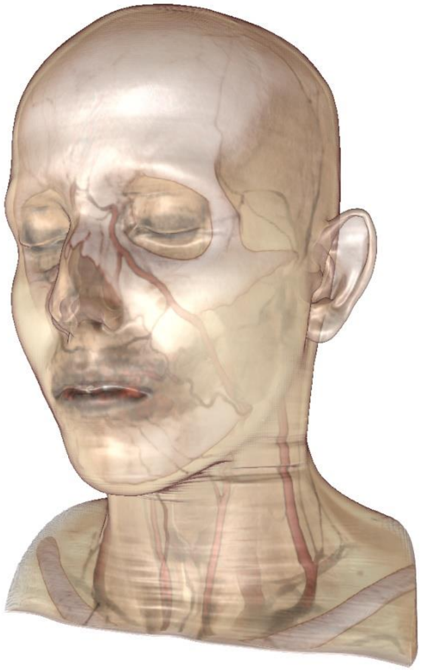


GPU-Based Ray-Casting

Scientific Visualization

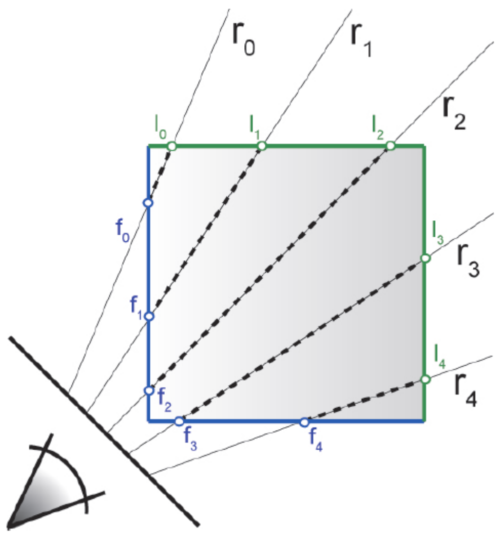
Pere-Pau Vázquez – Dep. Computer Science – UPC

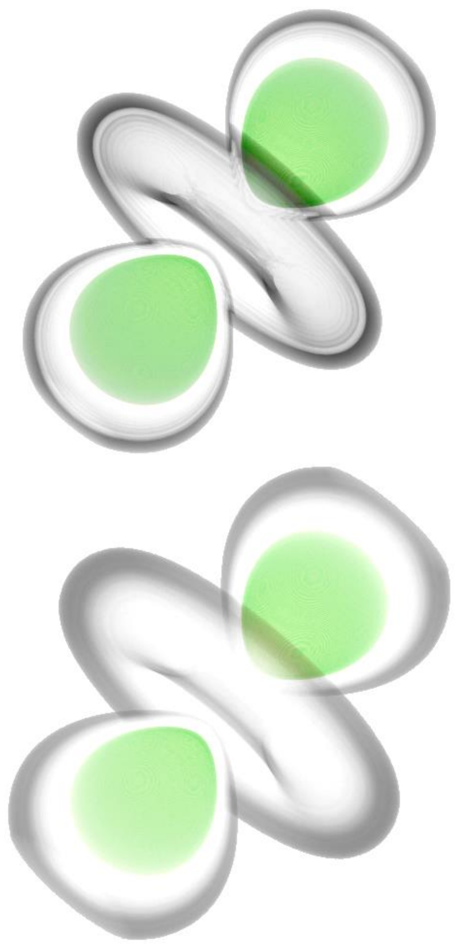
Outline

- Ray-casting of rectilinear (structured) grids
- Optimizations

Why Ray-Casting on GPUs?

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

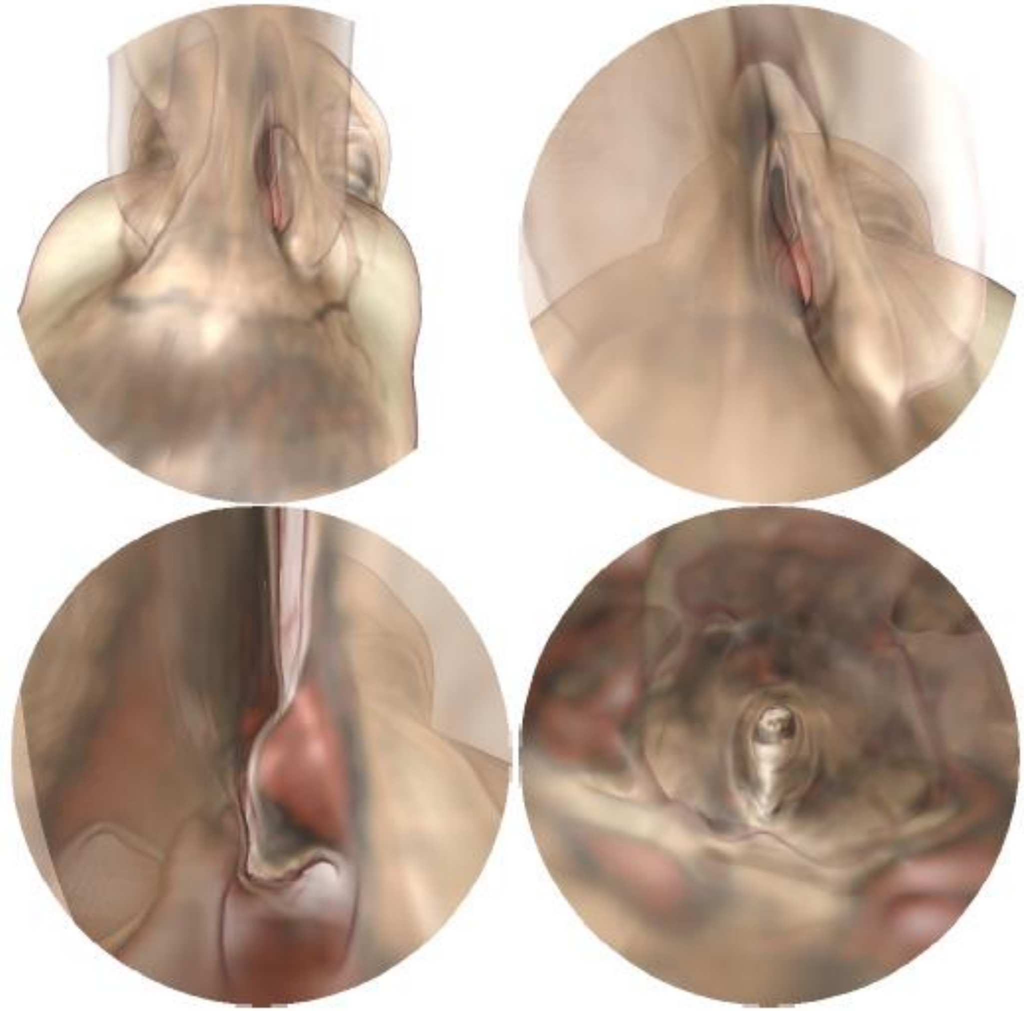




Where Is Correct Perspective Needed?

- Entering the volume
- Wide field of view

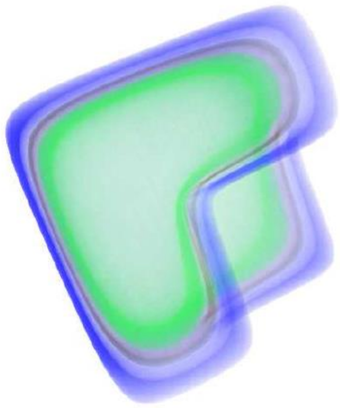
- Fly-throughs
- Virtual endoscopy
- Integration into perspective scenes, e.g., games



Recent GPU Ray-Casting Approaches

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

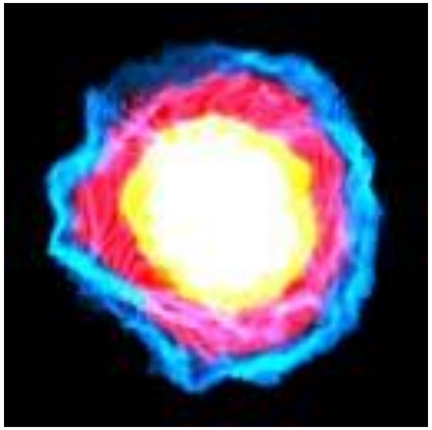
- Unstructured (tetrahedral) grids
  - [Weiler et al., 2002, 2003, 2004]
  - [Bernardon, 2004]



Single-Pass Ray-Casting

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

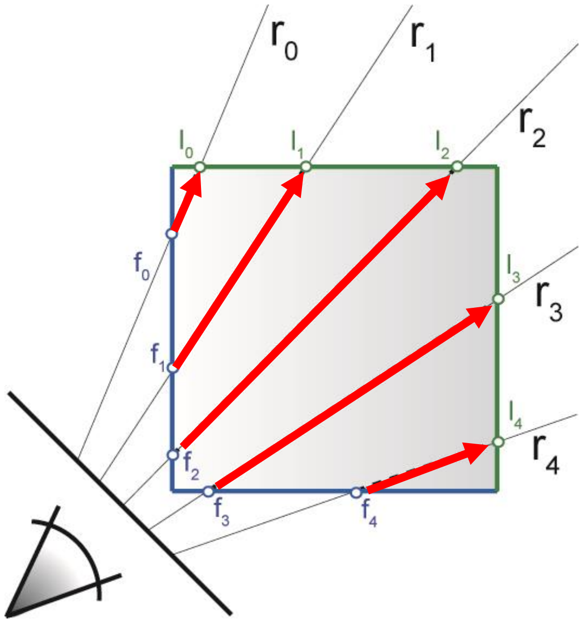
- NVIDIA example: compute ray intersections with bounding box, march along rays and composite



Basic Ray Setup / Termination

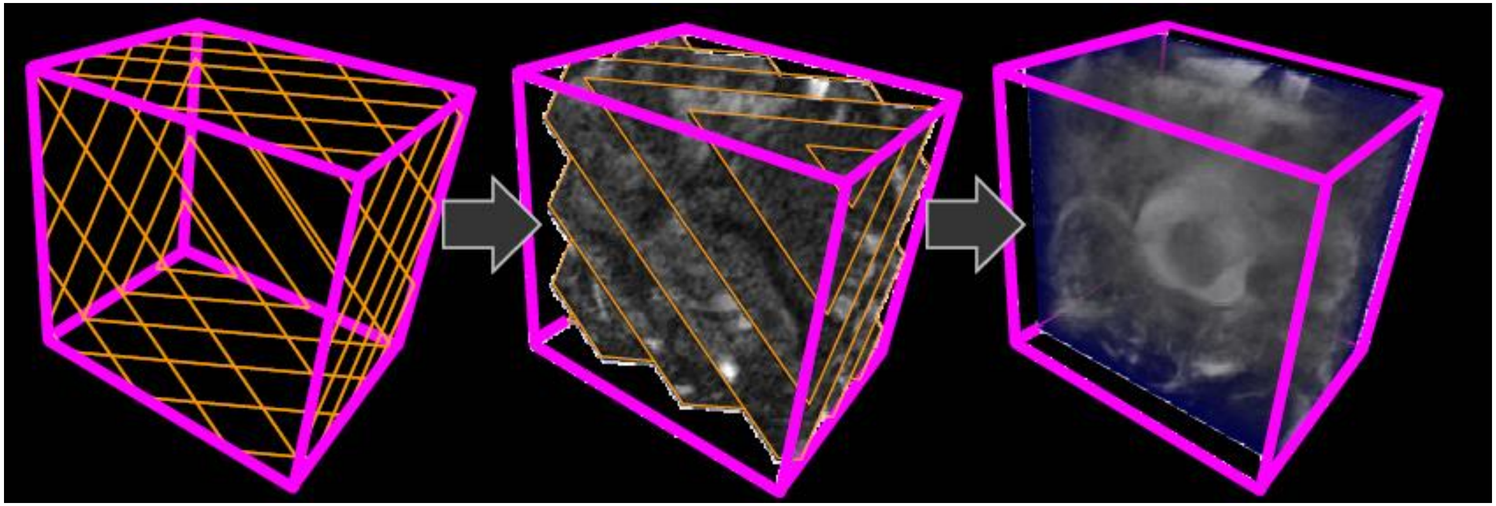
- Two main approaches:
  - Procedural ray/box intersection
    [Röttger et al., 2003], [Green, 2004]
  - Rasterize bounding box
    [Krüger and Westermann, 2003]

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



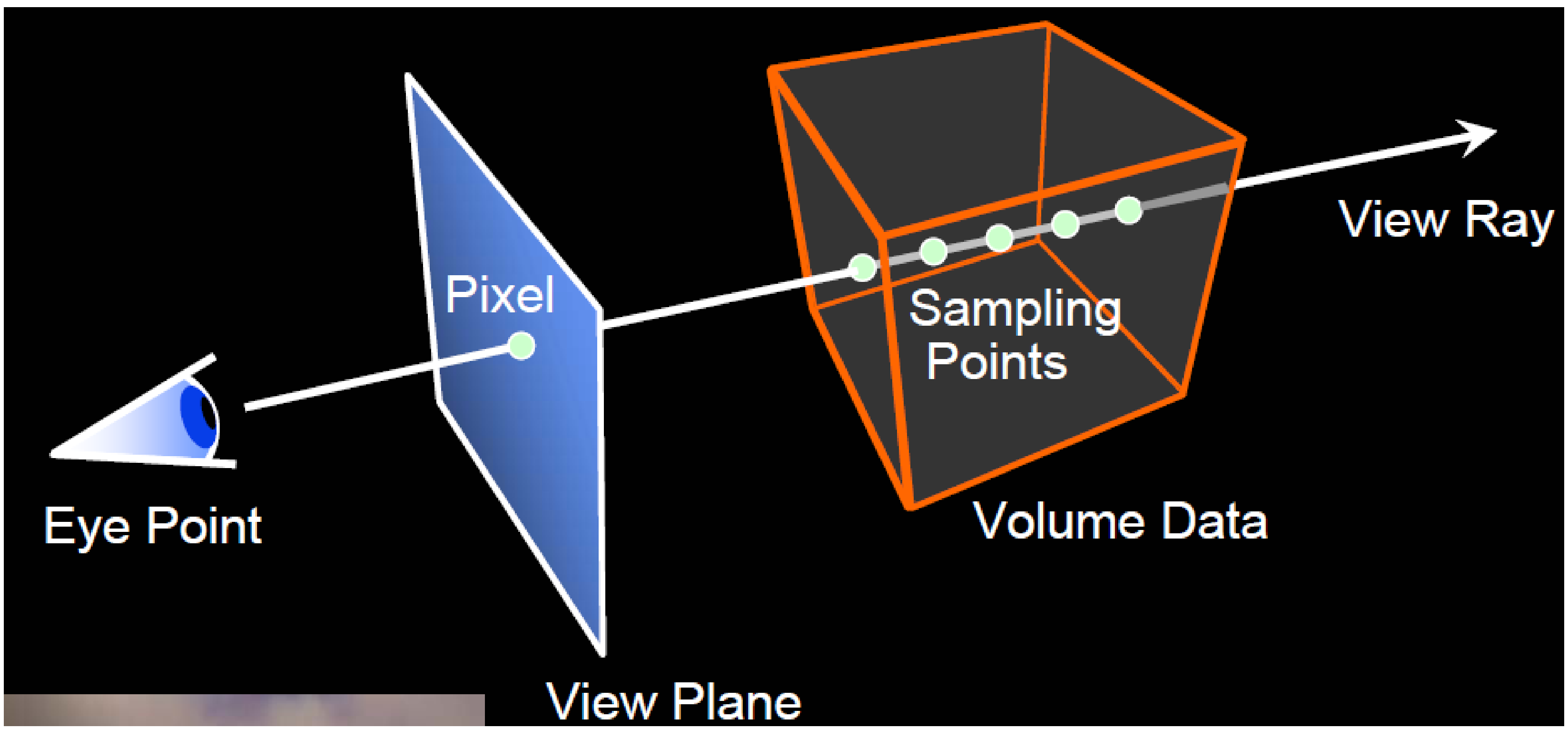
Parallel volume rendering with a single GPU

- Many volume rendering algorithms cannot fully exploit parallel pipelines
  - Cell projection: requires visibility sorting of cells
  - Textured slices: rasterize all fragments of all slices



Parallel volume rendering with a single GPU

- Ray casting can exploit parallel pixel pipelines:
  - Like ray tracing, ray casting is embarrassingly parallel

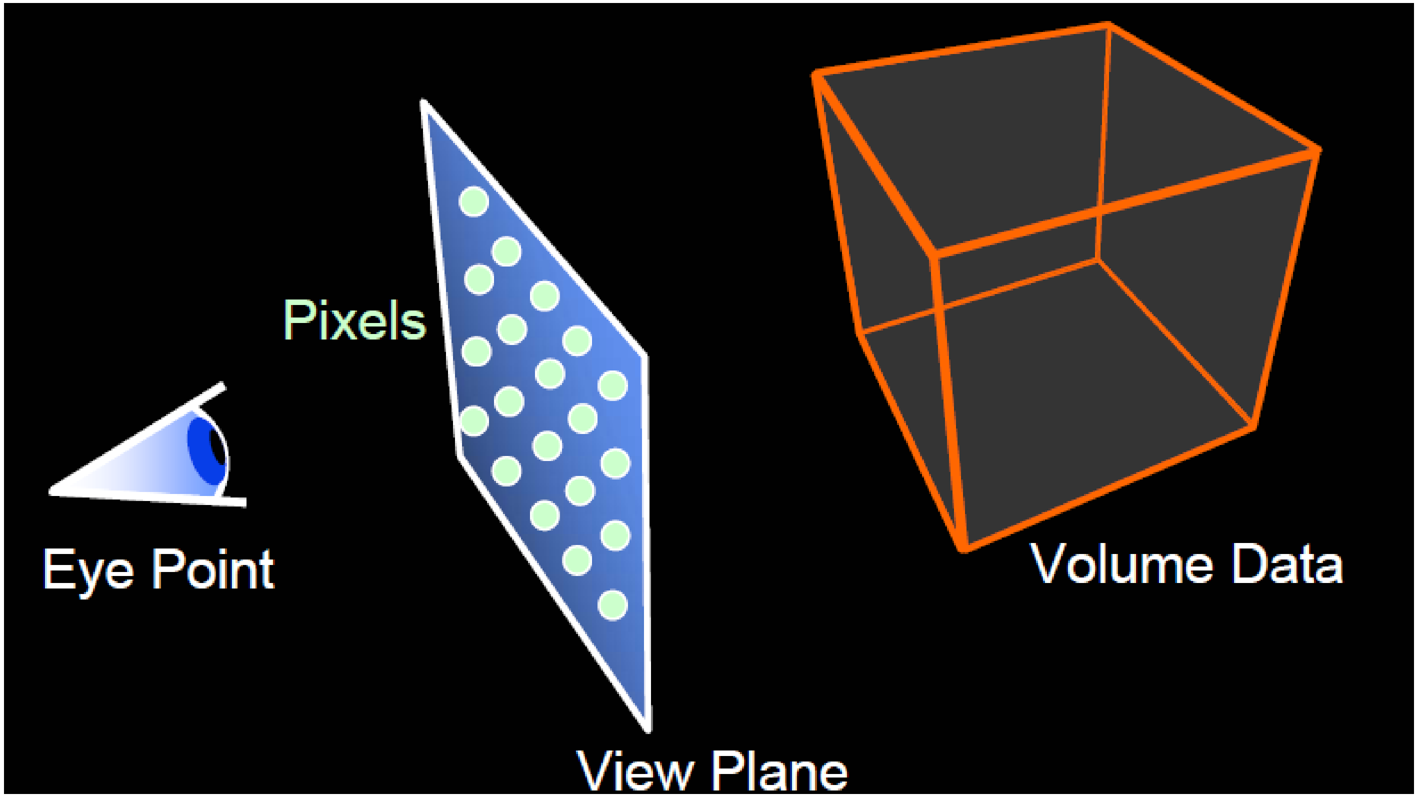


Parallel volume rendering with a single GPU

- More benefits of ray casting on a GPU:
  - Applicable to uniform and tetrahedral meshes
  - Important optimizations can be implemented
    - We will see later…

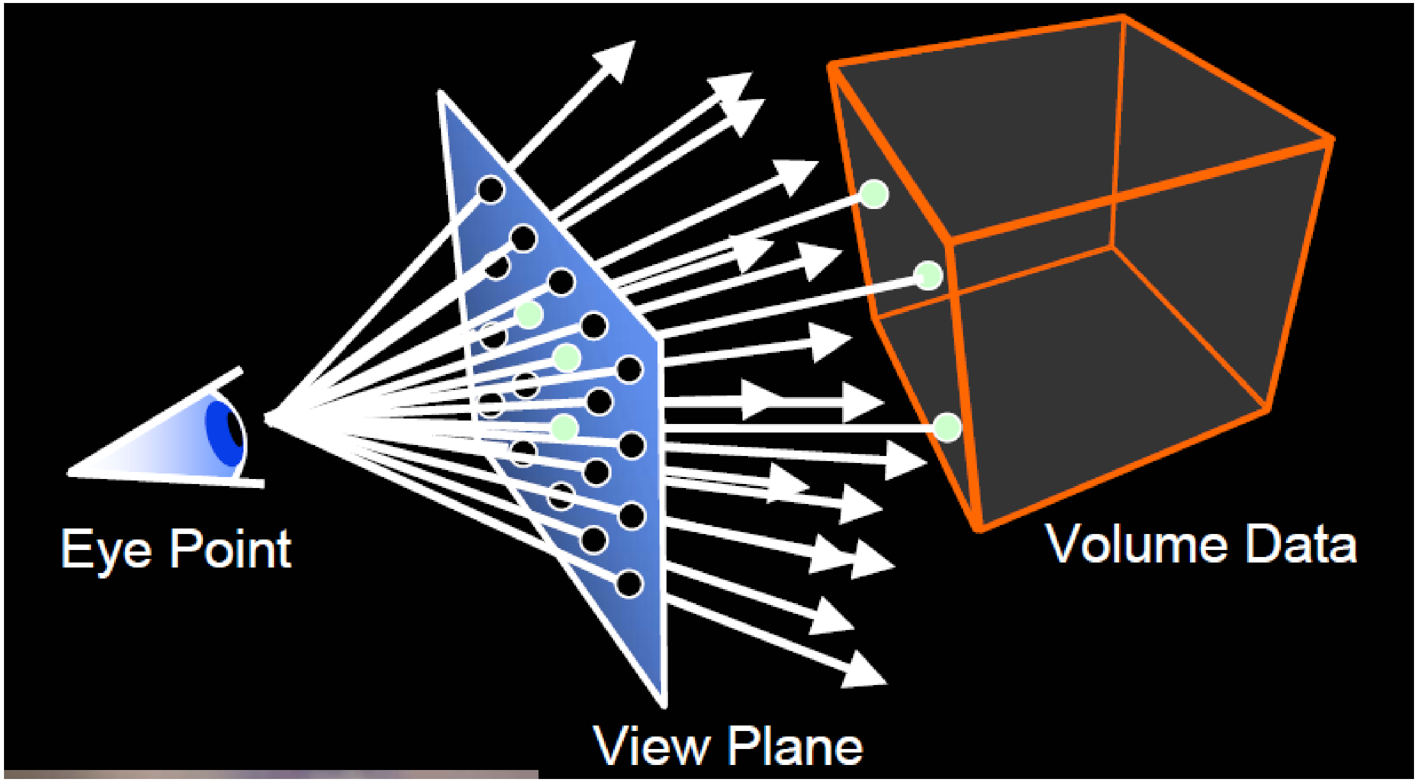
Parallel volume rendering with a single GPU

- Basic idea
  - Multi-pass approach: Render screen-filling rectangles to call a program for each pixel



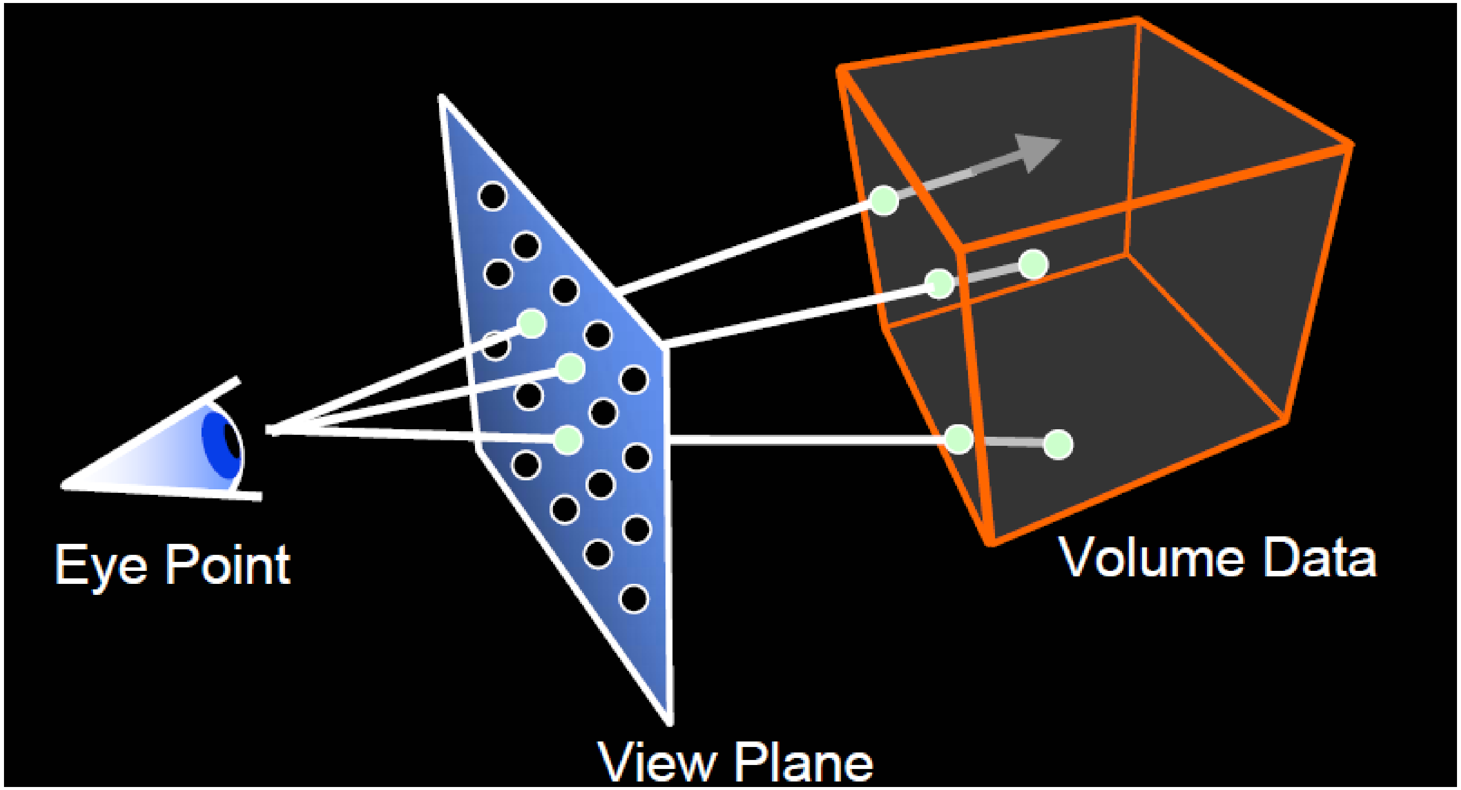
Parallel volume rendering with a single GPU

- Initialize pixels with first intersection of the rays with the volume data



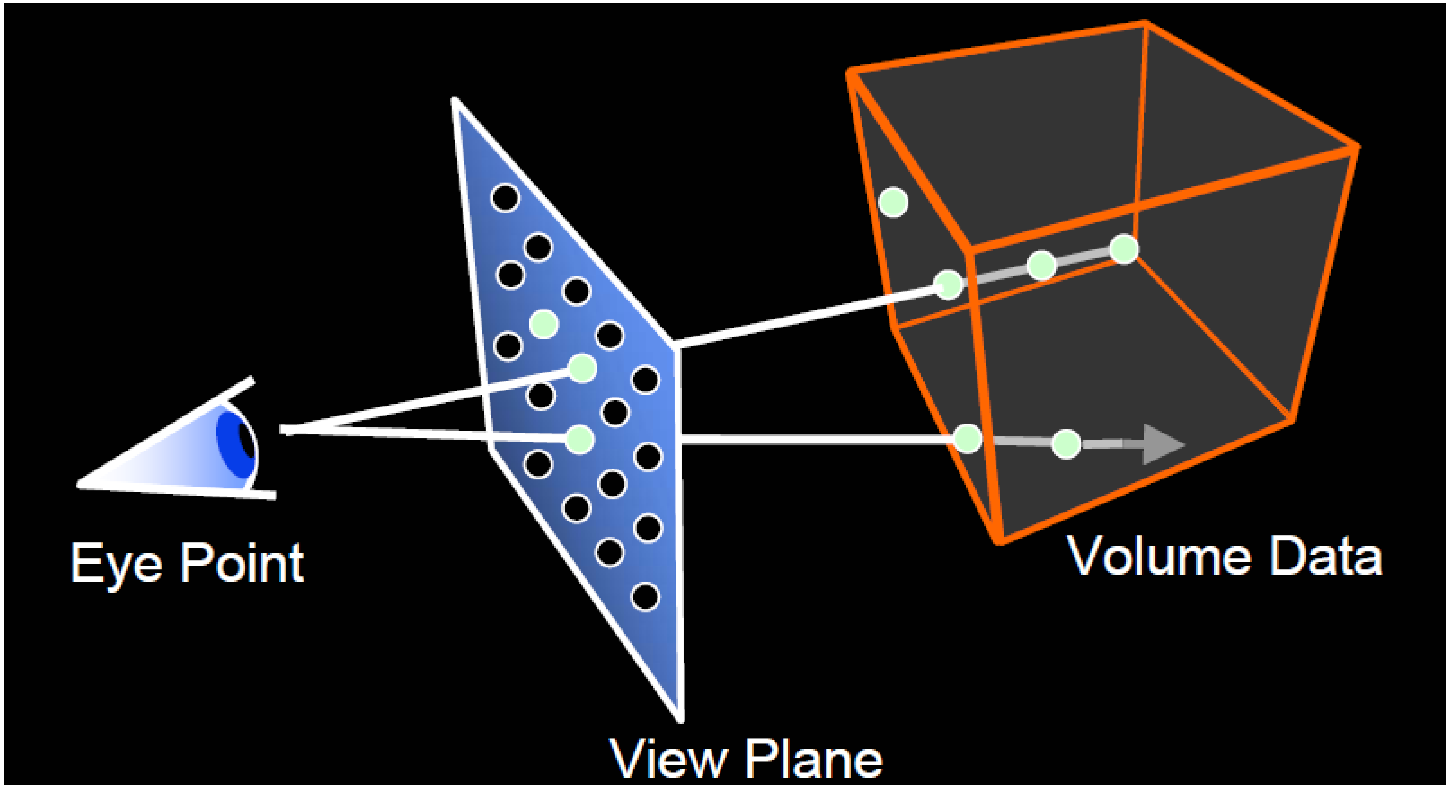
Parallel volume rendering with a single GPU

- In each pass, propagate view rays and store accumulated color and sampling position



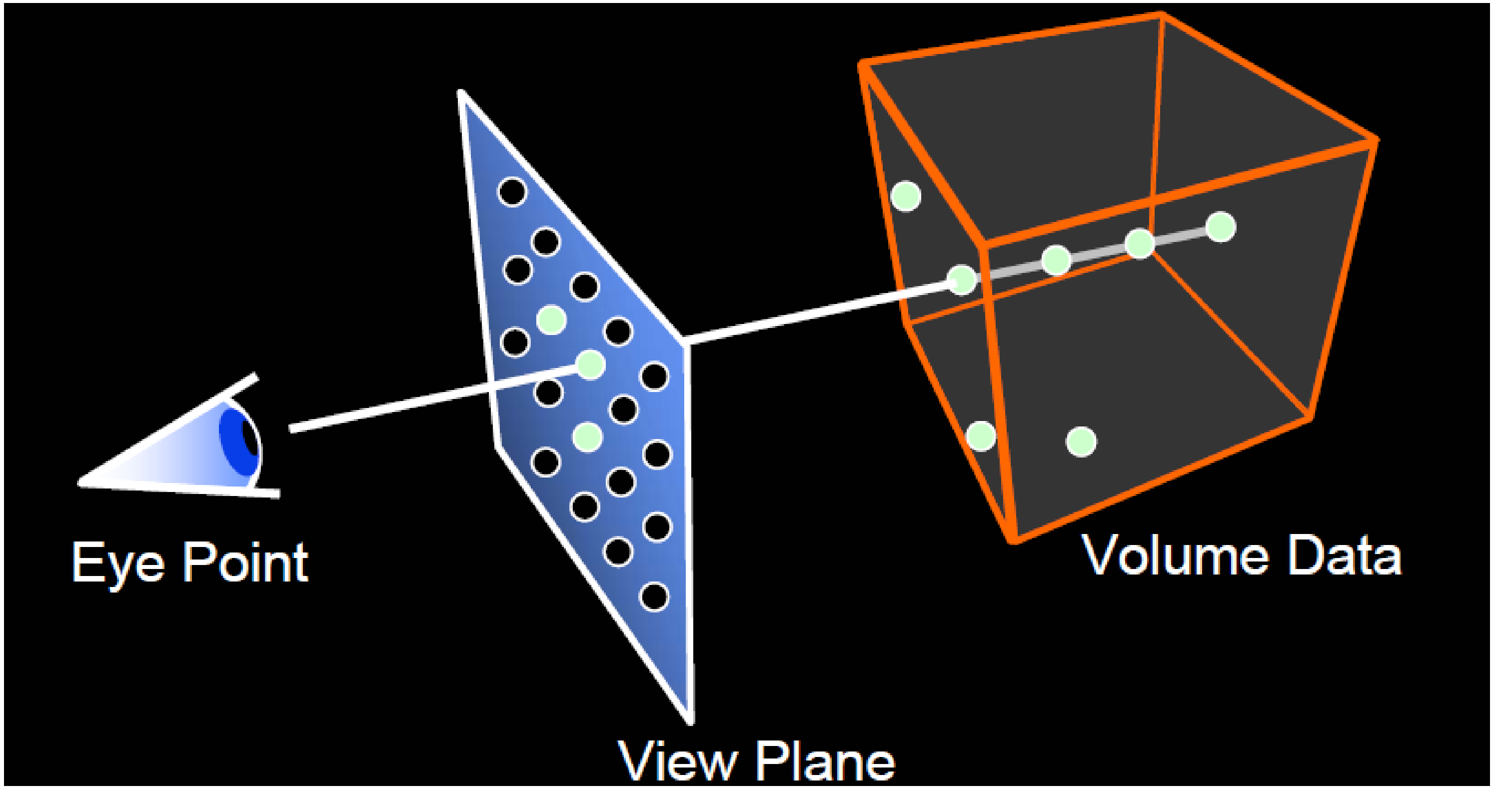
Parallel volume rendering with a single GPU

- In each pass, propagate view rays and store accumulated color and sampling position



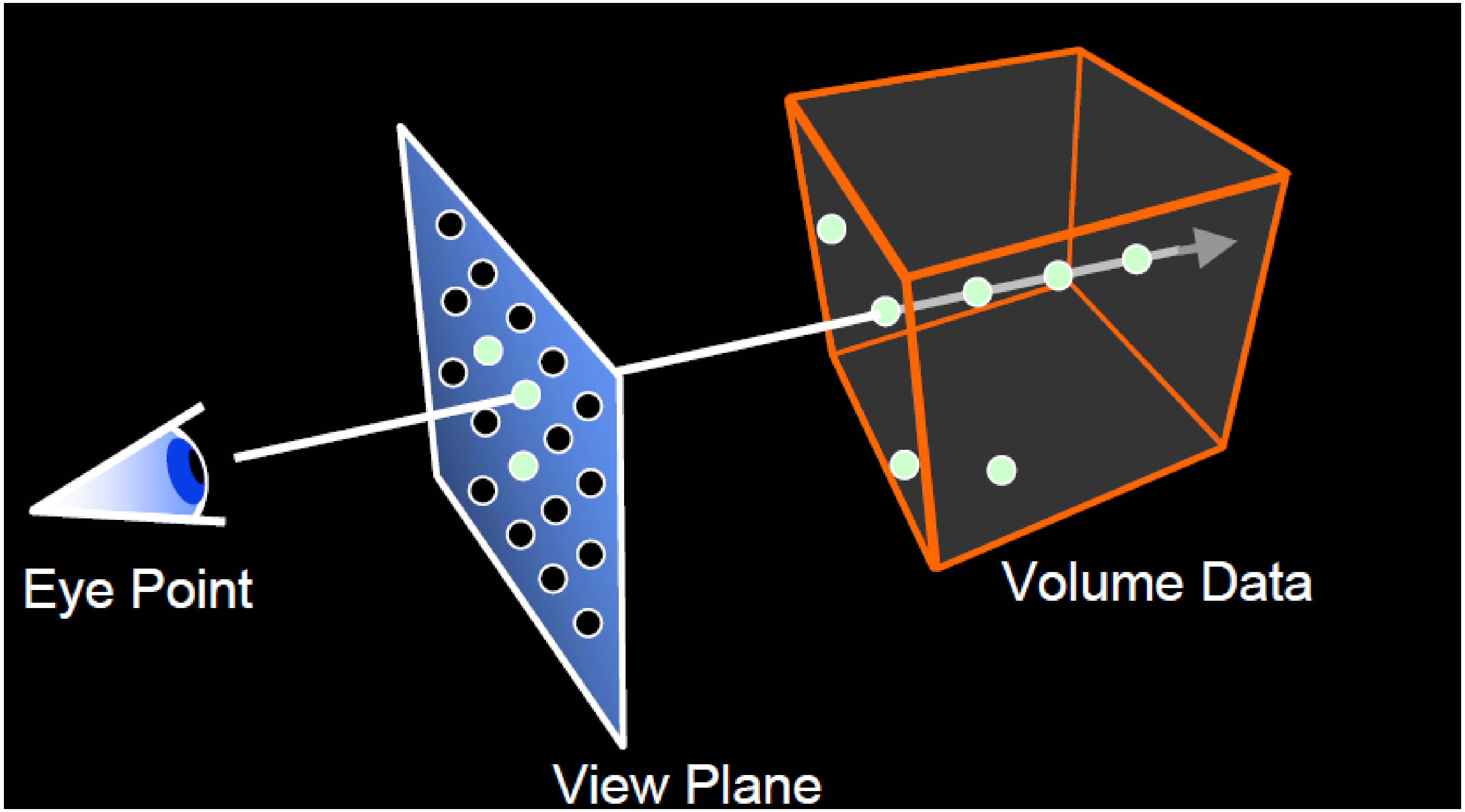
Parallel volume rendering with a single GPU

- In each pass, propagate view rays and store accumulated color and sampling position



Parallel volume rendering with a single GPU

- Stop when all rays have left the volume



Parallel volume rendering with a single GPU. Optimizations

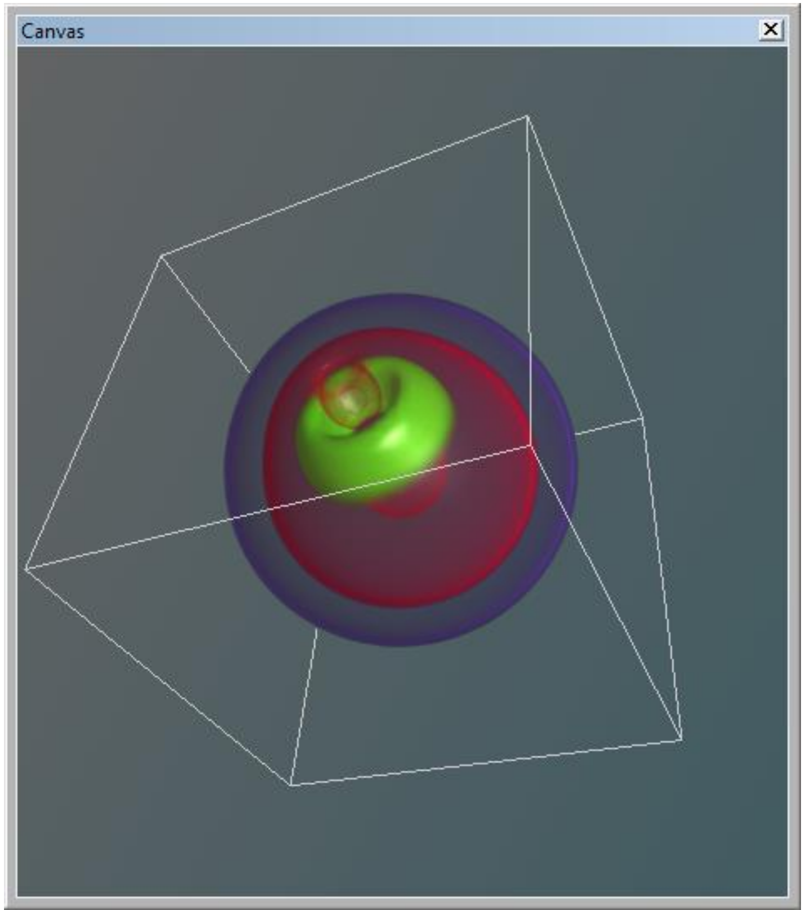
- Avoid work for rays that don’t intersect the volume
- Avoid work for rays that have accumulated full opacity
- Quickly test whether all rays have left the volume
- Quickly go trough empty regions
- Adapt sampling distance to data

Parallel volume rendering with a single GPU. Optimizations

- **Avoid work for rays that don’t intersect the volume**
- Avoid work for rays that have accumulated full opacity
- Quickly test whether all rays have left the volume
- Quickly go trough empty regions
- Adapt sampling distance to data

Avoid rays outside the volume

- Use rasterization to start rays
  - Render bbox & issue fragment shader on valid fragments

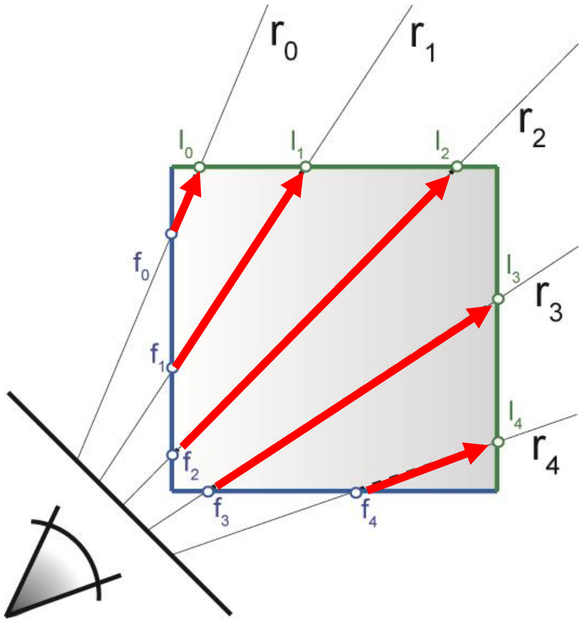


Procedural Ray Setup/Termination

- Everything handled in the fragment shader
- Procedural ray / bounding box intersection

- Ray is given by camera position
  and volume entry position
- Exit criterion needed

- Pro: simple and self-contained
- Con: full load on the fragment shader



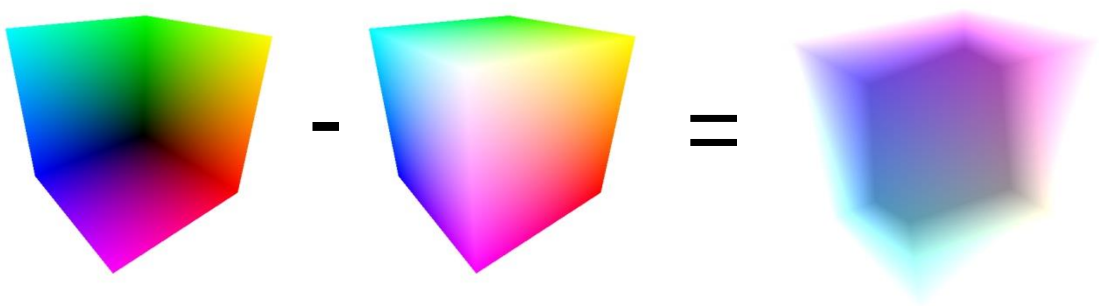
Fragment Shader

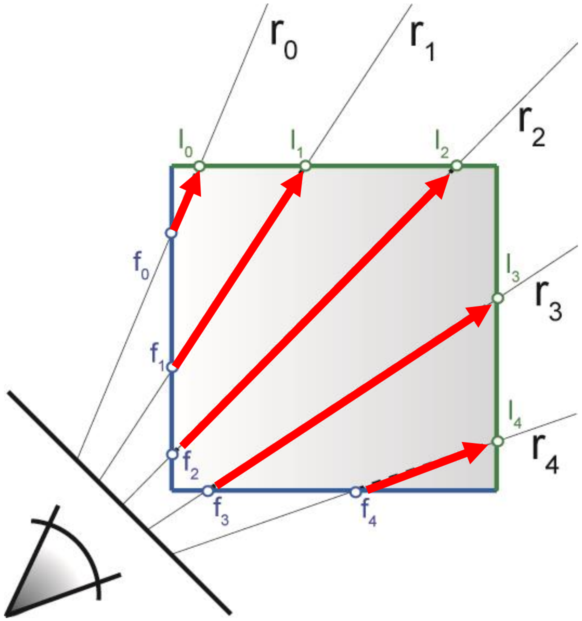
- Rasterize front faces of volume bounding box
- Texcoords are volume position in [0,1]
- Subtract camera position
- Repeatedly check for exit of bounding box

```
// Cg fragment shader code for single-pass ray casting
float4 main(VS_OUTPUT IN, float4 TexCoord0 :  TEXCOORD0,
            uniform sampler3D SamplerDataVolume,
            uniform sampler1D SamplerTransferFunction,
            uniform float3 camera,
            uniform float stepsize,
            uniform float3 volExtentMin,
            uniform float3 volExtentMax
            ) :  COLOR
{
    float4 value;
    float scalar;
    // Initialize accumulated color and opacity
    float4 dst =  float4(0,0,0,0);
    // Determine volume entry position
    float3 position = TexCoord0.xyz;
    // Compute ray direction
    float3 direction = TexCoord0.xyz - camera;
    direction = normalize(direction);
    // Loop for ray traversal
    for (int i = 0; i < 200; i++)  // Some large number
    {
        // Data access to scalar value in 3D volume texture
        value = tex3D(SamplerDataVolume, position);
        scalar = value.a;
        // Apply transfer function
        float4 src = tex1D(SamplerTransferFunction, scalar);
        // Front-to-back compositing
        dst = (1.0-dst.a) * src + dst;
        // Advance ray position along ray direction
        position = position + direction * stepsize;
        // Ray termination:  Test if outside volume ...
        float3 temp1 = sign(position - volExtentMin);
        float3 temp2 = sign(volExtentMax - position);
        float inside = dot(temp1, temp2);
        // ...  and exit loop
        if (inside < 3.0)
            break;
    }
    return dst;
}
```

"Image-Based" Ray Setup/Termination

- Rasterize bounding box front faces and back faces [Krüger and Westermann, 2003]
- Ray start position: front faces
- Direction vector: back−front faces

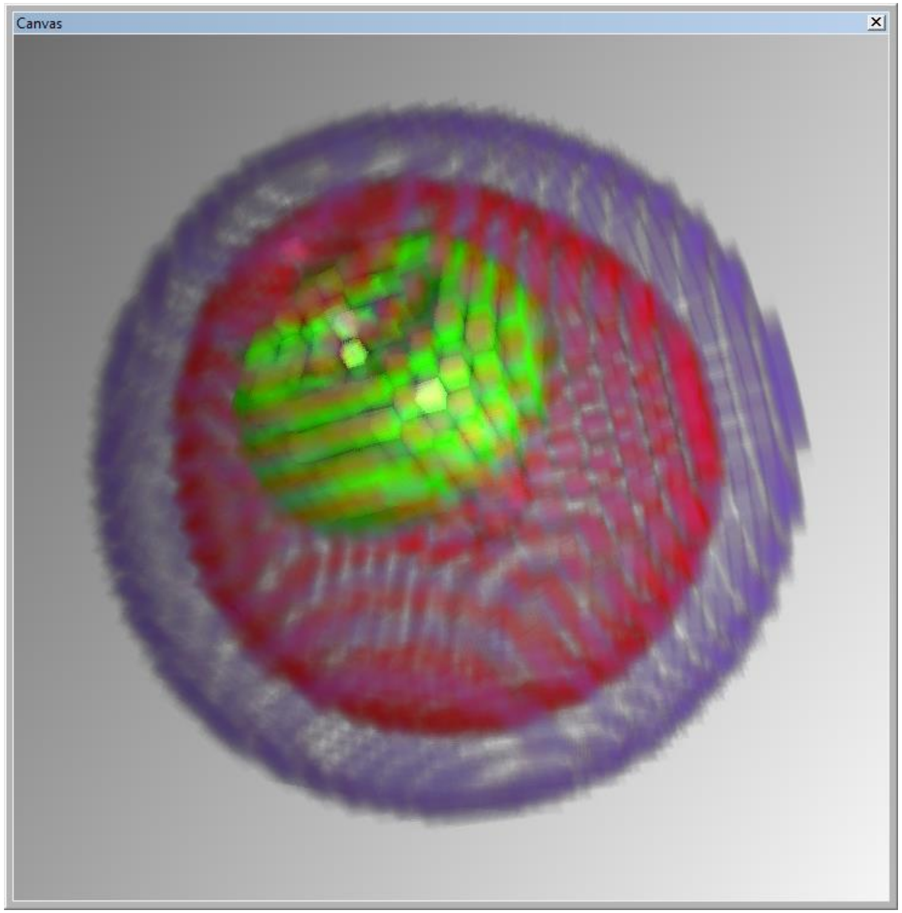


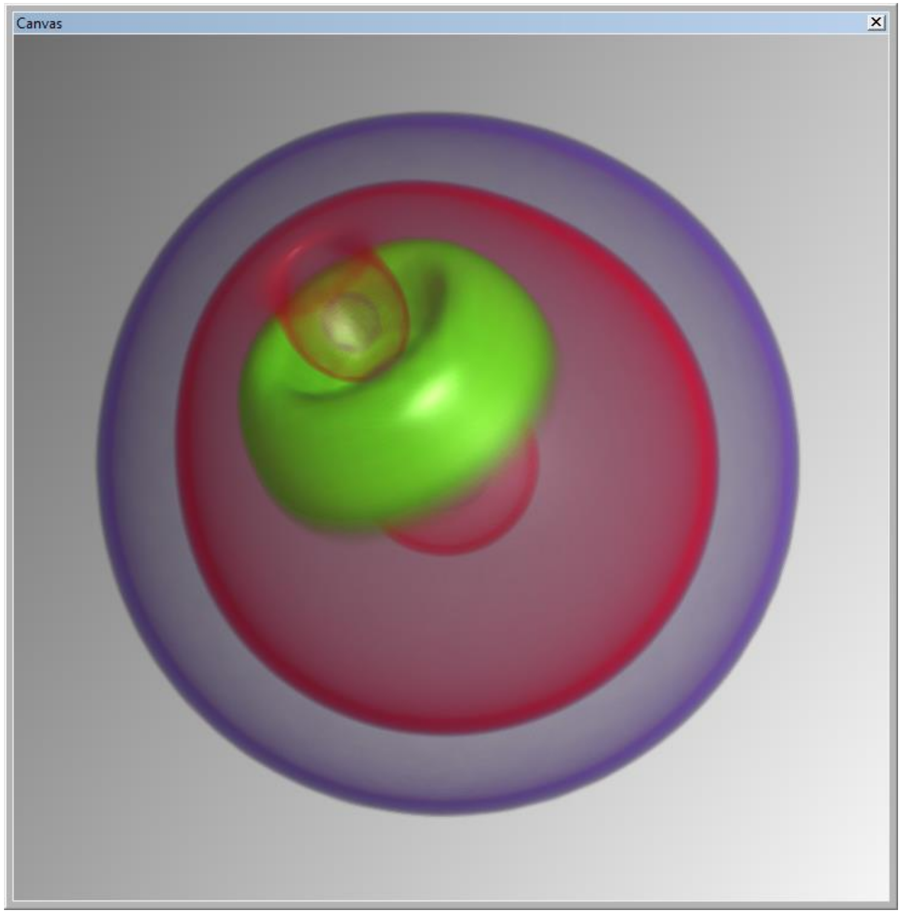


- Independent of projection (orthogonal/perspective)

Is all this perfect?

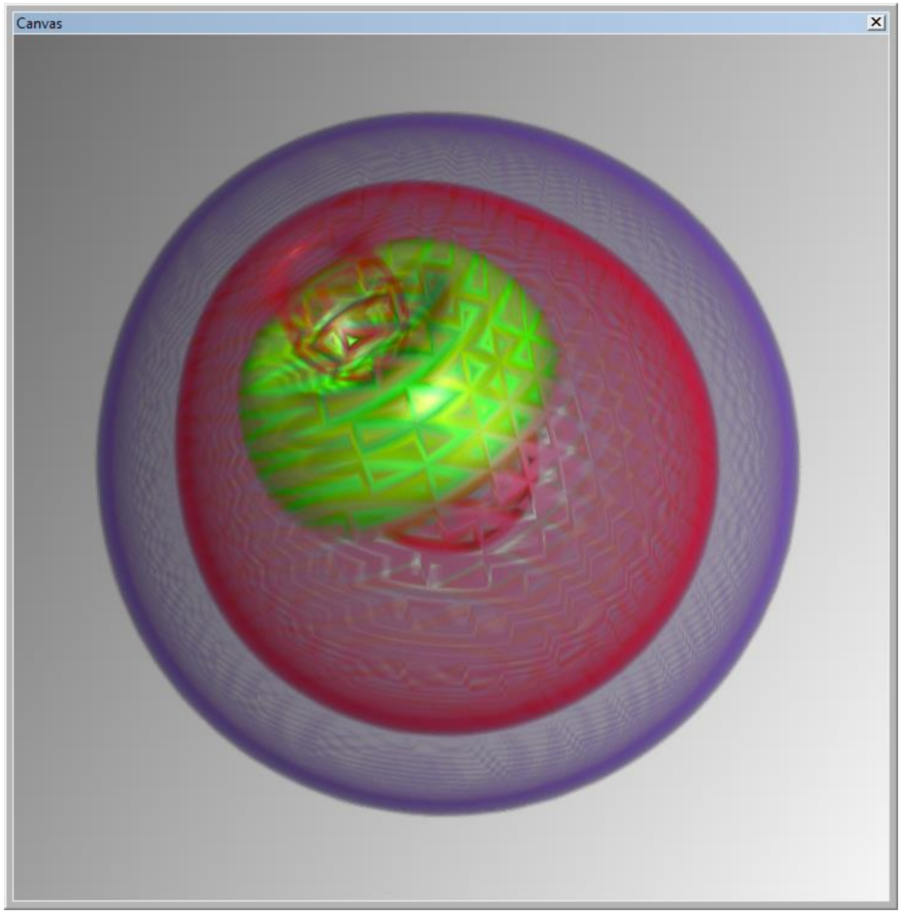
- Need interpolation hardware (or software)

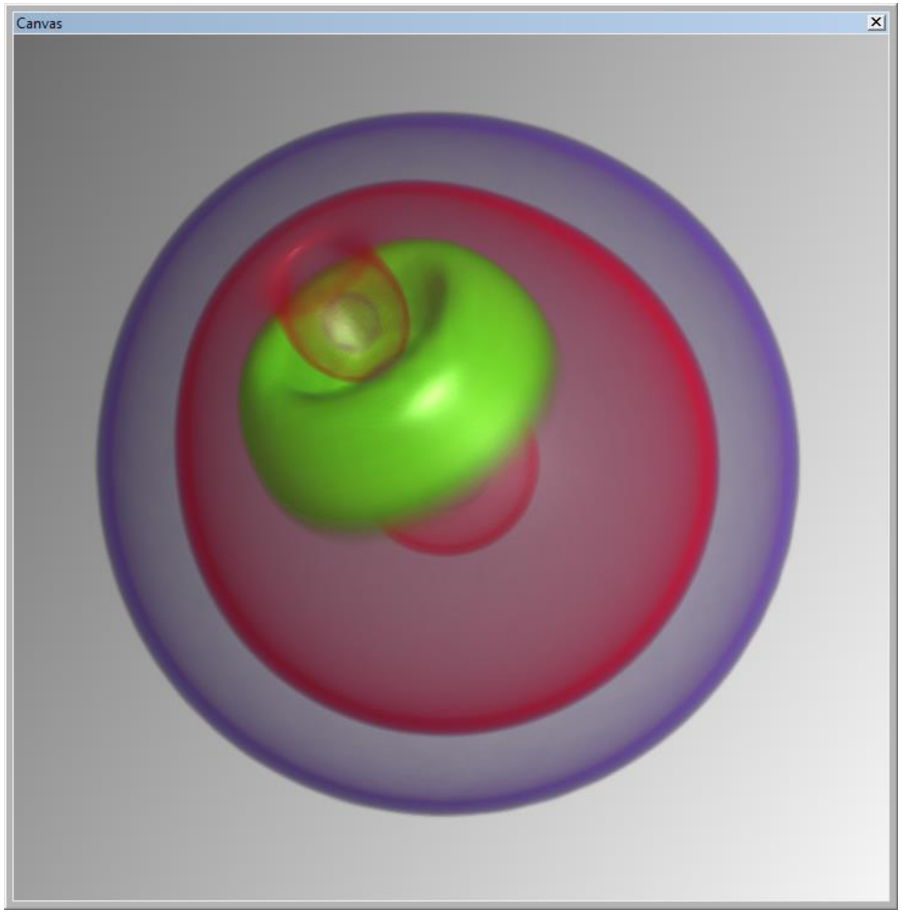




Is all this perfect?

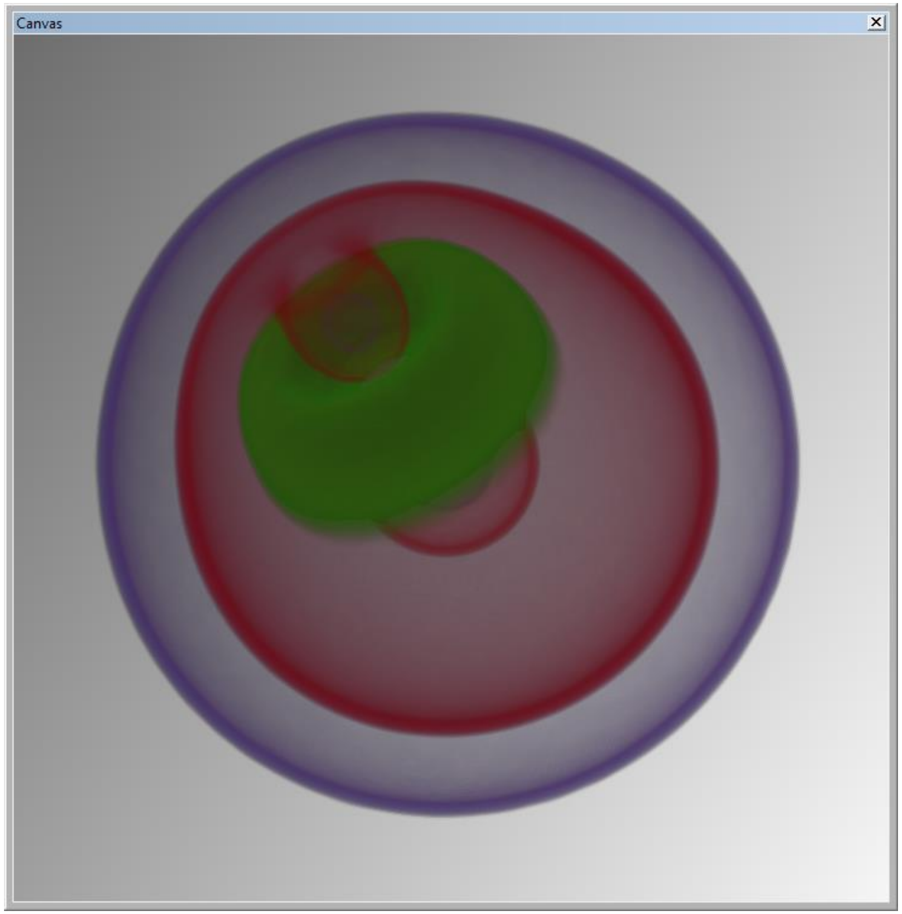
- No, it is not. Sampling rate must be high enough.

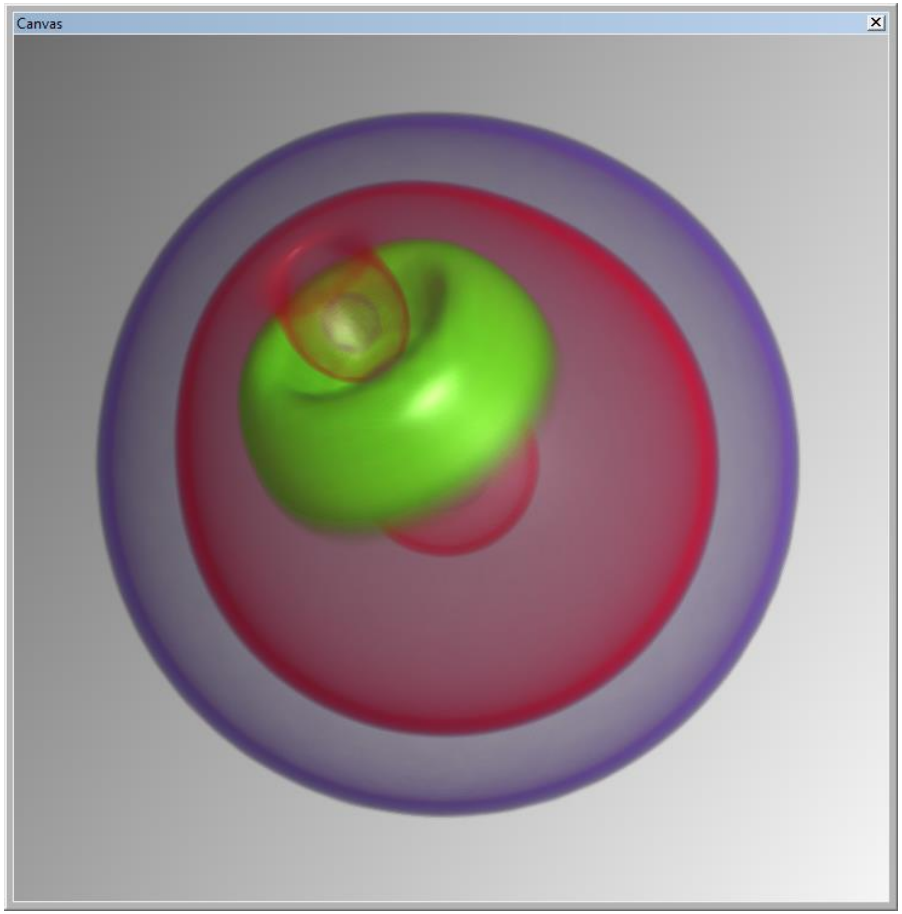




Is all this perfect?

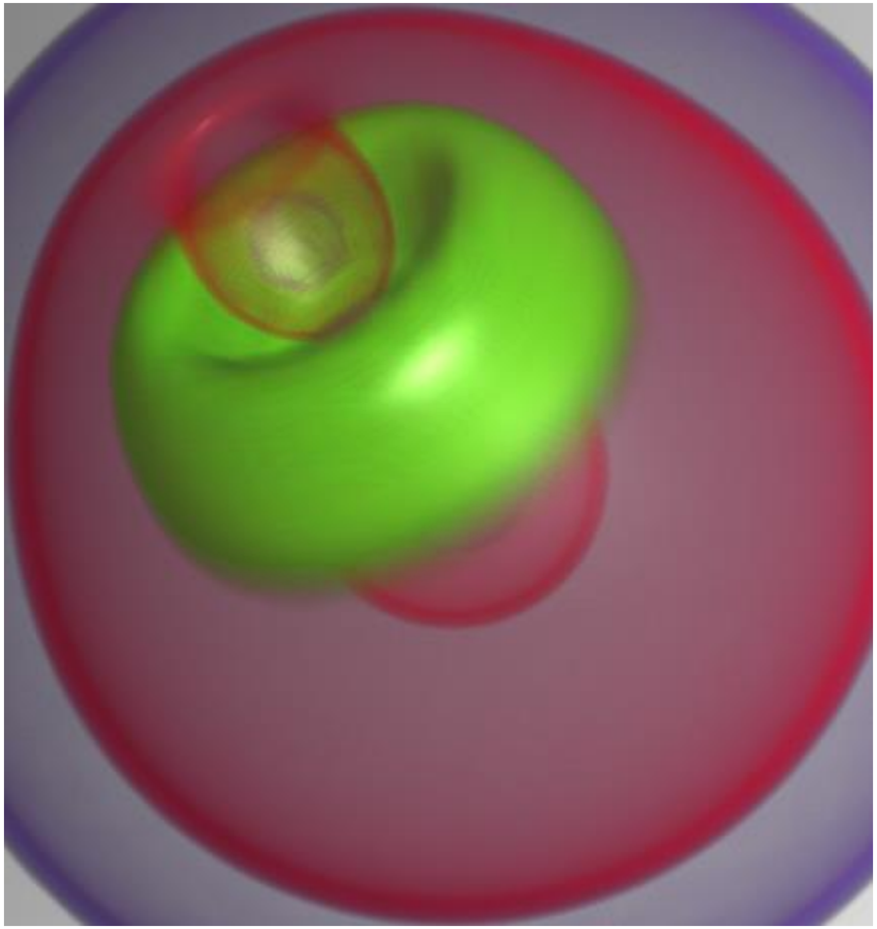
- Need to compute gradients to have a nice lighting solution





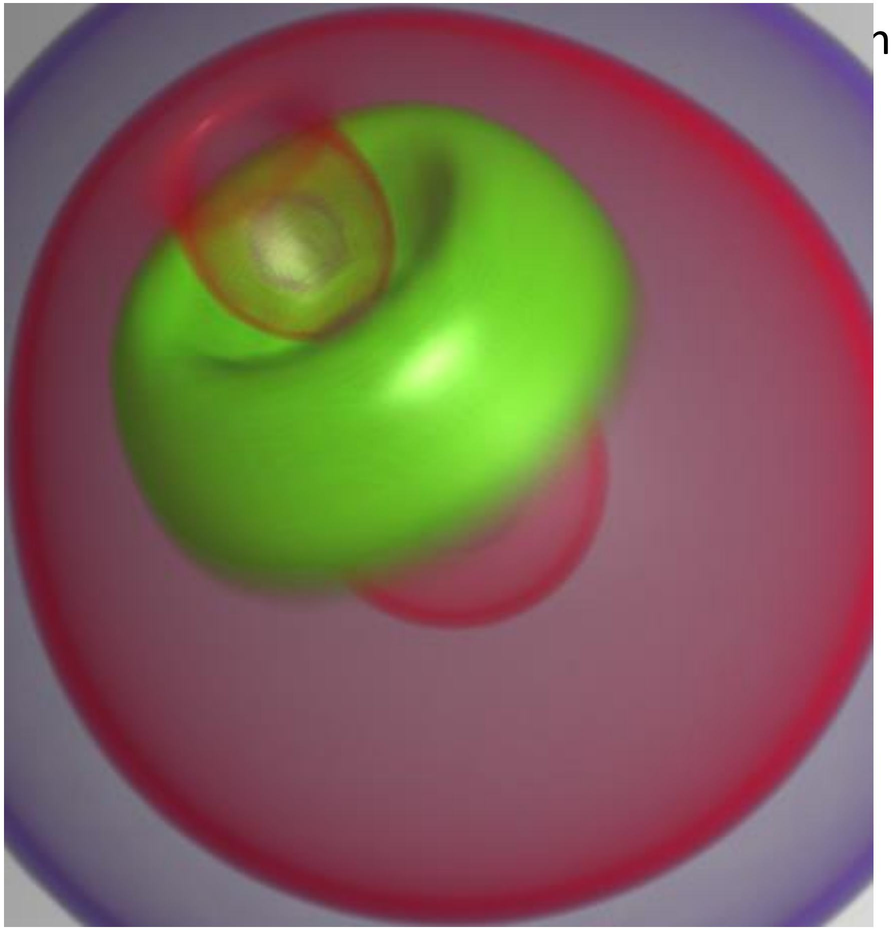
Is all this perfect?

- No, it is not.

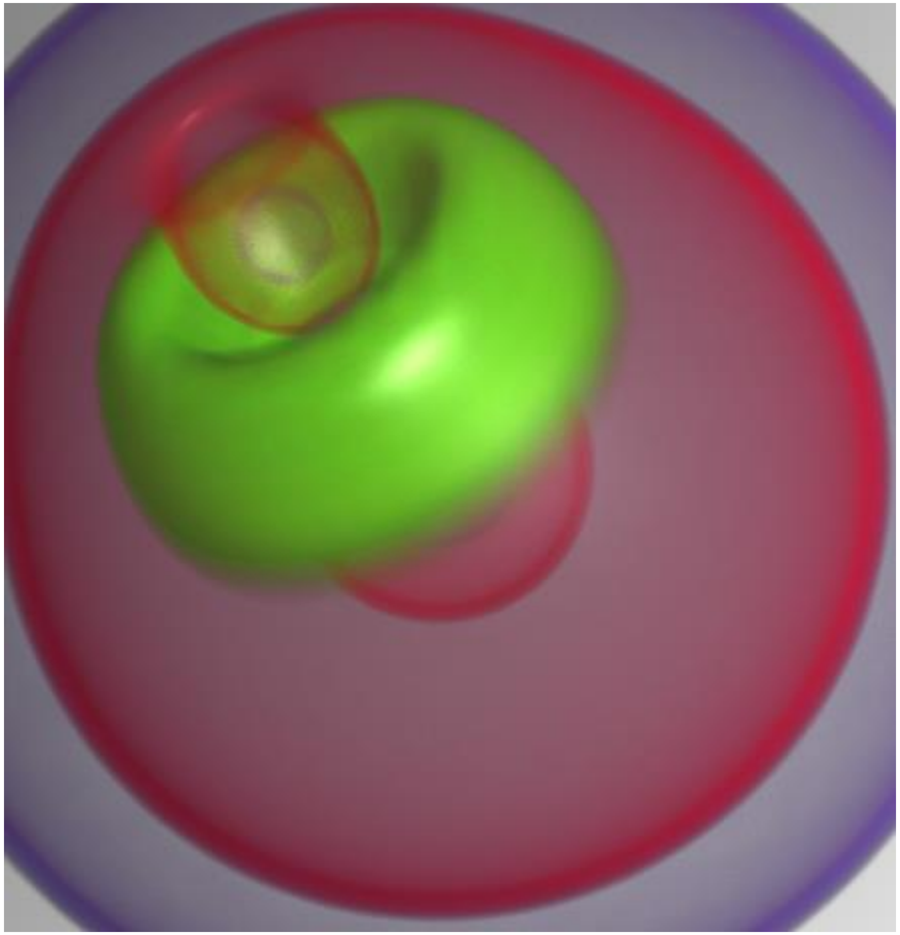


Is all this perfect?

- No, it is not.



ng



Parallel volume rendering with a single GPU. Optimizations

- *Avoid work for rays that don’t intersect the volume*
- **Avoid work for rays that have accumulated full opacity**
- Quickly test whether all rays have left the volume
- Quickly go trough empty regions
- Adapt sampling distance to data

Standard Ray-Casting Optimizations (1)

Early ray termination

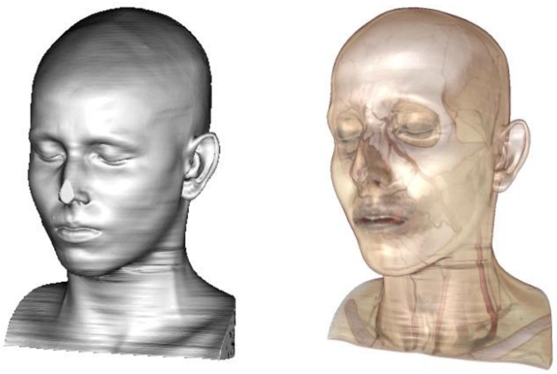
- Isosurfaces: stop when surface hit
- Direct volume rendering:
  stop when opacity >= threshold

- Several possibilities
  - Older GPUs: multi-pass rendering with early-z test
  - Shader model 3: break out of ray-casting loop
  - Current GPUs: early loop exit not optimal but good

Parallel volume rendering with a single GPU. Optimizations

- *Avoid work for rays that don’t intersect the volume*
- *Avoid work for rays that have accumulated full opacity*
- Quickly test whether all rays have left the volume
- **Quickly go trough empty regions**
- Adapt sampling distance to data

Standard Ray-Casting Optimizations (2)



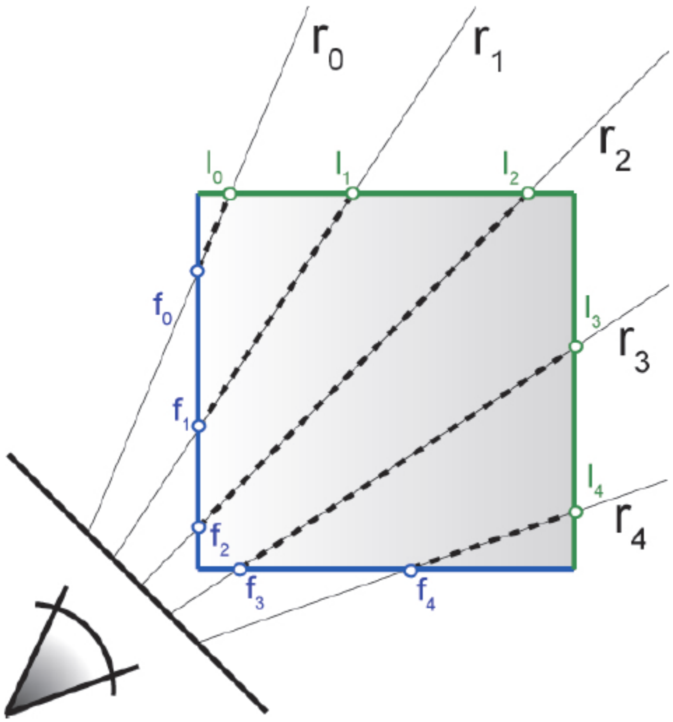
Empty space skipping

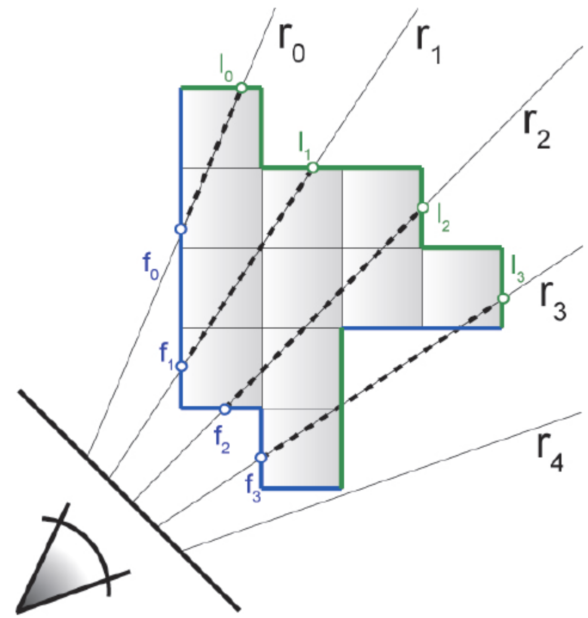
- Skip transparent samples
- Depends on transfer function
- Start casting close to first hit

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

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

Object-Order Empty Space Skipping (1)



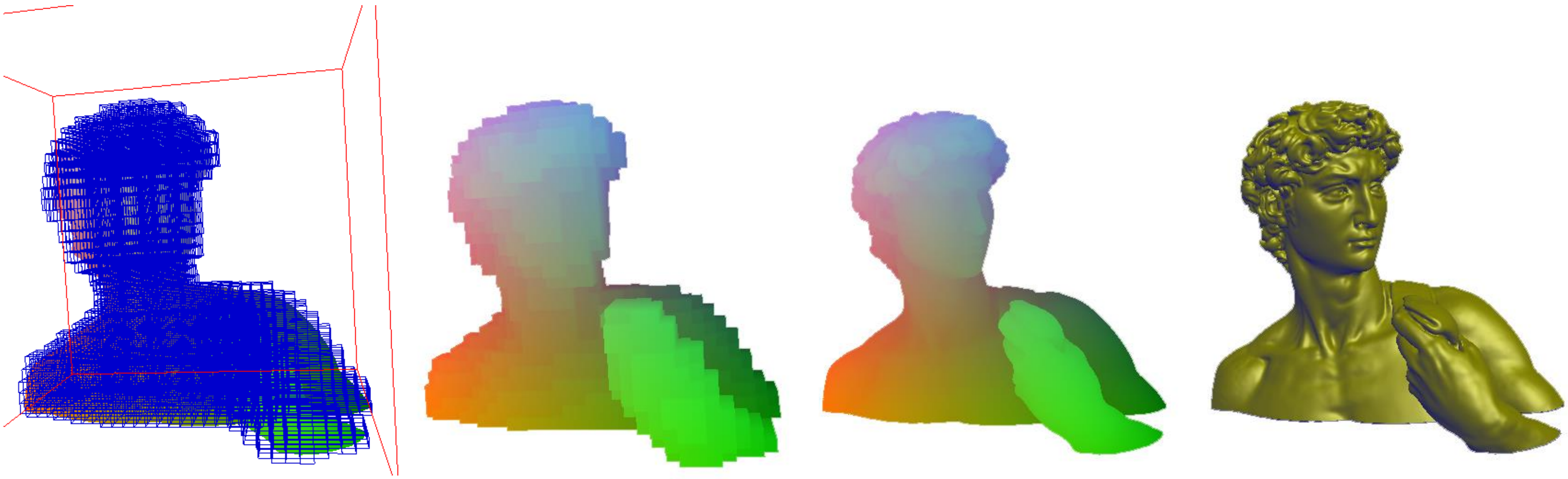


rasterize bounding box

rasterize “tight" bounding geometry

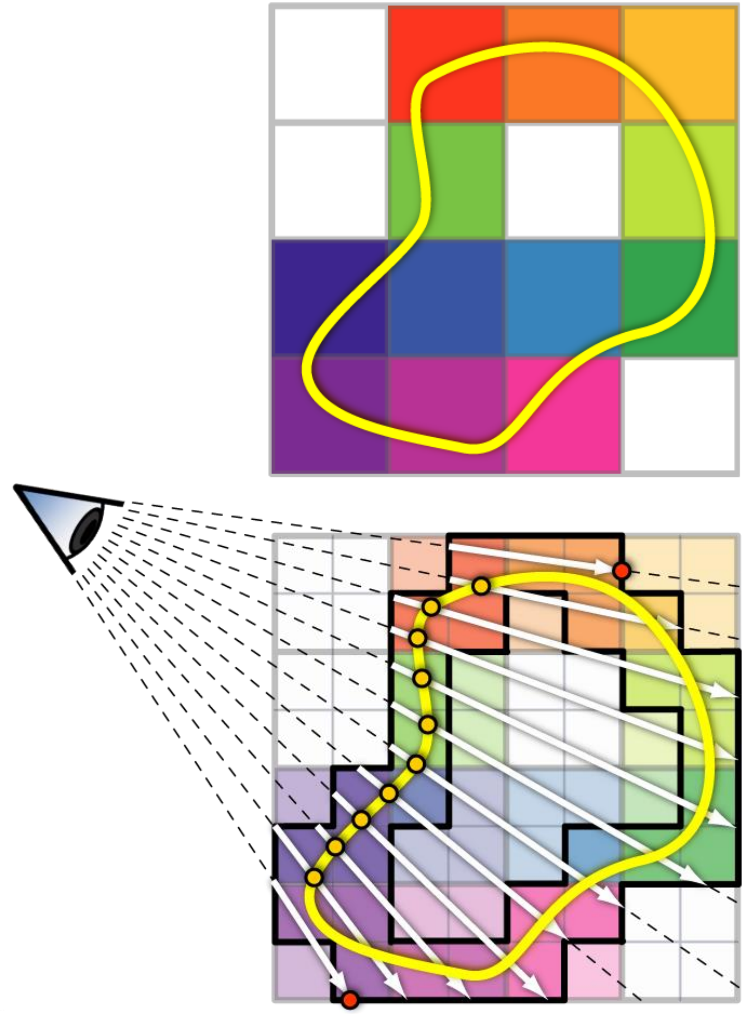
Object-Order Empty Space Skipping (2)

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



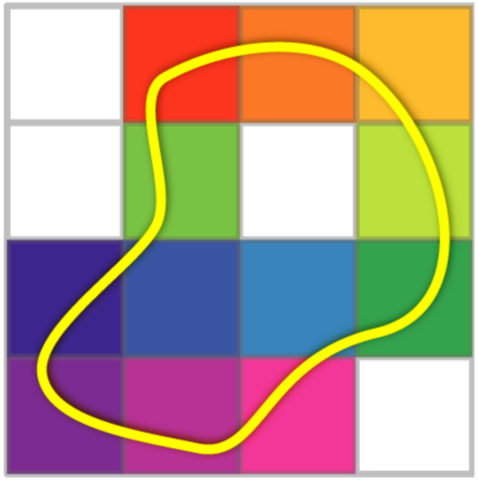
Object-Order Empty Space Skipping (3)

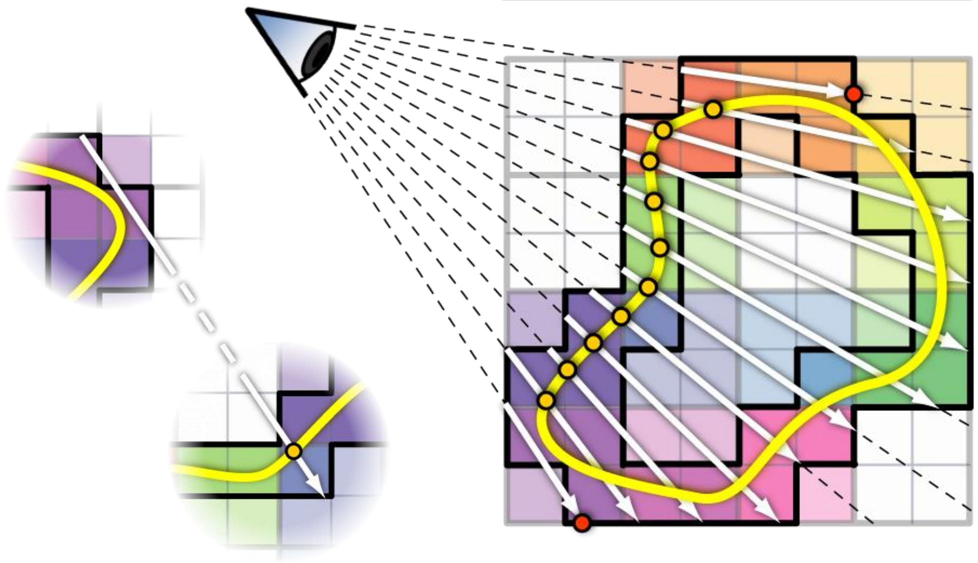
- Rasterize front and back faces of active min-max bricks
- Start rays on brick front faces
- Terminate when
  - Full opacity reached, or
  - Back face reached



Object-Order Empty Space Skipping (3)

- Rasterize front and back faces of active min-max bricks
- Start rays on brick front faces
- Terminate when
  - Full opacity reached, or
  - Back face reached
- Not all empty space is skipped



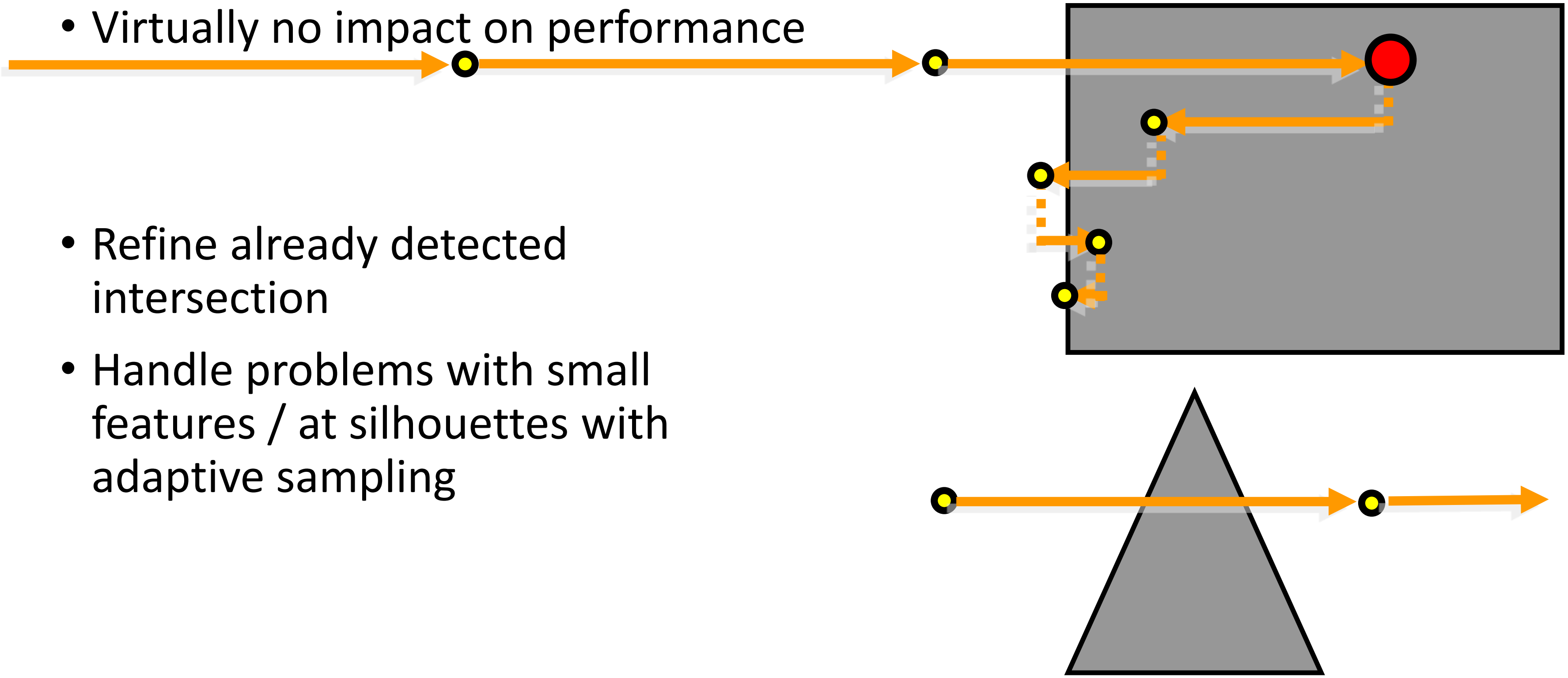


Isosurface Ray-Casting

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

Intersection Refinement (1)

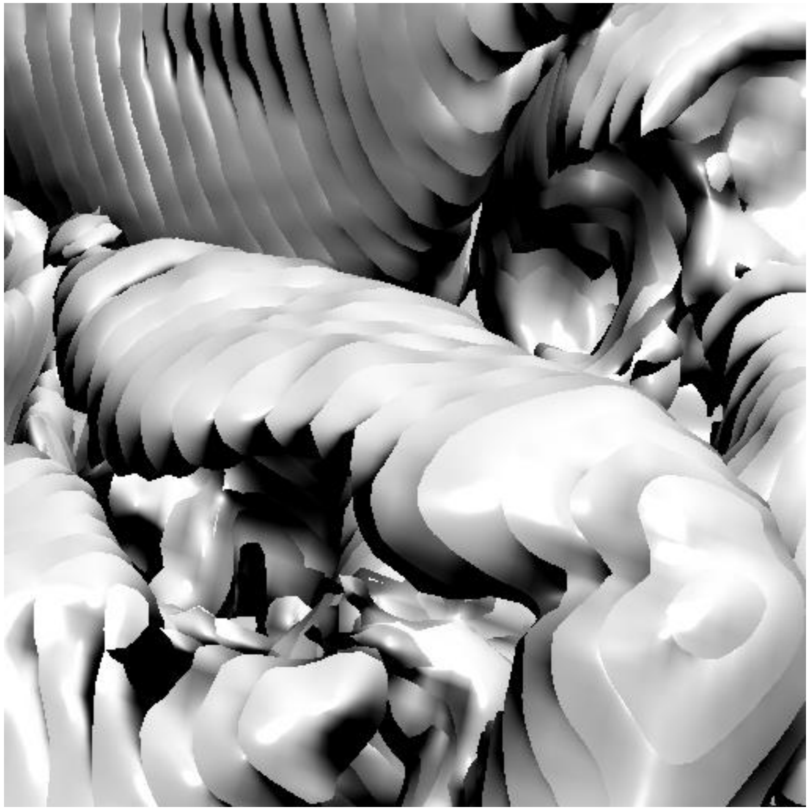
- Fixed number of bisection or binary search steps
- Virtually no impact on performance



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

Intersection Refinement (2)

without refinement



with refinement

sampling rate 1/5 voxel (no adaptive sampling)

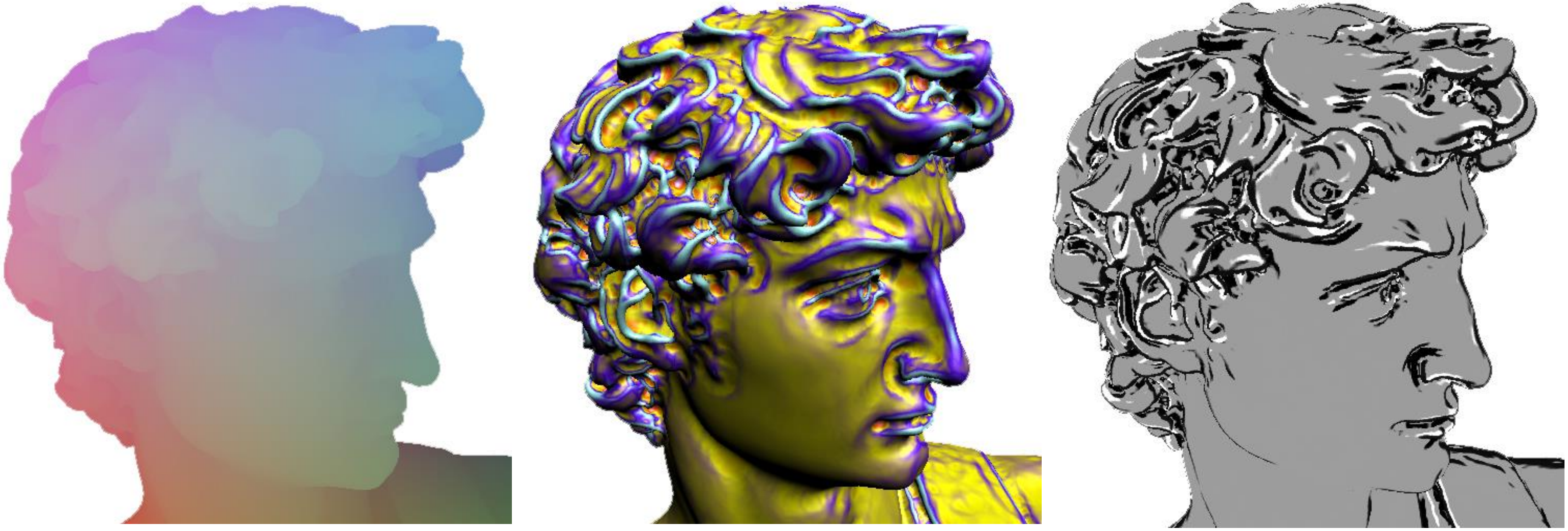
Intersection Refinement (3)

Sampling distance 1.0, 24 fps

Sampling distance 5.0, 66 fps

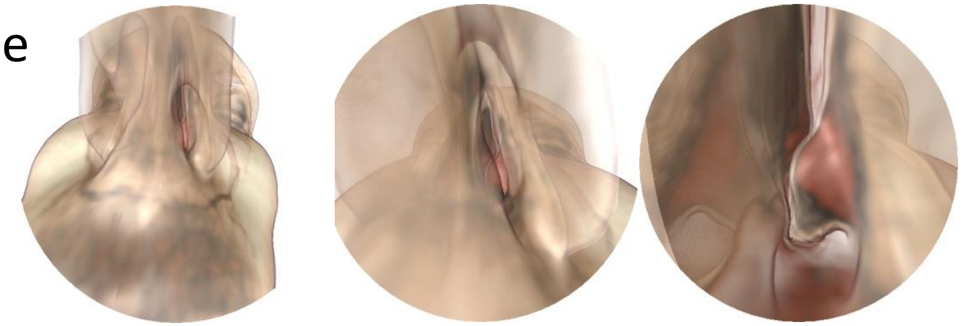
Deferred Isosurface Shading

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

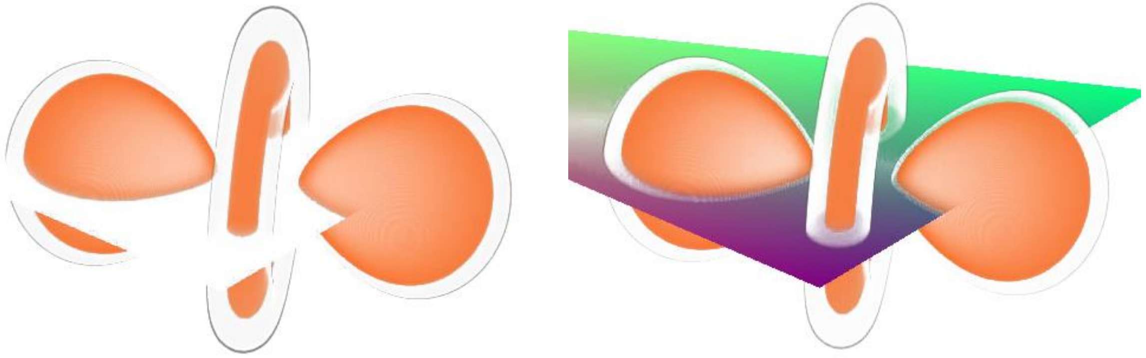


Enhancements (1)

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

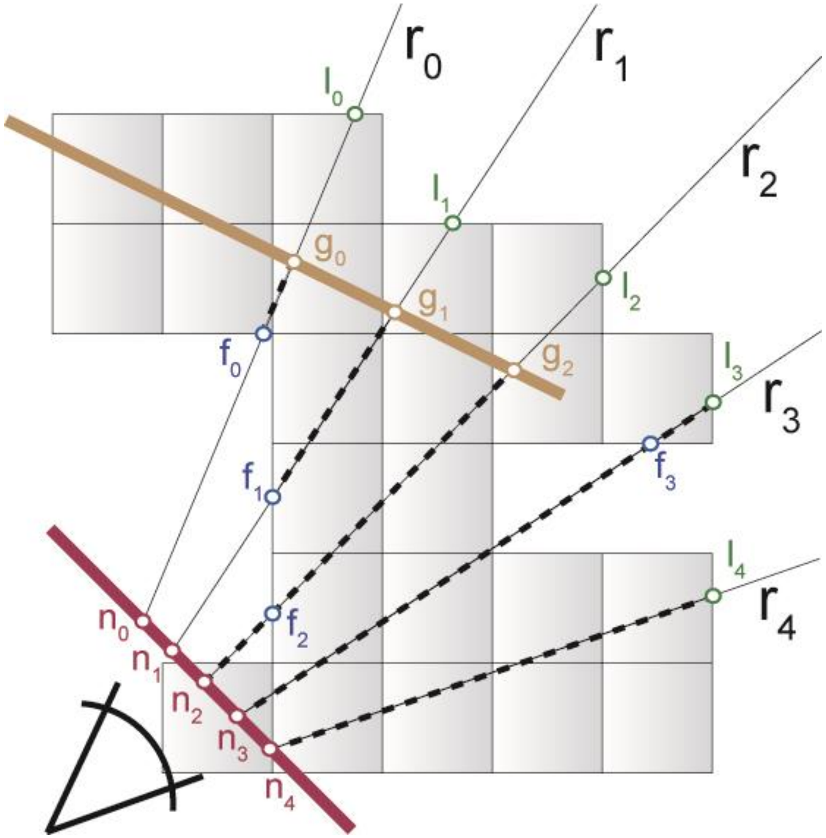


- Intersect polygonal geometry



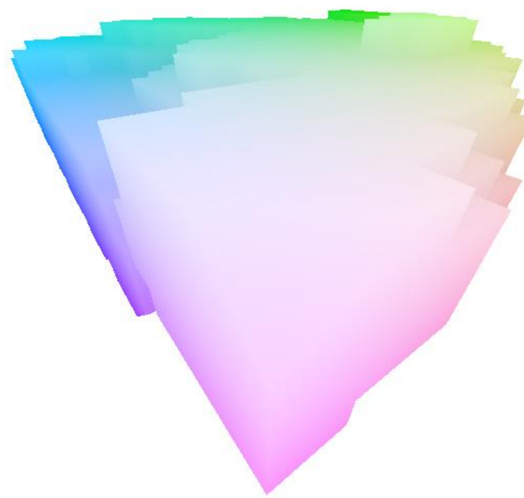
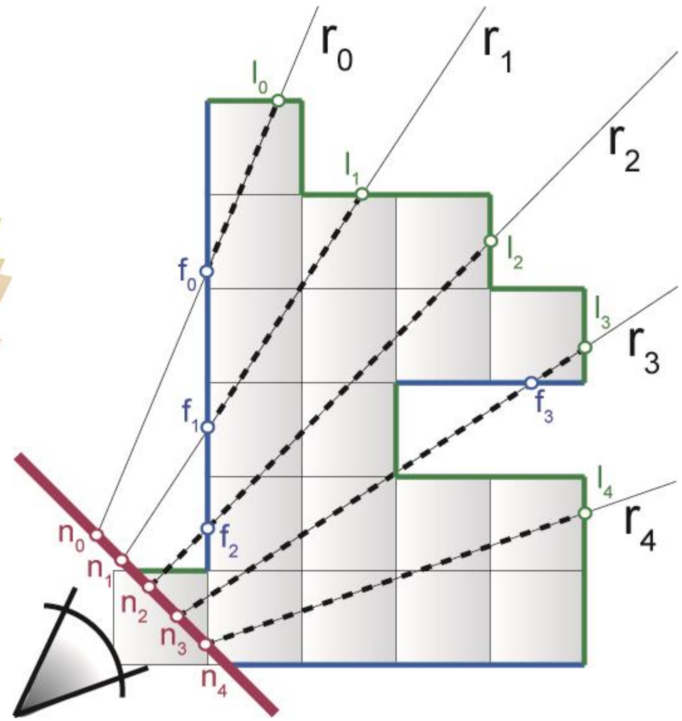
Enhancements (2)

1. Starting position computation
   ⇨ Ray start position image
2. Ray length computation
   ⇨ Ray length image
3. Render polygonal geometry
   ⇨ Modified ray length image
4. Raycasting
   ⇨ Compositing buffer
5. Blending
   ⇨ Final image



Moving Into The Volume (1)

- Near clipping plane clips into front faces

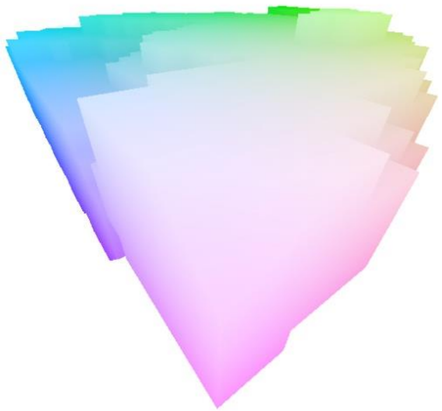
- Fill in holes with near clipping plane
- Can use depth buffer [Scharsach et al., 2006]

Moving Into The Volume (2)

1. Rasterize near clipping plane
   - Disable depth buffer, enable color buffer
   - Rasterize entire near clipping plane

2. Rasterize nearest back faces
   - Enable depth buffer, disable color buffer
   - Rasterize *nearest back faces* of active bricks

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



How to speed-up?

- Algorithmic improvements
  - Many opportunities
- More hardware
  - Clusters of computers
    - Client-server approach
- Better hardware
  - Specialized hardware?

Raycasting pipeline

Generate rays for pixels

While (in volume){

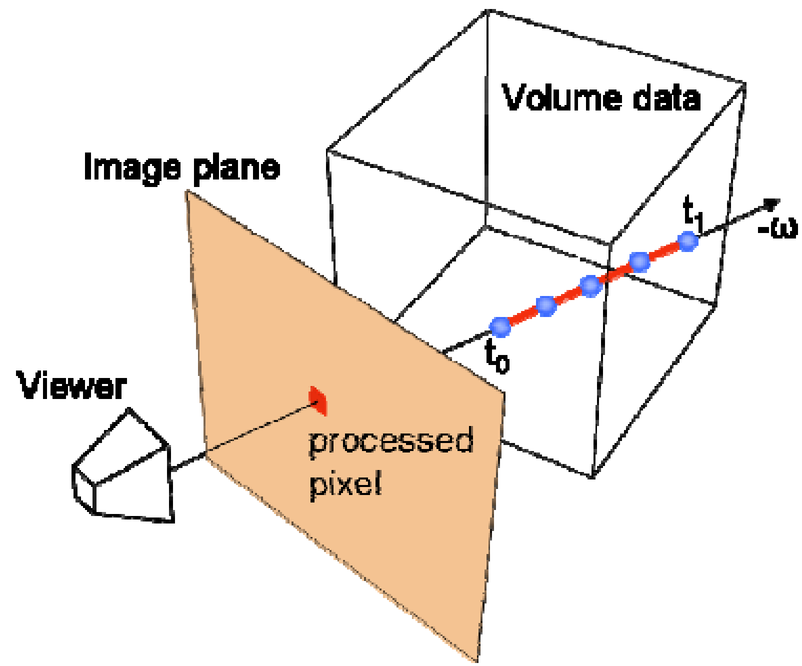
    read data from volume

    map data to color

    accumulate color

}

Write color to framebuffer



Raycasting pipeline

```
Generate rays for pixels
Generate less rays
While (in volume){
        read data from volume
        map data to color
        accumulate color
}
Write color to framebuffer
```

Raycasting pipeline

```
Generate rays for pixels
While (in volume){
        use fewer samples
        read data from volume
        map data to color
        accumulate color
}
Write color to framebuffer
```

Raycasting pipeline

```
Generate rays for pixels
While (in volume){
        read data from volume
        read data faster
        map data to color
        accumulate color
}
Write color to framebuffer
```

Raycasting pipeline

Generate rays for pixels
While (in volume){
    read data from volume
    map data to color
    map less often
    accumulate color
}
Write color to framebuffer

Raycasting pipeline

```
Generate rays for pixels
While (in volume){
        read data from volume
        map data to color
        accumulate color
        accumulate only if needed
}
Write color to framebuffer
```

Generate less rays

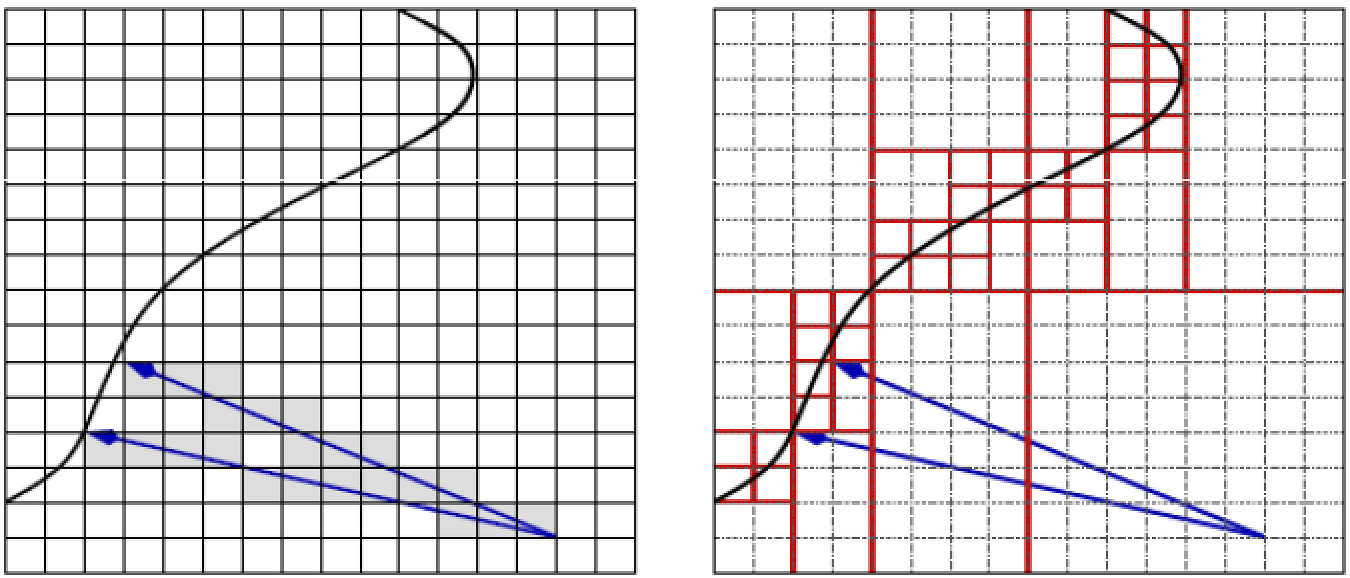
- Do we need one ray per pixel?
  - Often not!
- Shoot rays in interesting regions
  - How to identify them?
  - Adaptive image sampling

Generate less rays

- Tile-based sampling by Ljung
  - Split the image to fixed-size tiles
    - Balance overhead vs. gains
  - Shoot probe rays to evaluate tile importance
    - Pre-classified volume blocks importance
  - Assign tile sampling rate
    - Higher importance -> more samples in tile
  - Not restricted to pixel-aligned samples
    - Arbitrary sampling within the tile

Use fewer samples

- Only about 5-10% of voxels contribute to image
  - Why to traverse all?
  - Depends on transfer function!



Images by [Marmitt 06]

Skip empty regions

- Block-based skipping
  - Encode empty space in additional structure
    - Trade speed for space!
  - Octrees & kD-trees
    - Implicit tree with min/max values in each node
    - Summed-area table for transfer function
    - Both semi-transparent and isosurfaces
    - Size may be a problem
  - Fat blocks
    - Fixed-size block with min/max values
    - Limited effectivity

Read data faster

- Volume swizzling
  - Heavy memory bandwidth requirements
  - Raycasting generates scattered memory access
    - Cache unfriendly
  - Maps neigboring pixels to be close in memory
  - Increases cache efficiency
    - Noticeable speed-up (2-3x)

Map less often

- For expensive shading
  - Complex local lighting models
  - Shadows, Fresnel approximation, ambient occlusion
- Shade only if required
- Skip highly transparent samples
  - Contribution is insignificant
- Skip low gradient magnitude samples
  - Improves more quality than speed

Accumulate only if needed

- Early ray termination
  - Most samples are hidden behind opaque parts
  - Stop the computation if the ray is saturated
  - Empirical threshold = 0.95
  - Trivial implementation

Considerations on GPU-based ray-casting

- Most flexible algorithm
  - Can incorporate speed-up techniques easily
- Balances the requirements on the GPU
  - Can use texture interpolation hardware
  - Can use rasterizers, but **does not depend** on them
- Shader-based vs. CUDA-based implementations
  - On par

Considerations on GPU-based ray-casting

- Transfer function implemented as a texture
  - May require more complex systems for sophisticated transfer functions
- Empty space leaping
  - May require extremely high amounts of storage to make it efficient
- May use multiple-pass algorithms
  - E.g. for deferred shading

Acknowledgments

- Built from slides by Henning Scharsach, Christian Sigg, Daniel Weiskopf, Martin Kraus, Lukas Marsalek, Timo Ropinski, Christof Rezk-Salama, Klaus Engel, Markus Hadwiger... Thanks to all of them!!!
- Check www.real-time-volume-graphics.org for more materials