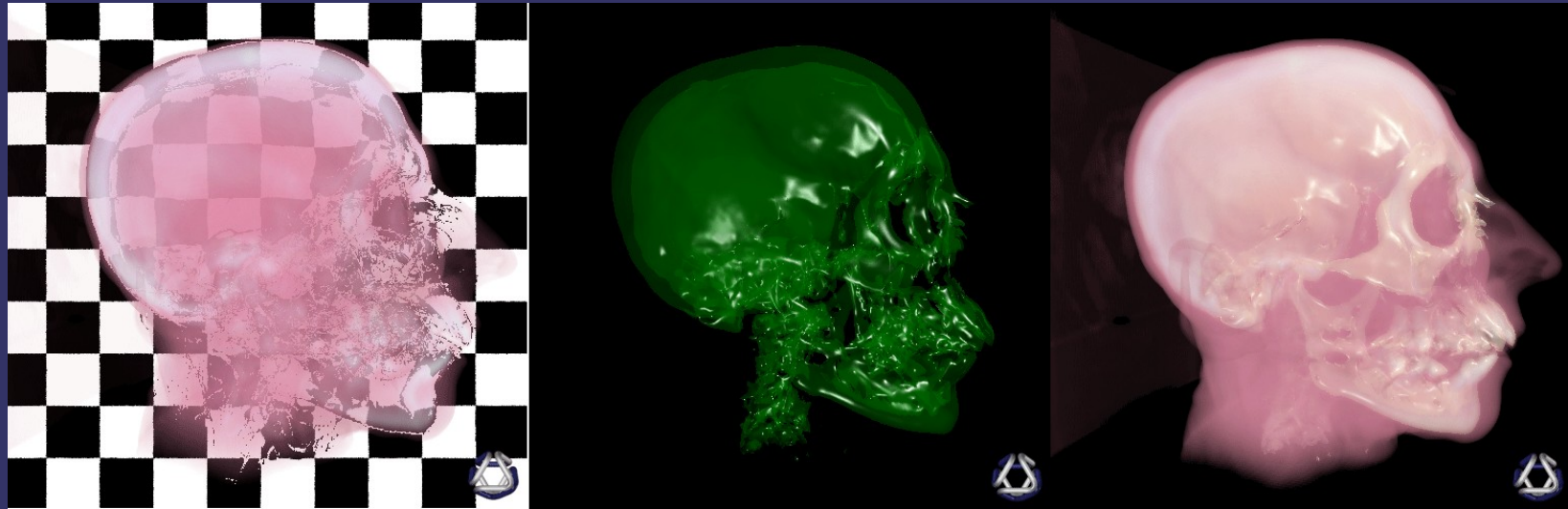


# *A volume rendering framework for GPU-based raycasting*

*by*

*Simon Stegmaier, Magnus Strengert, Thomas Klein  
University of Stuttgart*



**Andrea Sottoriva**

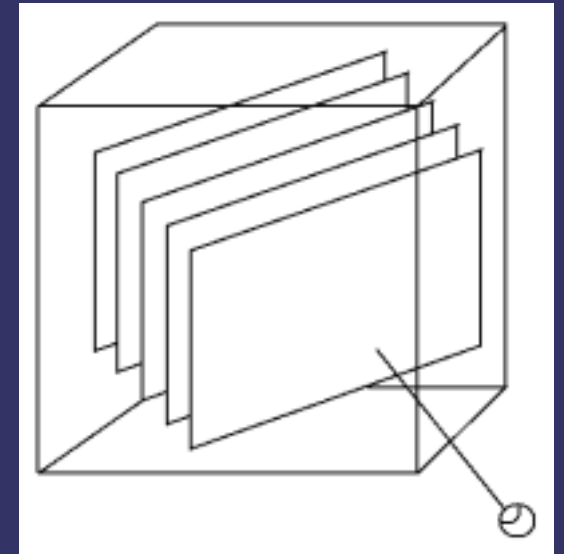
Scientific Visualization and Virtual Reality  
*Universiteit van Amsterdam*

2 Oct 2006

# *The classical approach: slice-based volume rendering*

Basic algorithm:

1. Sample a certain number of slices of data perpendicularly to the viewing direction
2. Render each slice to a quad as a 2D blended texture
3. Tune colors and alpha-blending values following a selected transfer function and let the OpenGL pipeline to do the nasty work



This is one of the most common volume rendering techniques and with the actual graphics hardware is fast but has a lot of obvious limitations that will be examined subsequently.

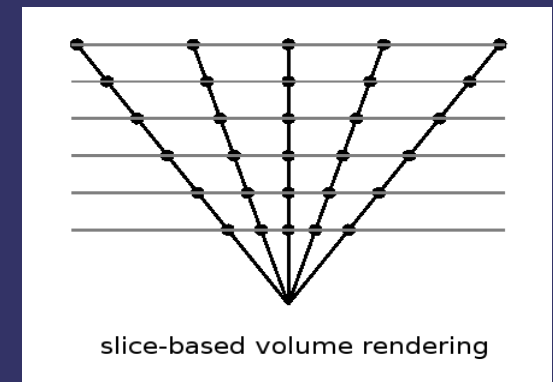
# *The classical approach: slice-based volume rendering*

## PROs

- ➔ It's a quite simple technic
- ➔ It's fast
- ➔ It permits a partial use of the graphics hardware for ray integration (alpha-blending)

## CONTROs

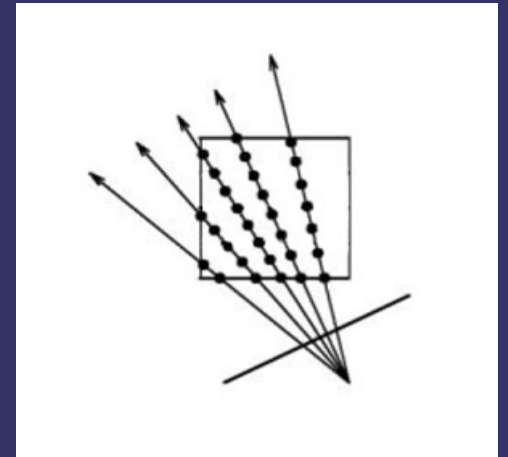
- ➔ Textures need to be recomputed if the camera is moved
- ➔ Artifacts due to the rasterization
- ➔ Artifacts due to the non-uniform integration step
- ➔ Low flexibility (hard to optimize and to extend)
- ➔ Volume light refraction is very hard to implement



# *The optimal approach: raycasting*

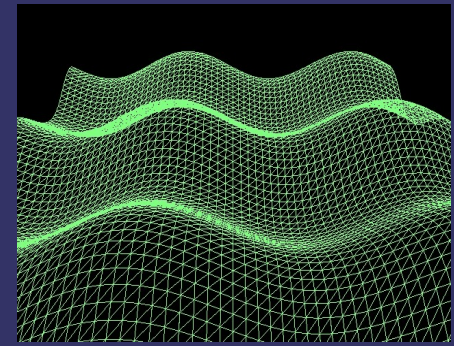
Basic algorithm:

1. Throw a beam of rays from the viewport to the volume following the perspective projection
2. For each ray integrate the volume along its direction applying the chosen integration technic
3. Eventually compute some additional graphic effect (example refraction)
4. Display the pixel color as function of the integral result



This technic offers an optimal solution for volume rendering visualizations but has usually to be implemented in-software, imposing an enormous amount of overhead (no real-time rendering).

# ***A solution: GPU-based raycasting***



Today all the new graphics card support the GPU programming specified by OpenGL 1.5 with NV\_fragment\_program and NV\_vertex\_program (nVidia Cg) and by OpenGL 2.0 with GLSL.

Briefly this technology provides a development environment for 3D computer graphics with the following features:

- ➔ Simplified C-style syntax language (pointers, dynamic memory allocation, I/O or multipurpose libraries are not allowed)
- ➔ Vertex/Fragment shader pipeline programming
- ➔ Large built-in math library
- ➔ Flow-control (allowing single-step rendering technics)
- ➔ Run-time shaders loading / unloading / compilation / parameters-providing from the application
- ➔ Total access to the OpenGL pipeline state via built-in variables
- ➔ Complex embedded types such as vectors, matrices and textures multidimensional arrays

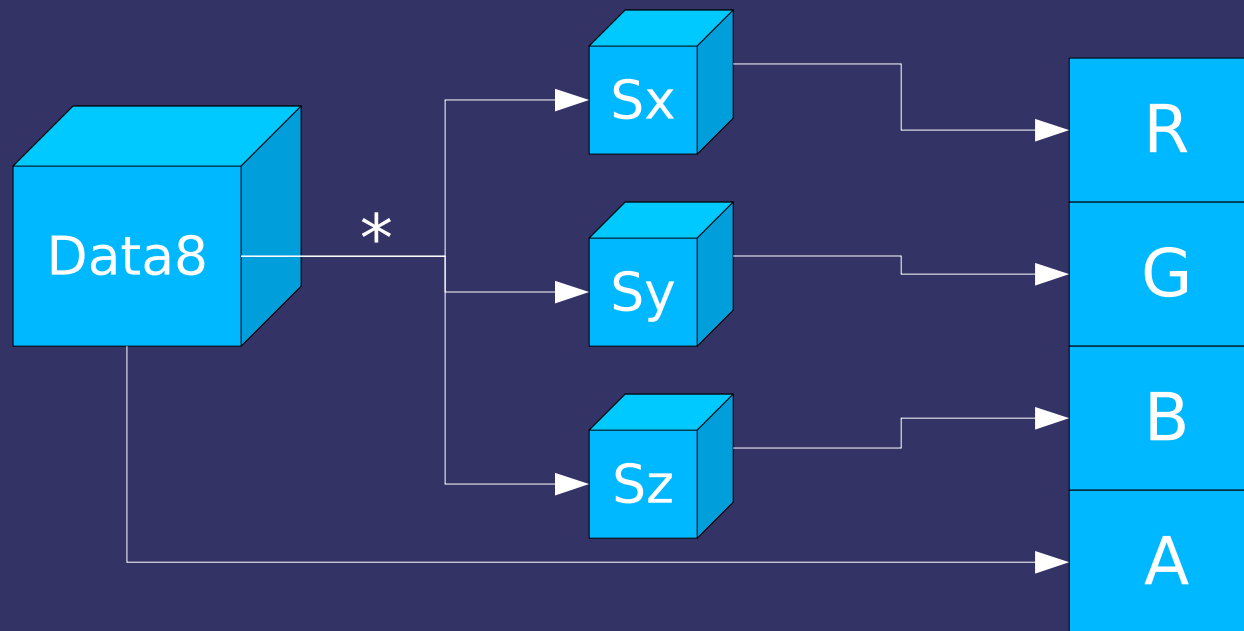
# ***A simple and flexible volume rendering framework for graphics-hardware-based raycasting***

A in-hardware raycasting framework developed by Simon Stegmaier, Magnus Strengert and Thomas Klein from the University of Stuttgart which provides:

- ➔ A set of shaders based on NV\_fragment\_program and NV\_vertex\_program ARB extensions (from OpenGL 1.5) written in the typical assembly-like language
- ➔ Flexible library for loading and unloading of shaders based on OpenGL/GLUT written in C
- ➔ Single step volume raycasting, volume clipping and isosurface visualizations
- ➔ Various kinds of additional graphics technics used in combination with the previously cited visualizations (shadows, sphere-mapping, refraction with Snell's law ecc...)

# Hardware based raycasting: initialization procedure

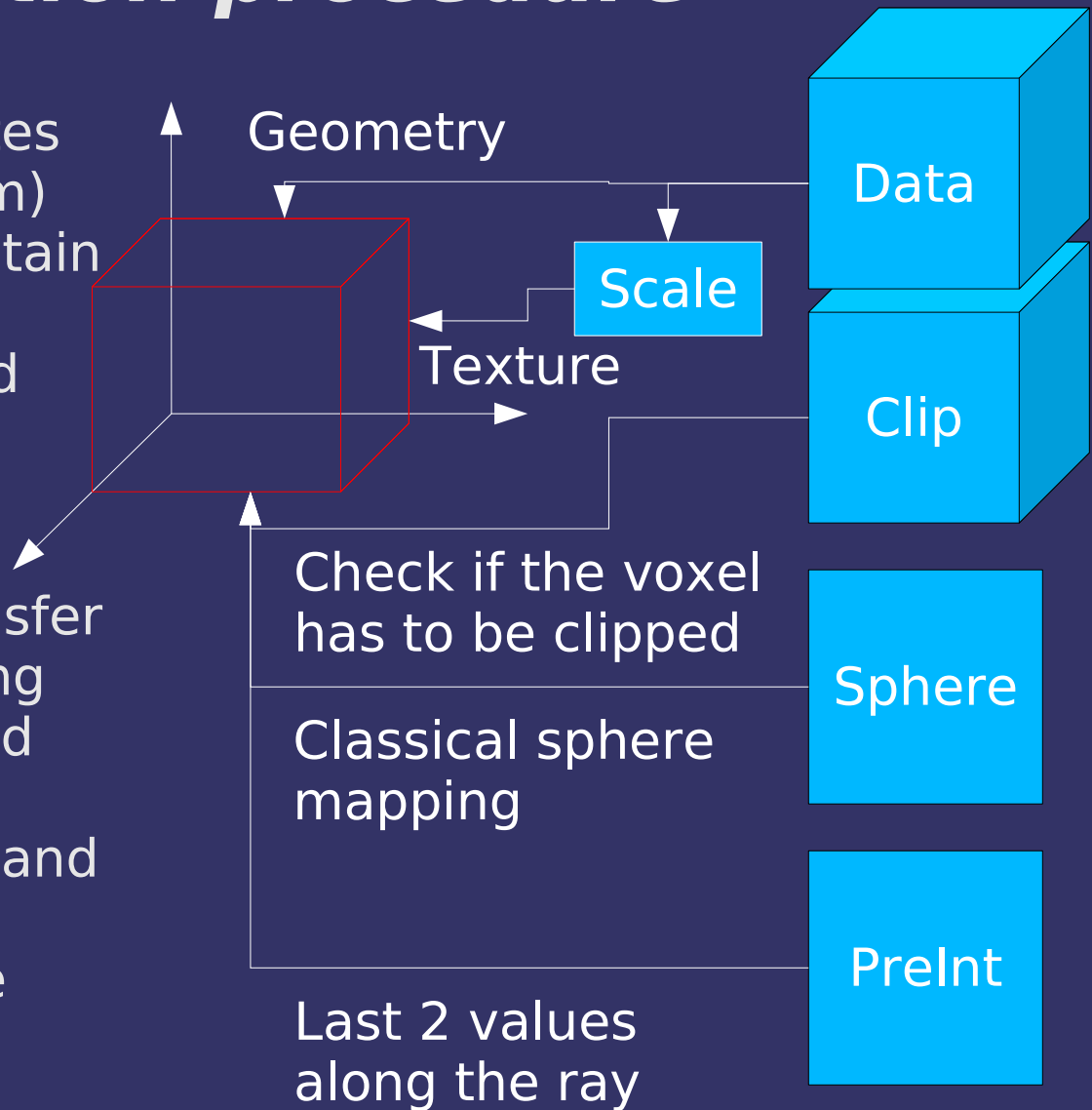
- ➔ Data are expected as *unsigned char*
- ➔ Values are stored inside the alpha component of a 3D RGBA texture
- ➔ The gradient is precalculated using a Sobel operator and stored inside the RGB components of the texture



- ➔ The volume extents are computed and normalized
- ➔ One corner of the volume bounding box is placed in the origin

# Hardware based raycasting: initialization procedure

- ➔ Assign data texture coordinates as geometry (not av. on fragm)
- ➔ Compute a Scale vector to obtain real data texture coordinates
- ➔ Eventually assign clipping and spheremapping coordinates
- ➔ A 2D pre-integration table is computed according to a transfer function to weight the blending value (approximated to a fixed step)
- ➔ The data, the pre-integration and eventually the other textures coordinates are passed to the fragment shader using the ARB\_multitexture extension



# *Hardware based raycasting: the fragment shader*

Every fragment (triangle) of which the bounding volume is made is processed by the fragment shader as shown in the following pseudocode:

```
geom_pos = data_texcoord
real_texcoord = geom_pos * scale_factor
camera_pos = invers(modelview).row[3]
ray_direction = (geom_pos - camera_pos).normalize()
last_voxel = data_texture[real_texcoord]
fragment_color = 0.0f
cur_pos = geom_pos + step * ray_direction

do {
    new_voxel = data_texture[cur_pos]
    src_color = preint_texture[last_voxel][new_voxel]
    fragment_color = PerformBlend(src_color, fragment_color)
    cur_pos = cur_pos + step * ray_direction
    last_voxel = new_voxel
} while (cur_pos is inside the volume)
```

**Background\_Raycasting()**

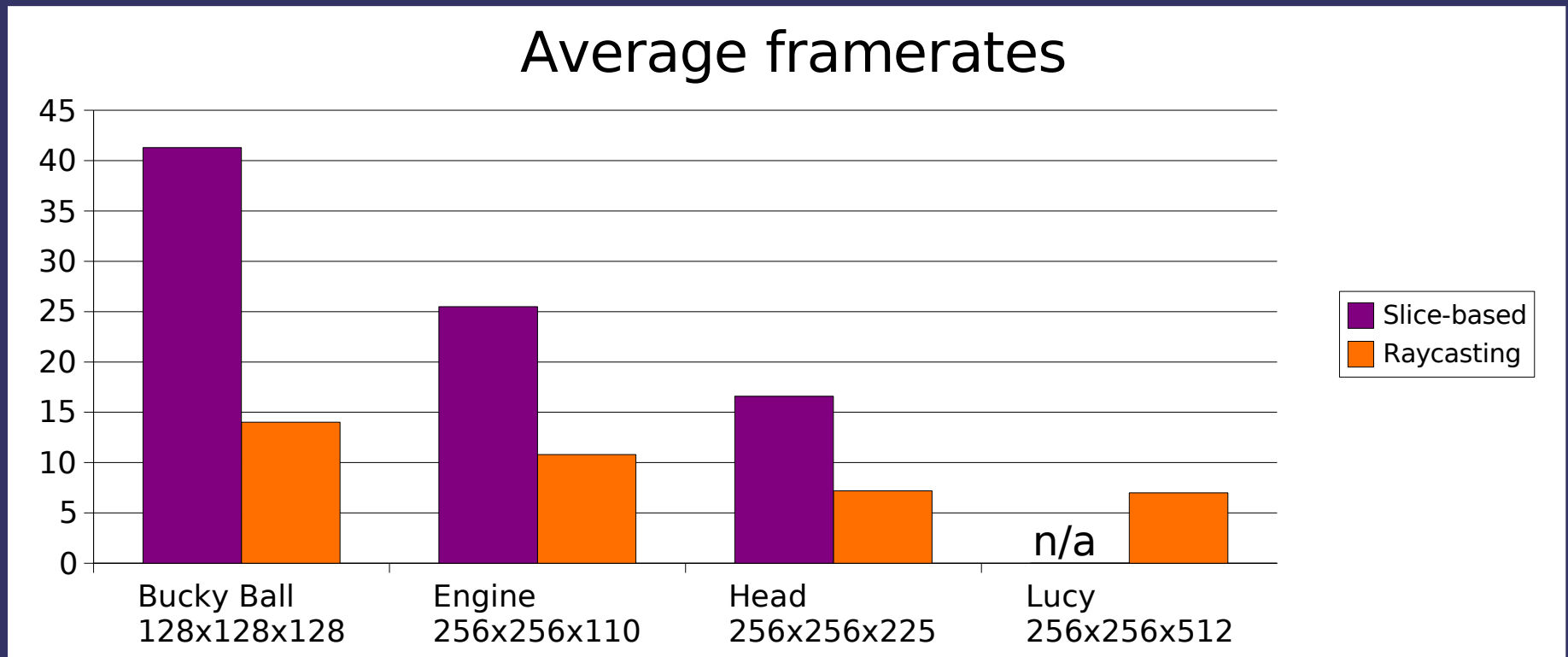
# Performance and results

Hardware:

standard PC with NVIDIA GeForce 6800 GT graphics card.

Benchmarks:

average framerates rotating volumes around the y-axis using a 512x512 resolution (50 slices for the slice-based VR)



# *Proposed optimizations*

- ➔ Porting of the shaders sources to Cg 1.5 or GLSL would result on a handler C code making available, at the same time, many new features and performance improvements (the bottleneck seems to be the flow-control mechanism which was at the beginning implementation on the used NVidia ARB extensions)
- ➔ 3-dimensional ray integration along the ray direction considering the entire volume involved. In fact, In the implemented algorithm much information on the backside of the volume is lost. This optimization can be performed directly on the fragment shader considering that the bounding volume is known a priori and a lot of pre-computed information can be passed to the shader through parameters at runtime. Example: consider the volume around the ray as a function of the dot product between the view direction and the ray direction.

# *Links and references*

*Framework authors page:*

<http://www.vis.uni-stuttgart.de/eng/research/fields/current/spvolren/>

*NVIDIA Cg:*

<http://developer.nvidia.com/page/home.html>

*OpenGL GLSLang (GLSL):*

<http://www.opengl.org/documentation/glsl/>

*Shaders with VTK:*

[http://www.vtk.org/Wiki/VTK\\_Shaders](http://www.vtk.org/Wiki/VTK_Shaders)

[http://www.vtk.org/Wiki/images/3/39/Vtk\\_shaders\\_tudelft\\_tut.pdf](http://www.vtk.org/Wiki/images/3/39/Vtk_shaders_tudelft_tut.pdf)

Andrea Sottoriva

<http://hifi.metalabs.org/>