UNIK-4660/TMA-4235, Summary day 1-2

Introduction and Basic computer graphics

Øyvind Andreassen and Anders Helgeland

oya@ffi.no and ahe@ffi.no

Forsvarets Forskningsinstitutt

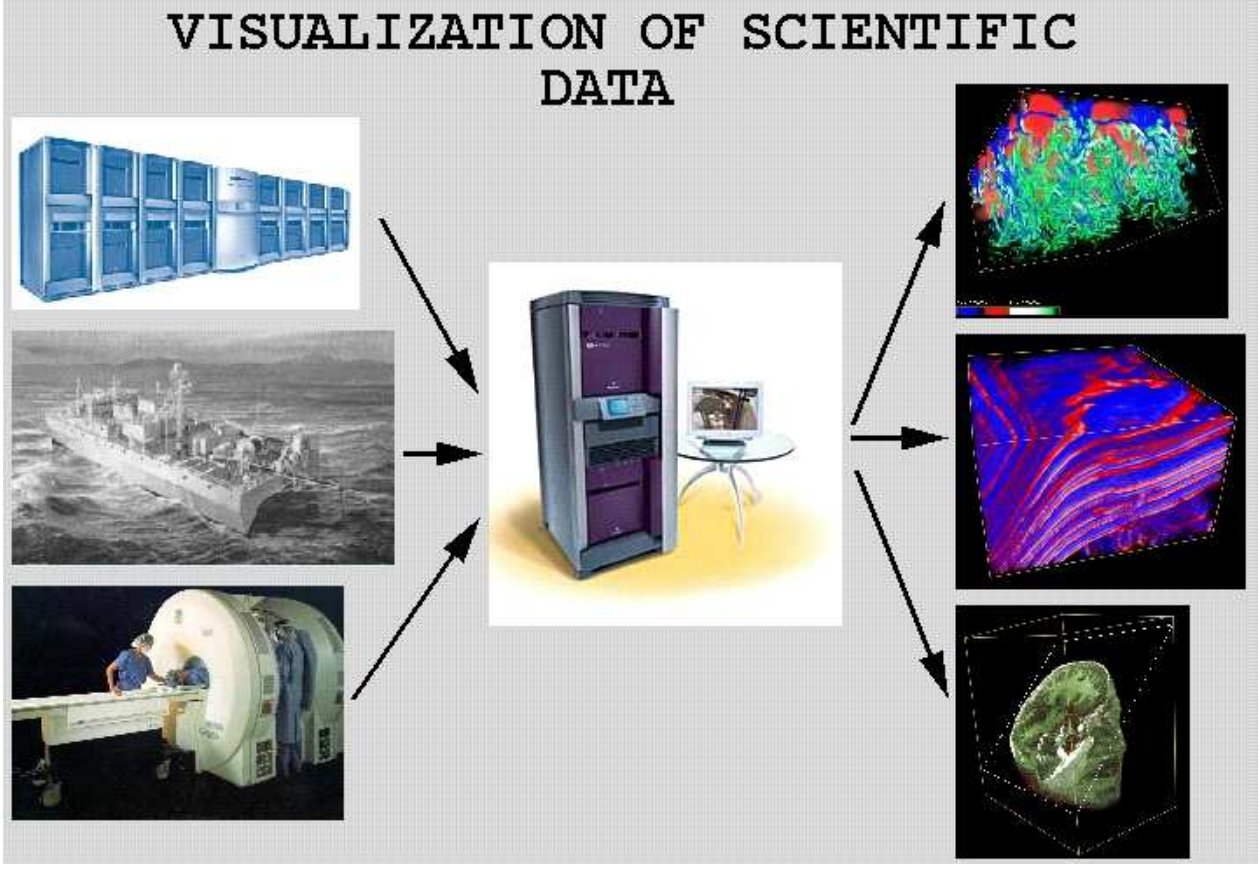
Why visualize?

Our eyes and brain have an amazing ability to identify geometrical patterns and to efficiently extract key visual information. This ability makes visualization to a powerful tool.

- Increasing computing capabilities/improved sensor performance $\Rightarrow$
- Generation of huge data sets $\Rightarrow$
- A challenge to extract key information from the data sets

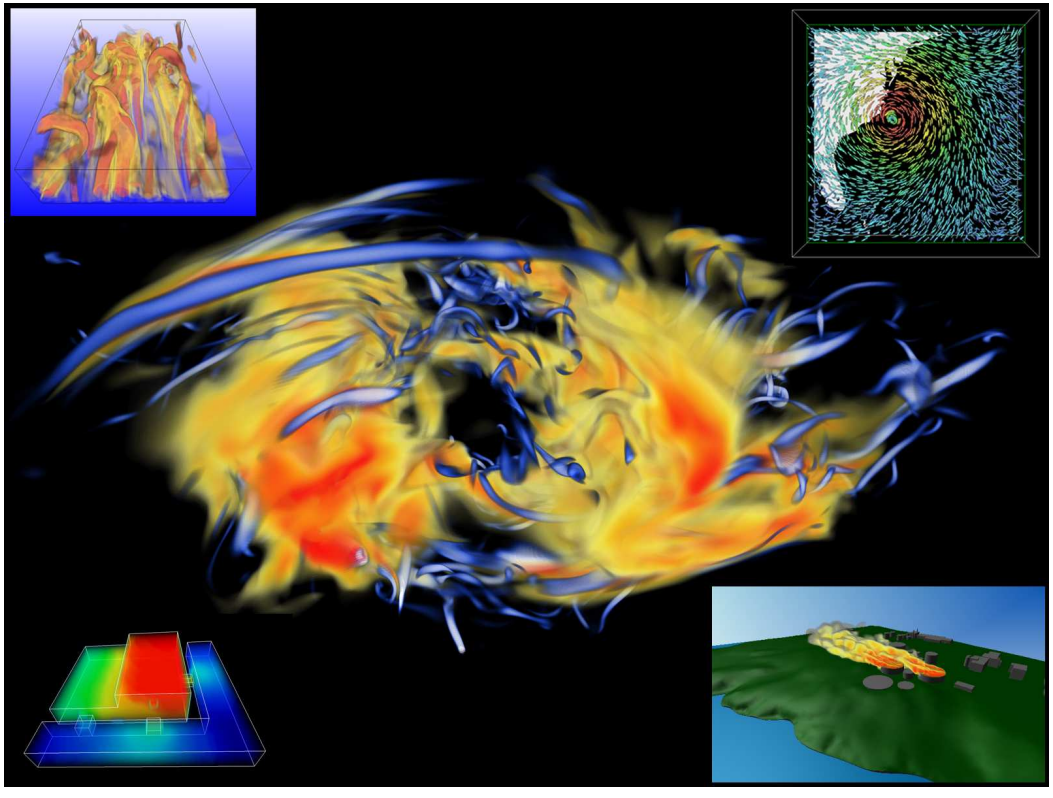
The primary goal of scientific visualization is to communicate relevant physical information contained in a given dataset.

A typical scenario



Situations where visualization is commonly used.

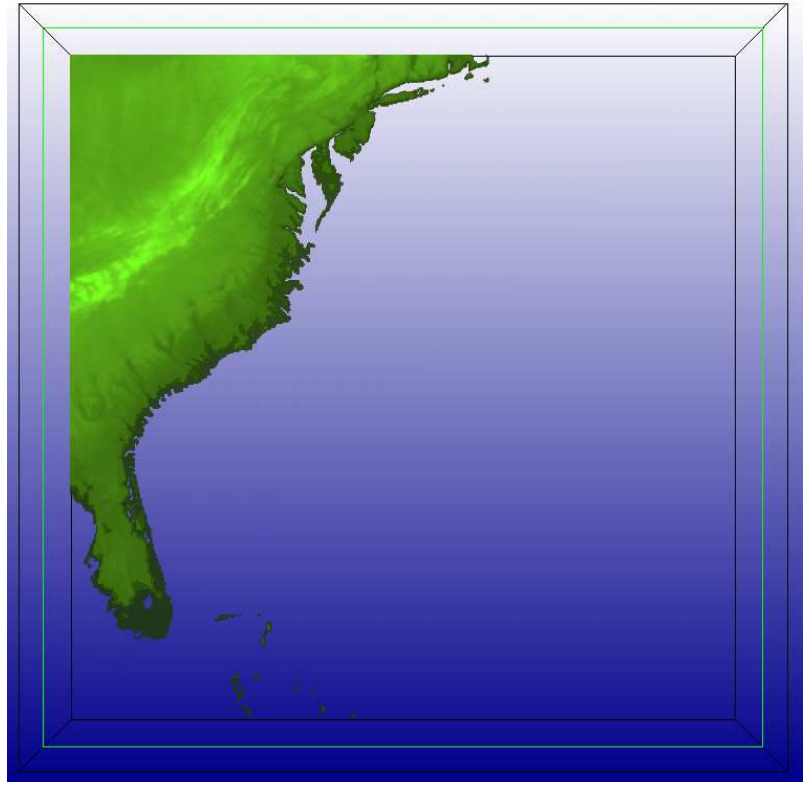
Examples



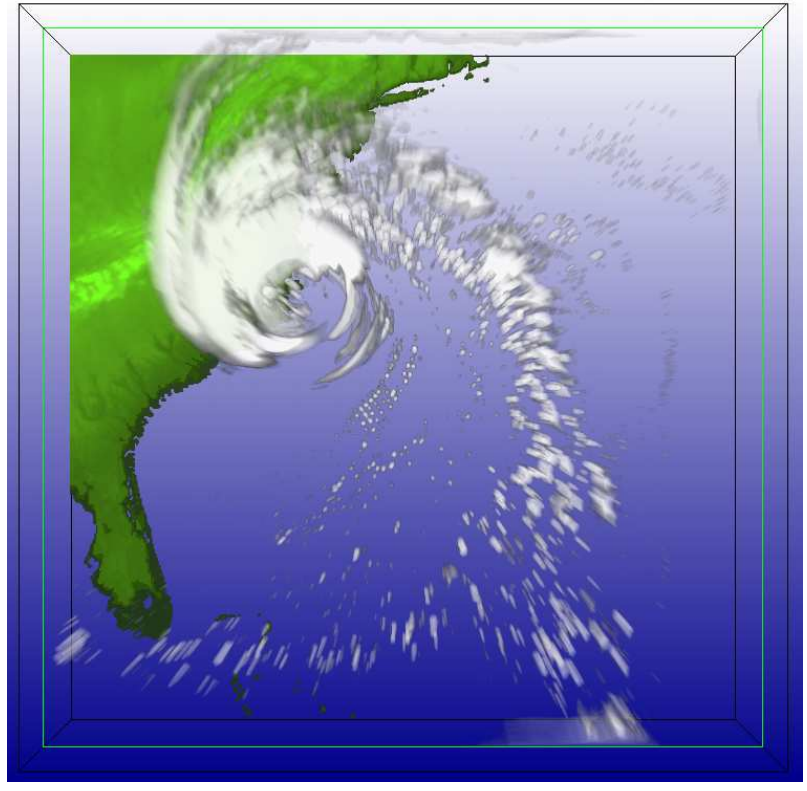
Why do we need visualization?

Need visualization to understand

Without visualization we are blind

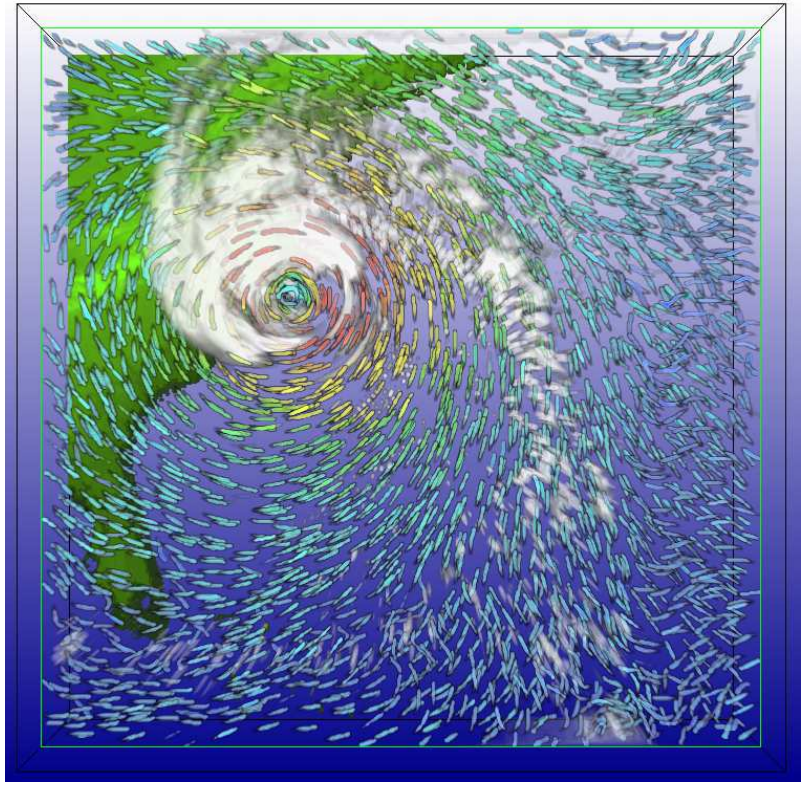


Verify simulation results



See what normally is invisible to the eye

Flow visualization: *'The art of making flow patterns visible'*



0.0 m/s 50 m/s

How did we do it in the past?

The capacity and power of computers have increased dramatically during the last years.

- Until 30 years ago, simulations were mostly one-dimensional
  - Output were printed on line printers or plotted on graph paper
- Until 15 years ago, simulations were mostly up to two-dimensional
  - Visual output were given as contour plots or as color coded images on CTRs

How did we do it recently?

- 15 years ago, a turning point came with the Cray X-MP/Cray Y-MP computer systems
- Low resolution 3D simulations $(\sim 50^3)$ became now feasible
- “Plotting” of 3D data became requested and the field of data visualization grew out of this need
- The need of showing the temporal evolution of 3D data became urgent
- Some degree of interactivity was requested
- Dedicated graphics hardware and software (OpenGL from SGI) designed by the simulator/entertainment industry became utilized for data visualization

How we are doing it today?

- Tflops computer systems are currently available
  - They have 1-Tbytes of memory and 500-Tbytes mass stores
  - Presently the largest Navier-Stokes DNS simulations ever done are for Reynolds number $R_e = UL/\nu \sim 10^4$
- We have still far to go before “useful” work can be done
  - For a cruising B747, $R_e \sim 10^8$
  - The Number of grid points required for this case is at least $N \sim R_e^{9/4} \quad \Rightarrow$
  - There is a need of computer systems 10^9 times larger than current supercomputers

Do we need visualization systems?

Current supercomputers generate a vast amount of data.

Visualization systems must balance computing systems.

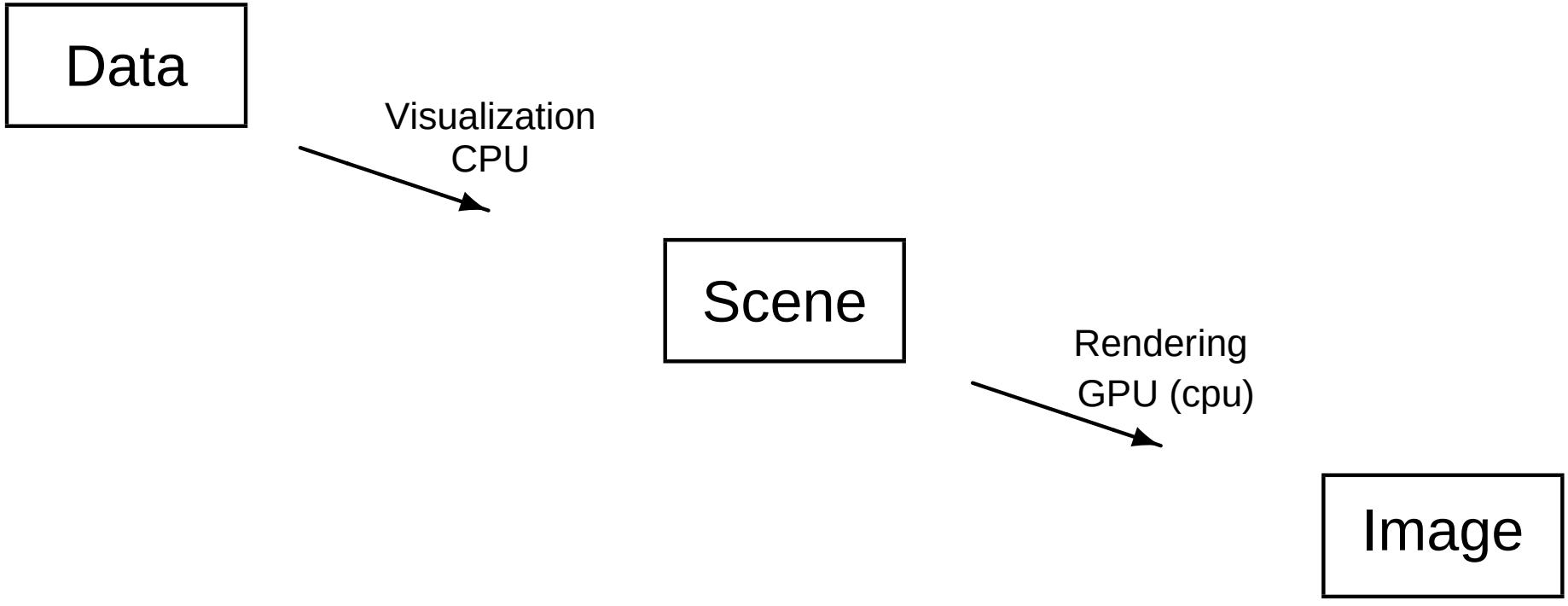
What is needed in visualization

In the visualization process, numerical data are “turned” into a suitable visual form. The process is multi disciplinary and involves among others

- “Physics” - system knowledge
- Numerical mathematics
- Computer graphics software
- Dedicated computer graphics hardware

High degree of interactivity is a requirement for best utilization of our short-time visual memory.

The visualization process



Some components of visualization

Visualization pipeline

- Computation of derived fields, vector magnitudes etc
- Thresholding by data value, clipping in space, and many others
- Color assignment
- Scalar/vector/tensor visualization techniques

Rendering pipeline

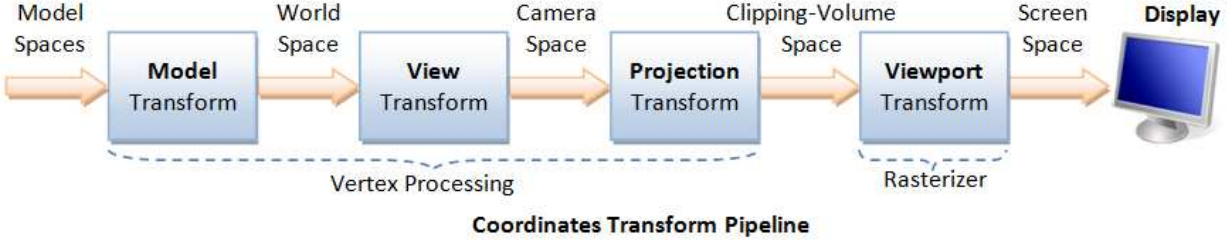
- Vertex transformations
- Lighting
- Rasterization
- Depth sorting (Z-buffering), antialiazing
- Texture mapping

Rendering

Render means represent or portray. In computer graphics, a render is the program that makes the scene objects visible as an image. Common render techniques are:

- Traditional rendering pipeline
- Ray-tracing or ray casting
- Radiosity
- Voxel-based volume rendering

Traditional Graphics Rendering Pipeline

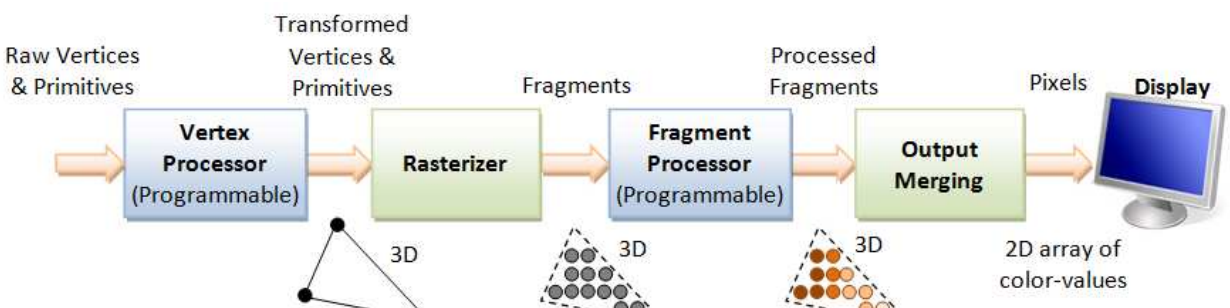


Coordinates Transform Pipeline

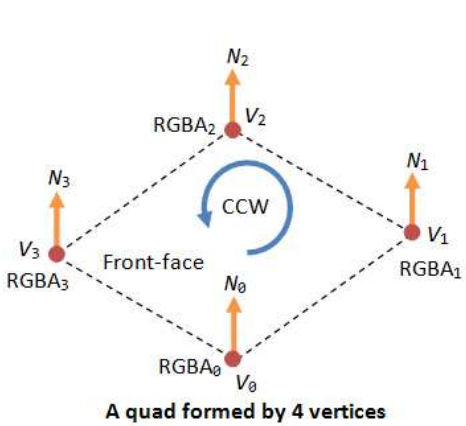
The Camera Analogy

- Modeling transformation -> positioning the model
- Viewing transformation -> positioning the camera
- Projection transformation -> Selecting camera lens, adjusting the zoom
- Viewport transformation -> cropping the image (print photo)

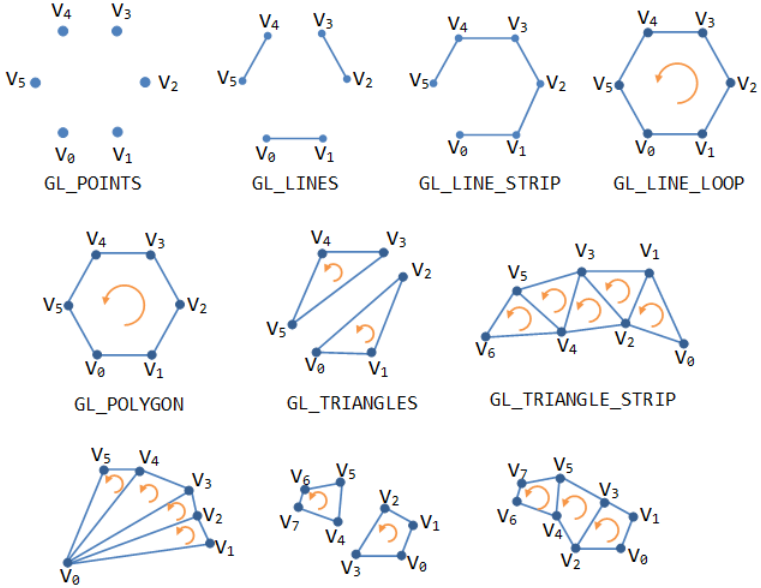
Traditional Graphics Rendering Pipeline



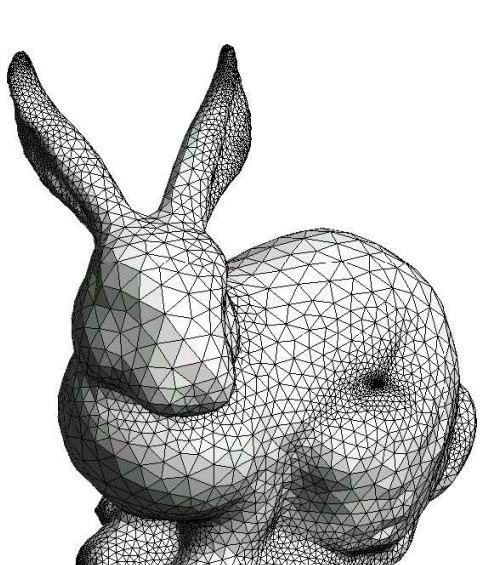
3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (x, y, z) position, color (RGB or RGBA), vertex-normal (n_x, n_y, n_z), and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.



A quad formed by 4 vertices (with color and normal)



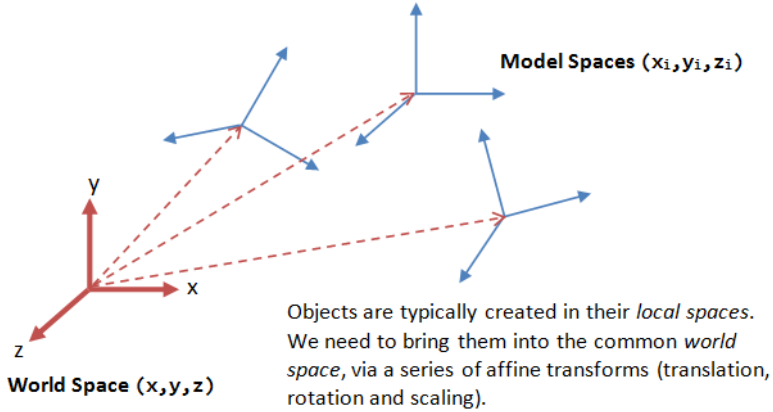
OpenGL Primitives



Modeling transformations

A vertex $\mathbf{v} = (x, y, z, w), w > 0$ (w is usually equal to 1) is transformed in computer hardware by 4x4 matrices.

- Translation: $\mathbf{T}(d_x, d_y, d_z) = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$
- Scaling: $\mathbf{S}(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$



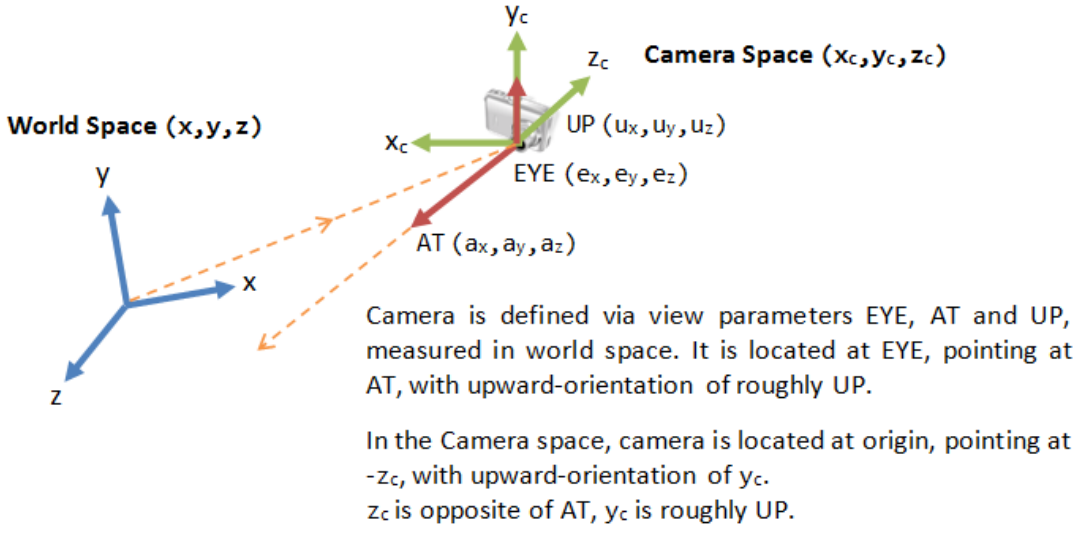
Objects are typically created in their *local spaces*. We need to bring them into the common *world space*, via a series of affine transforms (translation, rotation and scaling).

- Rotation:

$$R_x(\theta) = \begin{pmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, R_y(\theta) = \begin{pmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

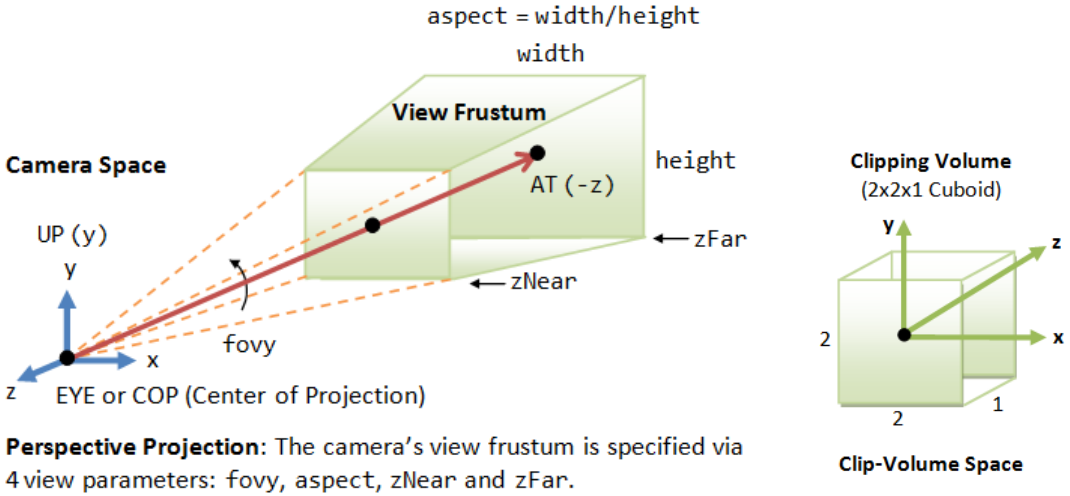
- Model Matrix: $\mathbf{M}_{combined} = \mathbf{M}_1\mathbf{M}_2\mathbf{M}_3$ (Dependent on order)

Viewing transformations



- In OpenGL: can be set by void *gluLookAt(**EYE**, **AT**, **UP**)*
- View transformations can also be performed by moving the objects relative to a fixed camera (*Model transform*).
- *View transform* (Moving the camera) and *Model transform* (moving the object) are equivalent in Computer graphics
- Therefore *Model transform* and *View transform* are combined into a matrix called the *Model-View Matrix*

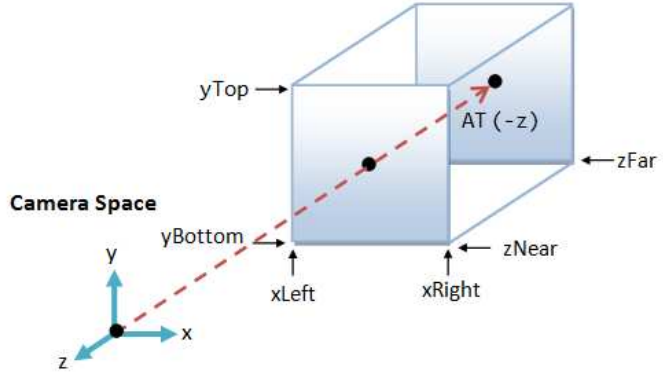
Projection transformations



- In OpenGL: Perspective projection can be set by void *gluPerspective(fovy, aspectRatio, zNear, zFar)*
- Projection Matrix:

$$\mathbf{M}(proj) = \begin{pmatrix} \frac{cot(fovy/2)}{aspect} & 0 & 0 & 0 \\ 0 & cot(fovy/2) & 0 & 0 \\ 0 & 0 & -\frac{zFar}{zFar-ZNear} & -\frac{zNear \times xFar}{zFar-zNear} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

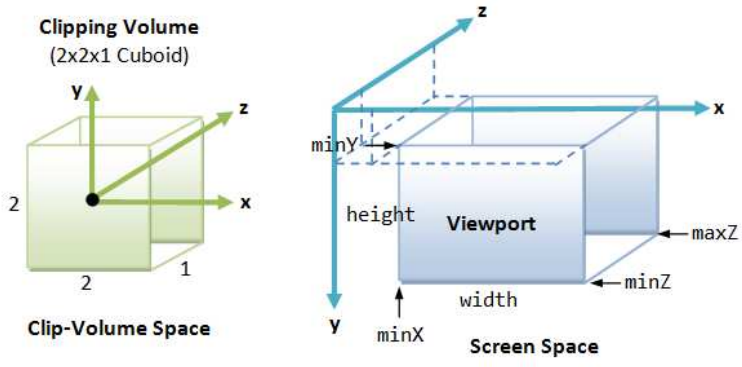
Projection transformations



Orthographic Projection: Camera positioned infinitely far away at $z = \infty$

- In OpenGL: Orthographic projection can be set by void *void glOrtho(xLeft, xRight, yBottom, yTop, zNear, zFar)*

Viewport transformations

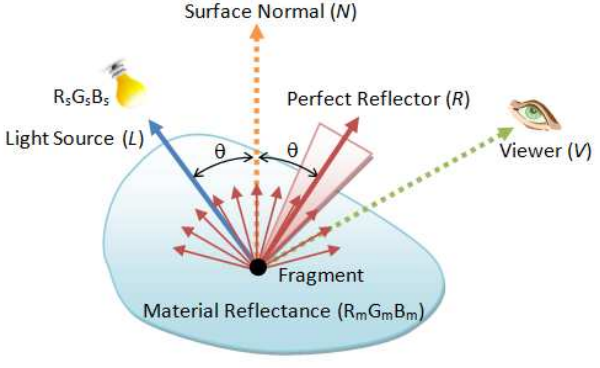


- In OpenGL: Viewport transformation can be set by void *void glViewport(xTopLeft, yTopLeft, width, height)*
- Viewport Matrix: $\mathbf{M}(viewport) = \begin{pmatrix} w/2 & 0 & 0 & minX+w/2 \\ 0 & -h/2 & 0 & minY+h/2 \\ 0 & 0 & maxZ-minZ & minZ \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Light and reflection models

Phong Lighting Model

- Surfaces can be characterized locally by a distinct outward normal vector $\mathbf{N}$. This normal vector plays an important role when describing the interaction of light with surface elements.
- Using the Phong reflection model, the light intensity at a particular surface point is given by

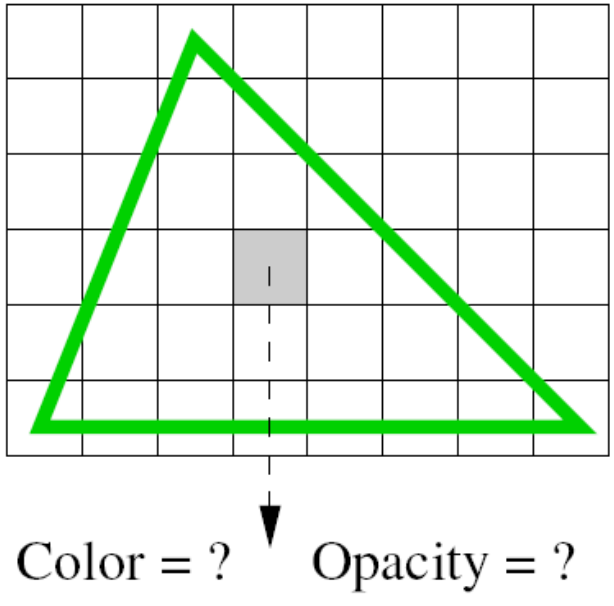


$$
\begin{aligned}
I &= I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}}, \\
&= k_a + k_d(\mathbf{L}\cdot\mathbf{N}) + k_s(\mathbf{V}\cdot\mathbf{R})^n,
\end{aligned}
\tag{1}
$$

- where $\mathbf{L}$ denotes the light direction, $\mathbf{V}$ the viewing direction and $\mathbf{R}$ the unit reflection vector. $\mathbf{R}$ is the vector in the $\mathbf{L}$-$\mathbf{N}$-plane with the same angle to the surface normal as the incident light.

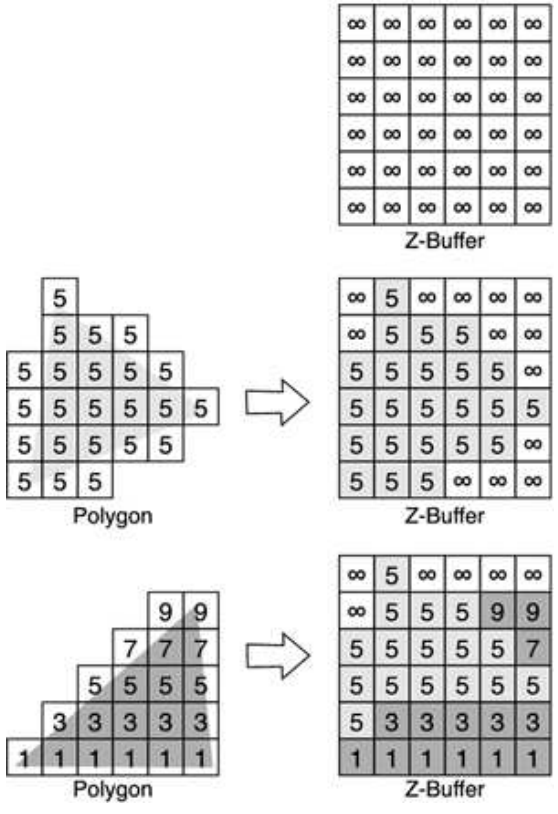
Rasterization

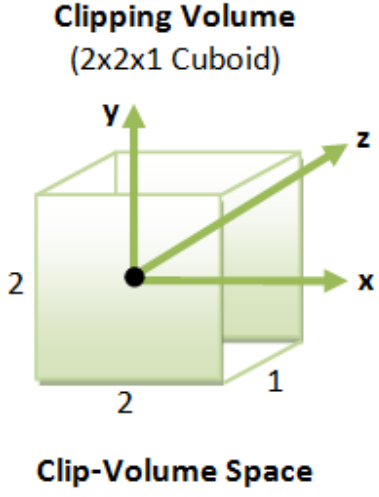
- In the rasterization stage, each primitive (such as triangle, quad, point and line), which is defined by one or more vertices, are raster-scan to obtain a set of fragments enclosed within the primitive. Fragments can be treated as 3D pixels, which are aligned with the pixel-grid. The 2D pixels have a position and a RGB color value. The 3D fragments, which are interpolated from the vertices, have the same set of attributes as the vertices, such as position, color, normal, texture.



Z-buffer, hidden surface removal

- A technique for hidden surface removal (without depth-sorting of the plygons)
- front clipping plane $z = 0$, back clipping plane $z = \infty$



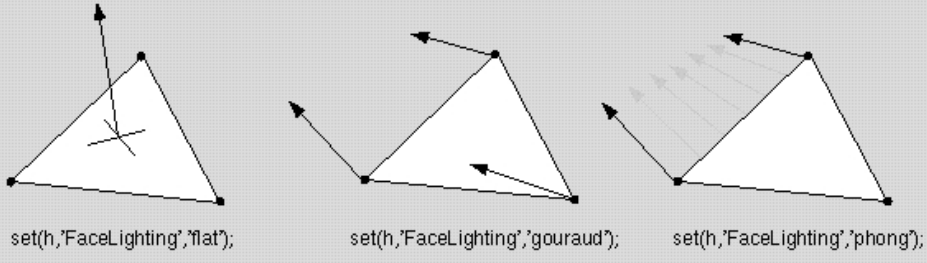


Shading

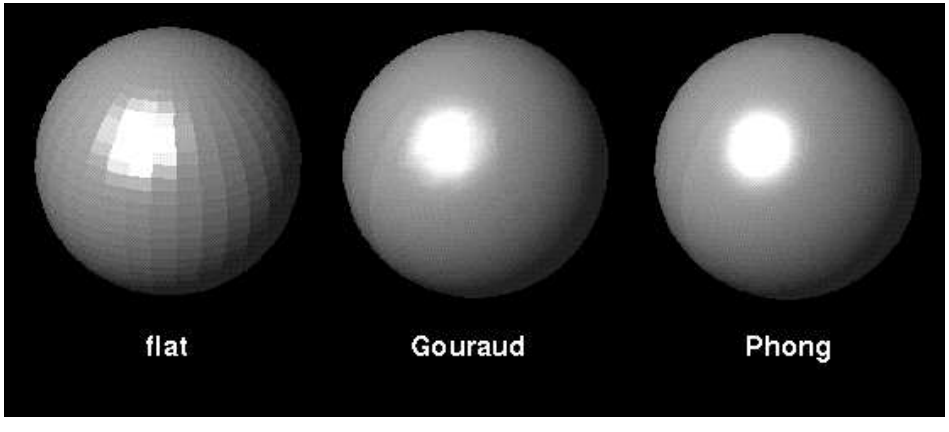
The figures below show the different shading techniques and the results when applied to a sphere.

Eksemplet er gjort med Matlab
ved hjelp av kommandoene:

```
f = figure;
set(f,'Color',[0 0 0]);
sphere(32);
axis vis3d off;
h = findobj('Type','surface');
shading interp;
set(h,'FaceColor',[0.5 0.5 0.5]);
set(h,'FaceLighting','phong');
light('Position',[-3 -1 3]);
set(h,'DiffuseStrength',1.0);
set(h,'SpecularStrength',1);
set(h,'SpecularExponent',10);
set(h,'AmbientStrength',0.25);
set(h,'BackFaceLighting','lit')
```



Concepts for flat, Goraud and Phong shading



Flat, Goraud and Phong shading of a sphere

GLSL example: Phong shading

```
varying vec3 N;
varying vec3 v;

void main(void)
{
   v = vec3(gl_ModelViewMatrix * gl_Vertex);
   N = normalize(gl_NormalMatrix * gl_Normal);

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Vertex Shader Source Code

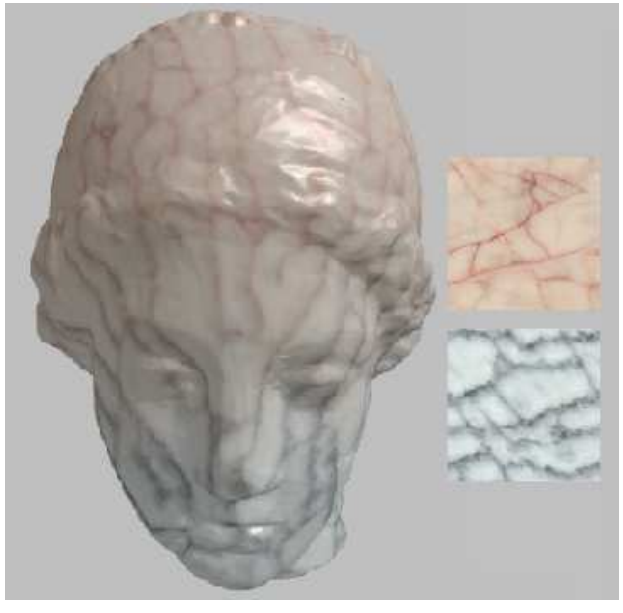
```
varying vec3 N;
varying vec3 v;

void main (void)
{
   vec3 L = normalize(gl_LightSource[0].position.xyz - v);
   vec3 E = normalize(-v); // we are in Eye Coordinates, so EyePos is (0,0,0)
   vec3 R = normalize(-reflect(L,N));

   //calculate Ambient Term:
   vec4 Iamb = gl_FrontLightProduct[0].ambient;

   //calculate Diffuse Term:
   vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 0.0);
   Idiff = clamp(Idiff, 0.0, 1.0);

   // calculate Specular Term:
   vec4 Ispec = gl_FrontLightProduct[0].specular
                * pow(max(dot(R,E),0.0),0.3*gl_FrontMaterial.shininess);
   Ispec = clamp(Ispec, 0.0, 1.0);

   // write Total Color:
   gl_FragColor = gl_FrontLightModelProduct.sceneColor + Iamb + Idiff + Ispec;
}
```

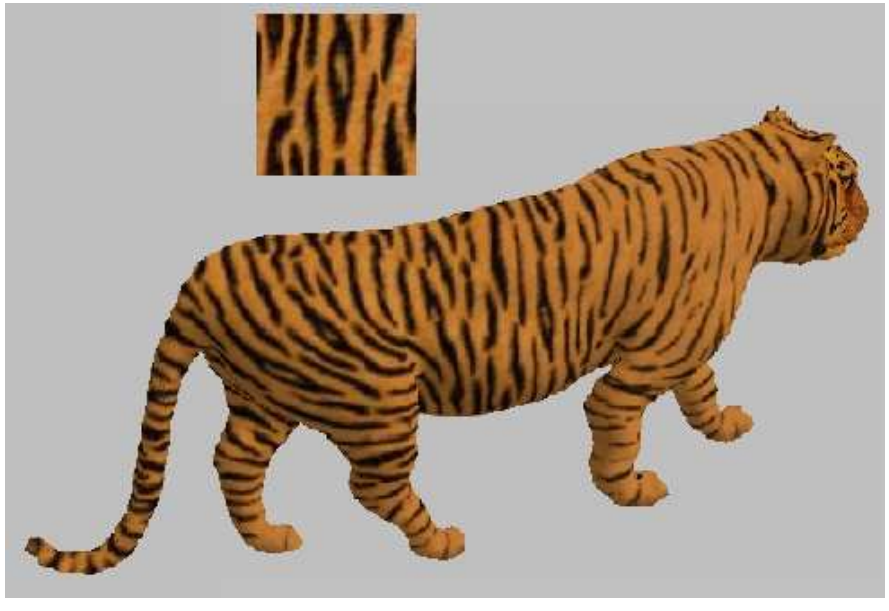
Fragment Shader Source Code

$$\begin{aligned}\mathbf{C} &= \mathbf{C}_{object} + I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}},\\ &= \mathbf{C}_{object} + k_a + k_d(\mathbf{L}\cdot\mathbf{N}) + k_s(\mathbf{V}\cdot\mathbf{R})^n.\end{aligned}$$

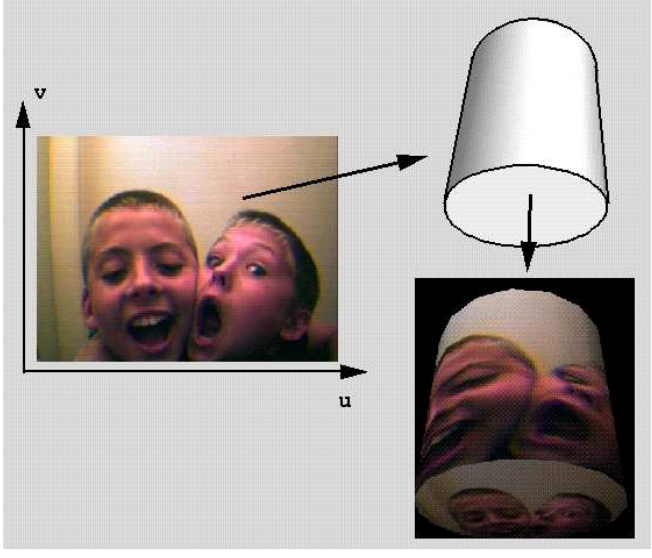
Texture mapping

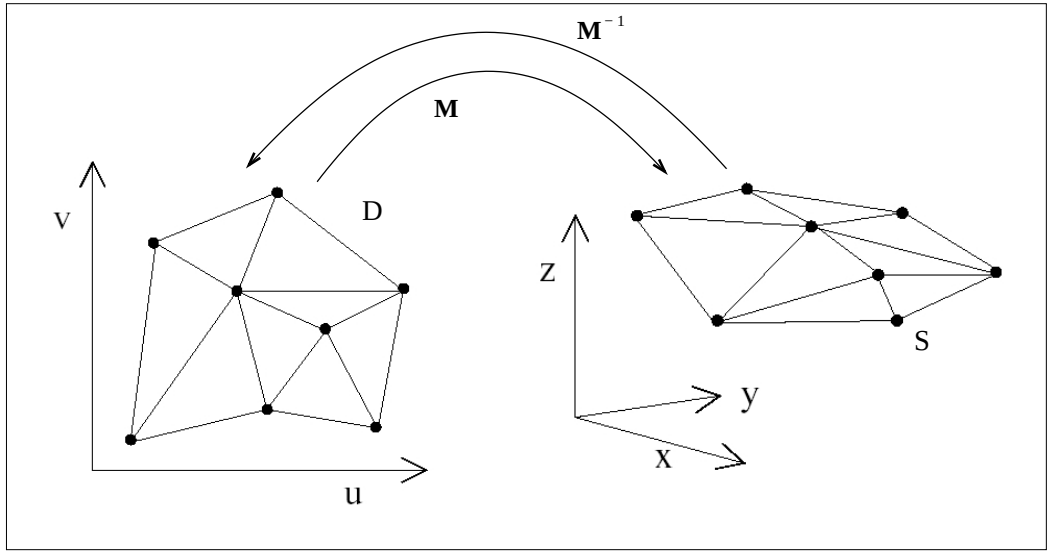
Can be used to reproduce the visual complexity of the real world without needing all details at the geometry level (vertices, geometric primitives).





Texture mapping - 2D example





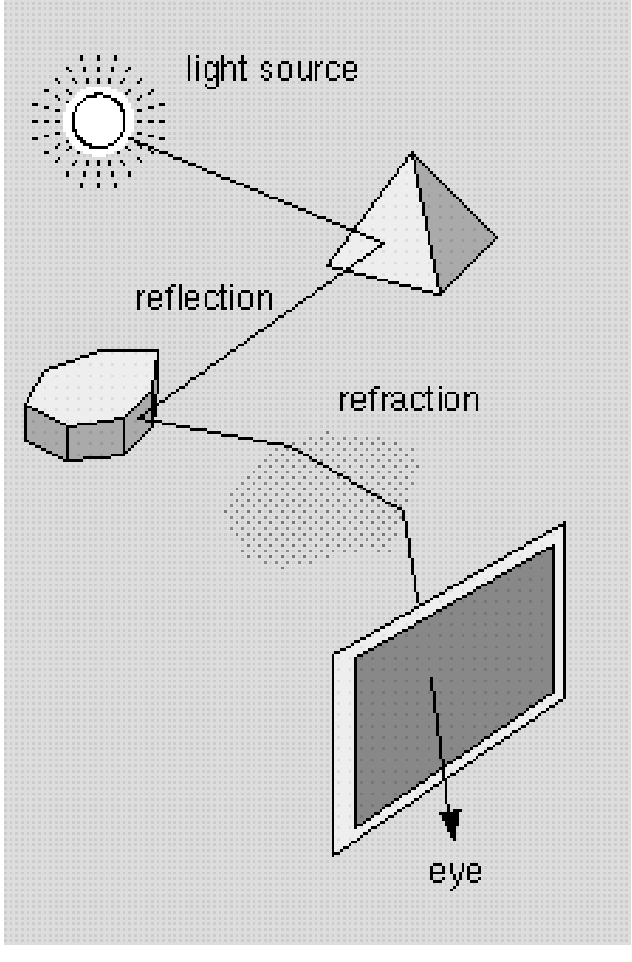
$$\mathbf{M}(u,v) = (x(u,v), y(u,v), z(u,v))$$

$$\mathbf{Q}(x,y,z) = \mathbf{M}^{-1}(x,y,z) = (u,v)$$

Alternative rendering techniques

- *Standard GPU pipeline* (OpenGL): real-time, but shading based on local effects. No shadows in basic pipeline.
- *Ray Tracing*: Global shading model particularly good at specular effects (shiny surfaces). Initially too computationally expensive to be real-time. Now, there exist hardware-accelerated (GPU-based) ray tracing techniques.
- *Radiosity*: Global shading model particularly good at diffuse effects (matte surfaces, indirect light). Initially too computationally expensive to be real-time. But well suited for storing results as textures (as diffuse light is not viewpoints dependent. Now, there exist GPU-based radiosity techniques.

Ray-tracing

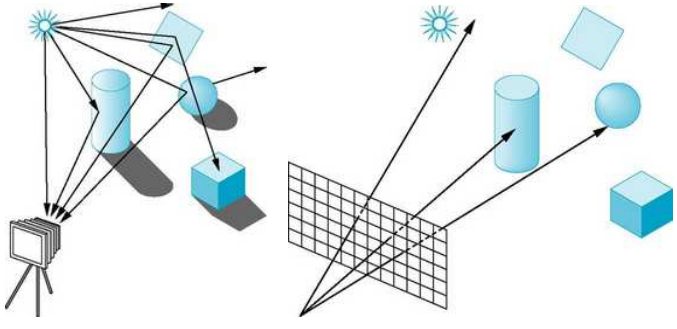


Ray-tracing

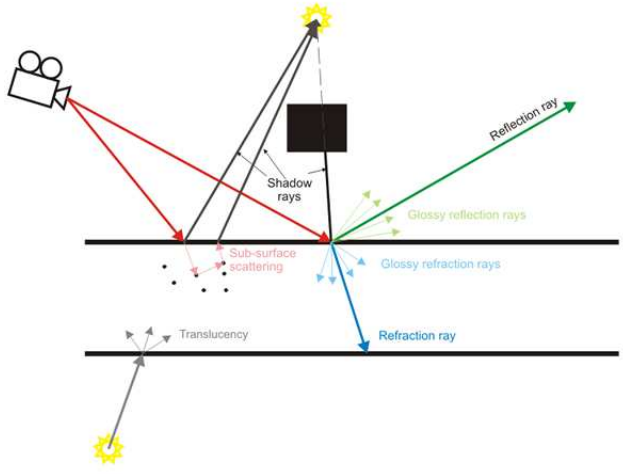
Ray-tracing

The rays are computed from an eye-point, through each pixel on the screen, refracted/reflected around until a light source or back plane of the scene is hit.

- Gives a high degree of realism.
- Can render effects like reflections, refractions and shadows.
- More computationally expensive that Standard GPU pipeline.



Ray-tracing



Four types for rays in ray tracing

- *Eye rays*: originate at the eye
- *Shadow rays*: from the surface point toward light source
- *Reflection rays*: from surface point in reflected direction
- *Refraction rays*: from (semi-transparent) surface in refracted direction

Ray-tracing algorithm

Recursive algorithm:

1. For each pixel, trace primary ray in view direction to the first visible surface
2. For each intersection, trace secondary rays:
   - *Shadow rays* in direction to light sources.
   - *Reflected ray*.
   - *Refracted* or *transmitted* ray.
   - Intensitive of ray $I(ray) = I_{local} + I_{reflected} + I_{transmitted}$, where I_{local} is computed with the Phong model for intersections where the shadow ray gets to a light source and not gets occluded by another object.

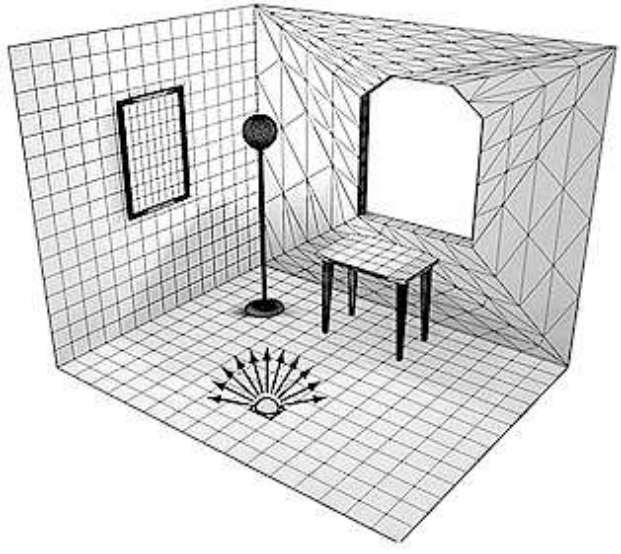
Ray-tracing

Radiosity

Global shading model particularly good at diffuse effects (matte surfaces, indirect light).

Radiosity

Entire model is divided into a number of patches.



- The radiosity (amount of energy) leaving a single patch (surface element) is the sum of light emitted plus the reflected light from all the other patches.
- View-independent rendering model.

Radiosity

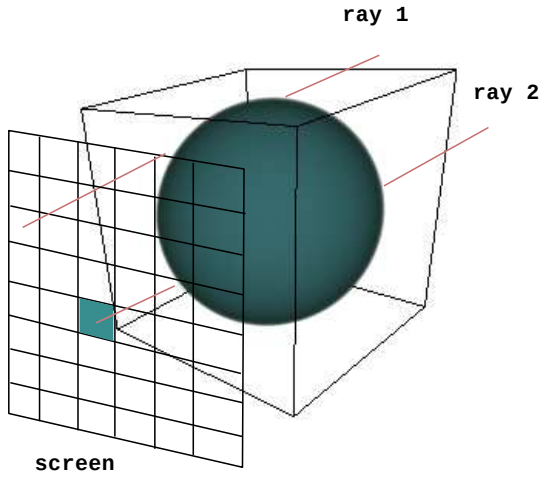
For Radiosity, the following equations are solved for each patch

$$A_i B_i = A_i E_i + R_i \Sigma_{j=1}^{n} B_j F_{ij} A_j$$

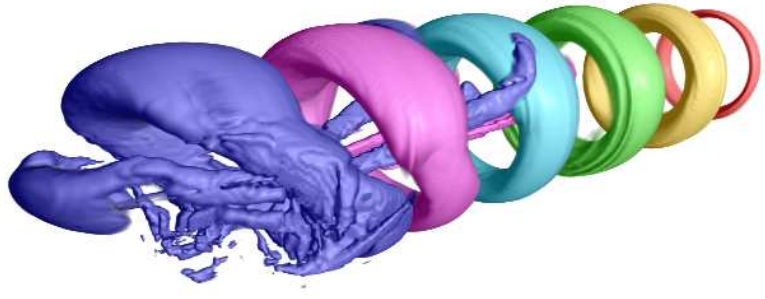
- Here B_i and B_j are the radiosities (emitted energy) from patches i and j.
- E_i is the emitted energy from patch i.
- R_i is the reflectivity coefficient of patch i.
- F_{ij} is the form factor and specifies the amount of energy leaving patch j that reaches patch i.
- A is the area.

Volume rendering

Ray Casting

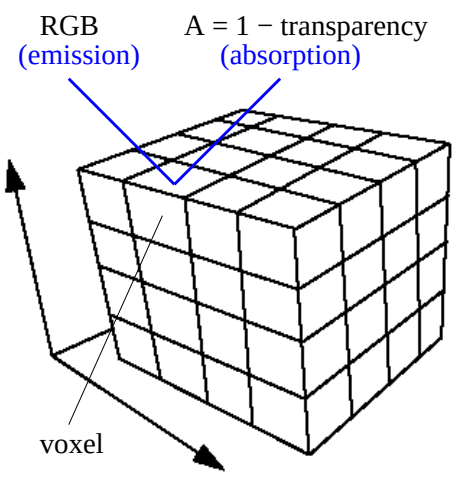


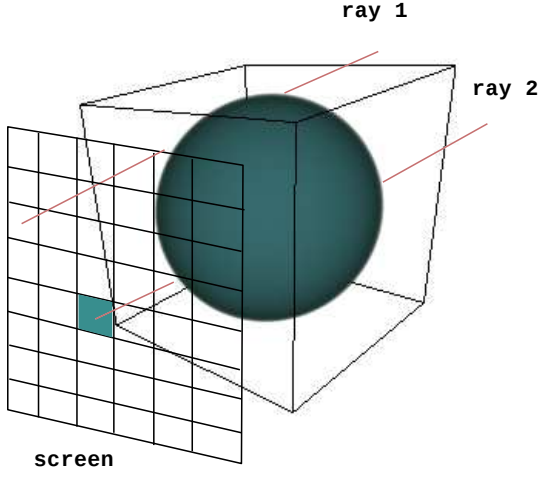
Texture-Based Volume Rendering

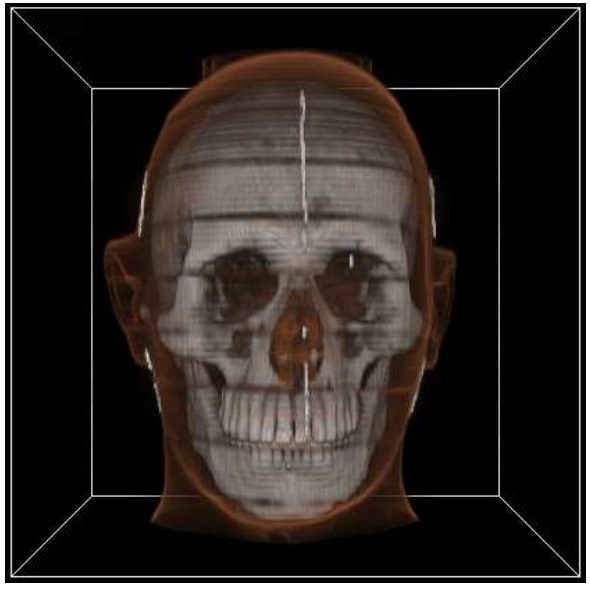


Volume rendering

- Technique for visualizing volume data
- Based on light transport models
- VRI: $I(D) = I(0)T(0, D) + \int_{s=0}^{D} g(s)T(s, D)$
- DVRI: $I(D) = \sum_{i=0}^{n} C_i A_i \prod_{j=i+1}^{n}(1 - A_j)$







Volume rendering

Compositing Schemes

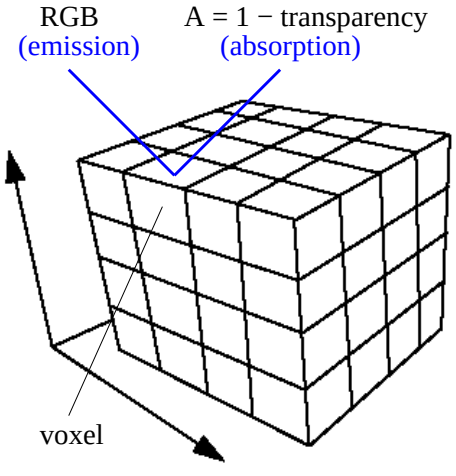
- Back-to-front

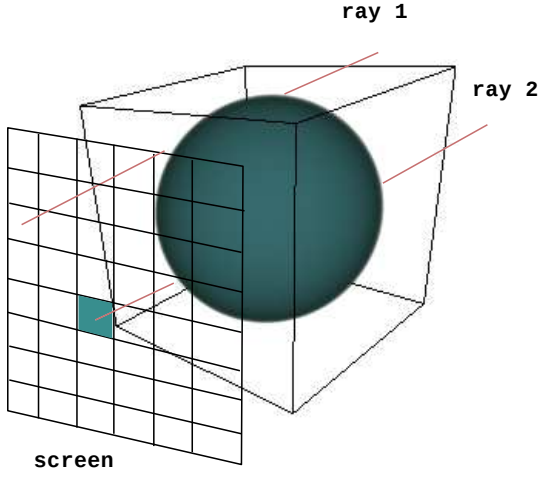
$$\tilde{C}'_i = A_i C_i + (1 - A_i)\tilde{C}'_{i-1}, \quad A'_i = A_i + (1 - A_i)A'_{i-1}$$

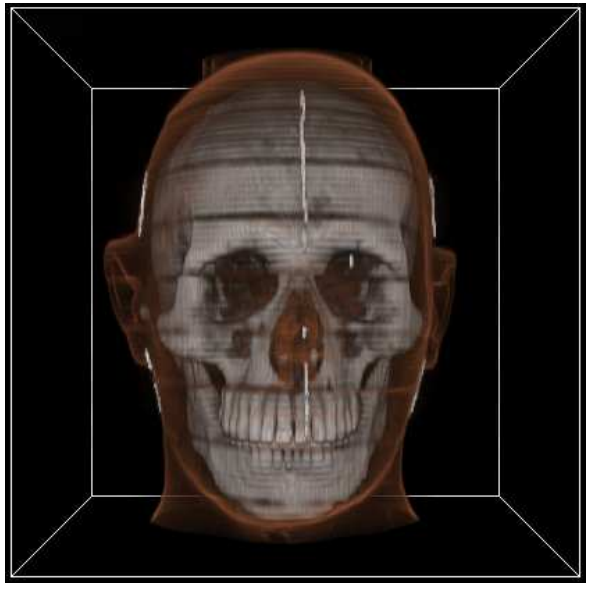
- Front-to-back

$$\tilde{C}'_i = \tilde{C}'_{i+1} + (1 - A'_{i+1})A_i C_i, \quad A'_i = A'_{i+1} + (1 - A'_{i+1})A_i$$

$$(\tilde{C}_i = A_i C_i)$$



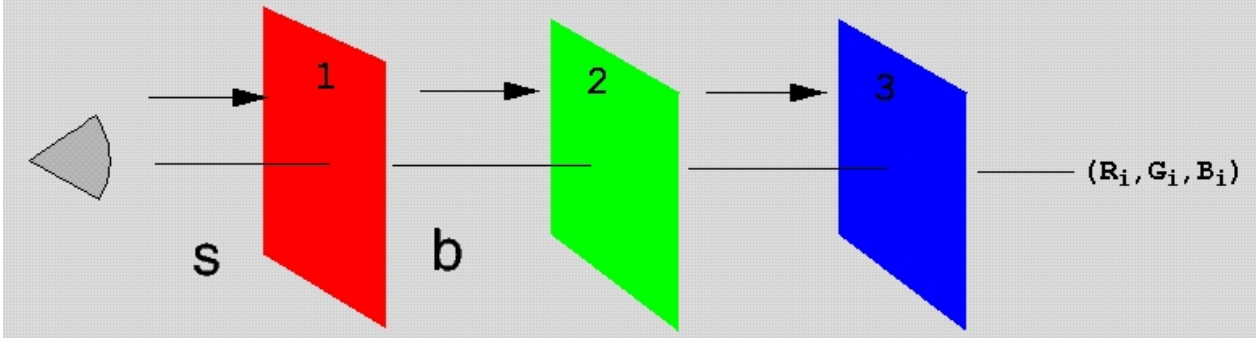




Example: Blending of three planes

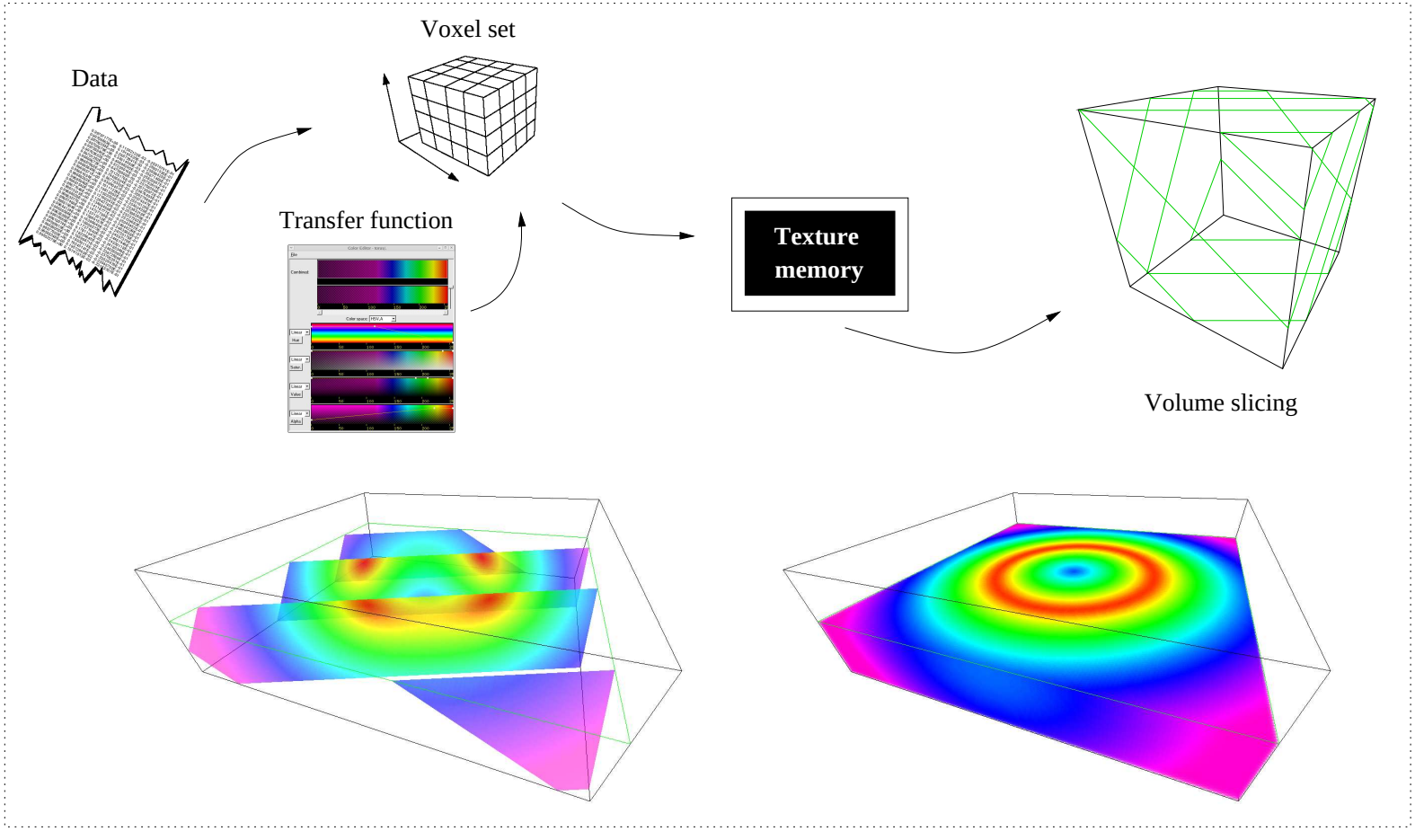
DVRI: $C = \sum_{i=0}^{n} C_i A_i \prod_{j=i+1}^{n} (1 - A_j)$

DVRI: $A = \sum_{i=0}^{n} A_i \prod_{j=i+1}^{n} (1 - A_j)$



$$
\begin{aligned}
R &= \alpha_1 R_1 + (1-\alpha_1)\{\alpha_2 R_2 + (1-\alpha_2)[\alpha_3 R_3 + (1-\alpha_3) R_i]\} \\
G &= \alpha_1 G_1 + (1-\alpha_1)\{\alpha_2 G_2 + (1-\alpha_2)[\alpha_3 G_3 + (1-\alpha_3) G_i]\} \\
B &= \alpha_1 B_1 + (1-\alpha_1)\{\alpha_2 B_2 + (1-\alpha_2)[\alpha_3 B_3 + (1-\alpha_3) B_i]\} \\
\alpha &= \alpha_1 + (1-\alpha_1)\{\alpha_2 + (1-\alpha_2)\alpha_3\}
\end{aligned}
$$

GPU-based volume slicing



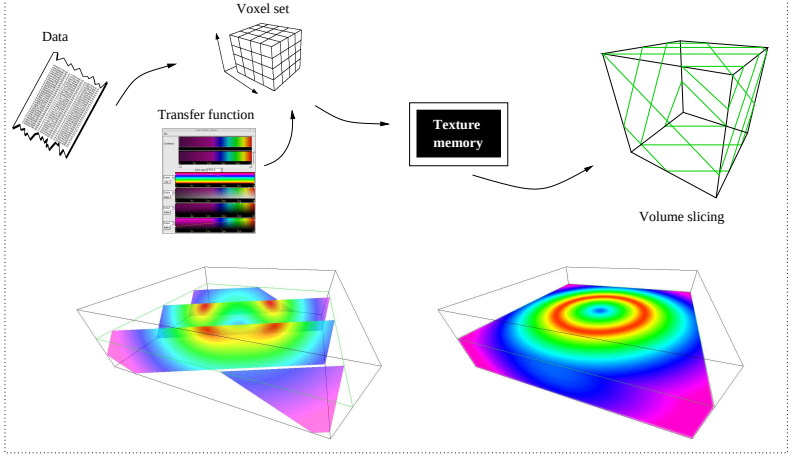
GPU-based volume slicing

Texturing in OpenGL

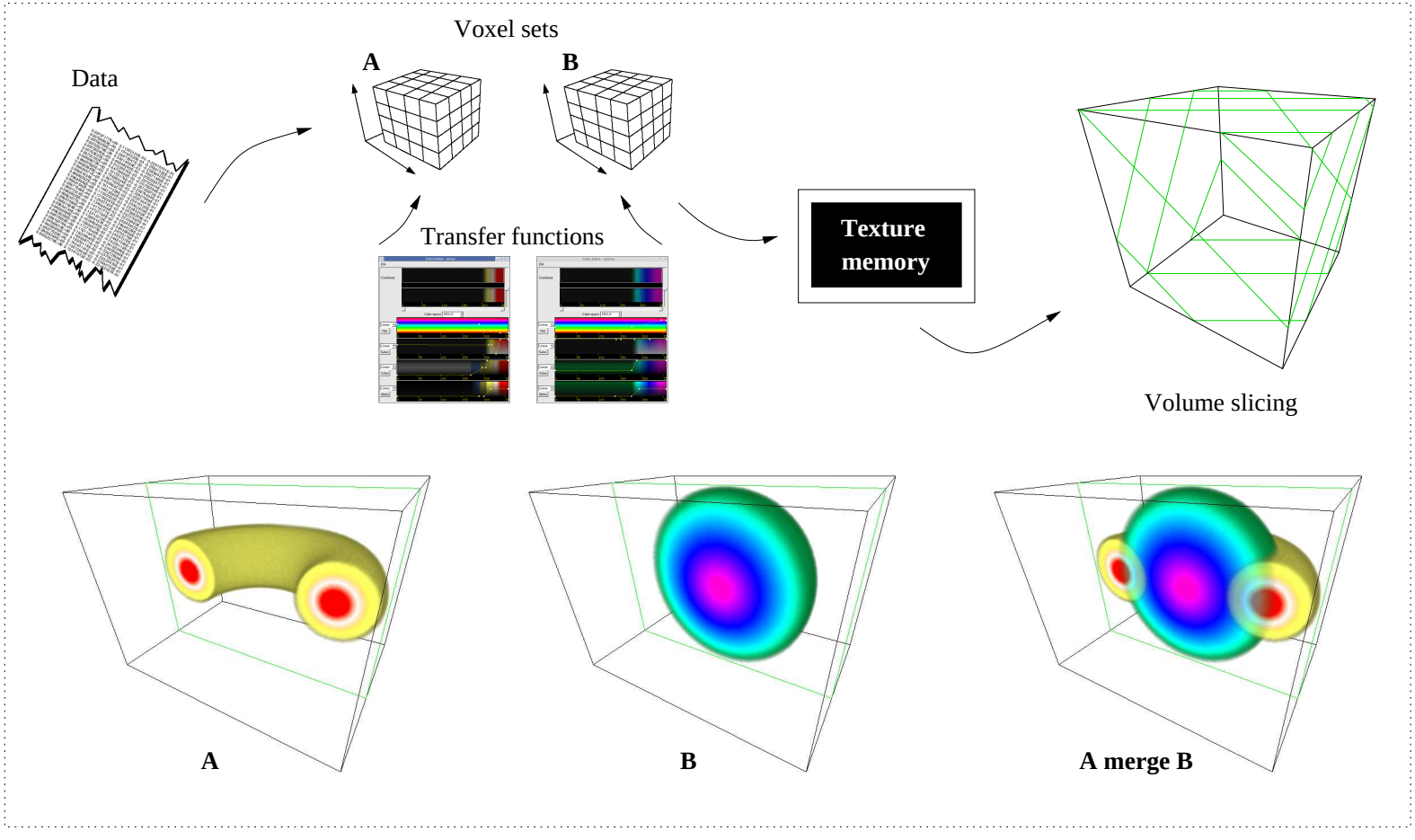
- glGenTextures
- glBindTexture
- glTexImage1D, glTexImage2D, glTexImage3D
- glTexParameteri

OpenGL code for **OVER** operator

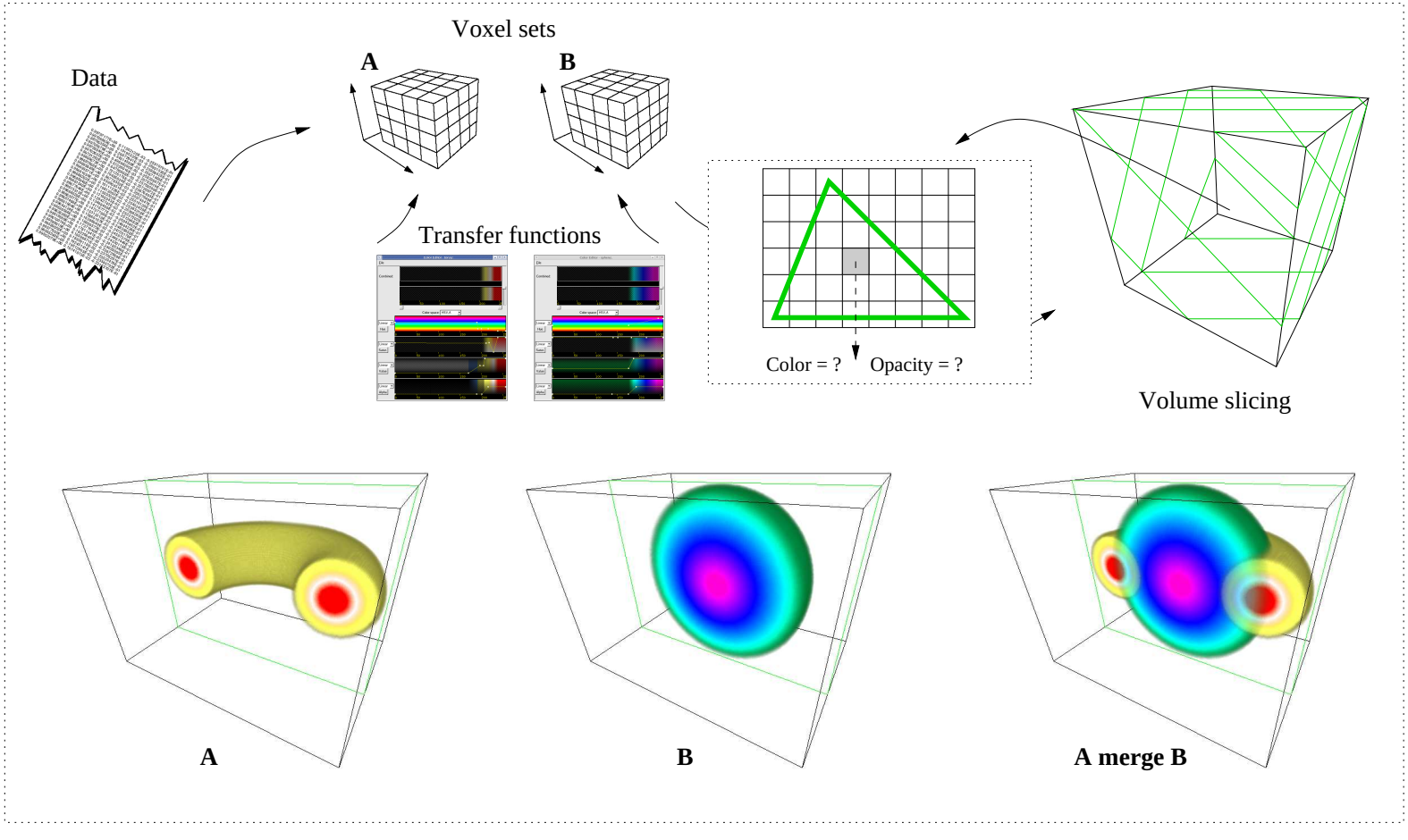
- glEnable (GL_BLEND)
- glBlendFunc (GL_ONE, GL_ONE_MINUS_SRC_ALPHA)



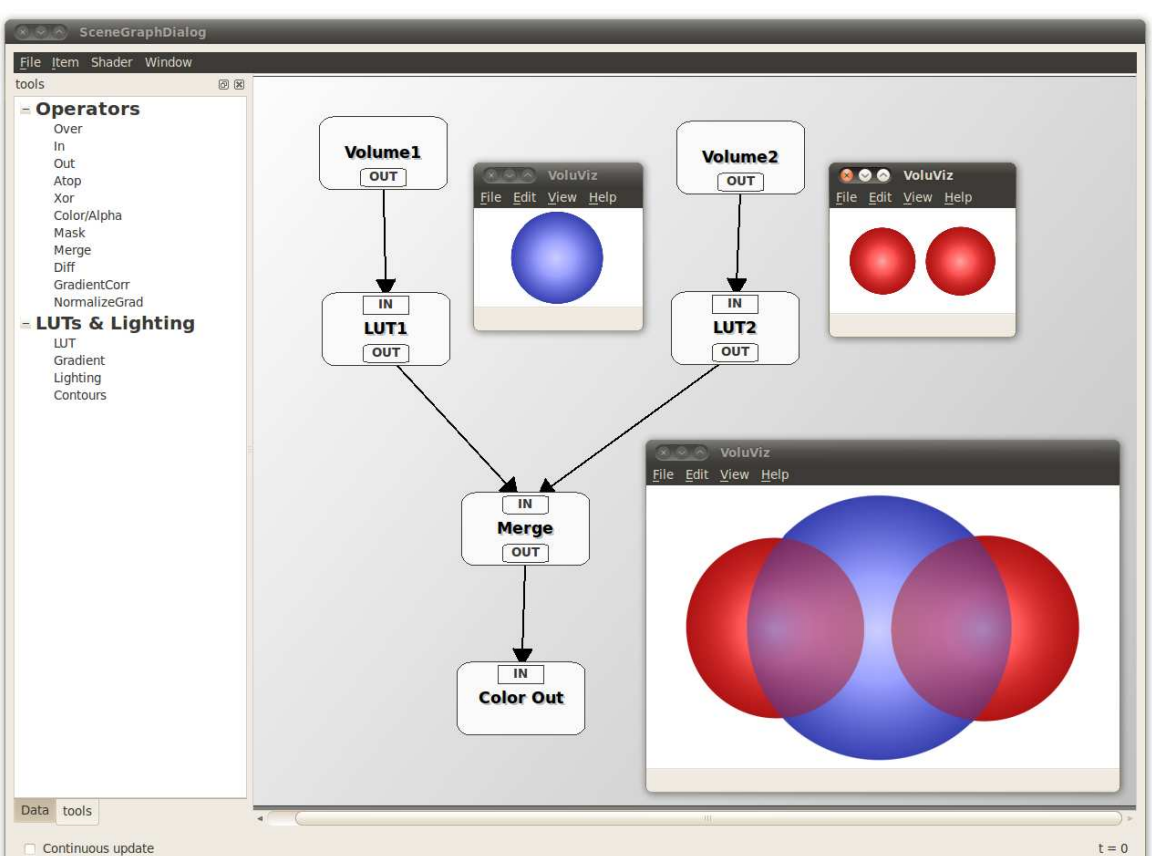
Flexible multifield volume rendering

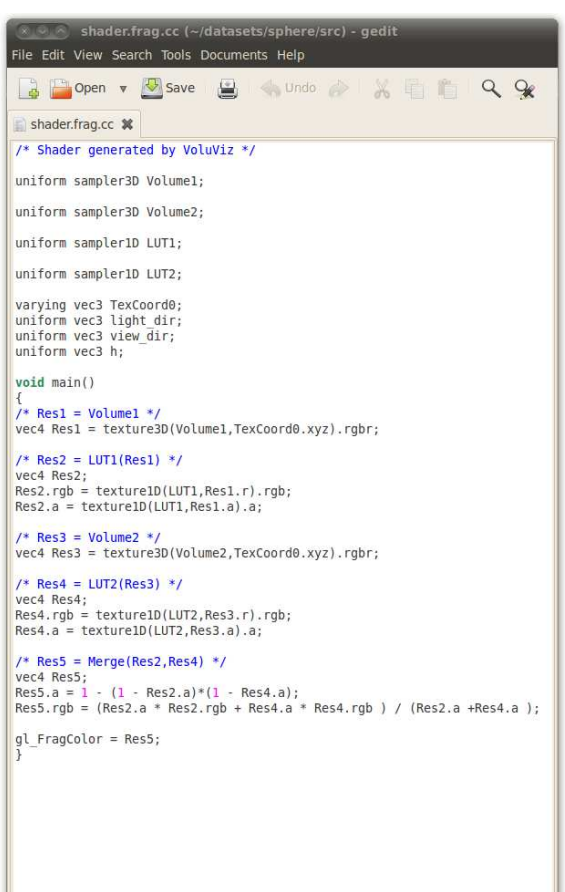


Flexible multifield volume rendering



Flexible VoluViz framework

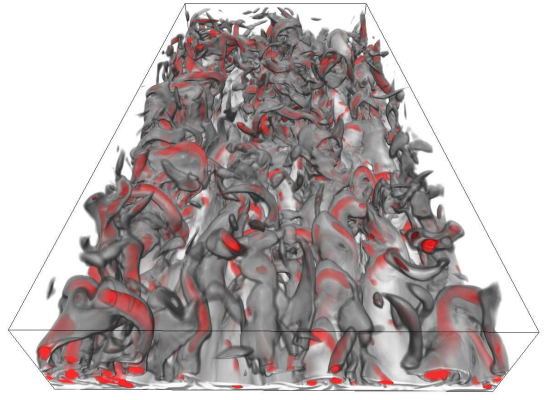


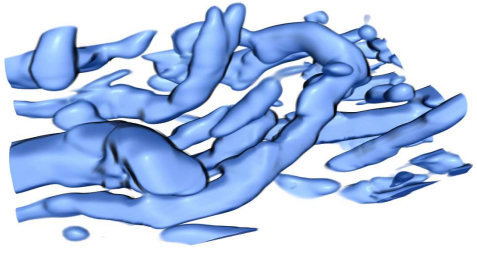


$$A = 1 - \prod_{i=1}^{n}(1 - a_i),$$
$$C = \left(\sum_{i=1}^{n} c_i a_i\right) / \sum_{i=1}^{n} a_i.$$

Operators

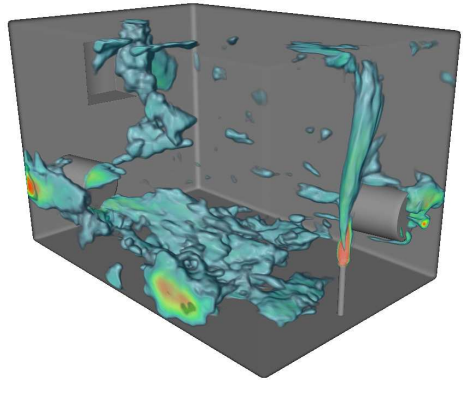
Compositing operator



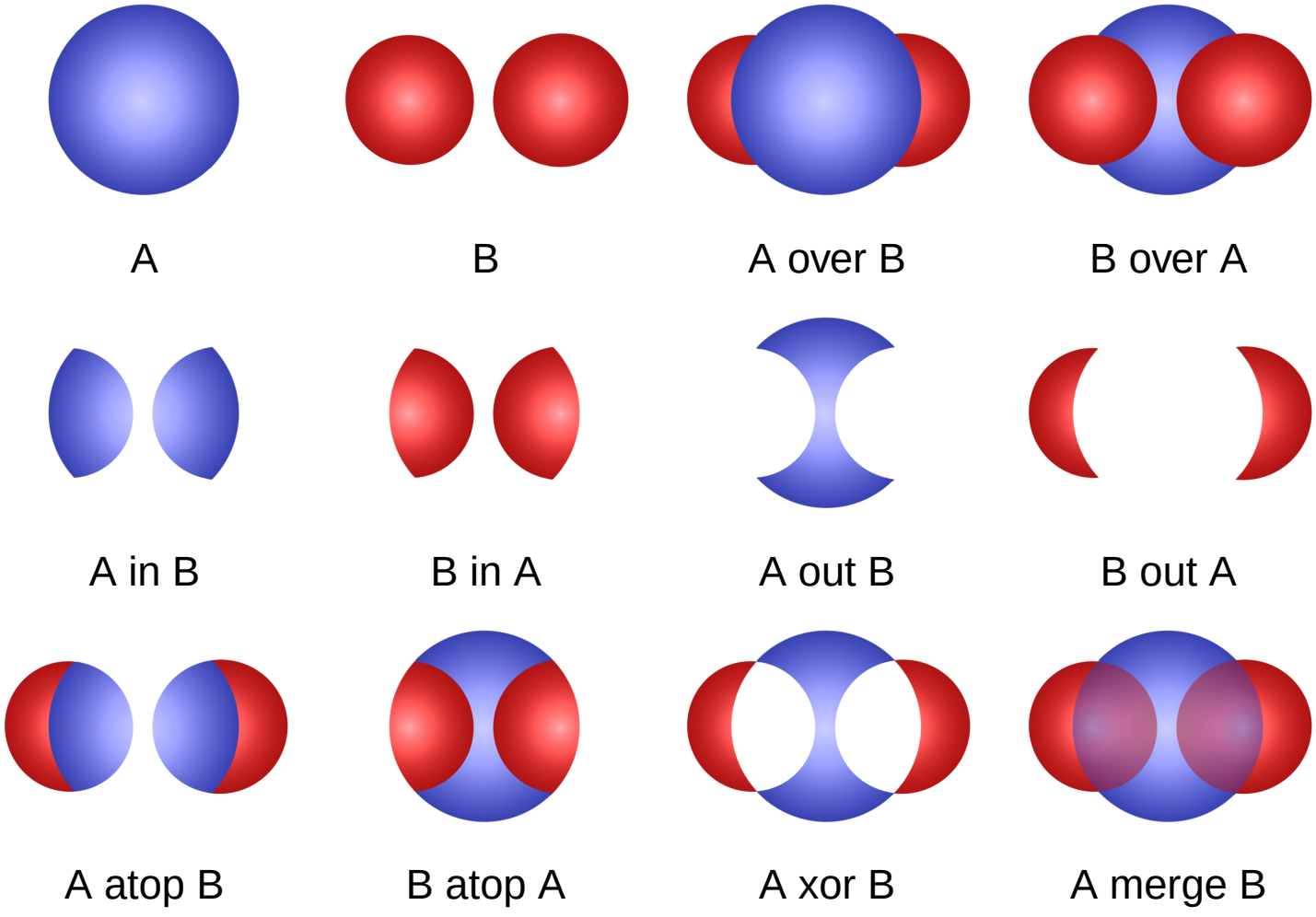


Feature enhancement operators

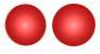
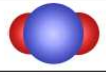
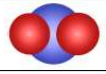
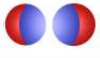
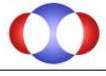
Numerical operators



Compositing operators



Compositing operators

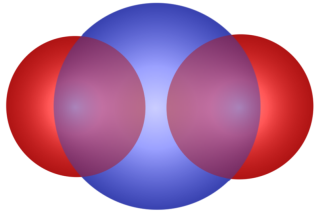
Operations	F_A	F_A	Result
A	1	0	
B	0	1	
A over B	1	$1-a_A$	
B over A	$1-a_B$	1	
A in B	a_B	0	
B in A	0	a_A	
A out B	$1-a_B$	0	
B out A	0	$1-a_A$	
A atop B	a_B	$1-a_A$	
B atop A	$1-a_B$	a_A	
A xor B	$1-a_B$	$1-a_A$	

Compositing (Porter and Duff)

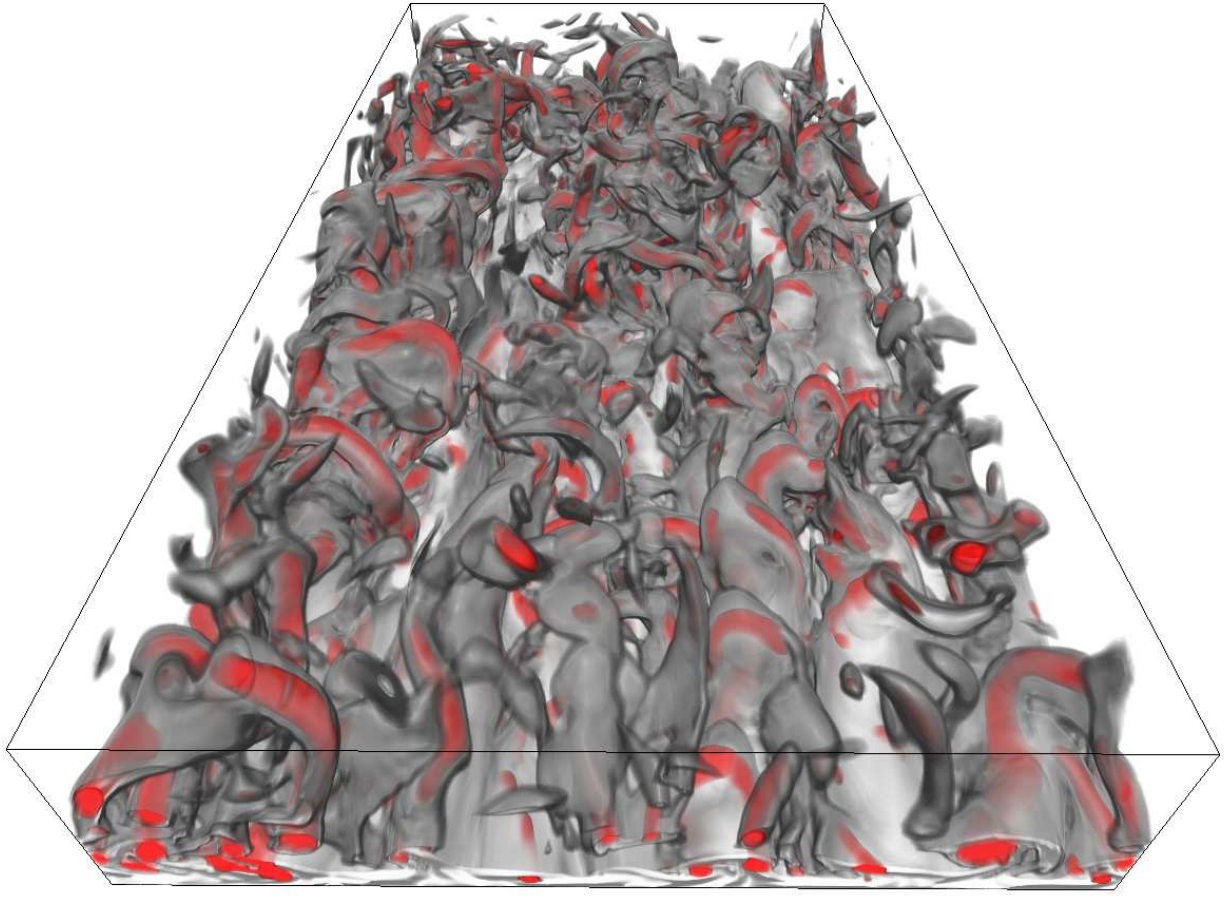
$$A = a_A F_A + a_B F_B,$$
$$C = \frac{c_A a_A F_A + c_B a_B F_B}{A}.$$

Merge (Helgeland)

$$A = 1 - \prod_{i=1}^{n}(1 - a_i),$$
$$C = \left(\sum_{i=1}^{n} c_i a_i\right) / \sum_{i=1}^{n} a_i.$$

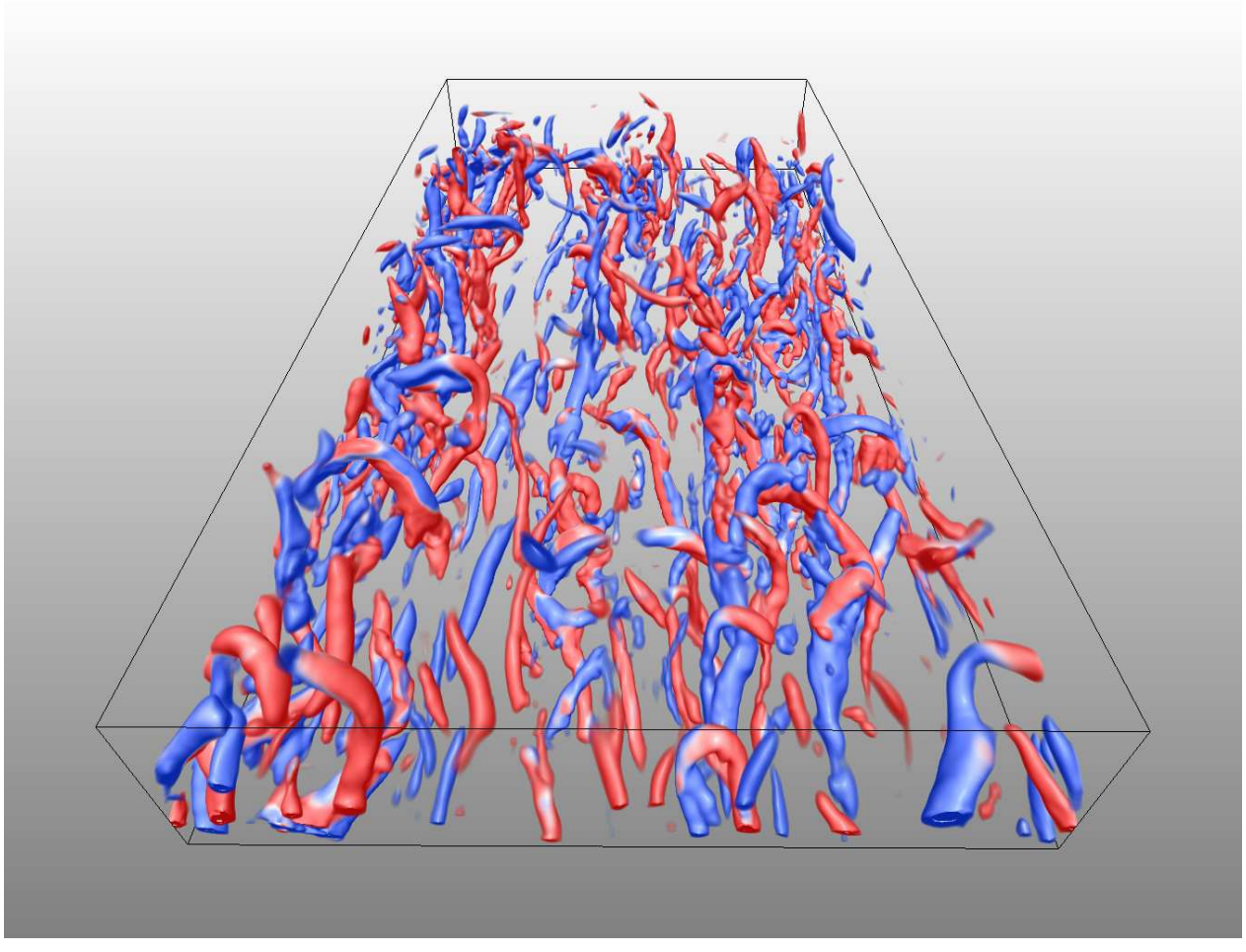


Spatial relationship



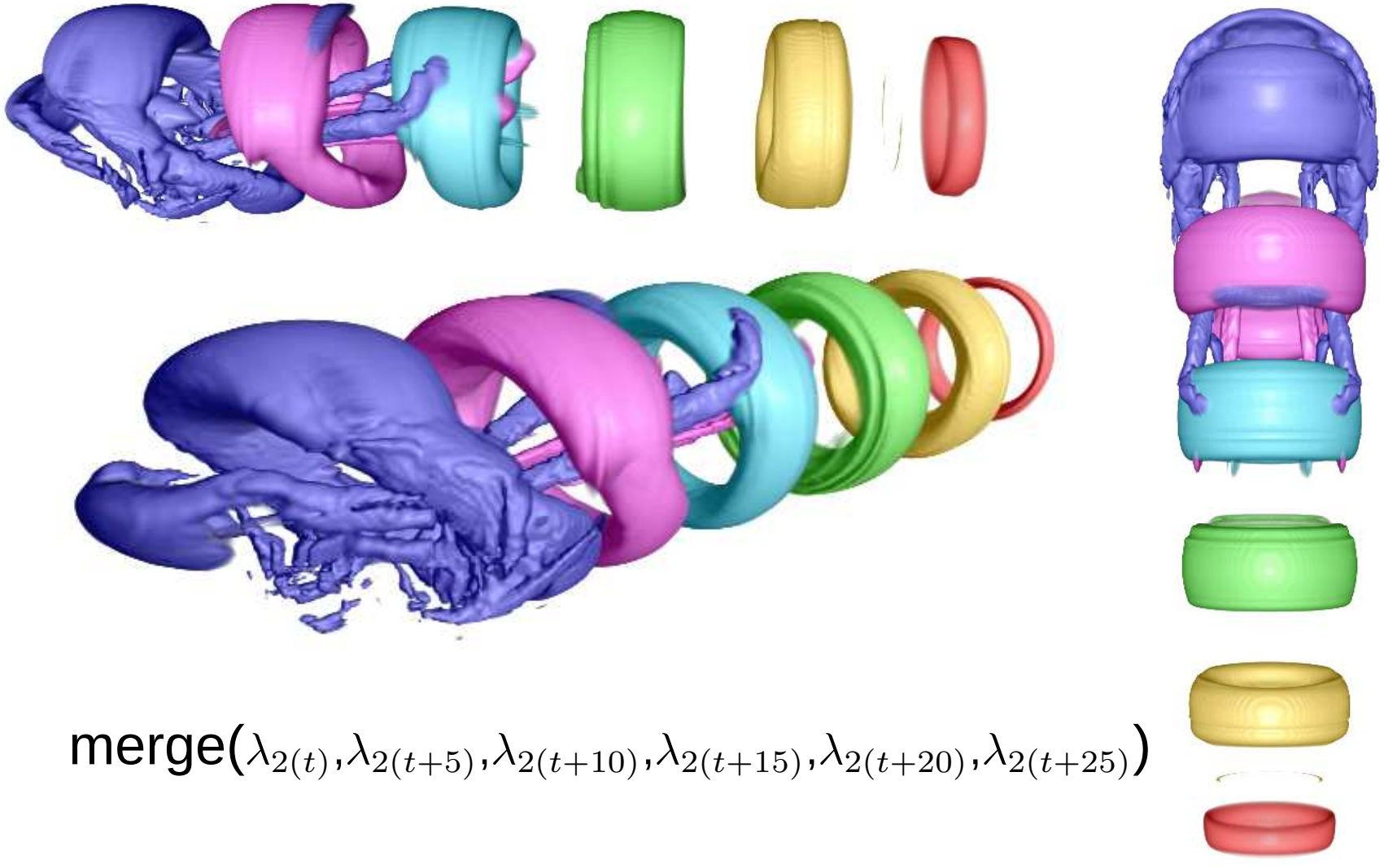
λ_2 **atop** enstrophy ($\|\boldsymbol{\omega}\|^2$)

Spatial relationship



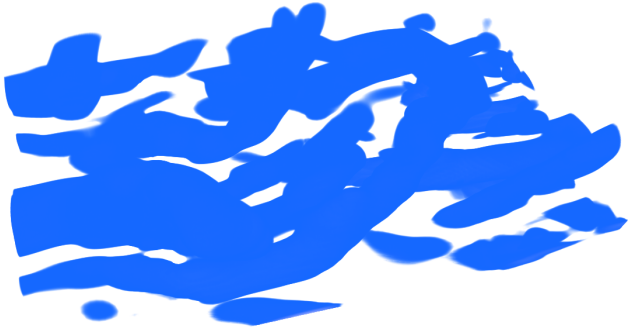
Helicity $(\mathbf{v} \cdot \boldsymbol{\omega})$ **in** λ_2

Spatial-temporal relationship

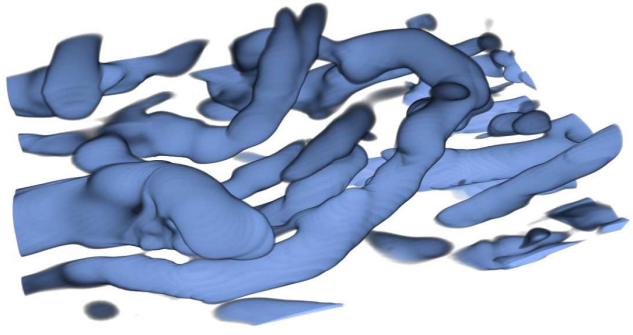


merge($\lambda_{2(t)}$,$\lambda_{2(t+5)}$,$\lambda_{2(t+10)}$,$\lambda_{2(t+15)}$,$\lambda_{2(t+20)}$,$\lambda_{2(t+25)}$)

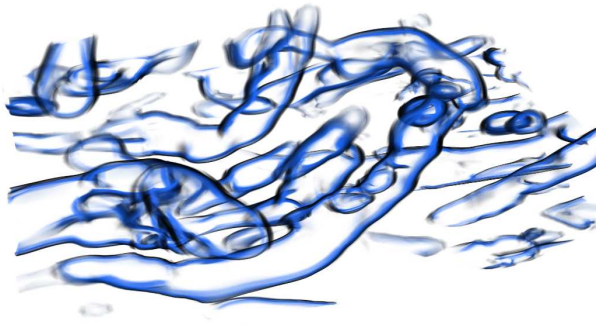
Feature enhancement operators



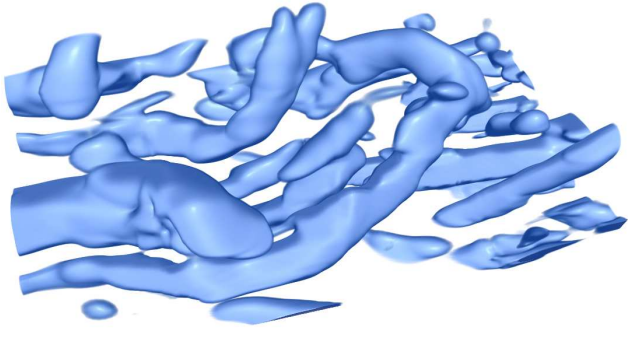
No shading



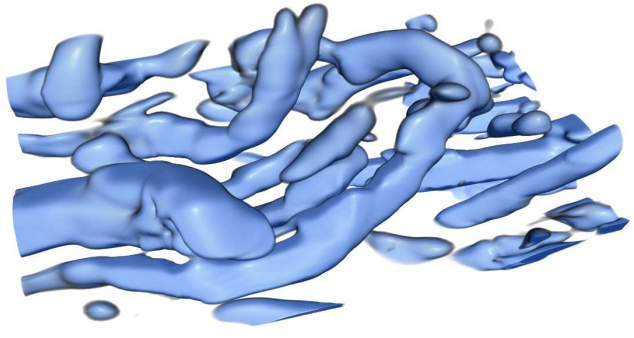
Limb darkening



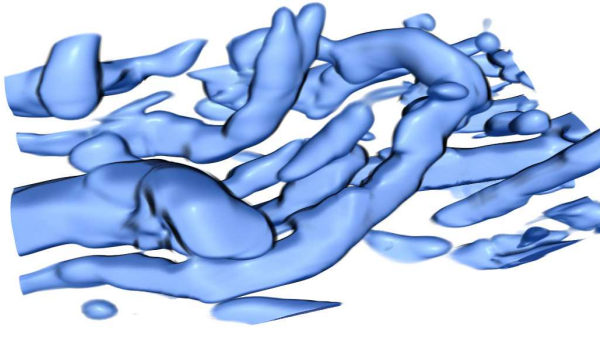
Silhouettes



Shading



Shading + limb darkening



Shading + silhouettes

Gradient-based volume illumination

- Traditional local illumination is based upon a normal vector which describes the orientation of a surface patch.
- In volume rendering no explicit surfaces exists.
- Instead the model is adapted assuming light is reflected at isosurfaces inside the volume data. For a given point $\mathbf{p}$ an isosurface is given as:

$$I(\mathbf{p}) = \{\mathbf{x} | f(x) = f(p)\}$$

with normal

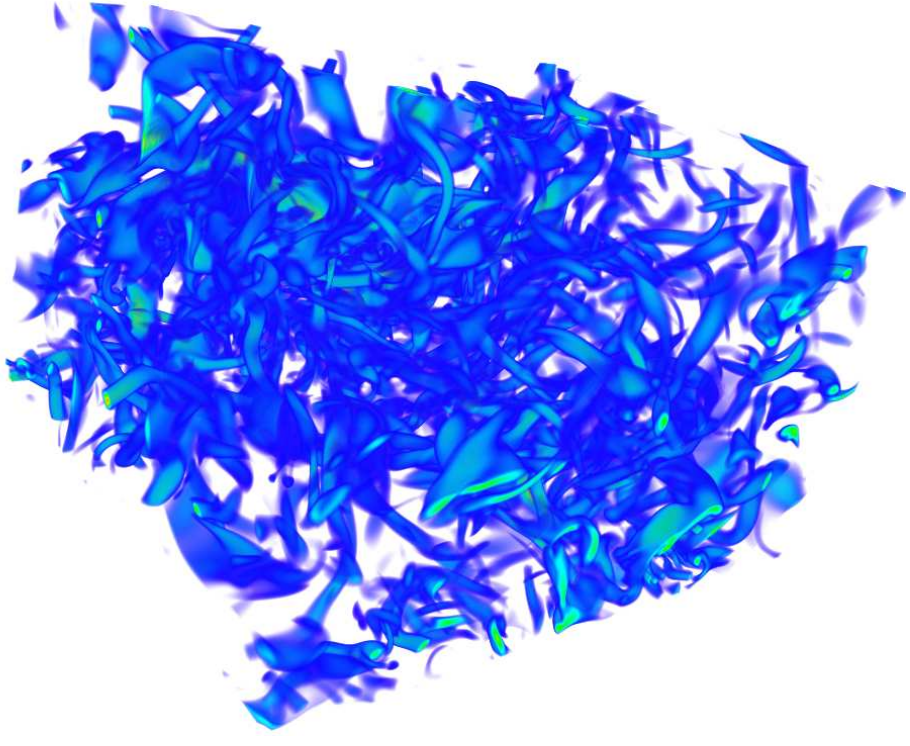
$$\mathbf{n}(\mathbf{p}) = \frac{\nabla f(\mathbf{p})}{\|\nabla f(\mathbf{p})\|}, \quad \nabla f(\mathbf{x}) = (\frac{\partial f(\mathbf{x})}{\partial x}, \frac{\partial f(\mathbf{x})}{\partial y}, \frac{\partial f(\mathbf{x})}{\partial z})$$

- A local illumination model can thus be incorporated into the emission-absorption model:

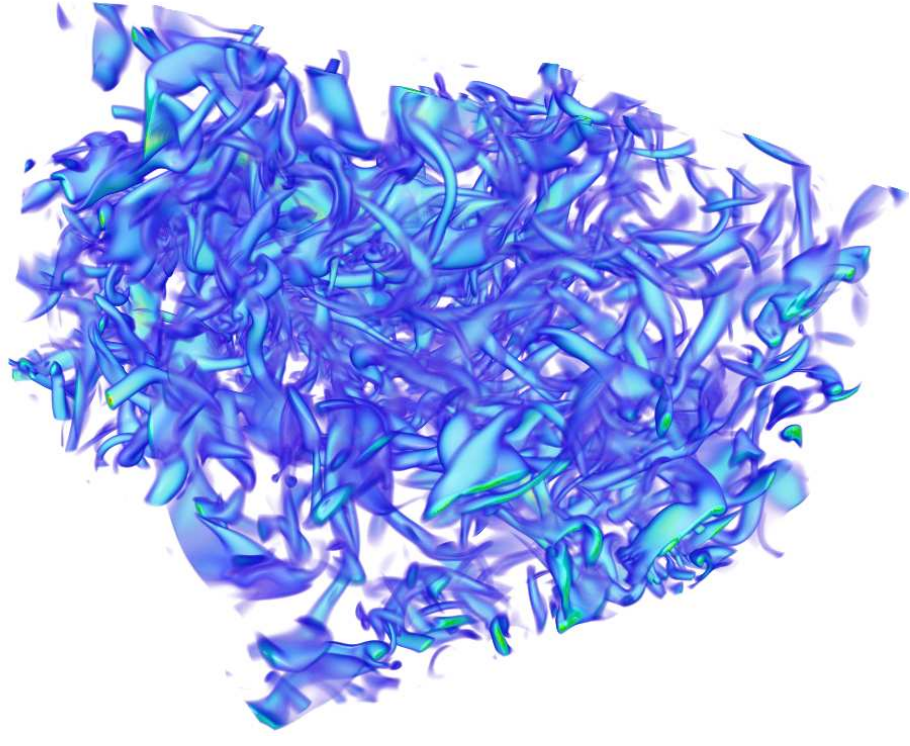
$$\boldsymbol{C}_{volume} = \boldsymbol{C}_{emission} + \mathbf{C}_{illu}, \quad (\mathbf{C}_{illu} = \mathbf{I}_{Phong} | \mathbf{I}_{BlinnPhong})$$

Volume illumination

$$I(D) = \sum_{i=0}^{n} C_i A_i \prod_{j=i+1}^{n} (1 - A_j)$$

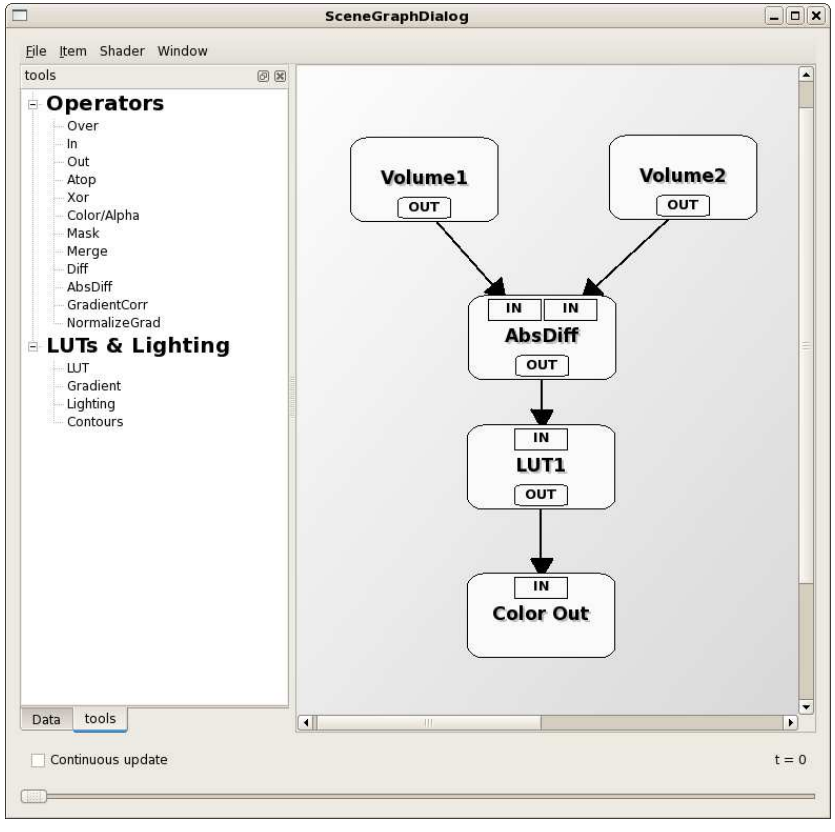


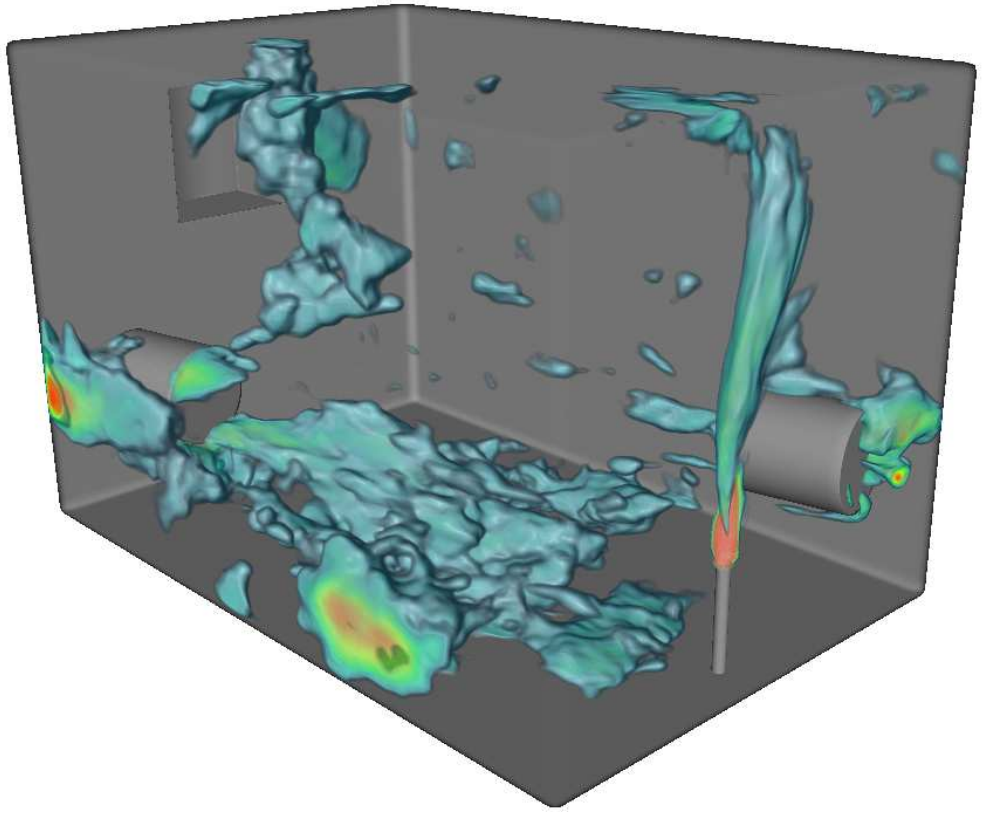
$(C_i = C_{emission})$



$(C_i = C_{emission} + C_{illu})$

Numerical operators





$$|RMS_{t1} - RMS_{t2}|$$

Flexible multifield volume rendering

Benifits of flexible Volume rendering

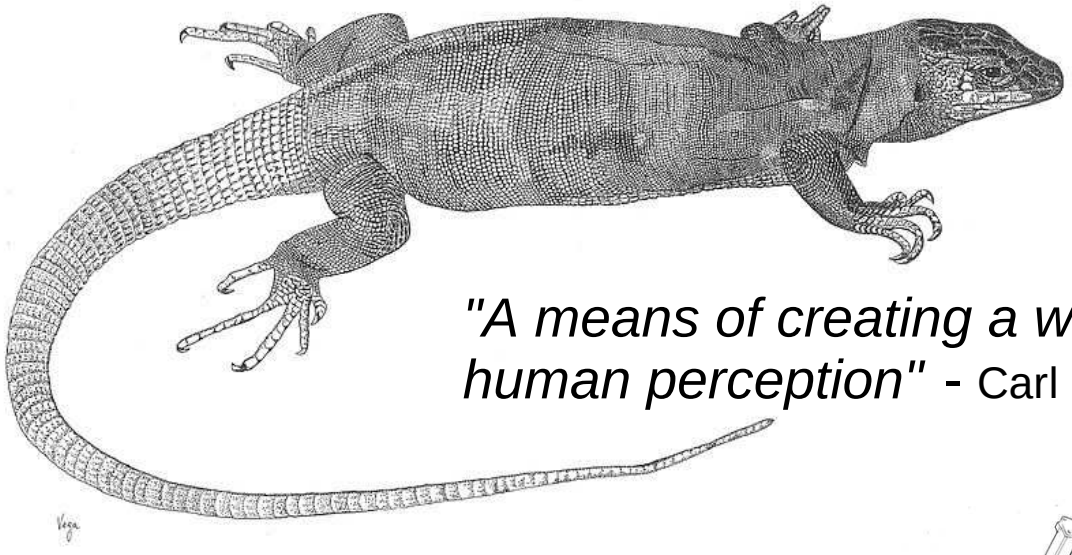
Well-suited for:

- Interactive investigation of 3D data
- Finding spatial and temporal relationships
- Enhancing volume data features
- Interactive post-processing of data

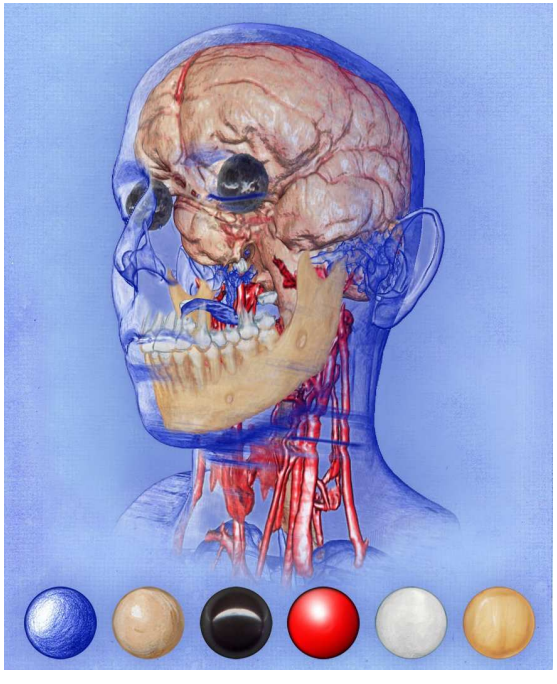
Photorealistic and non-photorealistic rendering

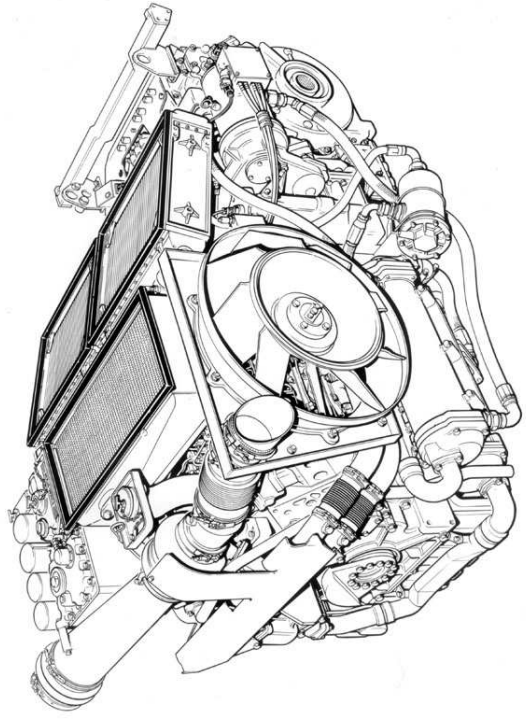
- **Photorealistic rendering**: Goal is to mimic the appearance of reality as closely as possible
  - Global illumination: Ray tracing, radiosity, ..
  - Local illumination: Phong model, Blinn-Phong model
- **Non-photorealistic rendering (NPR)**: The goal of NPR is to go beyond the means of photorealistic rendering and produce images that emphasize important features in the data, such as edges, boundaries, depth and detail.
  - Artistic effects (pen-and-ink drawings)
  - Illustrative effects (technical illustrations)
  - Cartoon effects
  - Scientific visualization

What is NPR?



"A means of creating a work of art that appeals to human perception" - Carl Marshall





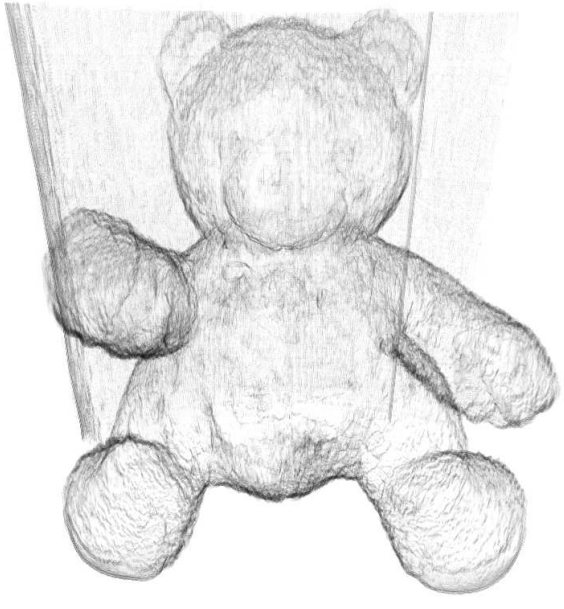
NPR techniques

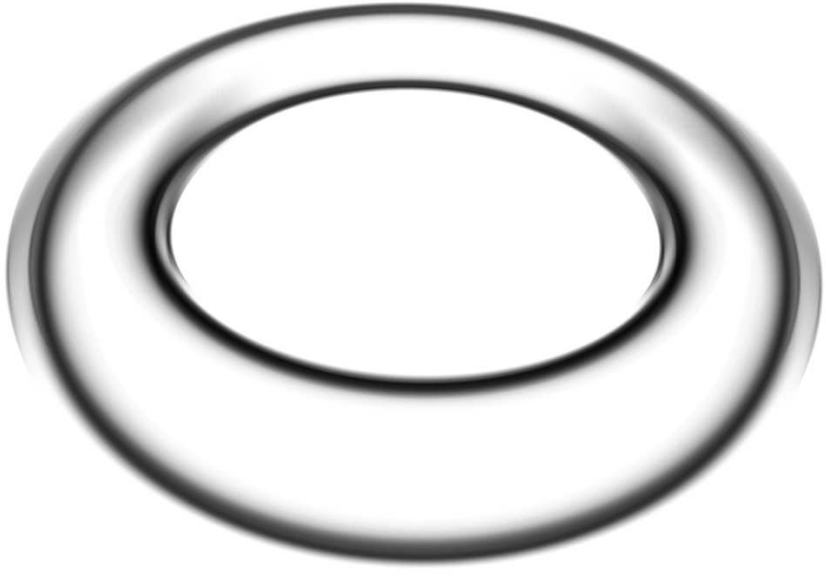
- **Feature line drawing**: Silhouettes (contour lines), Ridge and valley lines, Hatching, stippling
- **Advanced shading and texturing**: Tone shading, cartoon shading, artistic texturing (to convey shape)
- **Focus+context**: Visibility and importance-Driven visualization

Contours

- Contours (or silhouettes) are very important shape cues and are a basic part of almost all illustrations.
- For volumetric data a contour intensity $I_{contours}$ can be evaluated as:

$$I_{contours} = g(\|\nabla f\|)(1 - \|(\mathbf{v} \cdot \mathbf{n}\|)^n$$

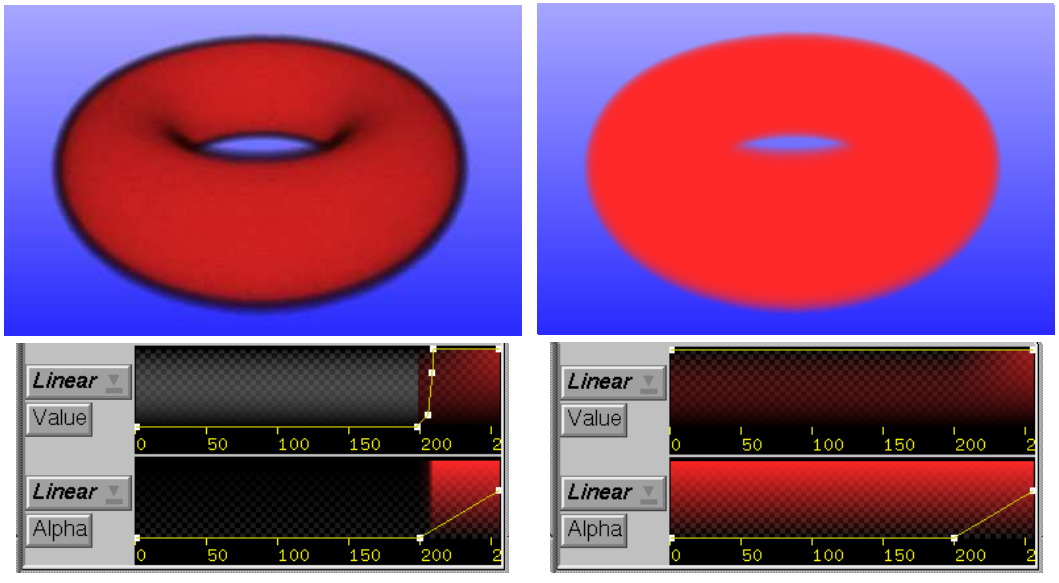


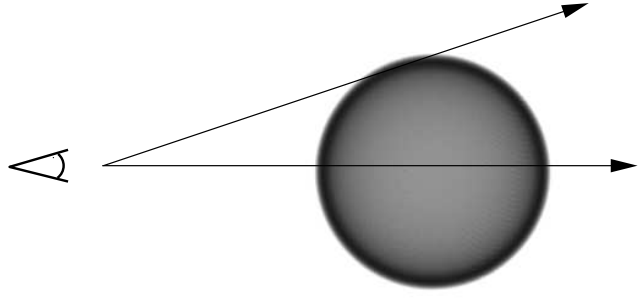


Here, the window function $g(\|\nabla f\|) = \|\nabla f\|$ and $n = 8$

Limb darkening

- Limb darkening is a silhouette technique obtained by manipulating transfer functions only.
- In particualar, by adjusting the **V**alue and **A**lpha values in the HSVA color system.
- Its a computationally efficient technique as it does not need to compute gradients.





[Helgeland et. al. 2004]

Tone shading

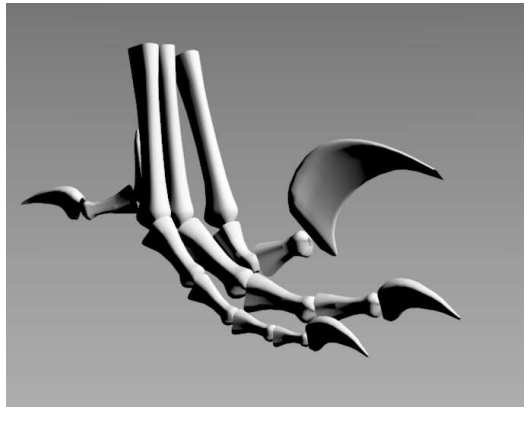
- In traditional diffuse shading black shaded regions hide details. Edge lines can also be difficult to see in these regions

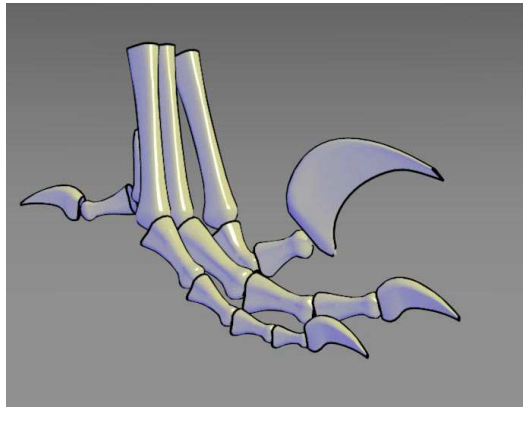
$$I = k_a + k_d \text{max}(0, \mathbf{l} \cdot \mathbf{n})$$

- With tone-based shading model, fine details are visible in the dark shaded regions as well edge lines. The techniques is well suited for illustrations.

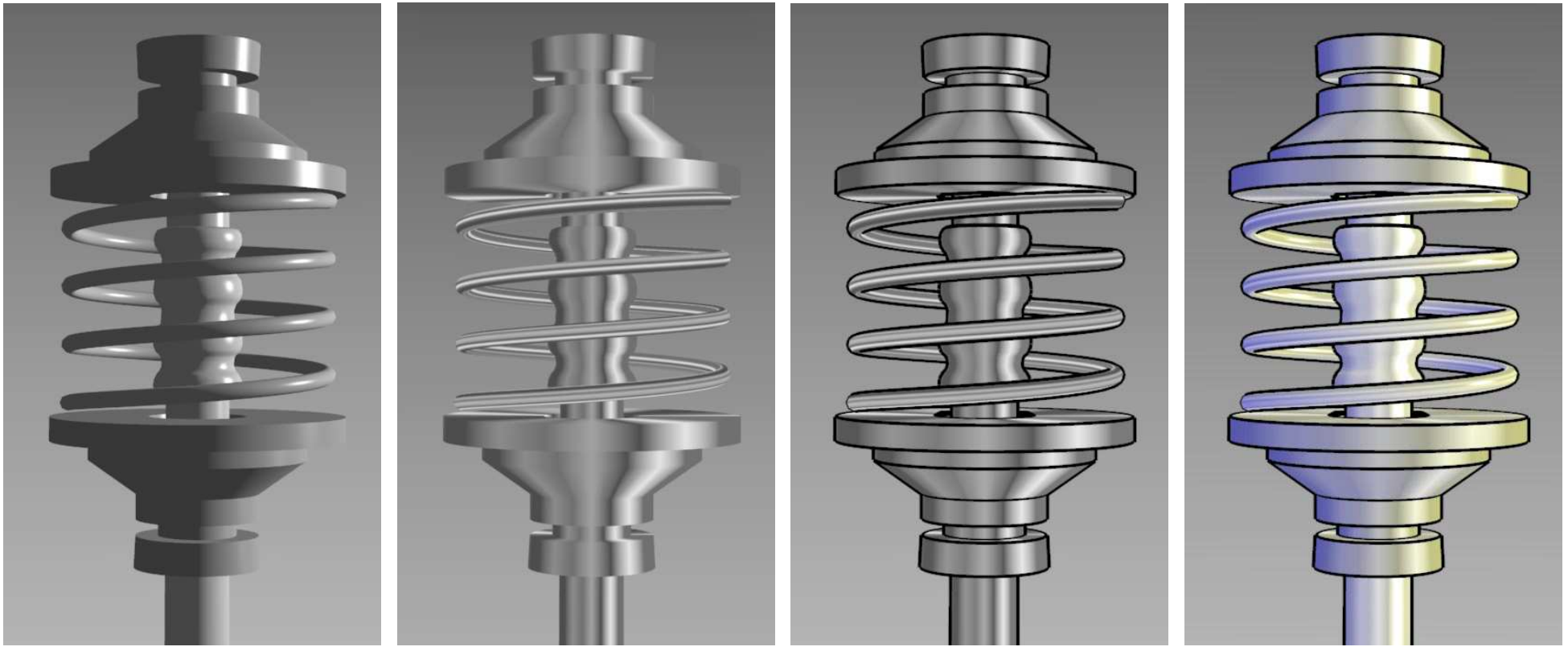
$$I = \left(\frac{1 + (\mathbf{l} \cdot \mathbf{n})}{2}\right)\mathbf{k}_{cool} + \left(1 - \frac{1 + (\mathbf{l} \cdot \mathbf{n})}{2}\right)\mathbf{k}_{warm}$$

$$\mathbf{k}_{cool} = \mathbf{k}_{blue} + \alpha k_d, \quad \mathbf{k}_{warm} = \mathbf{k}_{yellow} + \beta k_d$$





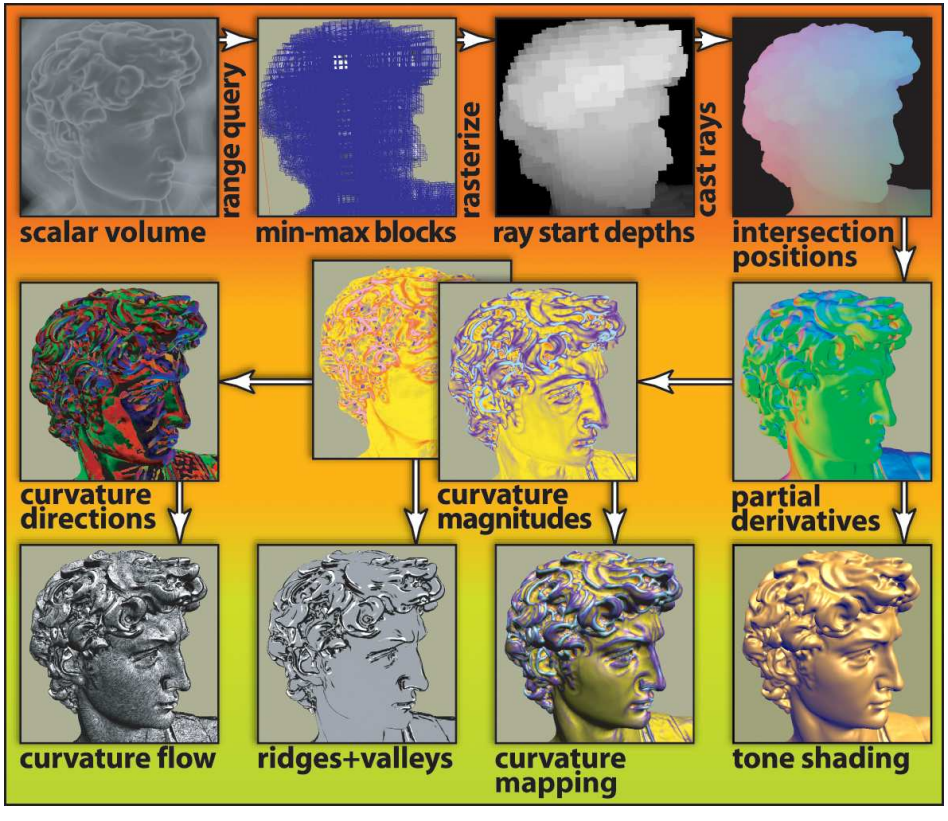
Tone shading



Phong shading, metal shading, metal shading with edge lines, Tone shading with edge lines

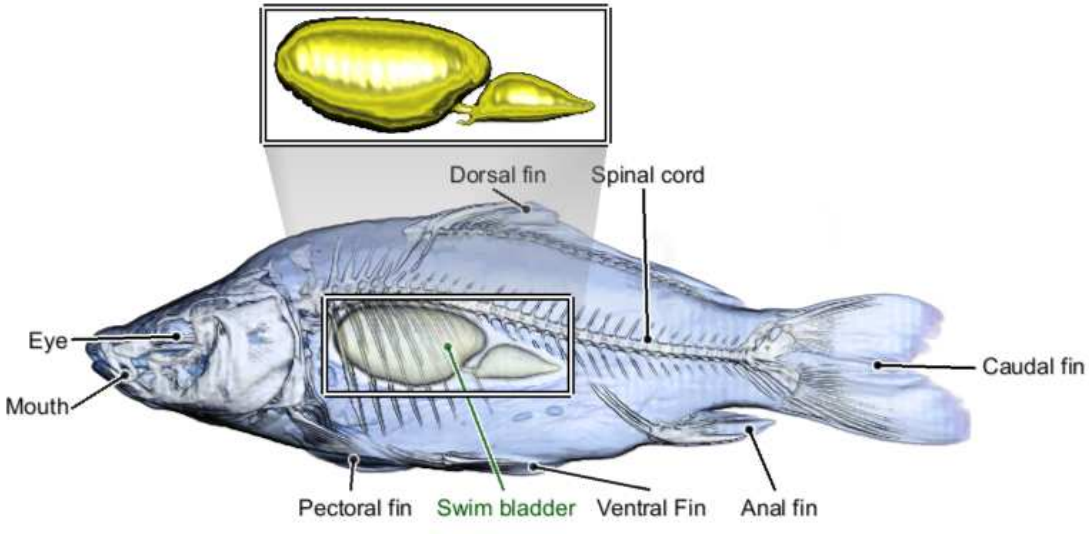
[*Gooch et. al. 1998*]

NPR example



[*Hadwiger et. al. 2005*]

NPR example



[*Bruckner et. al. 2005*]

Colors

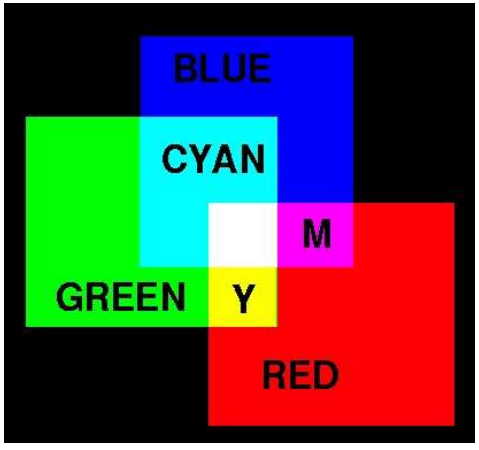
Several color models are available in computer graphics. In visualization the most important are:

1. RGB (red, green, blue) which is an additive color model
2. CMY (cyan, magenta, yellow) is a subtractive model
3. HSV (hue, saturation, value) is most suitable for visualization

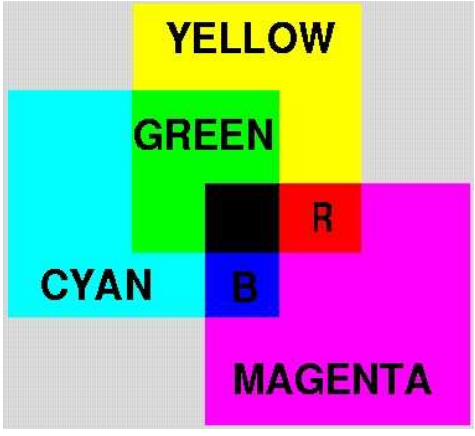
R-C, G-M and B-Y are complementary colors. They are related as

$$\begin{pmatrix} C \\ M \\ Y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

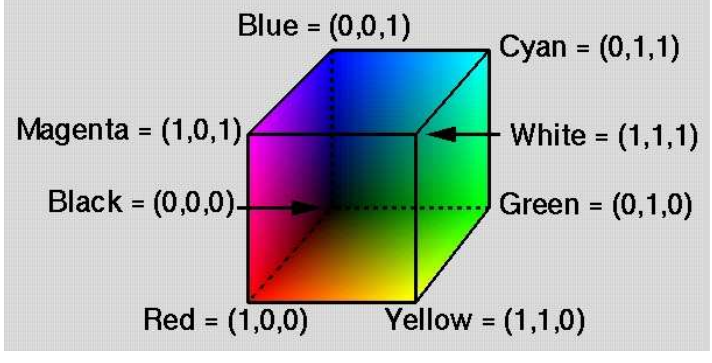
The RGB, CMY and HSV models



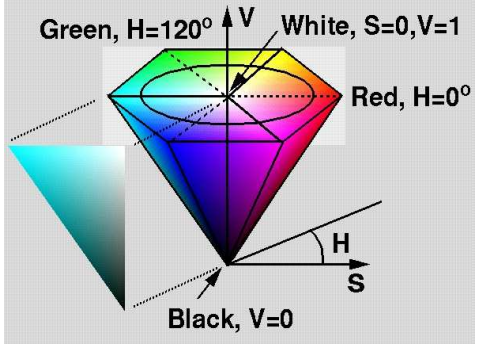
The RGB model is additive



The CMY model is subtractive



The RGB cube



The HSV cone