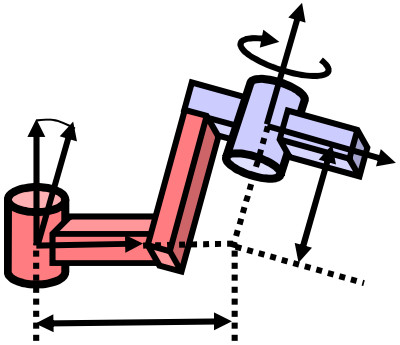
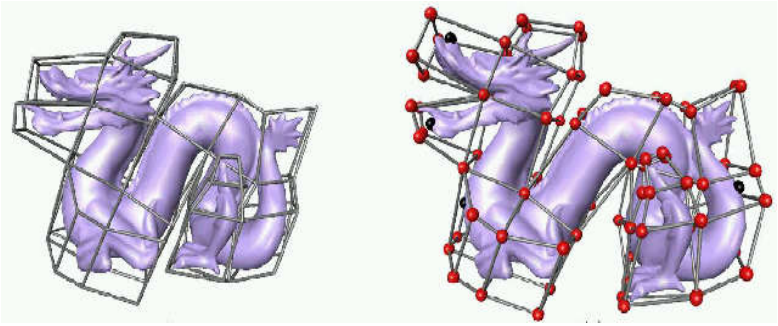
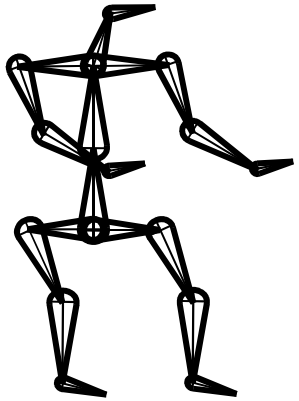
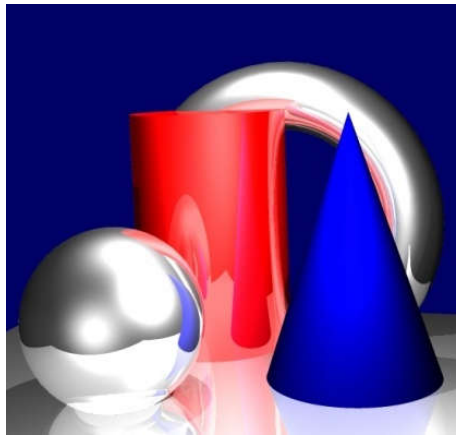
Computer Graphics 2021

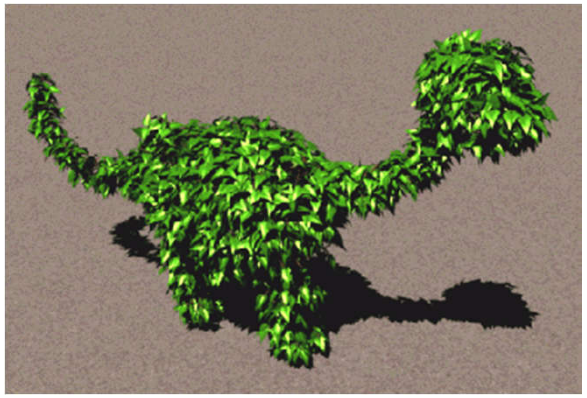
Martin Hering-Bertram











Overview

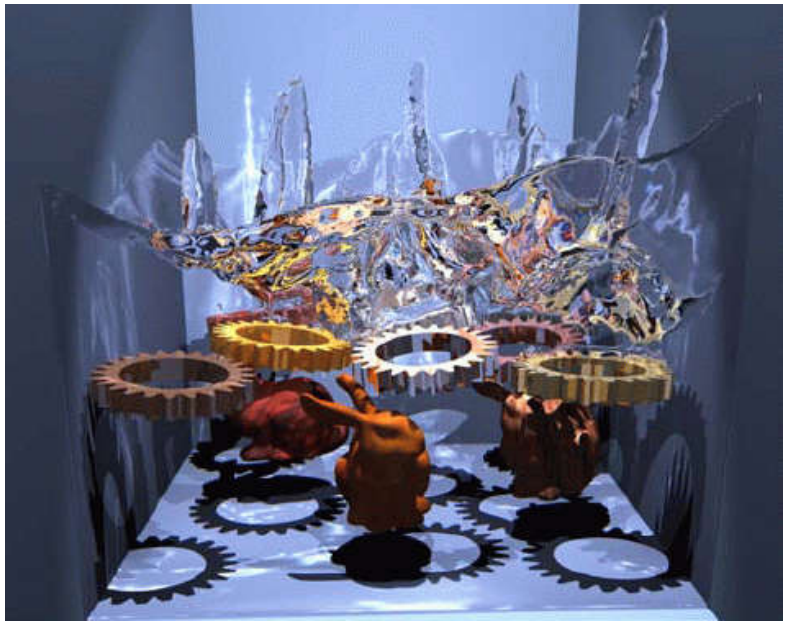
Modeling and Rendering Natural Phenomena

Advanced Rendering

- Rendering Pipeline (Open GL)
- Raytracing / Radiosity
- Photon Map

Simulation

- Simulated Acoustics
- Plants (L-Systems)
- Rigid Body Dynamics
- Fluid Dynamics (Stable Fluids)



Carlson et al., Siggraph 2004

Overview

Photo-realistic Rendering Examples

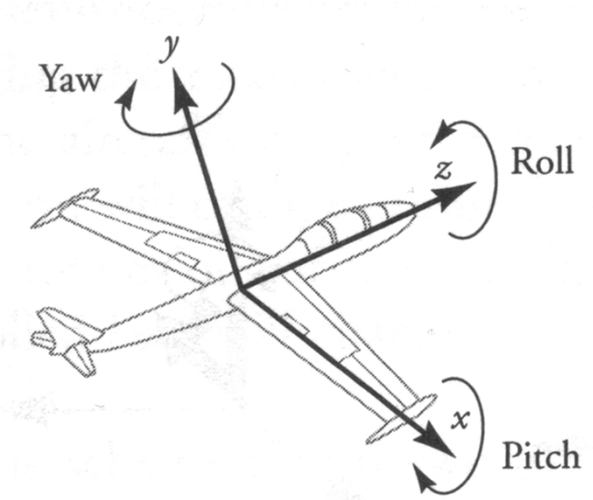
Caustics under water [Jensen/Christensen, Siggraph 98]

Overview

Computer-animated Processes

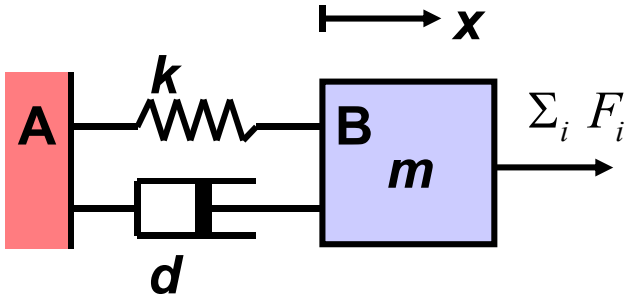
Transformations and Orientations

- Coordinate Transforms
- Homogeneous Coordinates
- Euler Angles
- Quaternions



Kinematics and Dynamics

- Inverse Kinematics
- Mechanical Systems
- Collision Handling



Overview

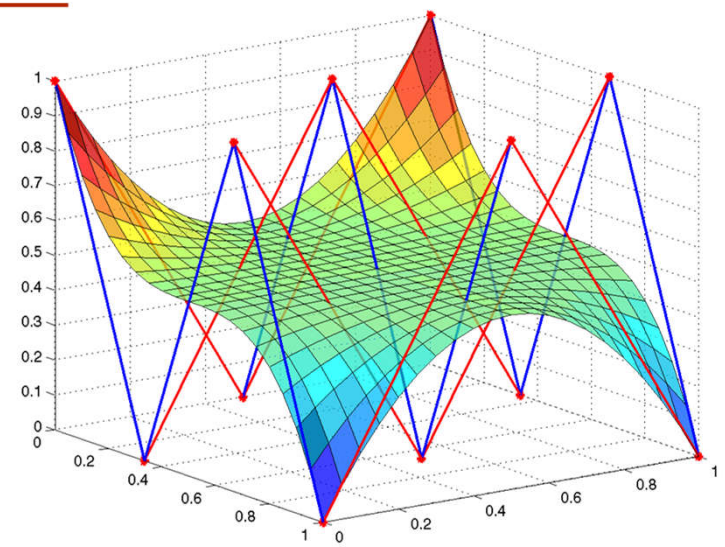
Modeling Fundamentals

Piecewise Polynomials

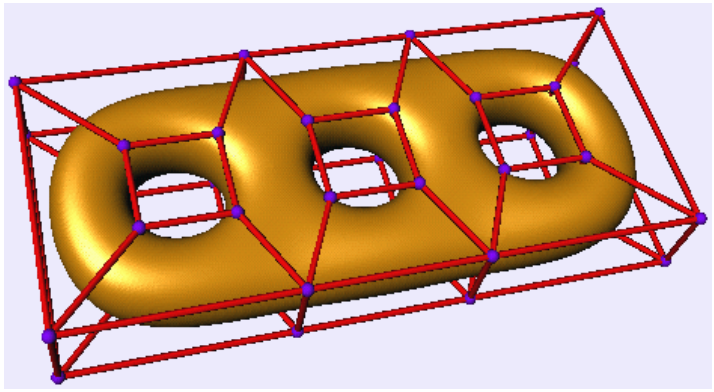
- Bézier Surfaces
- B(asis) Splines
- Interpolation

Advanced Modeling

- Subdivision Surfaces
- Least Squares Fitting
- Variational Design

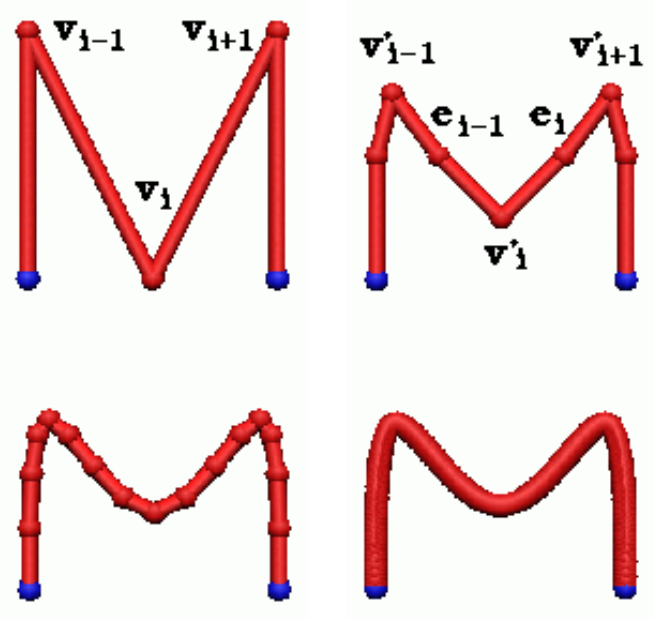


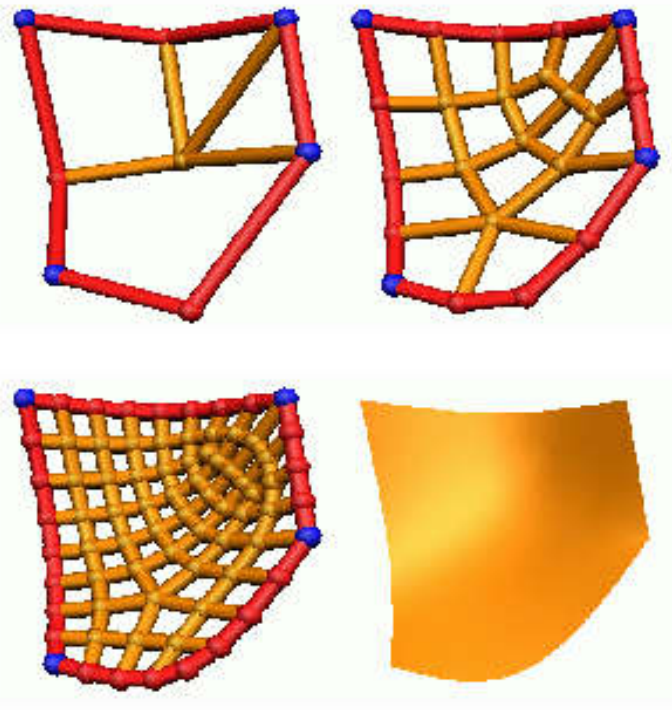
Bézier Surface (top) and Catmull-Clark Subdivision



Overview

Subdivision -- Piecewise Smooth Curves/Surfaces from Polygons/Meshes



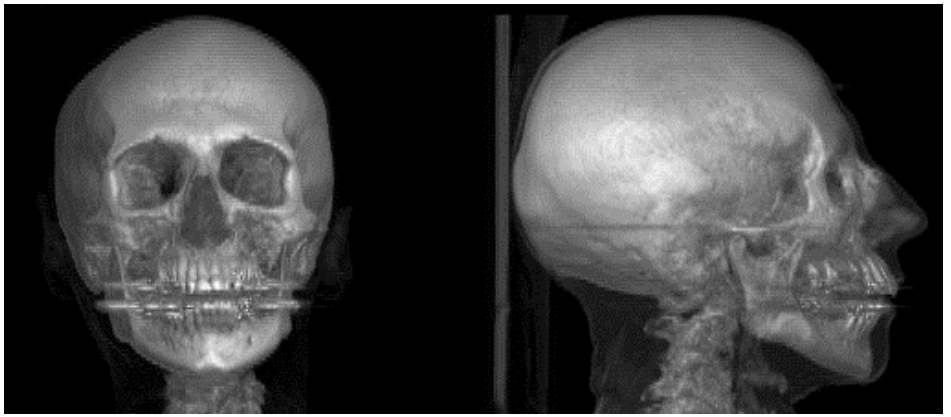


Overview

Computer Vision and Data Visualization

Image Processing

- Edge Detection
- Filtering / Smoothing
- Feature Extraction
- 3D Reconstruction



Volume Rendering

Scalar and Vector Fields

- Isosurfaces (Contouring)
- Volume Rendering
- Topology Extraction

Overview

Simulated Acoustics

Fourier Transform

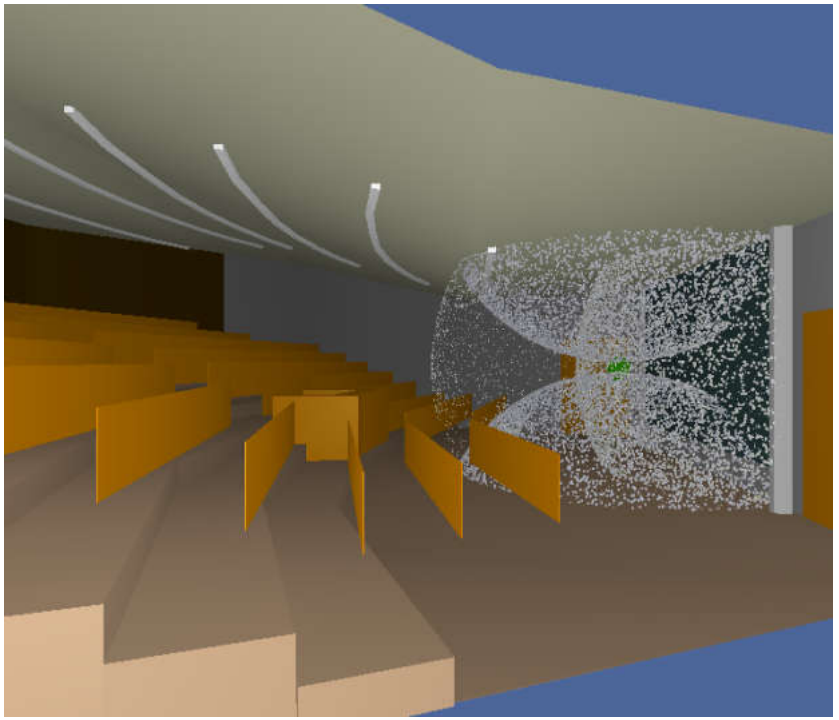
- Continuous and Discrete FT
- Convolution Theorem
- Signal Processing

Geometric Acoustics

- Image Source Method
- Absorption Coefficients
- Particle Tracing

Waves

- Wave Equation
- Complex Exponentials

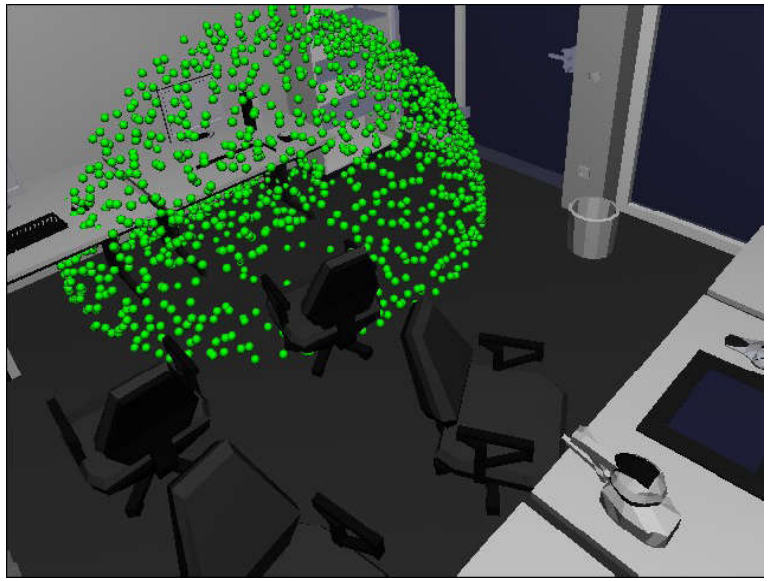


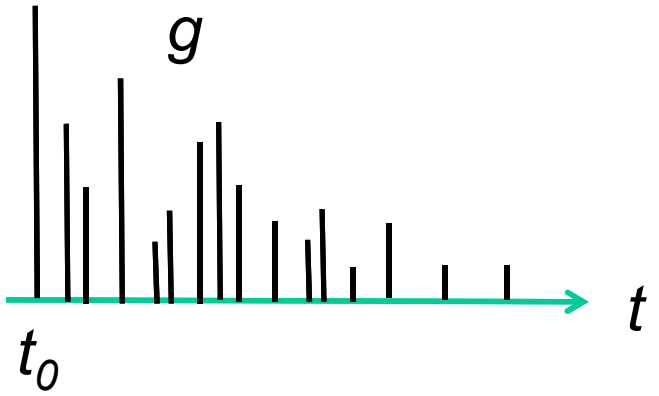
Simulated room acoustics using particles

Overview

Signal Processing

Digital filters may be computed from simulated room acoustics, tracing reflection paths from a source to a listener position.
A pulse response filter sums up all the echoes with their time delays and decreased amplitudes, due to absorption.





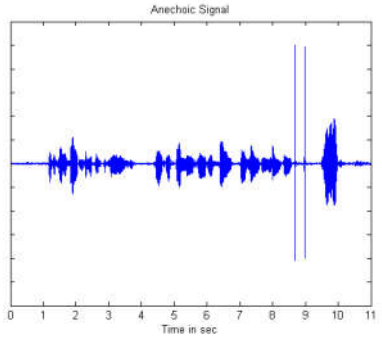
pulse response filter

Overview

Convolution -- Computed by the Fast Fourier Transform (FFT)

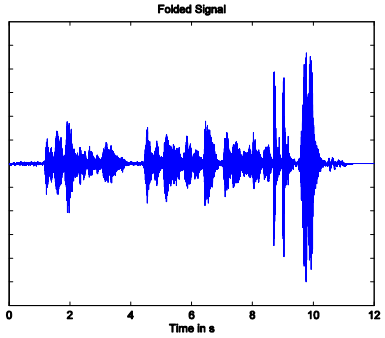
The **convolution** $f * g(t)$

represents the filtering operation in the time domain, where all echoes of *f* (according to their positions in *g*) are summed up.



f_{in} → g → $f_{out} = f_{in} * g$

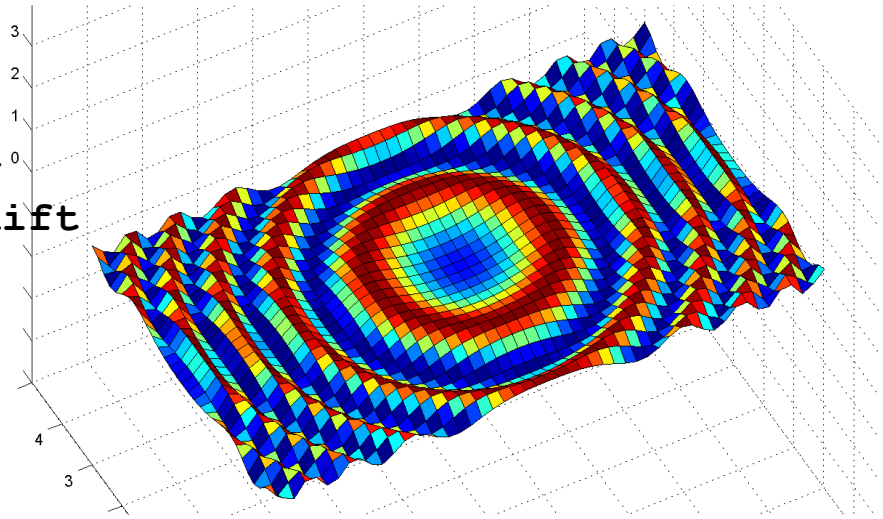
signal processing
by convolution



Overview

Regular Grids (Matlab Example)

```
x = -5:.2:5;
y = -3:.2:3;
[X,Y] = meshgrid(x, y); % matrix with x- and y-coordinates

D = X.^2 + Y.^2;        % distance field to center
F = .2 * exp( i*D);     % complex exponential field (cos + i*sin)
surf( X, Y, real(F));

% animate waves by phase shift
for phi = 0:.1:100  % phase shift
   F1 = exp( -i* phi) * F;
   surf( X, Y, real(F1));
   axis equal
   drawnow
end
```

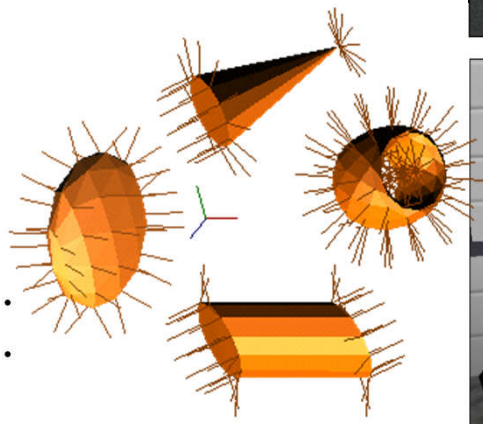


Overview

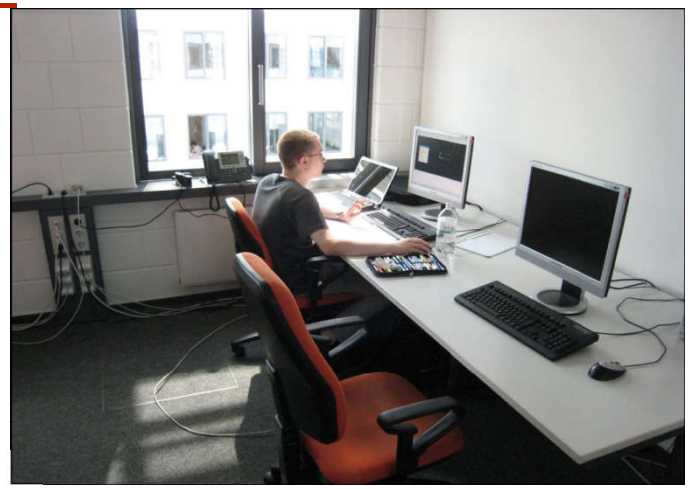
Triangulated Surfaces

The three-dimensional generalization of a polygon is a triangle mesh.
To define a triangle mesh, we need a list of points (3×n-matrix) and for each triangle three point indices.

Example:



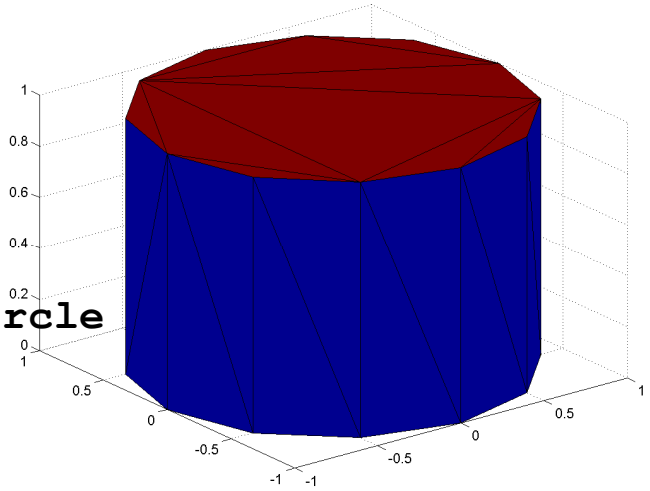
```
P = [0   1   0   1;...
     0   0   1   1;...
     0   0   0   0];
T = [1 2 3; 3 2 4];
% square with two triangles
```



Overview

Triangulated Surfaces

A simple way to generate triangles from a point set is by the convex hull. Matlab example:

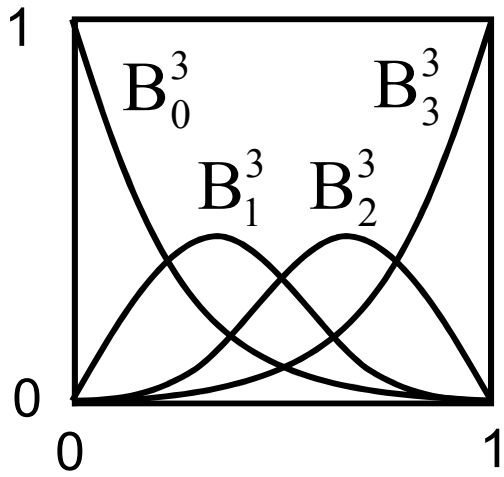


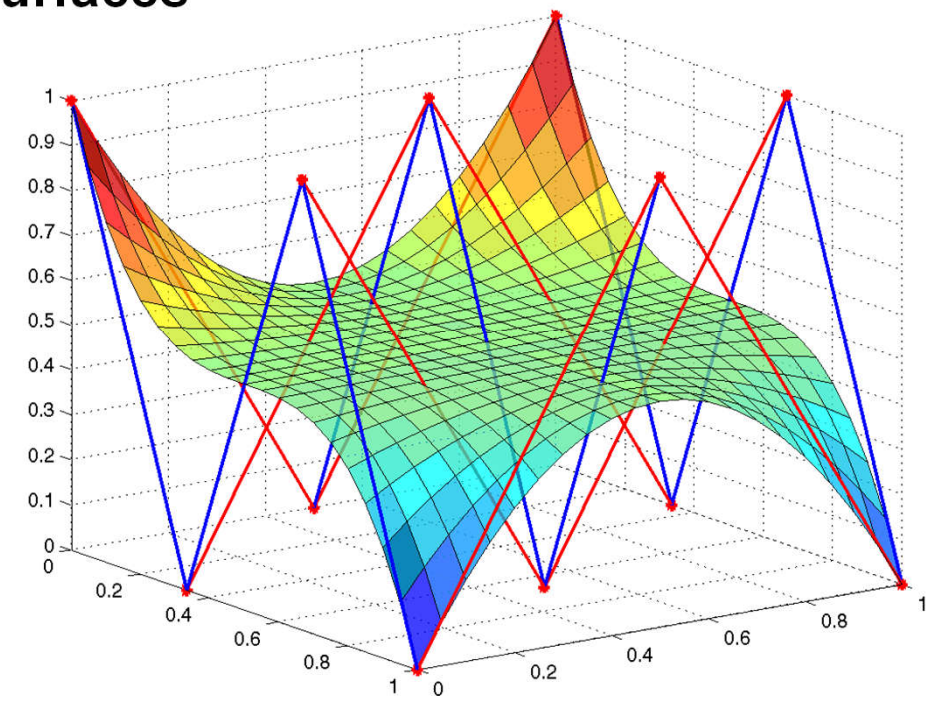

```
phi = 0:pi/6:2*pi;        % sampling for circle
P = [cos(phi), cos(phi);...
     sin(phi), sin(phi);...
     zeros(size(phi)),  ones(size(phi))];
T = convhull( P');        % triangle indices for CH
tr = TriRep(T, P');       % structure with T and P
trisurf(tr);              % draw it
```

In contrast to quadrilateral grids (try "cylinder" in Matlab), triangle meshes are much more flexible.

Overview

Example: Freeform Bézier Surfaces





$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

$$X(t) = \sum_{i=0}^{n} b_i B_i^n(t),$$

$$X(s,t) = \sum_{i=0}^{n}\sum_{j=0}^{m} b_{ij} B_i^n(s) B_i^m(t) ,$$

Useful Tools

- **C++ programming language**
- **OpenGL 3.2** (open graphics library for rendering)
- OpenAL (open audio library)
- OpenCV (computer vision library for image processing)
- QT (GUI programming)
- Matlab / Octave for development and testing
- WebGL (web graphics library)
- **LaTeX** (used for documentation)

1. Rendering Pipeline

Why 3D Computer Graphics?

- Computer Animation (video games, cartoons, special effects)
- Design (arcitecture, planning (production plants), CAD/CAM)
- Virtual Reality (VR)
- Scientific Visualization (quality analysis, medical imaging, etc.)
- Geographic Information Systems (GIS)
- …

1. Rendering Pipeline

Open Graphics Library (OpenGL) Versions

- 1982: IrisGL (SGI)
- 1992: OpenGL 1.0 (SGI)
- 1995: Direct3D 1.0 (Microsoft)
- 2001: OpenGL ES 1.0 (embedded systems)
- 2003: GLSL (shading language for GPU-implementation)
- **2009: OpenGL 3.2 (use this compability profile)**
- 2010: OpenGL 4.1 (re-design depreciating "old" functions)
- 2015: OpenGL ES 3.2
- 2016: Vulkan (by Khronos Group, very difficult to learn)
- 2018: OpenGL 4.6 (independent from vulkan)

1. Rendering Pipeline

The Rendering Pipeline

- Coordinate Transforms,
- Central and Parallel Projections
- Rasterization
- Visibility Culling
- Illumination and Shading
- Texture Mapping

Suggested Reading:

Dave Shreiner, et al., **OpenGL Programming Guide (Red Book),** Addison-Wesley, 2003, http://glprogramming.com/red/

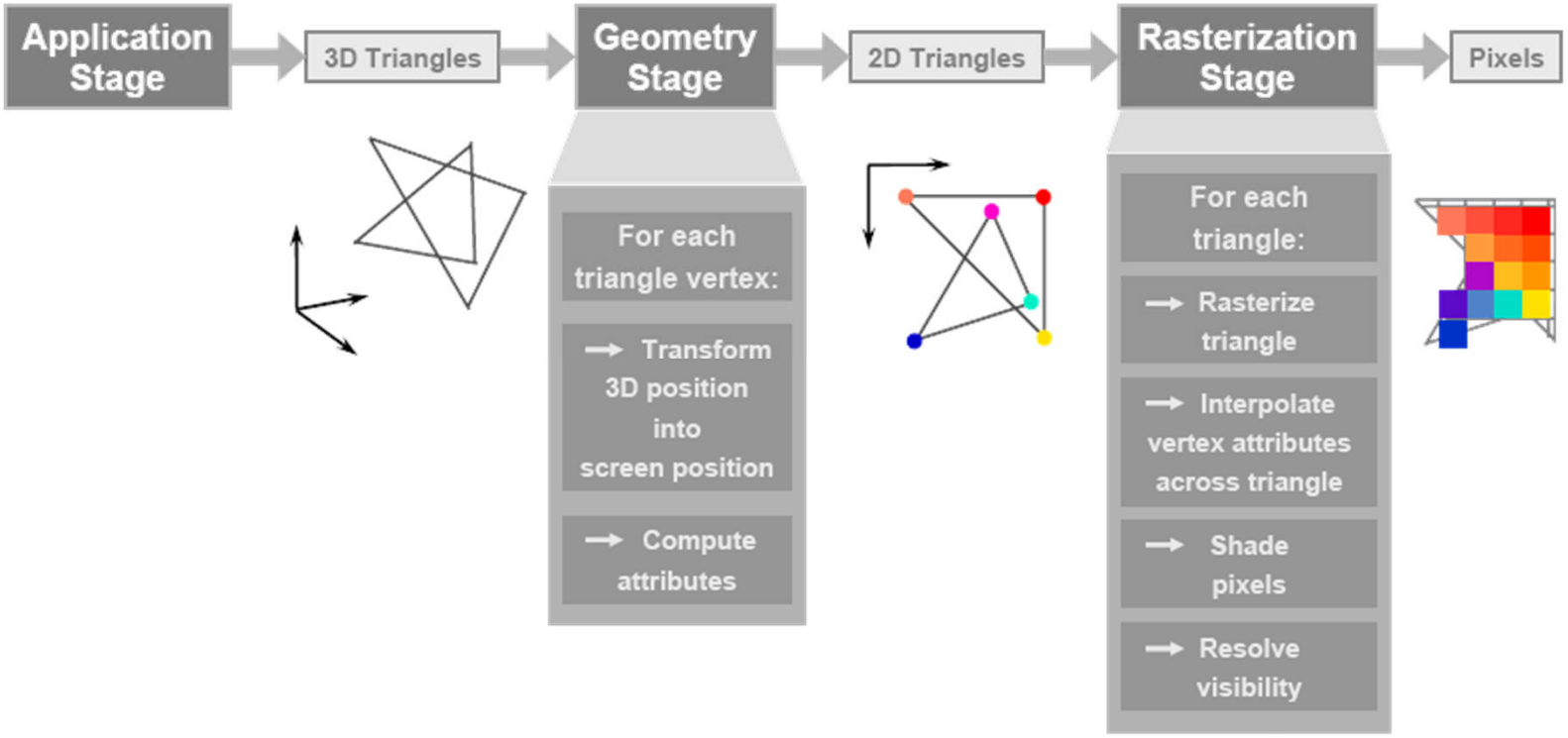
1. Rendering Pipeline

To make the display process as efficient as possible, the individual steps are organized in a ***rendering pipeline***:

- transform object coordinates into world coordinates
- transform world coordinates into camera coordinates (Modelview Matrix in Open GL)
- perspective (central) or parallel projection (Projection Matrix)
- clipping and perspective division (homogeneous coordinates)
- mapping into window coordinates
- rasterization stage (determine color for each pixel)

1. Rendering Pipeline

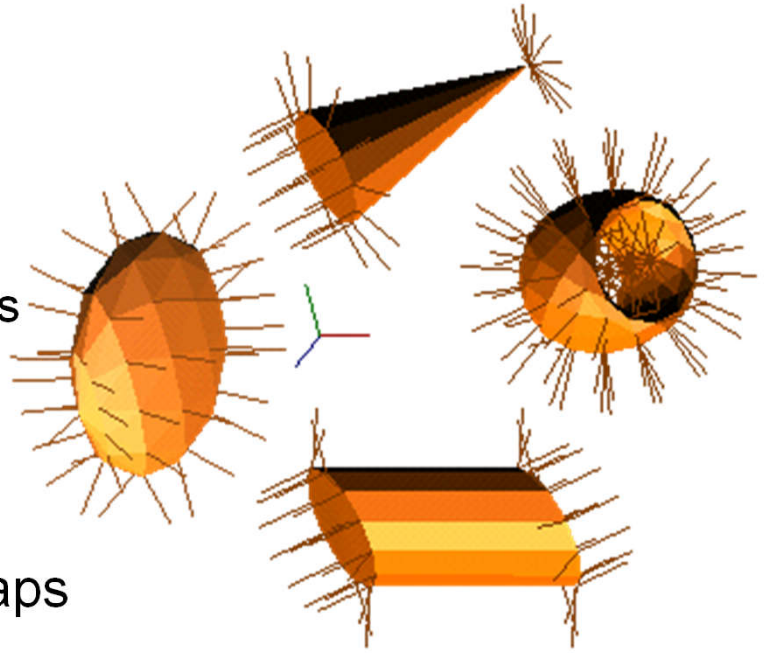
Overview



1. Rendering Pipeline

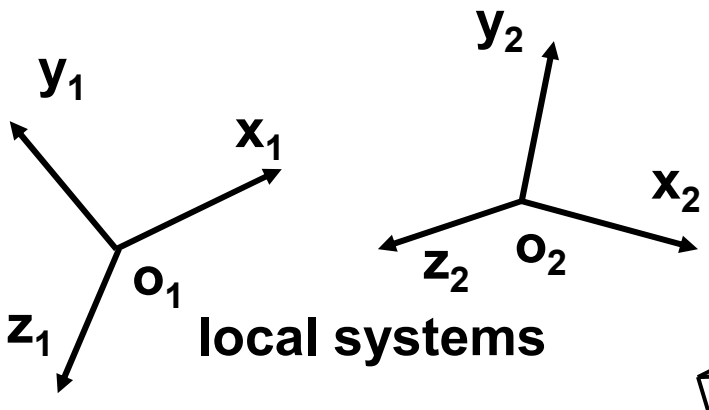
Triangulated Surface Geometry

- points
  - 3D coordinates
  - normal vectors
  - color index and texture coordinates
- triangles
  - 3 indices of points
- material colors and texture bitmaps

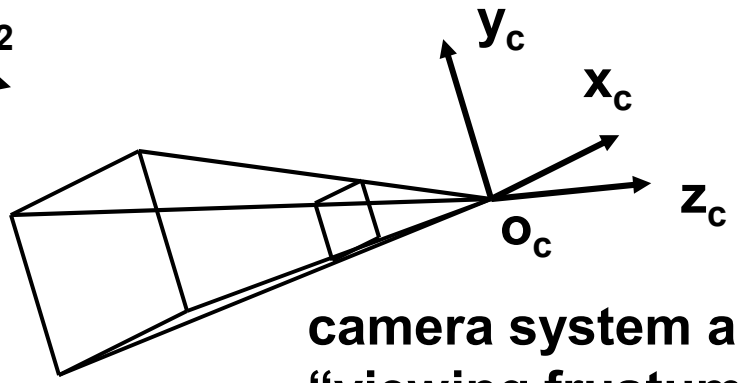


1. Rendering Pipeline

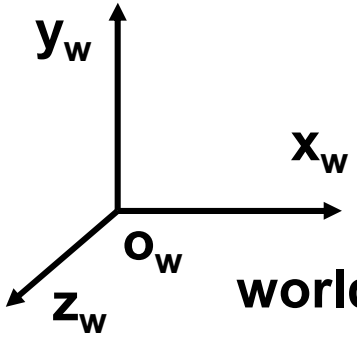
Affine Transformations $p' = Rp + t$



local systems



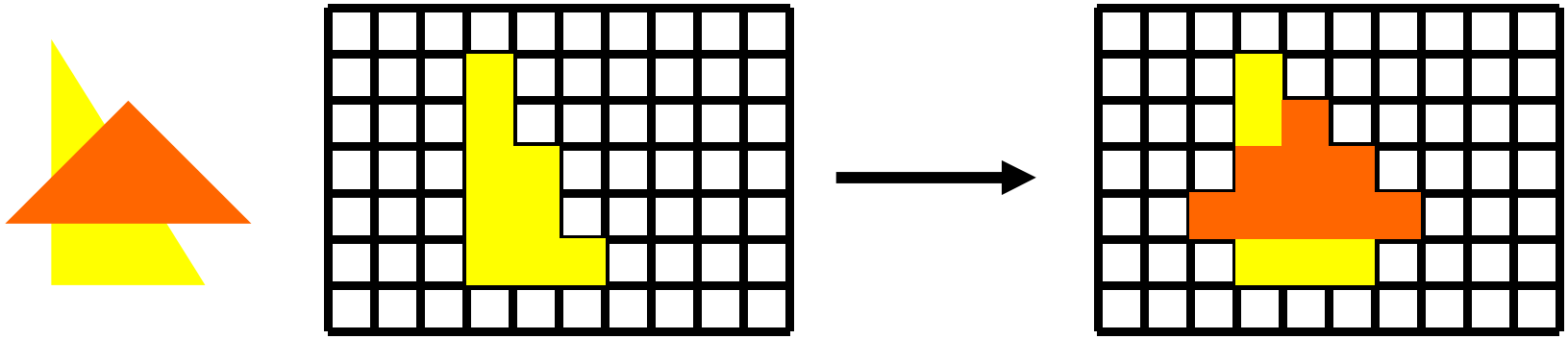
camera system and
"viewing frustum"



world system

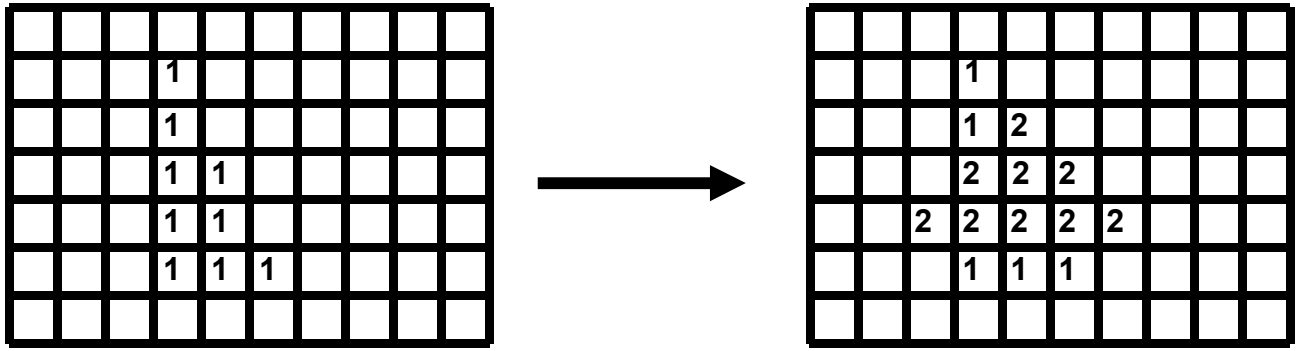
1. Rendering Pipeline

Rasterization, **Visibility Culling**, Viewport-Transformation



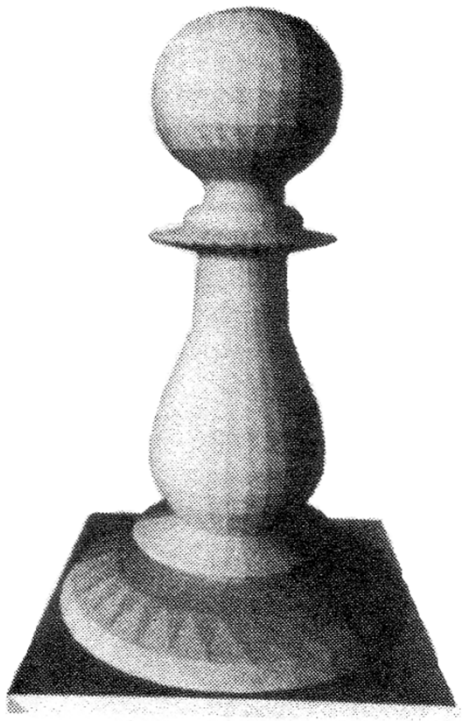
frame buffer (RGB-Farbe)

z-buffer (z-coordinate = depth)

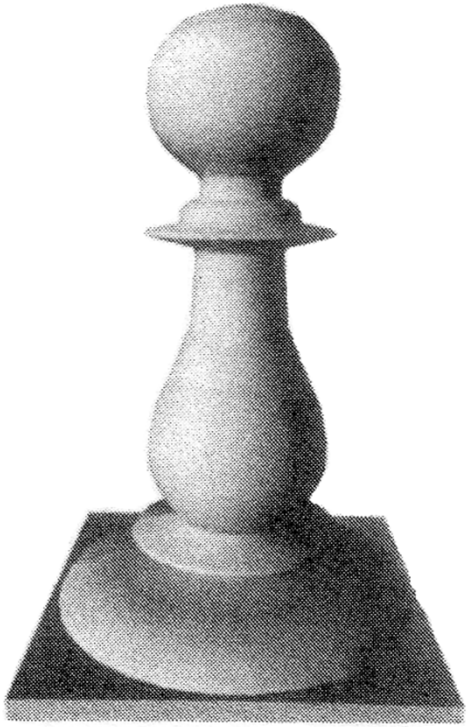


1. Rendering Pipeline

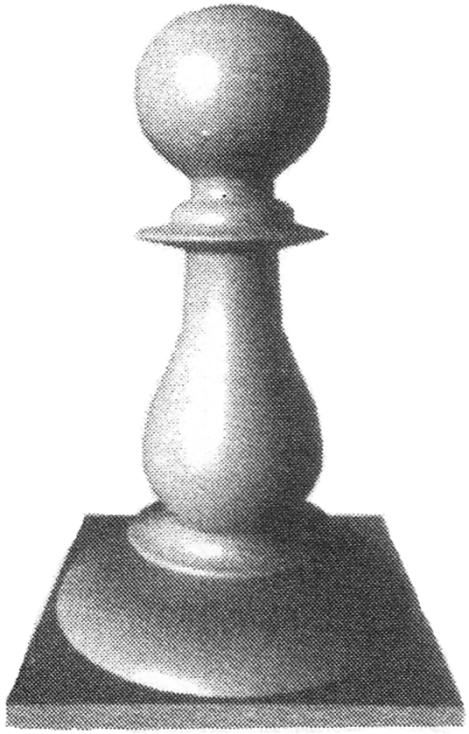
Illumination, **Shading**



Flat Shading (triangle normals)



Gouraud Shading (interpolated color)

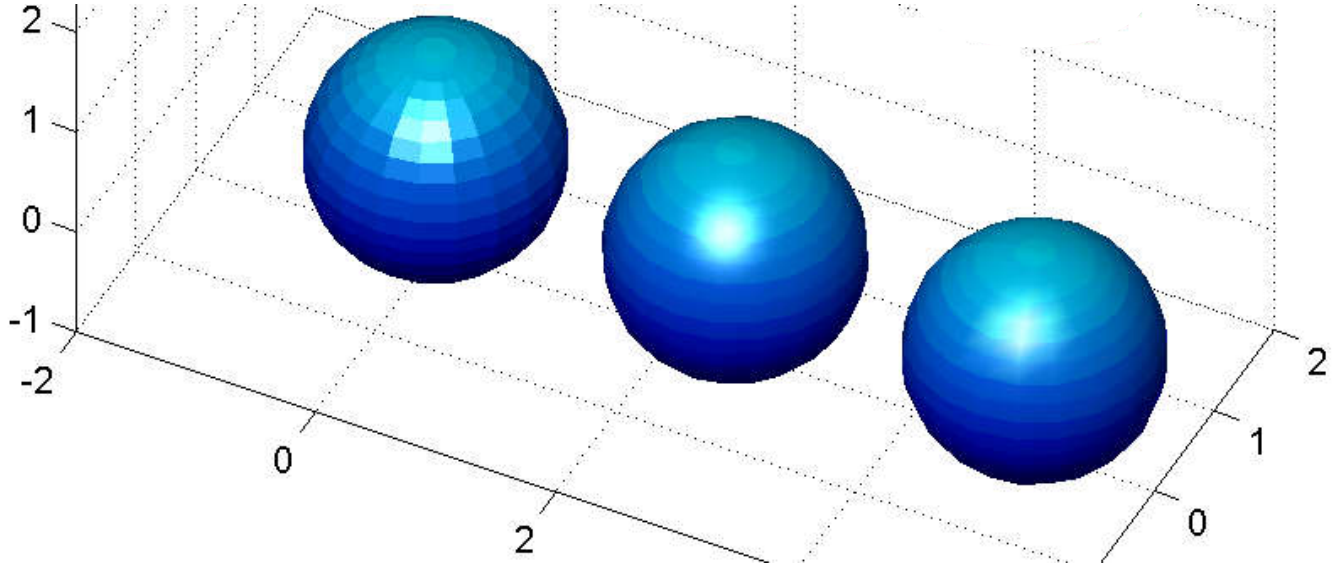


Phong Shading (interpolated normals)

1. Rendering Pipeline

Flat, Phong, and Gouraud Shading (Matlab Example)

```
[X,Y,Z] = sphere(20);
surf(X,Y,Z, 'FaceLighting', 'flat', 'Lines', 'none');
surf(X+2.5,Y,Z, 'FaceLighting', 'phong', 'Lines', 'none');
surf(X+5,Y,Z, 'FaceLighting', 'gouraud', 'Lines', 'none');
camlight right; % light source
```



1. Rendering Pipeline

OpenGL Example (C++ Programmer's View)

1. depth test (z-Buffering) for visibility
2. shading Gouraud of flat (or use your own GPU shader)
3. light sources
4. modelview matrix (camera position)
5. projection matrix (parallel / central, viewing frustum)
6. viewport transform
7. geometry + material color

1. Rendering Pipeline

```
1  glEnable( GL_DEPTH_TEST); // switch on z-buffer
   glDepthFunc( GL_LESS);
   // set background color
   glClearColor(0.8, 0.8, 1.0, 1.0); // bright blue
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
2  glShadeModel( GL_SMOOTH); // Gouraud shading
   //glShadeModel( GL_FLAT); // flat shading
3  // light positions and colors
   GLfloat LightPosition1[4] = { 10, 5, 10, 0}; // XYZW
   GLfloat LightPosition2[4] = { -5, 5, -10, 0};// XYZW
   GLfloat ColorRedish[4] = { 1.0, .8, .8, 1};  // RGBA
   GLfloat ColorBlueish[4] = { .8, .8, 1.0, 1}; // RGBA
```

1. Rendering Pipeline

❸

```
glEnable( GL_LIGHTING); // use lighting
glLightModeli( GL_LIGHT_MODEL_TWO_SIDE, 1);//front+back
// define and switch on light 0
glLightfv( GL_LIGHT0, GL_POSITION, LightPosition1);
glLightfv( GL_LIGHT0, GL_DIFFUSE, ColorRedish);
glLightfv( GL_LIGHT0, GL_SPECULAR, ColorRedish);
glEnable( GL_LIGHT0);
// define and switch on light 1
glLightfv( GL_LIGHT1, GL_POSITION, LightPosition2);
glLightfv( GL_LIGHT1, GL_DIFFUSE, ColorBlueish);
glLightfv( GL_LIGHT1, GL_SPECULAR, ColorBlueish);
glEnable( GL_LIGHT1);
```

1. Rendering Pipeline

4
```
glMatrixMode( GL_PROJECTION);
// all subsequent matrix op's define camera projection

glLoadIdentity(); // reset matrix to identity,
  // otherwise multipy all with existing matrix

glOrtho( -15, 15, -10, 10, -20, 20); // orthogonal
  // projection (xmin xmax ymin ymax zmin zmax)
//glFrustum( -10, 10, -8, 8, 2, 20); // perspective p.

glEnable(GL_NORMALIZE); // this is necessary when using
  // glScale to keep surface normals at unit length
```

1. Rendering Pipeline

❺
```
glMatrixMode( GL_MODELVIEW);
// all subsequent matrix operations define model view
glLoadIdentity(); // reset the current modelview matrix
glTranslated( 0 ,0 ,-10.0); // move 10 units backwards
  // in z, since camera is at origin
glScaled( 5.0, 5.0, 5.0); // scale objects
double alpha = 30.0; int w = 400, h = 300;
glRotated( alpha, 0, 3, 1); // rotation angleXYZ
  // glMultMatrixd( matrix);// use your own 4x4-matrix
```

❻
```
glViewport(0,0,w,h); // adjust viewport transform
```

1. Rendering Pipeline

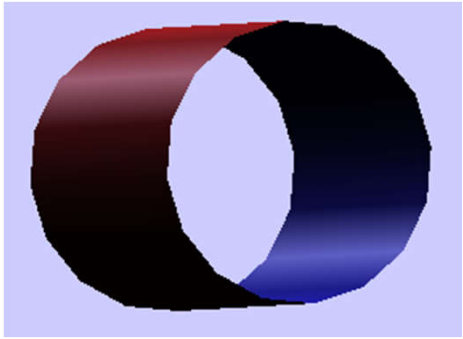
❼
```
// define material color properties for front and back
float amb[4]={.2,.1,.1, 1}; // RGBA
float dif[4]={.9,.3,.3, 1};
float spe[4]={ 1, 1, 1, 1};
int mat = GL_FRONT_AND_BACK;
// use GL_FRONT and GL_BACK for different colors

// define ambient, diffuse and specular components
glMaterialfv( mat, GL_AMBIENT, amb);
glMaterialfv( mat, GL_DIFFUSE, dif);
glMaterialfv( mat, GL_SPECULAR, spe);
glMaterialf( mat, GL_SHININESS, 50.0);// Phong constant
```

1. Rendering Pipeline

❼
```
void DrawCylinder( int reso = 16){// drawing a cylinder
  //allocate memory for x and y coordinates on a circle
  double *c = new double[ reso+1];
  double *s = new double[ reso+1];

  for( int i=0; i<=reso; i++){
    // compute x and y coordinates of circle
    c[i] = cos( 2.0 * M_PI * i / reso );
    s[i] = sin( 2.0 * M_PI * i / reso );
    //cout << i << " " << c[i] << endl;
  }
  glBegin( GL_QUADS); // 4 points define a polygon
```

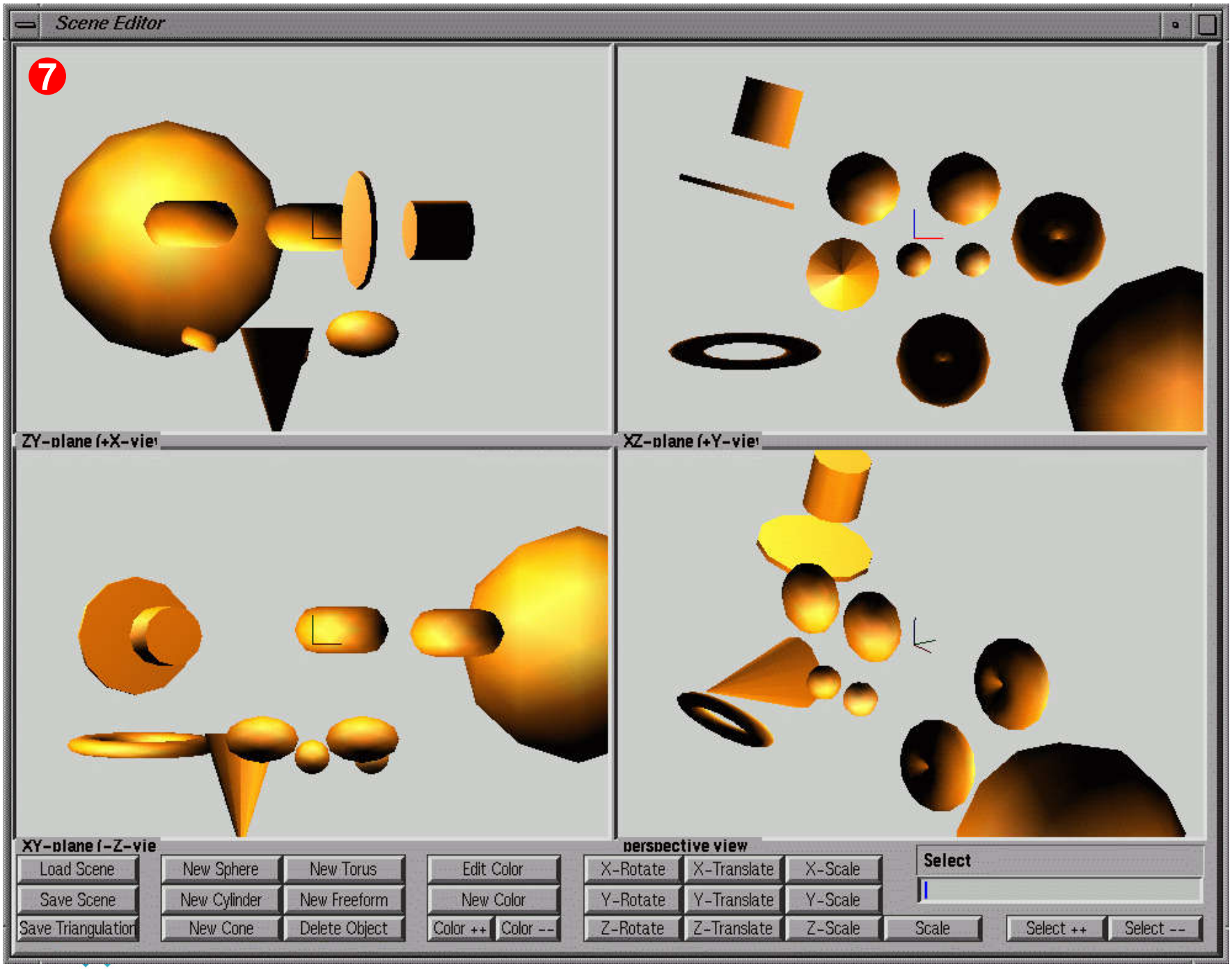
1. Rendering Pipeline

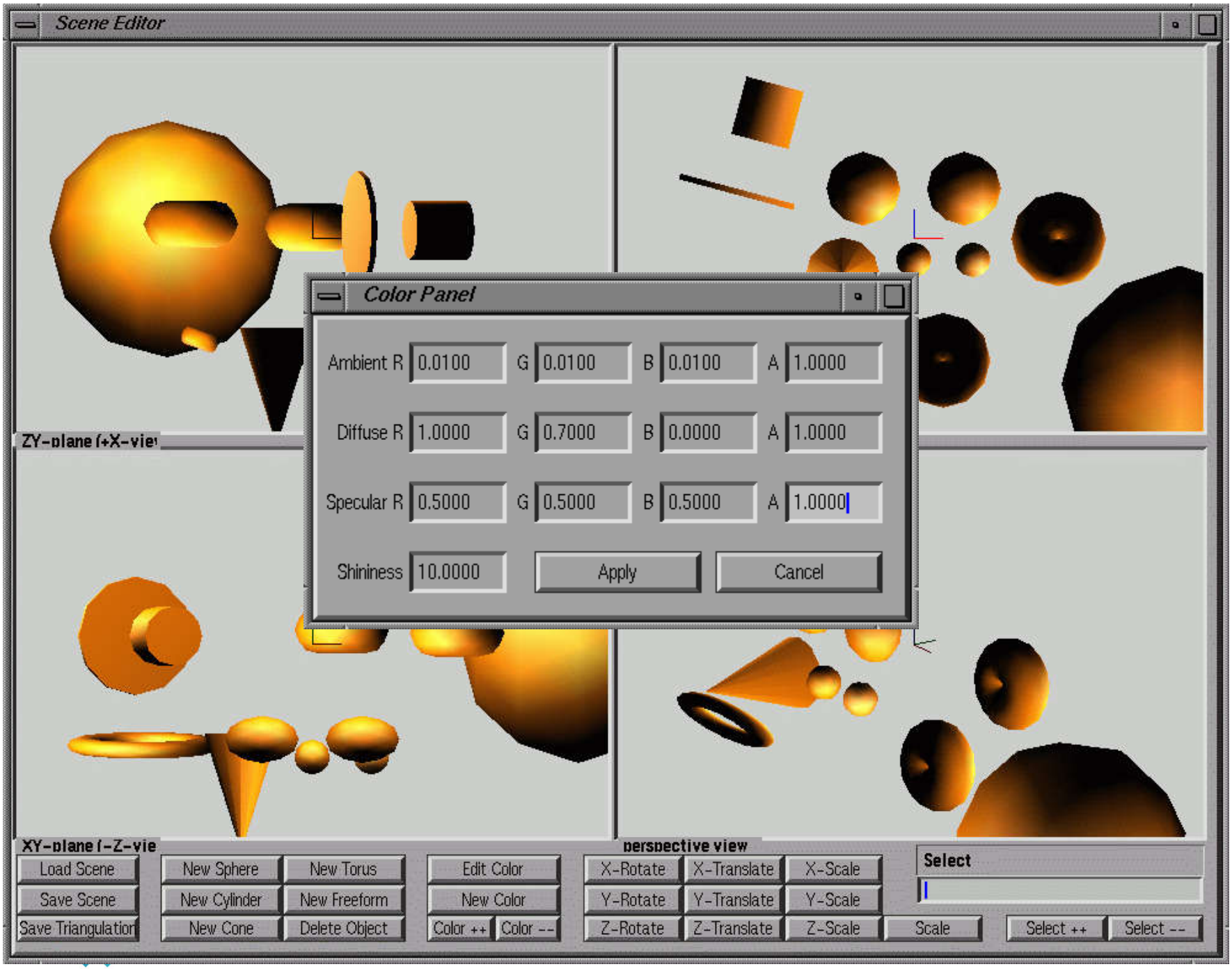
```
for( int i=0; i<reso; i++){
  // normal vector used for all consecutive points
  glNormal3f( c[i], s[i], 0.0);
  glVertex3f( c[i], s[i], 3.0); // 2 points ...
  glVertex3f( c[i], s[i], 0.0);
  // another normal with two more points
  glNormal3f( c[i+1], s[i+1], 0.0);
  glVertex3f( c[i+1], s[i+1], 0.0);
  glVertex3f( c[i+1], s[i+1], 3.0); }
glEnd(); // concludes GL_QUADS
delete[] c; // de-allocate space
delete[] s; }
```

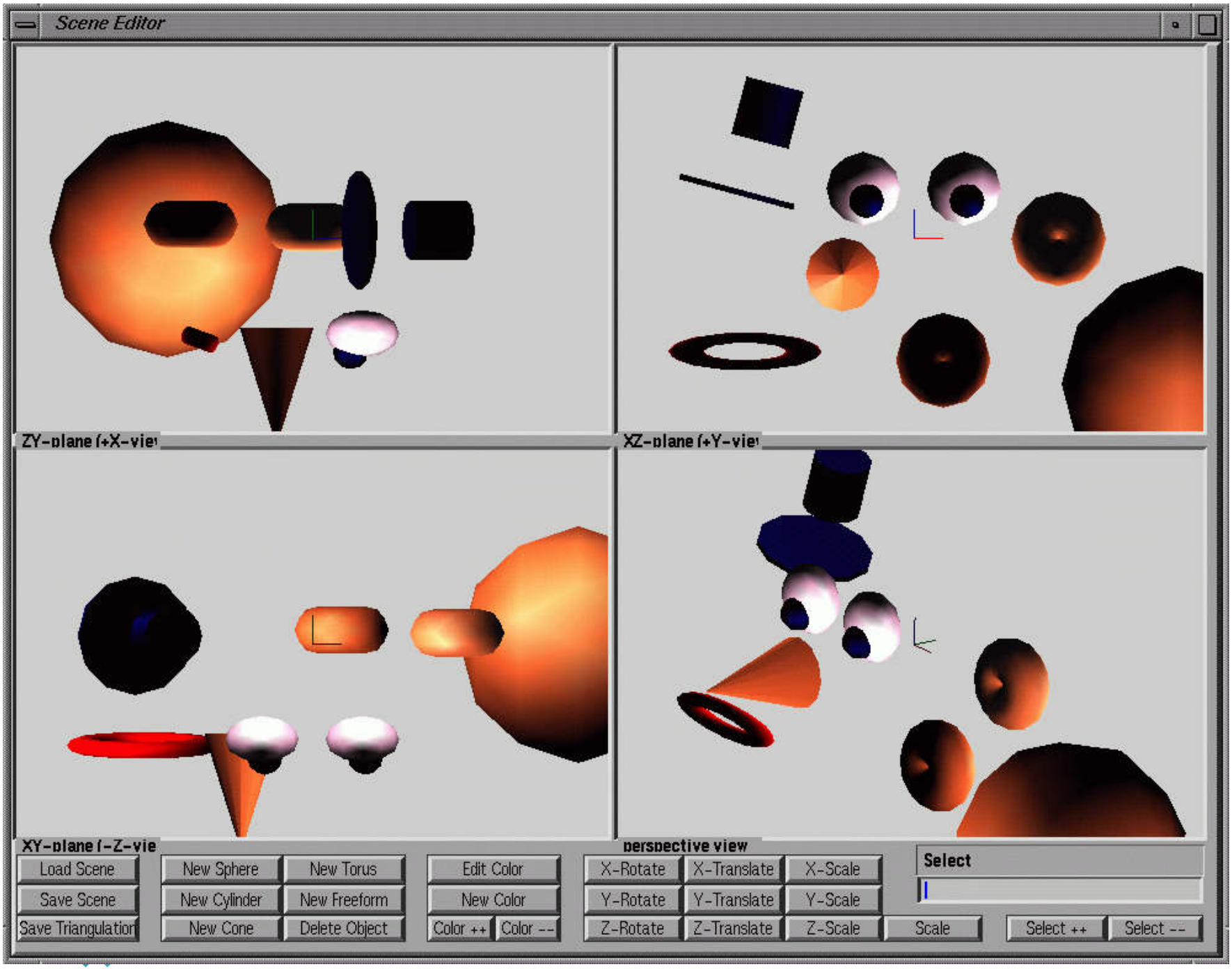


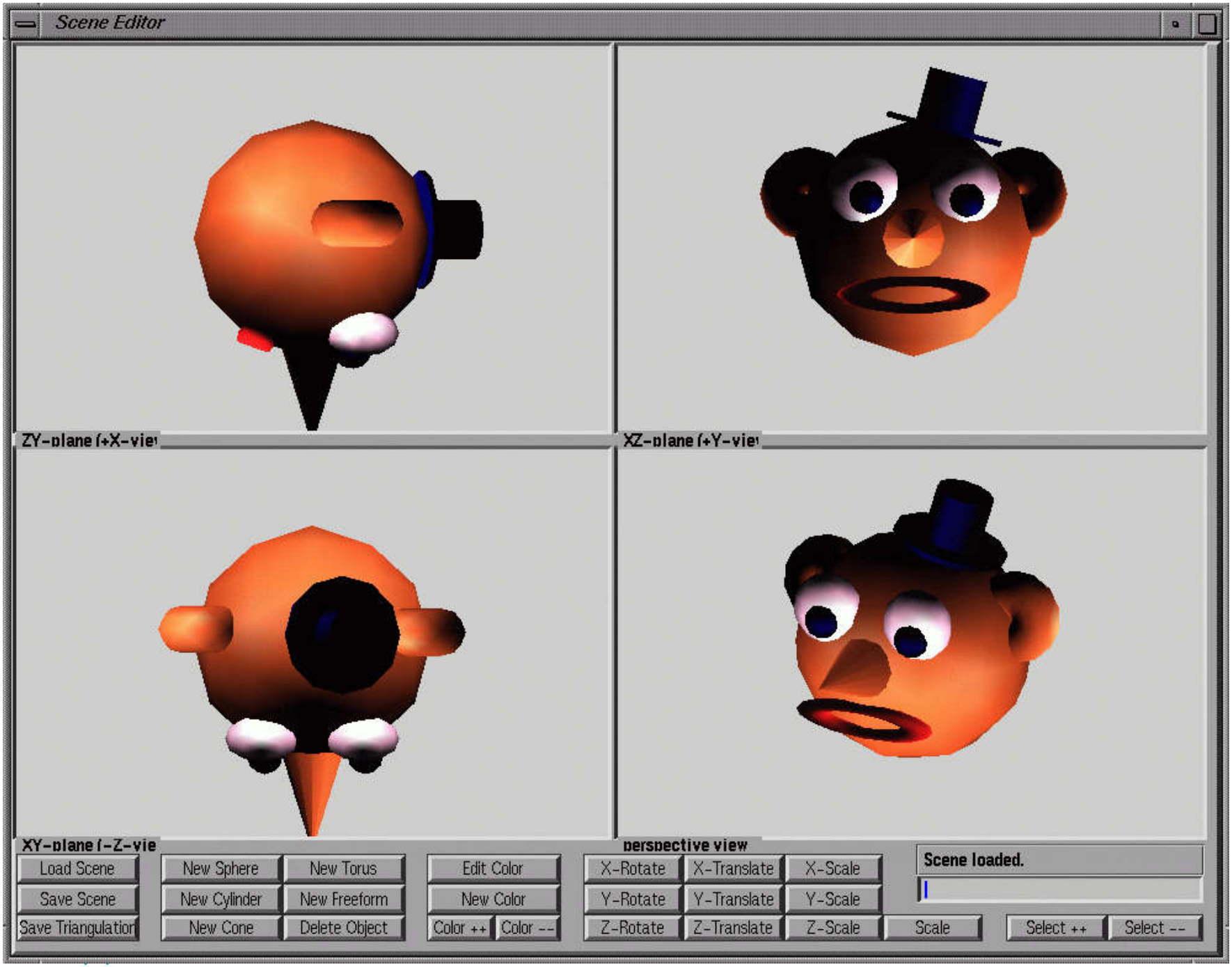
1. Rendering Pipeline

```
// you can also use glBegin and glEnd with
// GL_TRIANGLES : each 3 points define a triangle
// GL_TIANGLE_FAN : connecting all points with tri's
// GL_TRIANGLE_STRIP : triangles in a zig-zag stripe

// draw a cylinder with default resolution
DrawCylinder();
// glCallList( object); // faster with display lists

// make it appear (still hidden in the rear buffer)
glFlush();
```


Scene Editor
7
ZY-plane (+X-vie
XZ-plane (+Y-vie
XY-plane (-Z-vie
perspective view
Load Scene
Save Scene
Save Triangulation
New Sphere
New Torus
New Cylinder
New Freeform
New Cone
Delete Object
Edit Color
New Color
Color ++
Color --
X-Rotate
X-Translate
X-Scale
Y-Rotate
Y-Translate
Y-Scale
Z-Rotate
Z-Translate
Z-Scale
Scale
Select
Select ++
Select --


Scene Editor
Color Panel
Ambient R 0.0100 G 0.0100 B 0.0100 A 1.0000
Diffuse R 1.0000 G 0.7000 B 0.0000 A 1.0000
Specular R 0.5000 G 0.5000 B 0.5000 A 1.0000
Shininess 10.0000
Apply
Cancel
ZY-plane (+X-vie
XY-plane (-Z-vie
perspective view
Load Scene
Save Scene
Save Triangulation
New Sphere
New Torus
New Cylinder
New Freeform
New Cone
Delete Object
Edit Color
New Color
Color ++
Color --
X-Rotate
X-Translate
X-Scale
Y-Rotate
Y-Translate
Y-Scale
Z-Rotate
Z-Translate
Z-Scale
Scale
Select
Select ++
Select --


Scene Editor
ZY-plane (+X-vie
XZ-plane (+Y-vie
XY-plane (-Z-vie
perspective view
Load Scene
Save Scene
Save Triangulation
New Sphere
New Torus
New Cylinder
New Freeform
New Cone
Delete Object
Edit Color
New Color
Color ++
Color --
X-Rotate
X-Translate
X-Scale
Y-Rotate
Y-Translate
Y-Scale
Z-Rotate
Z-Translate
Z-Scale
Scale
Select
Select ++
Select --


Scene Editor
ZY-plane (+X-vie
XZ-plane (+Y-vie
XY-plane (-Z-vie
perspective view
Load Scene
Save Scene
Save Triangulation
New Sphere
New Torus
New Cylinder
New Freeform
New Cone
Delete Object
Edit Color
New Color
Color ++
Color --
X-Rotate
X-Translate
X-Scale
Y-Rotate
Y-Translate
Y-Scale
Z-Rotate
Z-Translate
Z-Scale
Scale
Scene loaded.
Select ++
Select --

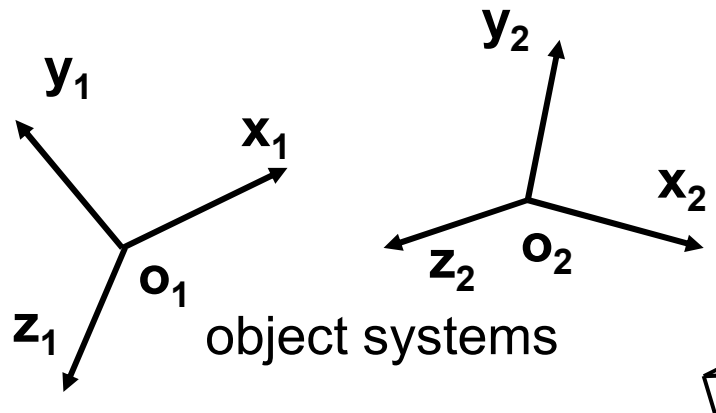
2. Coordinate Transforms

When modeling moving objects, position and orientation in space need to be represented as functions of time. Therefore, the scene is recorded at so-called ***key frames***, where location, orientation and possibly deformation parameters are specified for every object. These parameters are then interpolated over time. Hence, we also need an interpolation method for orientations in threespace.

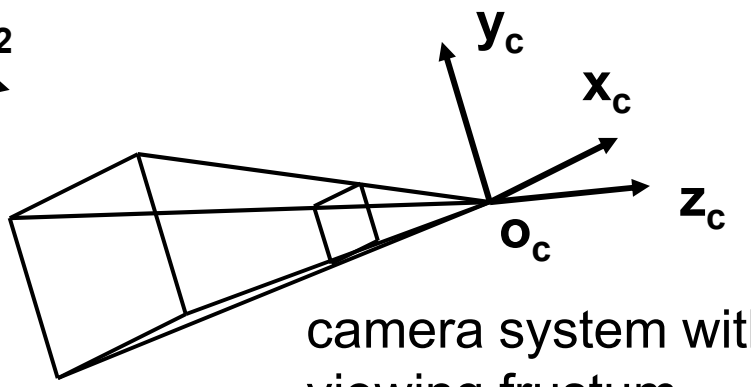
In principle, different objects can be described at individual key frames. Also the camera positioning and lighting can be defined with the aid of key frames.

2. Coordinate Transforms

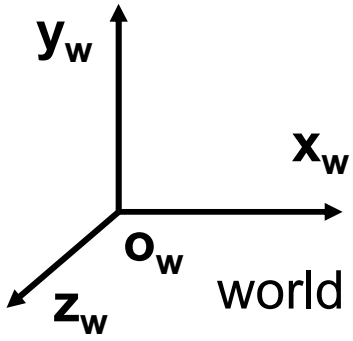
Every coordinate system is defined by an origin and three unit vectors.



object systems



camera system with
viewing frustum
(view direction towards $-z_c$)



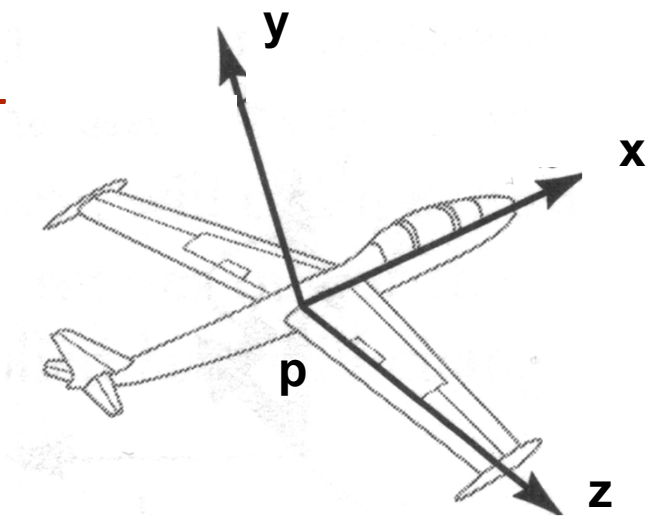
world system

2. Coordinate Transforms

Positioning in Threespace

An object may be placed at a point **p**(t) with orientation **R**(t),

$$\mathbf{q}_{global} = \mathbf{R}\mathbf{q}_{local} + \mathbf{p}$$

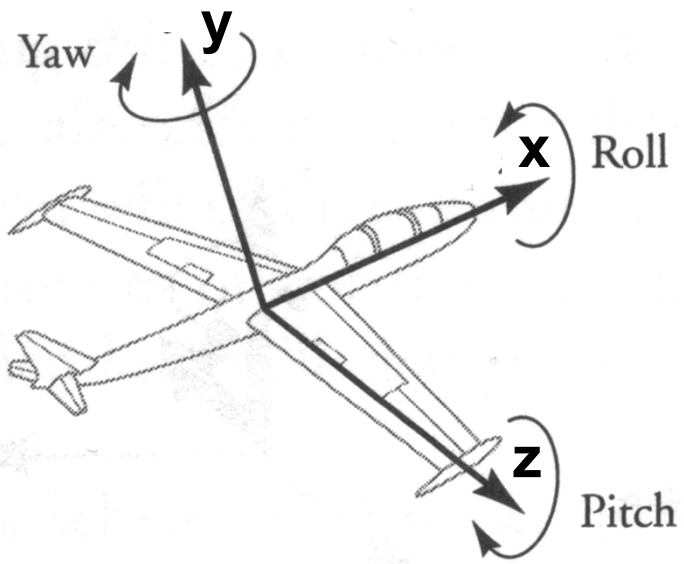


where **q** is any point of the object in local and global (world) coordinates.

The rotation matrix **R = [x,y,z]** simply contains the three axes of the local system (see figure) as column vectors.
Hence, the object is moved by placing and rotating its local system in the world system.

Figure: [Rick Parent, Computer Animation, MK 2007]

2. Coordinate Transforms



Euler Angles

An alternative way of defining orientations in threespace is by **Euler angles**, rotating around the coordinate axes:

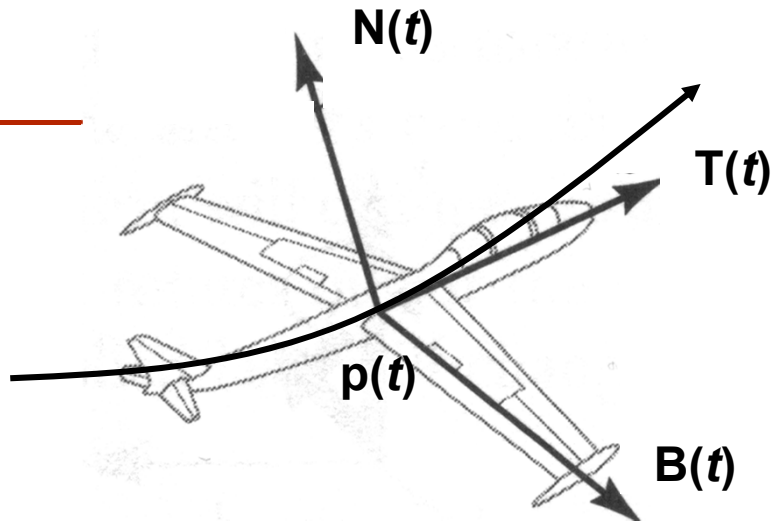
$$\mathbf{R} = \mathbf{R}_x(\alpha)\,\mathbf{R}_y(\beta)\,\mathbf{R}_z(\gamma)$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix} \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

2. Coordinate Transforms

Frenet-Serret Frame

The **Frenet Frame** is a coordinate system that travels with a curve.
It consists of **R** = [**T**, **N**; **B**],



$$\mathbf{T} = \frac{\dot{\mathbf{p}}}{\|\dot{\mathbf{p}}\|}, \qquad \mathbf{B} = \frac{\dot{\mathbf{p}} \times \ddot{\mathbf{p}}}{\|\dot{\mathbf{p}} \times \ddot{\mathbf{p}}\|}, \qquad \mathbf{N} = \mathbf{B} \times \mathbf{T}.$$

The orientation of the bi-normal is chosen such that the system is right-handed.

2. Coordinate Transforms

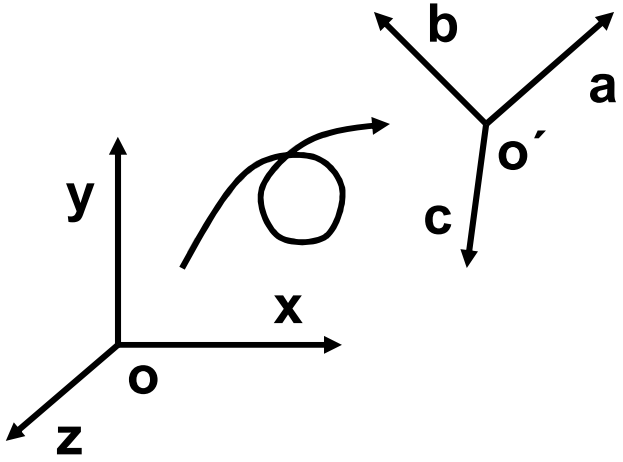
Homogeneous Coordinates

The mapping of a system S = {**o**,**x**,**y**,**z**} into a different system S´ = {**o´**,**a**,**b**,**c**} is equivalent to an ***affine transform***

p = **M**p´ + **t**

which is composed of a translation **t** = **o´** - **o** and a linear map **M**, in ***homogeneous coordinates***:

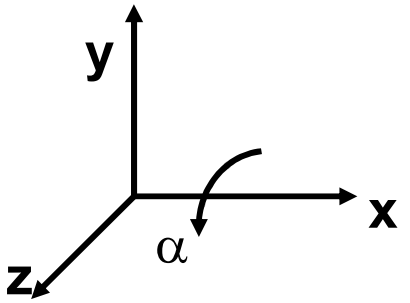
$$\begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & t_1 \\ m_{21} & m_{22} & m_{23} & t_2 \\ m_{31} & m_{32} & m_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \\ 1 \end{pmatrix}$$



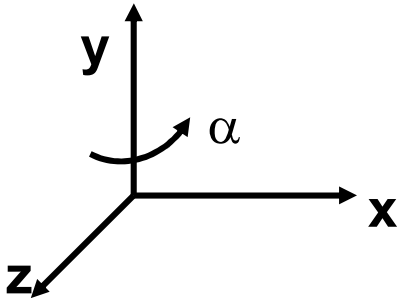
2. Coordinate Transforms

Linear Maps: Rotation Matrices

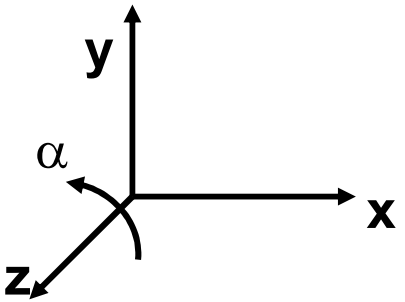
$$\mathbf{R}_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$



$$\mathbf{R}_y(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{pmatrix}$$



$$\mathbf{R}_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



2. Coordinate Transforms

Combining multiple Transforms

When calculating consecutive transforms, e.g. rotations around different axis (or scene graph traversal), the corresponding 3×3- or 4×4-matrices are multiplied **in reverse order**:

$$\mathbf{p' = R_3\, R_2\, R_1\, p}$$

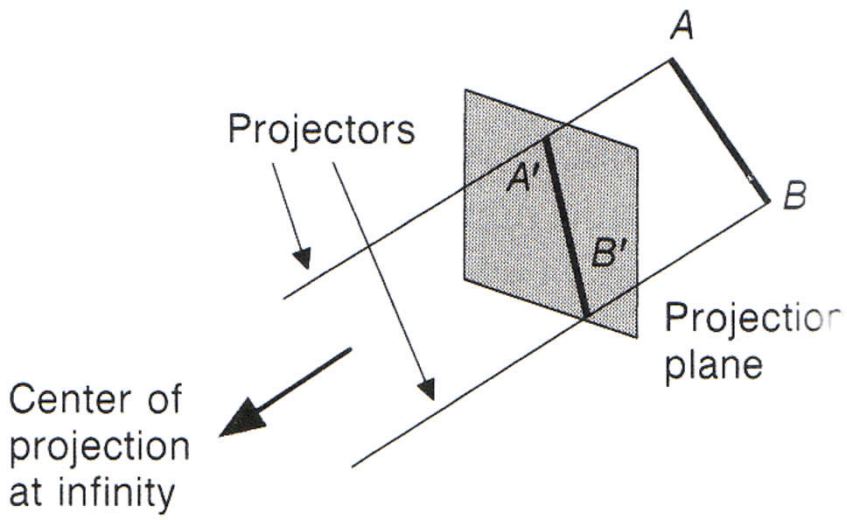
means that first $\mathbf{R_1}$, then $\mathbf{R_2}$ and last $\mathbf{R_3}$ is performed. Sometimes (for example in OpenGL documentations) the transposed notation is used:

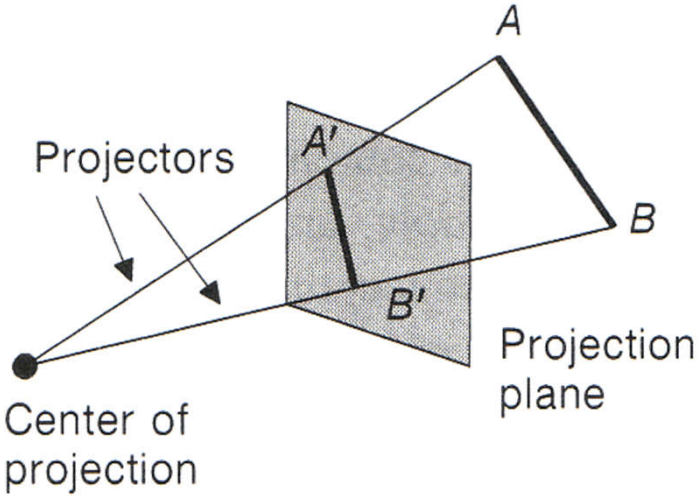
$$\mathbf{p'^T = p^T\, R_1^T\, R_2^T\, R_3^T}.$$

3. Projections

We need projections for mapping 3D geometry into 2D screen space. We distinguish between **parallel** and **central** projections.

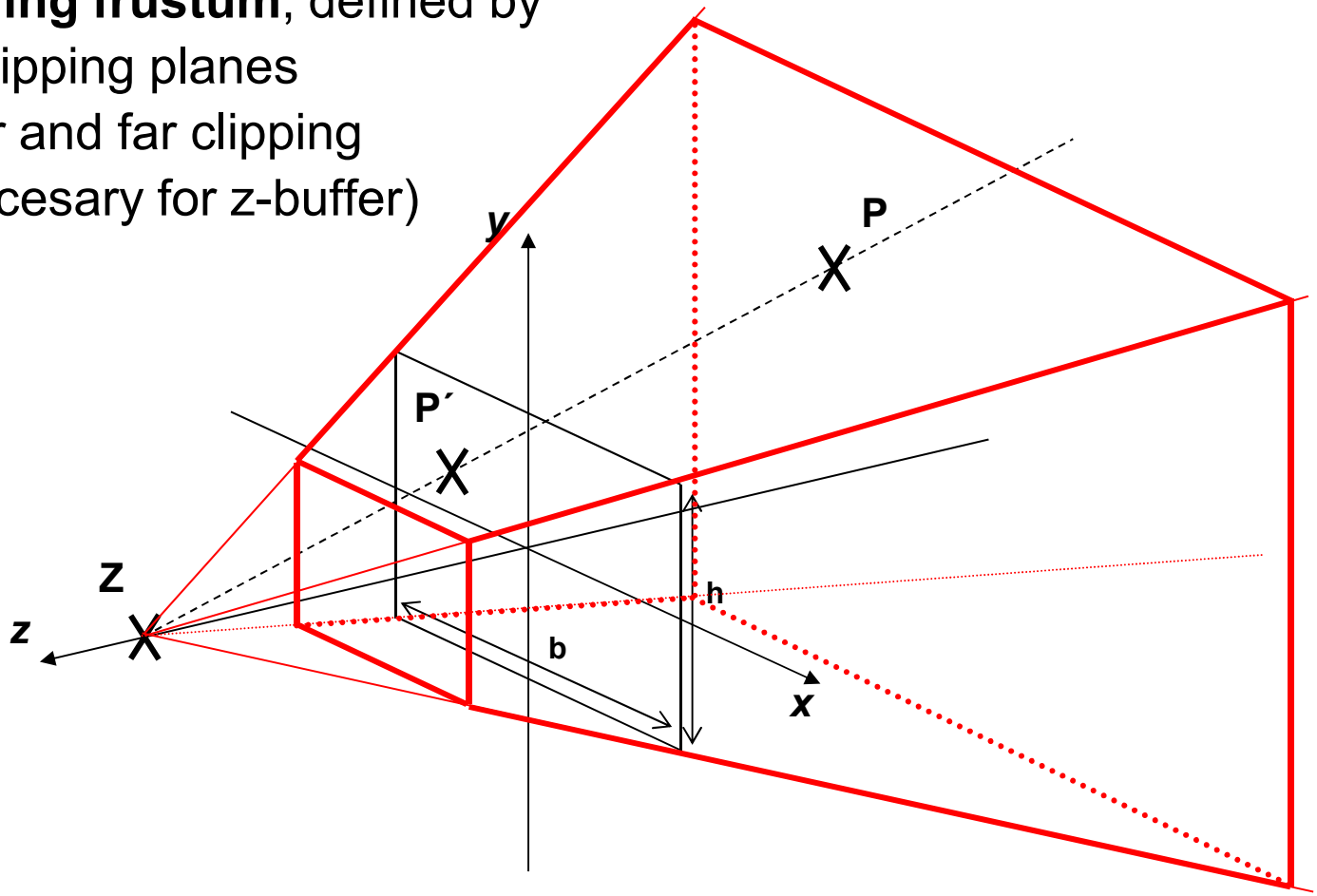
Central projectious use a finite focal distance (distance between projection center and plane) and thus warp the projected objects.





3. Projections

Viewing frustum, defined by six clipping planes (near and far clipping is necesary for z-buffer)



4. Visibility Culling and Rasterization

The goal of visibility methods is the efficient detection of the parts of a scene that are visible form a certain point. Removing all surfaces that are either outside the viewing frustum or hidden behind other objects is necessary for real-time rendering. Visibility methods are also used to determine shadows on surfaces (visibility between surfaces and light sources) as well as reflections (mutual visibility of two surface components).

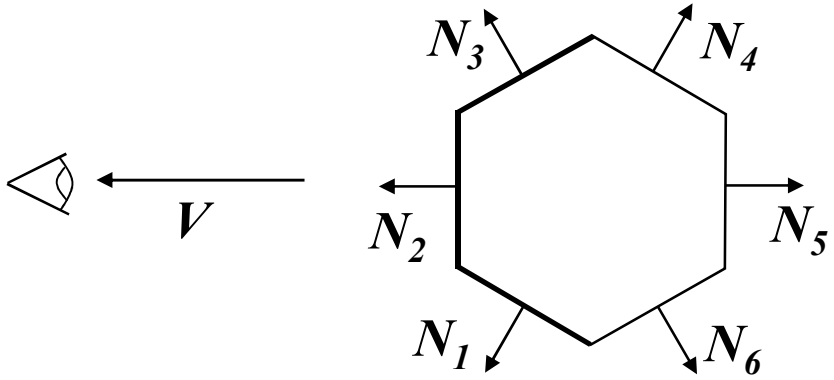
visibility methods operate either in

- **object space (in world coordinates)**
  using, for example, space partitioning trees
- **screen space**
  exploiting coherence along depth (viewing direction, z)

4. Visibility Culling and Rasterization

Back-Face Culling (Object Space)

An efficient way of removing hidden surfaces of solids is **back-face culling**. Polygons located on the rear side are identified by a negative dot product of viewing vector and normal $V \cdot N < 0$. These polygons need not to be rendered.

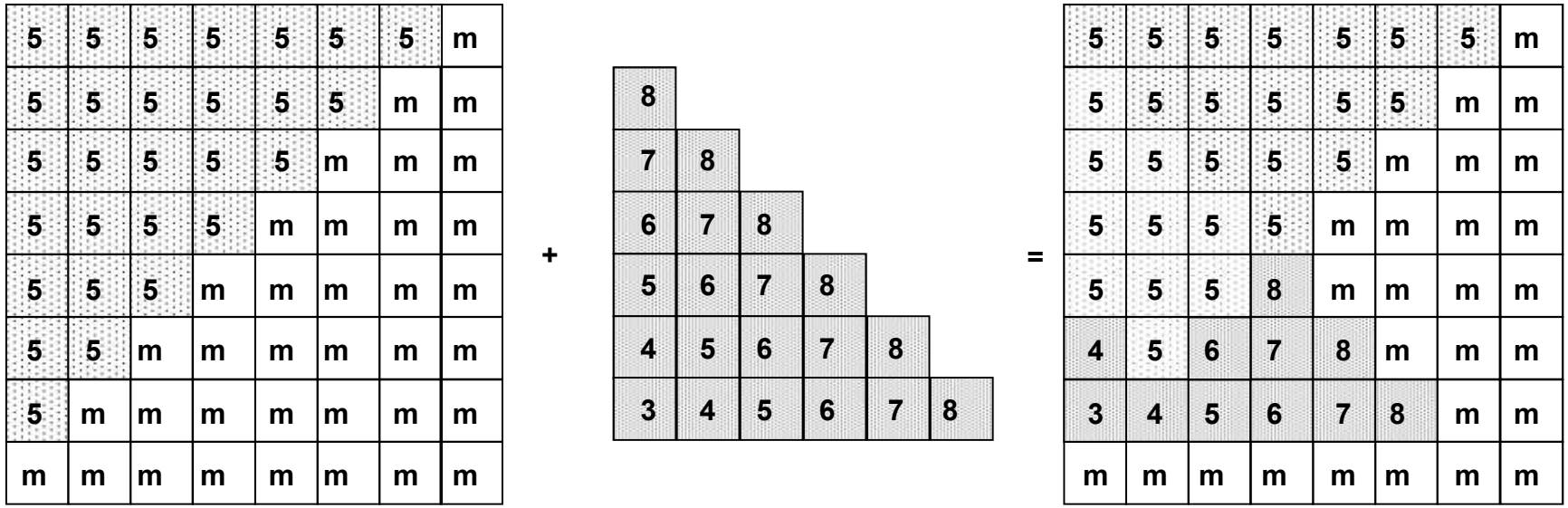


$$V \cdot N_1 > 0 \quad V \cdot N_4 < 0$$
$$V \cdot N_2 > 0 \quad V \cdot N_5 < 0$$
$$V \cdot N_3 > 0 \quad V \cdot N_6 < 0$$

4. Visibility Culling and Rasterization

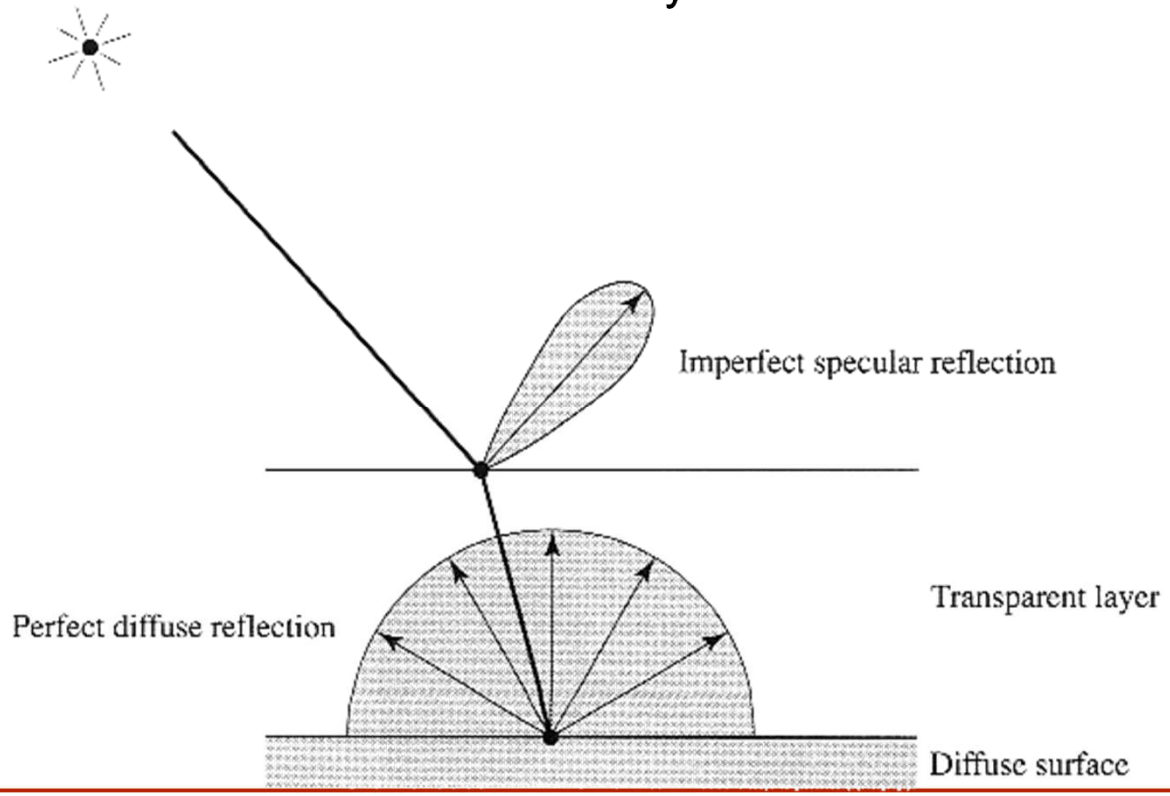
Z-Buffering (Screen Space)

During the rasterization stage, the depth (z-coordinate) is stored for every pixel, together with its color. Thus, a per-pixel decision of visibility is possible.



5. Local Illumination

In real world, many surfaces contain a polished layer with specular reflection and underneath a diffuse layer due to micro facets.



5. Local Illumination

This is the idea of the **Phong Illumination Model**

[Bui Tuong **Phong**, 1975]:

$$I = k_a I_a + k_d I_d + k_s I_s$$

- I_a = constant ambient intensity (multiply reflected light)
- I_d diffuse intensity (reflects equally in all directions)
- I_s specular intensity (distributed around reflection vector)
- k_a, k_d, k_s material constants in [0,1]

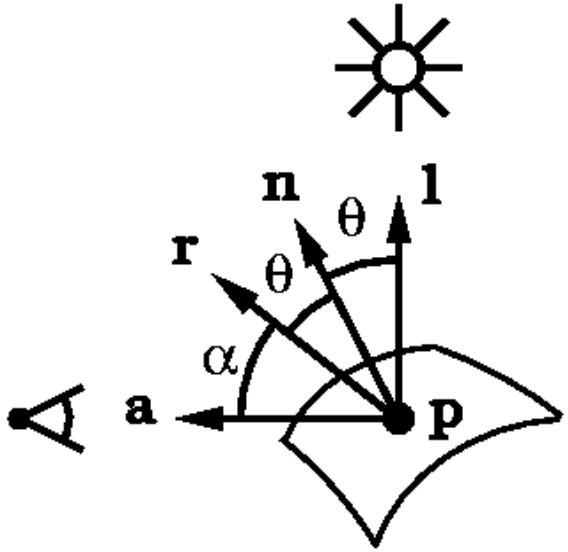
5. Local Illumination

The **diffusely reflected intensity** I_d is proportional to the orthogonally incoming light intensity (law of cosine by Lambert)

$$I_d = cos(\theta)\ I_{ein} = n \cdot l\ I_{ein}$$

l normalized vector pointing to light source

n surface normal



scalar product:

$$a \cdot b = a_x b_x + a_y b_y + a_z b_z$$

$$a \cdot b = \|a\| \|b\| cos \angle(a, b)$$

5. Local Illumination

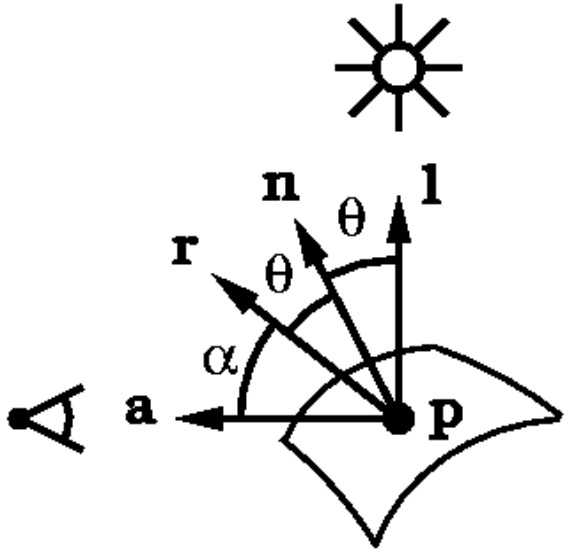
The **specularly reflected intensität** I_s has its maximum when the view direction is opposing to the reflection vector:

$$I_s = cos^s\alpha \; I_{ein} = (a \cdot r)^s \; I_{ein}$$

a normalized vector to eye point

r reflection vector,

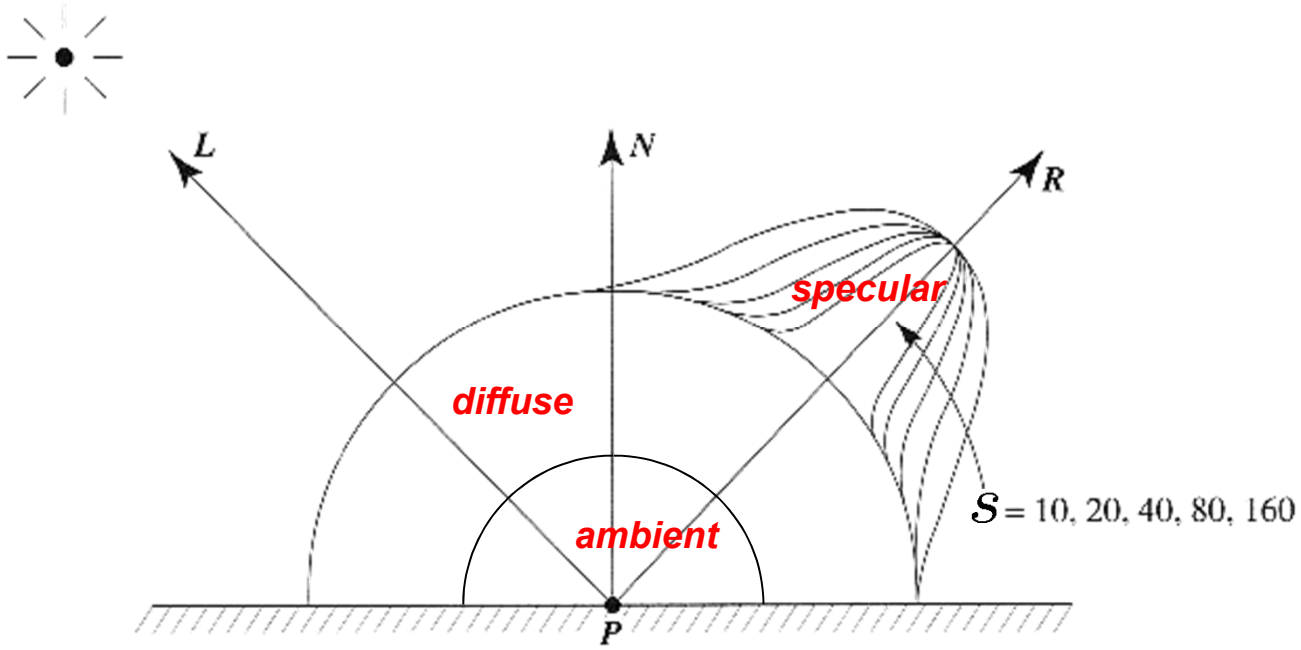
$$r = 2(n \cdot l)n - l$$



5. Local Illumination

For incoming light $I_{ein} = \frac{1}{d_{lp}} I_l$ the Phong model reads

$$I = k_a I_a + \frac{1}{d_{lp}} I_l \ (k_d(n \cdot l) + k_s(a \cdot r)^s)$$



5. Local Illumination

Example:

k_a constant

increasing k_s

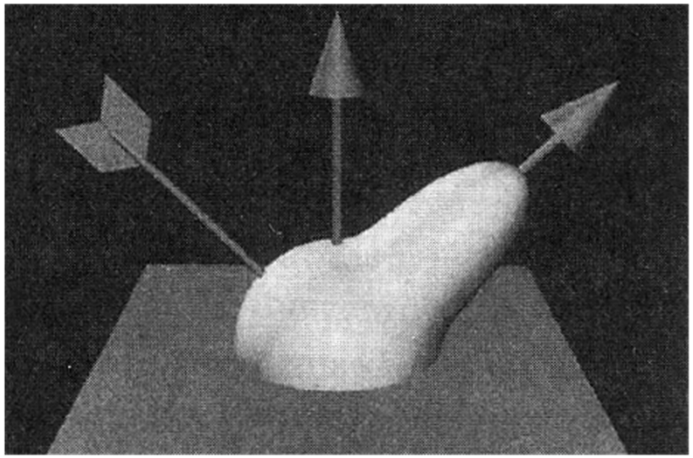
increasing s

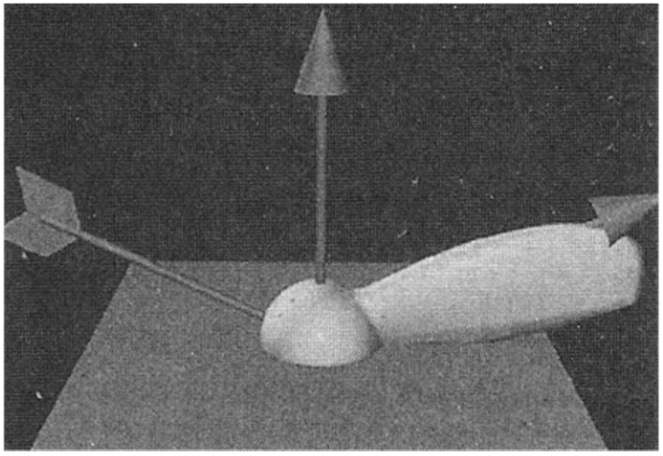
5. Local Illumination

The Phong illumination model has some shortcomings:

- surfaces often look "plastic like"
- mutual reflections are not correctly described by the constant ambient term (local illumination)

A more sophisticated model is the **Bidirectional Reflection Distribution Function (BRDF)** [Nicodemus et al. 1977]:



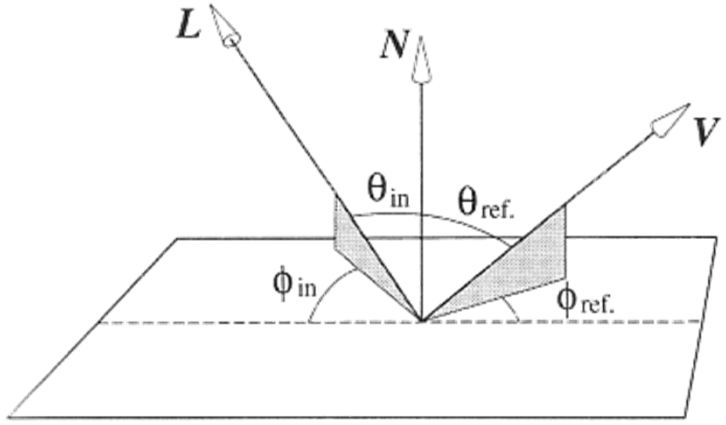


5. Local Illumination

- A BRDF defines the factor between incoming and reflected intensity as a function of the two corresponding directions.
- the vectors L and V ("viewing vector" to eye point) are defined by two angles, each:

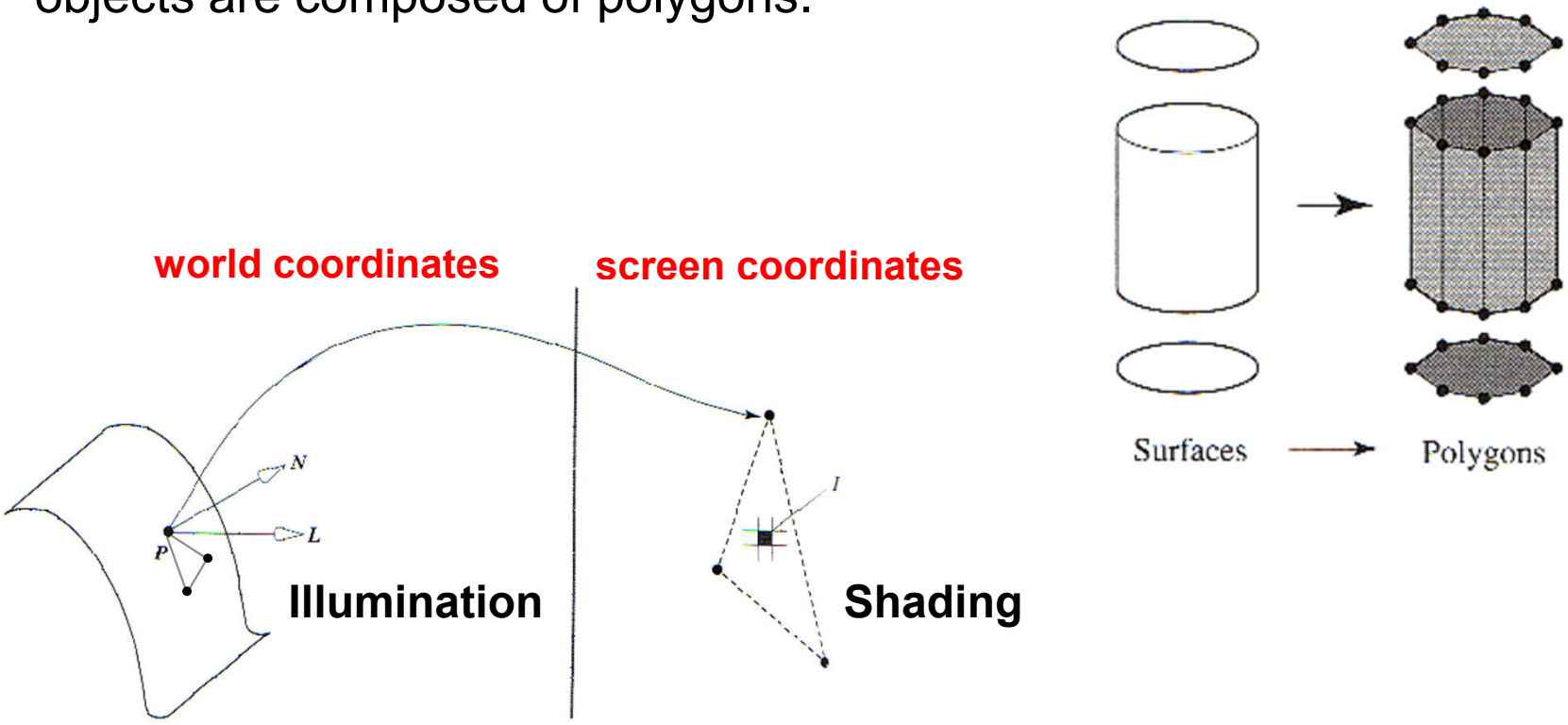
$$BRDF = f(L, V)$$
$$= f(\theta_{in}, \phi_{in}, \theta_{ref}, \phi_{ref})$$

- may be stored in a lookup table, e.g. in texture memory on GPU



6. Shading

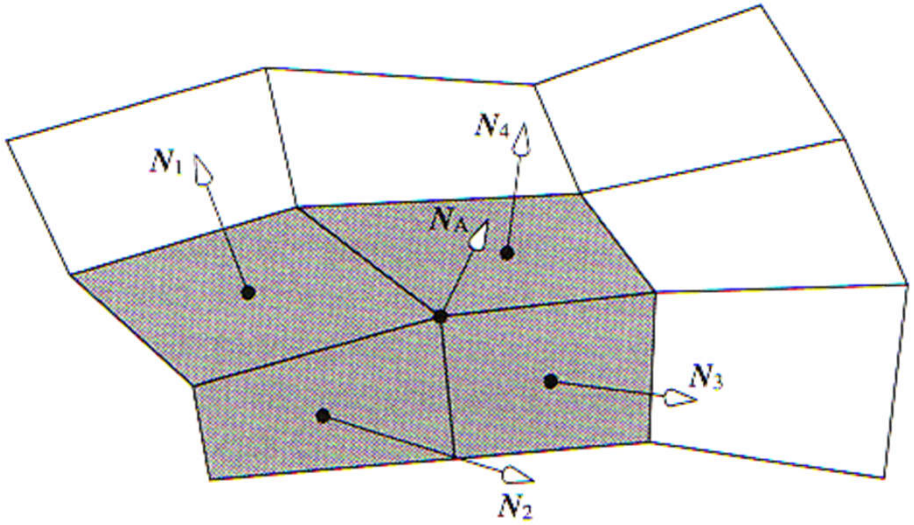
Shading algorithms are used to interpolate colors computed at vertices of the scene in screen space. We assume that the scene objects are composed of polygons.

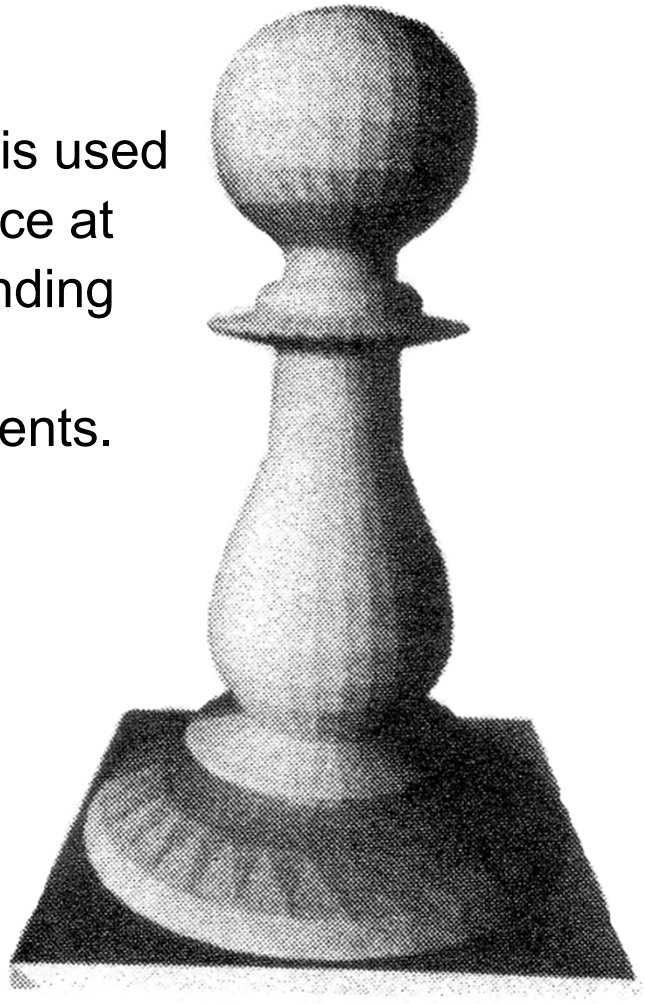


6. Shading

Flat Shading

For every polygon, only one normal vector is used and the lighting model is evaluated only once at the polygon‘s midpoint. All pixels corresponding to a polygon are set in the same color. The polygons appear to be flat surface components.



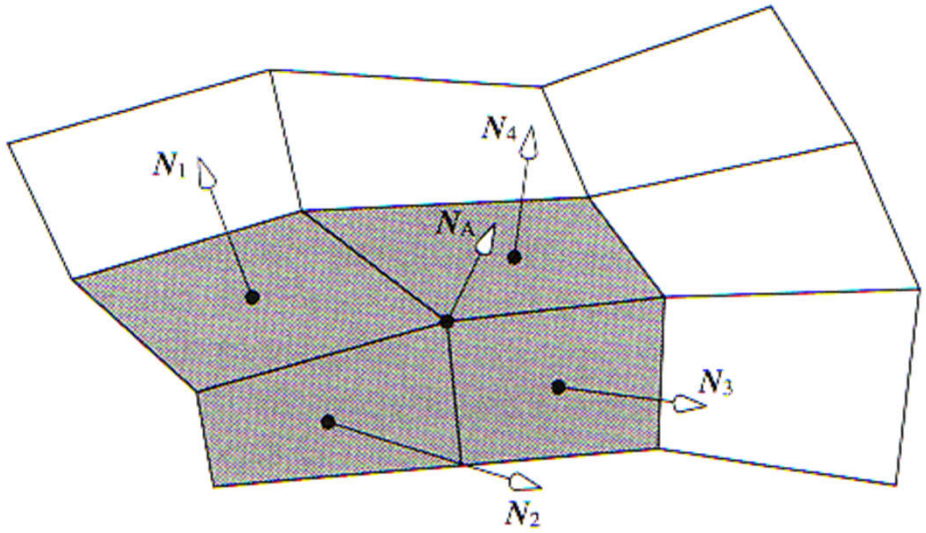


6. Shading

Gouraud Shading

For Gouraud Shading, the illumination is precomputed at every vertex, using an averaged normal of incident polygons (here: N_a).

The color of the individual pixels is then linearly interpolated on the screen, using a scan-line approach.

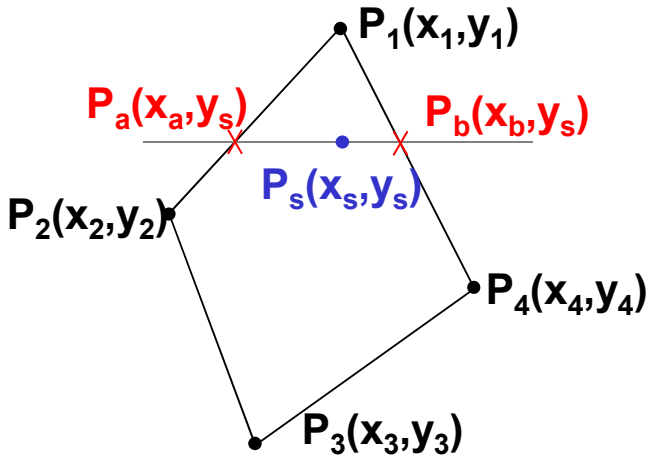


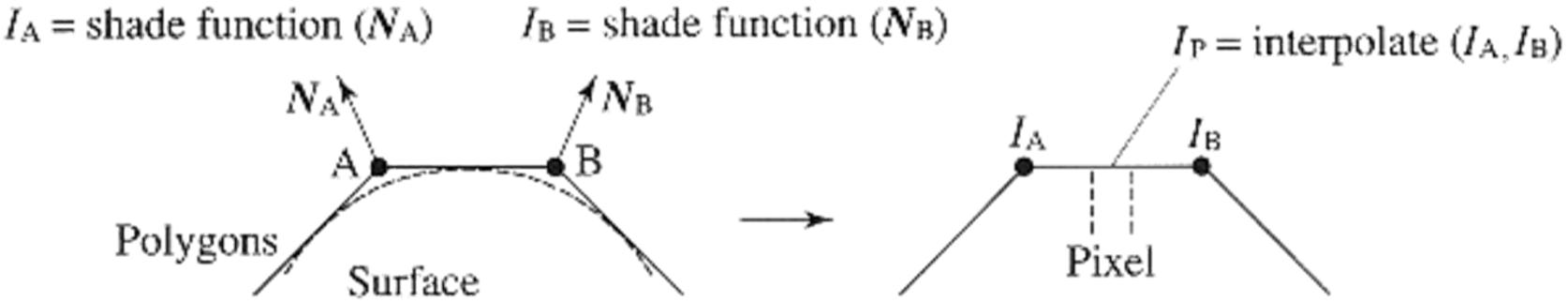
6. Shading

Gouraud Shading (cont'd)

First, the colors on the scan line's intersections with the polygon (here P_a and P_b) are computed from the vertex colors.

Then, for each pixel on the scan line, the color is linearly interpolated between P_a and P_b.





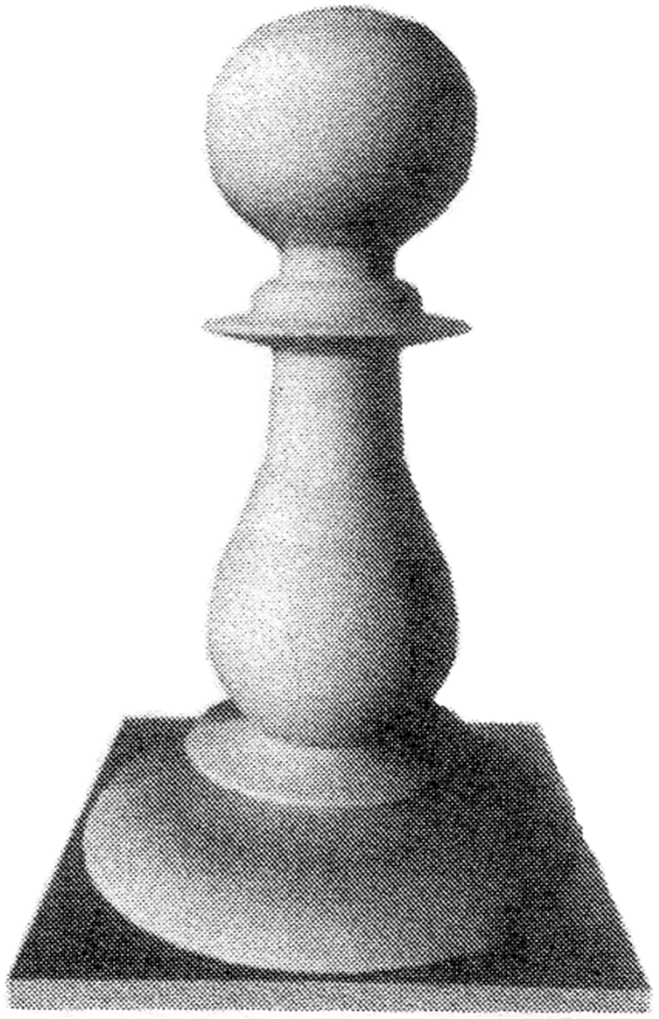
6. Shading

Gouraud Shading (cont‘d)

Gouraud shading has been used in early graphics hardware, due to its efficiency. However, it produces artefacts at low polygonal resolutions:

Highlights may disappear form a surface, due to coarse sampling. Also, the shape of highights appears to be polygonal and not round.

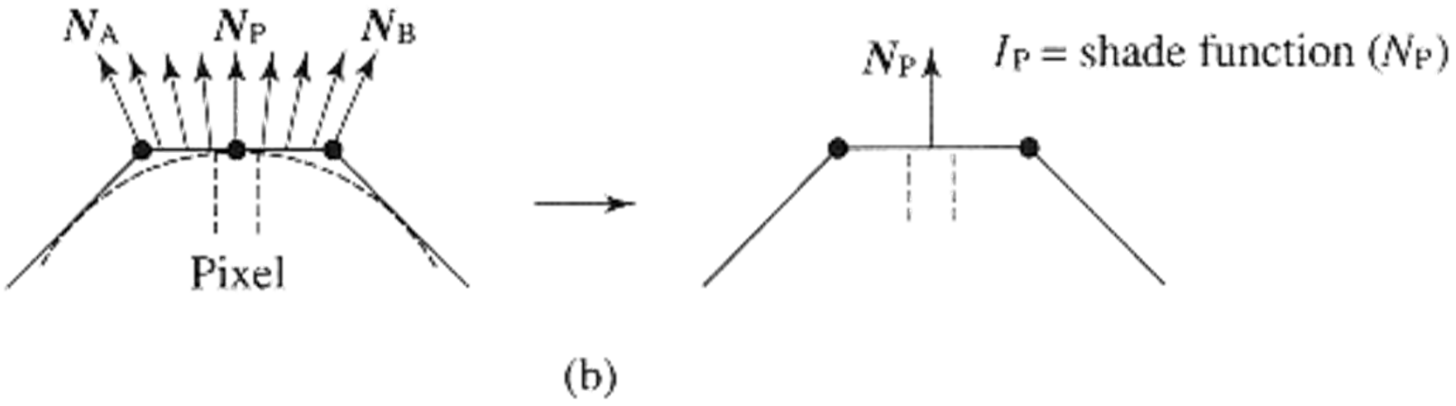
Images of higher quality are obtained using Phong Shading.



6. Shading

Phong Shading

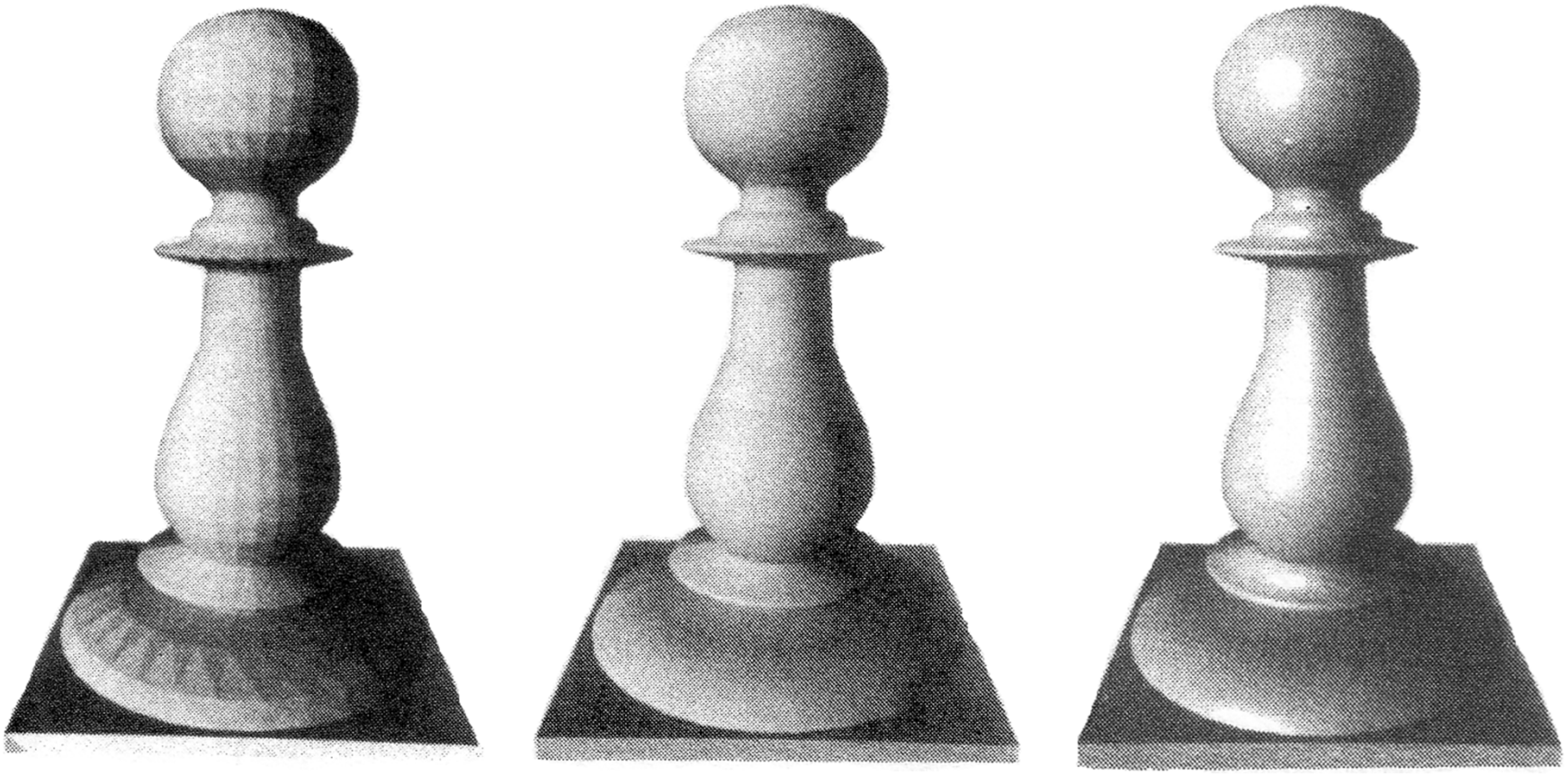
The lighting model is evaluated for every pixel. The corresponding surface normal is interpolated from the vertex normals (in the same way, as interpolating the vertex colors using Gouraud shading). Due to many evaluations of the lighting model, Phong Shading is computationally the most expensive shading algorithm, but it can be executed in parallel on the graphics processing unit (GPU).



(b)

6. Shading

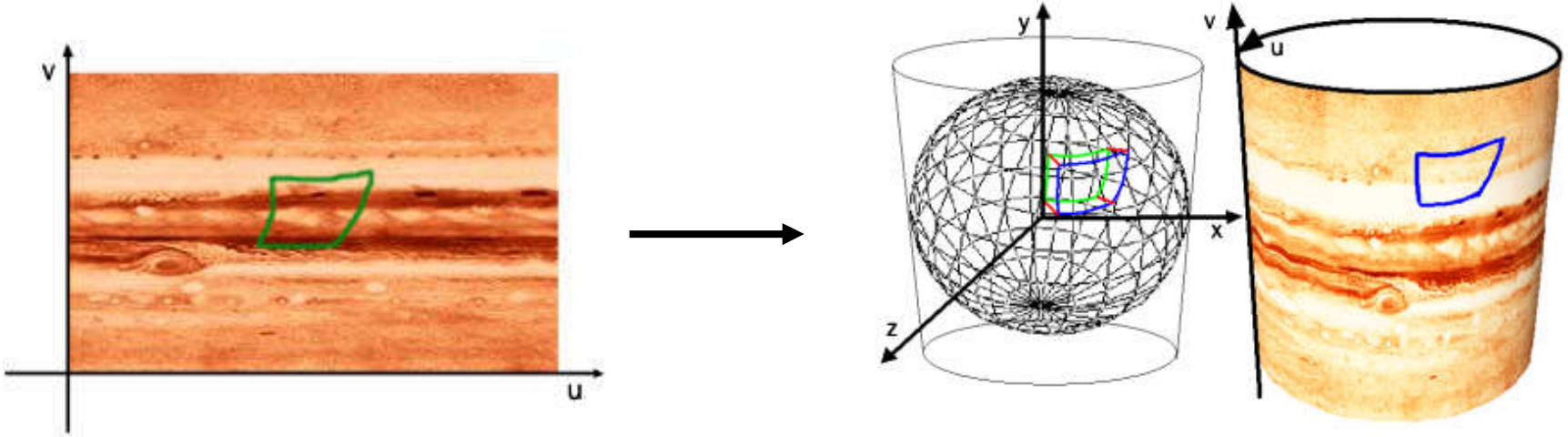
Comparing Flat, Gouraud, and Phong Shading



7. Mapping Techniques

Texture Mapping

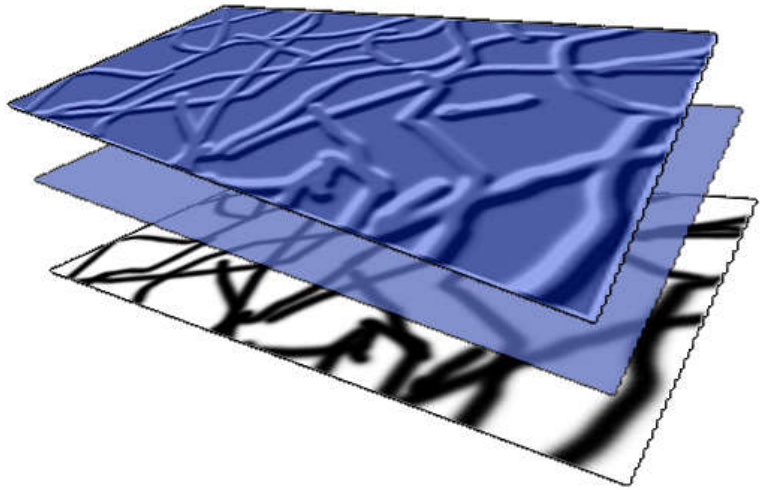
mapping images
onto geometry



7. Mapping Techniques

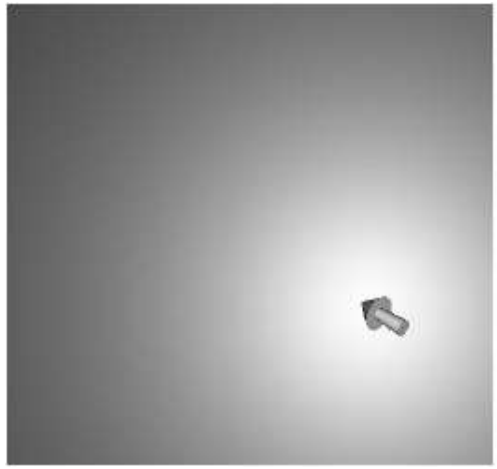
Bump Mapping

combining an image
of normal vectors
(used for local illumination)
with color texture

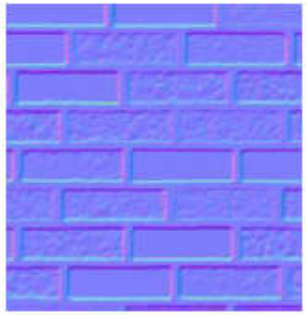


7. Mapping Techniques

Bump Mapping



light map



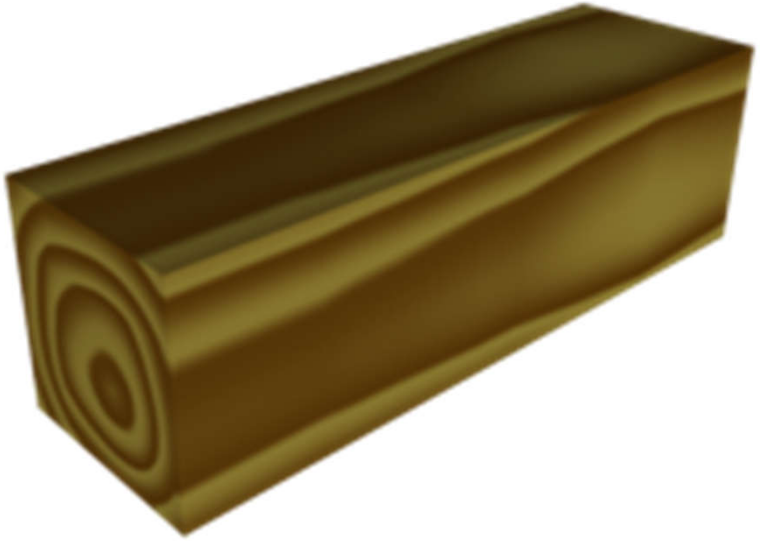
texture with
normal vectors
(color coded)

result

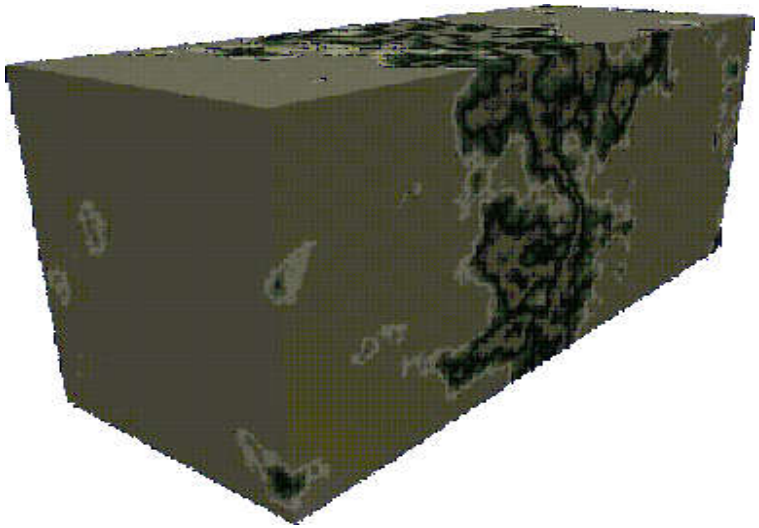
7. Mapping Techniques

3D Texture Mapping

Rather than a 2D image, a 3D color function may be used.
Textures generated by algorithms are known as “Perlin Textures“ [Ken Perlin, Siggraph 85].



Wood



Marble

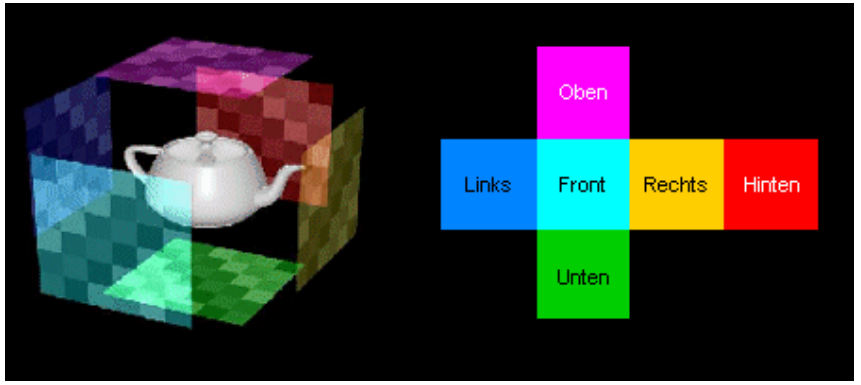
7. Mapping Techniques

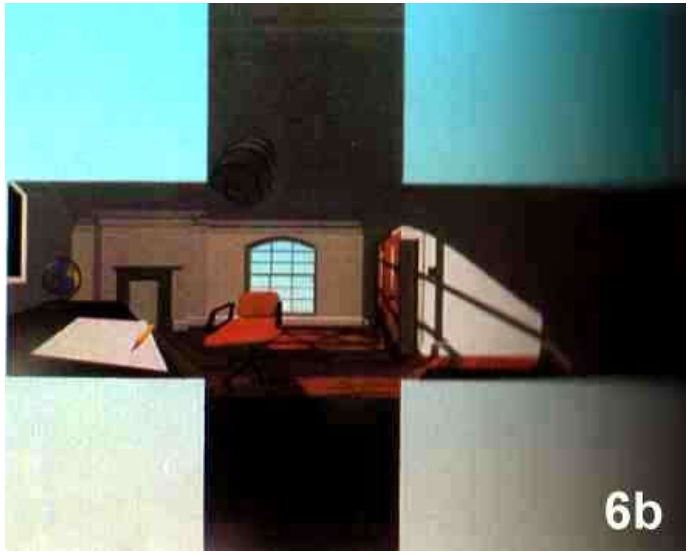
Environment Mapping

An image of the environment is mapped form all sides onto an object. Example:

7. Mapping Techniques

Environment Mapping



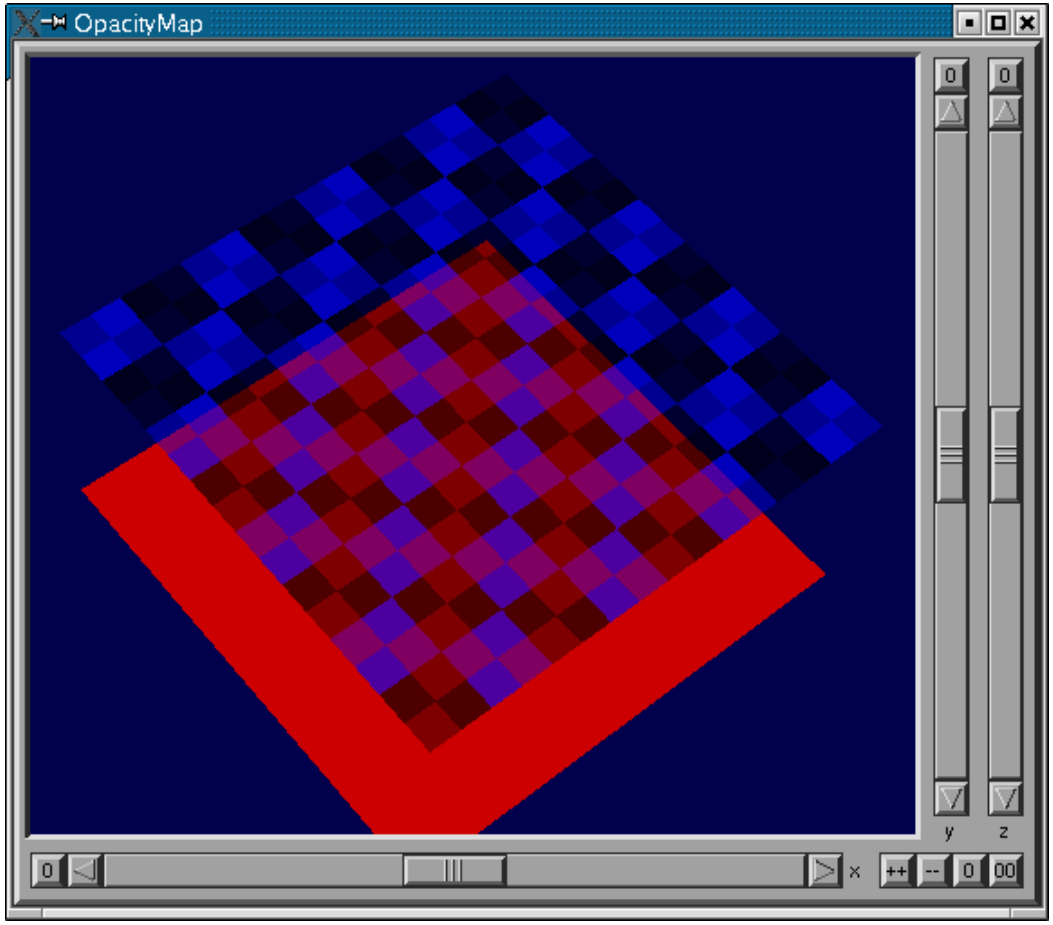


7. Mapping Techniques

Opacity Map

In OpenGL, colors are defined by four components: red, green, blue (base colors) and alpha (opacity).

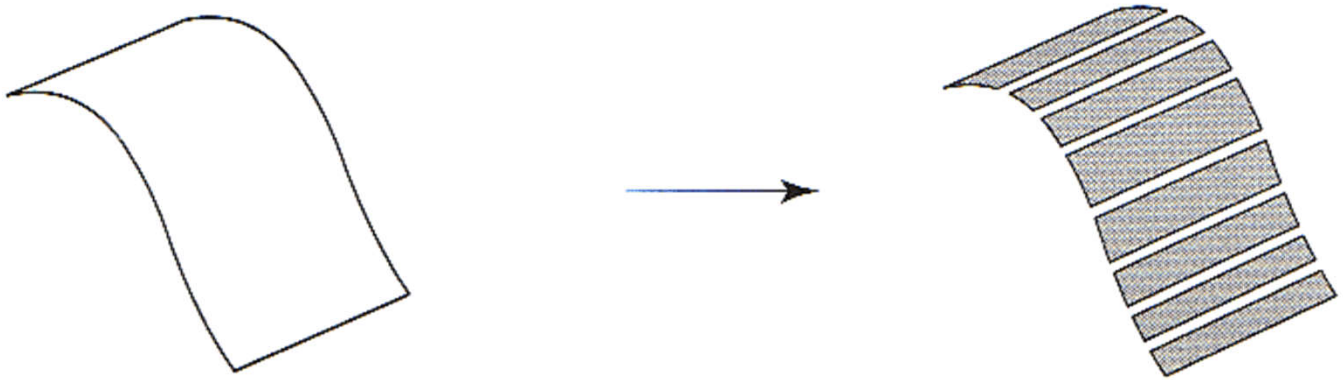
With alpha blending enabled, semi-transparent objects can be rendered.



8. Polygonal Representation

Properties

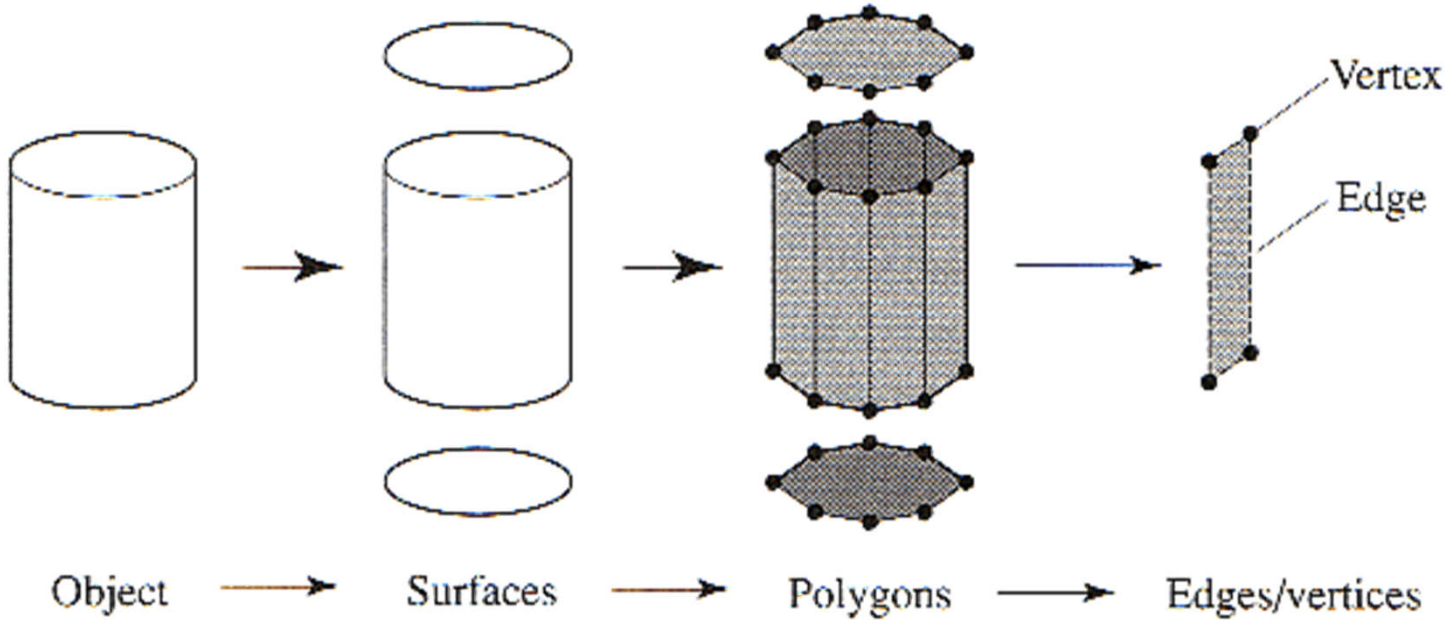
- triangles / polygons are commonly used in computer graphics
- triangulated surfaces (triangulations) provide piece-wise linear surface approximations
- the resolution (number of polygons) can be adapted to the principal curvatures of a surface.



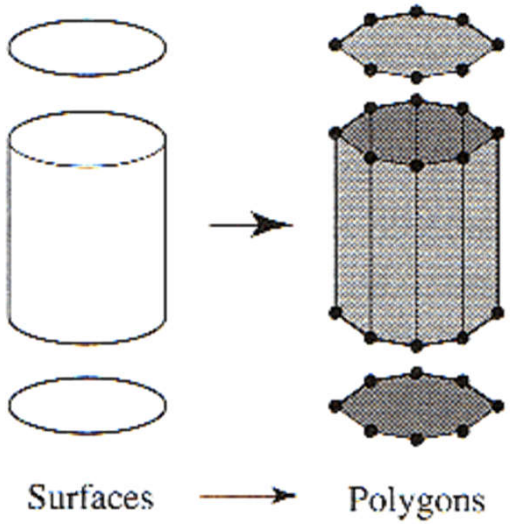
8. Polygonal Representation

Representation

Most objects are composed of multiple surfaces that are subdivided into polygons consisting of edges and vertices.



8. Polygonal Representation



Remark:

There exist two kinds of edges:

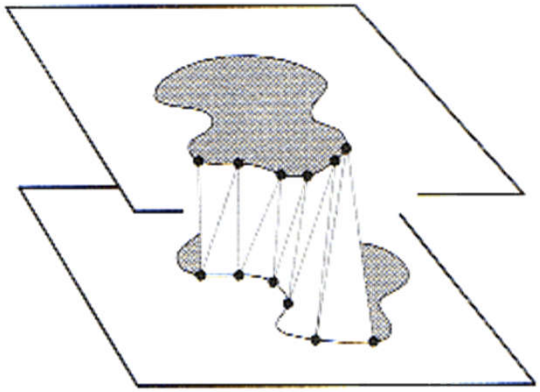
- crease edges (feature lines)
  → these should be visible when rendering
- virtual edges (inside smooth surfaces)
  → these should be „washed out“ when rendering with smooth shading

Data structures mostly store a list of vertices with normals and a list of polygons, each of which contains the indices of its vertices. Crease edges are typically stored twice, since every vertex on a feature line has (at least) two surface normals.

8. Polygonal Representation

Automatic Generation of Polygon Meshes (Reverse Engineering)

(a)



(b)

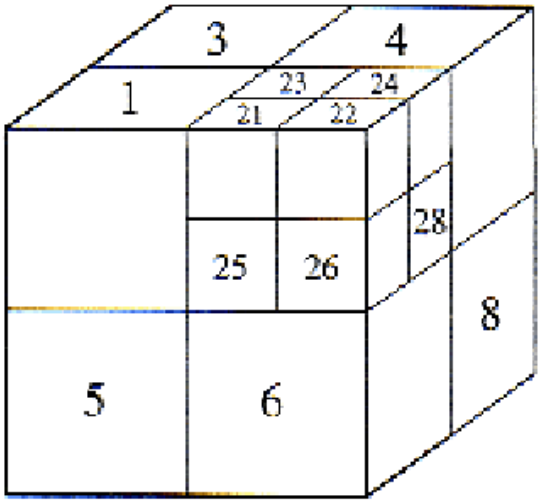
Laser-range scanners sample points from all sides of a real object. Connecting these vertices with triangles provides a surface representation (here: 400000 triangles).

9. Space Subdivision Techniques

For efficient rendering, it is crucial to decide which objects are visible, e.g. located in the viewing frustum. Space subdivision techniques provide efficient access to objects located in a certain region. Octrees are commonly used for subdividing 3D scenes.

Octrees

- start with one vlume element (voxel), e.g. bounding box for entire scene
- subdivide voxel into eight sub-voxels.
- recursively subdivide sub-voxels, until these are empty or have a prescribed resolution.



9. Space Subdivision Techniques

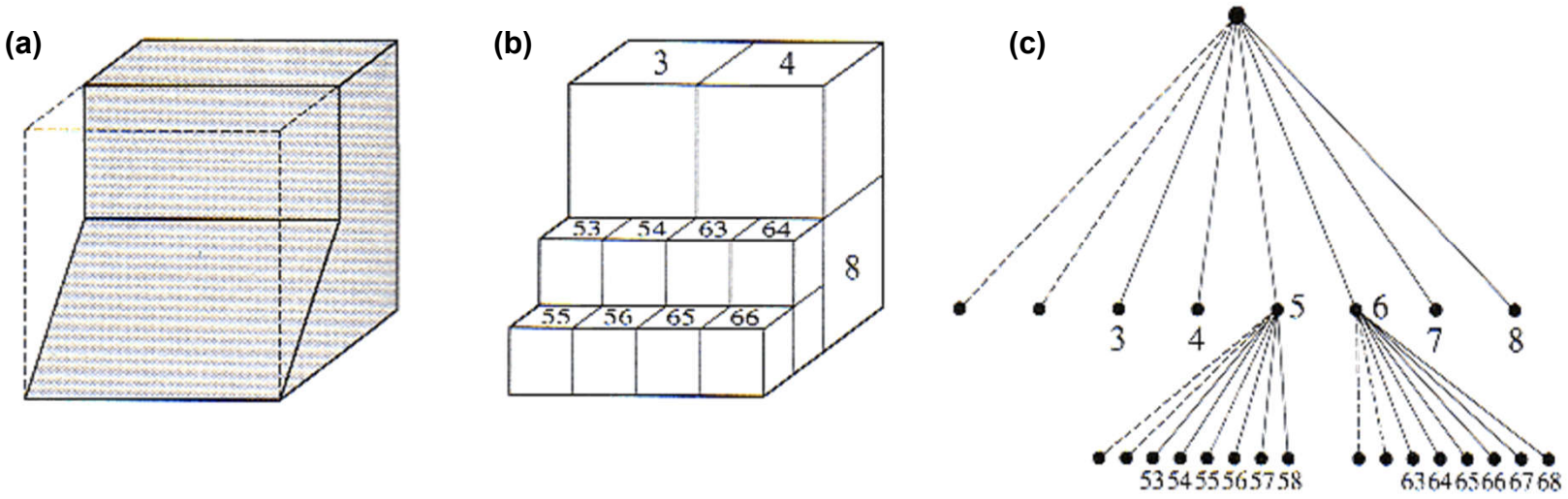
Octrees (cont'd)

Example: Octree representation of a solid

(a) solid within base voxel

(b) "inner" voxels after two levels of subdivision

(c) corresponding octree data structure.



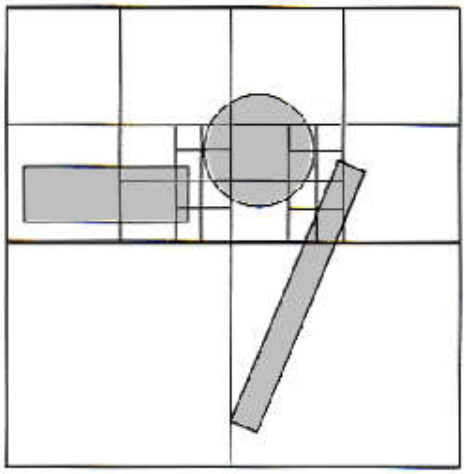
9. Space Subdivision Techniques

Quadtrees

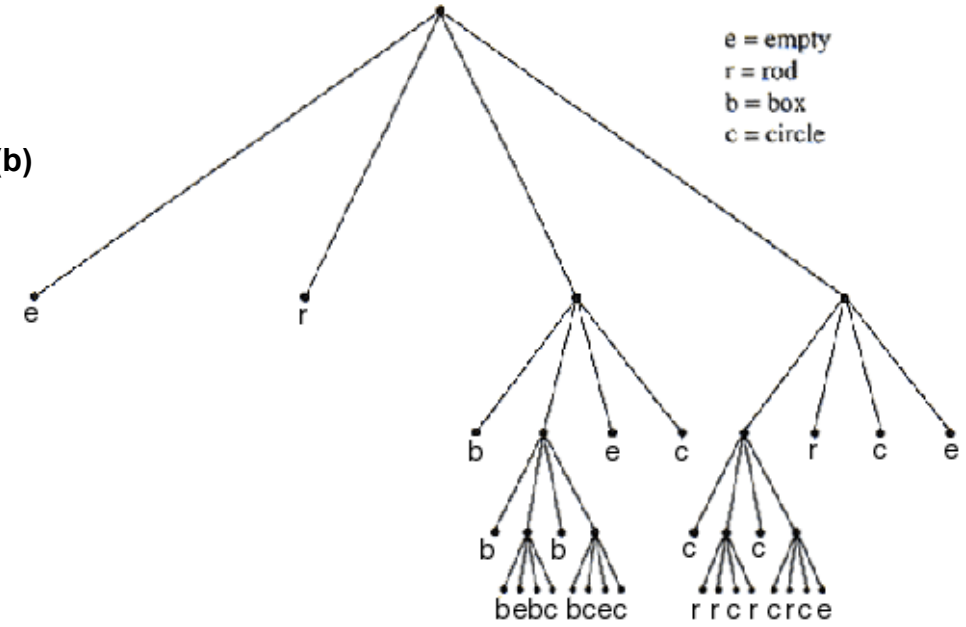
The 2D equivalent of octrees are quadtrees. Example:

(a) recursive subdivision of space
(b) corresponding quadtree structure

(a)



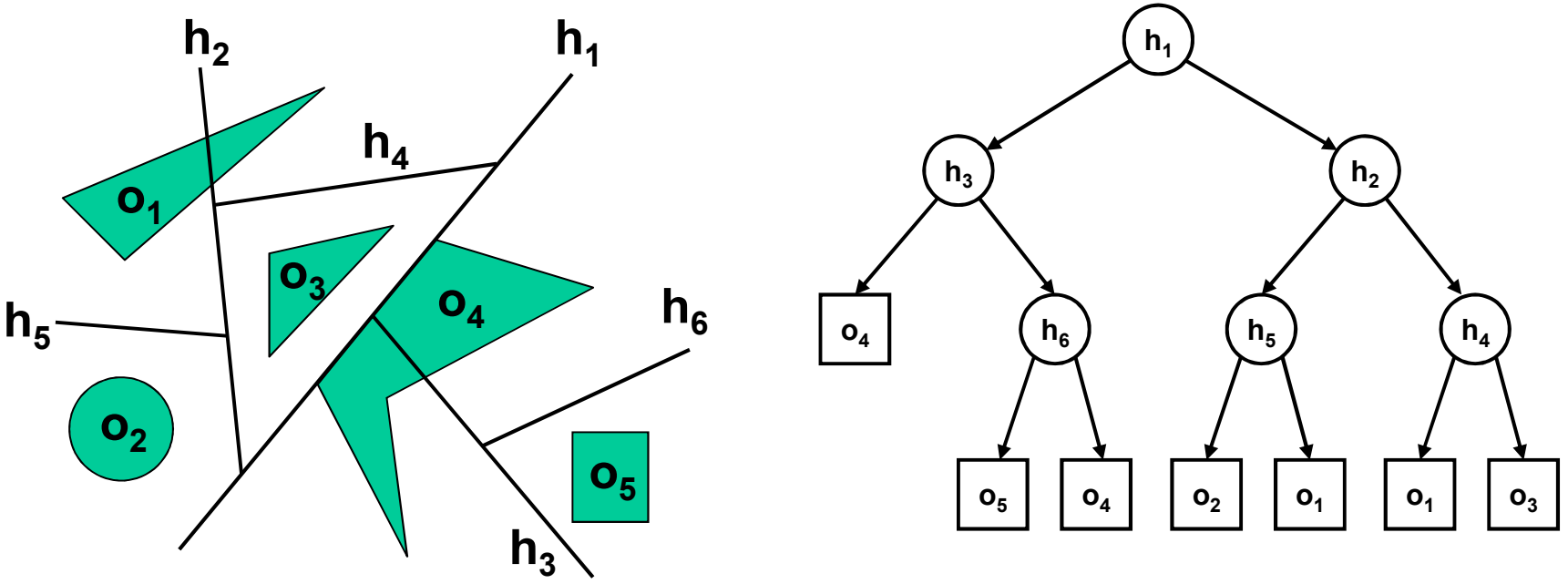
(b)



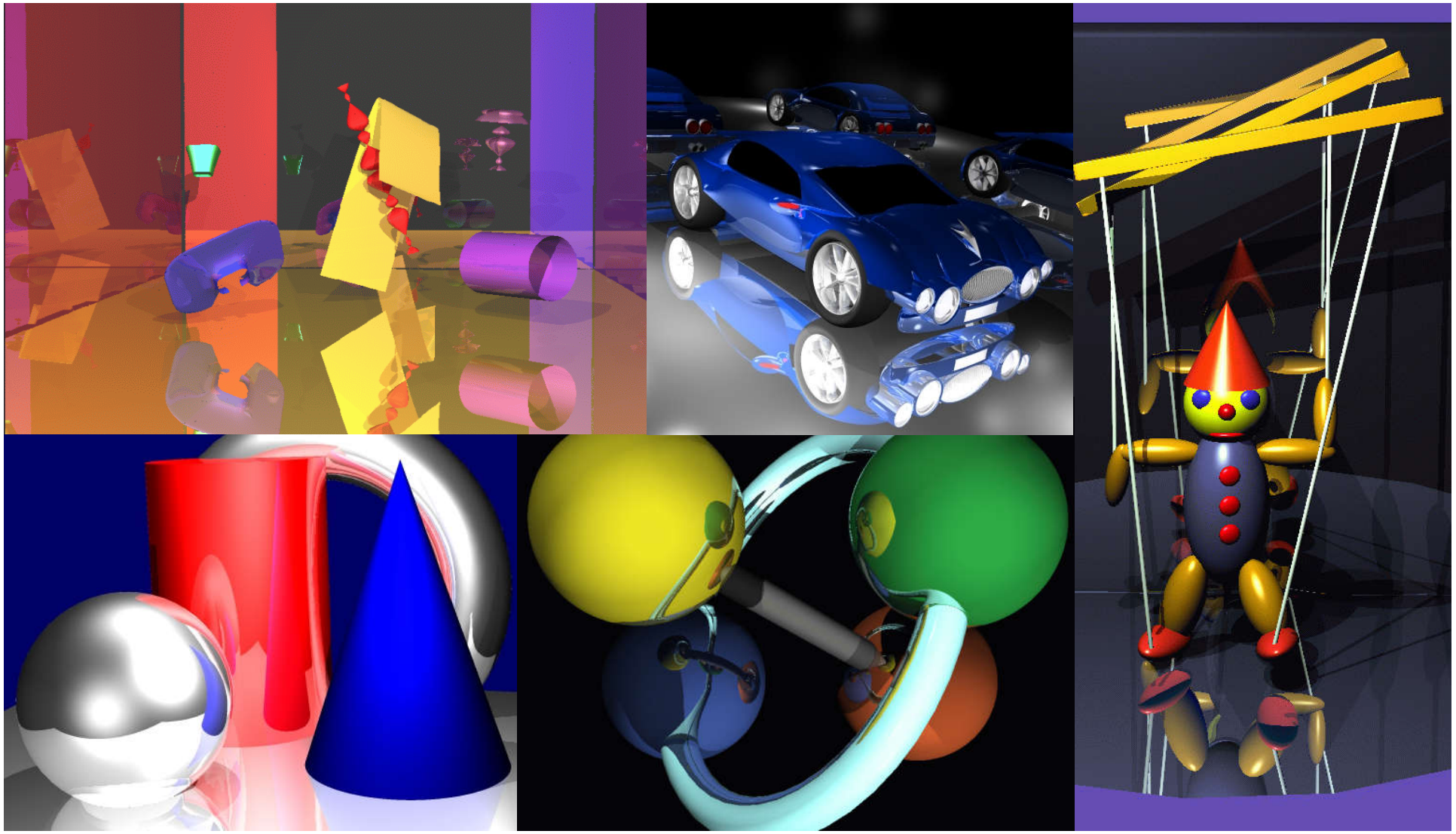
9. Space Subdivision Techniques

Binary Space-partition Trees (BSP-Trees)

BSP-trees recursively subdivide space along arbitrary planes / lines



10. Global Illumination: Raytracing



Books

- Dave Shreiner, et al., OpenGL Programming Guide (Red Book), Addison-Wesley, 2003, http://glprogramming.com/red/
- Randi J. Rost, et al. OpenGL Shading Language (Orange Book), 3rd edition, Pearson, 2009
- Graham Sellers et al., OpenGL Super Bible, 7th edition, Addison Wesley, 2015
- Rick Parent, *Computer Animation: Algorithms and Techniques*, 3rd edition, Morgan Kaufmann, 2012
- Gerald Farin, *Curves and Surfaces for CAGD*, 5th edition, Morgan Kaufmann, 2002
- John F. Hughes et al., *Computer Graphics: Principles and Practice, 3rd edition, Pearson, 2014 (TB 2018)*.
- Jorge Angeles, *Fundamentals of Robotic Mechanical Systems,* 4th edition, Springer, 2013

Books

- Alan Watt and Mark Watt, *Advanced Animation and Rendering Techniques,* Addison Wesley, 1992.
- Michael Bender and Manfred Brill, *Computergrafik* (in German), 2nd edition, Hanser 2005.
- Alfred Nischwitz et al., Computergrafik Band 1 (in German), 4th edition, Springer Vieweg, 2019