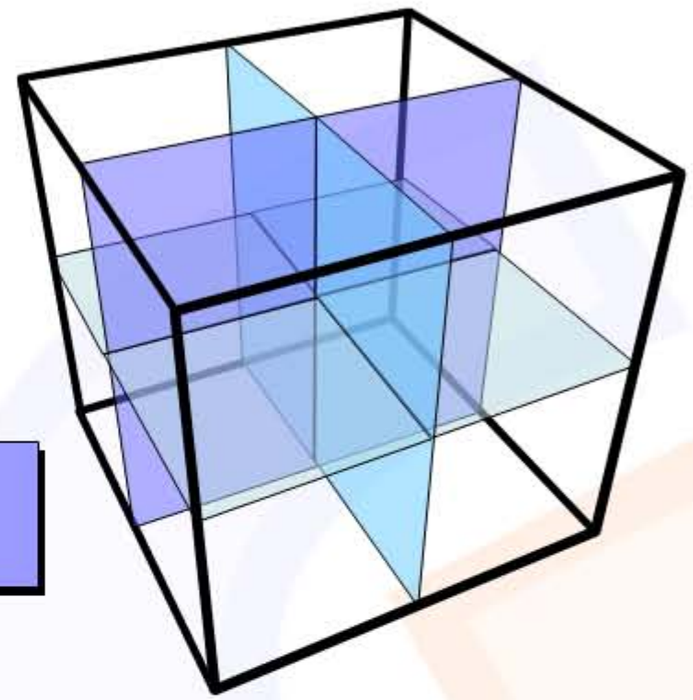


incorrect interpolation!

Bricking

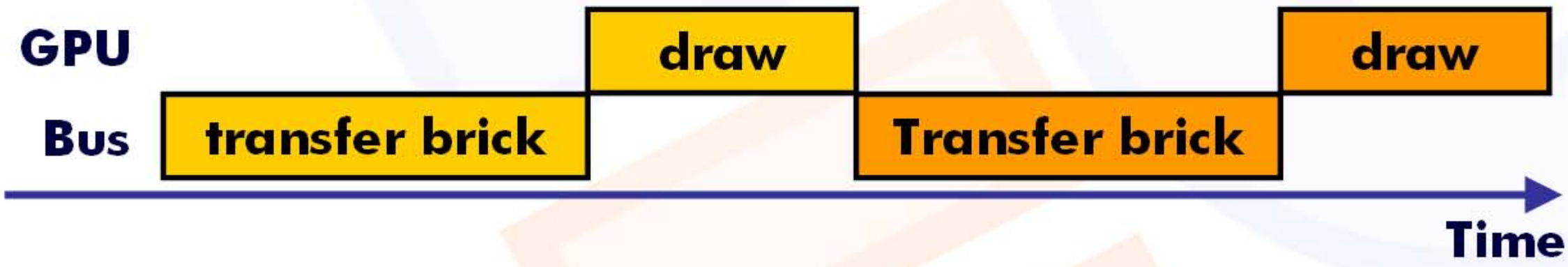
- What happens if data set is too large to fit into local video memory?

➡ Divide the data set into smaller chunks (bricks)



Problem: Bus-Bandwidth

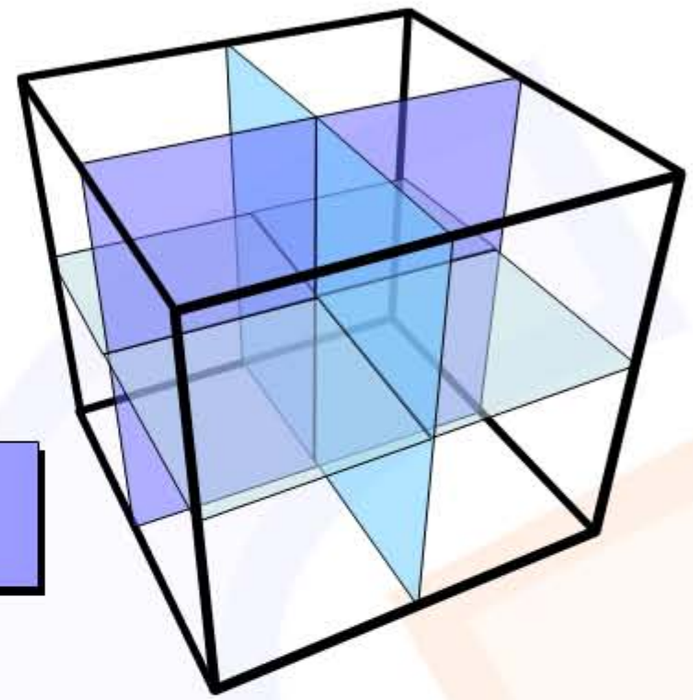
- Unbalanced Load for GPU und Memory Bus



Bricking

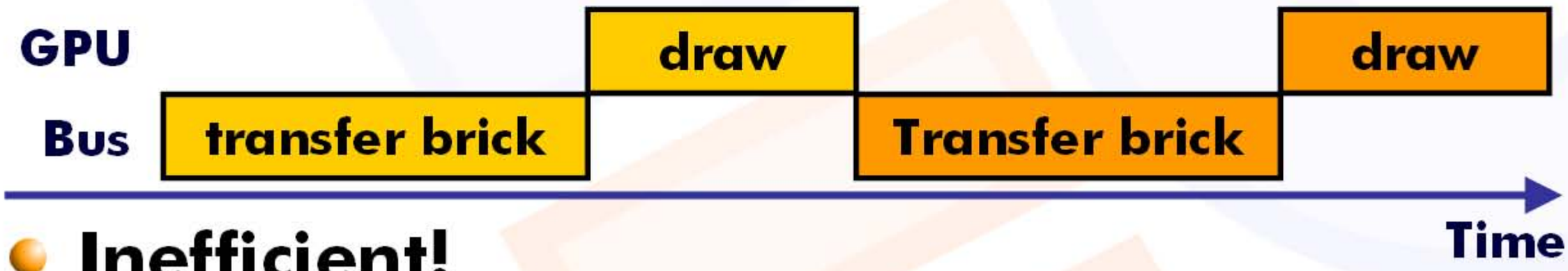
- What happens if data set is too large to fit into local video memory?

➡ Divide the data set into smaller chunks (bricks)



Problem: Bus-Bandwidth

- Unbalanced Load for GPU und Memory Bus

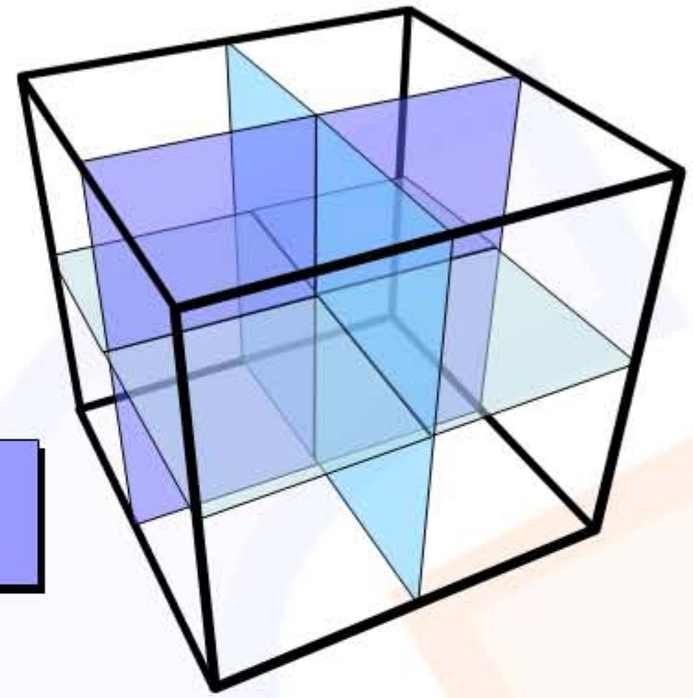


- **Inefficient!**

Bricking

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

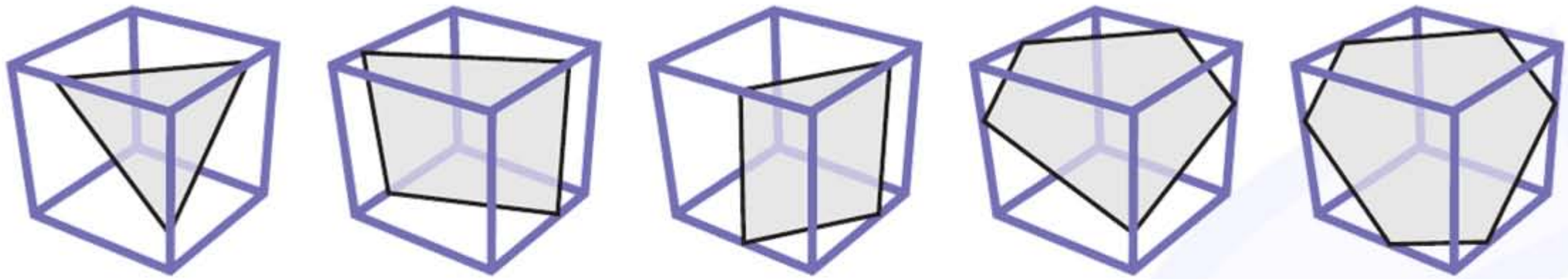
⇨ Divide the data set into smaller chunks (bricks)



Problem: Bus-Bandwidth

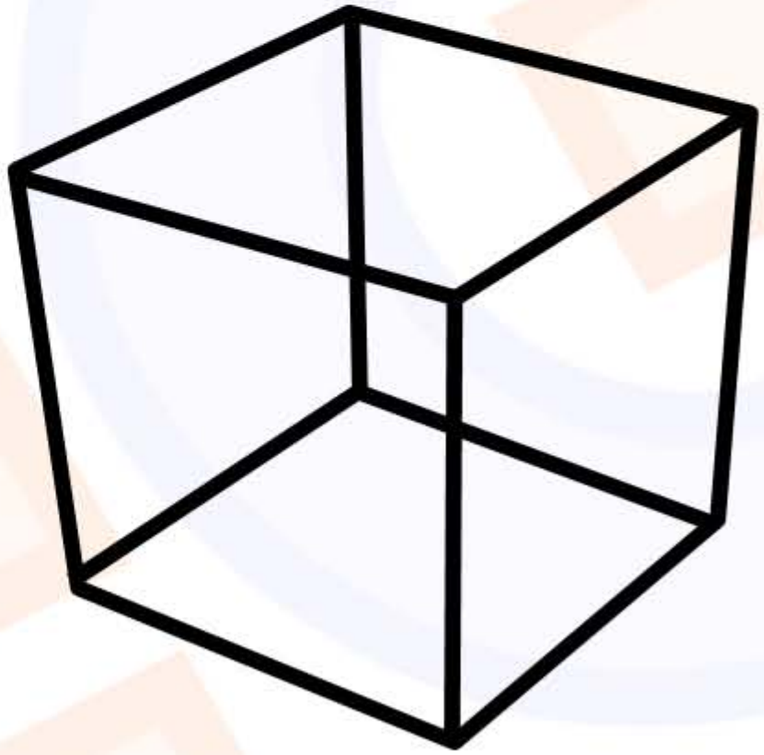
- Keep the bricks small enough!
  More than one brick must fit into video memory !
  - Transfer and Rendering can be performed in parallel
  - Increased CPU load for intersection calculation!
  - *Effective load balancing still very difficult!*

Cube-Slice Intersection

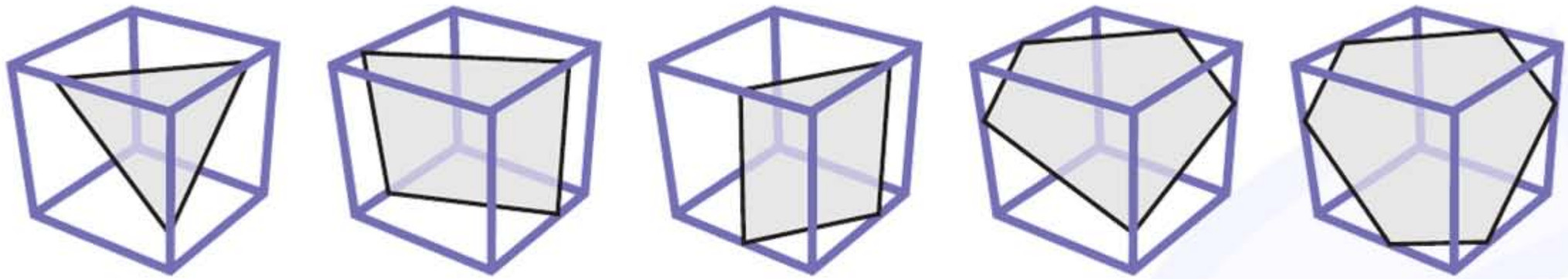


Question: Can we compute this in a vertex program?

Vertex program:
Input: 6 Vertices
Output: 6 Vertices

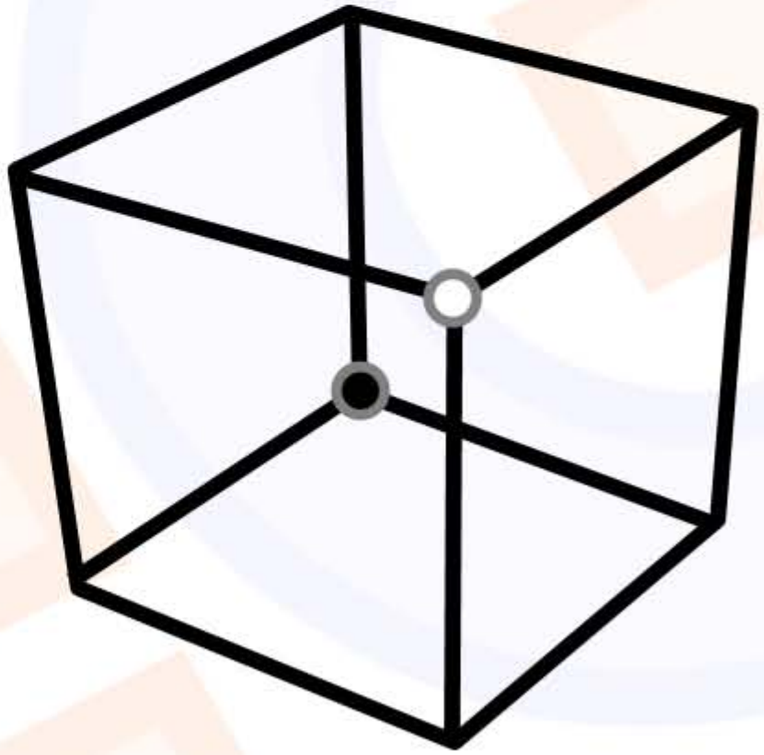


Cube-Slice Intersection

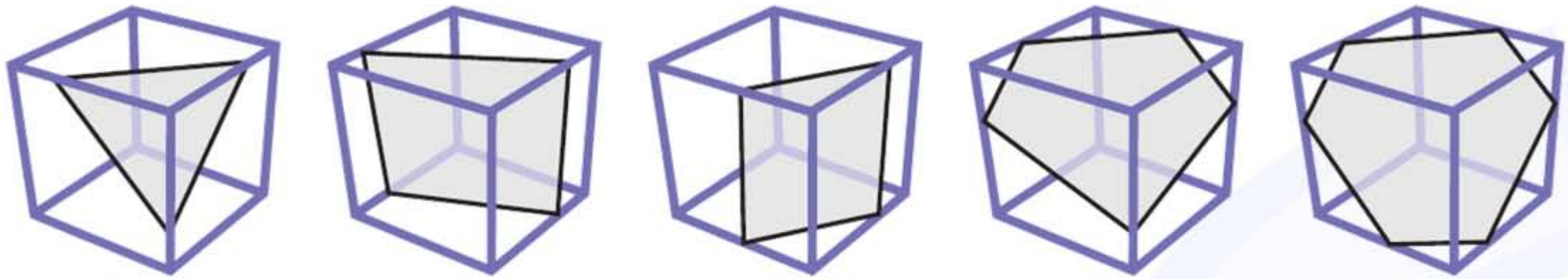


Question: Can we compute this in a vertex program?

Vertex program:
Input: 6 Vertices
Output: 6 Vertices

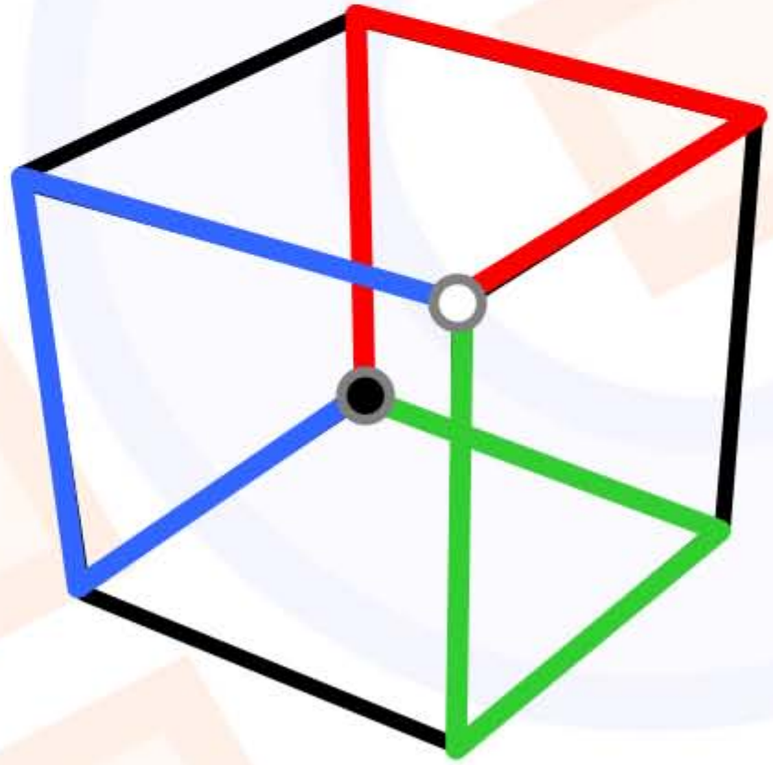


Cube-Slice Intersection

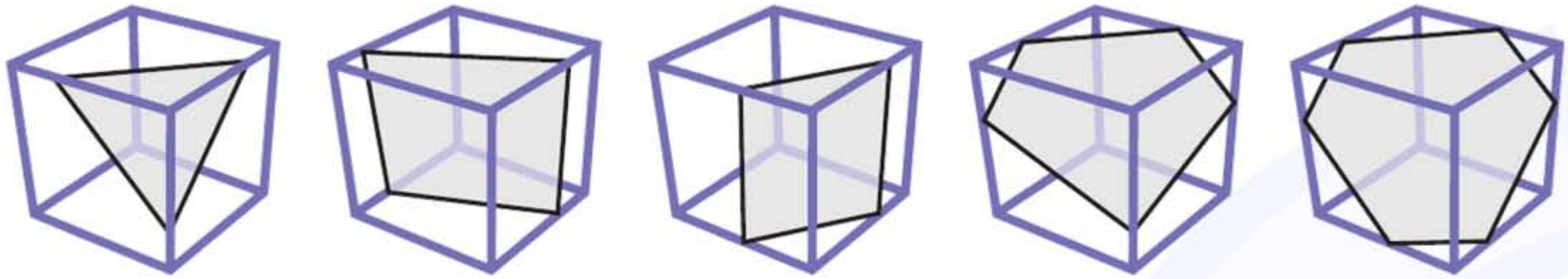


Question: Can we compute this in a vertex program?

Vertex program:
Input: 6 Vertices
Output: 6 Vertices

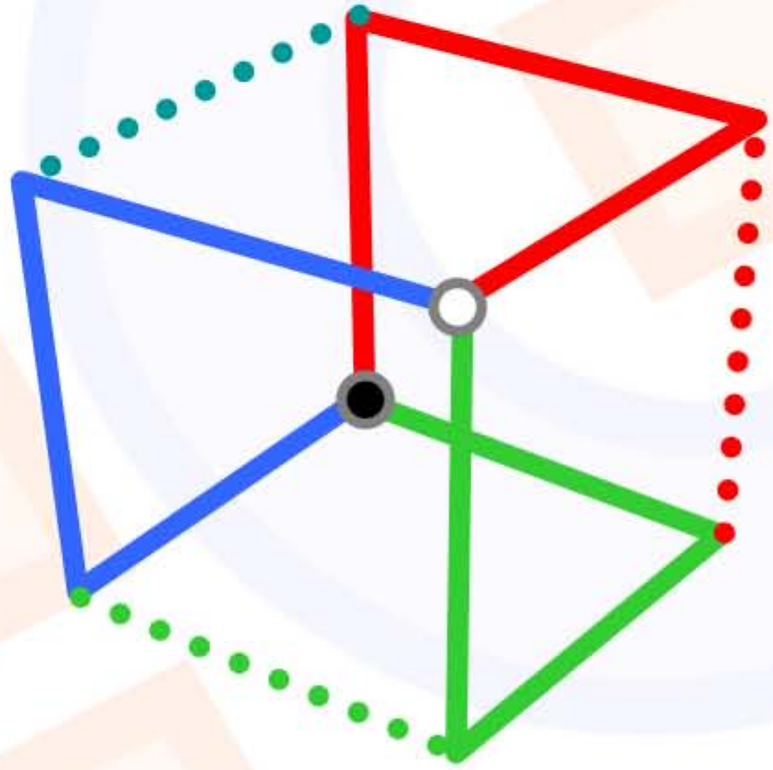


Cube-Slice Intersection

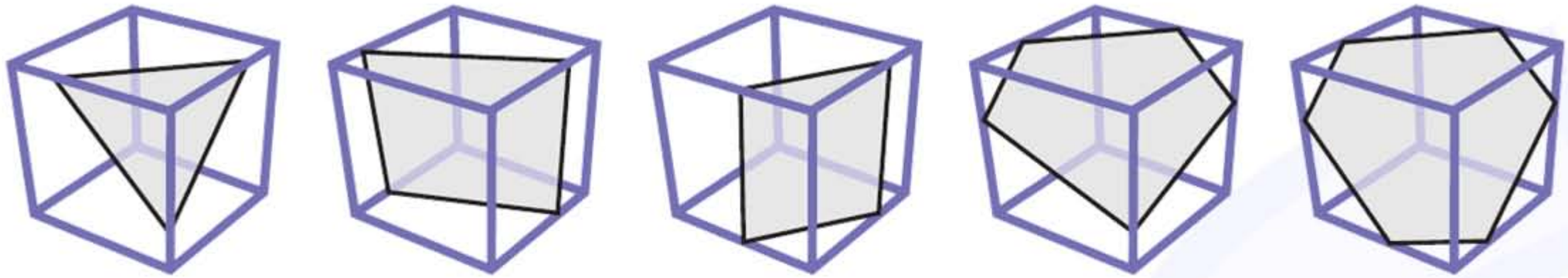


Question: Can we compute this in a vertex program?

Vertex program:
Input: 6 Vertices
Output: 6 Vertices



Cube-Slice Intersection



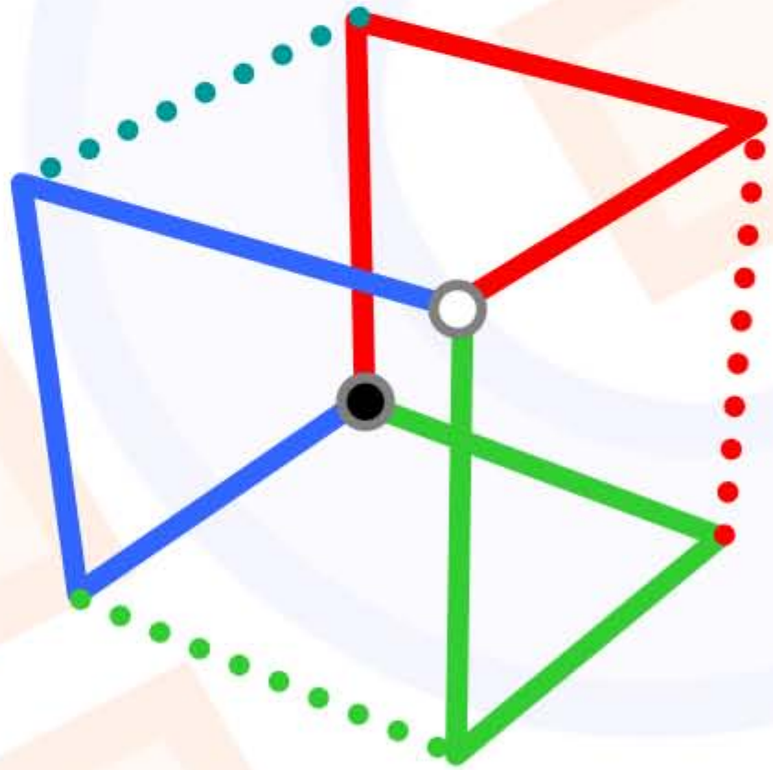
Question: Can we compute this in a vertex program?

Vertex program:
Input: 6 Vertices
Output: 6 Vertices

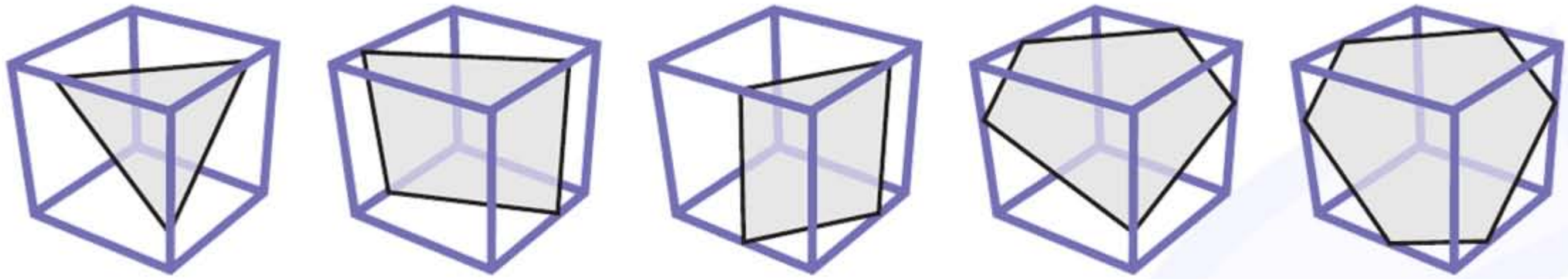
P0: Intersection with red path

P2: Intersection with green path

P4: Intersection with blue path



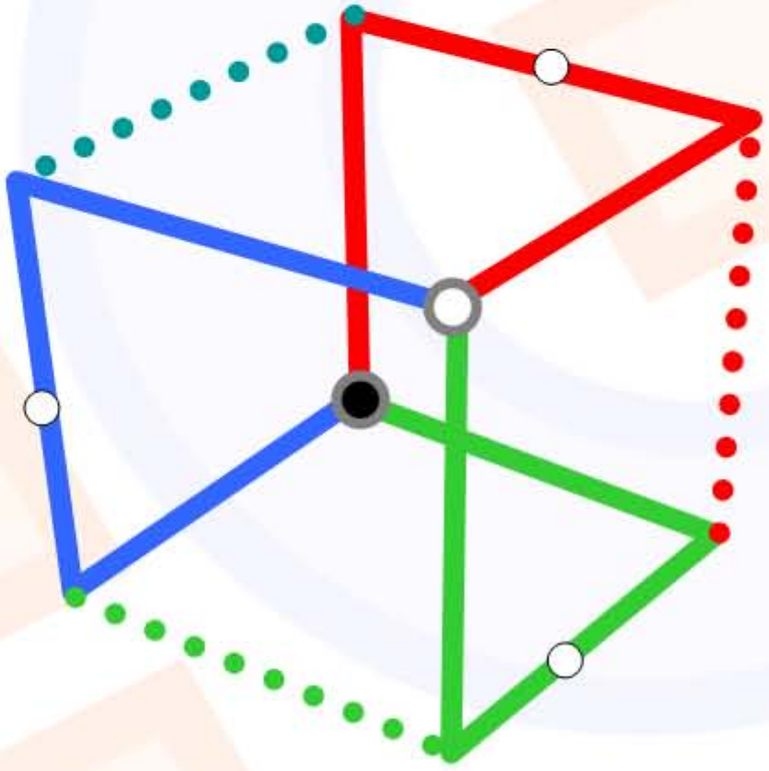
Cube-Slice Intersection



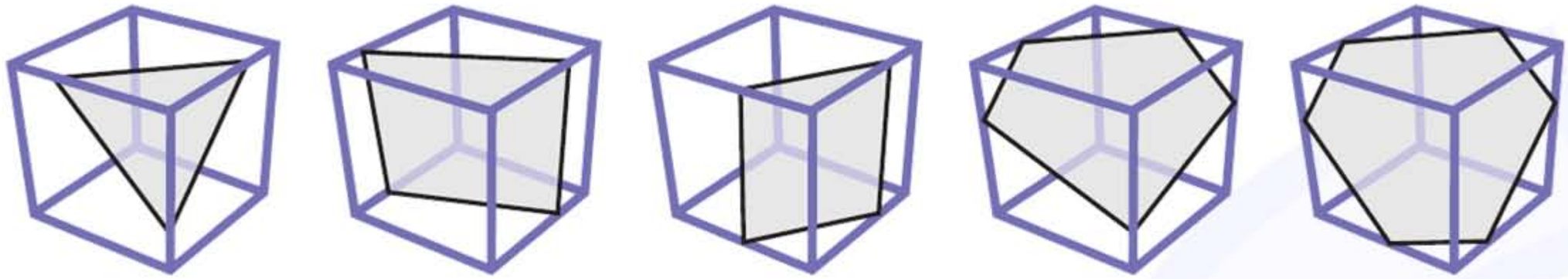
Question: Can we compute this in a vertex program?

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



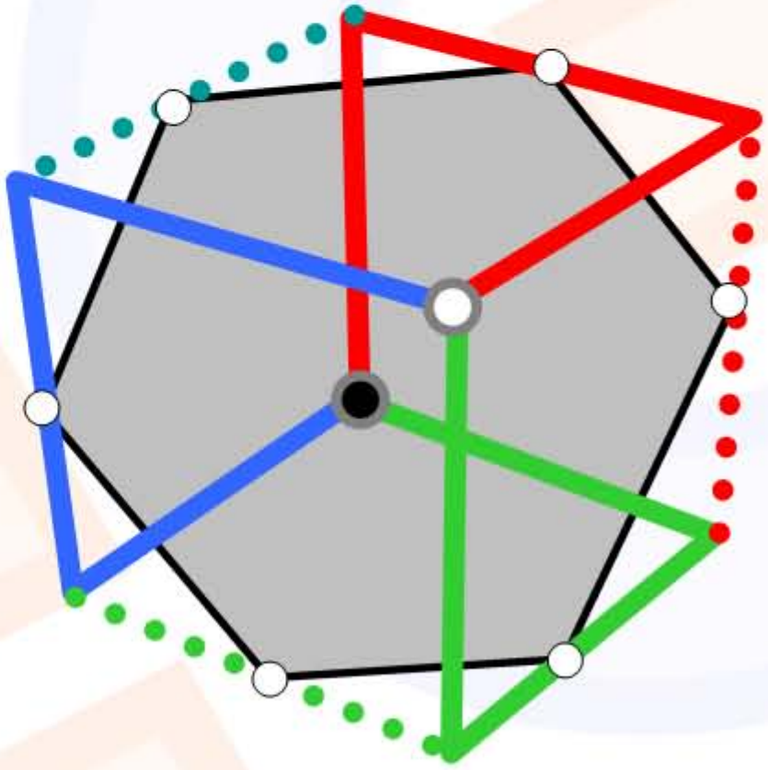
Cube-Slice Intersection



Question: Can we compute this in a vertex program?

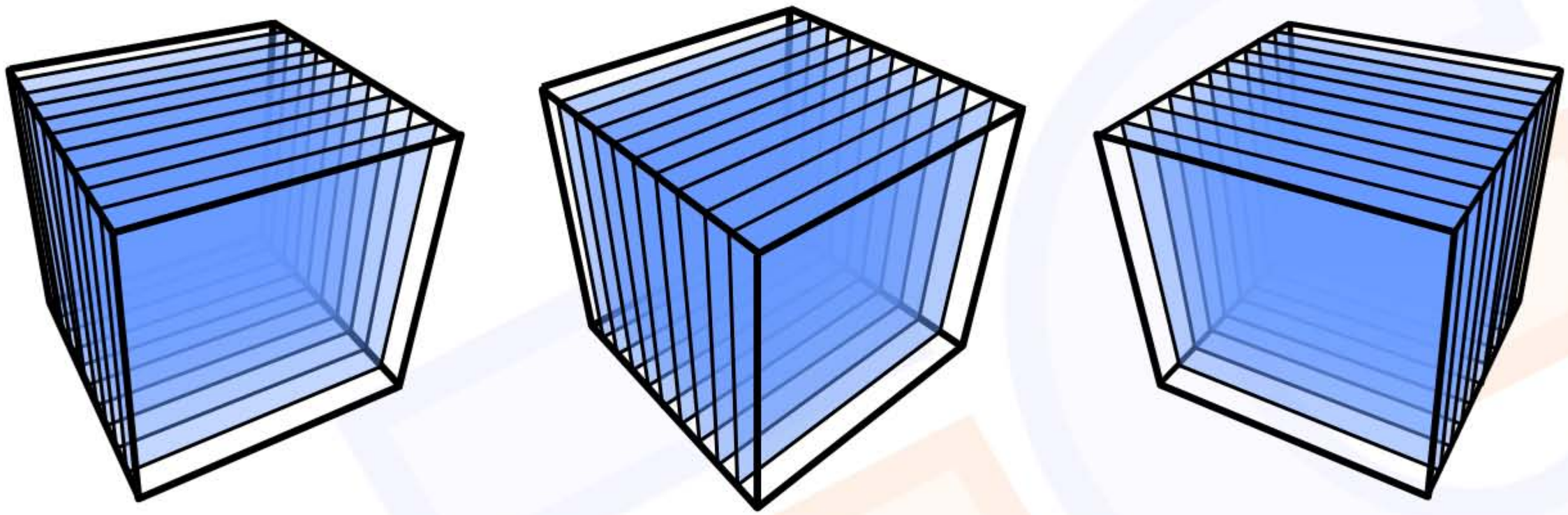
Vertex program:
Input: 6 Vertices
Output: 6 Vertices

- 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

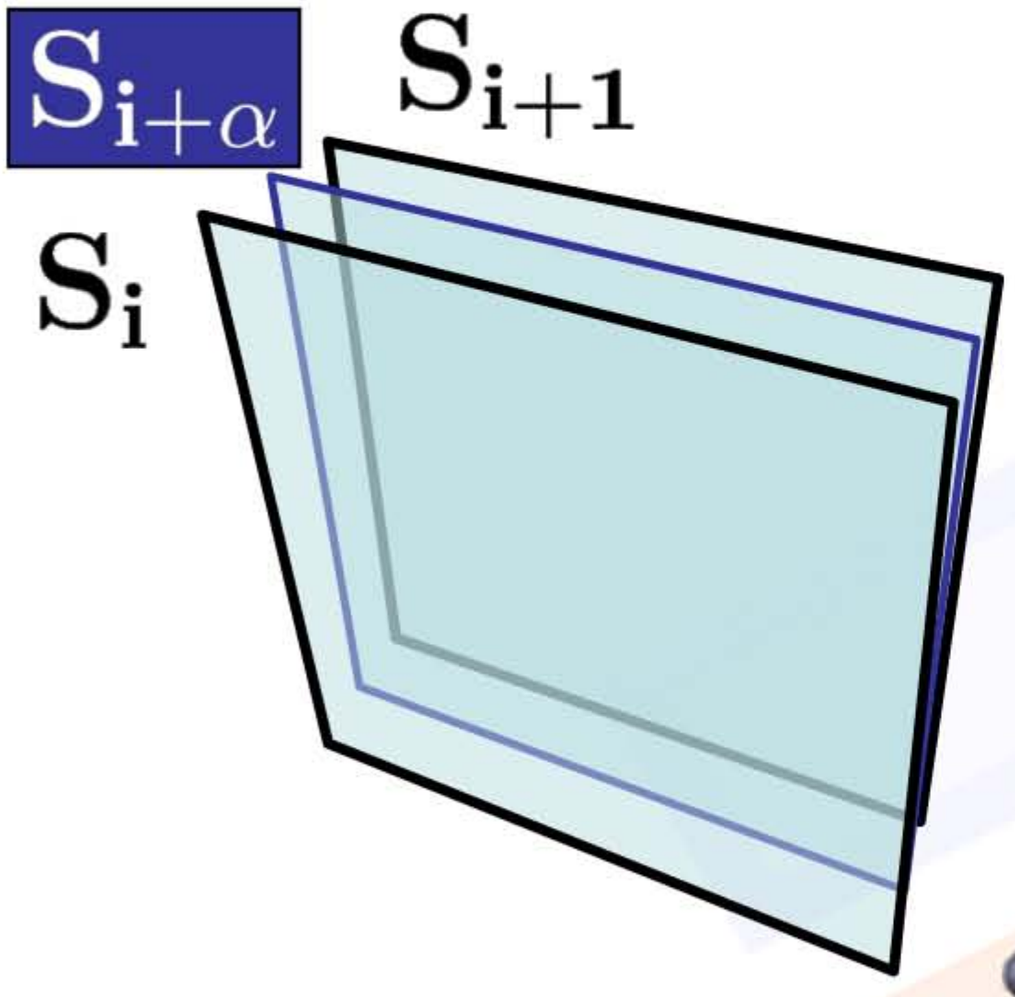
- ~~fixed~~ number of object aligned slices
- visual artifacts due to bilinear interpolation



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

2D Multi-Textures

Axis-Aligned Slices

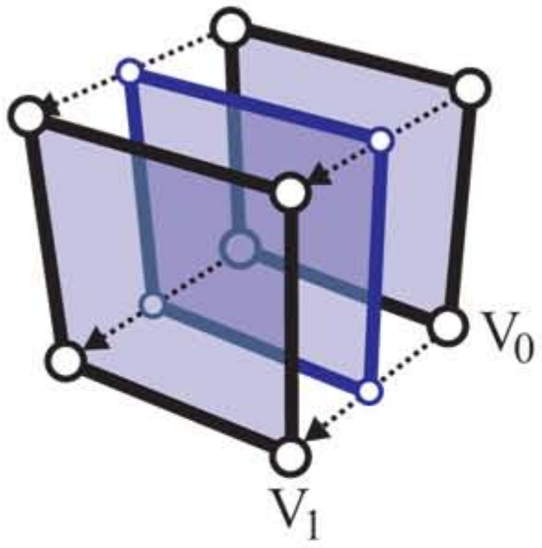


- Bilinear Interpolation by 2D Texture Unit
- Blending of two adjacent slice images

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

- Trilinear Interpolation

Implementation



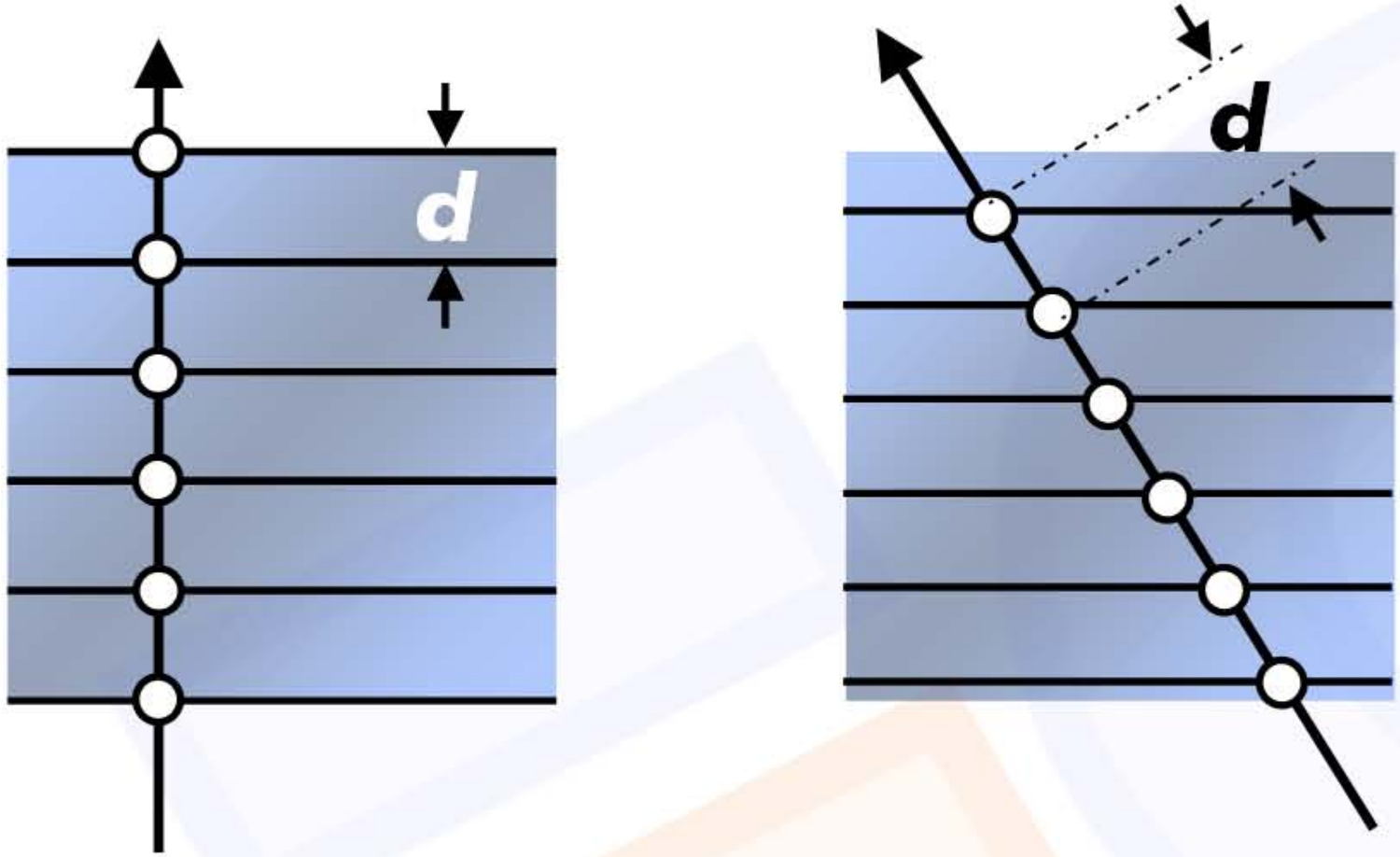

```
//vertex program for computing object aligned slices
void main( float4 Vertex0    :  POSITION,
           float4 Vertex1    :  TEXCOORD0,
           half2  TexCoord0  :  TEXCOORD1,

           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

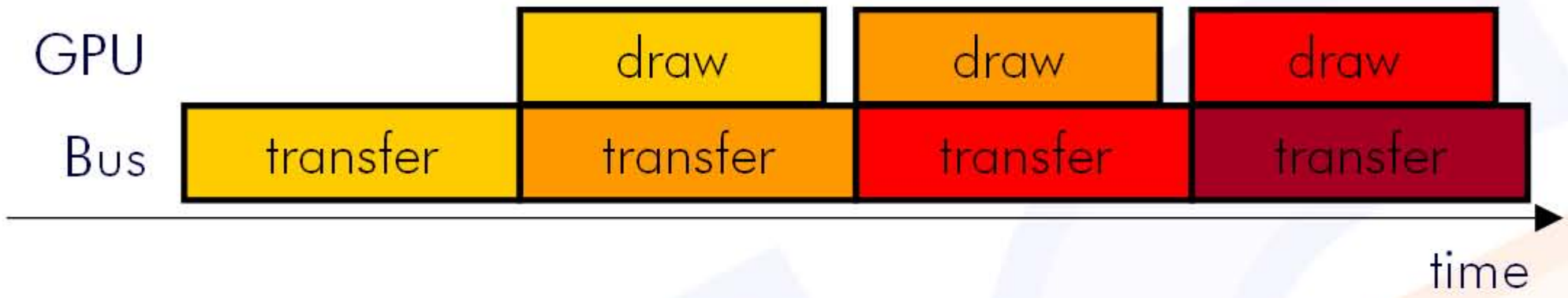
- Sampling rate is constant



- 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