Introduction to OpenGL

By Nick Gnedin

Largely based on a lecture by
Prof. G. Wolberg, CCNY

If You Want to Learn How to Do This…

… You are in a wrong place!

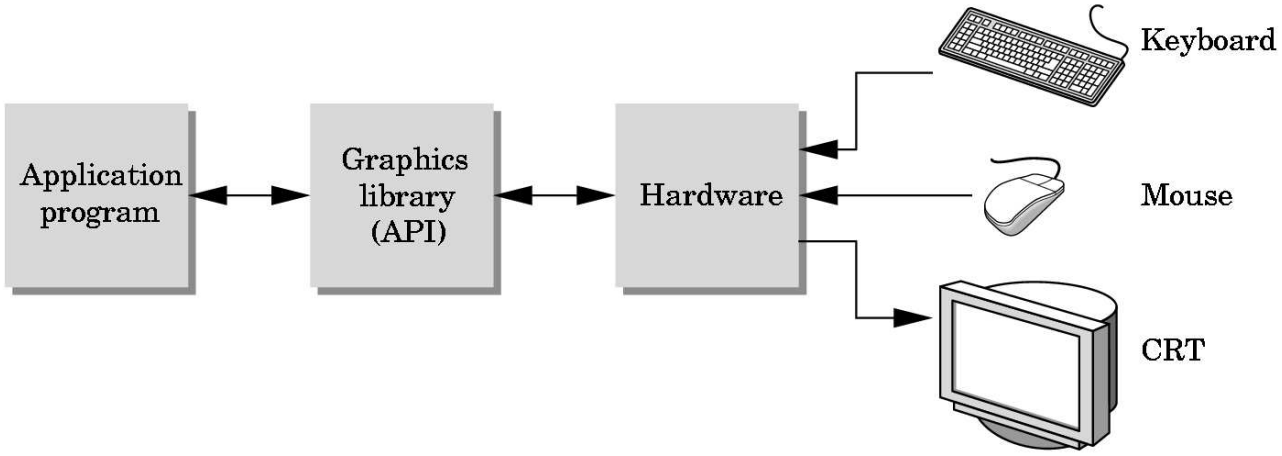
Overview

- What is OpenGL?
- Object Modeling
- Lighting and Shading
- Computer Viewing
- Rendering
- Texture Mapping
- Homogeneous Coordinates

What Is OpenGL?

The Programmer's Interface

- Programmer sees the graphics system through an interface: the Application Programmer Interface (API)



SGI and GL

- Silicon Graphics (SGI) revolutionized the graphics workstation by implementing the pipeline in hardware (1982)
- To use the system, application programmers used a library called GL
- With GL, it was relatively simple to program three dimensional interactive applications

OpenGL

- The success of GL lead to OpenGL in 1992, a platform-independent API that was
  - Easy to use
  - Close enough to the hardware to get excellent performance
  - Focused on rendering
  - Omitted windowing and input to avoid window system dependencies

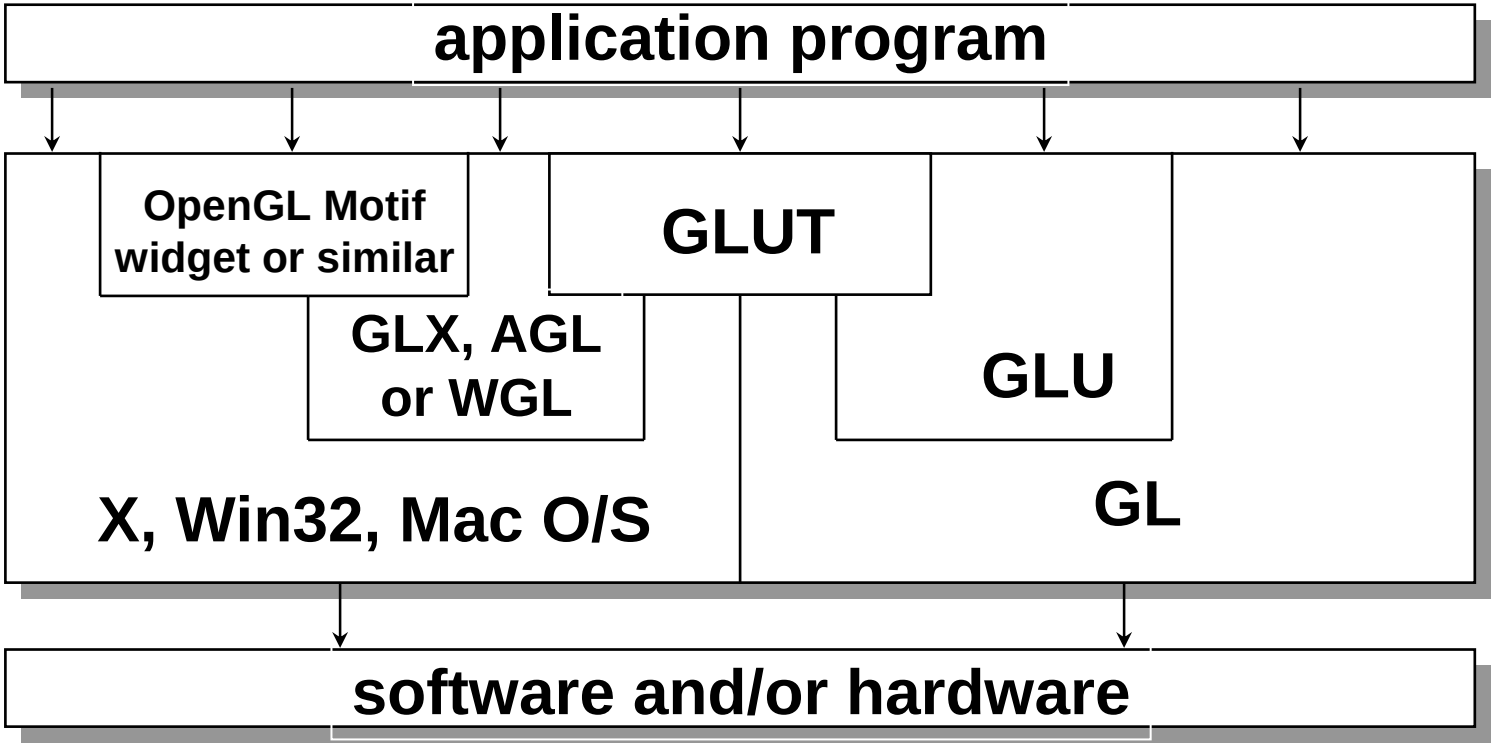
OpenGL Evolution

- Controlled by an Architectural Review Board (ARB)
  - Members include SGI, Microsoft, Nvidia, HP, 3DLabs,IBM,…….
  - Relatively stable (present version 1.4)
    - Evolution reflects new hardware capabilities
      - 3D texture mapping and texture objects
      - Vertex programs
  - Allows for platform specific features through extensions
  - See www.opengl.org for up-to-date info

OpenGL Libraries

- OpenGL core library
  - OpenGL32 on Windows
  - GL on most Unix/Linux systems
- OpenGL Utility Library (GLU)
  - Provides functionality in OpenGL core but avoids having to rewrite code
- Links with window system
  - GLX for X window systems
  - WGL for Windows
  - AGL for Macintosh

Software Organization



Windowing with OpenGL

- OpenGL is independent of any specific window system
- OpenGL can be used with different window systems
  - X windows (GLX)
  - MFC
  - …
- GLUT provide a portable API for creating window and interacting with I/O devices

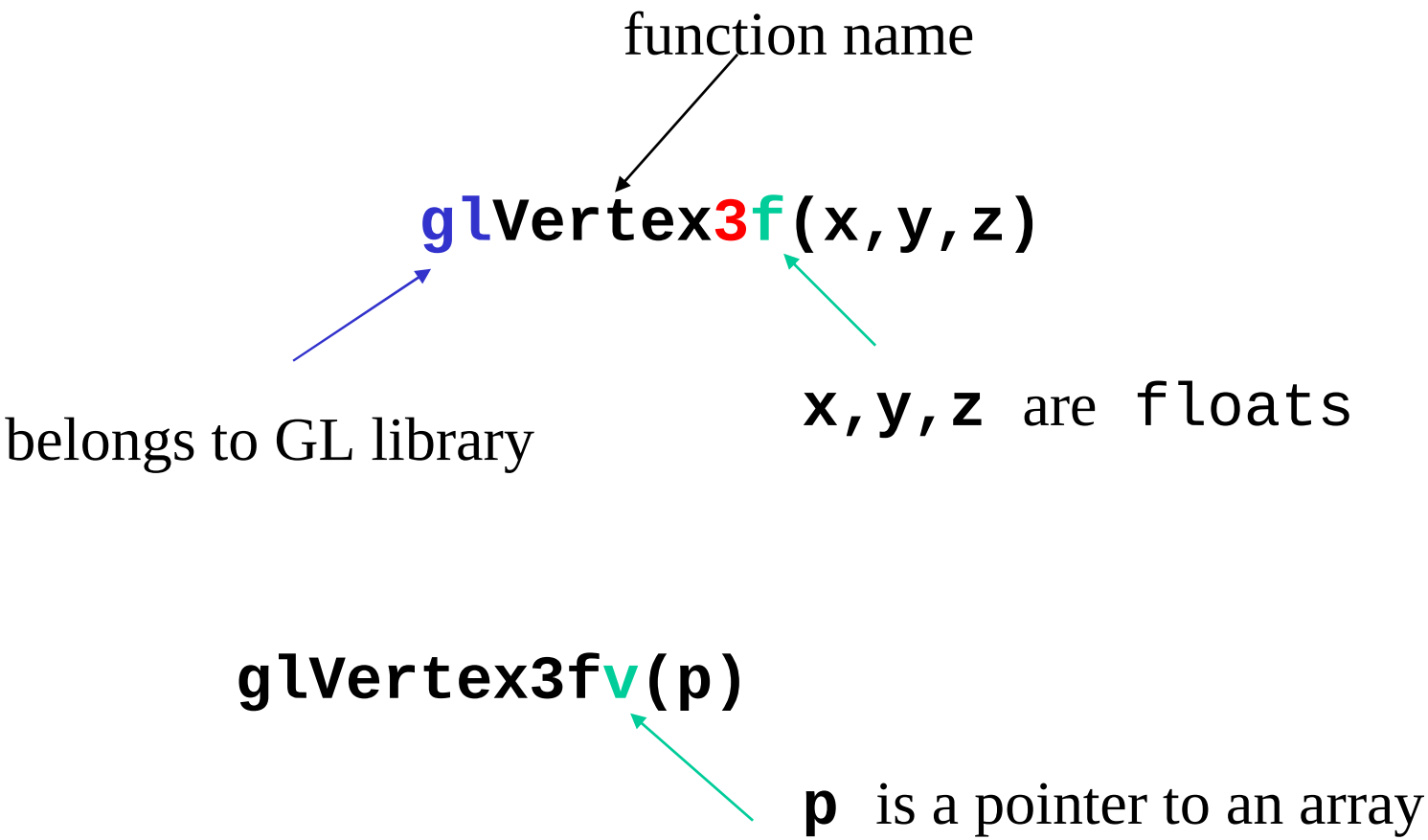
API Contents

- Functions that specify what we need to form an image
  - Objects
  - Viewer (camera)
  - Light Source(s)
  - Materials
- Other information
  - Input from devices such as mouse and keyboard
  - Capabilities of system

OpenGL State

- OpenGL is a state machine
- OpenGL functions are of two types
  - Primitive generating
    - Can cause output if primitive is visible
    - How vertices are processed and appearance of primitive are controlled by the state
  - State changing
    - Transformation functions
    - Attribute functions

OpenGL function format

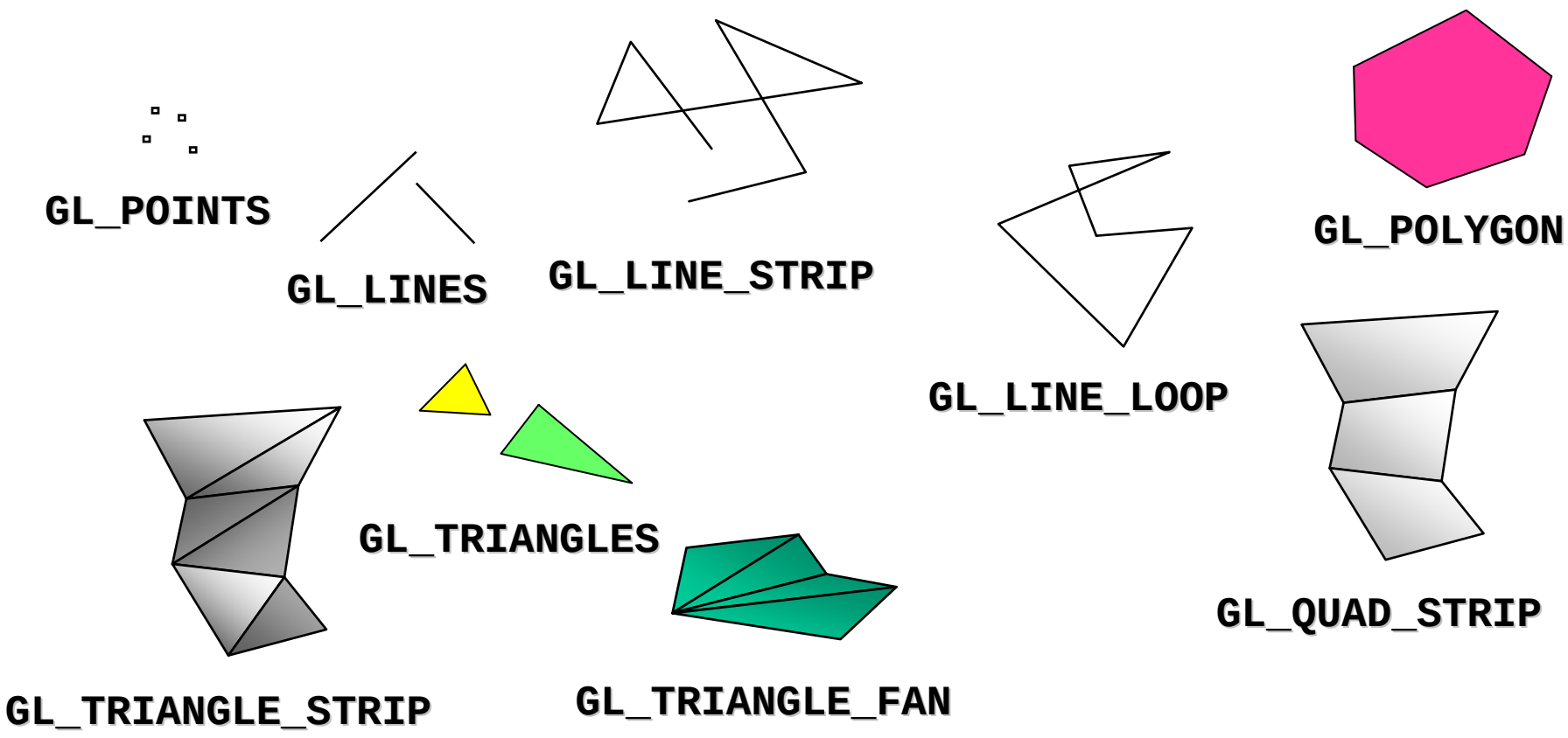


OpenGL #defines

- Most constants are defined in the include files **`gl.h`**, **`glu.h`** and **`glut.h`**
  - Note **`#include <glut.h>`** should automatically include the others
  - Examples
  - **`glBegin(GL_POLYGON)`**
  - **`glClear(GL_COLOR_BUFFER_BIT)`**
- include files also define OpenGL data types: **`Glfloat`**, **`Gldouble`**,….

Object Modeling

OpenGL Primitives



Example: Drawing an Arc

- Given a circle with radius *r,* centered at (*x*,*y*), draw an arc of the circle that sweeps out an angle θ.

$$(x, y) = (x_0 + r\cos\theta, y_0 + r\sin\theta),$$

$$\text{for } 0 \le \theta \le 2\pi.$$

The Line Strip Primitive

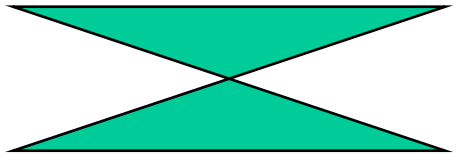
```
void drawArc(float x, float y, float r,
             float t0, float sweep)
{
      float t, dt;       /* angle */
      int   n = 30;      /* # of segments */
      int   i;

      t = t0 * PI/180.0;        /* radians */
      dt = sweep * PI/(180*n); /* increment */

      glBegin(GL_LINE_STRIP);
      for(i=0; i<=n; i++, t += dt)
            glVertex2f(x + r*cos(t), y + r*sin(t));
      glEnd();
}
```

Polygon Issues

- OpenGL will only display polygons correctly that are
  - Simple: edges cannot cross
  - Convex: All points on line segment between two points in a polygon are also in the polygon
  - Flat: all vertices are in the same plane
- User program must check if above true
- Triangles satisfy all conditions



nonsimple polygon

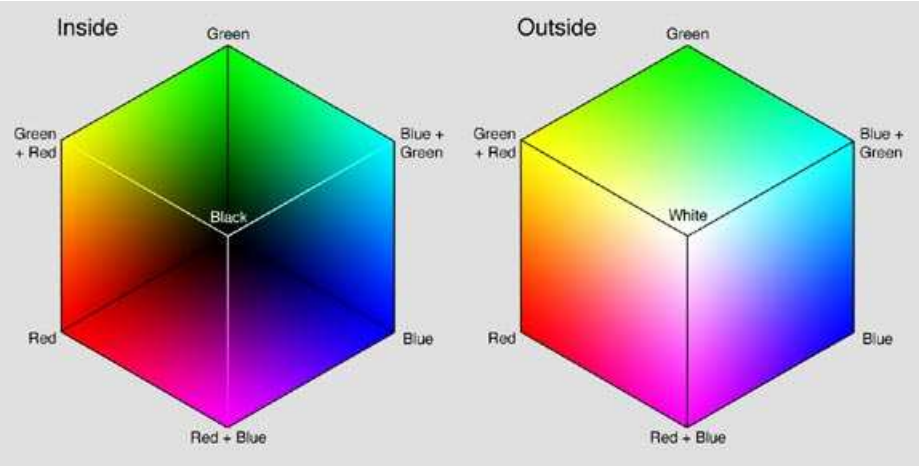
nonconvex polygon

Attributes

- Attributes are part of the OpenGL and determine the appearance of objects
  - Color (points, lines, polygons)
  - Size and width (points, lines)
  - Stipple pattern (lines, polygons)
  - Polygon mode
    - Display as filled: solid color or stipple pattern
    - Display edges

RGB color

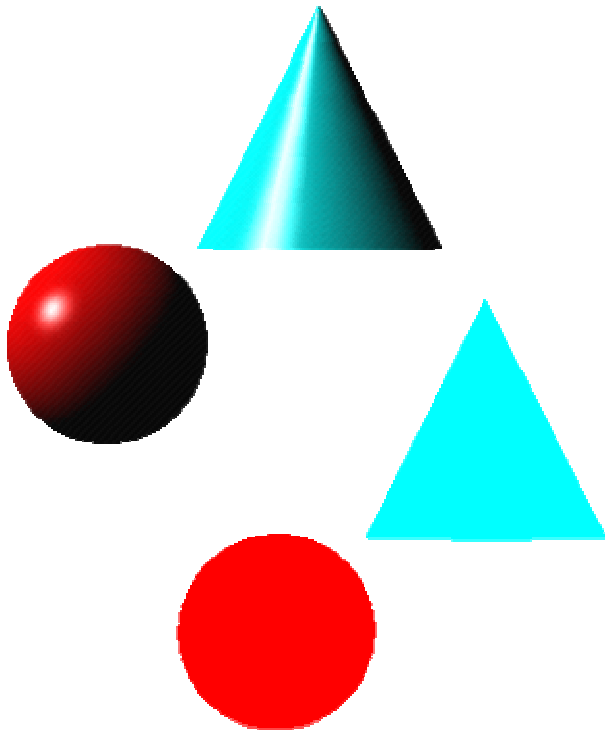
- Each color component stored separately (usually 8 bits per component)
- In OpenGL color values range from 0.0 (none) to 1.0 (all).



Lighting and Shading

Lighting Principles

- Lighting simulates how objects reflect light
  - material composition of object
  - light's color and position
  - global lighting parameters
    - ambient light
    - two sided lighting
  - available in both color index and RGBA mode

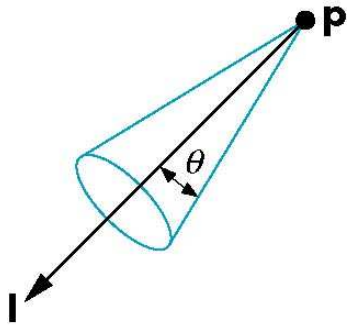


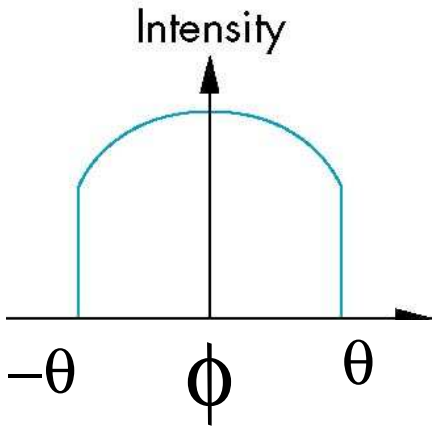
Types of Lights

- OpenGL supports two types of Lights
  - Local (Point) light sources
  - Infinite (Directional) light sources
- In addition, it has one global ambient light that emanates from everywhere in space (like glowing fog)
- A point light can be a *spotlight*

Spotlights

- Have:
  - Direction (vector)
  - Cutoff (cone opening angle)
  - Attenuation with angle





Moving Light Sources

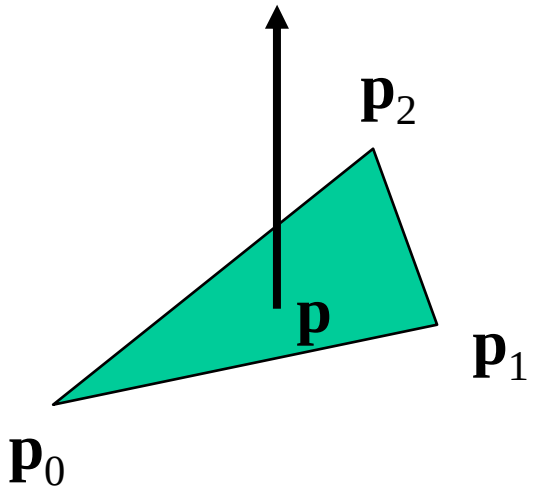
- Light sources are geometric objects whose positions or directions are user-defined
- Depending on where we place the position (direction) setting function, we can
  - Move the light source(s) with the object(s)
  - Fix the object(s) and move the light source(s)
  - Fix the light source(s) and move the object(s)
  - Move the light source(s) and object(s) independently

Steps in OpenGL shading

1. Enable shading and select model
2. Specify normals
3. Specify material properties
4. Specify lights

Normals

- In OpenGL the normal vector is part of the state
- Usually we want to set the normal to have unit length so cosine calculations are correct



Note that right-hand rule determines outward face

Polygonal Shading

- Shading calculations are done for each vertex; vertex colors become vertex shades
- By default, vertex colors are interpolated across the polygon (so-called *Phong* model)
- With *flat* shading the color at the first vertex will determine the color of the whole polygon

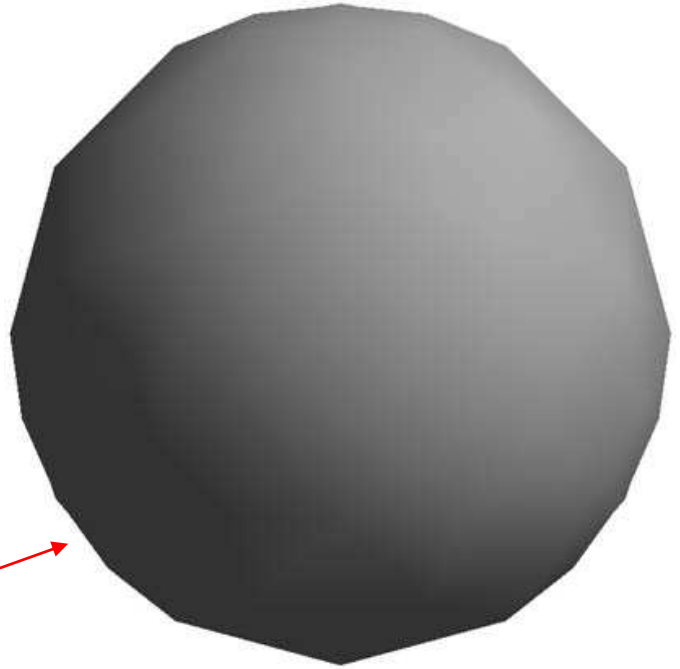
Polygon Normals

Consider model of sphere:

- Polygons have a single normal
- We have different normals at each vertex even though this concept is not quite correct mathematically

Smooth Shading

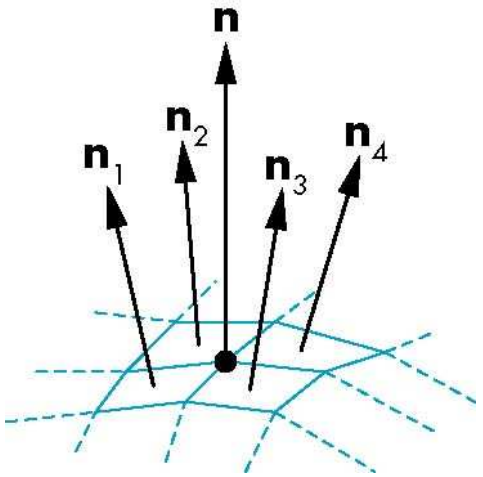
- We can set a new normal at each vertex
- Easy for sphere model
  - If centered at origin $\mathbf{n} = \mathbf{p}$
- Now smooth shading works
- Note *silhouette edge*



Mesh Shading

- The previous example is not general because we knew the normal at each vertex analytically
- For polygonal models, Gouraud proposed to use the average of normals around a mesh vertex

$$\mathbf{n} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4}{|\mathbf{n}_1| + |\mathbf{n}_2| + |\mathbf{n}_3| + |\mathbf{n}_4|}$$



Gouraud and Phong Shading

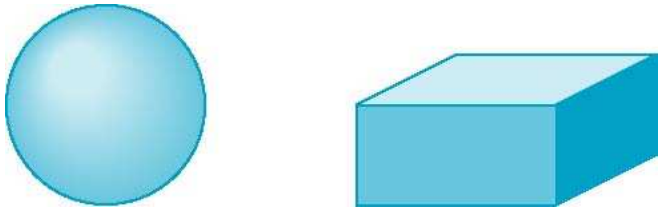
- Gouraud Shading
  - Find average normal at each vertex (vertex normals)
  - Apply Phong model at each vertex
  - Interpolate vertex shades across each polygon
- Phong shading
  - Find vertex normals
  - Interpolate vertex normals across edges
  - Find shades along edges
  - Interpolate edge shades across polygons

Comparison

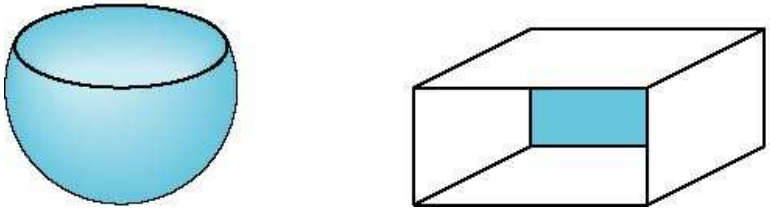
- If the polygon mesh approximates surfaces with a high curvatures, Phong shading may look smooth while Gouraud shading may show edges
- Phong shading requires much more work than Gouraud shading
  - Usually not available in real time systems
- Both need data structures to represent meshes so we can obtain vertex normals

Front and Back Faces

- The default is shade only front faces which works correct for convex objects
- If we set two sided lighting, OpenGL will shaded both sides of a surface
- Each side can have its own properties



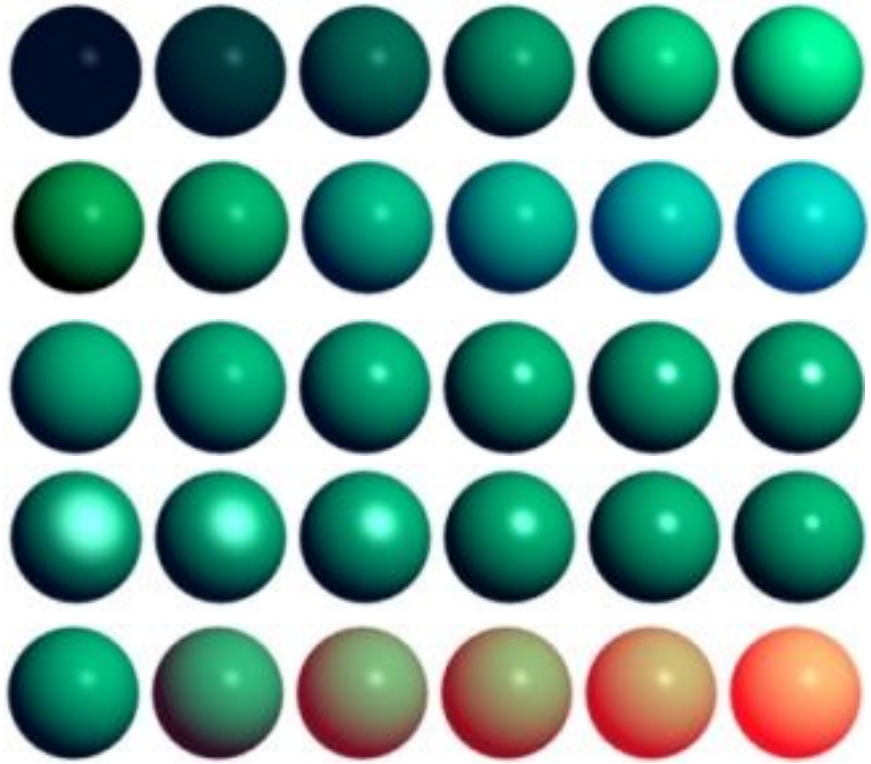
back faces not visible



back faces visible

Material Properties

- Define the surface properties of a primitive (separate materials for front and back)
  - Diffuse
  - Ambient
  - Specular
  - Shininess
  - Emission



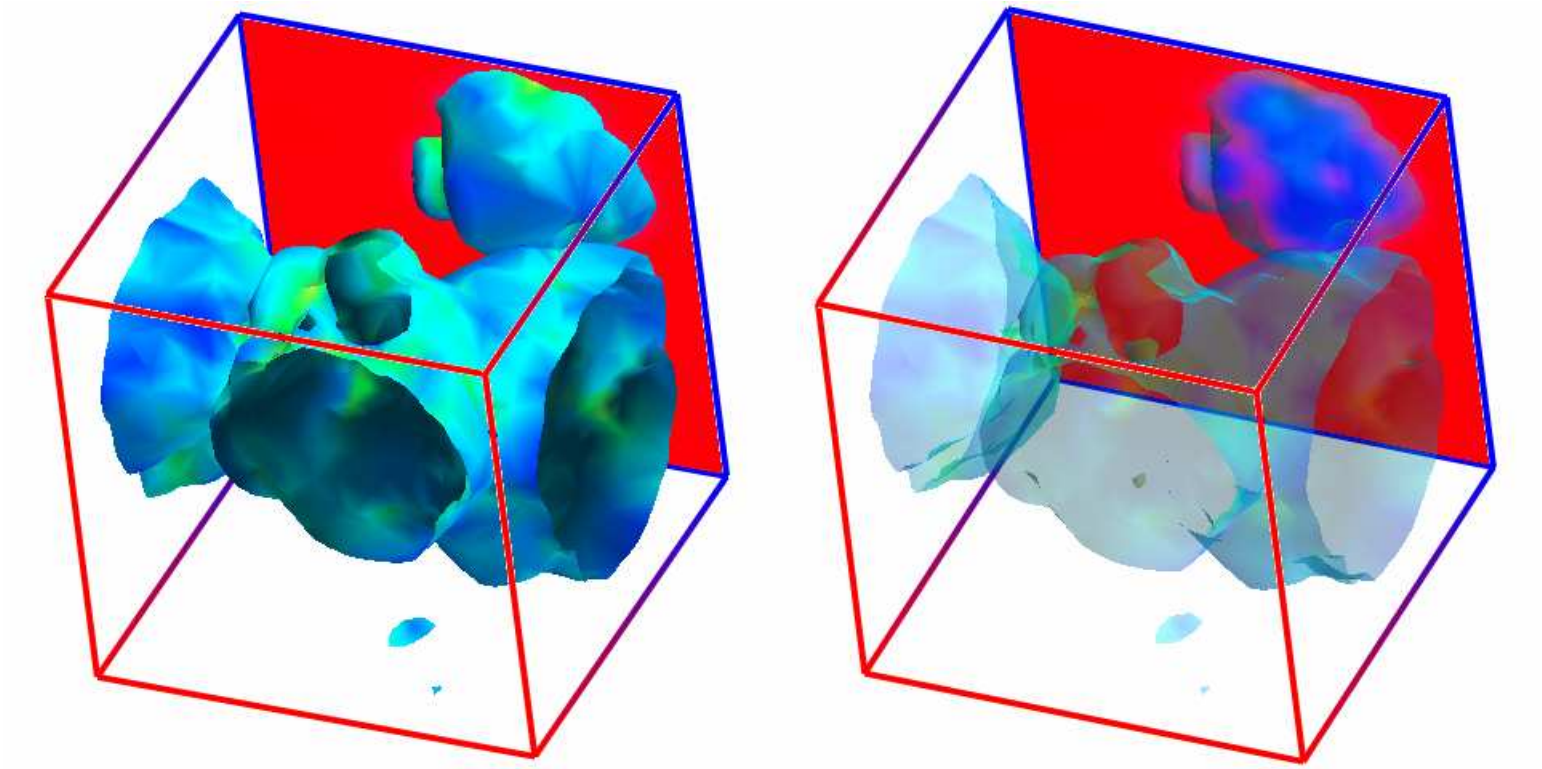
Emissive Term

- We can simulate a light source in OpenGL by giving a material an emissive component
- This color is unaffected by any sources or transformations

Transparency

- Material properties are specified as RGBA values
- The A (also called *alpha-value*) value can be used to make the surface translucent
- The default is that all surfaces are opaque

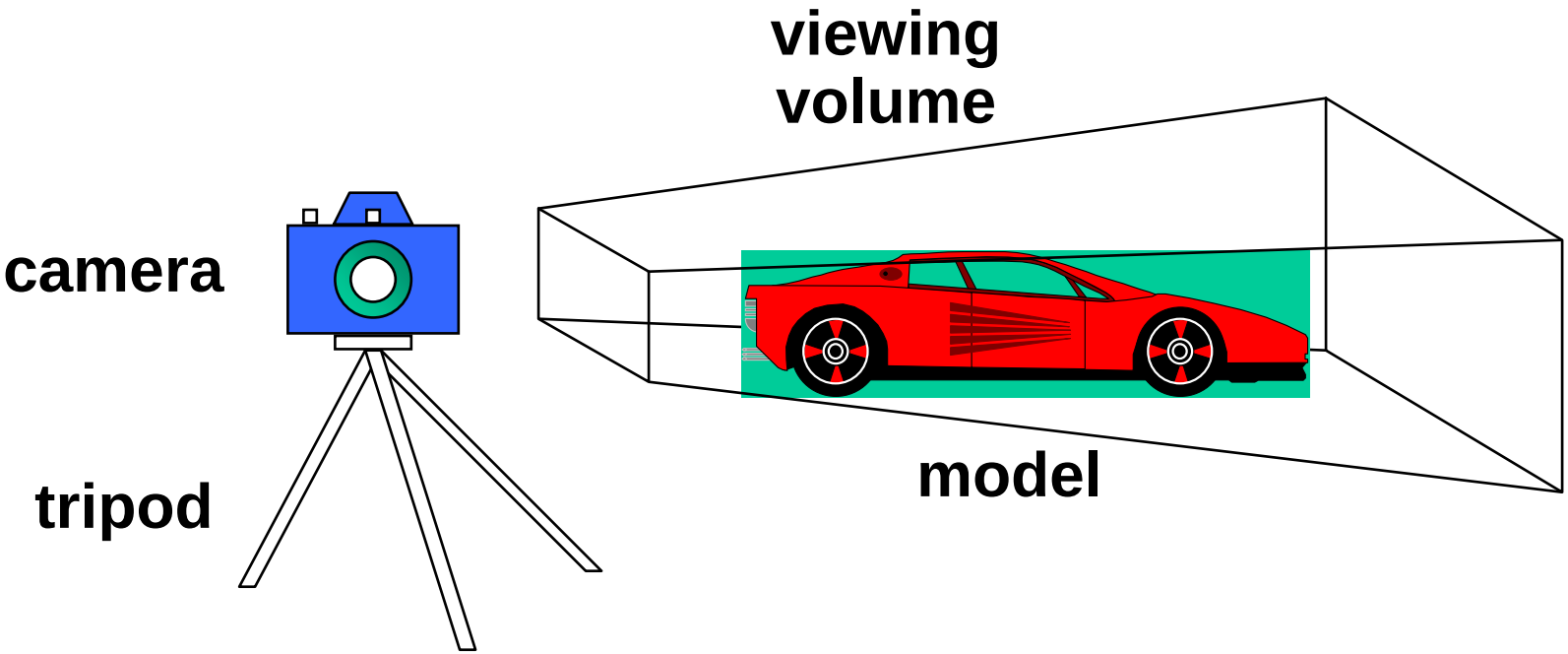
Transparency



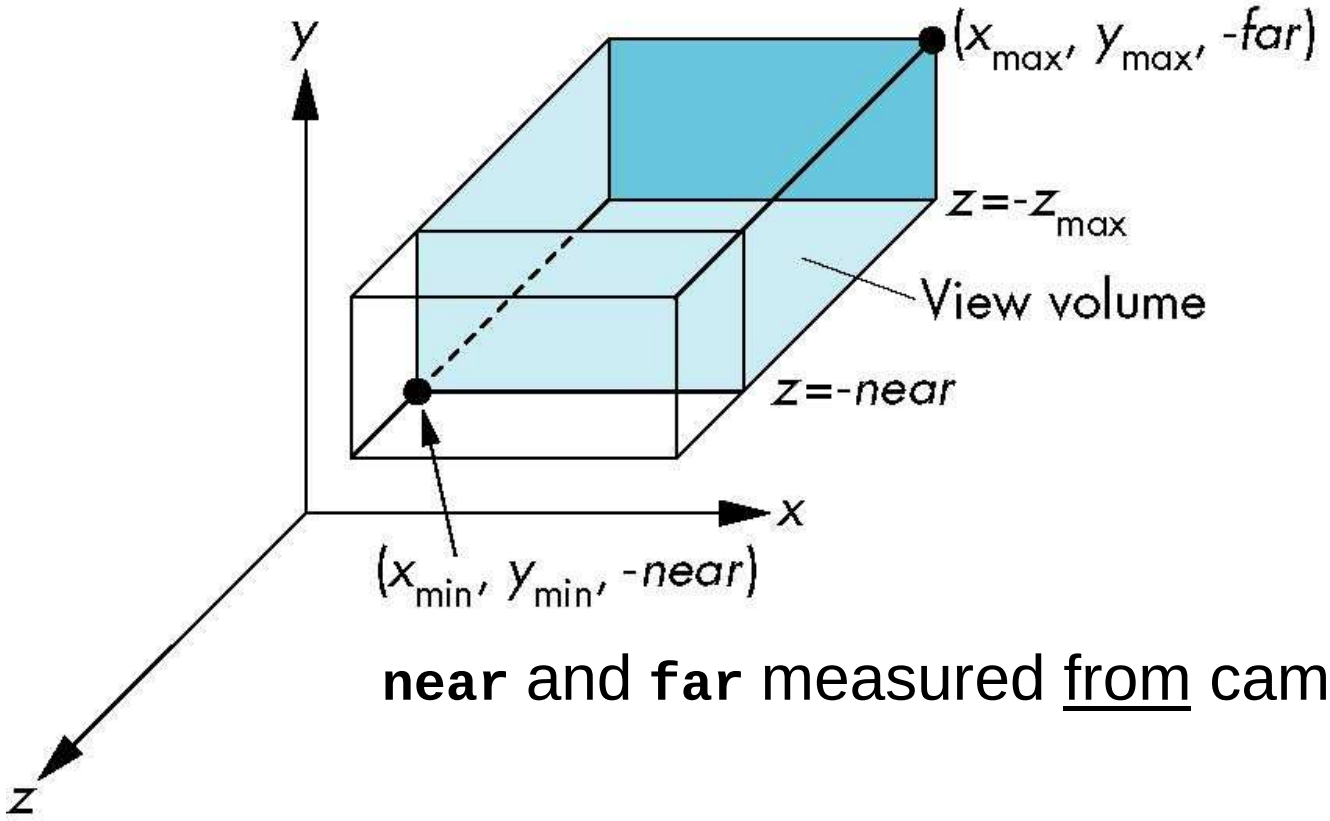
Computer Viewing

Camera Analogy

- 3D is just like taking a photograph (lots of photographs!)

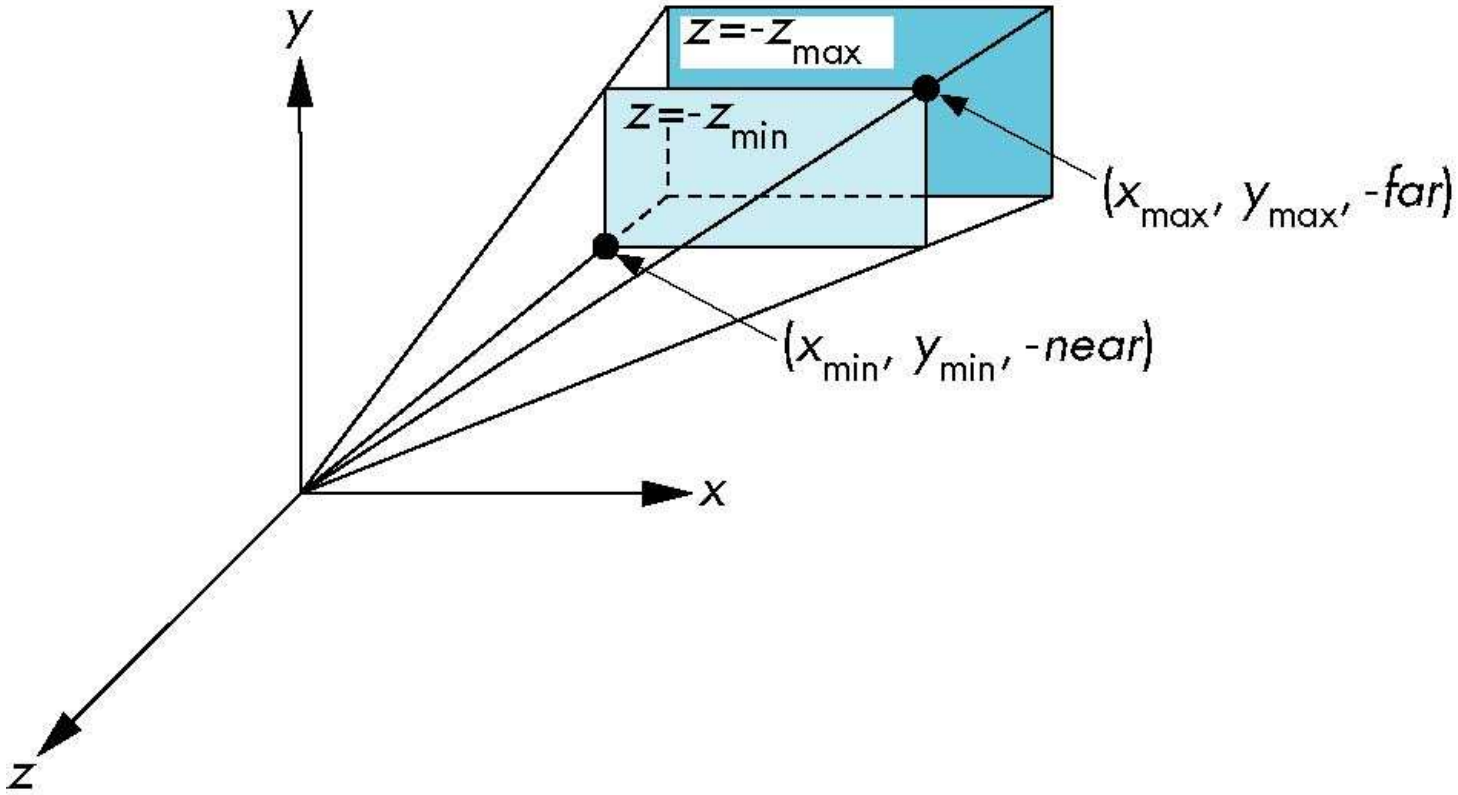


OpenGL Orthogonal (Parallel) Projection



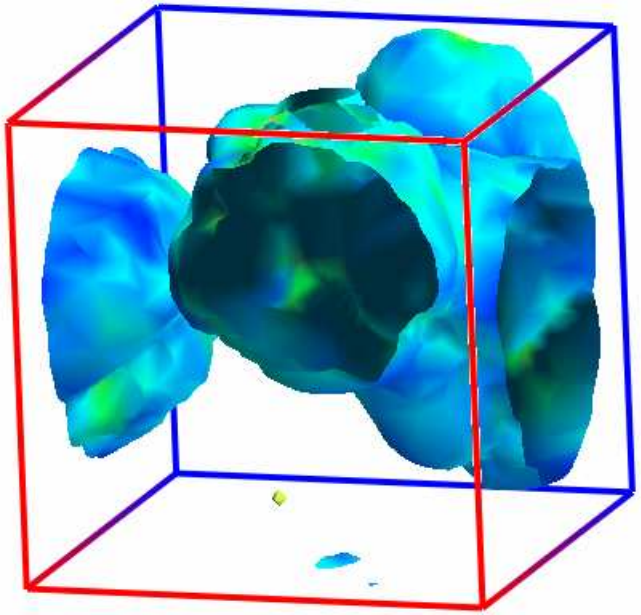
near and **far** measured from camera

OpenGL Perspective Projection

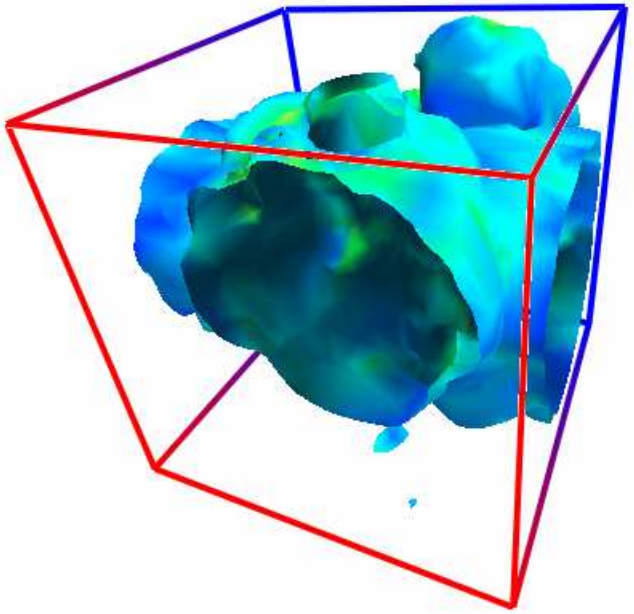


Projections

Orthogonal/Parallel

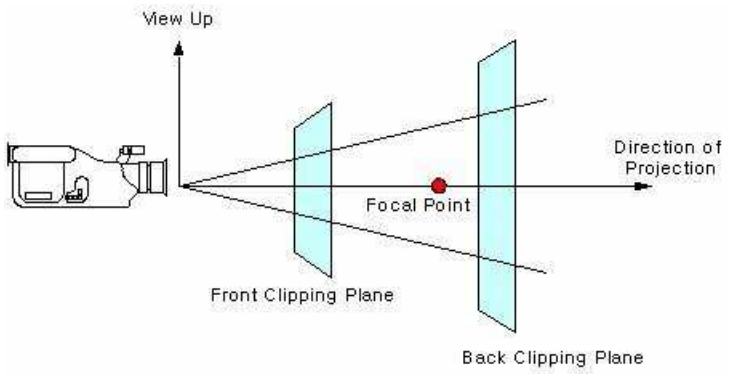


Perspective



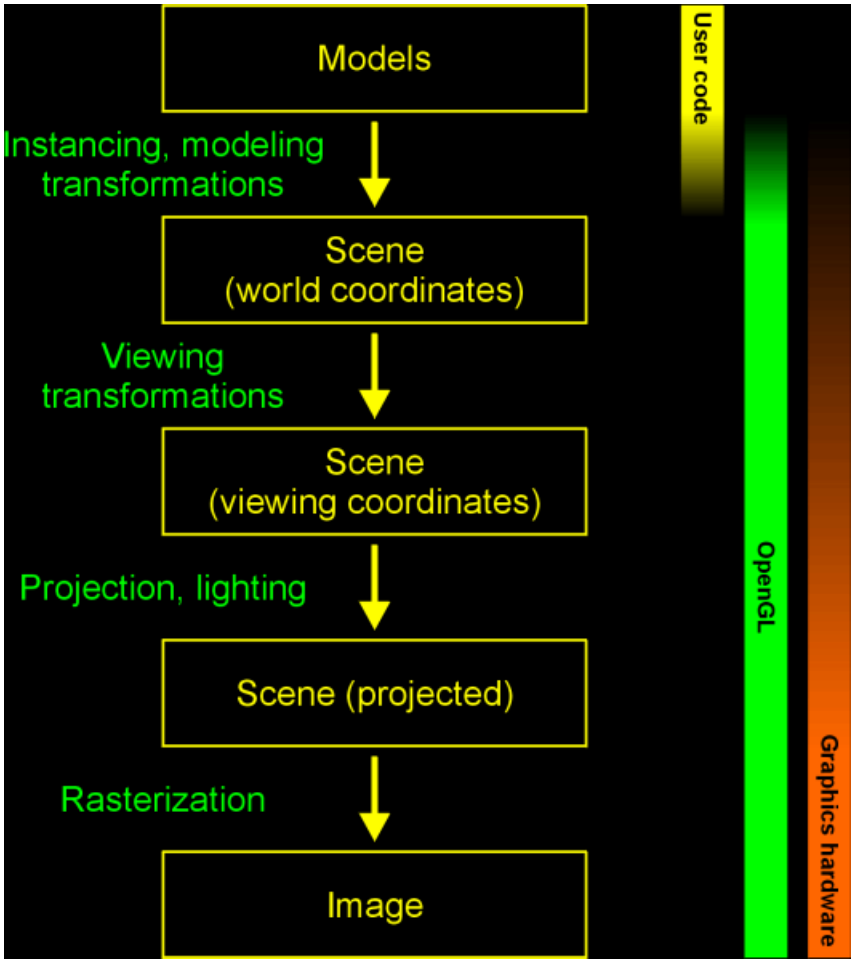
Clipping

- Just as a real camera cannot “see” the whole world, the virtual camera can only see part of the world space
  - Objects that are not within this volume are said to be *clipped* out of the scene



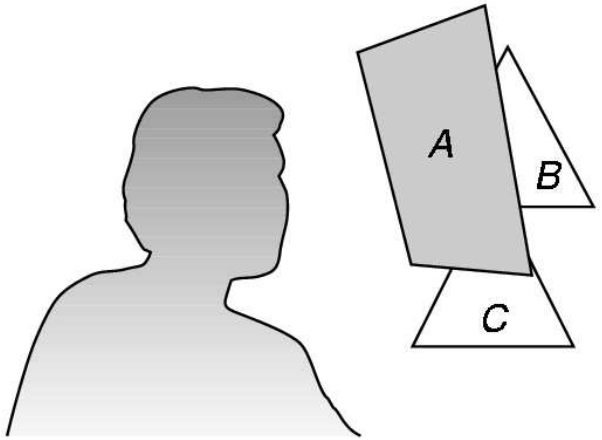
Rendering

Rendering Process



Hidden-Surface Removal

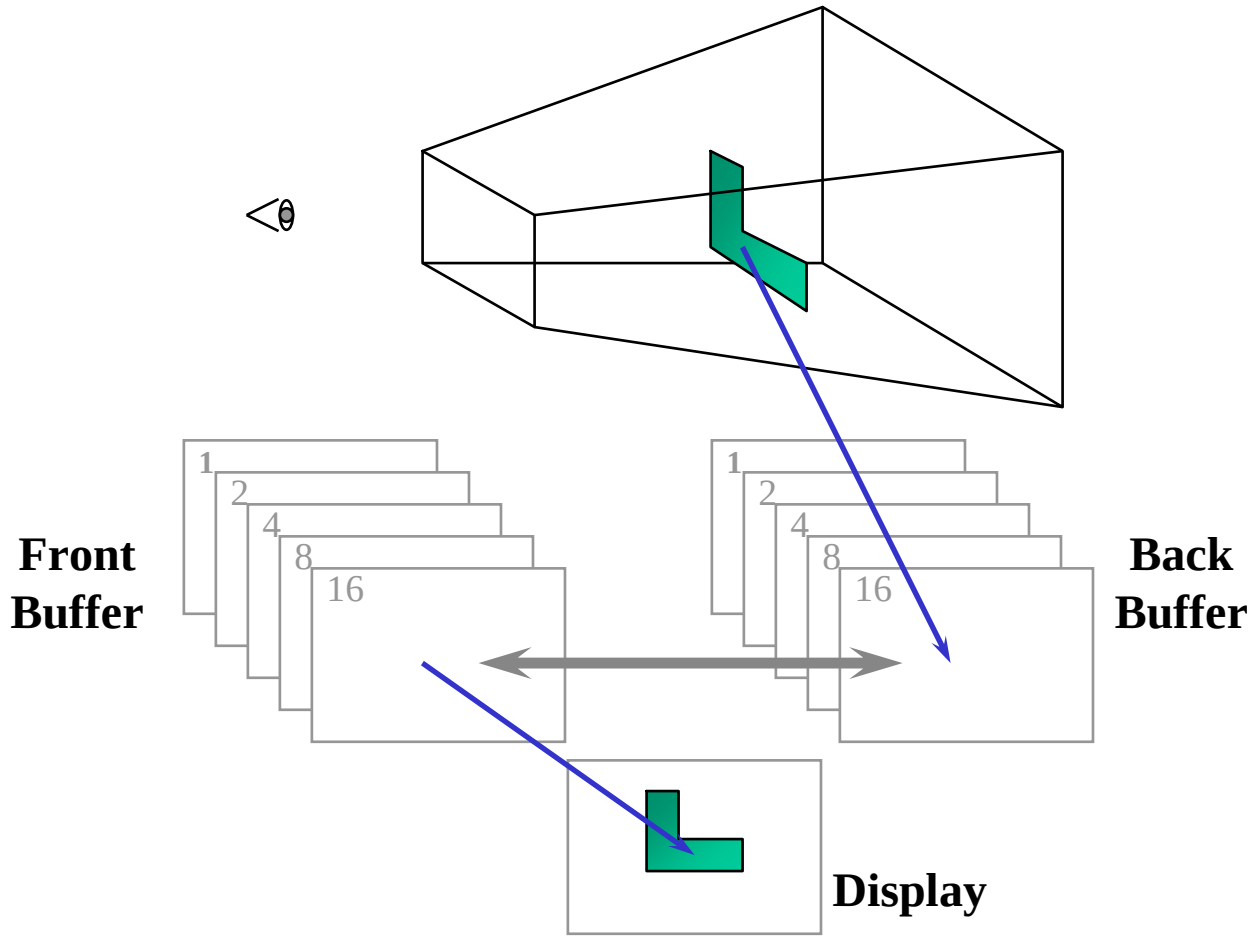
- We want to see only those surfaces in front of other surfaces
- OpenGL uses a *hidden-surface* method called the *z*-buffer algorithm that saves depth information as objects are rendered so that only the front objects appear in the image



Rasterization

- If an object is visible in the image, the appropriate pixels must be assigned colors
  - Vertices assembled into objects
  - Effects of lights and materials must be determined
  - Polygons filled with interior colors/shades
  - Must have also determined which objects are in front (hidden surface removal)

Double Buffering



Immediate and Retained Modes

- In a standard OpenGL program, once an object is rendered there is no memory of it and to redisplay it, we must re-execute the code for creating it
  - Known as *immediate mode graphics*
  - Can be especially slow if the objects are complex and must be sent over a network
- Alternative is define objects and keep them in some form that can be redisplayed easily
  - *Retained mode graphics*
  - Accomplished in OpenGL via *display lists*

Display Lists

- Conceptually similar to a graphics file
  - Must define (name, create)
  - Add contents
  - Close
- In client-server environment, display list is placed on server
  - Can be redisplayed without sending primitives over network each time

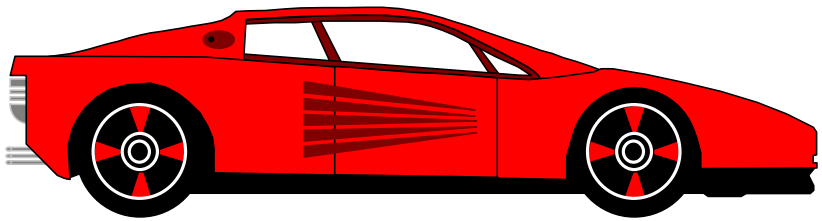
Display Lists and State

- Most OpenGL functions can be put in display lists
- State changes made inside a display list persist after the display list is executed
- If you think of OpenGL as a special computer language, display lists are its subroutines
- Rule of thumb of OpenGL programming: **Keep your display lists!!!**

Hierarchy and Display Lists

- Consider model of a car
  - Create display list for chassis
  - Create display list for wheel

```
glNewList( CAR, GL_COMPILE );
  glCallList( CHASSIS );
  glTranslatef( … );
  glCallList( WHEEL );
  glTranslatef( … );
  glCallList( WHEEL );
        …
glEndList();
```

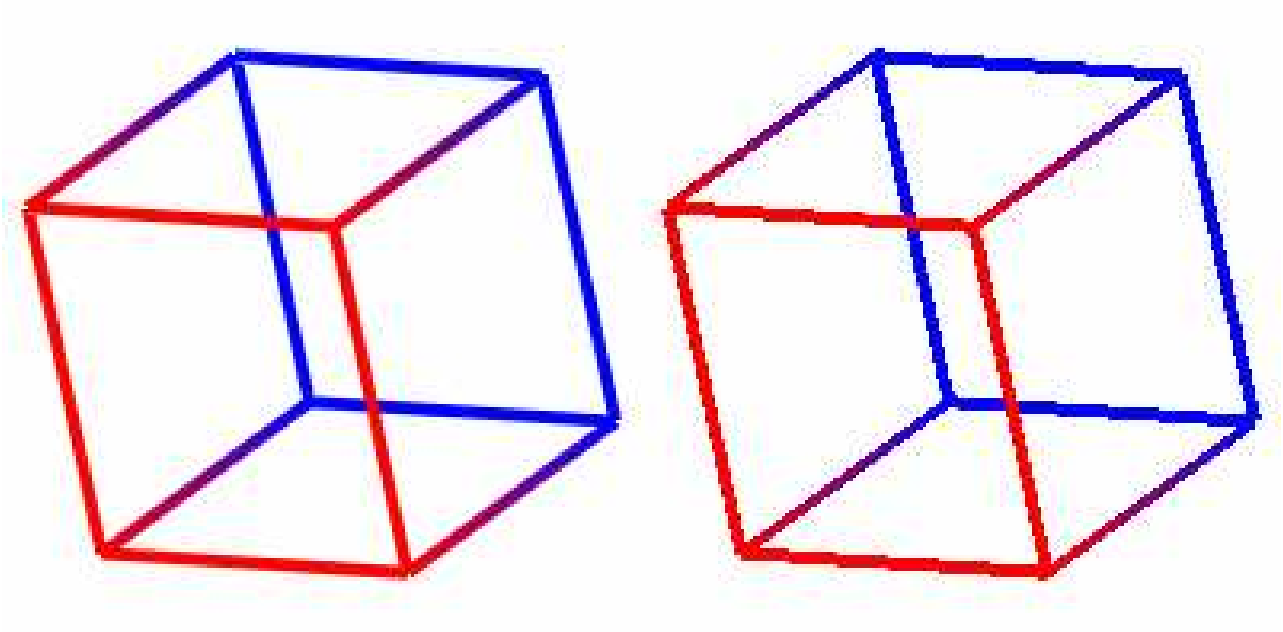


Antialiasing

- Removing the Jaggies

  `glEnable( `*`mode`*` )`

  - **`GL_POINT_SMOOTH`**
  - **`GL_LINE_SMOOTH`**
  - **`GL_POLYGON_SMOOTH`**



Texture Mapping

The Limits of Geometric Modeling

- Although graphics cards can render over 10 million polygons per second, that number is insufficient for many phenomena
  - Clouds
  - Grass
  - Terrain
  - Skin

Modeling an Orange

- Consider the problem of modeling an orange (the fruit)
- Start with an orange-colored sphere: too simple
- Replace sphere with a more complex shape:

  - Does not capture surface characteristics (small dimples)
  - Takes too many polygons to model all the dimples

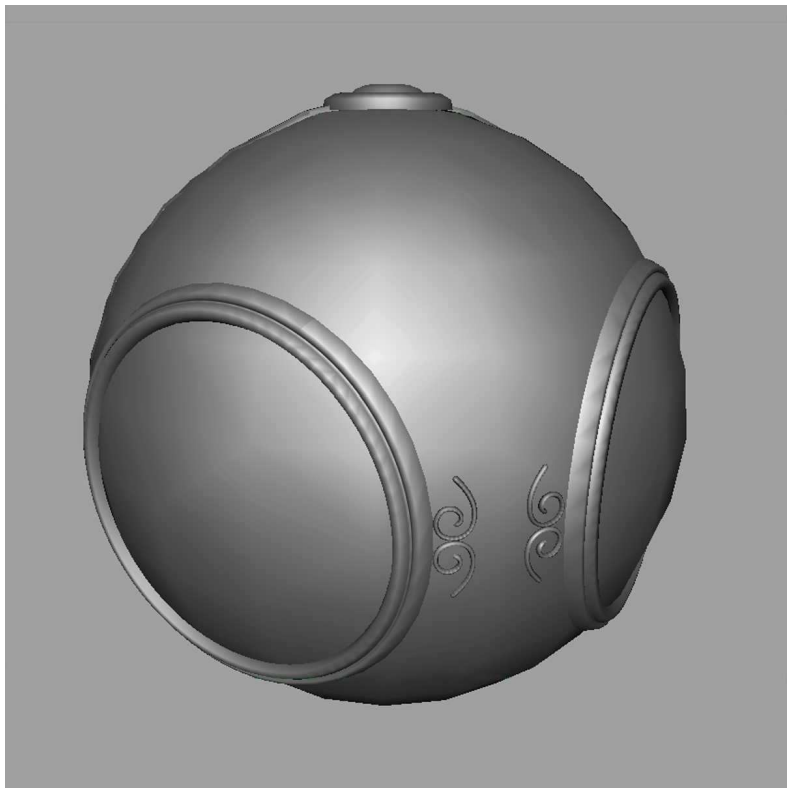
Modeling an Orange (2)

- Take a picture of a real orange, scan it, and “paste” onto simple geometric model
  - This process is called *texture mapping*
- Still might not be sufficient because resulting surface will be smooth
  - Need to change local shape
  - Bump mapping

Three Types of Mapping

- Texture Mapping
  - Uses images to fill inside of polygons
- Environmental (reflection mapping)
  - Uses a picture of the environment for texture maps
  - Allows simulation of highly specular surfaces
- Bump mapping
  - Emulates altering normal vectors during the rendering process

Texture Mapping

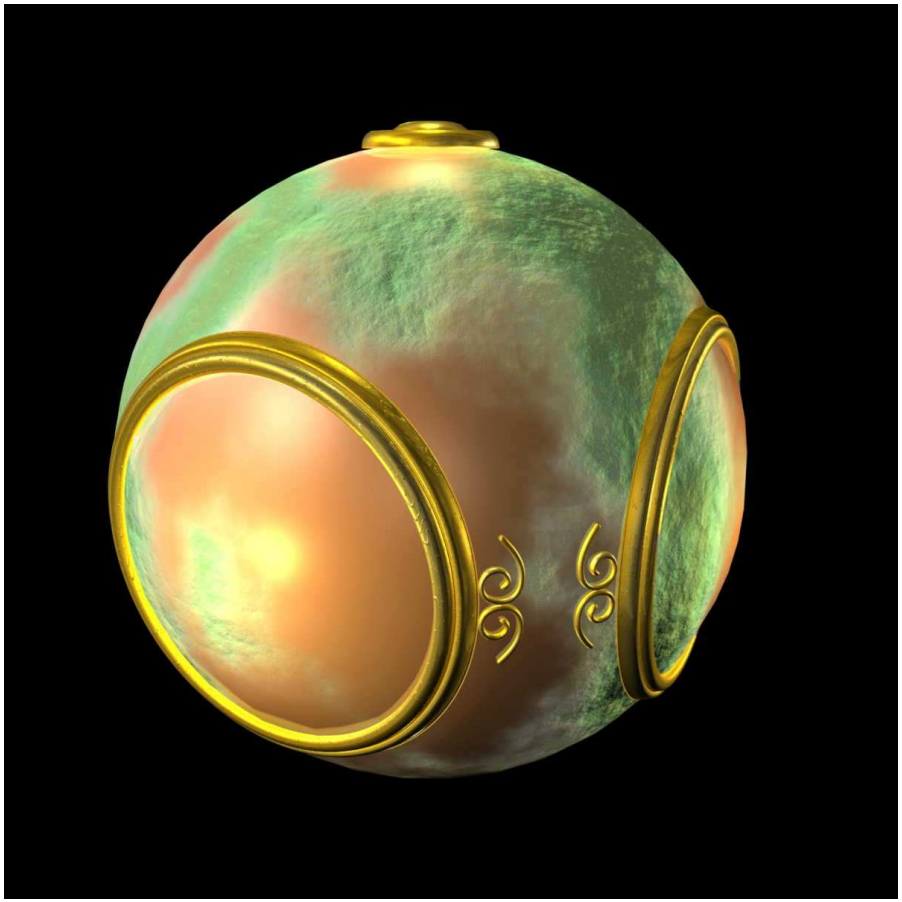


geometric model

texture mapped

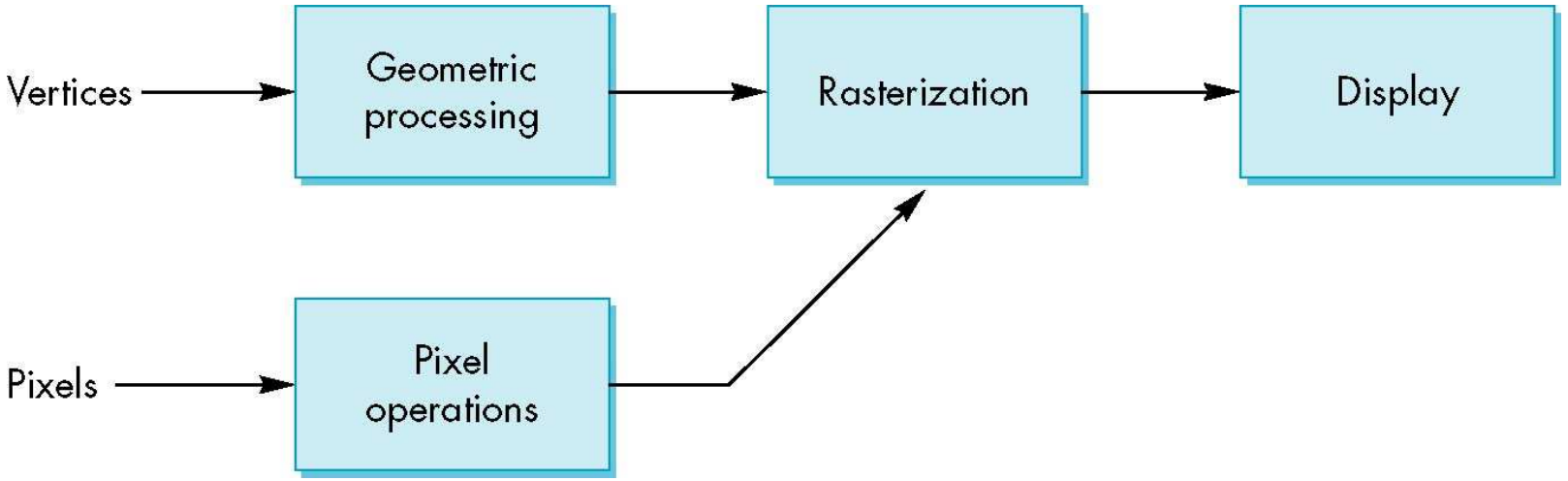
Environment Mapping

Bump Mapping



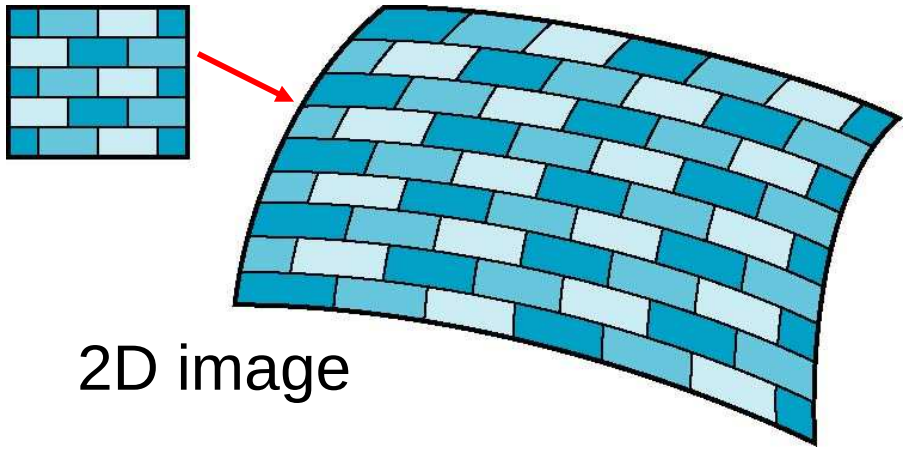
Where does mapping take place?

- Mapping techniques are implemented at the end of the rendering pipeline
  - Very efficient because few polygons pass down the geometric pipeline

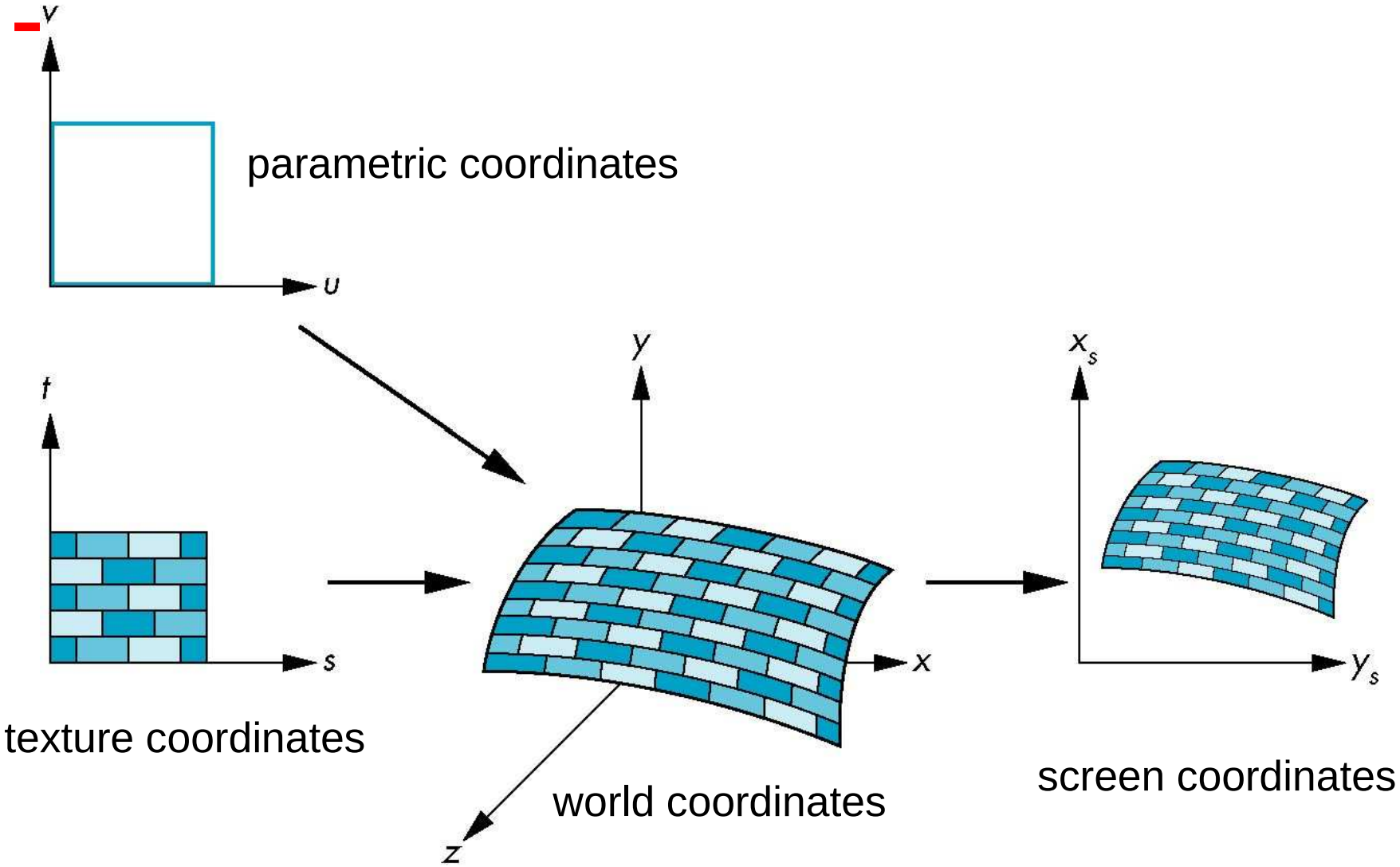


Is it simple?

- Although the idea is simple---map an image to a surface---there are 3 or 4 coordinate systems involved



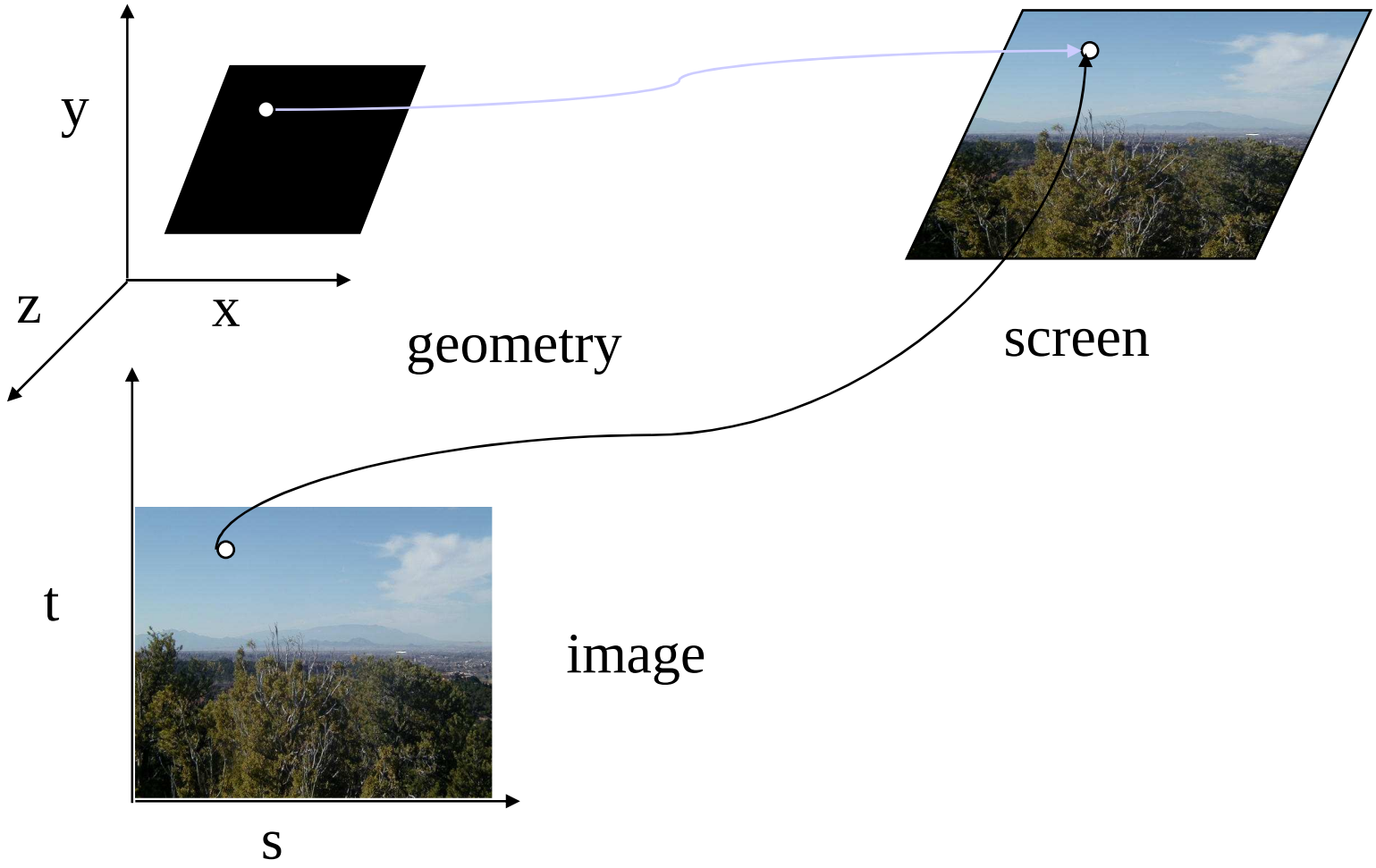
Texture Mapping



Basic Strategy

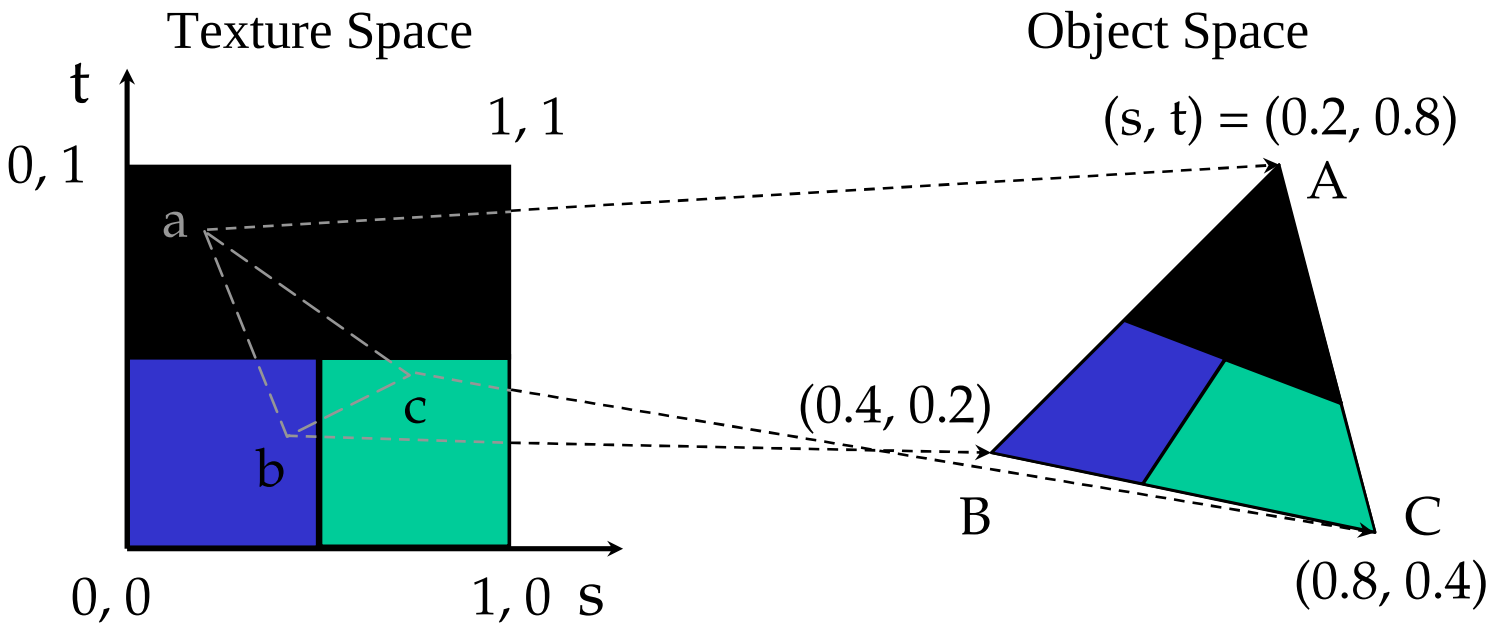
- Three steps to applying a texture
  1. specify the texture
     - read or generate image
     - assign to texture
     - enable texturing
  2. assign texture coordinates to vertices
     - Proper mapping function is left to application
  3. specify texture parameters
     - wrapping, filtering

Texture Mapping



Mapping a Texture

- Based on parametric texture coordinates specified at each vertex



Homogeneous Coordinates

A Single Representation

If we define $0 \bullet P = 0$ and $1 \bullet P = P$ then we can write

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [\alpha_1 \; \alpha_2 \; \alpha_3 \; 0 \,] \, [v_1 \; v_2 \; v_3 \; P_0]^T$$

$$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3 = [\beta_1 \; \beta_2 \; \beta_3 \; 1 \,] \, [v_1 \; v_2 \; v_3 \; P_0]^T$$

Thus we obtain the four-dimensional *homogeneous coordinate* representation

$$\mathbf{v} = [\alpha_1 \; \alpha_2 \; \alpha_3 \; 0 \,]^T$$

$$\mathbf{p} = [\beta_1 \; \beta_2 \; \beta_3 \; 1 \,]^T$$

Homogeneous Coordinates

The general form of four dimensional homogeneous coordinates is

$\mathbf{p}=[x\ y\ z\ w]^T$

We return to a three dimensional point (for $w\neq 0$) by

$x \leftarrow x/w$

$y \leftarrow y/w$

$z \leftarrow z/w$

If $w=0$, the representation is that of a vector

Note that homogeneous coordinates replaces points in three dimensions by lines through the origin in four dimensions

Homogeneous Coordinates and Computer Graphics

- Homogeneous coordinates are key to all computer graphics systems
  - All standard transformations (rotation, translation, scaling) can be implemented by matrix multiplications with 4 x 4 matrices
  - Hardware pipeline works with 4 dimensional representations
  - For orthographic viewing, we can maintain $w=0$ for vectors and $w=1$ for points
  - For perspective we need a *perspective division*

Change of Coordinate Systems

- Consider two representations of a the same vector with respect to two different bases. The representations are

$$\mathbf{a} = [\alpha_1 \ \alpha_2 \ \alpha_3]$$
$$\mathbf{b} = [\beta_1 \ \beta_2 \ \beta_3]$$

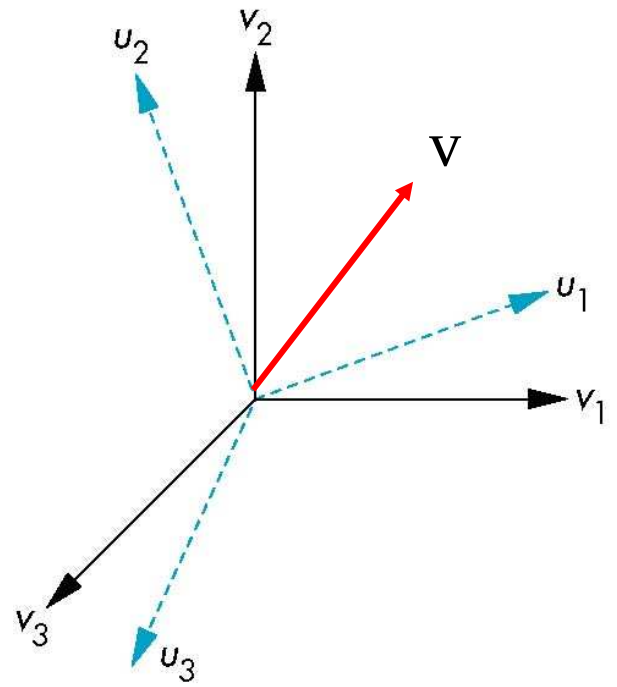
where

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [\alpha_1 \ \alpha_2 \ \alpha_3]\,[v_1 \ v_2 \ v_3]^T$$
$$= \beta_1 u_1 + \beta_2 u_2 + \beta_3 u_3 = [\beta_1 \ \beta_2 \ \beta_3]\,[u_1 \ u_2 \ u_3]^T$$

Representing second basis in terms of first

Each of the basis vectors, u1,u2, u3, are vectors that can be represented in terms of the first basis

$$u_1 = \gamma_{11}v_1+\gamma_{12}v_2+\gamma_{13}v_3$$
$$u_2 = \gamma_{21}v_1+\gamma_{22}v_2+\gamma_{23}v_3$$
$$u_3 = \gamma_{31}v_1+\gamma_{32}v_2+\gamma_{33}v_3$$



Matrix Form

The coefficients define a 3 x 3 matrix

$$\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix}$$

and the basis can be related by

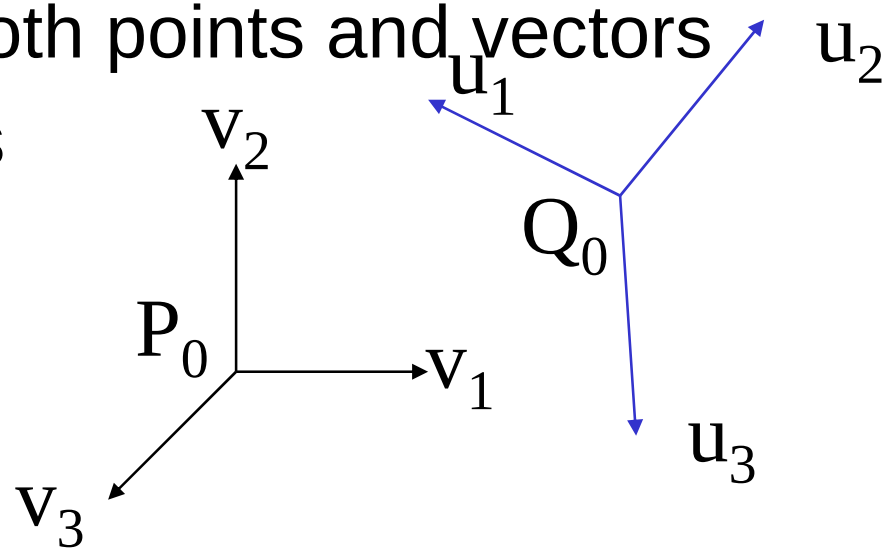
$$\mathbf{a}=\mathbf{M}^{\mathrm{T}}\mathbf{b}$$

see text for numerical examples

Change of Frames

- We can apply a similar process in homogeneous coordinates to the representations of both points and vectors
- Consider two frames

(P_0, v_1, v_2, v_3)
(Q_0, u_1, u_2, u_3)



- Any point or vector can be represented in each

Representing One Frame in Terms of the Other

Extending what we did with change of bases

$$u_1 = \gamma_{11}v_1+\gamma_{12}v_2+\gamma_{13}v_3$$
$$u_2 = \gamma_{21}v_1+\gamma_{22}v_2+\gamma_{23}v_3$$
$$u_3 = \gamma_{31}v_1+\gamma_{32}v_2+\gamma_{33}v_3$$
$$Q_0 = \gamma_{41}v_1+\gamma_{42}v_2+\gamma_{43}v_3 +P_0$$

defining a 4 x 4 matrix

$$\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix}$$

Working with Representations

Within the two frames any point or vector has a representation of the same form

$\mathbf{a}=[\alpha_1\ \alpha_2\ \ \alpha_3\ \alpha_4\ ]$ in the first frame
$\mathbf{b}=[\beta_1\ \beta_2\ \ \beta_3\ \beta_4\ ]$ in the second frame

where $\alpha_4=\beta_4=1$ for points and $\alpha_4=\beta_4=0$ for vectors and

$$\mathbf{a}=\mathbf{M}^T\mathbf{b}$$

The matrix $\mathbf{M}$ is 4 x 4 and specifies an affine transformation in homogeneous coordinates

Affine Transformations

- Every linear transformation is equivalent to a change in frames
- Every affine transformation preserves lines
- However, an affine transformation has only 12 *degrees of freedom* because 4 of the elements in the matrix are fixed and are a subset of all possible 4 x 4 linear transformations

Notation

We will be working with both coordinate-free representations of transformations and representations within a particular frame

P,Q, R: points in an affine space

u, v, w: vectors in an affine space

α, β, γ: scalars

p, **q**, **r**: representations of points

-array of 4 scalars in homogeneous coordinates

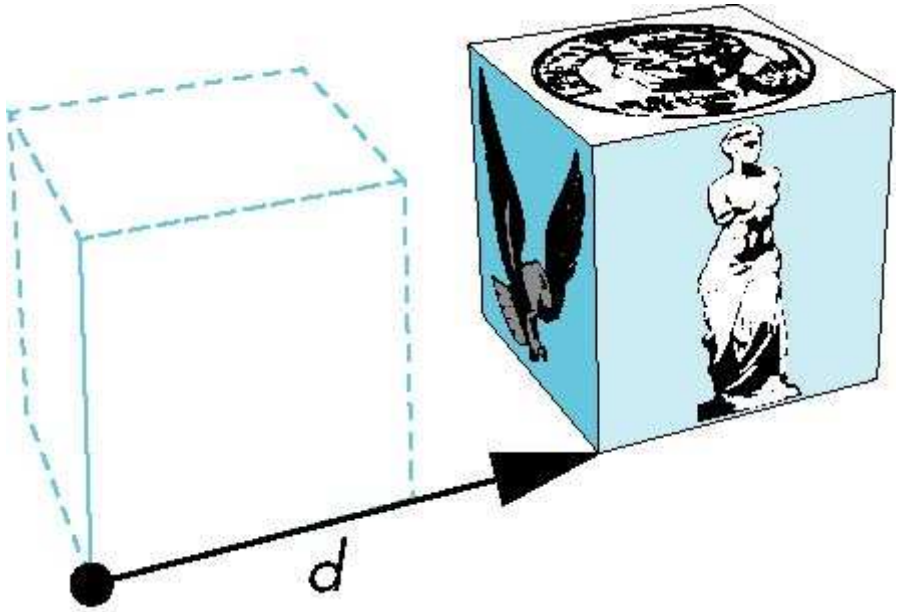
u, **v**, **w**: representations of points

-array of 4 scalars in homogeneous coordinates

Object Translation

Every point in object is displaced by same vector

object



Object translation

Translation Using Representations

Using the homogeneous coordinate representation in some frame

$\mathbf{p}=[\ x\ y\ z\ 1]^T$

$\mathbf{p}'=[x'\ y'\ z'\ 1]^T$

$\mathbf{d}=[dx\ dy\ dz\ 0]^T$

Hence $\mathbf{p}' = \mathbf{p} + \mathbf{d}$ or

$x'=x+dx$

$y'=y+dy$

$z'=z+dz$

note that this expression is in four dimensions and expresses that point = vector + point

Translation Matrix

We can also express translation using a 4 x 4 matrix $\mathbf{T}$ in homogeneous coordinates

$\mathbf{p}'=\mathbf{Tp}$ where

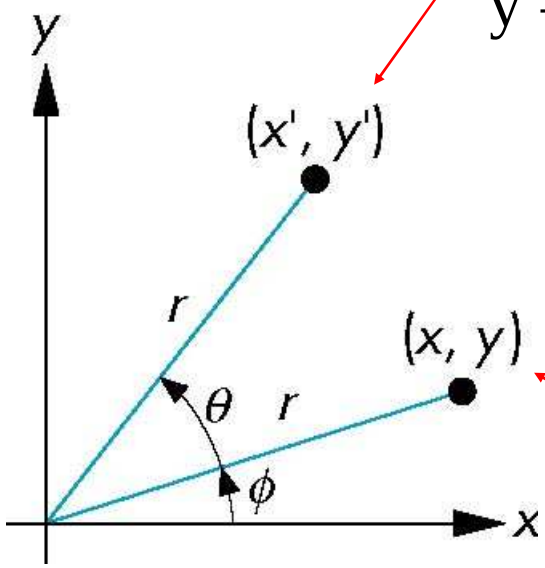
$$\mathbf{T} = \mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This form is better for implementation because all affine transformations can be expressed this way and multiple transformations can be concatenated together

Rotation (2D)

- Consider rotation about the origin by θ degrees
  - radius stays the same, angle increases by θ

$$x = r \cos(\phi + \theta)$$
$$y = r \sin(\phi + \theta)$$



$$x' = x \cos\theta - y \sin\theta$$
$$y' = x \sin\theta + y \cos\theta$$

$$x = r \cos\phi$$
$$y = r \sin\phi$$

Rotation about the z-axis

- Rotation about z axis in three dimensions leaves all points with the same z
  - Equivalent to rotation in two dimensions in planes of constant z

$$x' = x \cos\theta - y \sin\theta$$
$$y' = x \sin\theta + y \cos\theta$$
$$z' = z$$

  - or in homogeneous coordinates

$$\mathbf{p}' = \mathbf{R}_z(\theta)\mathbf{p}$$

Rotation Matrix

$$\mathbf{R} = \mathbf{R}_Z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

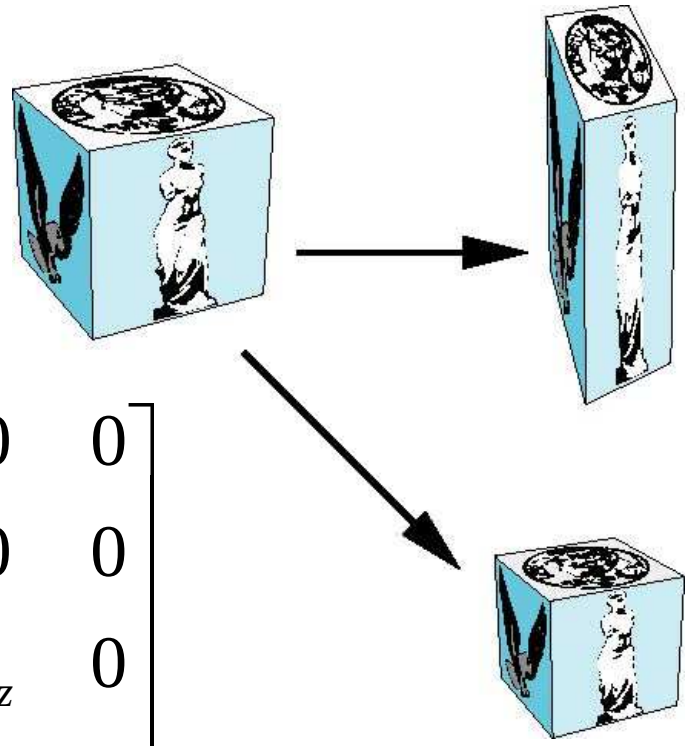
Scaling

Expand or contract along each axis (fixed point of origin)

$$x' = s_x x$$
$$y' = s_y y$$
$$z' = s_z z$$

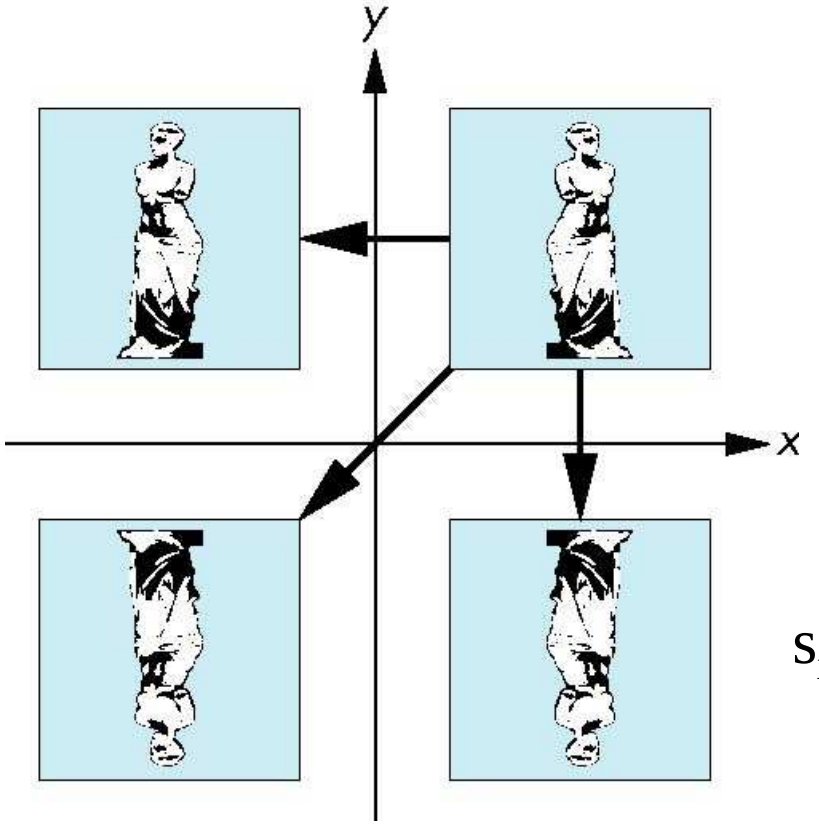
$$\mathbf{p}' = \mathbf{S}\mathbf{p}$$

$$\mathbf{S} = \mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Reflection

corresponds to negative scale factors



$s_x = -1\ s_y = 1$

original

$s_x = -1\ s_y = -1$

$s_x = 1\ s_y = -1$

Concatenation

- We can form arbitrary affine transformation matrices by multiplying together rotation, translation, and scaling matrices
- Because the same transformation is applied to many vertices, the cost of forming a matrix $\mathbf{M}=\mathbf{ABCD}$ is not significant compared to the cost of computing $\mathbf{Mp}$ for many vertices $\mathbf{p}$
- The difficult part is how to form a desired transformation from the specifications in the application

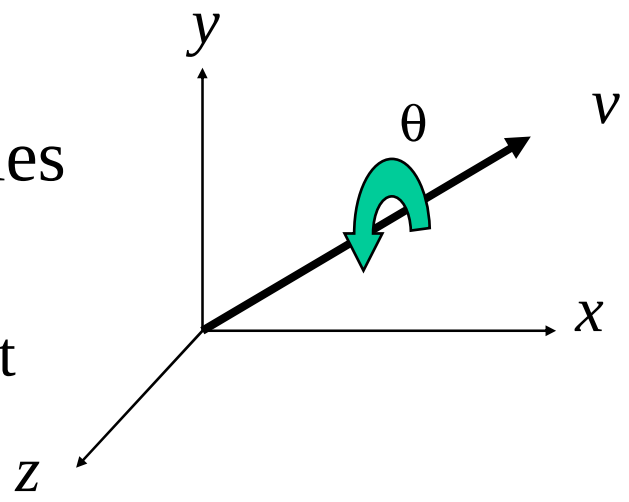
General Rotation About the Origin

A rotation by θ about an arbitrary axis can be decomposed into the concatenation of rotations about the *x*, *y*, and *z* axes

$$\mathbf{R}(\theta) = \mathbf{R}_z(\theta_z)\,\mathbf{R}_y(\theta_y)\,\mathbf{R}_x(\theta_x)$$

$\theta_x\ \theta_y\ \theta_z$ are called the Euler angles

Note that rotations do not commute
We can use rotations in another order but with different angles



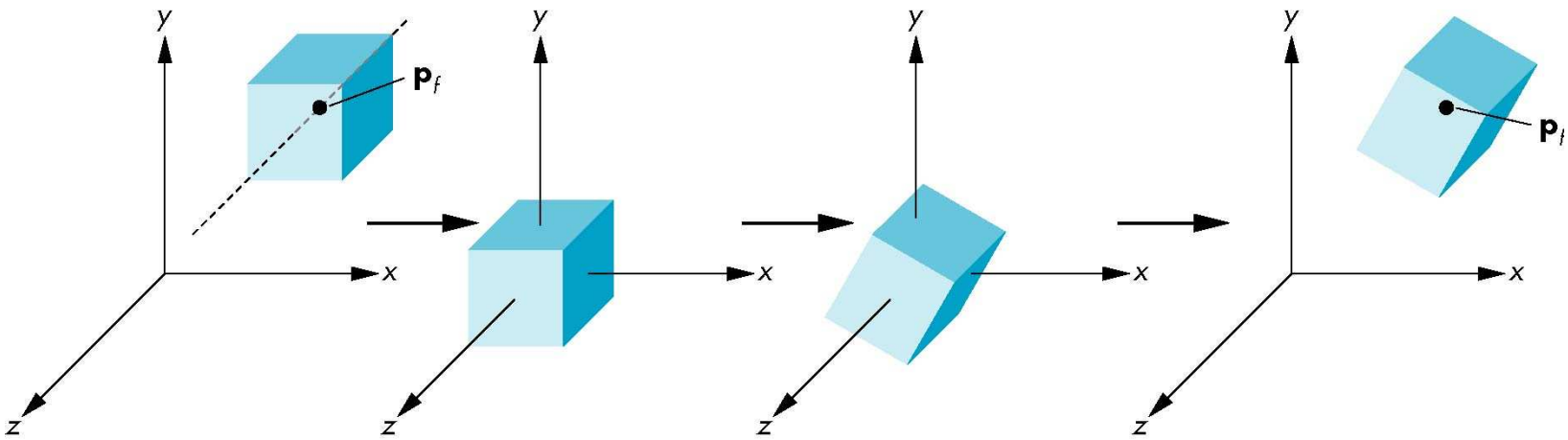
Rotation About a Fixed Point other than the Origin

Move fixed point to origin

Rotate

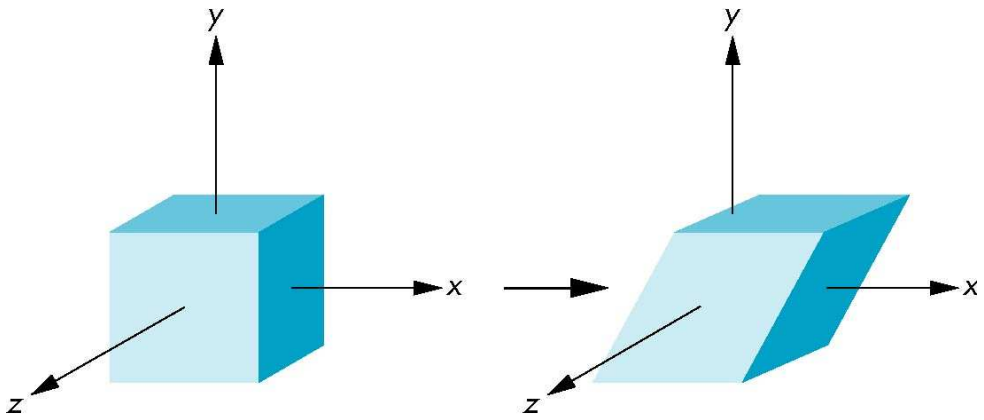
Move fixed point back

$\mathbf{M} = \mathbf{T}(-p_f)\,\mathbf{R}(\theta)\,\mathbf{T}(p_f)$



Shear

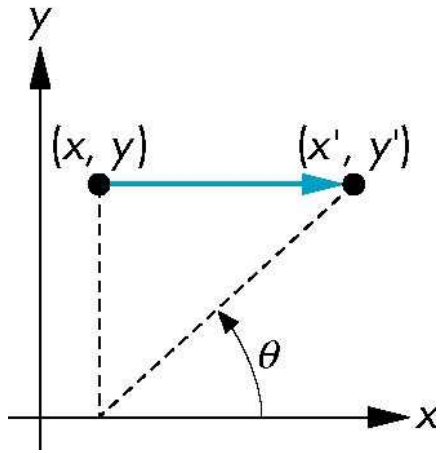
- Helpful to add one more basic transformation
- Equivalent to pulling faces in opposite directions



Shear Matrix

Consider simple shear along x axis

$$x' = x + y \cot \theta$$
$$y' = y$$
$$z' = z$$



$$\mathbf{H}(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Quaternions

- Extension of imaginary numbers from two to four dimensions
- Requires one real and three imaginary components $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$

$$q = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k}$$

- Quaternions can express rotations on sphere smoothly and efficiently. Process:
  - Model-view matrix $\rightarrow$ quaternion
  - Carry out operations with quaternions
  - Quaternion $\rightarrow$ Model-view matrix