

Interactive Computer Graphics

Notes on OpenGL

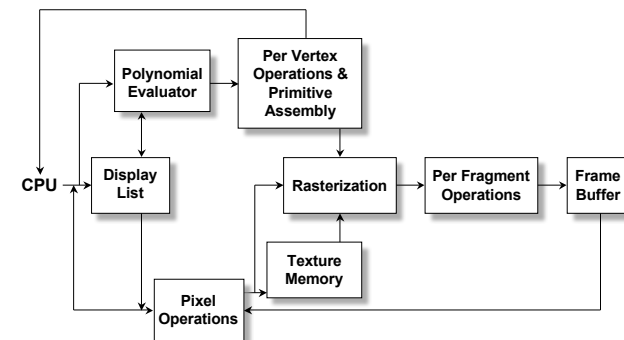
OpenGL and GLUT Overview

- What is OpenGL & what can it do for me?
- OpenGL in windowing systems
- Why GLUT
- A GLUT program template

What Is OpenGL?

- Graphics rendering API
 - high-quality color images composed of geometric and image primitives
 - window system independent
 - operating system independent

OpenGL Architecture



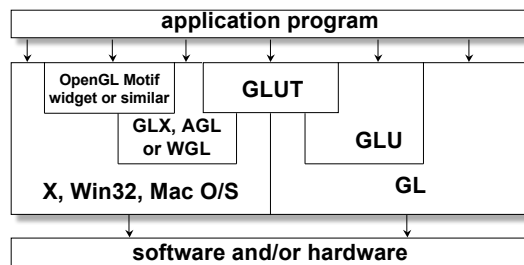
OpenGL as a Renderer

- Geometric primitives
 - points, lines and polygons
- Image Primitives
 - images and bitmaps
 - separate pipeline for images and geometry
 - linked through texture mapping
- Rendering depends on state
 - colors, materials, light sources, etc.

Related APIs

- AGL, GLX, WGL
 - glue between OpenGL and windowing systems
- GLU (OpenGL Utility Library)
 - part of OpenGL
 - NURBS, tessellators, quadric shapes, etc.
- GLUT (OpenGL Utility Toolkit)
 - portable windowing API
 - not officially part of OpenGL

OpenGL and Related APIs



Preliminaries

- Headers Files
 - `#include <GL/gl.h>`
 - `#include <GL/glu.h>`
 - `#include <GL/glut.h>`
- Libraries
- Enumerated Types
 - OpenGL defines numerous types for compatibility
 - GLfloat, GLint, GLenum, etc.

GLUT Basics

- Application Structure
 - Configure and open window
 - Initialize OpenGL state
 - Register input callback functions
 - render
 - resize
 - input: keyboard, mouse, etc.
 - Enter event processing loop

Sample Program

```
void main( int argc, char** argv )
{
    int mode = GLUT_RGB|GLUT_DOUBLE;
    glutInitDisplayMode( mode );
    glutCreateWindow( argv[0] );
    init();
    glutDisplayFunc( display );
    glutReshapeFunc( resize );
    glutKeyboardFunc( key );
    glutIdleFunc( idle );
    glutMainLoop();
}
```

OpenGL Initialization

- Set up whatever state you're going to use

```
void init( void )
{
    glClearColor( 0.0, 0.0, 0.0, 1.0 );
    glClearDepth( 1.0 );

    glEnable( GL_LIGHT0 );
    glEnable( GL_LIGHTING );
    glEnable( GL_DEPTH_TEST );
}
```

GLUT Callback Functions

- Routine to call when something happens
 - window resize or redraw
 - user input
 - animation
- “Register” callbacks with GLUT

```
glutDisplayFunc( display );
glutIdleFunc( idle );
glutKeyboardFunc( keyboard );
```

Rendering Callback

- Do all of your drawing here

```
glutDisplayFunc( display );

void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT );
    glBegin( GL_TRIANGLE_STRIP );
    glVertex3fv( v[0] );
    glVertex3fv( v[1] );
    glVertex3fv( v[2] );
    glVertex3fv( v[3] );
    glEnd();
    glutSwapBuffers();
}
```

Idle Callbacks

- Use for animation and continuous update

```
glutIdleFunc( idle );

void idle( void )
{
    t += dt;
    glutPostRedisplay();
}
```

User Input Callbacks

- Process user input

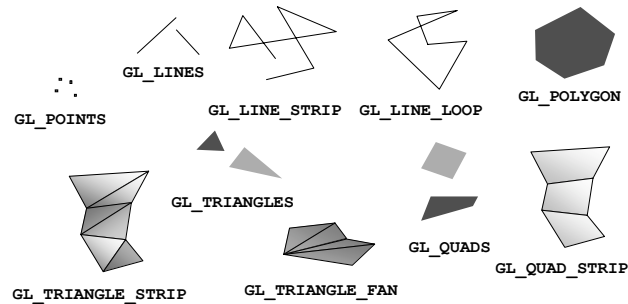
```
glutKeyboardFunc( keyboard );

void keyboard( char key, int x, int y )
{
    switch( key ) {
        case 'q' : case 'Q' :
            exit( EXIT_SUCCESS );
            break;
        case 'r' : case 'R' :
            rotate = GL_TRUE;
            break;
    }
}
```

Elementary Rendering

- Geometric Primitives
- Managing OpenGL State
- OpenGL Buffers

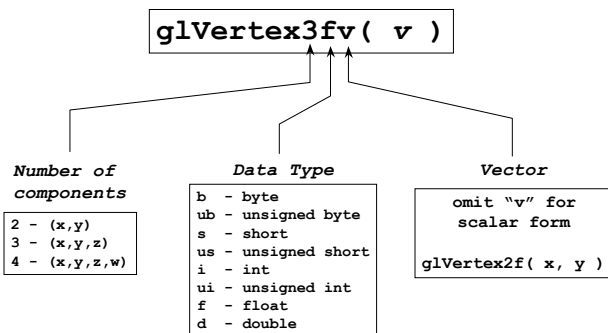
OpenGL Geometric Primitives



Simple Example

```
void drawRhombus( GLfloat color[] )
{
    glBegin( GL_QUADS );
    glColor3fv( color );
    glVertex2f( 0.0, 0.0 );
    glVertex2f( 1.0, 0.0 );
    glVertex2f( 1.5, 1.118 );
    glVertex2f( 0.5, 1.118 );
    glEnd();
}
```

OpenGL Command Formats



Specifying Geometric Primitives

- Primitives are specified using


```
glBegin( primType );
glEnd();
```

 - *primType* determines how vertices are combined

```
GLfloat red, green, blue;
GLfloat coords[3];
glBegin( primType );
for ( i = 0; i < nVerts; ++i ) {
    glColor3f( red, green, blue );
    glVertex3fv( coords );
}
glEnd();
```

OpenGL's State Machine

- All rendering attributes are encapsulated in the OpenGL State
 - rendering styles
 - shading
 - lighting
 - texture mapping

Manipulating OpenGL State

- Appearance is controlled by current state

```
for each ( primitive to render ) {  
    update OpenGL state  
    render primitive  
}
```

- Manipulating vertex attributes is most common way to manipulate state

```
glColor* ()  
glIndex* ()  
glNormal* ()  
glTexCoord* ()
```

Controlling current state

- Setting State

```
glPointSize( size );  
glLineStipple( repeat, pattern );  
glShadeModel( GL_SMOOTH );
```

- Enabling Features

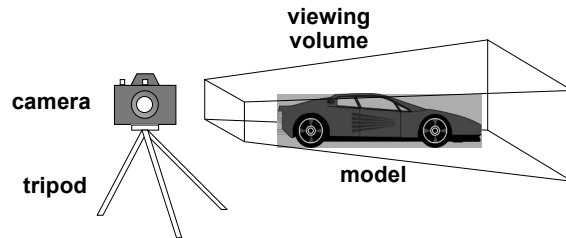
```
glEnable( GL_LIGHTING );  
glDisable( GL_TEXTURE_2D );
```

Transformations in OpenGL

- Modeling
- Viewing
 - orient camera
 - projection
- Animation
- Map to screen

Camera Analogy

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



Camera Analogy and Transformations

- Projection transformations
 - adjust the lens of the camera
- Viewing transformations
 - tripod—define position and orientation of the viewing volume in the world
- Modeling transformations
 - moving the model
- Viewport transformations
 - enlarge or reduce the physical photograph

Coordinate Systems and Transformations

- Steps in Forming an Image
 - specify geometry (world coordinates)
 - specify camera (camera coordinates)
 - project (window coordinates)
 - map to viewport (screen coordinates)
- Each step uses transformations
- Every transformation is equivalent to a change in coordinate systems (frames)

Affine Transformations

- Want transformations which preserve geometry
 - lines, polygons, quadrics
- Affine = line preserving
 - Rotation, translation, scaling
 - Projection
 - Concatenation (composition)

Homogeneous Coordinates

- each vertex is a column vector

$$\vec{v} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- w is usually 1.0
- all operations are matrix multiplications
- directions (directed line segments) can be represented with w = 0.0

3D Transformations

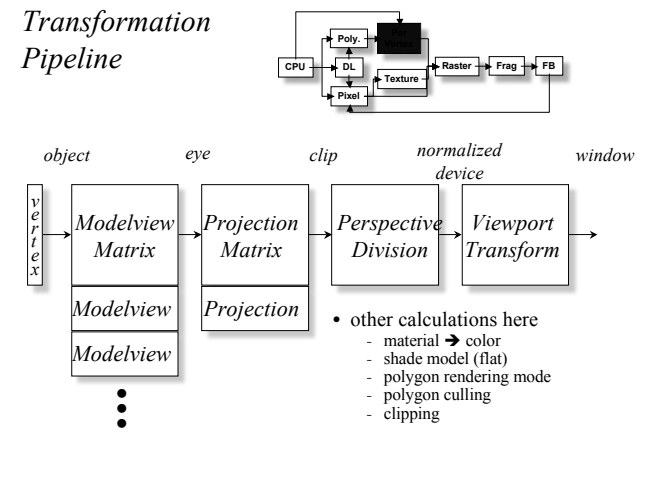
- A vertex is transformed by 4 x 4 matrices
 - all affine operations are matrix multiplications
 - all matrices are stored column-major in OpenGL
 - matrices are always post-multiplied
 - product of matrix and vector is $\mathbf{M}\vec{v}$

$$\mathbf{M} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

Specifying Transformations

- Programmer has two styles of specifying transformations
 - specify matrices (`glLoadMatrix`, `glMultMatrix`)
 - specify operation (`glRotate`, `glOrtho`)
- Prior to rendering, view, locate, and orient:
 - eye/camera position
 - 3D geometry
- Manage the matrices
 - including matrix stack
- Combine (composite) transformations

Transformation Pipeline

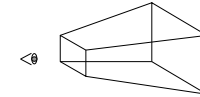


Matrix Operations

- Specify Current Matrix Stack
`glMatrixMode(GL_MODELVIEW or GL_PROJECTION)`
- Other Matrix or Stack Operations
`glLoadIdentity()`
`glPushMatrix()`
`glPopMatrix()`
- Viewport
 - usually same as window size
 - viewport aspect ratio should be same as projection transformation or resulting image may be distorted`glViewport(x, y, width, height)`

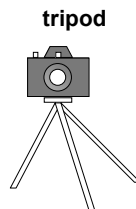
Projection Transformation

- Shape of viewing frustum
- Perspective projection
`gluPerspective(fovy, aspect, zNear, zFar)`
`glFrustum(left, right, bottom, top, zNear, zFar)`
- Orthographic parallel projection
`glOrtho(left, right, bottom, top, zNear, zFar)`
`gluOrtho2D(left, right, bottom, top)`
 - calls `glOrtho` with z values near zero
- Typical use (orthographic projection)
`glMatrixMode(GL_PROJECTION);`
`glLoadIdentity();`
`glOrtho(left, right, bottom, top, zNear, zFar);`



Viewing Transformations

- Position the camera/eye in the scene
 - place the tripod down; aim camera
- To “fly through” a scene
 - change viewing transformation and redraw scene
- `gluLookAt(eye_x, eye_y, eye_z, aim_x, aim_y, aim_z, up_x, up_y, up_z)`
 - up vector determines unique orientation
 - careful of degenerate positions

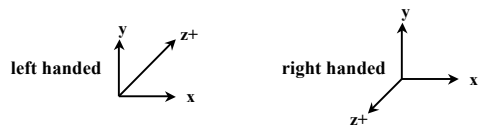


Modeling Transformations

- Moving camera is equivalent to moving every object in the world towards a stationary camera
- Move object
`glTranslate{fd}(x, y, z)`
- Rotate object around arbitrary axis
`glRotate{fd}(angle, x, y, z)`
 - angle is in degrees
- Dilate (stretch or shrink) or mirror object
`glScale{fd}(x, y, z)`

Projection is left handed

- Projection transformations (`gluPerspective`, `glOrtho`) are left handed
 - think of ***zNear*** and ***zFar*** as distance from view point
- Everything else is right handed, including the vertexes to be rendered



Common Transformation Usage

- 3 examples of `resize()` routine
 - restate projection & viewing transformations
- Usually called when window resized
- Registered as callback for `glutReshapeFunc()`

resize(): Perspective & LookAt

```
void resize( int w, int h )
{
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 65.0, (GLfloat) w / h,
                  1.0, 100.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    gluLookAt( 0.0, 0.0, 5.0,
              0.0, 0.0, 0.0,
              0.0, 1.0, 0.0 );
}
```

resize(): Perspective & Translate

- Same effect as previous LookAt

```
void resize( int w, int h )
{
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 65.0, (GLfloat) w/h,
                  1.0, 100.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    glTranslatef( 0.0, 0.0, -5.0 );
}
```

resize(): Ortho (part 1)

```
void resize( int width, int height )
{
    GLdouble aspect = (GLdouble) width / height;
    GLdouble left = -2.5, right = 2.5;
    GLdouble bottom = -2.5, top = 2.5;
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
```

... continued ...

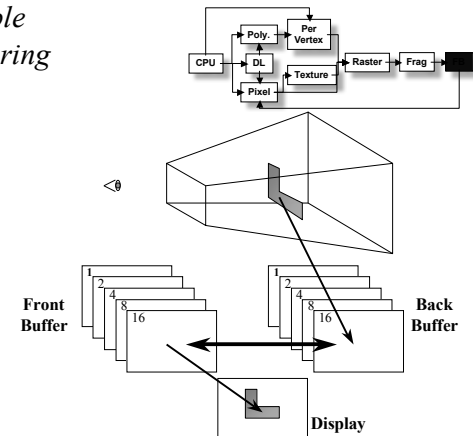
resize(): Ortho (part 2)

```
    if ( aspect < 1.0 ) {
        left /= aspect;
        right /= aspect;
    } else {
        bottom *= aspect;
        top *= aspect;
    }
    glOrtho( left, right, bottom, top, near, far );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
}
```

Compositing Modeling Transformations

- Problem 1: hierarchical objects
 - one position depends upon a previous position
 - robot arm or hand; sub-assemblies
- Solution 1: moving local coordinate system
 - modeling transformations move coordinate system
 - post-multiply column-major matrices
 - OpenGL post-multiplies matrices

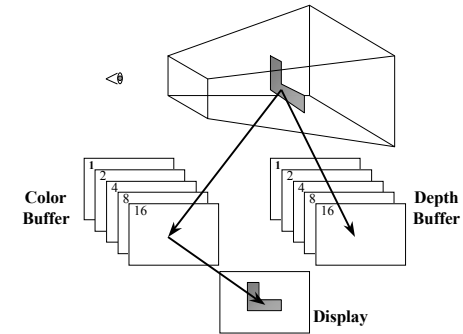
Double Buffering



Animation Using Double Buffering

- Request a double buffered color buffer
`glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);`
- Clear color buffer
`glClearColor(GL_COLOR_BUFFER_BIT);`
- Render scene
- Request swap of front and back buffers
`glutSwapBuffers();`
- Repeat steps 2 - 4 for animation

Depth Buffering and Hidden Surface Removal



Depth Buffering Using OpenGL

- Request a depth buffer
`glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);`
- Enable depth buffering
`glEnable(GL_DEPTH_TEST);`
- Clear color and depth buffers
`glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`
- Render scene
- Swap color buffers

An Updated Program Template

```
void main( int argc, char** argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_RGB |
        GLUT_DOUBLE | GLUT_DEPTH );
    glutCreateWindow( "Tetrahedron" );
    init();
    glutIdleFunc( idle );
    glutDisplayFunc( display );
    glutMainLoop();
}
```

An Updated Program Template (cont.)

```
void init( void )
{
    glClearColor( 0.0, 0.0, 1.0, 1.0 );
}

void idle( void )
{
    glutPostRedisplay();
}
```

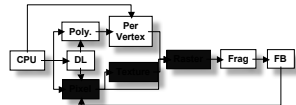
An Updated Program Template (cont.)

```
void drawScene( void )
{
    GLfloat vertices[] = { ... };
    GLfloat colors[] = { ... };
    glClear( GL_COLOR_BUFFER_BIT |
            GL_DEPTH_BUFFER_BIT );
    glBegin( GL_TRIANGLE_STRIP );

    /* calls to glColor*() and glVertex*() */

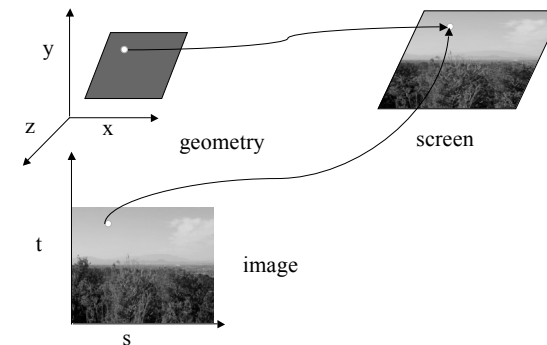
    glEnd();
    glutSwapBuffers();
}
```

Texture Mapping



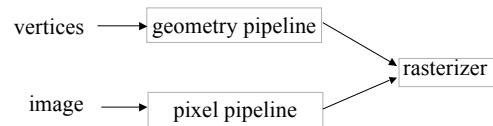
- Apply a 1D, 2D, or 3D image to geometric primitives
- Uses of Texturing
 - simulating materials
 - reducing geometric complexity
 - image warping
 - reflections

Texture Mapping



Texture Mapping and the OpenGL Pipeline

- Images and geometry flow through separate pipelines that join at the rasterizer
 - “complex” textures do not affect geometric complexity



Texture Example

- The texture (below) is a 256 x 256 image that has been mapped to a rectangular polygon which is viewed in perspective



Applying Textures I

- Three steps
 - ① specify texture
 - read or generate image
 - assign to texture
 - ② assign texture coordinates to vertices
 - ③ specify texture parameters
 - wrapping, filtering

Applying Textures II

- specify textures in texture objects
- set texture filter
- set texture function
- set texture wrap mode
- set optional perspective correction hint
- bind texture object
- enable texturing
- supply texture coordinates for vertex
 - coordinates can also be generated

Texture Objects

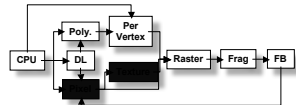
- Like display lists for texture images
 - one image per texture object
 - may be shared by several graphics contexts
- Generate texture names

```
glGenTextures( n, *texIds );
```

Texture Objects (cont.)

- Create texture objects with texture data and state
`glBindTexture(target, id);`
- Bind textures before using
`glBindTexture(target, id);`

Specify Texture Image

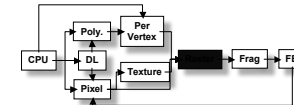


- Define a texture image from an array of texels in CPU memory

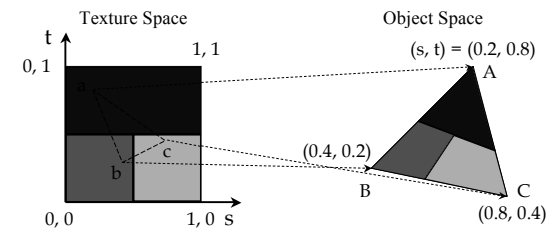
```
glTexImage2D( target, level, components,  
             w, h, border, format, type, *texels );
```

- dimensions of image must be powers of 2
- Texel colors are processed by pixel pipeline
 - pixel scales, biases and lookups can be done

Mapping a Texture



- Based on parametric texture coordinates
- `glTexCoord* ()` specified at each vertex



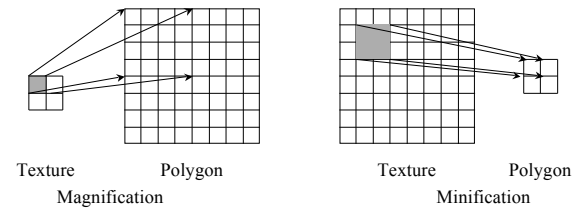
Texture Application Methods

- Filter Modes
 - minification or magnification
 - special mipmap minification filters
- Wrap Modes
 - clamping or repeating
- Texture Functions
 - how to mix primitive's color with texture's color
 - blend, modulate or replace texels

Filter Modes

Example:

```
glTexParameteri( target, type, mode );
```



Mipmapped Textures

- Mipmap allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects
- Declare mipmap level during texture definition

```
glTexImage*D( GL_TEXTURE_*D, level, ... )
```
- GLU mipmap builder routines

```
gluBuild*DMipmaps( ... )
```
- OpenGL 1.2 introduces advanced LOD controls

Wrapping Mode

• Example:

```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP )  
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT )
```



Texture Functions

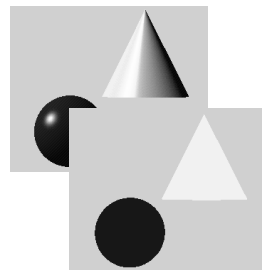
- Controls how texture is applied
`glTexEnv{fi}[v](GL_TEXTURE_ENV, prop, param)`
- `GL_TEXTURE_ENV_MODE` modes
 - `GL_MODULATE`
 - `GL_BLEND`
 - `GL_REPLACE`
- Set blend color with `GL_TEXTURE_ENV_COLOR`

Is There Room for a Texture?

- Query largest dimension of texture image
 - typically largest square texture
 - doesn't consider internal format size`glGetIntegerv(GL_MAX_TEXTURE_SIZE, &size)`
- Texture proxy
 - will memory accommodate requested texture size?
 - no image specified; placeholder
 - if texture won't fit, texture state variables set to 0
 - doesn't know about other textures
 - only considers whether this one texture will fit all of memory

Lighting Principles

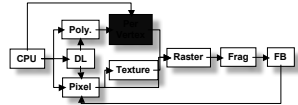
- Lighting simulates how objects reflect light
 - material composition of object
 - light's color and position
 - global lighting parameters
 - ambient light
 - two sided lighting
 - available in both color index and RGBA mode



How OpenGL Simulates Lights

- Phong lighting model
 - Computed at vertices
- Lighting contributors
 - Surface material properties
 - Light properties
 - Lighting model properties

Surface Normals



- Normals define how a surface reflects light

`glNormal3f(x, y, z)`

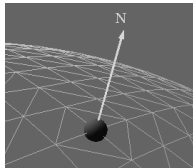
- Current normal is used to compute vertex's color

- Use *unit* normals for proper lighting

- scaling affects a normal's length

`glEnable(GL_NORMALIZE)`
or

`glEnable(GL_RESCALE_NORMAL)`



Material Properties

- Define the surface properties of a primitive
`glMaterialfv(face, property, value);`
- separate materials for front and back

<code>GL_DIFFUSE</code>	Base color
<code>GL_SPECULAR</code>	Highlight Color
<code>GL_AMBIENT</code>	Low-light Color
<code>GL_EMISSION</code>	Glow Color
<code>GL_SHININESS</code>	Surface Smoothness

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

Light Sources (cont.)

- Light color properties
 - `GL_AMBIENT`
 - `GL_DIFFUSE`
 - `GL_SPECULAR`

Types of Lights

- OpenGL supports two types of Lights
 - Local (Point) light sources
 - Infinite (Directional) light sources
- Type of light controlled by w coordinate

$w = 0$ Infinite Light directed along $(x \ y \ z)$

$w \neq 0$ Local Light positioned at $(\frac{x}{w} \ \frac{y}{w} \ \frac{z}{w})$

Turning on the Lights

- Flip each light's switch
`glEnable(GL_LIGHTn);`
- Turn on the power
`glEnable(GL_LIGHTING);`

Controlling a Light's Position

- Modelview matrix affects a light's position
 - Different effects based on when position is specified
 - eye coordinates
 - world coordinates
 - model coordinates
 - Push and pop matrices to uniquely control a light's position

Advanced Lighting Features

- Spotlights
 - localize lighting affects
 - `GL_SPOT_DIRECTION`
 - `GL_SPOT_CUTOFF`
 - `GL_SPOT_EXPONENT`

Advanced Lighting Features

- Light attenuation
 - decrease light intensity with distance
 - `GL_CONSTANT_ATTENUATION`
 - `GL_LINEAR_ATTENUATION`
 - `GL_QUADRATIC_ATTENUATION`

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

Light Model Properties

- `glLightModelfv(property, value);`
- Enabling two sided lighting
`GL_LIGHT_MODEL_TWO_SIDE`
- Global ambient color
`GL_LIGHT_MODEL_AMBIENT`
- Local viewer mode
`GL_LIGHT_MODEL_LOCAL_VIEWER`
- Separate specular color
`GL_LIGHT_MODEL_COLOR_CONTROL`

Tips for Better Lighting

- Recall lighting computed only at vertices
 - model tessellation heavily affects lighting results
 - better results but more geometry to process
- Use a single infinite light for fastest lighting
 - minimal computation per vertex

On-Line Resources

- <http://www.opengl.org>
 - start here; up to date specification and lots of sample code
- `news:comp.graphics.api.opengl`
- <http://www.sgi.com/software/opengl>
- <http://www.mesa3d.org/>
 - Brian Paul's Mesa 3D
- <http://www.cs.utah.edu/~narobins/opengl.html>
 - very special thanks to Nate Robins for the OpenGL Tutors
 - source code for tutors available here!

Books

- OpenGL Programming Guide, 3rd Edition
- OpenGL Reference Manual, 3rd Edition
- OpenGL Programming for the X Window System
 - includes many GLUT examples
- Interactive Computer Graphics: A top-down approach with OpenGL, 2nd Edition