Introduction to OpenGL

2009 Autumn

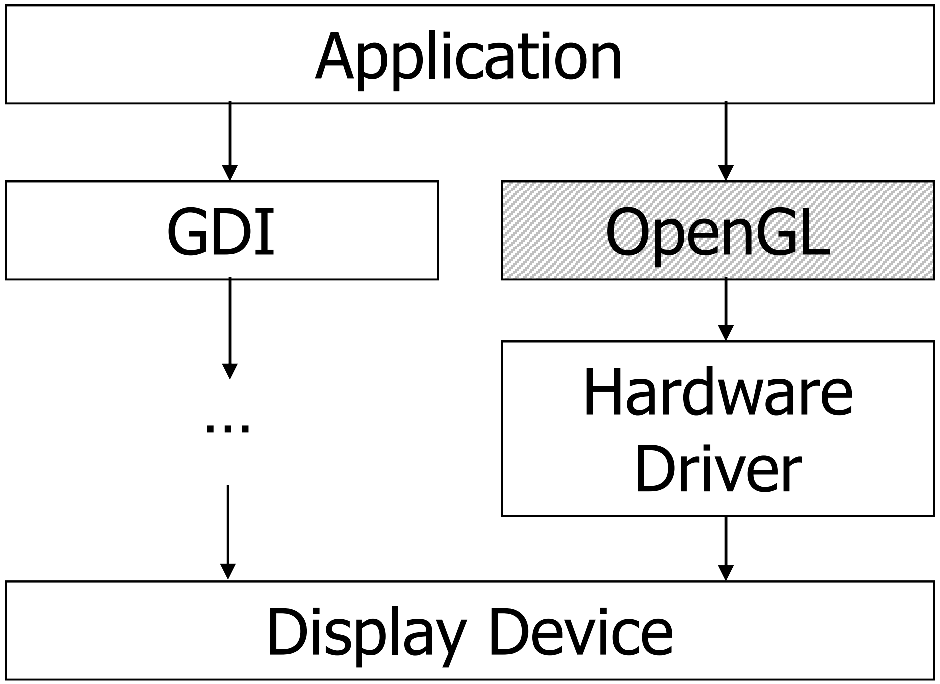
Animação e Visualização Tridimesional

2009/2010

Graphics API

- A software interface for graphics hardware.
- Provide the low-level functions to access graphics hardware directly.
- OpenGL / Direct3D

API Hierarchy



What is OpenGL 1/2

- Industry standard.
- Hardware independent.
- OS independent.

What is OpenGL 2/2

- No commands for performing windowing tasks or obtaining user input are included.
- No high-level commands for describing models of 3D objects are provided.

OpenGL Evolution

- Originally controlled by an Architectural Review Board (ARB)
  - Members included SGI, Microsoft, Nvidia, HP, 3DLabs, IBM,…….
  - Relatively stable (present version 2.1)
    - Evolution reflects new hardware capabilities
      - **3D texture mapping and texture objects**
      - **Vertex programs**
  - Allows for platform specific features through extensions
  - ARB replaced by Kronos

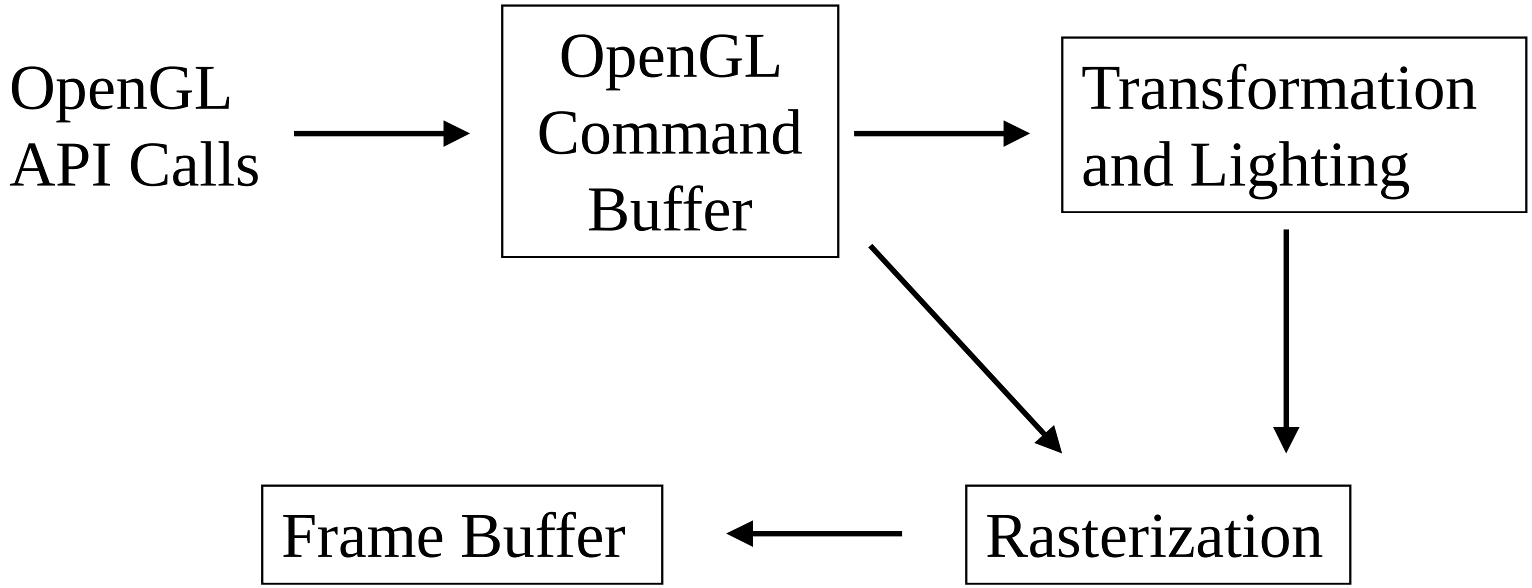
What OpenGL provides

- Draw with points, lines, and polygons.
- Attributes
- Matrix(View) Transformation
- Hidden Surface Removal (Z-Buffer)
- Light effects
- Gouraud Shading
- Texture mapping
- Pixels operation

The Buffers

- A buffer is a memory area in the graphics hardware for some special purposes.
- An OpenGL system can manipulate the four buffers:
  - Color buffers
  - Depth buffer (Z-Buffer)
  - Stencil buffer
  - Accumulation buffer

OpenGL Rendering Pipeline



OpenGL Libraries

- OpenGL Library - GL
  - The basic library to access the graphics hardware.
  - OpenGL32 on Windows
  - GL on most unix/linux systems (libGL.a)
- GLU
  - Provide some useful utilities based on the OpenGL library.
  - Provides functionality in OpenGL core but avoids having to rewrite code
- GLX / WGL / AGL
  - OS dependent libraries to bind the OpenGL library with specific window system.
  - GLX for X-window, WGL for win32, AGL for Apple.

OpenGL Utility Toolkit (GLUT) 1/3

- A window system-independent toolkit to hide the complexities of differing window system APIs.
- Use the prefix of **glut**. (ex: glutDisplayFunc())
- Provide following operations:
  - Initializing and creating window
  - Handling window and input events
  - Drawing basic three-dimensional objects
  - Running the program
  - Event-driven
  - No slide bars

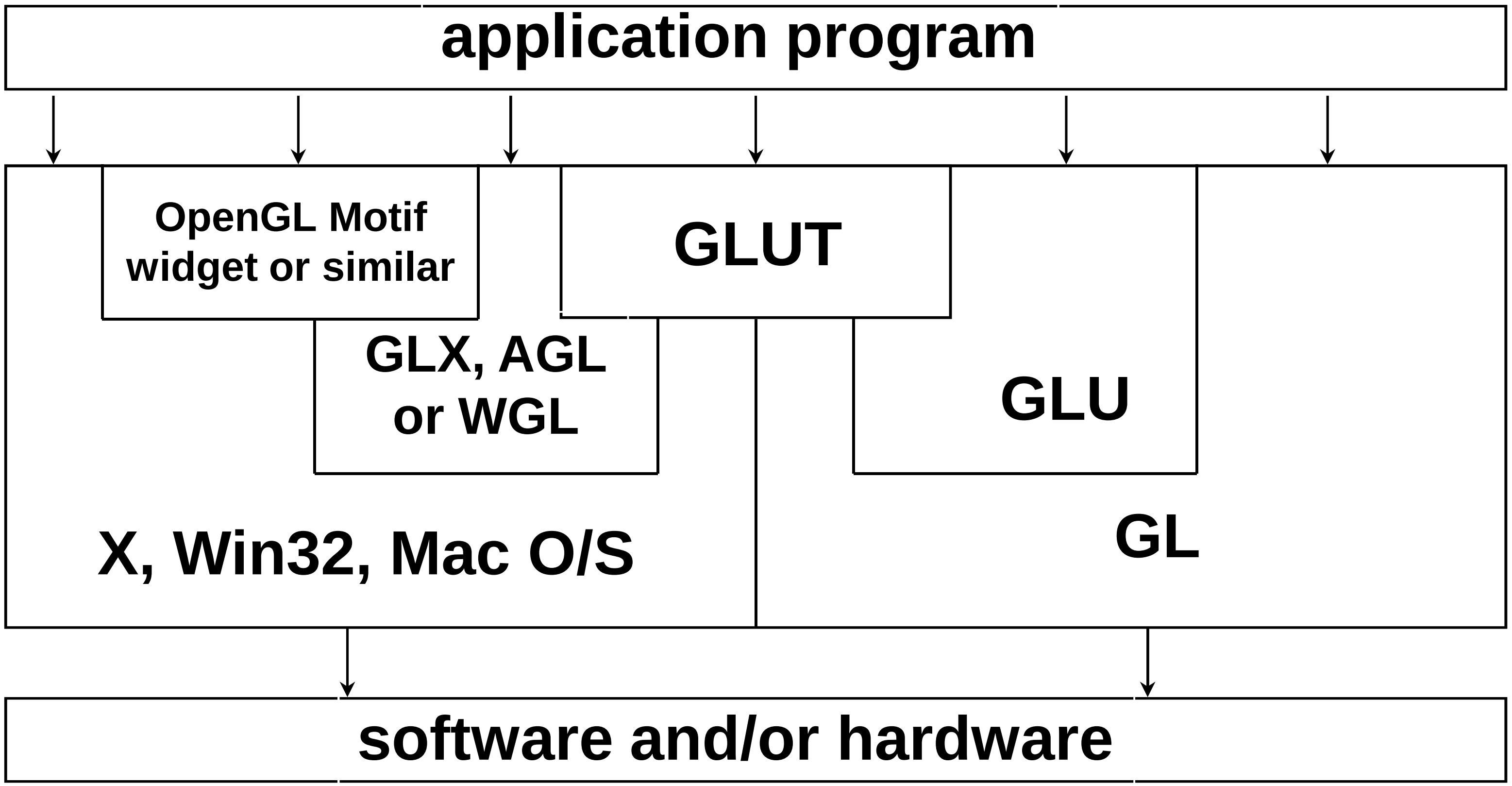
OpenGL Utility Toolkit (GLUT) 2/3

- Where can I get GLUT for Win32 and for Unix?
  - www.opengl.org/resources/libraries/**glut**/
- For Mac OS X:
  - developer.apple.com/mac/library/samplecode/glut/

OpenGL Utility Toolkit (GLUT)
3/3

- On Microsoft Visual C++ 6:
  - Put glut.h into <MSVC>/include/GL/
  - Put glut.lib into <MSVC>/lib/
  - Put glut32.dll into <window>/System32/
- On Microsoft Visual C++ .NET:
  - Put glut.h into <MSVC>/platformSDK/include/GL/
  - Put glut.lib into <MSVC>/platformSDK/lib/
  - Put glut32.dll into <window>/System32/

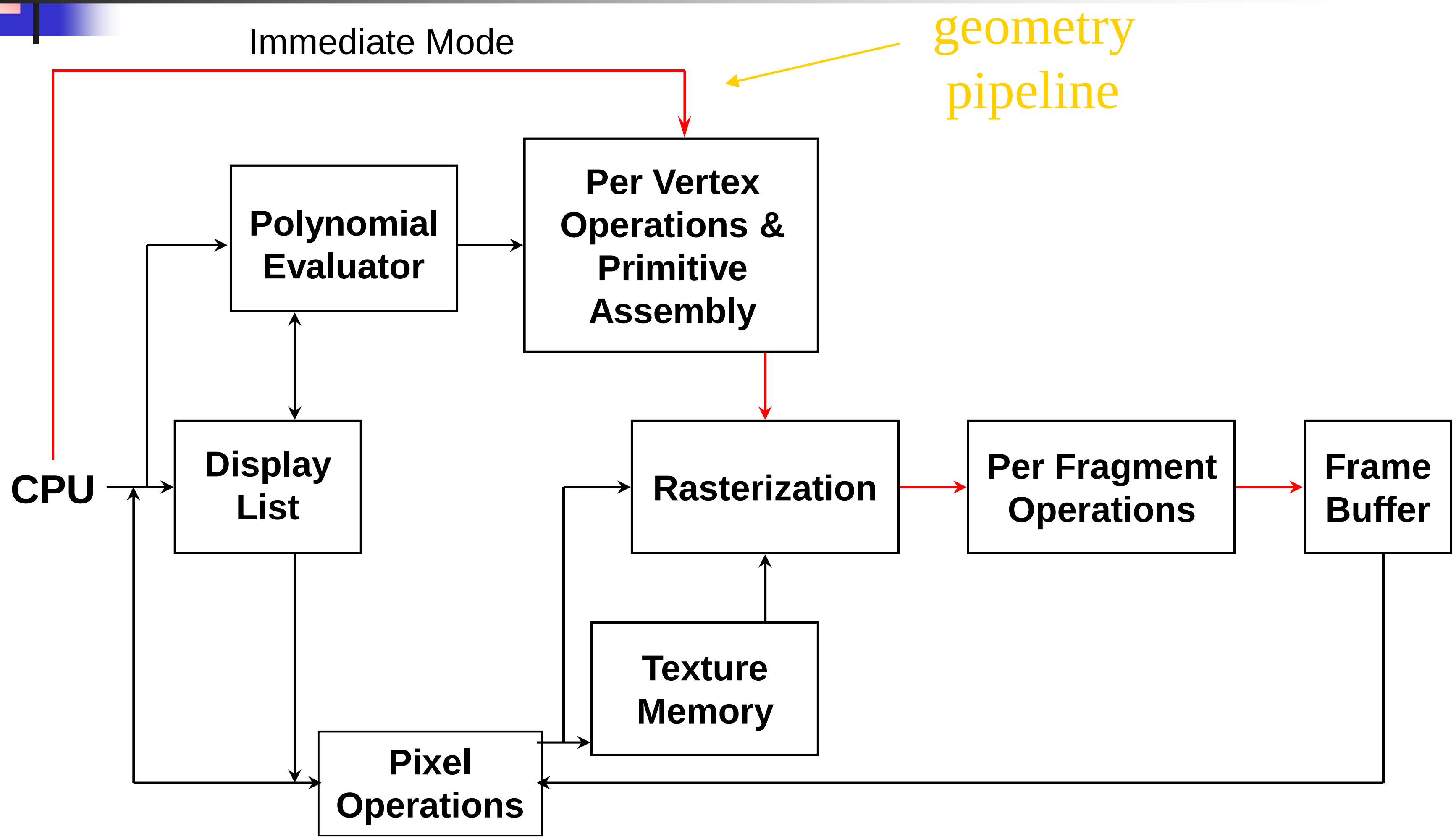
Software Organization



How to Compile 1/4

- On Microsoft Visual C++ 6:
  - Create a new Project with ***Win32 Console Application***
  - Open **Project Settings** dialog and add ***opengl32.lib glu32.lib glut32.lib*** into Link/Objects/library modules.
  - Writing your OpenGL code.
  - Compile it.

OpenGL Architecture



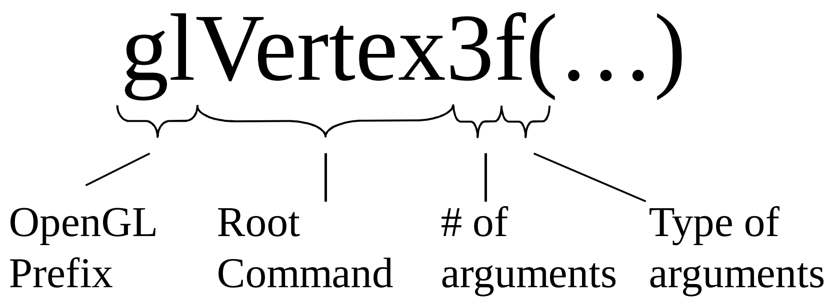
Fog Demo

- Nate Robins Tutors OpenGL examples
  - http://www.xmission.com/~nate/tutors.html
- OpenGL sintax
- Several models
- 2D (text) and 3D drawing
- Image effects
- Graphics Windows hierarchy
- Menu capabilities
- Picking Operation

Drawing Geometric Objects

OpenGL Command Syntax -1

- OpenGL commands use the prefix **gl** and initial capital letters for each word.
- OpenGL defined constants begin with **GL_**, use all capital letters and underscores to separate words.



OpenGL Command Syntax -2

function name

dimensions

`glVertex3f(x,y,z)`

belongs to GL library

`x,y,z` are `floats`

`glVertex3fv(p)`

`p` is a pointer to an array

Lack of Object Orientation

- OpenGL is not object oriented so that there are multiple functions for a given logical function
  - **glVertex3f**
  - **glVertex2i**
  - **glVertex3fv**
- Underlying storage mode is the same
- Easy to create overloaded functions in C++ but issue is efficiency

OpenGL Data Type

OpenGL Type	Internal representation	C-Language Type	Suffix
GLbyte	8-bit integer	signed char	b
GLshort	16-bit integer	short	s
GLint, GLsizei	32-bit integer	int or long	i
GLfloat	32-bit floating	float	f
GLflampf	pointer		
GLfouble	64-bit floating	double	d
GLclampd	pointer		
Glubyte	8-bit unsigned integer	unsigned char	ub
GLuboolean	8-bit unsigned integer	unsigned char	ub
GLushort	16-bit unsigned integer	unsigned short	us
GLuint, GLenum	32-bit unsigned integer	unsigned long	ui
GLbitfield	32-bit unsigned integer		

State Management$_{1/2}$

- OpenGL is a state machine.
  - You put it into various states (or modes) that then remain in effect until you change them.
  - Each state variable or mode has a default value, and at any point you can query the system for each variable's current value.

State Management$_{2/2}$

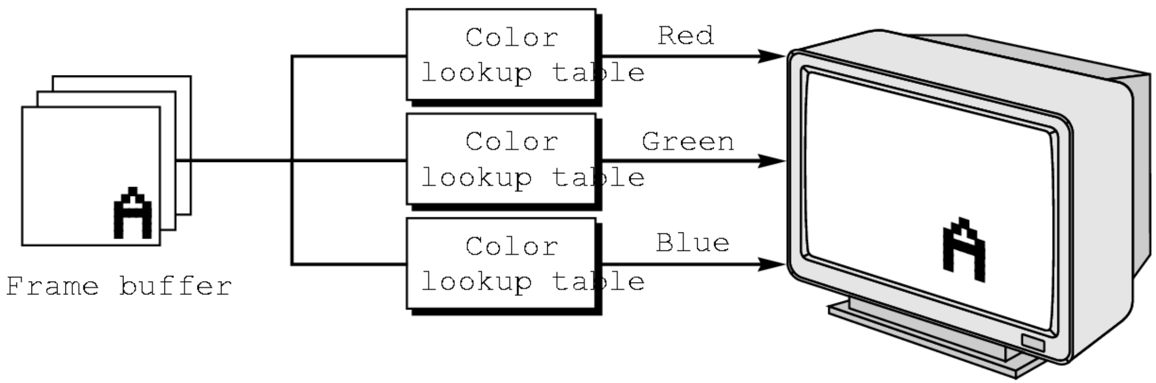
- **glEnable(GLenum); glDisable(GLenum);**
  - enable and disable some state.
- **glIsEnabled(GLenum);**
  - Query if the specific state is enabled
- **glGetBooleanv(); glGetIntegerv(); glGetFloatv(); glGetDoublev(); glGetPointerv();**
  - Query the specific state value.
- See ***OpenGL Programming Guide : Appendix B*** for all the state variables.

Color Representation 1/2

- RGBA
  - 4 channels : Red, Green, Blue, and Alpha.
  - Each channel has intensity from 0.0 ~ 1.0
    - Values outside this interval will be clamp to 0.0 or 1.0.
  - Alpha is used in blending and transparency
    - Ex. glColor4f(0.0, 1.0, 0.0, 1.0); // Green
      glColor4f(1.0, 1.0, 1.0, 1.0); // White

Color Representation 2/2

- Color-Index
  - Small numbers of colors accessed by indices (8 bits) from a color map(lookup table).
    - Ex.   glIndex(…);
  - Less colors
  - The OpenGL has no command about creating the color map, it's window system's business.
    - glutSetColor();



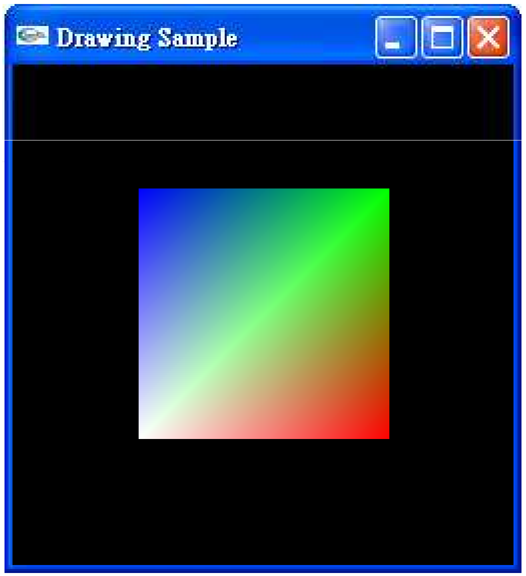
Drawing Sample 1/3

```
#include <GL/glut.h>
void GL_display() {
      glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
      glClear(GL_COLOR_BUFFER_BIT);
      glBegin(GL_POLYGON);
             glColor3f(1.0f, 1.0f, 1.0f);
            glVertex3f (-1.0, -1.0, 0.0);
             glColor3f(1.0f, 0.0f, 0.0f);
            glVertex3f (1.0, -1.0, 0.0);
             glColor3f(0.0f, 1.0f, 0.0f);
            glVertex3f (1.0, 1.0, 0.0);
             glColor3f(0.0f, 0.0f, 1.0f);
            glVertex3f (-1.0, 1.0, 0.0);
      glEnd();
      glFlush();
}
```

Drawing Sample 2/3

```
void GL_reshape(GLsizei w, GLsizei h) {
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if(w <= h)
        glOrtho(-2.0f, 2.0f, -2.0f * h/w, 2.0f * h/w, -2.0f, 2.0f);
    else
        glOrtho(-2.0f * w/h, 2.0f * w/h, -2.0f, 2.0f, -2.0f, 2.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
void main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (250, 250);
    glutInitWindowPosition (100, 100);
    glutCreateWindow("Drawing Sample");
    glutDisplayFunc(GL_display);
    glutReshapeFunc(GL_reshape);
    glutMainLoop();
}
```

Drawing Sample$_{3/3}$



OpenGL #defines

- Most constants are defined in the include files **`gl.h`**, **`glu.h`** and **`glut.h`**
  - Note **`#include <GL/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`**,….

Program Detail (GLUT) 1/5

- Initializing and Creating a window
  - **void glutInit(int, char**);**
    - Initialize the GLUT library.
    - Should be called before any other GLUT routine.
  - **void glutInitDisplayMode(unsigned int);**
    - Specify a display mode for windows created.
    - GLUT_RGBA / GLUT_INDEX
    - GLUT_SINGLE / GLUT_DOUBLE
    - GLUT_DEPTH, GLUT_STENCIL, GLUT_ACCUM

Program Detail (GLUT) 2/5

- **glutInitWindowPosition(int, int);**
  - **From top-left corner of display**
- **glutInitWindowSize(int, int);**
  - Initial the window position and size when created.
- **glutCreateWindow(char*);**
  - Open a window with previous settings.

Program Detail (GLUT)3/5

- Handling Window and Input Events
  - These functions are registered by user and called by GLUT simultaneously.
  - **glutDisplayFunc(void (*func)(void));**
    - Called whenever the contents of the window need to be redrawn.
    - Put whatever you wish to draw on screen here.
    - Use **glutPostRedisplay()** to manually ask GLUT to recall this display function.

Program Detail (GLUT) 4/5

- **glutReshapeFunc(void (*func)(int, int));**
  - Called whenever the window is resized or moved.
  - You should always call **glViewport()** here to resize your viewport.
- Other call back functions:
  - glutKeyboardFunc();
  - glutMouseFunc();
  - glutIdleFunc();
  - …
  - See ***OpenGL Programming Guide : Appendix D*** for more detail

Program Detail (GLUT)5/5

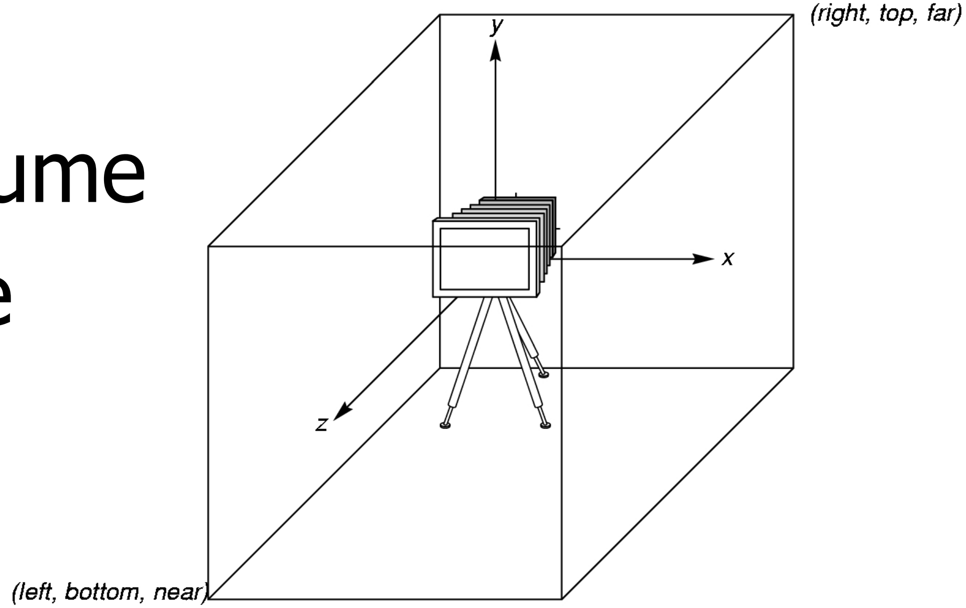
- Running the Program
  - **glutMainLoop();**
    - Enter the GLUT processing loop and never return.

glutReshapeFunc()

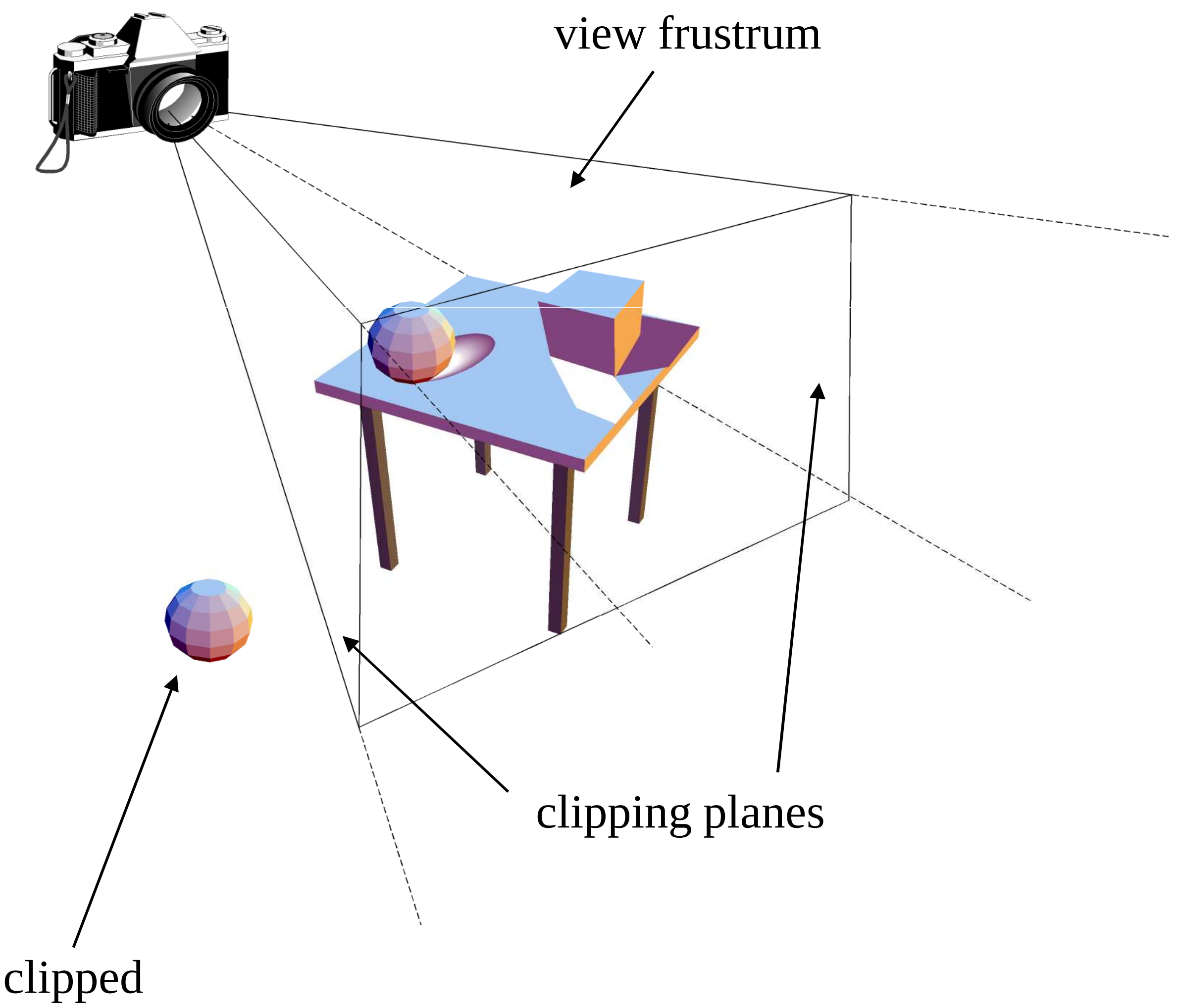
```
void GL_reshape(GLsizei w, GLsizei h) {
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if(w <= h)
        glOrtho(-2.0f, 2.0f, -2.0f * h/w, 2.0f * h/w, -2.0f,
    2.0f);
    else
        glOrtho(-2.0f * w/h, 2.0f * w/h, -2.0f, 2.0f, -2.0f,
    2.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

OpenGL Camera

- OpenGL places a camera at the origin in object space pointing in the negative z direction
- The default viewing volume is a box centered at the origin with a side of length 2

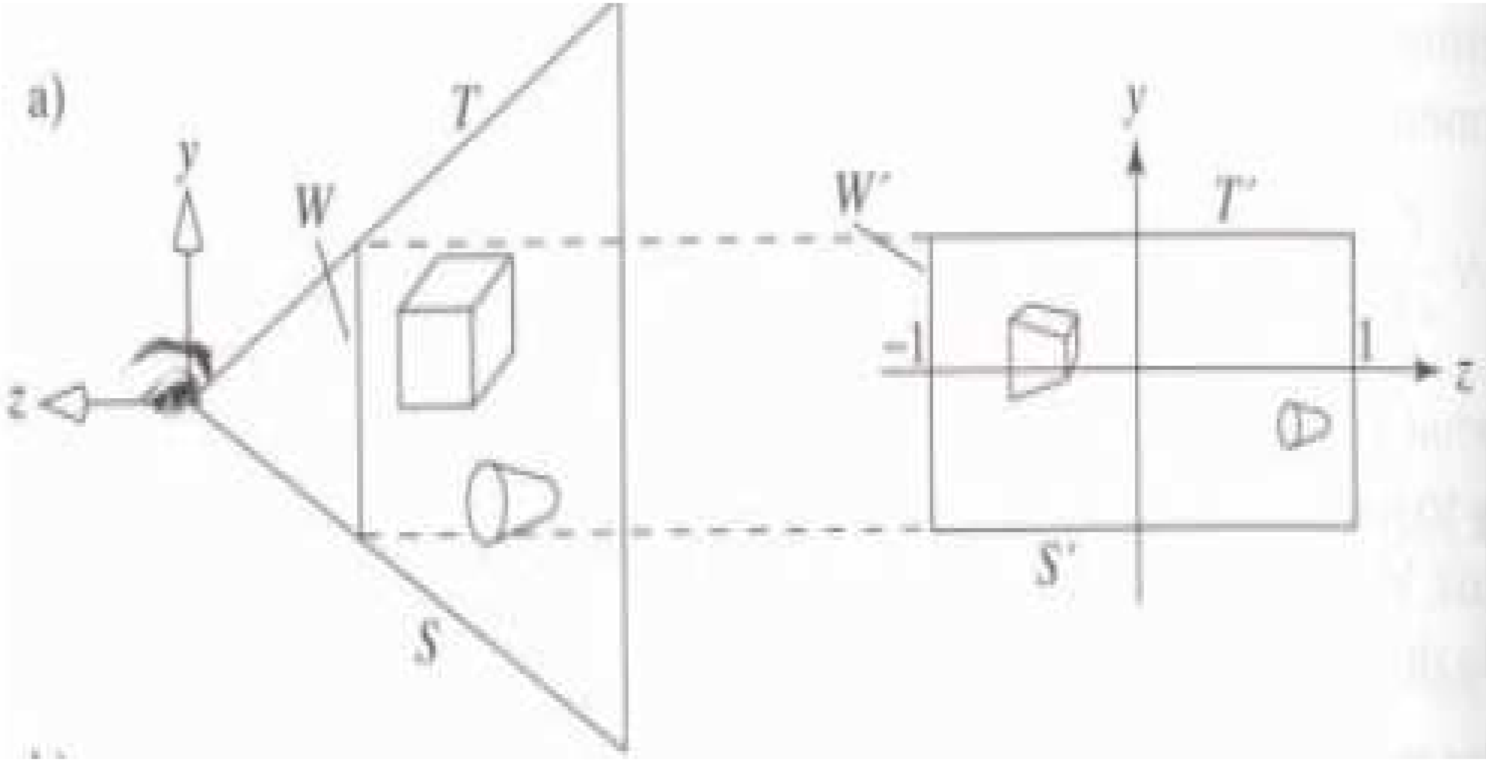


Viewing System



Viewing and Projection transforms

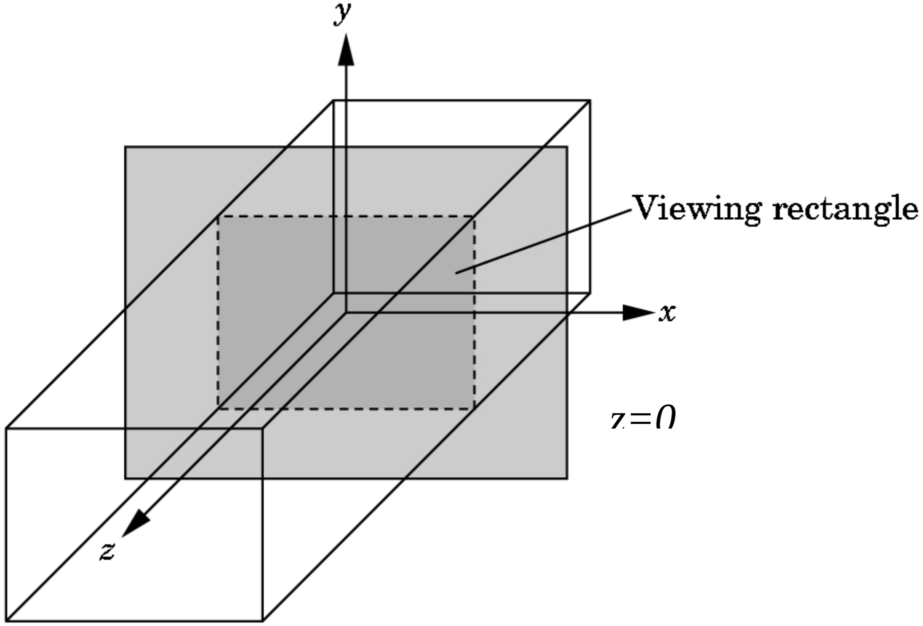
```
glMatrixMode
(GL_MODELVIEW)
gluLookAt()
...
```

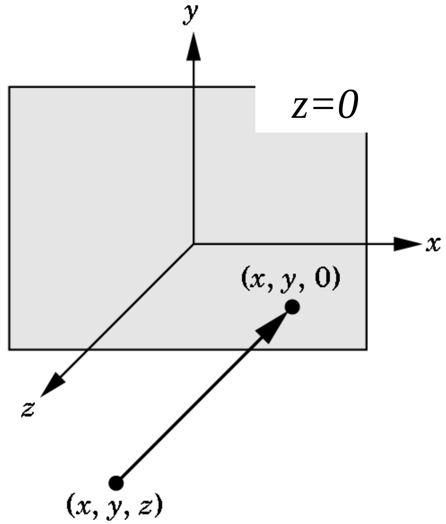



```
glMatrixMode(GL_PROJECTION)
...
```

Orthographic Viewing

In the default orthographic view, points are projected forward along the *z* axis onto the plane *z=0*





Projection transform

- In OpenGL, projection is carried out by a projection matrix (transformation)
- There is only one set of transformation functions so we must set the matrix mode first

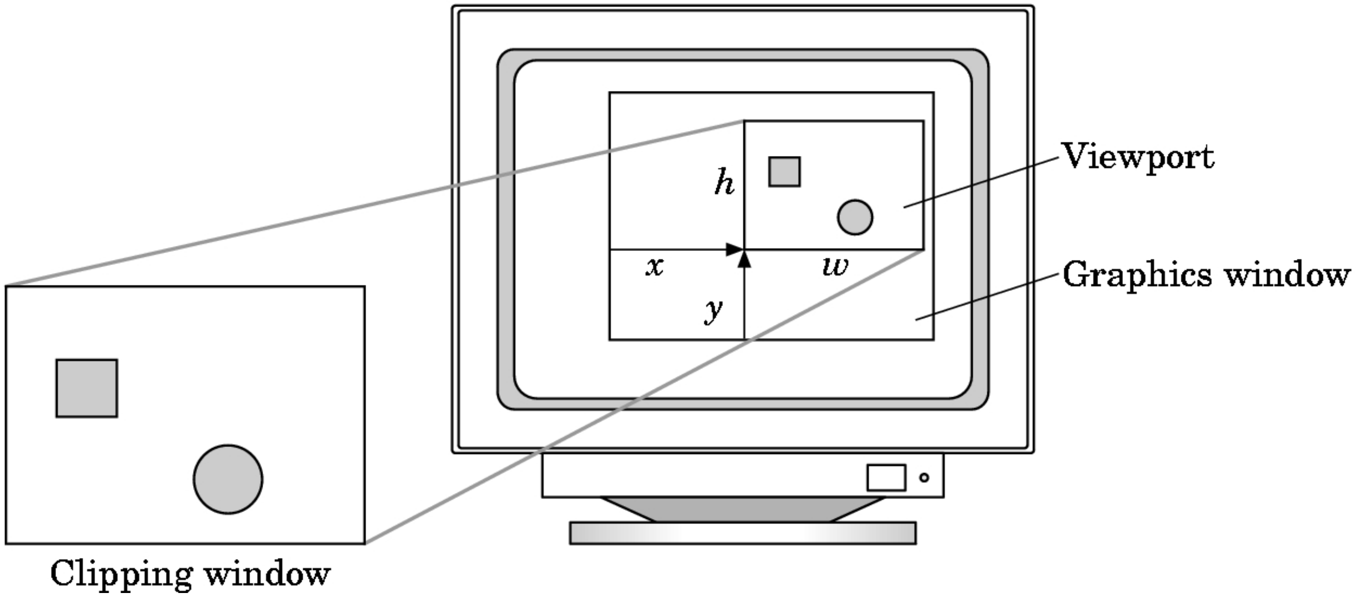
```
glMatrixMode (GL_PROJECTION)
```

- Transformation functions are incremental so we start with an identity matrix and alter it with a projection matrix that gives the view volume

```
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0,
1.0);
```

Viewport

- Do not have use the entire window for the image: **`glViewport(x,y,w,h)`**
- Values in pixels (screen coordinates)



Two- and three-dimensional viewing

- In **`glOrtho(left, right, bottom, top, near, far)`** the near and far distances are measured **from** the camera
- Two-dimensional vertex commands place all vertices in the plane z=0
- If the application is in two dimensions, we can use the function

  `gluOrtho2D(left, right,bottom,top)`
- In two dimensions, the view or clipping volume becomes a *clipping window*

A Drawing Survival Kit

- Clear the Buffers
- Describe Points, Lines, and Polygons
- Forcing Completion of Drawing

Clear the Buffers

- **glClearColor(...);**
- **glClearDepth(...);**
  - Set the current clearing values for use in clearing color buffers in RGBA mode (or depth buffer).
- **glClear(GLbitfield mask);**
  - Clear the specified buffers to their current clearing values.
  - GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, ...

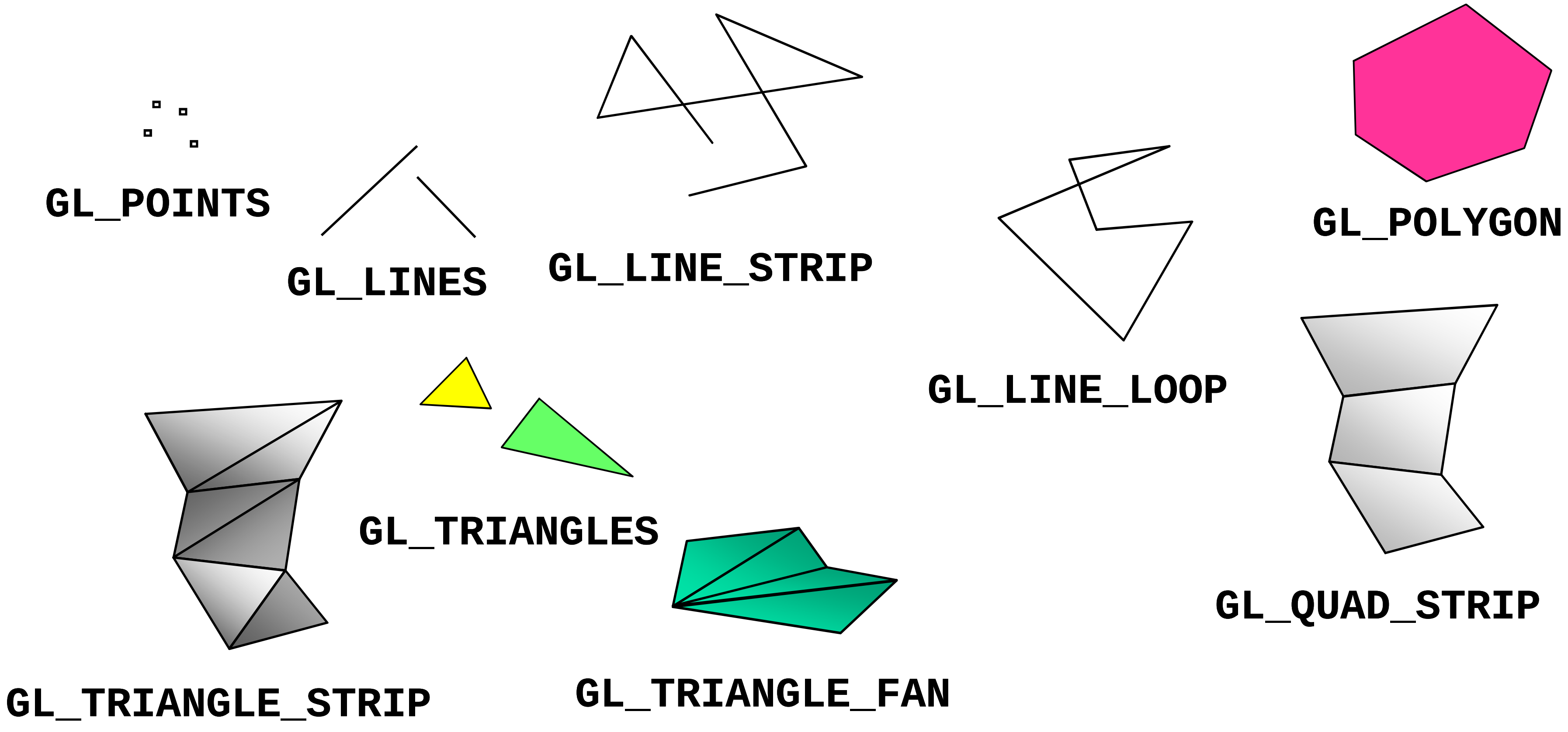
Points, Lines and Polygons 1/4

- Specifying a Color
  - **glColor {34}{sifd}[v](TYPE colors);**
- Describing Points, Lines, Polygons
  - **void glBegin(GLenum mode);**
    - Marks the beginning of a vertex-data list.
    - The mode can be any of the values in next page.
  - **void glEnd();**
    - Marks the end of a vertex-data list.

Points, Lines and Polygons 2/4

Value	Meaning
GL_POINTS	individual points
GL_LINES	pairs of vertices interpreted as individual line segments
GL_LINE_STRIP	serious of connected line segments
GL_LINE_LOOP	same as above, with a segment added between last and first vertices
GL_TRIANGLES	triples of vertices interpreted as triangles
GL_TRIANGLE_STRIP	linked strip of triangles
GL_TRIANGLE_FAN	linked fan of triangles
GL_QUADS	quadruples of vertices interpreted as four-sided polygons
GL_QUAD_STRIP	linked strip of quadrilaterals
GL_POLYGON	boundary of a simple, convex polygon

Points, Lines and Polygons 3/4

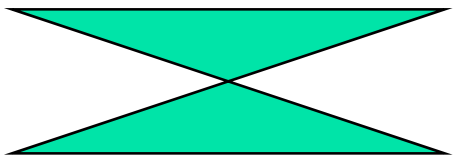


Points, Lines and Polygons 4/4

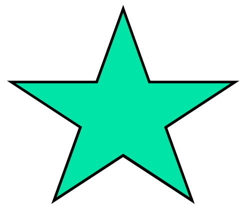
- valid calls between **glBegin()** and **glEnd()**
  - glVertex*(); glNormal*(); glColor*(); glIndex*(); glTexCoord*(); glMaterial*(); …
- Specifying Vertices
  - **glVertex{234}{sifd}[v](TYPE coords);**
    - Specifies a vertex for use in describing a geometric object.
    - Can only effective between a **glBegin()** and **glEnd()** pair.

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 can check if above true
  - OpenGL will produce output if these conditions are violated but it may not be what is desired
- Triangles satisfy all conditions



nonsimple polygon



nonconvex polygon

Attributes

- Attributes are part of the OpenGL state 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
    - Display vertices

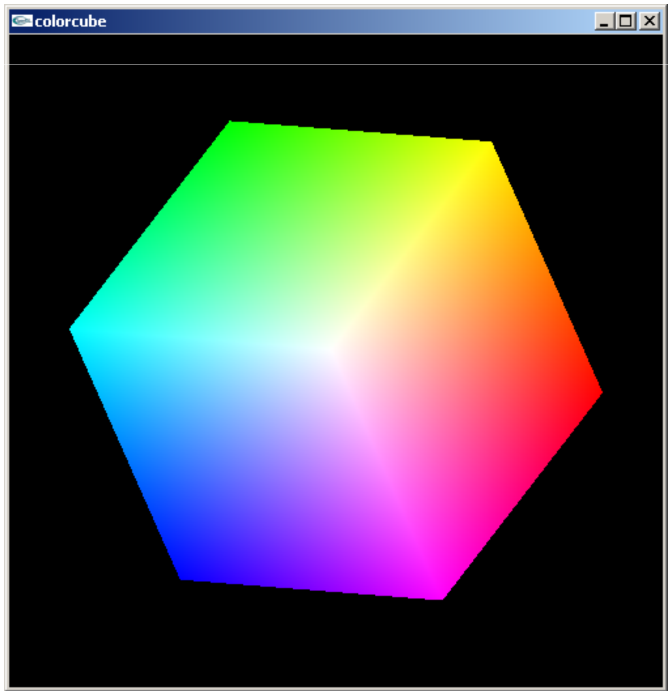
Smooth Color

- Default is *smooth* shading
  - OpenGL interpolates vertex colors across visible polygons
- Alternative is *flat shading*
  - Color of first vertex

  determines fill color
- **`glShadeModel`**

  `(GL_SMOOTH)`

  or **`GL_FLAT`**



GLUT Objects

- Drawing 3D objects using GLUT
  - GLUT provides the following objects:
    - Sphere, Cube, Torus, Icosahedron, Octahedron, Tetrahedron, Teapot, Dodecahedron, Cone, Teapot
    - Both wireframe and solid.
    - Ex:
      - **glutSolidSphere(1.0, 24, 24);**
      - **glutWireCube(1.0);**

Completion of Drawing

- **glFlush();**
  - Forces previously issued OpenGL commands to begin execution. (asynchronous)
- **glFinish();**
  - Forces all previous issued OpenGL commands to complete. (synchronous)
- **glutSwapBuffers();**
  - Swap front and back buffers. (double buffers)

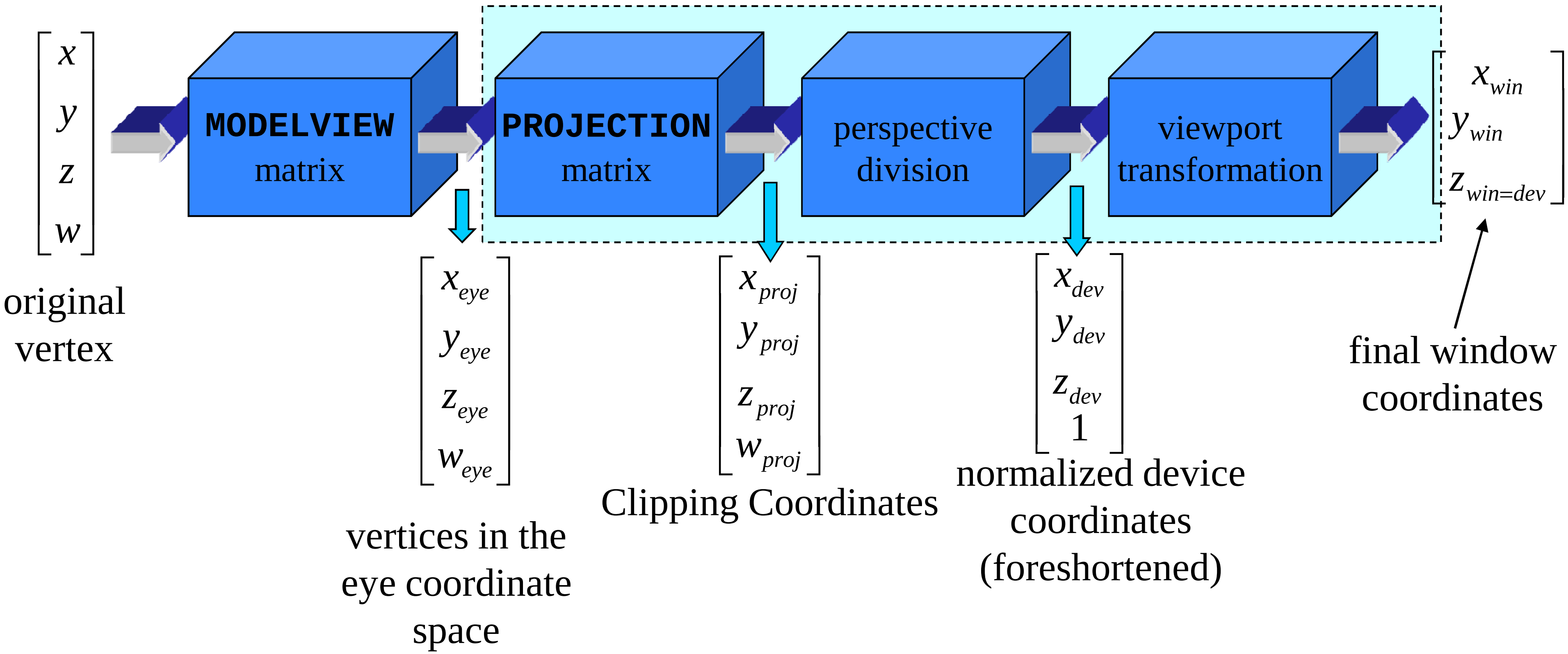
Polygon Details$_{1/2}$

- Polygon Details
  - **glPolygonMode(Glenum face, Glenum mode);**
    - Controls the drawing mode for a polygon's front and back faces.
    - face can be GL_FRONT_AND_BACK, GL_FRONT, GL_BACK
    - mode can be GL_POINT, GL_LINE, GL_FILL

Polygon Details 2/2

- **glFrontFace(Glenum mode);**
  - Controls how front-facing polygons are determined.
  - GL_CW for clockwise and GL_CCW(default) for counterclockwise
- **glCullFace(Glenum mode);**
  - Indicates which polygons should be discarded before converted to screen coordinate.
  - mode can be GL_FRONT_AND_BACK, GL_FRONT, GL_BACK

OpenGL Geometry Pipeline



Transformation -2

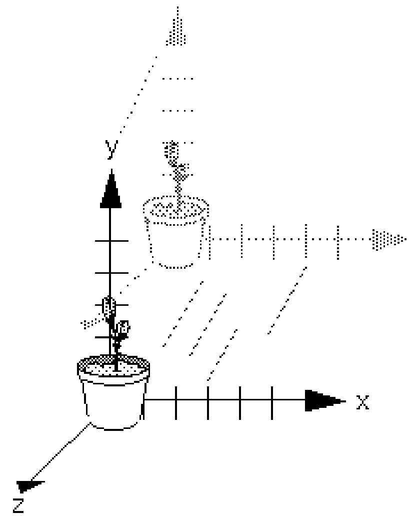
- There are three **matrix stacks** in OpenGL architecture
  - MODELVIEW, PROJECTION, TEXTURE
  - glMatrixMode(GLenum mode);
    - mode: GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE
  - **Current matrix mode (CTM)** is also a OpenGL state variable.

Transformation -3

- Matrix Manipulation
  - glLoadIdentity();
    - Set current matrix to the 4x4 identity matrix
  - glLoadMatrix{f,d}(const TYPE* m);
  - glMultMatrix{f,d}(const TYPE* m);
  - glPushMatrix();
  - glPopMatrix();
    - Stack operation of matrix is very useful for constructing a hierarchical structures.
    - Ex: Render a car with four wheels.

Transformation -4

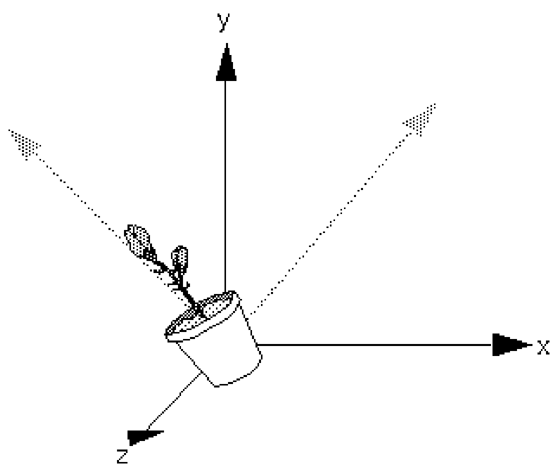
- OpenGL built-in transformation:
  - glTranslate{f,d}(TYPE x, TYPE, y, TYPE z);
    - Multiply a translation matrix into current matrix stack



The effect of **glTranslate()**

Transformation -5

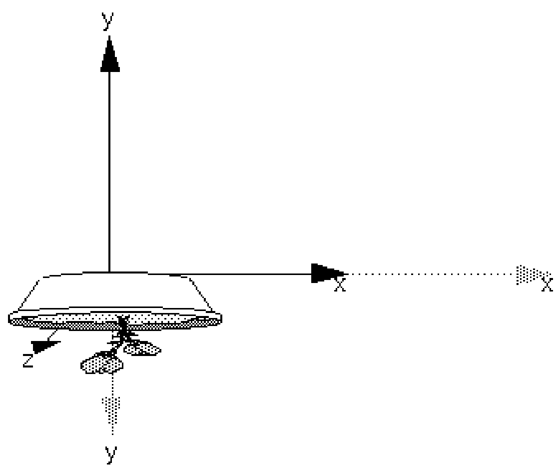
- OpenGL built-in transformation:
  - glRotate{f,d}(TYPE angle, TYPE x, TYPE y, TYPE z);
    - Multiply a rotation matrix about an arbitrary axis into current matrix stack



The effect of **glRotatef(**45.0, 0.0, 0.0, 1.0**)**

Transformation -6

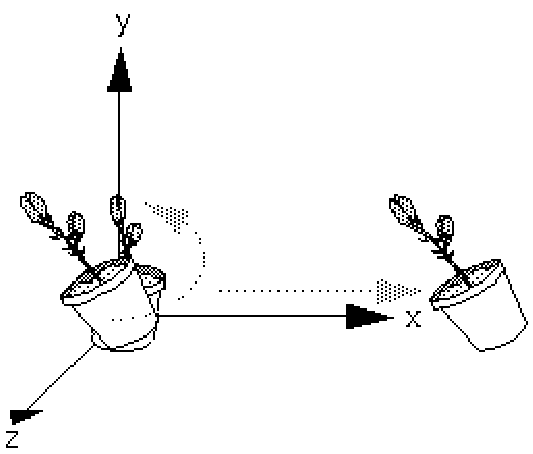
- OpenGL built-in transformation:
  - glScale{f,d}(TYPE x, TYPE y, TYPE z);
    - Multiplies current matrix by a matrix that scales an object along axes.

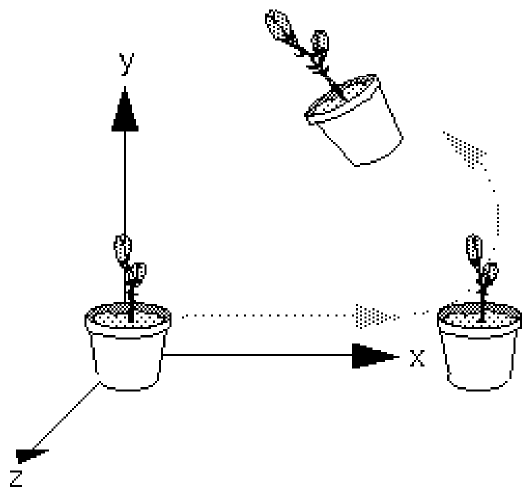


The effect of **glScalef(**2.0, -0.5, 1.0**)**

Transformation -7

- Rotating First or Translating First :





Transformation -8a

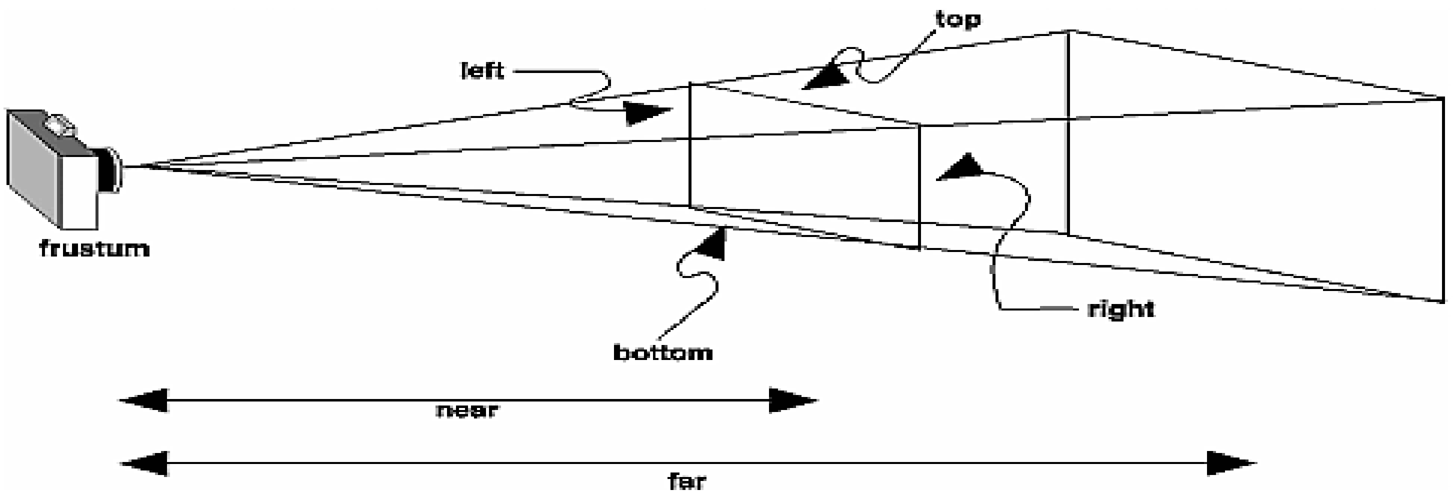
- Note:
  - By default, the viewpoint as well as objects in the scene are originally situated at the origin, and is looking down the negative z-axis, and has the positive y-axis as straight up.

Transformation -8b

- Viewing transformation
  - Choose your viewing system
    - Center-orientation-up system
      - Apply **gluLookAt** Utility routine.
        - gluLookAt(cx, cy, cz, atx, aty, atz, upx, upy, upz);
        - (cx, cy, cz) is the center of the camera
        - (atx, aty, atz) is where the camera look at
        - (upx, upy, upz) is the up vector of the camera
    - Polar coordinate system
      - Combine translation and two rotation.

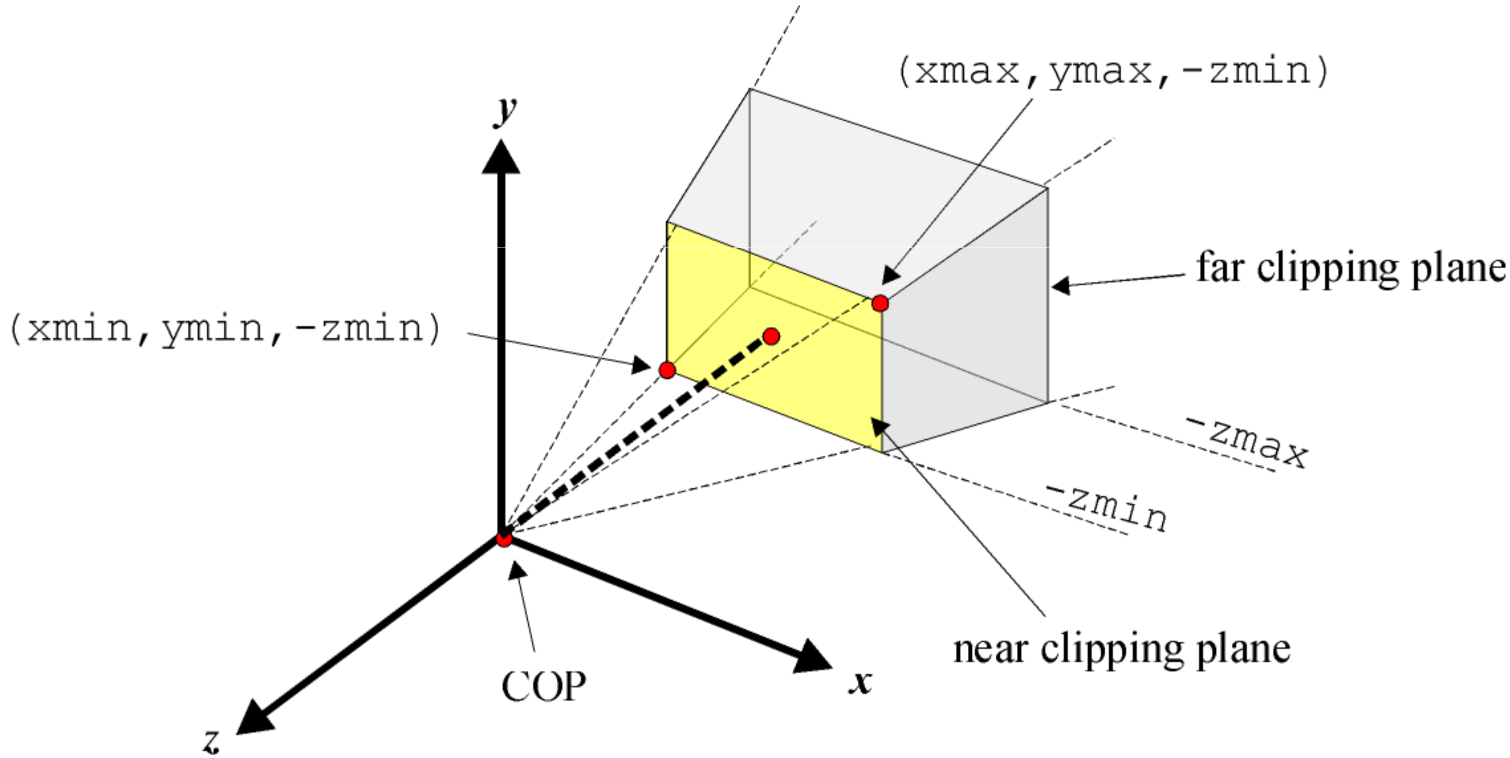
Transformation -9a

- Projection transformation: Perspective projection
  - glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);



Transformation -9b

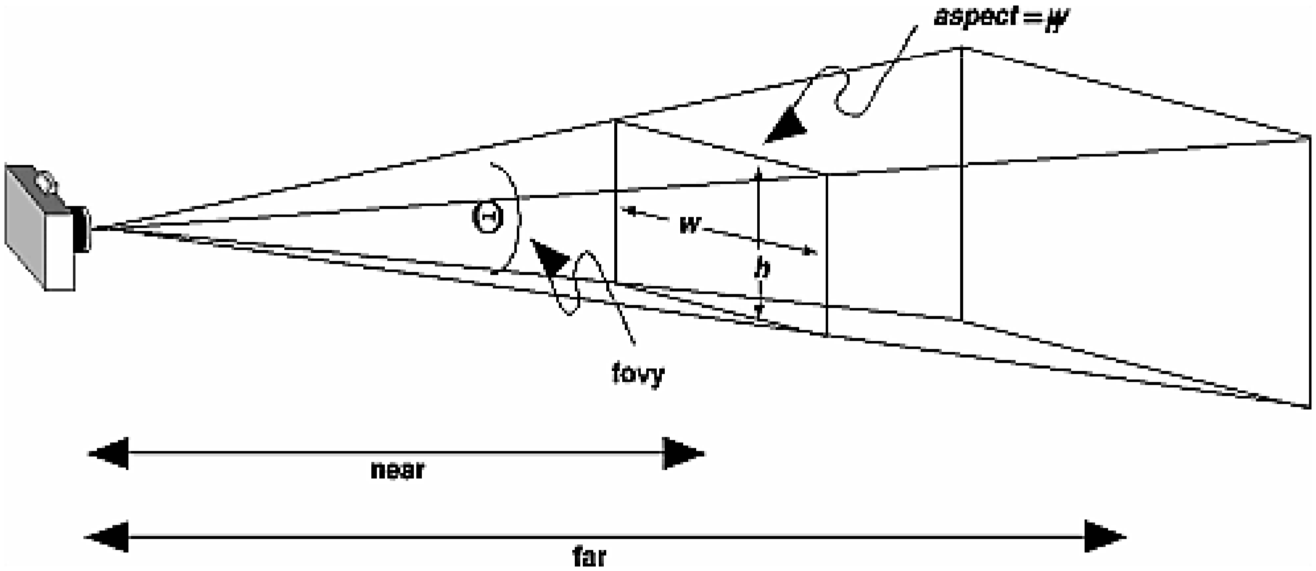
```
glFrustrum(xmin, xmax, ymin, ymax, zmin, zmax);
```



Non symmetric frustrums introduce *obliqueness* into the projection.
zmin and **zmax** are specified as positive distances along ***-z***

Transformation -10a

- gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far);

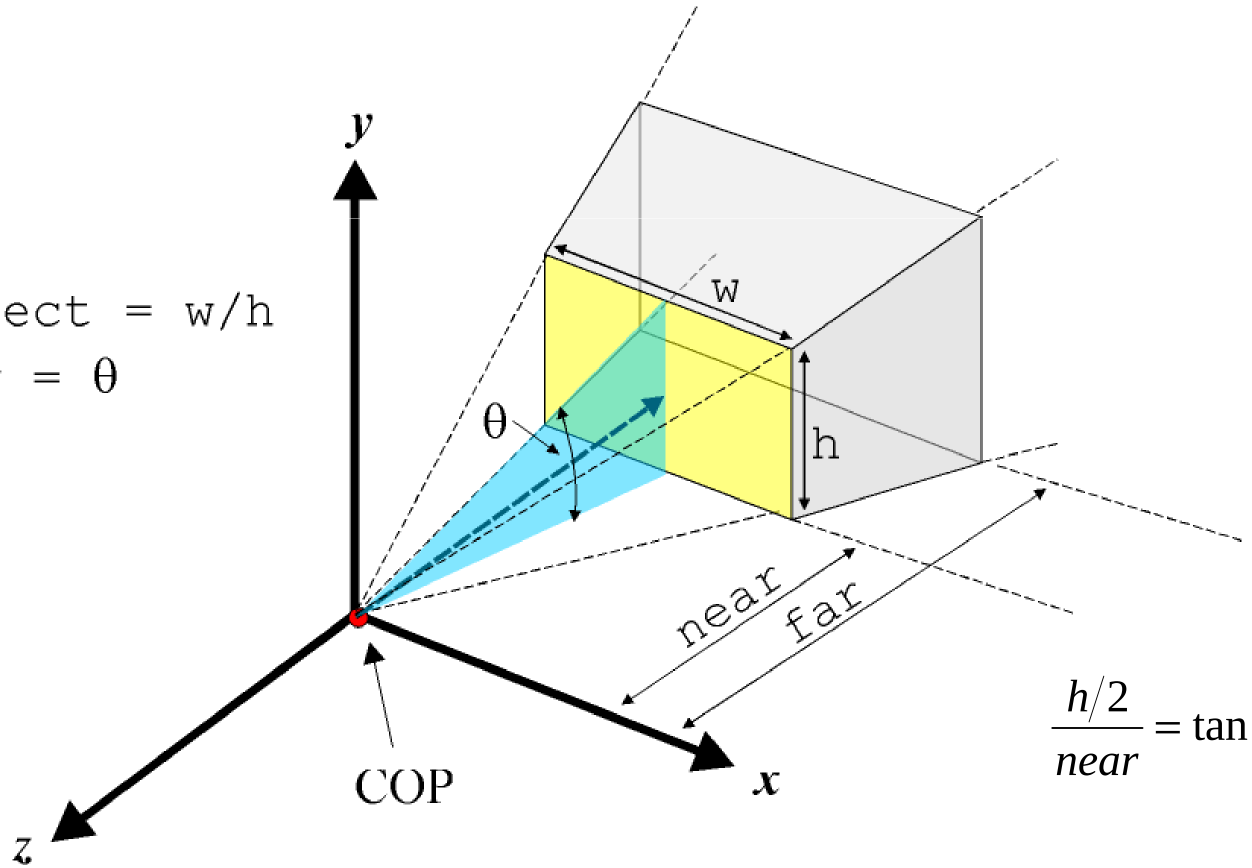


Transformation -10b

```
gluPerspective(fov, aspect, near, far);
```

aspect = w/h

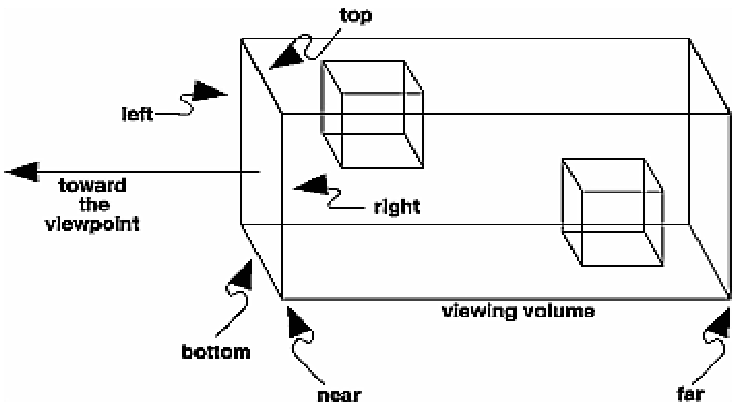
fov = θ



$$\frac{h/2}{near} = \tan\frac{\theta}{2} \Rightarrow h = 2\,near\tan\frac{\theta}{2}$$

Transformation -11a

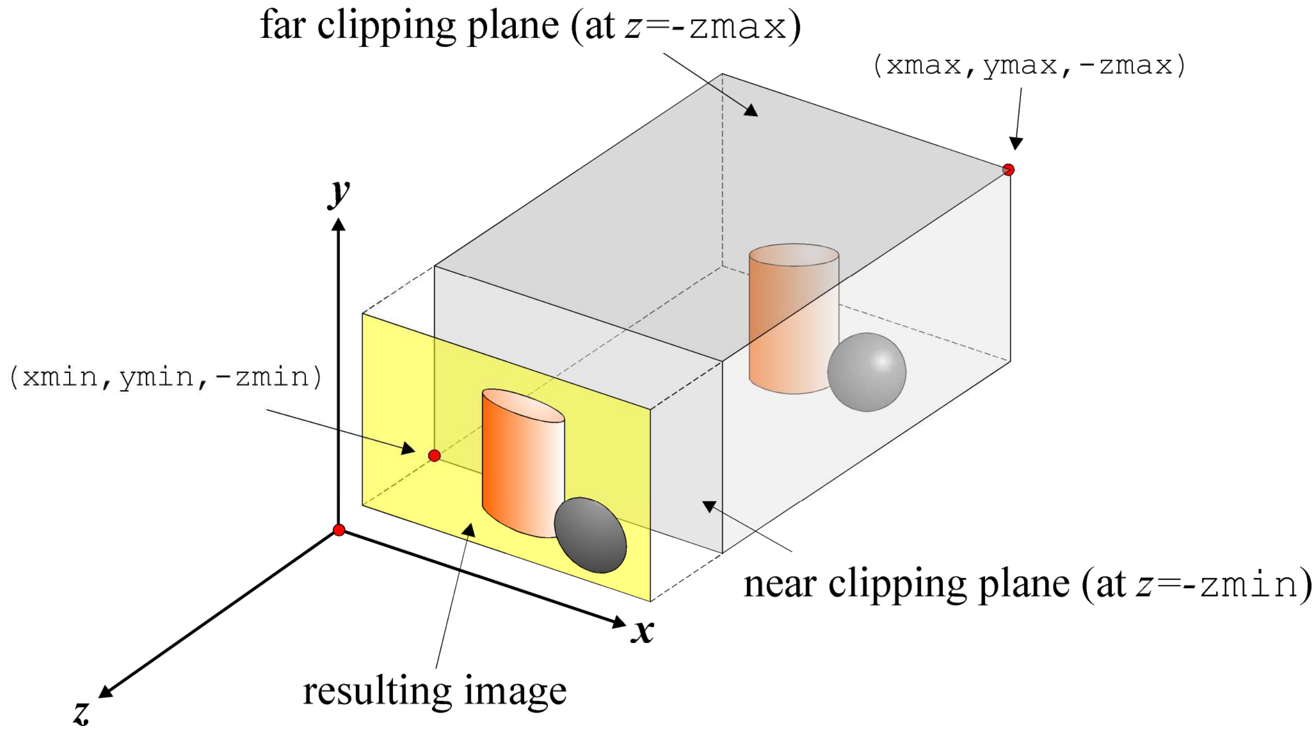
- Projection transformation: Orthogonal projection
  - glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);



  - gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);
    - A helper to create a 2D projection matrix

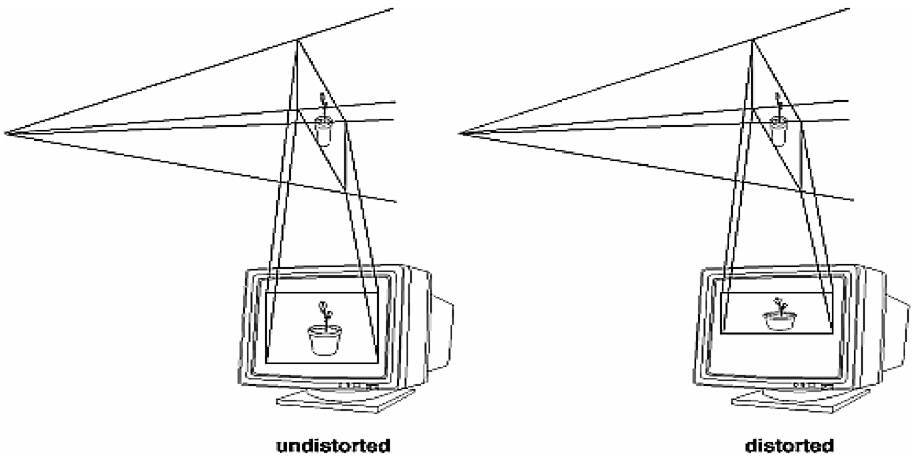
Transformation -11b

```
glOrtho(xmin, xmax, ymin, ymax, zmin, zmax);
```



Transformation -12

- Viewport transformation
  - glViewport(GLint x, GLint y, GLsizei w, GLsizei h);
    - Initial viewport is as the same size as the window



Viewport to Window Transformation

- `(x,y)` = location of bottom left of viewport within the window
- `width,height` = dimension in pixels of the viewport $\Rightarrow$

$$x_w = (x_n + 1)\left(\frac{\texttt{width}}{2}\right) + \texttt{x} \quad y_w = (y_n + 1)\left(\frac{\texttt{height}}{2}\right) + \texttt{y}$$

- normally we re-create the window after a window resize event to ensure a correct mapping between viewport and window dimensions

Shading

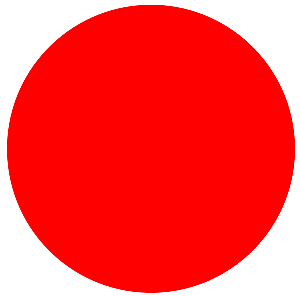
74

Objectives

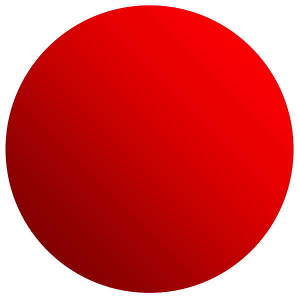
- Learn to shade objects so their images appear three-dimensional
- Introduce the types of light-material interactions
- Build a simple reflection model---the Phong model--- that can be used with real time graphics hardware

Why we need shading

- Suppose we build a model of a sphere using many polygons and color it with **`glColor`**. We get something like

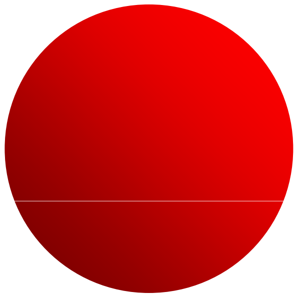


- But we want



Shading

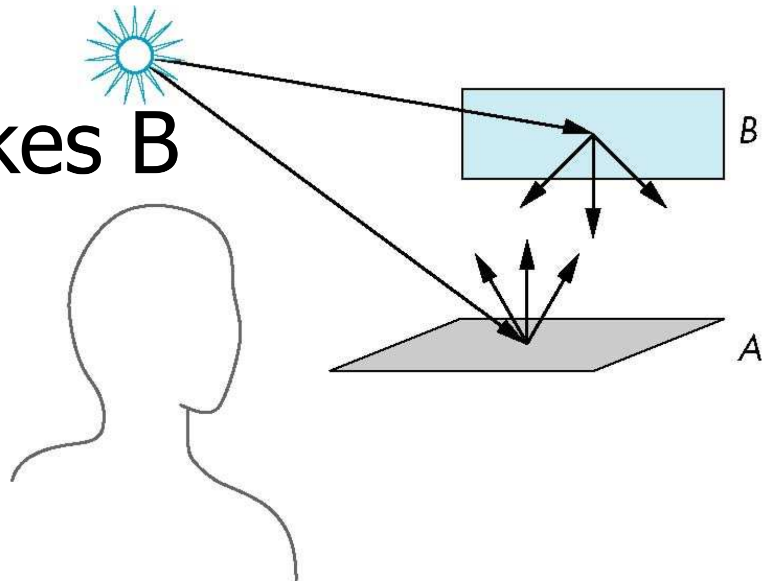
- Why does the image of a real sphere look like



- Light-material interactions cause each point to have a different color or shade
- Need to consider
  - Light sources
  - Material properties
  - Surface orientation

Scattering

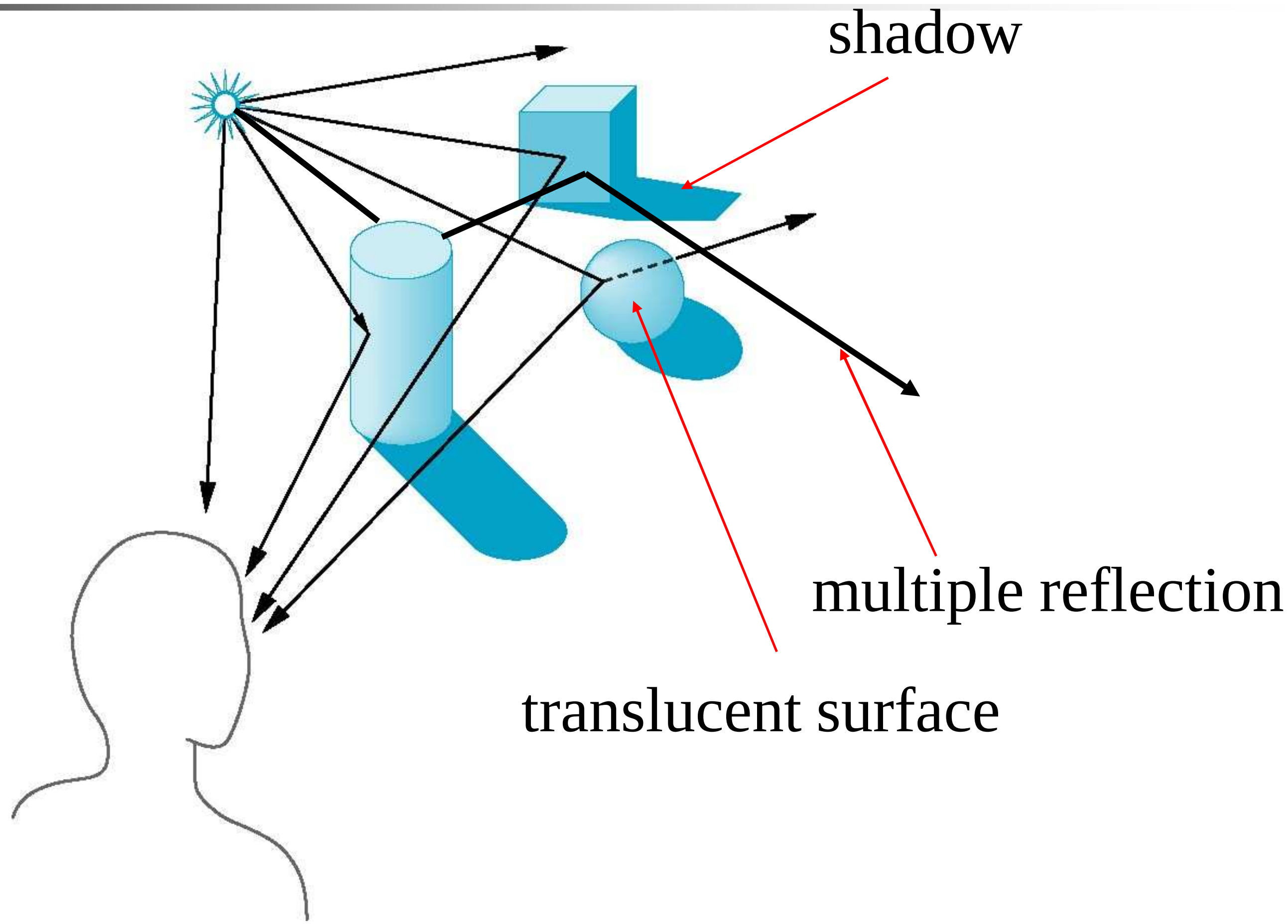
- Light strikes A
  - Some scattered
  - Some absorbed
- Some of scattered light strikes B
  - Some scattered
  - Some absorbed
- Some of this scattered light strikes A and so on



Rendering Equation

- The infinite scattering and absorption of light can be described by the *rendering equation*
  - Cannot be solved in general
  - Ray tracing is a special case for perfectly reflecting surfaces
- Rendering equation is global and includes
  - Shadows
  - Multiple scattering from object to object

Global Effects



Local vs Global Rendering

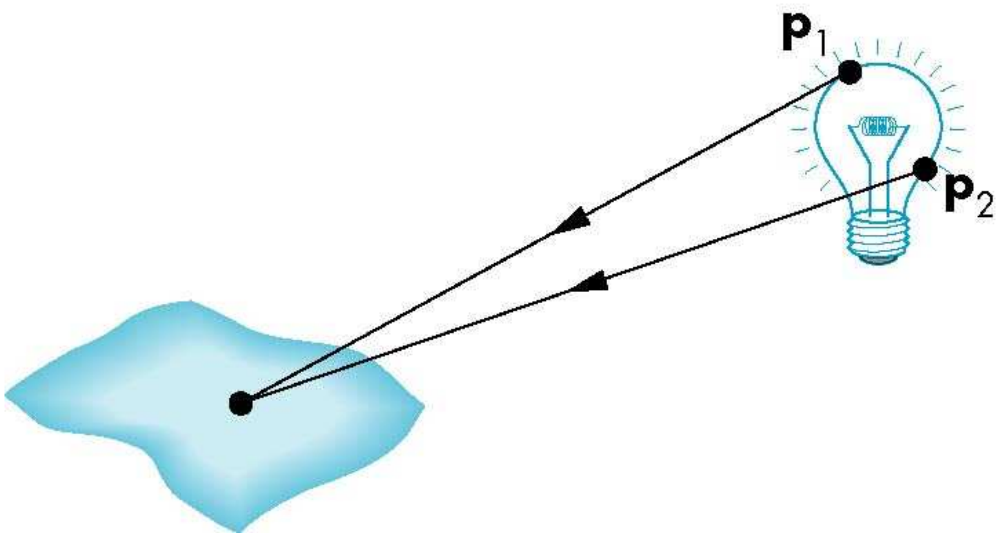
- Correct shading requires a global calculation involving all objects and light sources
  - Incompatible with pipeline model which shades each polygon independently (local rendering)
- However, in computer graphics, especially real time graphics, we are happy if things "look right"
  - Exist many techniques for approximating global effects

Light-Material Interaction

- Light that strikes an object is partially absorbed and partially scattered (reflected)
- The amount reflected determines the color and brightness of the object
  - A surface appears red under white light because the red component of the light is reflected and the rest is absorbed
- The reflected light is scattered in a manner that depends on the smoothness and orientation of the surface

Light Sources

General light sources are difficult to work with because we must integrate light coming from all points on the source

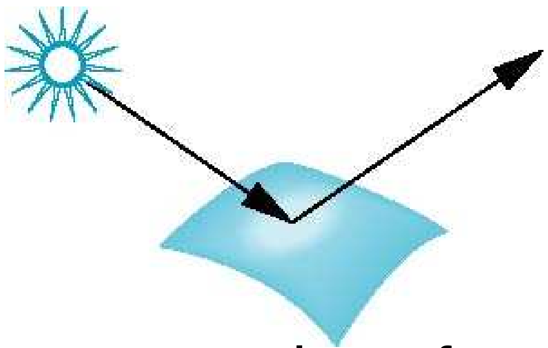


Simple Light Sources

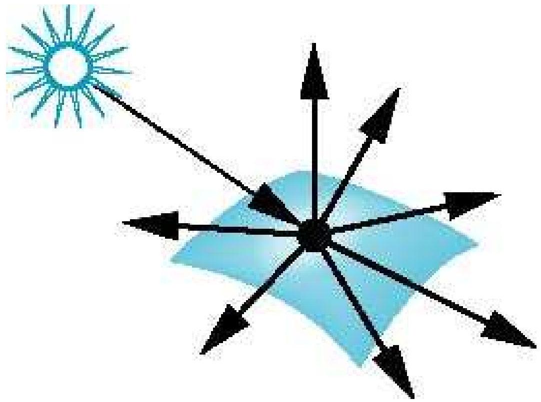
- Point source
  - Model with position and color
  - Distant source = infinite distance away (parallel)
- Spotlight
  - Restrict light from ideal point source
- Ambient light
  - Same amount of light everywhere in scene
  - Can model contribution of many sources and reflecting surfaces

Surface Types

- The smoother a surface, the more reflected light is concentrated in the direction a perfect mirror would reflected the light
- A very rough surface scatters light in all directions



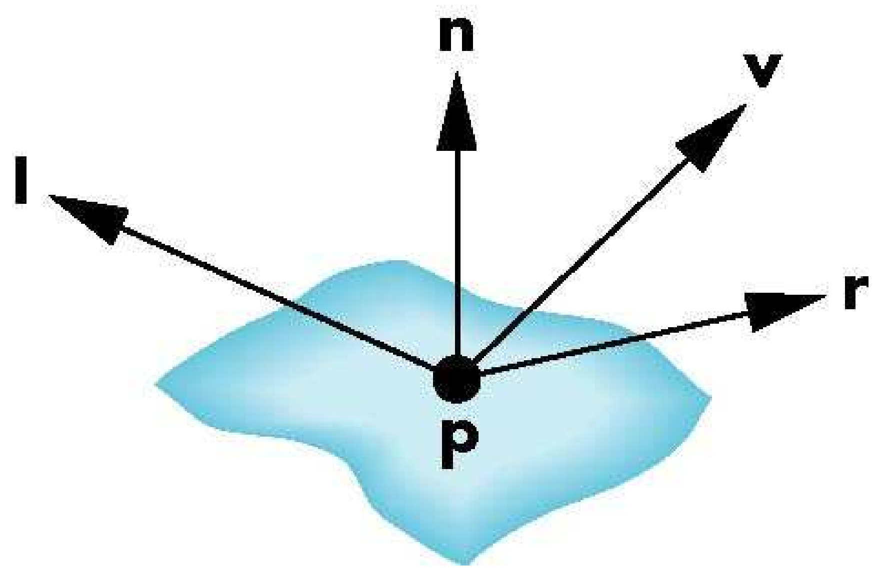
smooth surface



rough surface

Phong Model

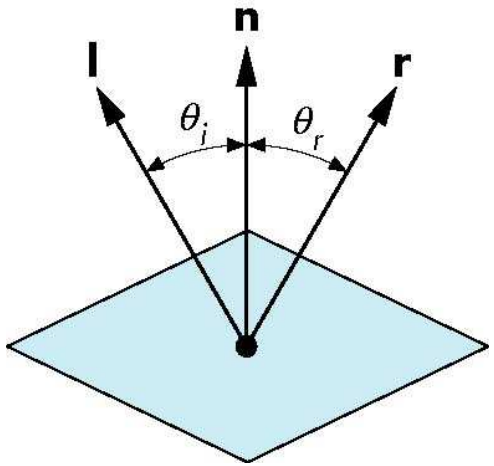
- A simple model that can be computed rapidly
- Has three components
  - Diffuse
  - Specular
  - Ambient
- Uses four vectors
  - To source
  - To viewer
  - Normal
  - Perfect reflector



Ideal Reflector

- Normal is determined by local orientation
- Angle of incidence = angle of relection
- The three vectors must be coplanar

$$\mathbf{r} = 2\,(\mathbf{l} \cdot \mathbf{n})\,\mathbf{n} - \mathbf{l}$$

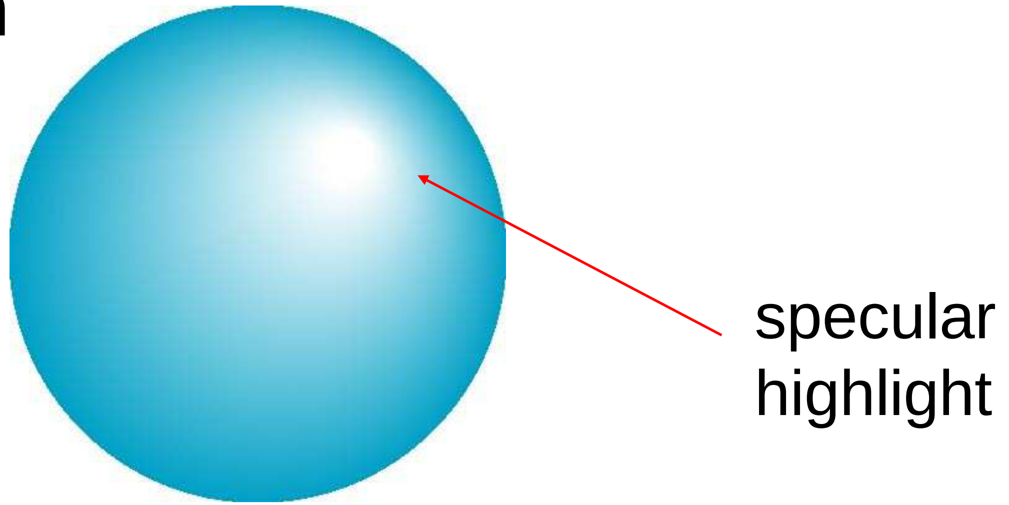


Lambertian Surface

- Perfectly diffuse reflector
- Light scattered equally in all directions
- Amount of light reflected is proportional to the vertical component of incoming light
  - reflected light ~$\cos\theta_i$
  - $\cos\theta_i = \mathbf{l} \cdot \mathbf{n}$ if vectors normalized
  - There are also three coefficients, k_r, k_b, k_g that show how much of each color component is reflected

Specular Surfaces

- Most surfaces are neither ideal diffusers nor perfectly specular (ideal reflectors)
- Smooth surfaces show specular highlights due to incoming light being reflected in directions concentrated close to the direction of a perfect reflection



Modeling Specular Relections

- Phong proposed using a term that dropped off as the angle between the viewer and the ideal reflection increased

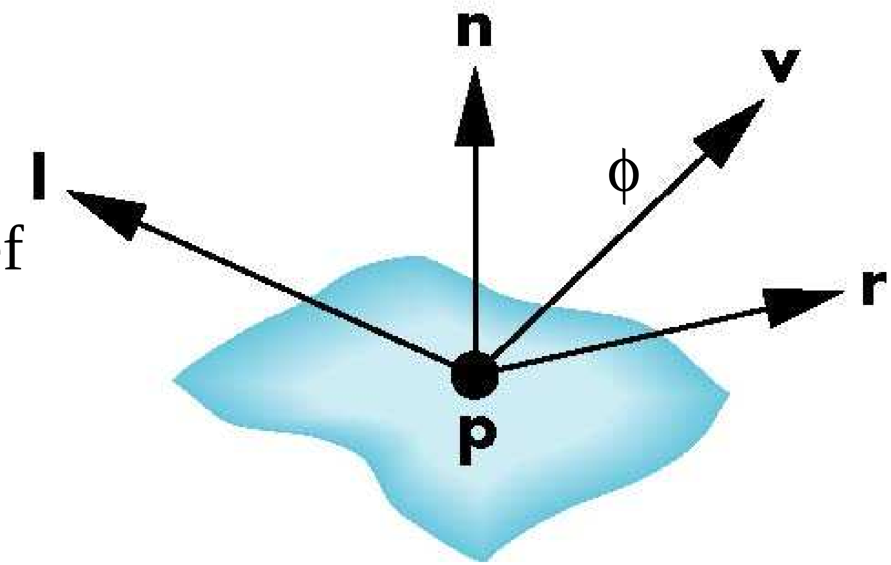
$$\mathbf{I}_r \sim k_s\, \mathbf{I} \cos^{\alpha}\phi$$

reflected intensity

absorption coef

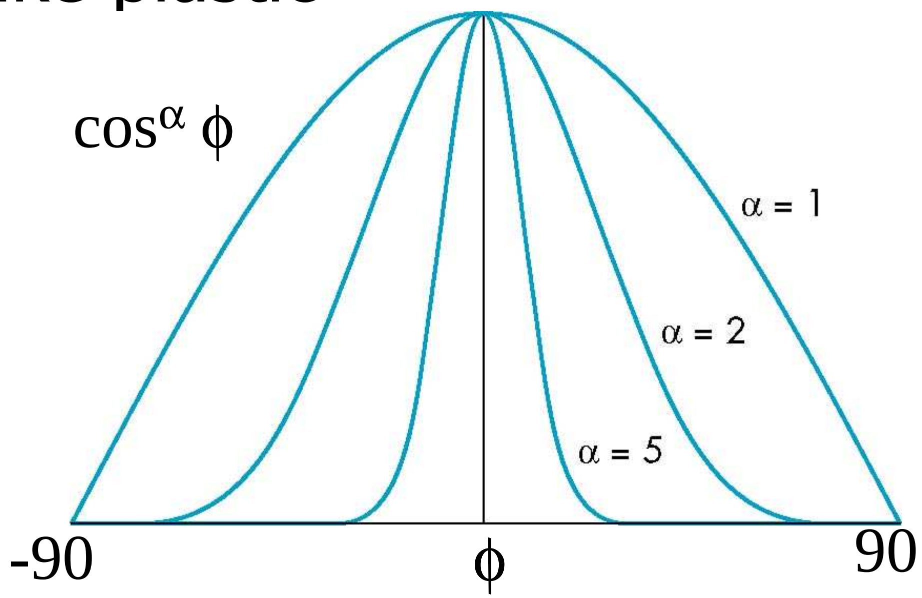
incoming intensity

shininess coef



The Shininess Coefficient

- Values of α between 100 and 200 correspond to metals
- Values between 5 and 10 give surface that look like plastic



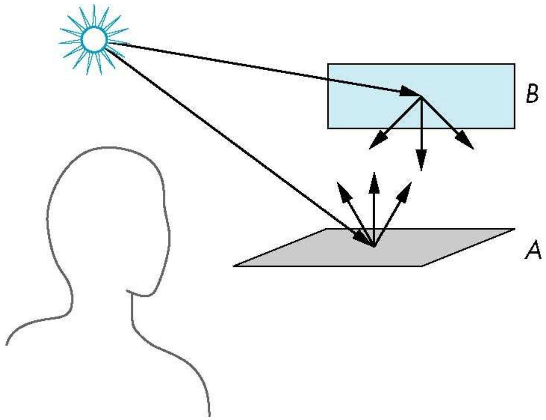
Ambient Light

- Ambient light is the result of multiple interactions between (large) light sources and the objects in the environment
- Amount and color depend on both the color of the light(s) and the material properties of the object
- Add k_a I_a to diffuse and specular terms

reflection coef (k_a)   intensity of ambient light (I_a)

Distance Terms

- The light from a point source that reaches a surface is inversely proportional to the square of the distance between them
- We can add a attenuation factor of the form $1/(ad + bd + cd^2)$ to the diffuse and specular terms



- The constant and linear terms soften the effect of the point source
- Also known as depth-cueing

Light Sources

- In the Phong Model, we add the results from each light source
- Each light source has separate diffuse, specular, and ambient terms to allow for maximum flexibility even though this form does not have a physical justification
- Separate red, green and blue components
- Hence, 9 coefficients for each point source
  - I_{dr}, I_{dg}, I_{db}, I_{sr}, I_{sg}, I_{sb}, I_{ar}, I_{ag}, I_{ab}

Material Properties

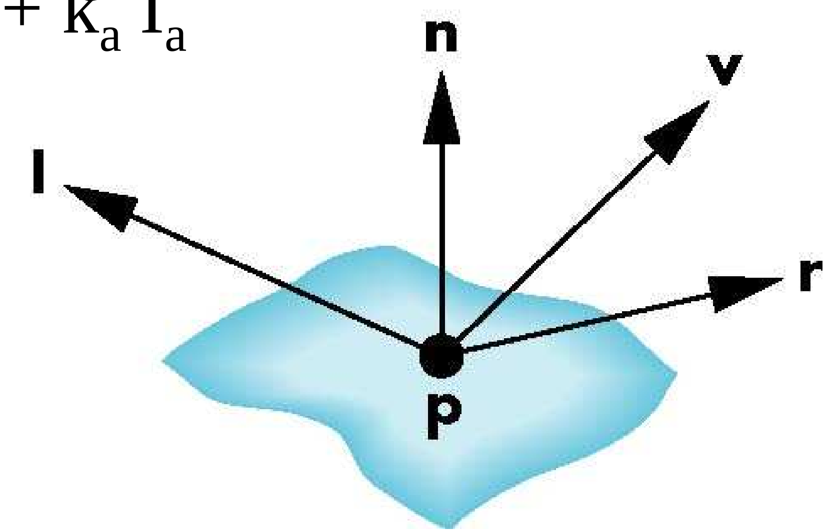
- Material properties match light source properties
  - Nine absorbtion coefficients
    - k_{dr}, k_{dg}, k_{db}, k_{sr}, k_{sg}, k_{sb}, k_{ar}, k_{ag}, k_{ab}
  - Shininess coefficient α

Adding up the Components

For each light source and each color component, the Phong model can be written (without the attenuation factor) as

$$I = k_d I_d\ \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{v} \cdot \mathbf{r})^{\alpha} + k_a I_a$$

For each color component we add contributions from all sources



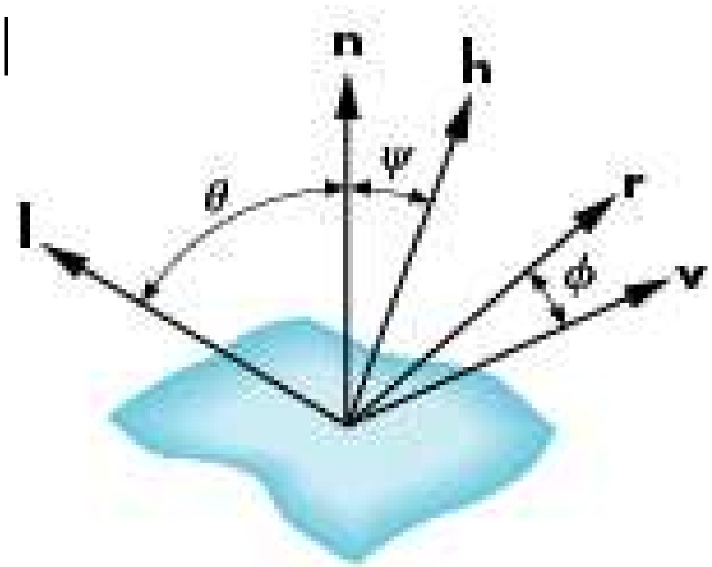
Modified Phong Model

- The specular term in the Phong model is problematic because it requires the calculation of a new reflection vector and view vector for each vertex
- Blinn suggested an approximation using the halfway vector that is more efficient

The Halfway Vector

- **h** is normalized vector halfway between **l** and **v**

$$\mathbf{h} = (\mathbf{l} + \mathbf{v}) / |\mathbf{l} + \mathbf{v}|$$

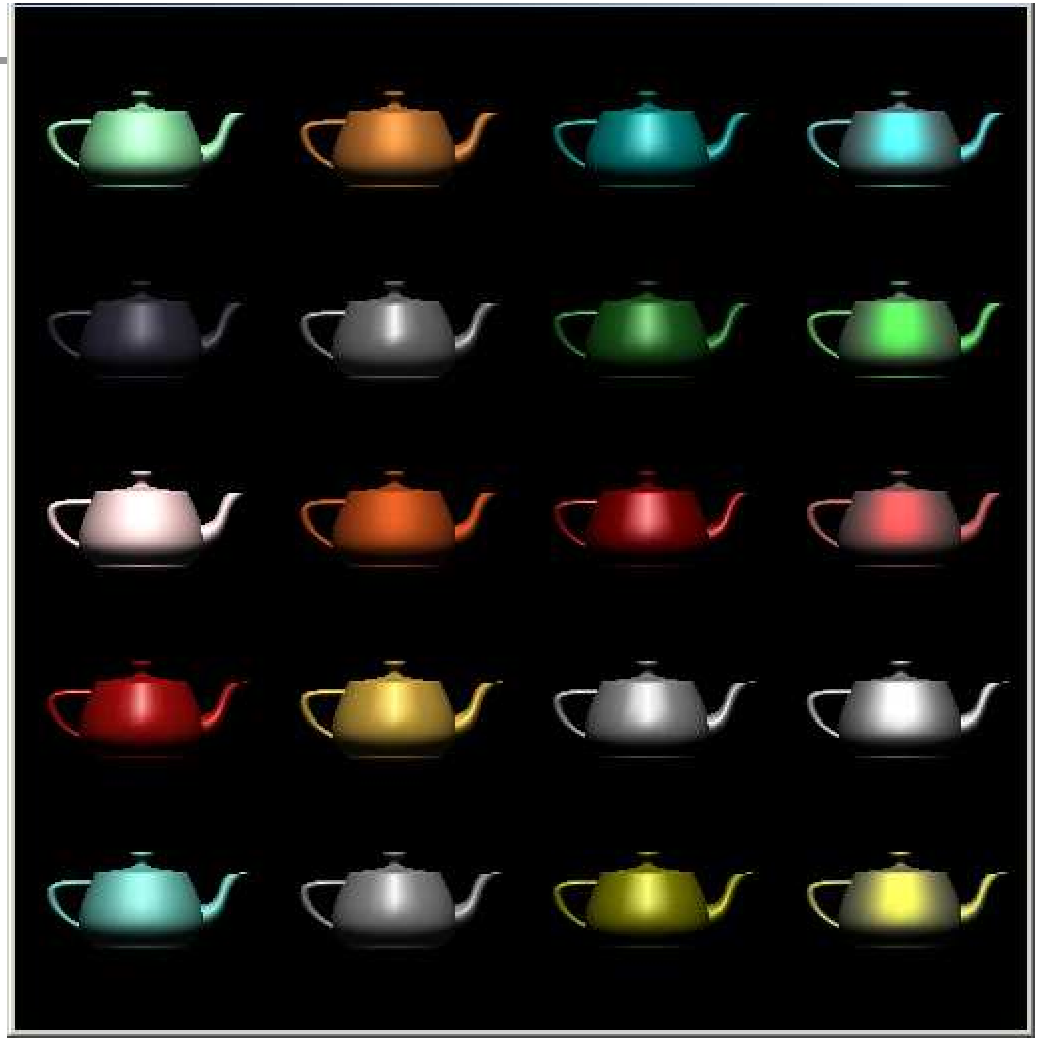


Using the halfway vector

- Replace $(\mathbf{v} \cdot \mathbf{r})^{\alpha}$ by $(\mathbf{n} \cdot \mathbf{h})^{\beta}$
- β is chosen to match shineness
- Note that halway angle is half of angle between $\mathbf{r}$ and $\mathbf{v}$ if vectors are coplanar
- Resulting model is known as the modified Phong or Blinn lighting model
  - Specified in OpenGL standard

Example

Only differences in these teapots are the parameters in the modified Phong model



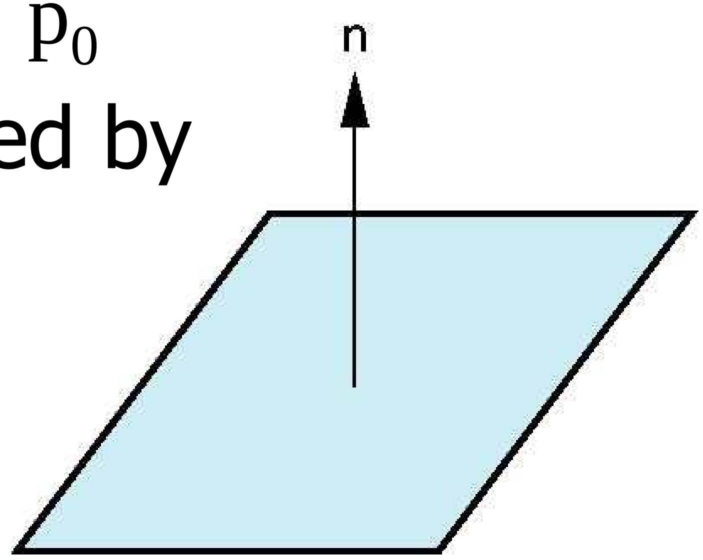
Computation of Vectors

- **l** and **v** are specified by the application
- computer calculates **r** from **l** and **n**
- Problem is determining **n**
- how we determine **n** differs depending on underlying representation of surface
- OpenGL leaves determination of normal to application
  - Exception for GLU quadrics and Bezier surfaces (

Plane Normals

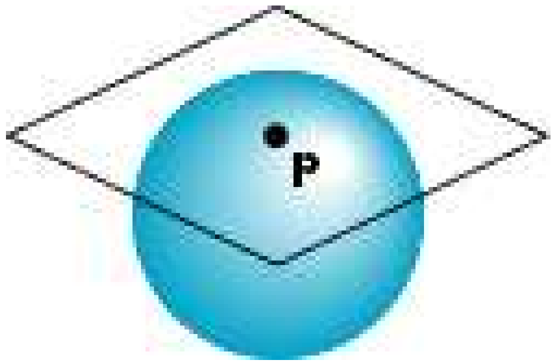
- Equation of plane: $ax+by+cz+d = 0$
- plane is determined by three points p_0, p_2, p_3 or normal $\mathbf{n}$ and p_0
- Normal can be obtained by

$$\mathbf{n} = (p_2-p_0) \times (p_1-p_0)$$



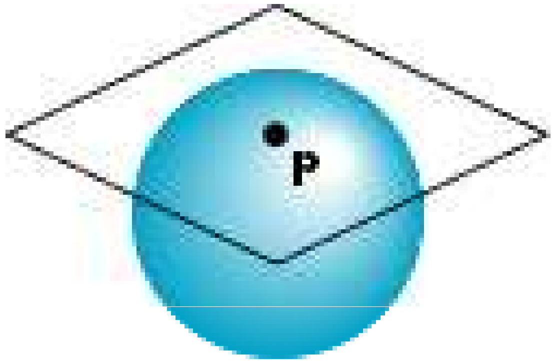
Normal to Sphere

- Implicit function $f(x,y.z)=0$
- Normal given by gradient
- $f(x, y, z) = x^2 + y^2 + z^2 - 1 = 0$
- Sphere $f(\mathbf{p})=\mathbf{p}\cdot\mathbf{p} - 1 = 0$
- $n = [df/dx, df/dy, df/dz]^T=\mathbf{p}$



Parametric Form

- For sphere



$$x=x(u,v)=\cos u \sin v$$
$$y=y(u,v)=\cos u \cos v$$
$$z= z(u,v)=\sin u$$

- Tangent plane determined by vectors

$$\partial \mathbf{p}/\partial u = [\partial x/\partial u, \partial y/\partial u, \partial z/\partial u]T$$
$$\partial \mathbf{p}/\partial v = [\partial x/\partial v, \partial y/\partial v, \partial z/\partial v]T$$

- Normal given by cross product

$$\mathbf{n} = \partial \mathbf{p}/\partial u \times \partial \mathbf{p}/\partial v$$

General Case

- We can compute parametric normals for other simple cases
  - Quadrics
  - Parameteric polynomial surfaces
    - Bezier surface patches

Steps in OpenGL shading

1. Enable shading and select Lighting Model
2. Specify normals
3. Specify material properties
4. Specify light sources

Normals

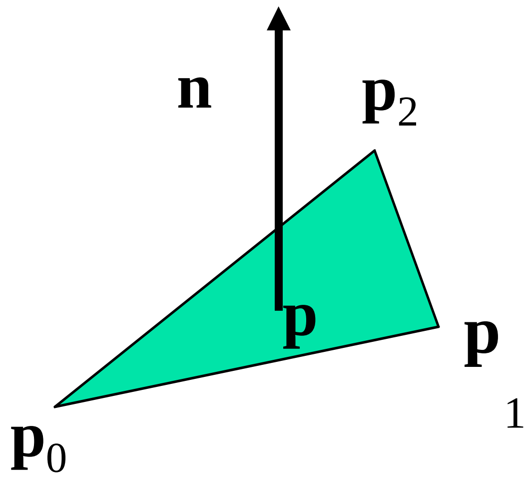
- In OpenGL the normal vector is part of the state
- Set by **`glNormal*()`**
  - **`glNormal3f(x, y, z);`**
  - **`glNormal3fv(p);`**
- Usually we want to set the normal to have unit length so cosine calculations are correct
  - Length can be affected by transformations
  - Note that scaling does not preserved length
  - **`glEnable(GL_NORMALIZE)`** allows for auto normalization at a performance penalty

Normal for Triangle

plane $\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0$

$\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)$

normalize $\mathbf{n} \leftarrow \mathbf{n} / |\mathbf{n}|$



Note that right-hand rule determines outward face

Enabling Shading

- Shading calculations are enabled by
  - **`glEnable(GL_LIGHTING)`**
  - Once lighting is enabled, glColor() ignored
- Must enable each light source individually
  - **`glEnable(GL_LIGHTi)`** i=0,1…..
- Can choose light model parameters
  - **`glLightModeli(name,parameter)`**
    - **`GL_LIGHT_MODEL_AMBIENT -`** ambient RGBA intensity of the entire scene
    - **`GL_LIGHT_MODEL_LOCAL_VIEWER-`**how specular reflection angles are calculated
    - **`GL_LIGHT_MODEL_TWO_SIDED -`** specifies one-sided or two-sided lighting
    - **`GL_LIGHT_MODEL_COLOR_CONTROL – assumes GL_SINGLE_COLOR or GL_SEPARATE_SPECULAR_COLOR`**

Light Properties

`glLightfv( light, property, value );`

- ***`light`*** specifies which light
  - multiple lights, starting with **`GL_LIGHT0`**

```
glGetIntegerv( GL_MAX_LIGHTS, &n );
```

- ***`properties`***
  - colors
  - position and type
  - attenuation

Defining a Light Source

- For each light source, we can set an RGBA for the diffuse, specular, and ambient components, and for the position

```
GL float diffuse0[]={1.0, 0.0, 0.0, 1.0};
GL float ambient0[]={1.0, 0.0, 0.0, 1.0};
GL float specular0[]={1.0, 0.0, 0.0, 1.0};
Glfloat light0_pos[]={1.0, 2.0, 3,0, 1.0};

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glLightv(GL_LIGHT0, GL_POSITION, light0_pos);
glLightv(GL_LIGHT0, GL_AMBIENT, ambient0);
glLightv(GL_LIGHT0, GL_DIFFUSE, diffuse0);
glLightv(GL_LIGHT0, GL_SPECULAR, specular0);
```

Distance and Direction

- The source colors are specified in RGBA
- The position is given in homogeneous coordinates
  - If w =1.0, we are specifying a finite location
  - If w =0.0, we are specifying a parallel source with the given direction vector
- The coefficients in the distance terms are by default a=1.0 (constant terms), b=c=0.0 (linear and quadratic terms). Change by

```
a= 0.80;
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, a);
```

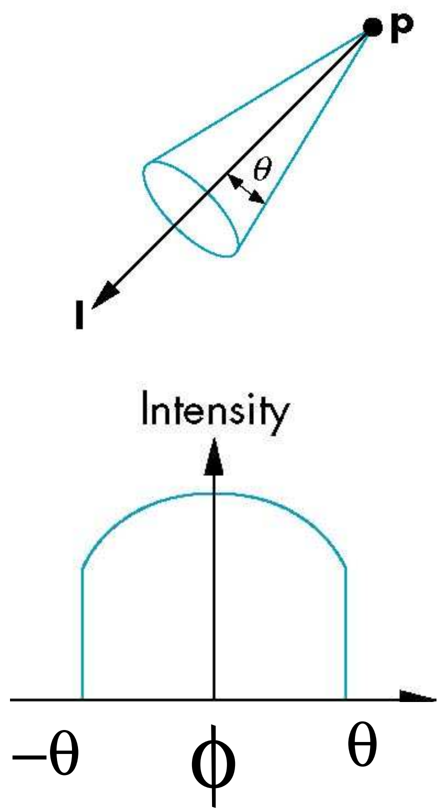
Light Attenuation

- decrease light intensity with distance
  - ***GL_CONSTANT_ATTENUATION***
  - ***GL_LINEAR_ATTENUATION***
  - ***GL_QUADRATIC_ATTENUATION***

$$f_i = \frac{1}{k_c + k_l d + k_q d^2}$$

Spotlights

- Use **`glLightv`** to set
  - Direction **`GL_SPOT_DIRECTION`**
  - Cutoff **`GL_SPOT_CUTOFF`**
  - Attenuation **`GL_SPOT_EXPONENT`**
    - Proportional to $\cos^{\alpha}\phi$



Global Ambient Light

- Ambient light depends on color of light sources
  - A red light in a white room will cause a red ambient term that **disappears** when the light is turned off
- OpenGL also allows a global ambient term that is often helpful for testing
  - **`glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient)`**

Material Properties

- Material properties are also part of the OpenGL state and match the terms in the modified Phong model
- Set by **`glMaterialv()`**

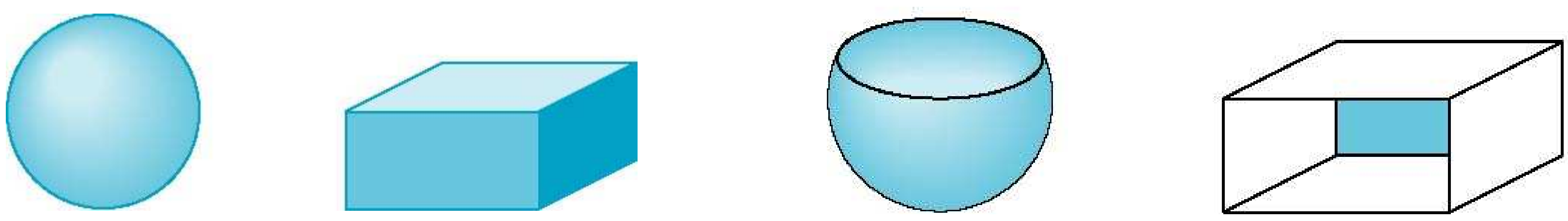
```
GLfloat ambient[] = {0.2, 0.2, 0.2, 1.0};
GLfloat diffuse[] = {1.0, 0.8, 0.0, 1.0};
GLfloat specular[] = {1.0, 1.0, 1.0, 1.0};
GLfloat shine = 100.0
glMaterialf(GL_FRONT, GL_AMBIENT, ambient);
glMaterialf(GL_FRONT, GL_DIFFUSE, diffuse);
glMaterialf(GL_FRONT, GL_SPECULAR, specular);
glMaterialf(GL_FRONT, GL_SHININESS, shine);
```

Transparency

- Material properties are specified as RGBA values
- The A value can be used to make the surface translucent
- The default is that all surfaces are opaque regardless of A
- Later we will enable blending and use this feature

Front and Back Faces

- The default is shade only front faces which works correctly for convex objects
- If we set two sided lighting, OpenGL will shade both sides of a surface
- Each side can have its own properties which are set by using **`GL_FRONT`**, **`GL_BACK`**, or **`GL_FRONT_AND_BACK`** in **`glMaterialf`**



back faces not visible back faces visible

Emissive Term

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

```
GLfloat emission[] = 0.0, 0.3, 0.3, 1.0);
glMaterialf(GL_FRONT, GL_EMISSION, emission);
```

Efficiency

- Because material properties are part of the state, if we change materials for many surfaces, we can affect performance
- We can make the code cleaner by defining a material structure and setting all materials during initialization

```
typedef struct materialStruct {
   GLfloat ambient[4];
   GLfloat diffuse[4];
   GLfloat specular[4];
   GLfloat shineness;
} MaterialStruct;
```

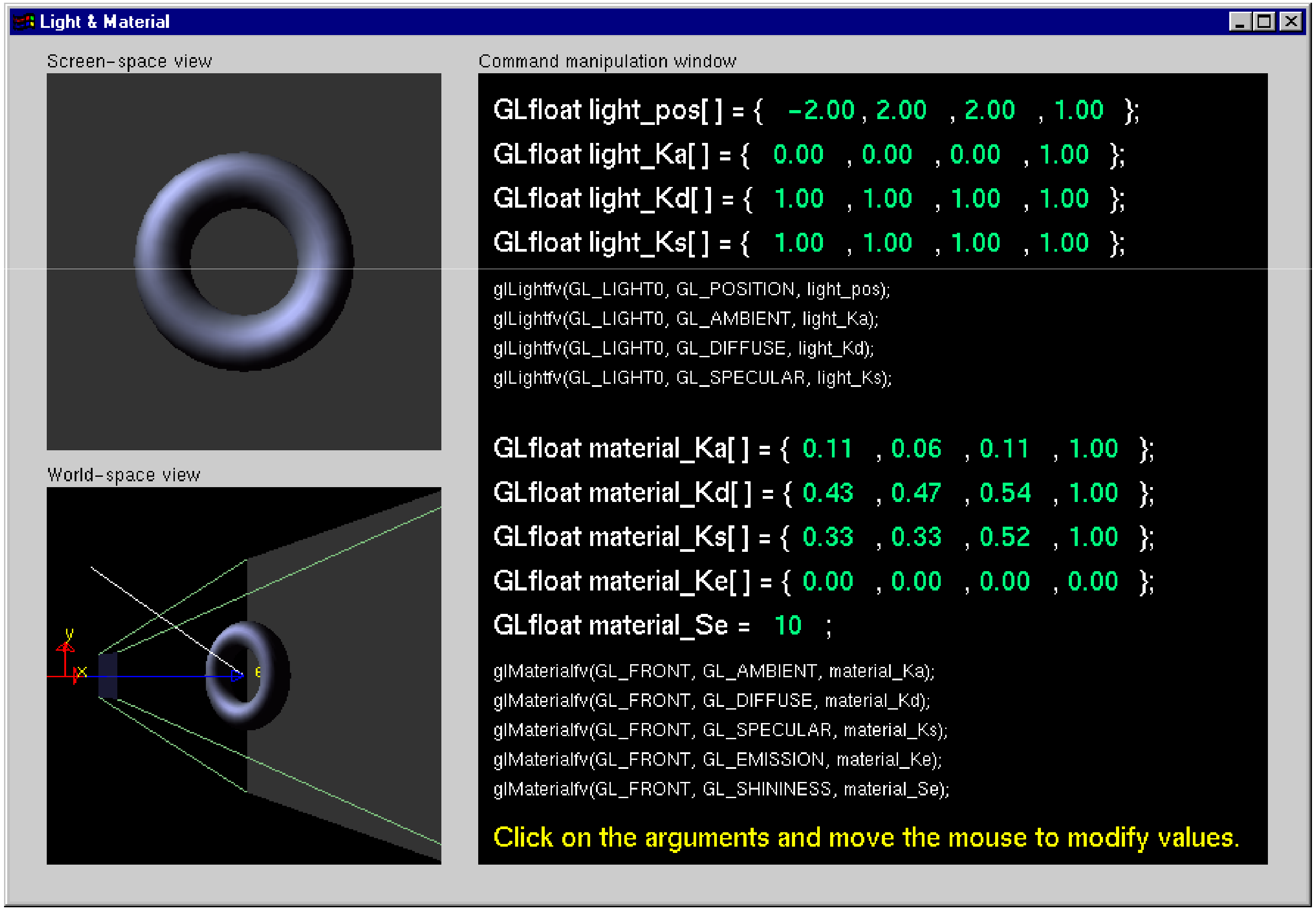
- We can then select a material by a pointer

The Mathematics of Lighting

$$
\begin{aligned}
\text{VertexColor} = {} & \text{emission}_{\text{material}} + \\
& \text{ambient}_{\text{lightmodel}} * \text{ambient}_{\text{material}} + \\
& \sum_{i=0}^{n-1} \left[1 / (K_c + K_l * d + K_q * d^2) \right]_i * (\text{spotlight\_effect})_i * \\
& \quad [\, \text{ambient}_{\text{light}} * \text{ambient}_{\text{material}} + \\
& \quad (\max \{ l \cdot n, 0 \}) * \text{diffuse}_{\text{light}} * \text{diffuse}_{\text{material}} + \\
& \quad (\max \{ h \cdot n, 0 \})^{\text{shininess}} * \text{specular}_{\text{light}} * \text{specular}_{\text{material}} \,]_i
\end{aligned}
$$

n: vertex normal
l: light vector – (light_pos – vertex)
h: half-vector - sum of the light vector with the viewing vector (view_pos – vertex)

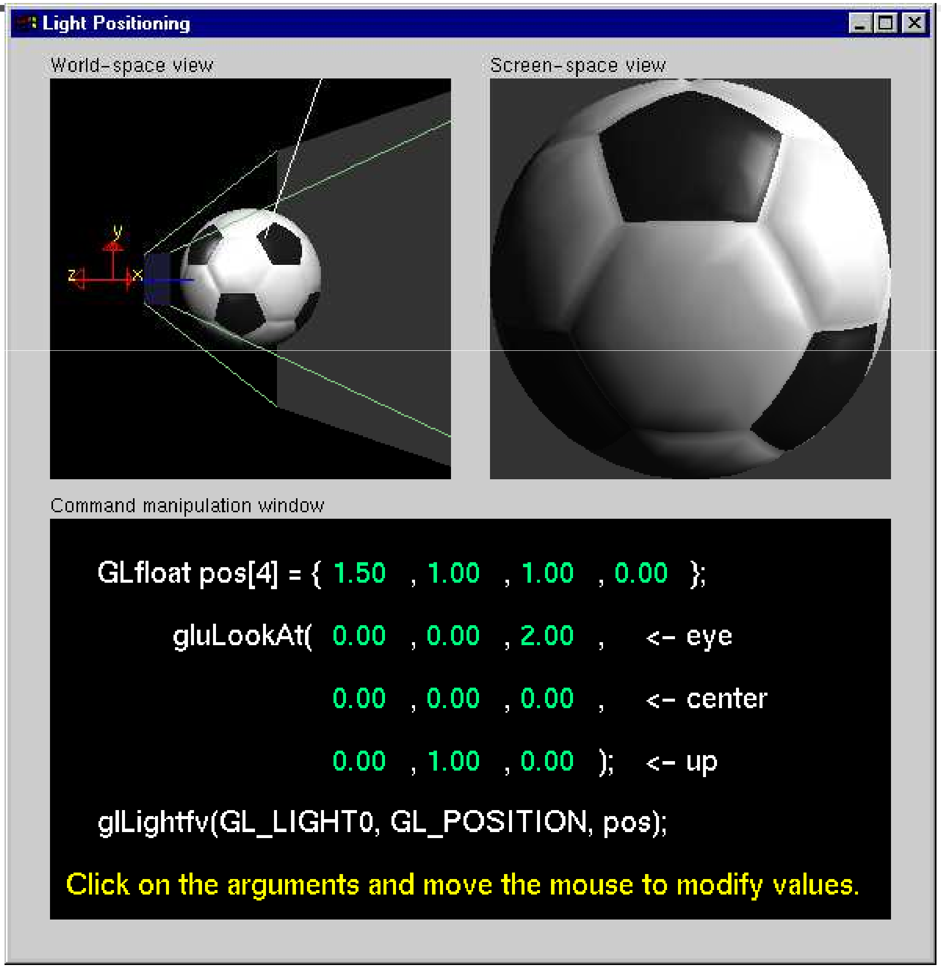
Light Material Tutorial



Moving Light Sources

- Light sources are geometric objects whose positions or directions are affected by the model-view matrix
- 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

Light Position Tutorial



Polygonal Shading

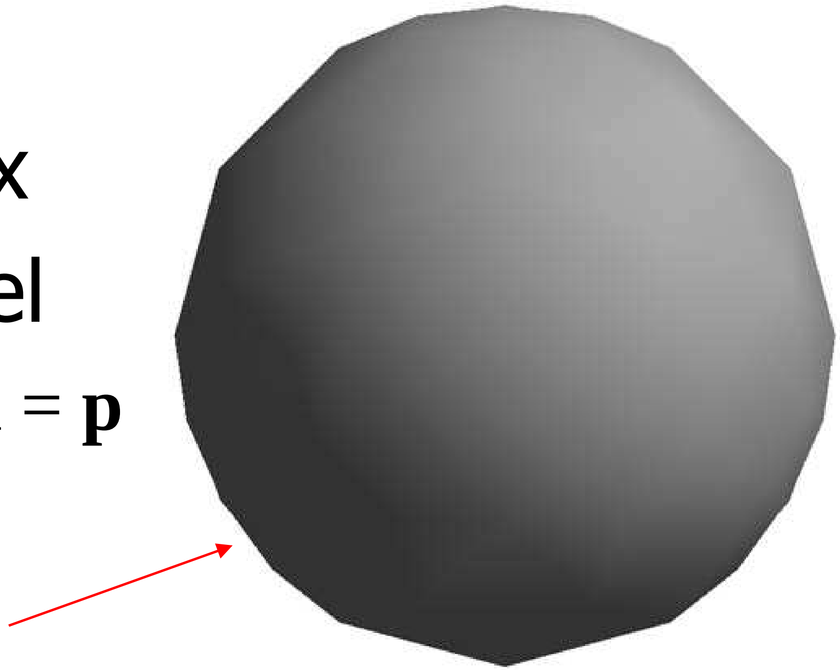
- Shading calculations are done for each vertex
  - Vertex colors become vertex shades
- By default, vertex shades are interpolated across the polygon
  - **`glShadeModel(GL_SMOOTH);`**
- If we use **`glShadeModel(GL_FLAT);`** the color at the first vertex will determine the shade of the whole polygon

Polygon Normals

- Polygons have a single normal
  - Shades at the vertices as computed by the Phong model can be almost same
  - Identical for a distant viewer (default) or if there is no specular component
- Consider model of sphere
- Want 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*

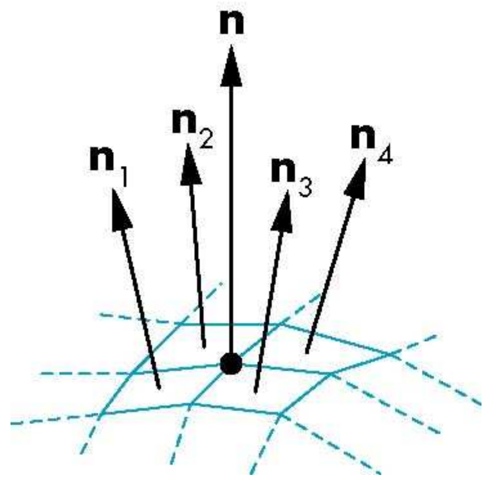


127

Mesh Shading

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

$$\mathbf{n} = (\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4)/\,|\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4|$$



Gouraud and Phong Shading

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

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
  - Until recently not available in real time systems
  - Now can be done using fragment shaders
  - Both need data structures to represent meshes so we can obtain vertex normals

DISPLAY LISTS IN OPENGL

- **A display list is a group of OpenGL commands that have been stored for later execution.**

- **Most OpenGL commands can be either stored in a display list or issued in immediate mode.**

For example, suppose you want to draw a circle with 100 line segments

```
drawCircle()
{  GLint i;
   GLfloat cosine, sine;
   glBegin(GL_POLYGON);
      for(i=0;i<100;i++){
         cosine=cos(i*2*PI/100.0);
         sine=sin(i*2*PI/100.0);
         glVertex2f(cosine,sine);
      }
   glEnd();
}
```

This method is terribly inefficient because the trigonometry has to be performed each time the circle is rendered. Save the coordinates in a table:

```
drawCircle()
{  GLint i;
   GLfloat cosine, sine;
   static GLfloat circoords[100][2];
   static GLint inited=0;
   if(inited==0){
      inited=1;
      for(i=0;i<100;i++){
         circcoords[i][0]=cos(i*2*PI/100.0);
         circcoords[i][1]=sin(i*2*PI/100.0);
      }
   }
   glBegin(GL_POLYGON);
      for(i=0;i<100;i++)
         glVertex2fv(&circcoords[i][0]);
   glEnd();
}
```

• **Draw the circle once and have OpenGL remember how to draw it for later use.**

```
#define MY_CIRCLE_LIST 1
 buildCircle() {
    GLint i;
    GLfloat cosine, sine;
      glNewList    (MY_CIRCLE_LIST, GL_COMPILE);
       glBegin(GL_POLYGON);
          for(i=0;i<100;i++){
            cosine=cos(i*2*PI/100.0);
            sine=sin(i*2*PI/100.0);
            glVertex2f(cosine,sine);
          }
       glEnd();
     glEndList    ();
 }
```

_MY_CIRCLE_LIST_ is an integer index that uniquely identifies this display list.

You can execute the display list later with this glCallList() command: glCallList(MY_CIRCLE_LIST);

- **A display list contains only OpenGL calls.**
- **The coordinates and other variables are evaluated and copied into the display list when the list is compiled.**
- **You can delete a display list and create a new one, but you can't edit an existing display list.**
- **Display lists reside with the server and network traffic is minimized. Matrix computations, lighting models, textures, etc.**
- **Display List disadvantages: large storage; immutability of the contents of a display list.**

Use a Display List: list.c

```
glNewList (listName, GL_COMPILE);
      glColor3f(1.0, 0.0, 0.0);
      glBegin (GL_TRIANGLES);
       glVertex2f(0.0,0.0);glVertex2f(1.0,0.0); glVertex2f (0.0, 1.0);
      glEnd ();
      glTranslatef (1.5, 0.0, 0.0);
  glEndList ();
  glShadeModel (GL_FLAT);

 void display(void)
 {   GLuint i;
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f(0.0, 1.0, 0.0);
    for (i = 0; i < 10; i++)
       glCallList (listName);
    drawLine (); /* color red; affected by the 10 translate */
    glFlush ();
 }
```

Constants are stored and won't change

```
GLfloat color_vector[3]={0.0,0.0,0.0};

glNewList(1,GL_COMPILE);
    glColor3fv(color_vector);
glEndList();
color_vector[0]=1.0; // color will be black if you use the display list
```

Use glPushAttrib() to save a group of state variables and glPopAttrib() to restore

```
glNewList(listIndex,GL_COMPILE);
   glPushMatrix();
   glPushAttrib(GL_CURRENT_BIT);
      glColor3f(1.0, 0.0, 0.0);
      glBegin(GL_POLYGON);
         glVertex2f(0.0,0.0);
         glVertex2f(1.0,0.0);
         glVertex2f(0.0,1.0);
      glEnd();
   glTranslatef(1.5,0.0,0.0);
   glPopAttrib();
   glPopMatrix();
glEndList();
```

The code below would draw a green, untranslated line.

```
void display(void)
{   GLint i;
   glClear (GL_COLOR_BUFFER_BIT); glColor3f(0.0, 1.0, 0.0);
   for (i = 0; i < 10; i++)  glCallList (listIndex);
   drawLine ();     glFlush ();
}
```

Hierarchical Display Lists

- You can create a hierarchical display list, a display list that executes another display list.
- Useful for an object that's made of components which are used more than once.

```
glNewList(listIndex,GL_COMPILE);
  glCallList(handlebars);
  glCallList(frame);
  glTranslatef(1.0,0.0,0.0);
  glCallList(wheel);
  glTranslatef(3.0,0.0,0.0);
  glCallList(wheel);
glEndList();
```

Editable Display Lists

- **Example editable display list: To render the polygon, call display list number 4. To edit a vertex, you need only recreate the single display list corresponding to that vertex.**

```
glNewList(1,GL_COMPILE);
   glVertex3f(v1);
glEndList();
glNewList(2,GL_COMPILE);
   glVertex3f(v2);
glEndList();
glNewList(3,GL_COMPILE);
   glVertex3f(v3);
glEndList();

glNewList(4,GL_COMPILE);
   glBegin(GL_POLYGON);
      glCallList(1); glCallList(2); glCallList(3);
   glEnd();
glEndList();
```

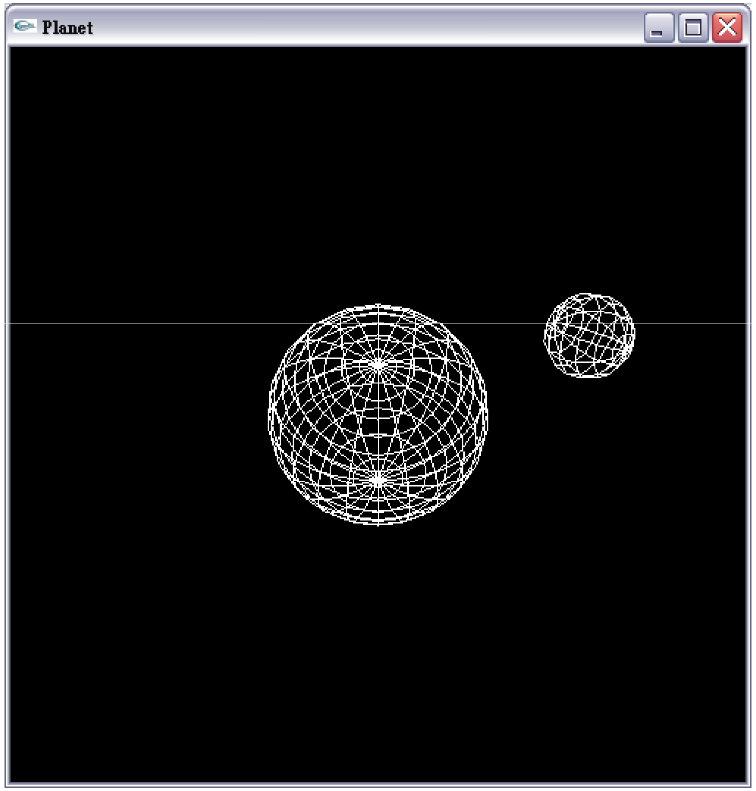
Managing Display List Indices

List Indices can be automatically generated:

```
listIndex=glGenLists(1);
if(listIndex!=0) {
   glNewList(listIndex,GL_COMPILE);
      ...
   glEndList();
}
```

Example -1

- planet.c
  - Control:
    - ‘d’
    - ‘y’
    - ‘a’
- ‘A’
    - ESC



Example -2

```
#include <GL/glut.h>
static GLfloat year=0.0f, day=0.0f;

void init()
{    glClearColor(0.0, 0.0, 0.0, 0.0);                    }

void GL_reshape(GLsizei w, GLsizei h)          // GLUT reshape function
{
     glViewport(0, 0, w, h);                   // viewport transformation
     glMatrixMode(GL_PROJECTION);              // projection transformation
     glLoadIdentity();
     gluPerspective(60.0, (GLfloat)w/(GLfloat)h, 1.0, 20.0);
     glMatrixMode(GL_MODELVIEW);               // viewing and modeling transformation
     glLoadIdentity();
     gluLookAt(0.0, 3.0, 5.0,         // eye
               0.0, 0.0, 0.0,         // center
               0.0, 1.0, 0.0);        // up
}
```

Example -3

```
void GL_display()        // GLUT display function
{
    // clear the buffer
    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f(1.0, 1.0, 1.0);
    glPushMatrix();
        glutWireSphere(1.0, 20, 16);      // the Sun
        glRotatef(year, 0.0, 1.0, 0.0);
        glTranslatef(3.0, 0.0, 0.0);
        glRotatef(day, 0.0, 1.0, 0.0);
        glutWireSphere(0.5, 10, 8);       // the Planet
    glPopMatrix();
    // swap the front and back buffers
    glutSwapBuffers();
}
```

Example -4

```
void GL_idle()            // GLUT idle function
{
        day += 10.0;
        if(day > 360.0)  day -= 360.0;

        year += 1.0;
        if(year > 360.0) year -= 360.0;

        // recall GL_display() function
        glutPostRedisplay();
}
```

Example -5

```
void GL_keyboard(unsigned char key, int x, int y)          // GLUT keyboard function
{
    switch(key)
    {
        case 'd':   day += 10.0;
                    if(day > 360.0) day -= 360.0;
                    glutPostRedisplay();
                    break;
        case 'y':   year += 1.0;
                    if(year > 360.0) year -= 360.0;
                    glutPostRedisplay();
                    break;
        case 'a':   glutIdleFunc(GL_idle); // assign idle function
                    break;
        case 'A':   glutIdleFunc(0);
                    break;
        case 27:    exit(0);
    }
}
```

Example -6

```
int main(int argc, char** argv)
{
      glutInit(&argc, argv);
      glutInitWindowSize(500, 500);
      glutInitWindowPosition(0, 0);
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
      glutCreateWindow("Planet");
      init();
      glutDisplayFunc(GL_display);
      glutReshapeFunc(GL_reshape);
      glutKeyboardFunc(GL_keyboard);
      glutMainLoop();
      return 0;
}
```

Reference$_{2/2}$

- Further Reading
  - OpenGL Programming Guide (Red Book)
  - Interactive Computer Graphics: A To-Down Approach Using OpenGL

Any Question?

?