SIGGRAPH2007

Course #10

An Interactive Introduction to OpenGL Programming

Ed Angel
Dave Shreiner
Vicki Shreiner

OpenGL is the most widely available graphics programming library, and is used for almost every discipline of computer graphics: research, scientific visualization, entertainment and visual effects, computer-aided design, interactive gaming, and many more. This course provides an accelerated introduction to creating applications using OpenGL.

The course takes a beginning OpenGL programmer from the basics of what's required for OpenGL operation, through geometric modeling and transformations, thoroughly discusses topics such as lighting, depth buffering, alpha blending, and texture mapping, and concludes with a brief introduction to advanced study topics such as the stencil and accumulation buffers, and display lists.

Course Prerequisites

Our only prerequisite is that students should be able to read simple computer programs written in the C computer language.

The course does present a few concepts from linear algebra (vector notation and matrix multiplication), but knowledge of those subjects is not required for the student to successfully understand the material. Additionally, although not a course on computer graphics, we generally do introduce and define most major concepts (e.g., depth buffering, simulated lighting, etc.)

Table of Contents

- 1 -

An Interactive Introduction to OpenGL Programming

SIGGRAPH2007

Ed Angel

Dave Shreiner

Vicki Shreiner

- 2 -

Welcome

- Today's Goals and Agenda
  - Describe OpenGL and its uses
  - Demonstrate and describe OpenGL's capabilities and features
  - Enable you to write an interactive, 3D computer graphics program in OpenGL

OpenGL

OpenGL makes creating interactive, 3D computer-graphics programs simple and fast. By the end of today's course, you'll have all the skills and techniques to make interactive games, drawing programs, or anything else you might imagine drawing on a computer screen.

- 3 -

Downloading Our Tutorials

http://www.opengl-redbook.com/s2007/

- Executables and Source Code available
  - Linux
  - Mac OS X
  - Microsoft Windows

OpenGL

   The programs we use in the course are freely available for your use and experimentation. The programs are available pre-compiled for the major operating systems, and we provide the source code as well, so you can see how OpenGL's used in practice.

What Is OpenGL, and What Can It Do for Me?

- OpenGL is a computer graphics *rendering* API
  - Generate high-quality color images by rendering with geometric and image primitives
  - Create interactive applications with 3D graphics
  - OpenGL is
    - operating system independent
    - window system independent

OpenGL is a library for doing computer graphics. By using it, you can create interactive applications that render high-quality color images composed of 3D geometric objects and images.

OpenGL is window and operating system independent. As such, the part of your application which does rendering is platform independent. However, in order for OpenGL to be able to render, it needs a window to draw into. Generally, this is controlled by the windowing system on whatever platform you are working on.

OpenGL and Its Related APIs
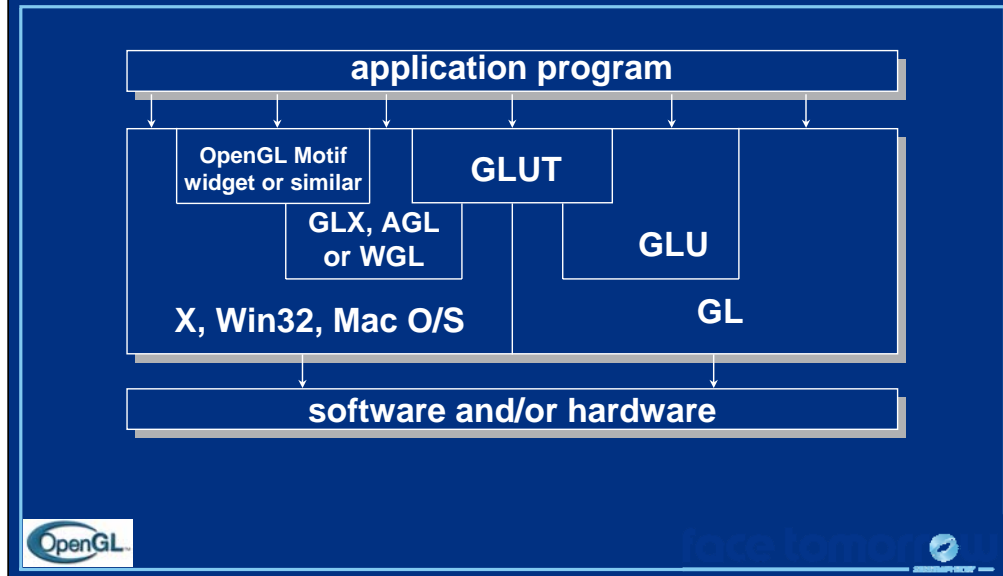
The above diagram illustrates the relationships of the various libraries and window system components.

Generally, applications which require more user interface support will use a library designed to support those types of features (i.e., buttons, menu and scroll bars, etc.) such as Motif or the Win32 API.

Prototype applications, or ones which do not require all the bells and whistles of a full GUI, may choose to use GLUT instead because of its simplified programming model and window system independence.

Related APIs

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

OpenGL.

As mentioned, OpenGL is window and operating system independent. To integrate it into various window systems, additional libraries are used to modify a native window into an OpenGL capable window. Every window system has its own unique library and functions to do this. Some examples are:

- GLX for the X Windows system, common on Unix platforms
- Cocoa, AGL, and CGL for the Apple Macintosh
- WGL for Microsoft Windows

OpenGL also includes a utility library, GLU, to simplify common tasks such as: rendering quadric surfaces (i.e., spheres, cones, cylinders, etc.), working with NURBS and curves, and concave polygon tessellation.

Finally to simplify programming and window system dependence, we will be using the freeware library, GLUT. GLUT, written by Mark Kilgard, is a public domain window system independent toolkit for making simple OpenGL applications. It simplifies the process of creating windows, working with events in the window system and handling animation.
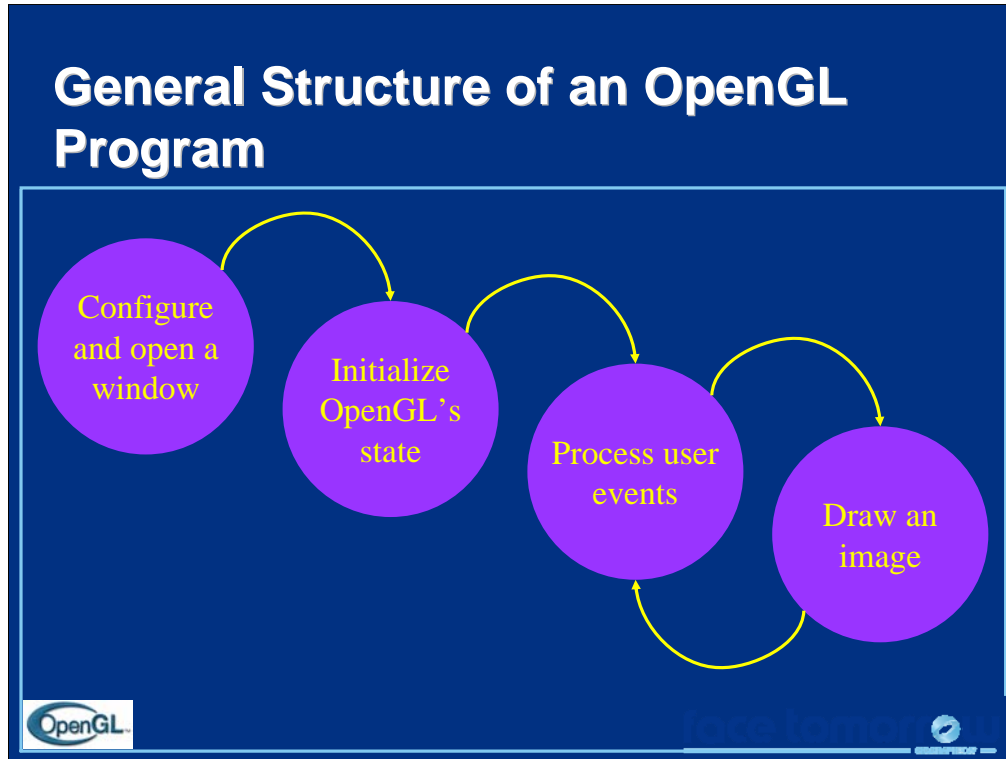
OpenGL was primarily designed to be able to draw high-quality images fast enough so that an application could draw many of them a second, and provide the user with an interactive application, where each *frame* could be customized by input from the user.

The general flow of an interactive application, including OpenGL applications is:

1. Configure and open a window suitable for drawing OpenGL into.

2. Initialize any OpenGL state that you will need to use throughout the application.

3. Process any events that the user might have entered. These could include pressing a key on the keyboard, moving the mouse, or even moving or resizing the application's window.

4. Draw your 3D image using OpenGL with values that may have been entered from the user's actions, or other data that the program has available to it.
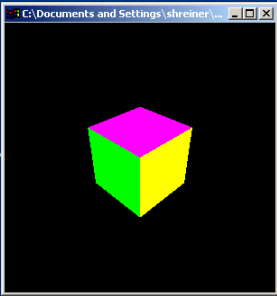
This slide contains the program statements for the `main()` routine of a C program that uses OpenGL and GLUT. For the most part, all of the programs you will see today, and indeed may of the programs available as examples of OpenGL programming that use GLUT will look very similar to this program.

All GLUT-based OpenGL programs begin with configuring the GLUT window that gets opened.

Next, in the routine `init()` (detailed on the following slide), "global" OpenGL state is configured. By "global", we mean state that will be left on for the duration of the application. By setting that state once, we can make our OpenGL applications run as efficiently as possible.

After initialization, we set up our GLUT *callback functions*, which are routines that you write to have OpenGL draw objects and other operations. Callback functions, if you're not familiar with them, make it easy to have a generic library (like GLUT), that can easily be configured by providing a few routines of your own construction.

Finally, as with all interactive programs, the event loop is entered. For GLUT-based programs, this is done by calling `glutMainLoop()`. As `glutMainLop()` never exits (it is essentially an infinite loop), any program statements that follow `glutMainLoop()` will never be executed.

An OpenGL Program (cont'd.)

```
void init( void )
{
  glClearColor( 0, 0, 0, 1 );
  gluLookAt( 2, 2, 2, 0, 0, 0, 0, 1, 0 );
  glEnable( GL_DEPTH_TEST );
}
```

Set up some initial OpenGL *state*

```
void reshape( int width, int height )
{
  glViewport( 0, 0, width, height );
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  gluPerspective( 60, (GLfloat) width / height,
                  1.0, 10.0 );
  glMatrixMode( GL_MODELVIEW );
}
```

Handle when the user resizes the window

OpenGL

First on this slide is the `init()` routine, which as mentioned, is where we set up the "global" OpenGL state. In this case, `init()` sets the color that the background of the window should be painted to when the window is cleared, as well as configuring where the eye should be located and enabling the depth test. Although you may not know what these mean at the moment, we will discuss each of those topics. What is important to notice is that what we set in `init()` remains in affect for the rest of the program's execution. There is nothing that says we can not turn these features off later; the separation of these routines in this manner is purely for clarity in the program's structure.

The `reshape()` routine is called when the user of a program resizes the application's window. We do a number of things in this routine, all of which will be explained in detail in the *Transformations* section later today.

An OpenGL Program (cont'd.)

```
void display( void )
{
  int   i, j;

  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

  glBegin( GL_QUADS );
  for ( i = 0; i < NUM_CUBE_FACES; ++i ) {
    glColor3fv( faceColor[i] );
    for ( j = 0; j < NUM_VERTICES_PER_FACE; ++j ) {
      glVertex3fv( vertex[face[i][j]] );
    }
  }
  glEnd();

  glFlush();
}
```

Have OpenGL draw a cube from some 3D points *(vertices)*

OpenGL

Finally, we see the `display()` routine which is used by GLUT to call our OpenGL calls to make our image. Almost all of your OpenGL drawing code should be called from `display()` (or routines that `display()` calls).

As with most `display()`-like functions, a number of common things occur in the following order:

1. The window is cleared with a call to `glClear()`. This will color all of the pixels in the window with the color set with `glClearColor()` (see the previous slide and look in the `init()` routine). Any image that was in the window is overwritten.

2. Next, we do all of our OpenGL rendering. In this case, we draw a cube, setting the color of each face with a call to `glColor3fv()`, and specify where the *vertices* of the cube should be positioned by calling `glVertex3fv()`.

3. Finally, when all of the OpenGL rendering is completed, we either call `glFlush()` or `glutSwapBuffers()` to "swap the buffers," which will be discussed in the *Animation and Depth Buffering* section.

OpenGL Command Formats

```
glVertex3fv( v )
```

Number of components	Data Type	Vector
2 - (x,y) 3 - (x,y,z) 4 - (x,y,z,w)	b - byte ub - unsigned byte s - short us - unsigned short i - int ui - unsigned int f - float d - double	omit "v" for scalar form glVertex2f(x, y)
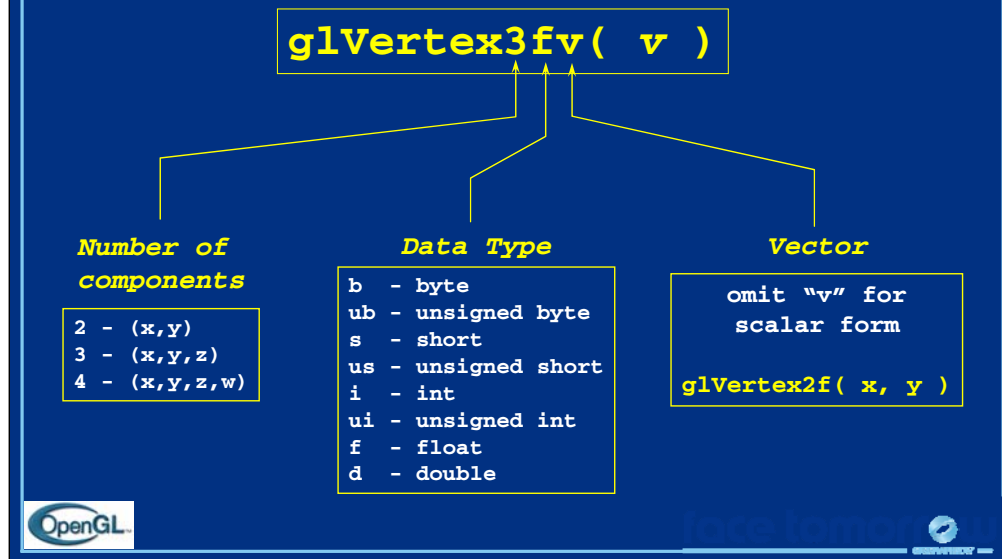
The OpenGL API calls are designed to accept almost any basic data type, which is reflected in the calls name. Knowing how the calls are structured makes it easy to determine which call should be used for a particular data format and size.

For instance, vertices from most commercial models are stored as three component floating point vectors. As such, the appropriate OpenGL command to use is glVertex3fv(coords).

As mentioned before, OpenGL uses homogenous coordinates to specify vertices. For glVertex*() calls which do not specify all the coordinates (i.e., glVertex2f()), OpenGL will default $z = 0.0$, and $w = 1.0$.

What's Required in Your Programs

- 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.

All of our discussions today will be presented in the C computer language.

For C, there are a few required elements which an application must do:

- *Header files* describe all of the function calls, their parameters and defined constant values to the compiler. OpenGL has header files for GL (the core library), GLU (the utility library), and GLUT (freeware windowing toolkit).

Note: glut.h includes gl.h and glu.h. On Microsoft Windows, including *only* glut.h is recommended to avoid warnings about redefining Windows macros.

- *Libraries* are the operating system dependent implementation of OpenGL on the system you are using. Each operating system has its own set of libraries. For Unix systems, the OpenGL library is commonly named libGL.so (which is usually specified as -lGL on the compile line) and for Microsoft Windows, it is named opengl32.lib.

- Finally, *enumerated types* are definitions for the basic types (i.e., float, double, int, etc.) which your program uses to store variables. To simplify platform independence for OpenGL programs, a complete set of enumerated types are defined. Use them to simplify transferring your programs to other operating systems.

GLUT Basics

- Application Structure:
  - Configure and open window
  - Initialize OpenGL state
  - Register input callback functions
  - Enter event processing loop

Here is the basic structure that we will be using in our applications. This is generally what you would do in your own OpenGL applications.

The steps are:

1. Choose the type of window that you need for your application and initialize it.

2. Initialize any OpenGL state that you do not need to change every frame of your program. This might include things like the background color, light positions and texture maps.

3. Register the *callback* functions that you will need. Callbacks are routines you write that GLUT calls when a certain sequence of events occurs, like the window needing to be refreshed, or the user moving the mouse. The most important callback function is the one to render your scene, which we will discuss in a few slides.

4. Enter the main event processing loop. This is where your application receives events, and schedules when callback functions are called.

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 );
```

GLUT uses a *callback mechanism* to do its event processing. Callbacks simplify event processing for the application developer. As compared to more traditional event driven programming, where the author must receive and process each event, and call whatever actions are necessary, callbacks simplify the process by defining what actions are supported, and automatically handling the user events. All the author must do is fill in what should happen when.

GLUT supports many different callback actions, including:

- `glutDisplayFunc()` - called when pixels in the window need to be refreshed.
- `glutReshapeFunc()` - called when the window changes size
- `glutKeyboardFunc()` - called when a key is struck on the keyboard
- `glutMouseFunc()` - called when the user presses a mouse button on the mouse
- `glutMotionFunc()` - called when the user moves the mouse while a mouse button is pressed
- `glutPassiveMouseFunc()` - called when the mouse is moved regardless of mouse button state
- `glutIdleFunc()` - a callback function called when nothing else is going on. Very useful for animations.

What can OpenGL Draw?

- 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.

As mentioned, OpenGL is a library for rendering computer graphics. Generally, there are two operations that you do with OpenGL:

- draw something
- change the state of how OpenGL draws

OpenGL has two types of things that it can render: geometric primitives and image primitives. *Geometric primitives* are points, lines and polygons. *Image primitives* are bitmaps and graphics images (i.e., the pixels that you might extract from a JPEG image after you have read it into your program.) Additionally, OpenGL links image and geometric primitives together using *texture mapping*, which is an advanced topic we will discuss this afternoon.

The other common operation that you do with OpenGL is *setting state*. "Setting state" is the process of initializing the internal data that OpenGL uses to render your primitives. It can be as simple as setting up the size of points and the color that you want a vertex to be, to initializing multiple mipmap levels for texture mapping.

Every OpenGL geometric primitive is specified by its vertices, which are *homogenous coordinates*. Homogenous coordinates are of the form (*x, y, z, w*). Depending on how vertices are organized, OpenGL can render any of the shown primitives.

Specifying Geometric Primitives

- Primitives are specified using

  glBegin(*primType*);

  glEnd();

  - *primType* determines how vertices are combined

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

OpenGL organizes vertices into primitives based upon which type is passed into `glBegin()`. The possible types are:

GL_POINTS	GL_LINE_STRIP
GL_LINES	GL_LINE_LOOP
GL_POLYGON	GL_TRIANGLE_STRIP
GL_TRIANGLES	GL_TRIANGLE_FAN
GL_QUADS	GL_QUAD_STRIP

We also see an example of setting OpenGL's state, which is the topic of the next few slides, and most of the course. In this case, the color that our primitive is going to be drawn is set using the `glColor()` call.
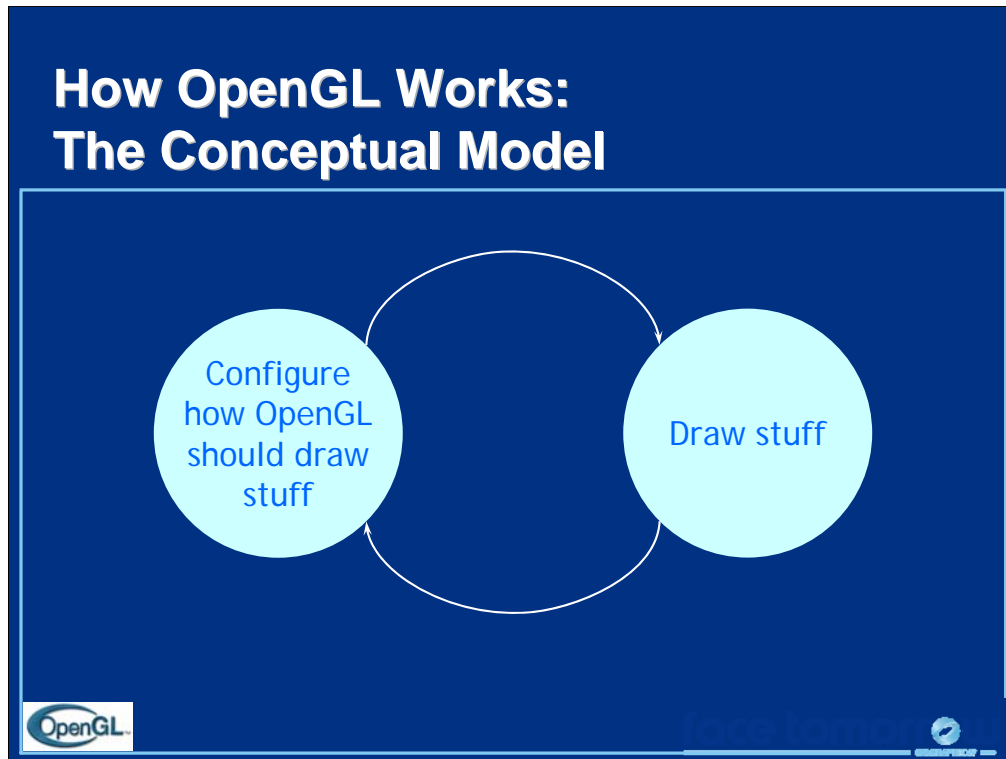
Conceptually, OpenGL allows you, the application designer, to do two things:

1. Control how the next items you draw will be processed. This is done by setting the OpenGL's state. OpenGL's state includes the current drawing color, parameters that control the color and location of lights, texture maps, and many other configurable settings.

2. Draw, or using the technical term, *render* graphical objects called primitives.

Your application will consist of cycles of setting state, and rendering using the state that you just set.

Controlling OpenGL's Drawing

- **Set OpenGL's rendering state**
  - State controls how things are drawn
    - shading
    - texture maps
    - polygon patterns
    - lighting
    - line styles (*stipples*)
    - transparency

Most of programming OpenGL is controlling its internal configuration, called *state*. State is just the set of values that OpenGL uses when it draws something. For example, if you wanted to draw a blue triangle, you would first tell OpenGL to set the current vertex color to blue, using the `glColor()` function. Then you pass the geometry to draw the triangle using the `glVertex()` calls you just saw.

OpenGL has over 400 function calls in it, most of which are concerned with setting the rendering state. Among the things that state controls are:

- current rendering color
- parameters used for simulating lighting
- processing data to be used as texture maps
- patterns (called *stipples*, in OpenGL) for lines and polygons
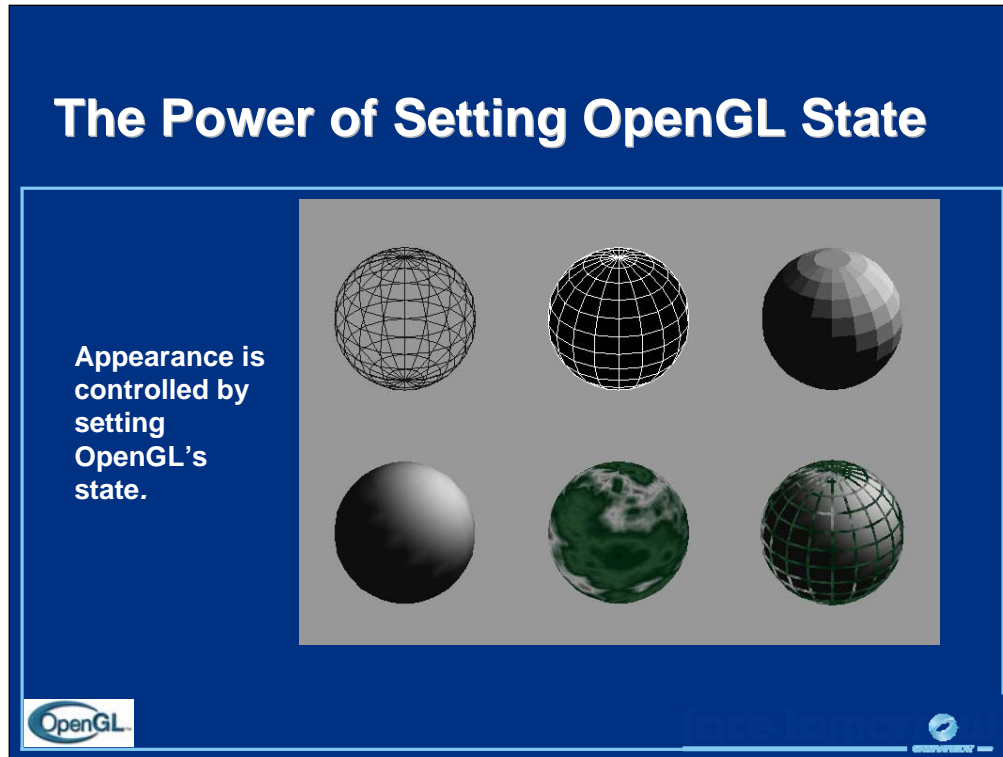
By only changing different parts of OpenGL's state, the same geometry (in the case of the image in the slide, a sphere) can be used to generate drastically different images.

Going across the top row, the first sphere is merely a wire-frame rendering of the sphere. The middle image was made by drawing the sphere twice, once solid in black, and a second time as a white wire-frame sphere over the solid black one. The right-most image shows a *flat-shaded* sphere, under the influence of OpenGL lighting. Flat-shading means that each geometric primitive has the same color.

For the bottom row, the first image is the same sphere, only this time, *gouraud- (or smooth-) shaded*. The only difference in the programs between the top-row right, and bottom-row left is a single line of OpenGL code. The middle sphere was generated using texture mapping. The final image is the smooth-shaded sphere, with texture-mapped lines over the solid sphere.

Setting OpenGL State

- Three ways to set OpenGL state:

    1. Set values to be used for processing vertices

        - most common methods of setting state
            - **glColor()** / **glIndex()**
            - **glNormal()**
            - **glTexCoord()**
        - state must be set before calling **glVertex()**

The most common state setting operation is that of modifying attributes associated with vertices. While we'll discuss setting vertex colors, lighting normals, and texture coordinates, that's only a small part–but the most common set– of the state associated with vertices.

Setting OpenGL State (cont'd.)

2. Turning on a rendering mode

glEnable() / glDisable()

3. Configuring the specifics of a particular rendering mode

- Each mode has unique commands for setting its values

glMaterialfv()

There are two actions that are required to control how OpenGL renders.

1.The first is turning on or off a rendering feature. This is done using the OpenGL calls glEnable() and glDisable(). When glEnable() is called for a particular feature, all OpenGL rendering after that point in the program will use that feature until it is turned off with glDisable().

2.Almost all OpenGL features have configurable values that you can set. Whether it is the color of the next thing you draw, or specifying an image that OpenGL should use as a texture map, there will be some calls unique to that feature that control all of its state. Most of the OpenGL API, and most of what you will see today, is concerned with setting the state of the individual features.

Every OpenGL feature has a default set of values so that even without setting any state, you can still have OpenGL render things. The initial state is pretty boring; it renders most things in white.

It's important to note that initial state is identical for every OpenGL implementation, regardless of which operating system, or which hardware system you are working on.

OpenGL and Color

- The OpenGL Color Model
  - OpenGL uses the *RGB* color space
    - There is also a color-index mode, but we do not discuss it
- Colors are specified as floating-point numbers in the range [0.0, 1.0]
  - for example, to set a window's background color, you would call
    ```
    glClearColor( 1.0, 0.3, 0.6, 1.0 );
    ```

Since computer graphics are all about color, it is important to know how to specify colors when using OpenGL. Conceptually, OpenGL uses the *RGB* (red, green, and blue) color space. Each of the three colors is a *component* of the color. The value of each color component is a real (floating-point) number between 0.0 and 1.0. Values outside of that range are clamped.

As an example, the call to set a window's background color in OpenGL is glClearColor(), as demonstrated on the slide. The colors specified for the background color are (1.0, 0.3, 0.6), for red, green, and blue, respectively. The fourth value in glClearColor() is named *alpha* and is discussed later in the course. Generally, when you call glClearColor(), you want to set the alpha component to 1.0.

OpenGL also supports color-index mode rendering, but as RGB based rendering is the most common, and there are some features that require RGB (most notably, texture mapping), we do not discuss color-index mode rendering in the scope of this class.
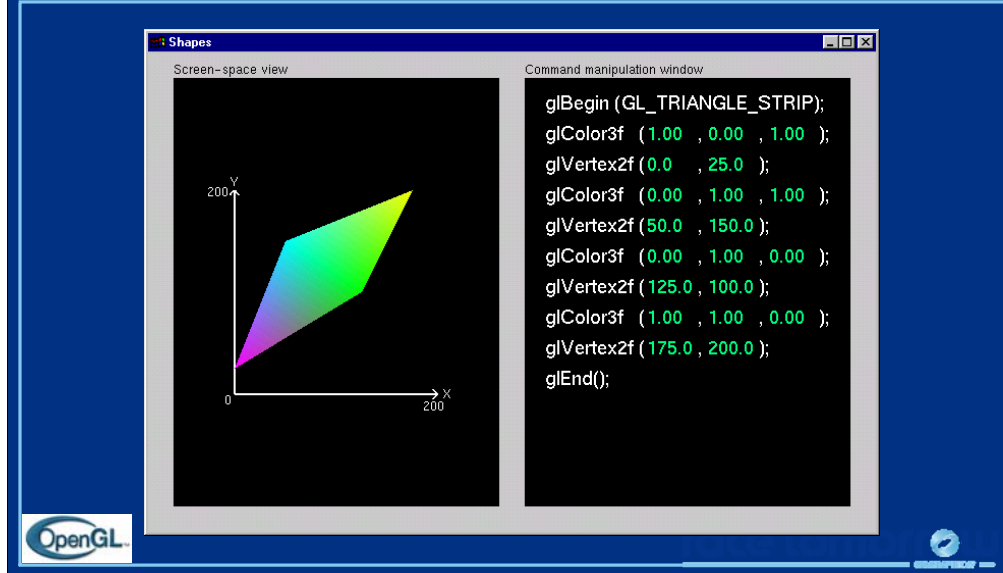
This is the first of the series of Nate Robins' tutorials. This tutorial illustrates the principles of rendering geometry, specifying both colors and vertices.

The shapes tutorial has two views: a screen-space window and a command manipulation window.

In the command manipulation window, pressing the LEFT mouse while the pointer is over the green parameter numbers allows you to move the mouse in the y-direction (up and down) and change their values. With this action, you can change the appearance of the geometric primitive in the other window. With the RIGHT mouse button, you can bring up a pop-up menu to change the primitive you are rendering. (Note that the parameters have minimum and maximum values in the tutorials, sometimes to prevent you from wandering too far. In an application, you probably do not want to have floating-point color values less than 0.0 or greater than 1.0, but you are likely to want to position vertices at coordinates outside the boundaries of this tutorial.)

In the screen-space window, the RIGHT mouse button brings up a different pop-up menu, which has menu choices to change the appearance of the geometry in different ways.

The left and right mouse buttons will do similar operations in the other tutorials.
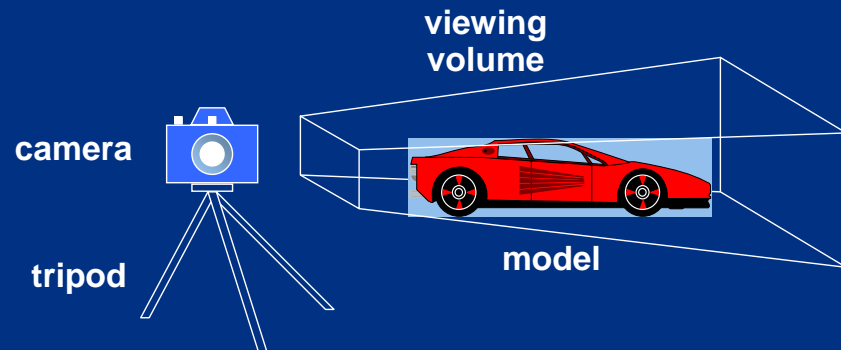
This model has become know as the synthetic camera model.

Note that both the objects to be viewed and the camera are three-dimensional while the resulting image is two dimensional.

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

Note that human vision and a camera lens have cone-shaped viewing volumes. OpenGL (and almost all computer graphics APIs) describe a pyramid-shaped viewing volume. Therefore, the computer will "see" differently from the natural viewpoints, especially along the edges of viewing volumes. This is particularly pronounced for wide-angle "fish-eye" camera lenses.

Coordinate Systems and Transformations

- Steps in forming an image
  1. specify geometry (world coordinates)
  2. specify camera (camera coordinates)
  3. project (window coordinates)
  4. map to viewport (screen coordinates)
- Each step uses transformations
- Every transformation is equivalent to a change in coordinate systems (frames)

OpenGL

Every transformation can be thought of as changing the representation of a vertex from one coordinate system or frame to another. Thus, initially vertices are specified in world or application coordinates. However, to view them, OpenGL must convert these representations to ones in the reference system of the camera. This change of representations is described by a transformation matrix (the model-view matrix). Similarly, the projection matrix converts from camera coordinates to window coordinates.

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 $w = 0.0$

A 3D vertex is represented by a 4-tuple vector (homogeneous coordinate system).

Why is a 4-tuple vector used for a 3D *(x, y, z)* vertex? To ensure that all matrix operations are multiplications.

If w is changed from 1.0, we can recover x, y and z by division by w. Generally, only perspective transformations change w and require this perspective division in the pipeline.

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
    - this is opposite of what "C" programmers expect
  - matrices are always post-multiplied
  - product of matrix and vector is $\mathbf{M}\vec{v}$

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

By using 4×4 matrices, OpenGL can represent all geometric transformations using one matrix format. Perspective projections and translations require the 4th row and column. Otherwise, these operations would require an vector-addition operation, in addition to the matrix multiplication.

While OpenGL specifies matrices in column-major order, this is often confusing for "C" programmers who are used to row-major ordering for two-dimensional arrays. OpenGL provides routines for loading both column- and row-major matrices. However, for standard OpenGL transformations, there are functions that automatically generate the matrices for you, so you don't generally need to be concerned about this until you start doing more advanced operations.

For operations other than perspective projection, the fourth row is always (0, 0, 0, 1) which leaves the w-coordinate unchanged..

Because OpenGL only multiplies a matrix on the right, the programmer should remember that the last matrix specified is the first applied.

Specifying Transformations

- Programmer has two styles of specifying transformations
  - specify matrices (**glLoadMatrix, glMultMatrix**)
  - specify operation (**glRotate, glOrtho**)
- Programmer does not have to remember the exact matrices
  - see appendix of the OpenGL Programming Guide

OpenGL

Generally, a programmer can obtain the desired matrix by a sequence of simple transformations that can be concatenated together, e.g., glRotate(), glTranslate(), and glScale().

For the basic viewing transformations, OpenGL and the Utility library have supporting functions.

Programming Transformations

- Prior to rendering, view, locate, and orient:
  - eye/camera position
  - 3D geometry
- Manage the matrices
  - including matrix stack
- Combine (composite) transformations

OpenGL

Because transformation matrices are part of the state, they must be defined prior to any vertices to which they are to apply.

In modeling, we often have objects specified in their own coordinate systems and must use OpenGL transformations to bring the objects into the scene.

OpenGL provides matrix stacks for each type of supported matrix (model-view, projection, texture) to store matrices.
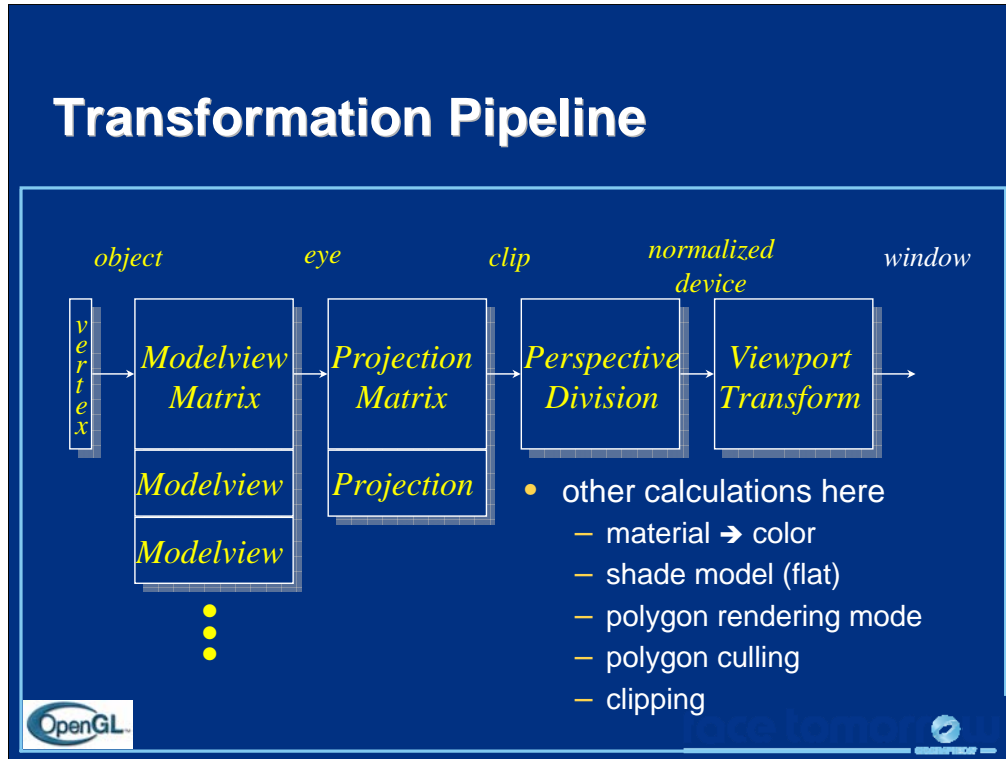
The depth of matrix stacks are implementation-dependent, but the Modelview matrix stack is guaranteed to be at least 32 matrices deep, and the Projection matrix stack is guaranteed to be at least 2 matrices deep.

The material-to-color, flat-shading, and clipping calculations take place after the Modelview matrix calculations, but before the Projection matrix. The polygon culling and rendering mode operations take place after the Viewport operations.

There is also a texture matrix stack, which is outside the scope of this course. It is an advanced texture mapping topic.

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*)

glLoadMatrix*() replaces the matrix on the top of the current matrix stack. glMultMatrix*(), post-multiples the matrix on the top of the current matrix stack. The matrix argument is a column-major 4 x 4 double or single precision floating point matrix.

Matrix stacks are used because it is more efficient to save and restore matrices than to calculate and multiply new matrices. Popping a matrix stack can be said to "jump back" to a previous location or orientation.

glViewport() clips the vertex or raster position. For geometric primitives, a new vertex may be created. For raster primitives, the raster position is completely clipped.

There is a per-fragment operation, the scissor test, which works in situations where viewport clipping does not. The scissor operation is particularly good for fine clipping raster (bitmap or image) primitives.

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

OpenGL
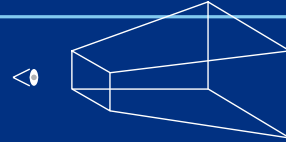
For perspective projections, the viewing volume is shaped like a truncated pyramid (frustum). There is a distinct camera (eye) position, and vertexes of objects are "projected" to camera. Objects which are further from the camera appear smaller. The default camera position at (0, 0, 0), looks down the *z*-axis, although the camera can be moved by other transformations.

For `gluPerspective()`, `fovy` is the angle of field of view (in degrees) in the y direction. `fovy` must be between 0.0 and 180.0, exclusive. `aspect` is *x/y* and should be the same as the viewport to avoid distortion. `zNear` and `zFar` define the distance to the near and far clipping planes.

The `glFrustum()` call is rarely used in practice.

Warning: for `gluPerspective()` or `glFrustum()`, do not use zero for `zNear`!

For `glOrtho()`, the viewing volume is shaped like a rectangular parallelepiped (a box). Vertices of an object are "projected" towards infinity, and as such, distance does not change the apparent size of an object, as happens under perspective projection. Orthographic projection is used for drafting, and design (such as blueprints).

Applying Projection Transformations

- Typical use (orthographic projection)

```
glMatrixMode( GL_PROJECTION );

glLoadIdentity();

glOrtho(left, right, bottom, top, zNear, zFar);
```
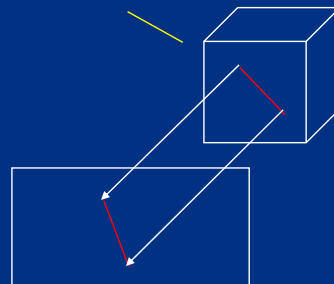
Many users would follow the demonstrated sequence of commands with a `glMatrixMode(GL_MODELVIEW)` call to return to modelview stack.

In this example, the red line segment is inside the view volume and is projected (with parallel projectors) to the green line on the view surface. The blue line segment lies outside the volume specified by `glOrtho()` and is clipped.

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

tripod

$$\text{gluLookAt(} eye_x, \ eye_y, \ eye_z,$$
$$aim_x, \ aim_y, \ aim_z,$$
$$up_x, \ up_y, \ up_z \text{)}$$

  - up vector determines unique orientation
  - careful of degenerate positions

OpenGL
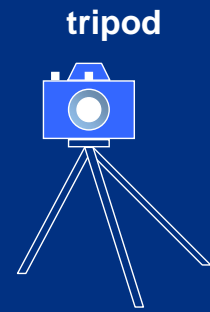
gluLookAt() multiplies itself onto the current matrix, so it usually comes after glMatrixMode(GL_MODELVIEW) and glLoadIdentity().

Because of degenerate positions, gluLookAt() is not recommended for most animated fly-over applications.

An alternative is to specify a sequence of rotations and translations that are concatenated with an initial identity matrix.

Note: that the name modelview matrix is appropriate since moving objects in the model front of the camera is equivalent to moving the camera to view a set of objects.

The RIGHT mouse button controls different menus. The screen-space view menu allows you to choose different models. The command-manipulation menu allows you to select different projection commands (including glOrtho and glFrustum).

Modeling Transformations

- Move object

  glTranslate{fd}(*x, y, z*)

- Rotate object around arbitrary axis $\begin{pmatrix} x & y & z \end{pmatrix}$

  glRotate{fd}(*angle, x, y, z*)

  – angle is in degrees

- Dilate (stretch or shrink) or mirror object

  glScale{fd}(*x, y, z*)

OpenGL

glTranslate(), glRotate(), and glScale() multiplies itself onto the current matrix, so it usually comes after glMatrixMode(GL_MODELVIEW). There are many situations where the modeling transformation is multiplied onto a non-identity matrix.

A vertex's distance from the origin changes the effect of glRotate() or glScale(). These operations have a fixed point for the origin. Generally, the further from the origin, the more pronounced the effect. To rotate (or scale) with a different fixed point, we must first translate, then rotate (or scale) and then undo the translation with another translation.
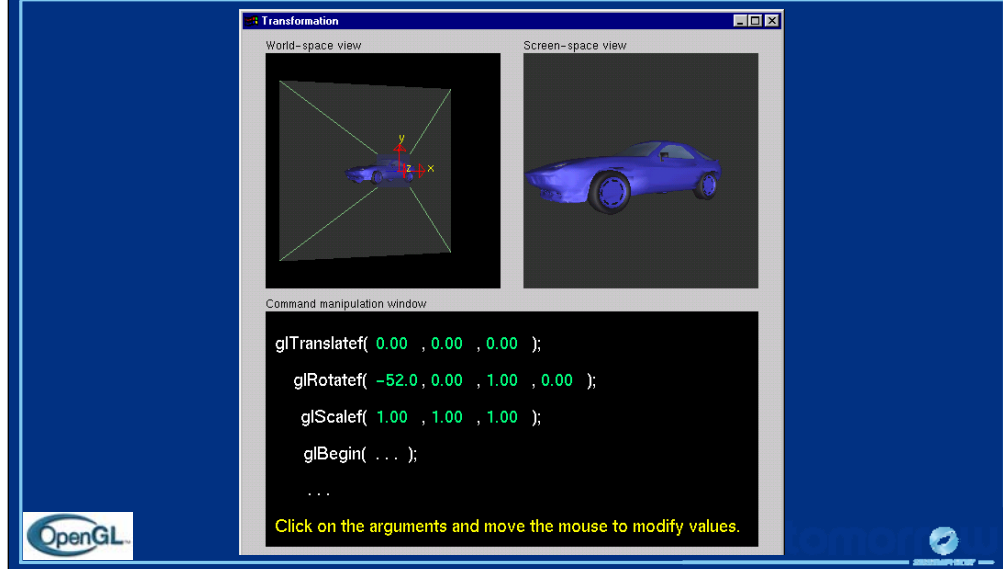
For right now, concentrate on changing the effect of one command at a time. After each time that you change one command, you may want to reset the values before continuing on to the next command.

The RIGHT mouse button controls different menus. The screen-space view menu allows you to choose different models. The command-manipulation menu allows you to change the order of the `glTranslatef()` and `glRotatef()` commands. Later, we will see the effect of changing the order of modeling commands.

Connection: Viewing and Modeling

- Moving camera is equivalent to moving every object in the world towards a stationary camera

- Viewing transformations are equivalent to several modeling transformations
  - **gluLookAt()** has its own command
  - can make your own *polar view* or *pilot view*

Instead of gluLookAt(), one can use the following combinations of glTranslate() and glRotate() to achieve a viewing transformation. Like gluLookAt(), these transformations should be multiplied onto the ModelView matrix, which should have an initial identity matrix.

To create a viewing transformation in which the viewer orbits an object, use this sequence (which is known as "polar view"):

```
glTranslated(0, 0, -distance)
glRotated(-twist, 0, 0, 1)
glRotated(-incidence, 1, 0, 0)
glRotated(azimuth, 0, 0, 1)
```

To create a viewing transformation which orients the viewer (roll, pitch, and heading) at position (*x, y, z*), use this sequence (known as "pilot view"):

```
glRotated(roll, 0, 0, 1)
glRotated(pitch, 0, 1, 0)
glRotated(heading, 1, 0, 0)
glTranslated(-x, -y, -z)
```

Compositing Modeling Transformations

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

OpenGL

- 42 -

The order in which modeling transformations are performed is important because each modeling transformation is represented by a matrix, and matrix multiplication is not commutative. So a rotate followed by a translate is different from a translate followed by a rotate.

Compositing Modeling Transformations (cont'd.)

- Problem: objects move relative to absolute world origin
  - my object rotates around the wrong origin
    - make it spin around its center or something else
- Solution: fixed coordinate system
  - modeling transformations move objects around fixed coordinate system
  - pre-multiply column-major matrices
  - OpenGL post-multiplies matrices
  - must <u>reverse order of operations</u> to achieve desired effect

OpenGL

You will adjust to reading a lot of code backwards!

Typical sequence

```
glTranslatef(x,y,z);
glRotatef(theta, ax, ay, az);
glTranslatef(-x,-y,-z);
object();
```

Here (*x, y, z*) is the fixed point. We first (last transformation in code) move it to the origin. Then we rotate about the axis (*ax, ay, az*) and finally move fixed point back.

Additional Clipping Planes

- At least 6 more clipping planes available

- Good for cross-sections

- Modelview matrix moves clipping plane

- $Ax + By + Cz + D < 0$ clipped

`glEnable( GL_CLIP_PLANEi )`

`glClipPlane( GL_CLIP_PLANEi, GLdouble* coeff )`

OpenGL.

Additional clipping planes, usually called *user-clip planes*, are very useful for "cutting away" part of a 3D model to allow a cross section view.

The clipping planes you define using `glClipPlane()` are described using the equation of a plane, with the *(A, B, C)* coefficients describing the orientation (think of a plane normal), and *D* representing the distance from the origin.

When you specify a clipping plane, the plane coefficients you provide are transformed by the current modelview matrix. This enables you to transform the plane using the standard modelview matrix stack operations, as compared to doing a bunch of vector math to transform the clipping plane itself.

Double Buffering

Front Buffer / Back Buffer / Display
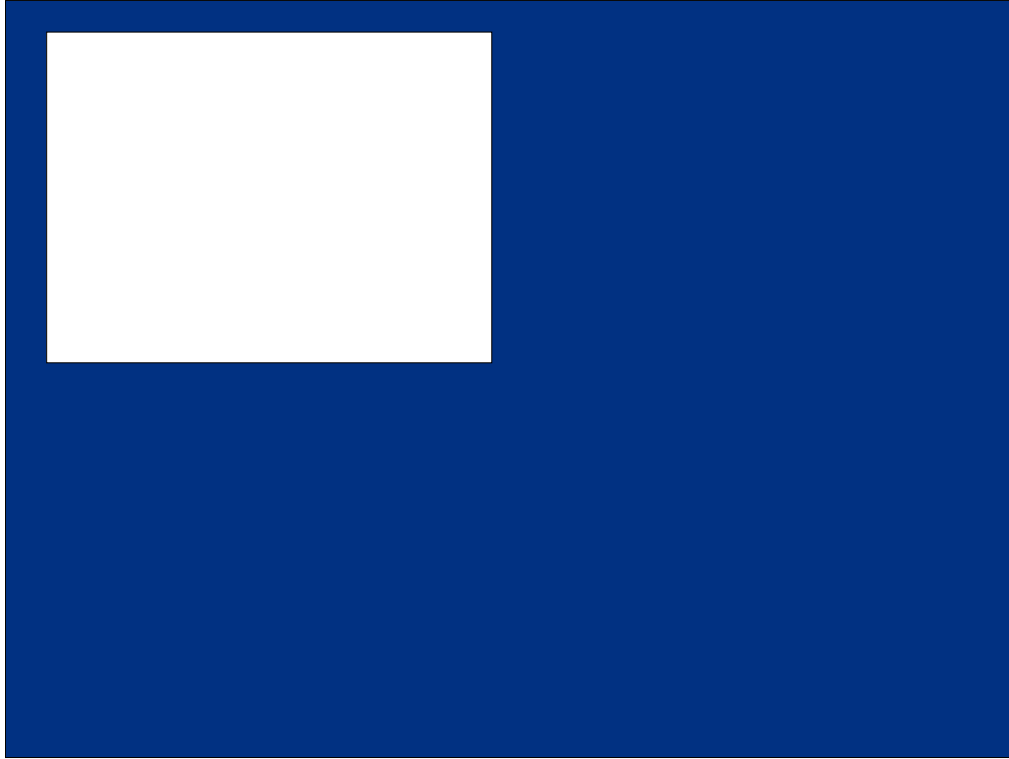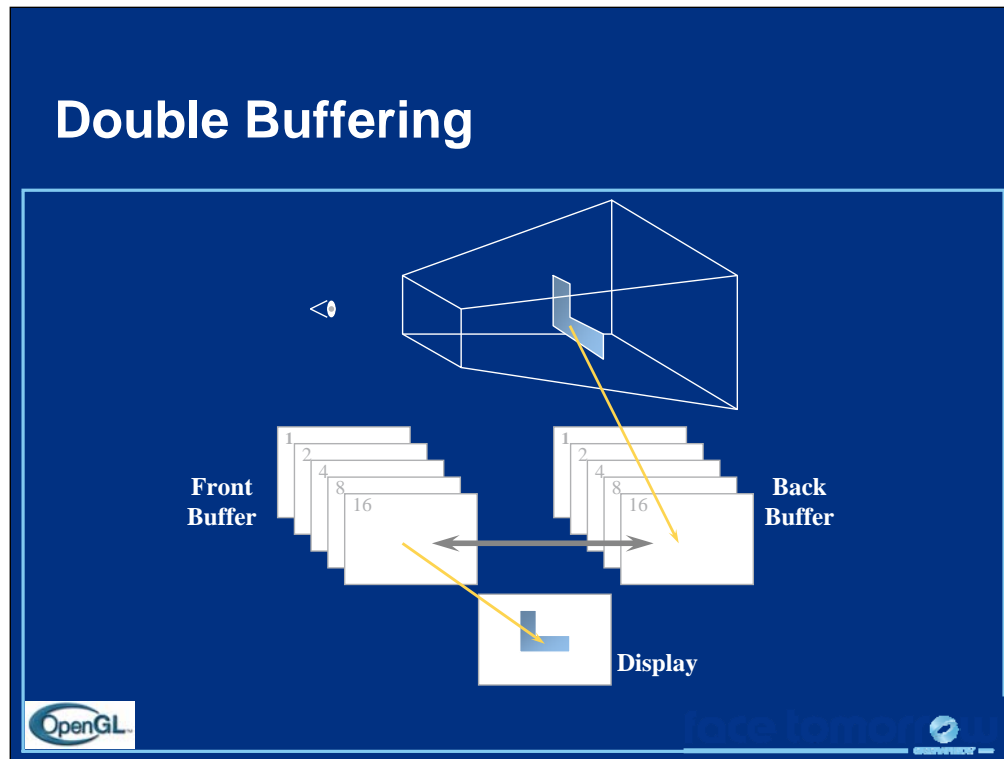
Double buffer is a technique for tricking the eye into seeing smooth animation of rendered scenes. The color buffer is usually divided into two equal halves, called the *front buffer* and the *back buffer.*

The front buffer is displayed while the application renders into the back buffer. When the application completes rendering to the back buffer, it requests the graphics display hardware to swap the roles of the buffers, causing the back buffer to now be displayed, and the previous front buffer to become the new back buffer.

Animation Using Double Buffering

1. Request a double buffered color buffer
    ```
    glutInitDisplayMode( GLUT_RGB | GLUT_DOUBLE );
    ```
2. Clear color buffer
    ```
    glClear( GL_COLOR_BUFFER_BIT );
    ```
3. Render scene
4. Request swap of front and back buffers
    ```
    glutSwapBuffers();
    ```
•   Repeat steps 2 - 4 for animation
    – Use a `glutIdleFunc()` callback

Requesting double buffering in GLUT is simple. Adding GLUT_DOUBLE to your glutInitDisplayMode() call will cause your window to be double buffered.

When your application is finished rendering its current frame, and wants to swap the front and back buffers, the glutSwapBuffers() call will request the windowing system to update the window's color buffers.
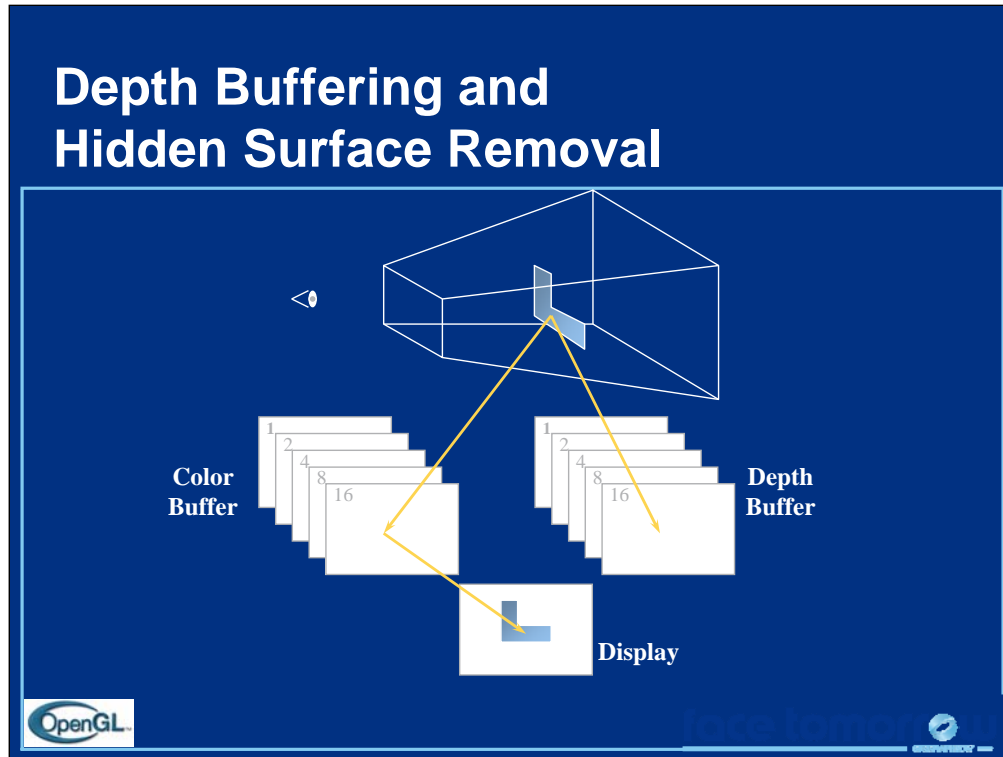
Depth buffering is a technique to determine which primitives in your scene are occluded by other primitives. As each pixel in a primitive is rasterized, its distance from the eyepoint (depth value), is compared with the values stored in the depth buffer. If the pixel's depth value is less than the stored value, the pixel's depth value is written to the depth buffer, and its color is written to the color buffer.

The depth buffer algorithm is:

```
if ( pixel->z < depthBuffer(x,y)->z ) {
    depthBuffer(x,y)->z = pixel->z;
    colorBuffer(x,y)->color = pixel->color;
}
```

OpenGL depth values range from [0.0, 1.0], with 1.0 being essentially infinitely far from the eyepoint. Generally, the depth buffer is cleared to 1.0 at the start of a frame.

Depth Buffering Using OpenGL

1. Request a depth buffer

    `glutInitDisplayMode( GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH );`

2. Enable depth buffering

    `glEnable( GL_DEPTH_TEST );`

3. Clear color and depth buffers

    `glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );`

4. Render scene

5. Swap color buffers

OpenGL

Enabling depth testing in OpenGL is very straightforward.

A depth buffer must be requested for your window, once again using the glutInitDisplayMode(), and the GLUT_DEPTH bit.

Once the window is created, the depth test is enabled using glEnable(GL_DEPTH_TEST).

- 50 -
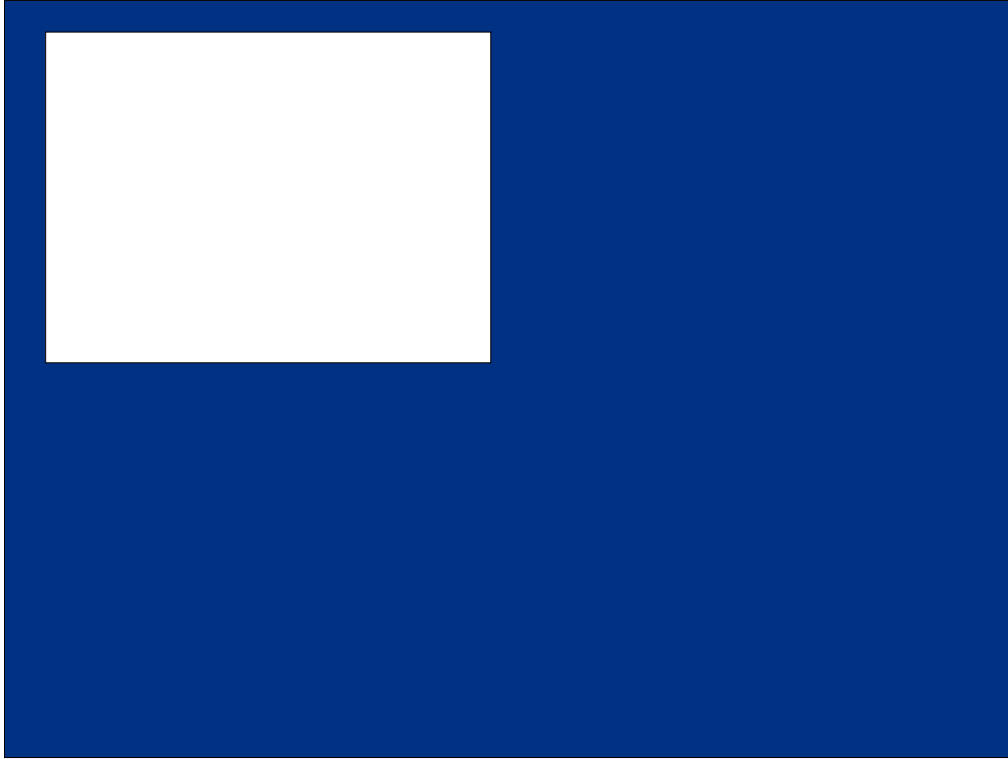
Lighting Principles

- 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
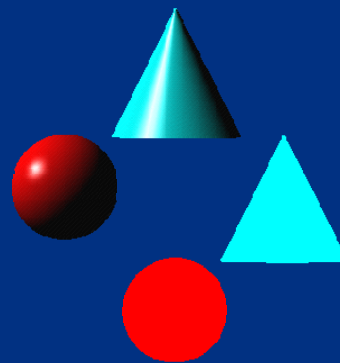
Lighting is an important technique in computer graphics. Without lighting, objects tend to look like they are made out of plastic.

OpenGL divides lighting into three parts: material properties, light properties and global lighting parameters.

Lighting is available in both RGBA mode and color index mode. RGBA is more flexible and less restrictive than color index mode lighting.

OpenGL Shading

- OpenGL computes a color or shade for each vertex using a lighting model (the modified Phong model) that takes into account
  - Diffuse reflections
  - Specular reflections
  - Ambient light
  - Emision
- Vertex shades are interpolated across polygons by the rasterizer

OpenGL can use the shade at one vertex to shade an entire polygon (constant shading) or interpolated the shades at the vertices across the polygon (smooth shading), the default.

The Modified Phong Model

- The model is a balance between simple computation and physical realism
- The model uses
  - Light positions and intensities
  - Surface orientation (normals)
  - Material properties (reflectivity)
  - Viewer location
- Computed for each source and each color component

OpenGL

The orientation of a surface is specified by the normal at each point. For a flat polygon the normal is constant over the polygon. Because normals are specified by the application program and can be changed between the specification of vertices, when we shade a polygon it can appear to be curved.

How OpenGL Simulates Lights

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

OpenGL lighting is based on the Phong lighting model. At each vertex in the primitive, a color is computed using that primitives material properties along with the light settings.

The color for the vertex is computed by adding four computed colors for the final vertex color. The four contributors to the vertex color are:

- *Ambient* is color of the object from all the undirected light in a scene.

- *Diffuse* is the base color of the object under current lighting. There must be a light shining on the object to get a diffuse contribution.

- *Specular* is the contribution of the shiny highlights on the object.

- *Emission* is the contribution added in if the object emits light (i.e., glows)

The lighting normal tells OpenGL how the object reflects light around a vertex. If you imagine that there is a small mirror at the vertex, the lighting normal describes how the mirror is oriented, and consequently how light is reflected.

`glNormal*()` sets the current normal, which is used in the lighting computation for all vertices until a new normal is provided.

Lighting normals should be normalized to unit length for correct lighting results. `glScale*()` affects normals as well as vertices, which can change the normal's length, and cause it to no longer be normalized. OpenGL can automatically normalize normals, by enabling `glEnable(GL_NORMALIZE)`. or `glEnable(GL_RESCALE_NORMAL)`. `GL_RESCALE_NORMAL` is a special mode for when your normals are uniformly scaled. If not, use `GL_NORMALIZE` which handles all normalization situations, but requires the computation of a square root, which can potentially lower performance

OpenGL evaluators and NURBS can provide lighting normals for generated vertices automatically.

Material Properties

- Define the surface properties of a primitive

  glMaterialfv(*face, property, value*);

GL_DIFFUSE	Base color
GL_SPECULAR	Highlight Color
GL_AMBIENT	Low-light Color
GL_EMISSION	Glow Color
GL_SHININESS	Surface Smoothness

  – separate materials for front and back

OpenGL

Material properties describe the color and surface properties of a material (dull, shiny, etc.). OpenGL supports material properties for both the front and back of objects, as described by their vertex winding.

The OpenGL material properties are:

- GL_DIFFUSE - base color of object
- GL_SPECULAR - color of highlights on object
- GL_AMBIENT - color of object when not directly illuminated
- GL_EMISSION - color emitted from the object (think of a firefly)
- GL_SHININESS - concentration of highlights on objects. Values range from 0 (very rough surface - no highlight) to 128 (very shiny)

Material properties can be set for each face separately by specifying either GL_FRONT or GL_BACK, or for both faces simultaneously using GL_FRONT_AND_BACK.

Light Sources

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

The glLight() call is used to set the parameters for a light. OpenGL implementations must support at least eight lights, which are named GL_LIGHT0 through GL_LIGHT*n*, where *n* is one less than the maximum number supported by an implementation.

OpenGL lights have a number of characteristics which can be changed from their default values. Color properties allow separate interactions with the different material properties. Position properties control the location and type of the light and attenuation controls the natural tendency of light to decay over distance.

Light Sources (cont'd.)

- Light color properties
  - **GL_AMBIENT**
  - **GL_DIFFUSE**
  - **GL_SPECULAR**

OpenGL lights can emit different colors for each of a materials properties. For example, a light's GL_AMBIENT color is combined with a material's GL_AMBIENT color to produce the ambient contribution to the color - Likewise for the diffuse and specular colors.

Types of Lights

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

$$w = 0 \quad \textbf{\textit{Infinite Light directed along}} \; (x \quad y \quad z)$$

$$w \neq 0 \quad \textbf{\textit{Local Light positioned at}} \; \left(\tfrac{x}{w} \quad \tfrac{y}{w} \quad \tfrac{z}{w} \right)$$

OpenGL

OpenGL supports two types of lights: infinite (directional) and local (point) light sources. The type of light is determined by the *w* coordinate of the light's position.

$$\text{if} \begin{cases} w = 0 & \text{define an infinite light at } (x \quad y \quad z) \\ w \neq 0 & \text{define a local light at } \left(\tfrac{x}{w} \quad \tfrac{y}{w} \quad \tfrac{z}{w} \right) \end{cases}$$

A light's position is transformed by the current ModelView matrix when it is specified. As such, you can achieve different effects by when you specify the position.

Turning on the Lights

- Flip each light's switch

  `glEnable( GL_LIGHTn );`

- Turn on the power

  `glEnable( GL_LIGHTING );`

Each OpenGL light is controllable separately, using `glEnable()` and the respective light constant `GL_LIGHTn`. Additionally, global control over whether lighting will be used to compute primitive colors is controlled by passing `GL_LIGHTING` to `glEnable()`. This provides a handy way to enable and disable lighting without turning on or off all of the separate components.
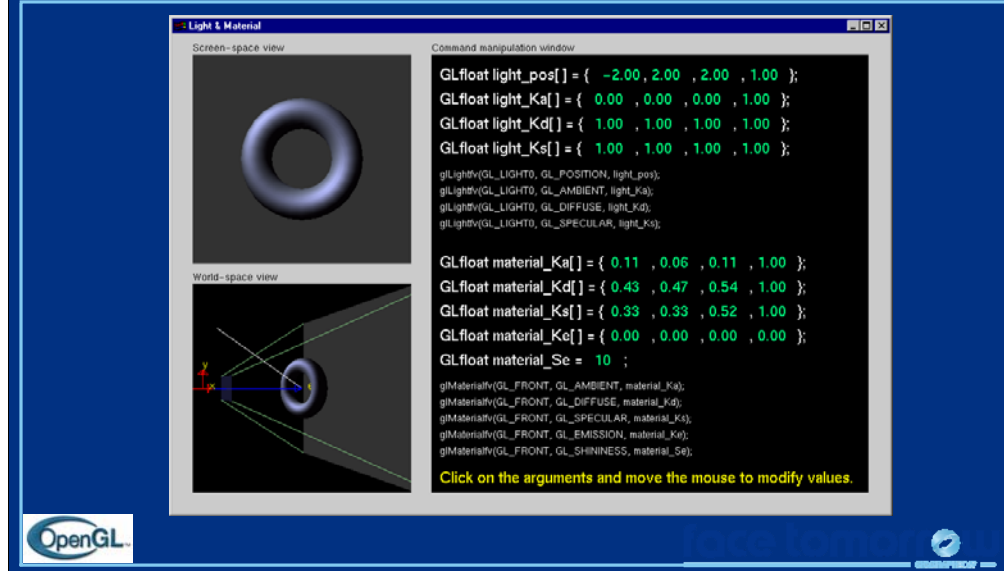
In this tutorial, concentrate on noticing the affects of different material and light properties. Additionally, compare the results of using a local light versus using an infinite light.

In particular, experiment with the GL_SHININESS parameter to see its affects on highlights.

Controlling a Light's Position

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

OpenGL

As mentioned previously, a light's position is transformed by the current ModelView matrix when it is specified. As such, depending on when you specify the light's position, and what values are in the ModelView matrix, you can obtain different lighting effects.

In general, there are three coordinate systems where you can specify a light's position/direction

1) *Eye coordinates* - which is represented by an identity matrix in the ModelView. In this case, when the light's position/direction is specified, it remains fixed to the imaging plane. As such, regardless of how the objects are manipulated, the highlights remain in the same location relative to the eye.

2) *World Coordinates* - when only the viewing transformation is in the ModelView matrix. In this case, a light's position/direction appears fixed in the scene, as if the light were on a lamppost.

3) *Model Coordinates* - any combination of viewing and modeling transformations is in the ModelView matrix. This method allows arbitrary, and even animated, position of a light using modeling transformations.
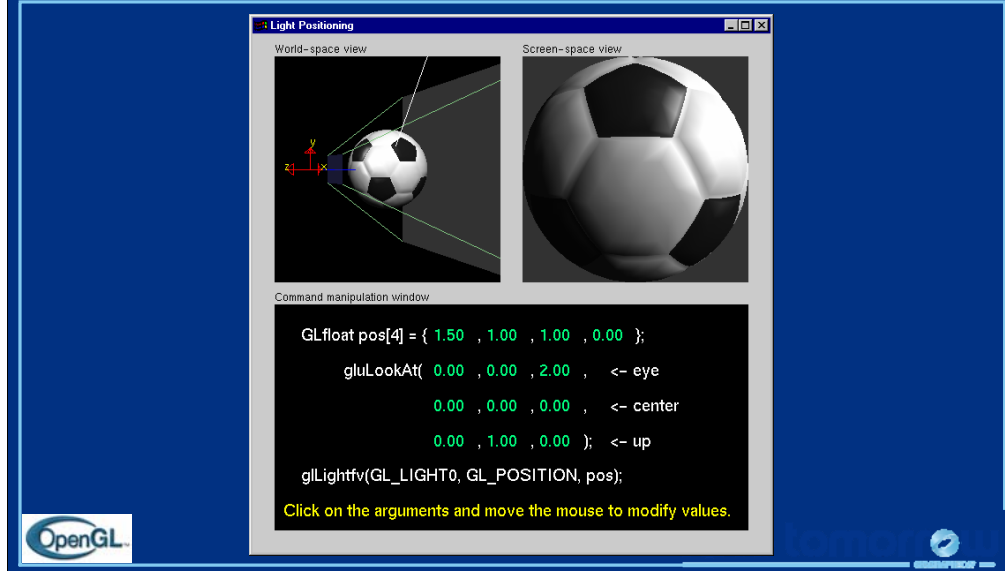
This tutorial demonstrates the different lighting affects of specifying a light's position in eye and world coordinates. Experiment with how highlights and illuminated areas change under the different lighting position specifications.

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

OpenGL

As with all of computing, time versus space is the continual tradeoff. To get the best results from OpenGL lighting, your models should be finely tessellated to get the best specular highlights and diffuse color boundaries. This yields better results, but usually at a cost of more geometric primitives, which could slow application performance.

To achieve maximum performance for lighting in your applications, use a single infinite light source. This minimizes the amount of work that OpenGL has to do to light every vertex.

Note that with programmable shaders (see advanced topics) we can do lighting calculations for each pixel.
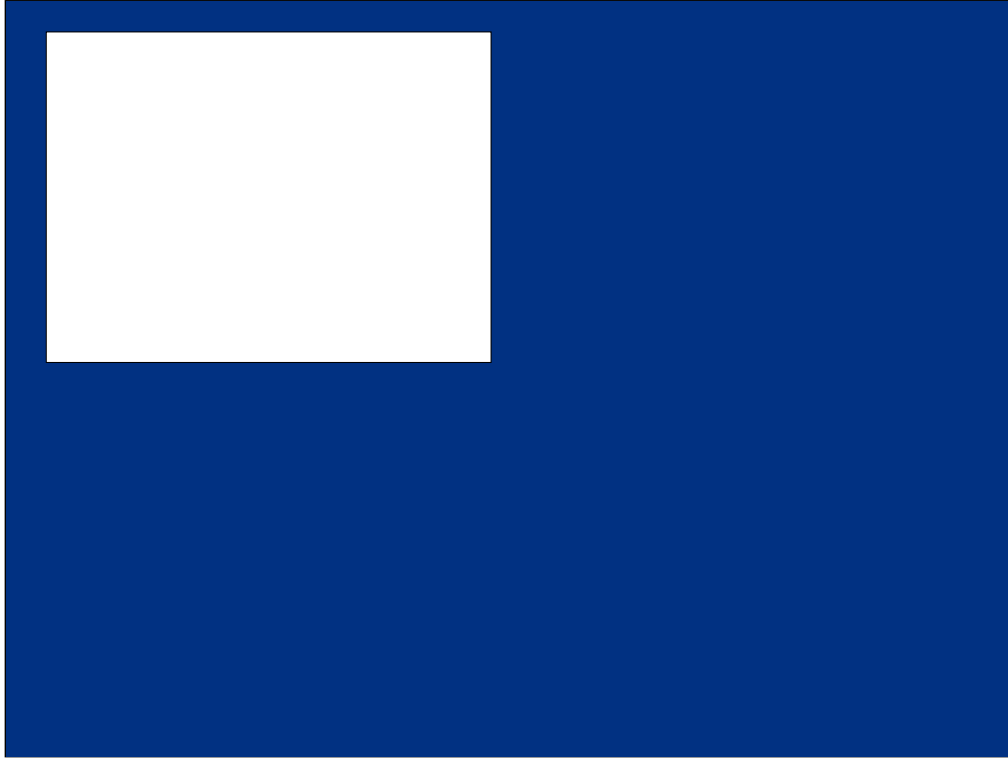
Pixel-based primitives

- Bitmaps
  - 2D array of bit masks for pixels
    - update pixel color based on current color
- Images
  - 2D array of pixel color information
    - complete color information for each pixel
- OpenGL does not understand image formats

In addition to geometric primitives, OpenGL also supports *pixel-based primitives.* These primitives contain explicit color information for each pixel that they contain. They come in two types:

> *Bitmaps* are single bit images, which are used as a mask to determine which pixels to update. The current color, set with `glColor()` is used to determine the new pixel color.

> *Images* are blocks of pixels with complete color information for each pixel.

OpenGL, however, does not understand image formats, like JPEG, PNG or GIFs. In order for OpenGL to use the information contained in those file formats, the file must be read and decoded to obtain the color information, at which point, OpenGL can rasterize the color values.

Pixel Pipeline

- Programmable pixel storage
  and transfer operations

```
glBitmap(), glDrawPixels()
```

| CPU → | Pixel Storage Modes | → | Pixel-Transfer Operations (and Pixel Map) | → | Rasterization (including Pixel Zoom) | → | Per Fragment Operations | → | Frame Buffer |

Texture Memory ← `glCopyTex*Image();`

`glReadPixels(), glCopyPixels()`
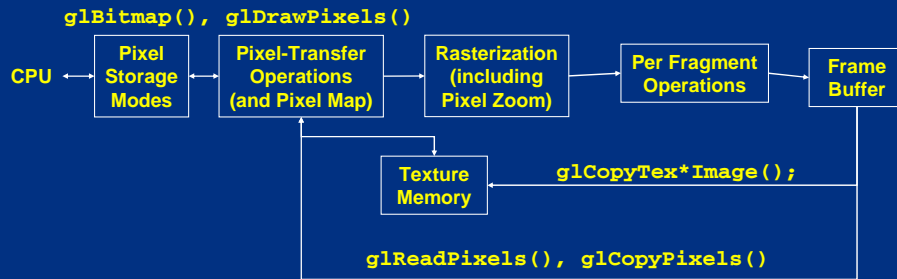
Just as there is a pipeline that geometric primitives go through when they are processed, so do pixels. The pixels are read from main storage, processed to obtain the internal format which OpenGL uses, which may include color translations or byte-swapping. After this, each pixel is rasterized into the framebuffer.

In addition to rendering into the framebuffer, pixels can be copied from the framebuffer back into host memory, or transferred into texture mapping memory.

For best performance, the internal representation of a pixel array should match the hardware. For example, with a 24 bit frame buffer, 8-8-8 RGB would probably be a good match, but 10-10-10 RGB could be bad. Warning: non-default values for pixel storage and transfer can be very slow.

Positioning Image Primitives

```
glRasterPos3f( x, y, z )
glWindosPos3f( x, y, z )
```

– raster position transformed like geometry

– discarded if raster position is outside of viewport

  • may need to fine tune viewport for desired results

Raster Position

Images are positioned by specifying the *raster position*, which maps the lower left corner of an image primitive to a point in space. Raster positions are transformed and clipped the same as vertices. If a raster position fails the clip check, no fragments are rasterized.

Rendering Images

glDrawPixels(*width, height, format, type, pixels*)

- render pixels with lower left of image at current raster position

- numerous formats and data types for specifying storage in memory
  - best performance by using format and type that matches hardware

Rendering images is done with the `glDrawPixels()` command. A block of pixels from host CPU memory is passed into OpenGL with a format and data type specified. For each pixel in the image, a fragment is generated using the color retrieved from the image, and further processed.

OpenGL supports many different formats for images including:

- *RGB* images with an RGB triplet for every pixel

- *intensity* images which contain only intensity for each pixel. These images are converted into greyscale RGB images internally.

- *depth images* which are depth values written to the depth buffer, as compared to the color framebuffer. This is useful in loading the depth buffer with values and then rendering a matching color images with depth testing enabled.

- *stencil images* which copy stencil masks in the stencil buffer. This provides an easy way to load a complicated per pixel mask.

The *type* of the image describes the format that the pixels stored in host memory. This could be primitive types like `GL_FLOAT` or `GL_INT`, or pixels with all color components packed into a primitive type, like `GL_UNSIGNED_SHORT_5_6_5`.

Reading Pixels

```
glReadPixels( x, y, width, height,
        format, type, pixels )
```

- read pixels from specified (x, y) position in framebuffer

- pixels automatically converted from framebuffer format into requested format and type

- Framebuffer pixel copy

```
glCopyPixels( x, y, width,
        height, type )
```

Just as you can send pixels to the framebuffer, you can read the pixel values back from the framebuffer to host memory for doing storage or image processing.

Pixels read from the framebuffer are processed by the pixel storage and transfer modes, as well as converting them into the format and type requested, and placing them in host memory.

Additionally, pixels can be copied from the framebuffer from one location to another using glCopyPixels(). Pixels are processed by the pixel storage and transfer modes before being returned to the framebuffer.

Texture Mapping
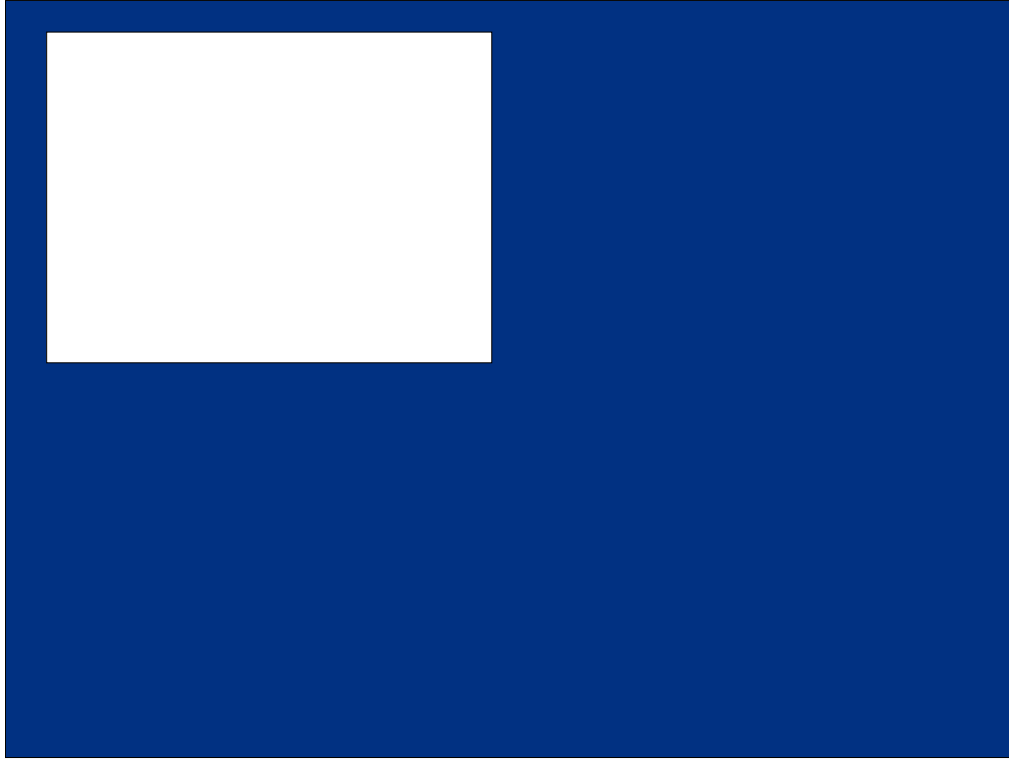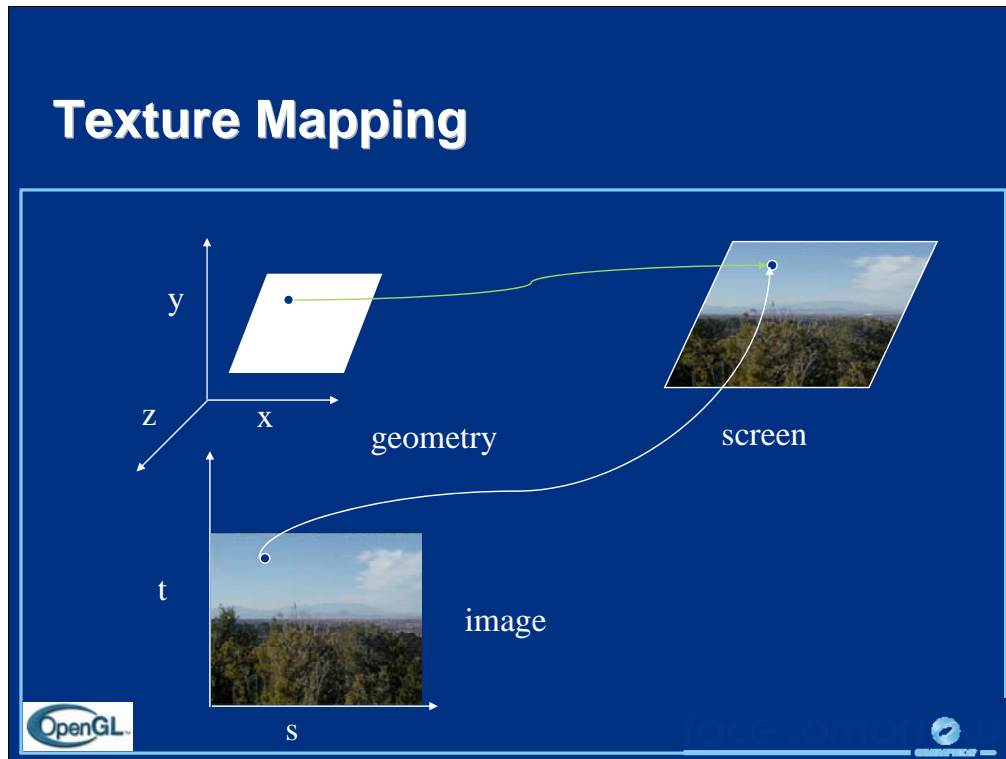
Textures are images that can be thought of as continuous and be one, two, three, or four dimensional. By convention, the coordinates of the image are s, t, r and q. Thus for the two dimensional image above, a point in the image is given by its (s, t) values with $(0, 0)$ in the lower-left corner and $(1, 1)$ in the top-right corner.

A texture map for a two-dimensional geometric object in (x, y, z) world coordinates maps a point in (s, t) space to a corresponding point on the screen.

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
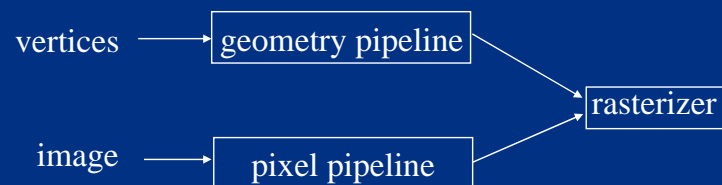
OpenGL

The advantage of texture mapping is that visual detail is in the image, not in the geometry. Thus, the complexity of an image does not affect the geometric pipeline (transformations, clipping) in OpenGL. Texture is added during rasterization where the geometric and pixel pipelines meet.

Texture Example

- The texture (below) is a 256 × 256 image that has been mapped to a rectangular polygon which is viewed in perspective
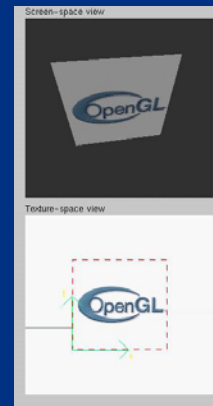
This example is from the texture mapping tutorial demo.

The size of textures must be a power of two. However, we can use image manipulation routines to convert an image to the required size.

Texture can replace lighting and material effects or be used in combination with them.

Applying Textures I

- 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
    3. specify texture parameters
        - wrapping, filtering

In the simplest approach, we must perform these three steps.

Textures reside in texture memory. When we assign an image to a texture it is copied from processor memory to texture memory where pixels are formatted differently.

Texture coordinates are actually part of the state as are other vertex attributes such as color and normals. As with colors, OpenGL interpolates texture inside geometric objects.

Because textures are really discrete and of limited extent, texture mapping is subject to aliasing errors that can be controlled through filtering.

Texture memory is a limited resource and having only a single active texture can lead to inefficient code.

Texture Objects

- Have OpenGL store your images
  - one image per texture object
  - may be shared by several graphics contexts
- Generate texture names

  glGenTextures(*n, *texIds*);

The first step in creating texture objects is to have OpenGL reserve some indices for your objects. glGenTextures() will request *n* texture ids and return those values back to you in texIds.

To begin defining a texture object, you call glBindTexture() with the id of the object you want to create. The target is one of GL_TEXTURE_{123}D(). All texturing calls become part of the object until the next glBindTexture() is called.

To have OpenGL use a particular texture object, call glBindTexture() with the target and id of the object you want to be active.

To delete texture objects, use glDeleteTextures(n, *texIds), where texIds is an array of texture object identifiers to be deleted.

Texture Objects (cont'd.)

- Create texture objects with texture data and state

  glBindTexture(*target, id*);

- Bind textures before using

  glBindTexture(*target, id*);

OpenGL

Specifying a Texture Image

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

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

- Texel colors are processed by pixel pipeline
  - pixel scales, biases and lookups can be done

OpenGL

Specifying the texels for a texture is done using the `glTexImage{123}D()` call. This will transfer the texels in CPU memory to OpenGL, where they will be processed and converted into an internal format.

The array of texels can be any size if you're using OpenGL 2.0 or greater (or the GL_ARB_texture_non_power_of_two extension is supported); otherwise, the width and height of the pixel rectangle must be a power-of-two in size. An optional one texel wide border may be added around the image. This is useful for certain wrapping modes.

The level parameter is used for defining how OpenGL should use this image when mapping texels to pixels. Generally, you'll set the level to 0, unless you are using a texturing technique called mipmapping, which we will discuss in the next section.

Mapping a Texture

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

Texture Space

Object Space
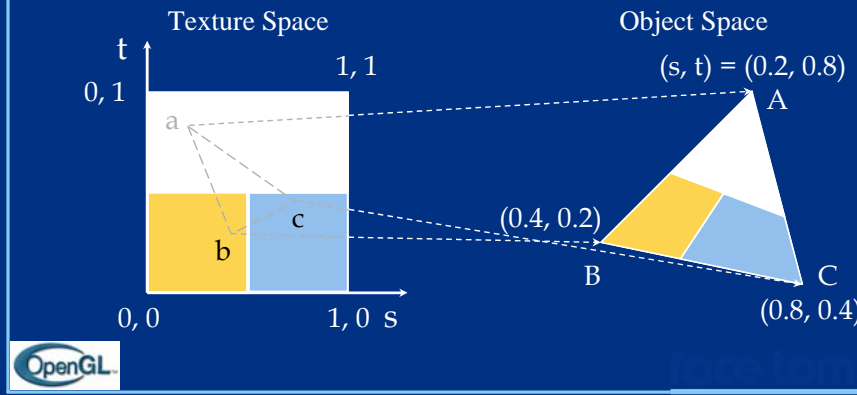
When you want to map a texture onto a geometric primitive, you need to provide texture coordinates. The glTexCoord*() call sets the current texture coordinates. Valid texture coordinates are between 0 and 1, for each texture dimension, and the default texture coordinate is (0, 0, 0, 1). If you pass fewer texture coordinates than the currently active texture mode (for example, using glTexCoord1d() while GL_TEXTURE_2D is enabled), the additionally required texture coordinates take on default values.

Generating Texture Coordinates

- Automatically generate texture coords

  `glTexGen{ifd}[v]()`

- specify a plane $Ax + By + Cz + D = 0$

  – generate texture coordinates based upon distance from plane

- generation modes

  – **GL_OBJECT_LINEAR**

  – **GL_EYE_LINEAR**

  – **GL_SPHERE_MAP**

OpenGL

You can have OpenGL automatically generate texture coordinates for vertices by using the `glTexGen()` and `glEnable(GL_TEXTURE_GEN_{STRQ})`. The coordinates are computed by determining the vertex's distance from each of the enabled generation planes.

As with lighting positions, texture generation planes are transformed by the ModelView matrix, which allows different results based upon when the `glTexGen()` is issued.

There are three ways in which texture coordinates are generated:

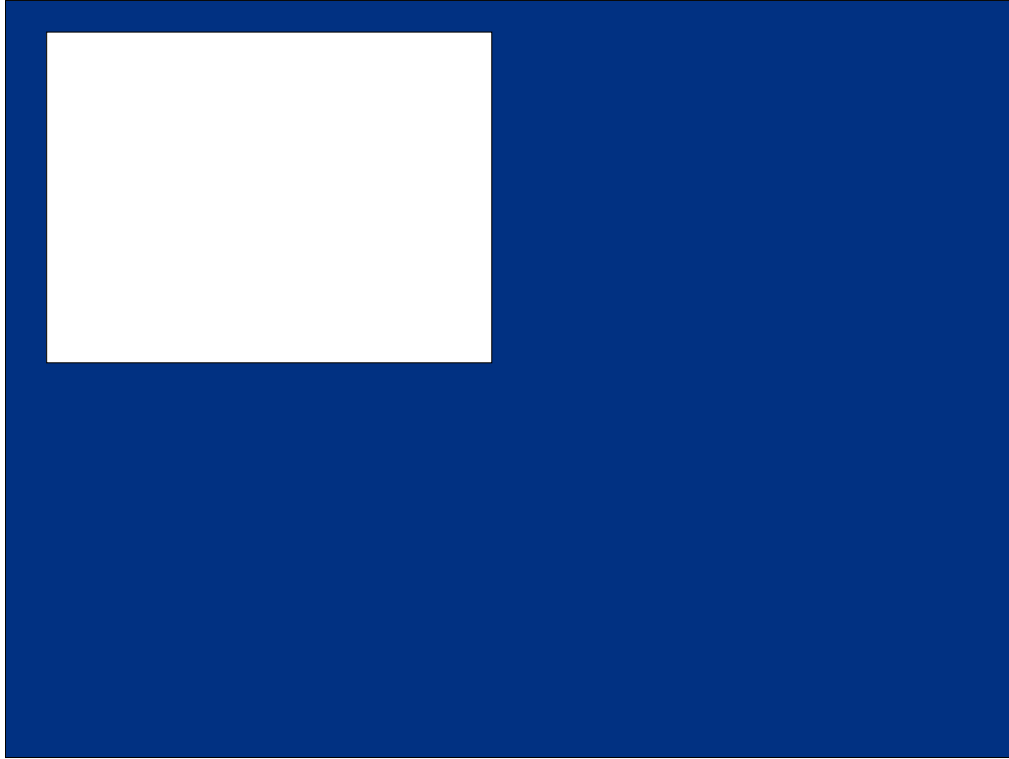`GL_OBJECT_LINEAR` - textures are fixed to the object (like wallpaper)

`GL_EYE_LINEAR` - texture fixed in space, and object move through texture (like underwater light shining on a swimming fish)

`GL_SPHERE_MAP` - object reflects environment, as if it were made of mirrors (like the shiny guy in Terminator 2)

Texture Tutorial
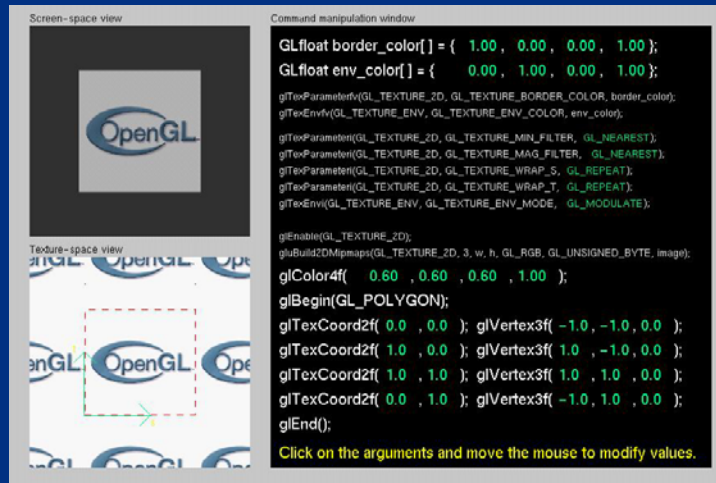
Applying Textures II

- – 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

The general steps to enable texturing are listed above. Some steps are optional, and due to the number of combinations, complete coverage of the topic is outside the scope of this course.

Here we use the *texture object* approach. Using texture objects may enable your OpenGL implementation to make some optimizations behind the scenes.

As with any other OpenGL state, texture mapping requires that `glEnable()` be called. The tokens for texturing are:

> `GL_TEXTURE_1D` - one dimensional texturing
>
> `GL_TEXTURE_2D` - two dimensional texturing
>
> `GL_TEXTURE_3D` - three dimensional texturing

2D texturing is the most commonly used. 1D texturing is useful for applying contours to objects (like altitude contours to mountains). 3D texturing is useful for volume rendering.

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

OpenGL

Textures and the objects being textured are rarely the same size (in pixels). Filter modes determine the methods used by how texels should be expanded (magnification), or shrunk (minification) to match a pixel's size. An additional technique, called mipmapping is a special instance of a minification filter.

Wrap modes determine how to process texture coordinates outside of the [0,1] range. The available modes are:

GL_CLAMP - clamp any values outside the range to closest valid value, causing the edges of the texture to be "smeared" across the primitive

GL_REPEAT - use only the fractional part of the texture coordinate, causing the texture to repeat across an object

Finally, the texture environment describes how a primitives fragment colors and texel colors should be combined to produce the final framebuffer color. Depending upon the type of texture (i.e., intensity texture vs. RGBA texture) and the mode, pixels and texels may be simply multiplied, linearly combined, or the texel may replace the fragment's color altogether.

Filter Modes

Example:

glTexParameteri(*target, type, mode*);

Texture	Polygon	Texture	Polygon
Magnification		Minification	
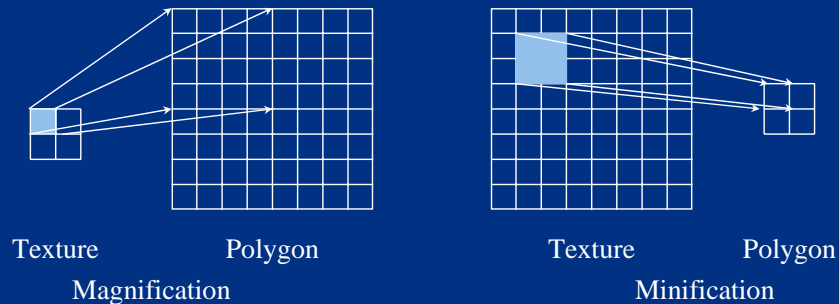
Filter modes control how pixels are minified or magnified. Generally a color is computed using the nearest texel or by a linear average of several texels.

The filter type, above is one of GL_TEXTURE_MIN_FILTER or GL_TEXTURE_MAG_FILTER.

The mode is one of GL_NEAREST, GL_LINEAR, or special modes for mipmapping. Mipmapping modes are used for minification only, and can have values of:

GL_NEAREST_MIPMAP_NEAREST

GL_NEAREST_MIPMAP_LINEAR

GL_LINEAR_MIPMAP_NEAREST

GL_LINEAR_MIPMAP_LINEAR

Full coverage of mipmap texture filters is outside the scope of this course.

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

OpenGL

As primitives become smaller in screen space, a texture may appear to shimmer as the minification filters creates rougher approximations. Mipmapping is an attempt to reduce the shimmer effect by creating several approximations of the original image at lower resolutions.

Each mipmap level should have an image which is one-half the height and width of the previous level, to a minimum of one texel in either dimension. For example, level 0 could be 32 x 8 texels. Then level 1 would be 16 x 4; level 2 would be 8 x 2; level 3, 4 x 1; level 4, 2 x 1, and finally, level 5, 1 x 1.

The `gluBuild*Dmipmaps()` routines will automatically generate each mipmap image, and call `glTexImage*D()` with the appropriate level value.

OpenGL 1.2 introduces control over the minimum and maximum mipmap levels, so you do not have to specify every mipmap level (and also add more levels, on the fly).

Wrapping Mode

- Example:

```
glTexParameteri( GL_TEXTURE_2D,
    GL_TEXTURE_WRAP_S, GL_CLAMP )

glTexParameteri( GL_TEXTURE_2D,
    GL_TEXTURE_WRAP_T, GL_REPEAT )
```
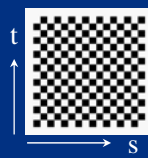
texture

GL_REPEAT
wrapping

GL_CLAMP
wrapping

Wrap mode determines what should happen if a texture coordinate lies outside of the [0,1] range. If the GL_REPEAT wrap mode is used, for texture coordinate values less than zero or greater than one, the integer is ignored and only the fractional value is used.

If the GL_CLAMP wrap mode is used, the texture value at the extreme (either 0 or 1) is used.

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**

The texture mode determines how texels and fragment colors are combined. The most common modes are:

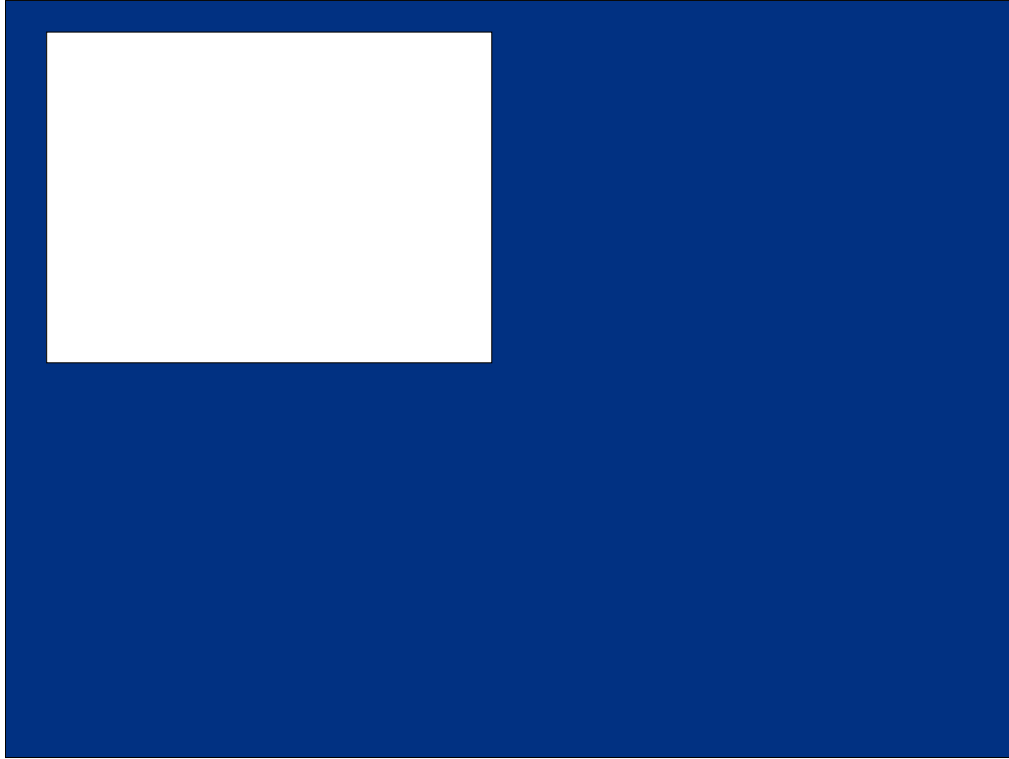GL_MODULATE - multiply texel and fragment color

GL_BLEND - linearly blend texel, fragment, env color

GL_REPLACE - replace fragment's color with texel

If prop is GL_TEXTURE_ENV_COLOR, param is an array of four floating point values representing the color to be used with the GL_BLEND texture function.

Immediate Mode versus Display Listed Rendering

- Immediate Mode Graphics
  - Primitives are sent to pipeline and display right away
  - No memory of graphical entities
- Display Listed Graphics
  - Primitives placed in display lists
  - Display lists kept on graphics server
  - Can be redisplayed with different state
  - Can be shared among OpenGL graphics contexts

OpenGL

If display lists are shared, texture objects are also shared.

To share display lists among graphics contexts in the X Window System, use the `glXCreateContext()` routine.

Immediate Mode versus Display Lists
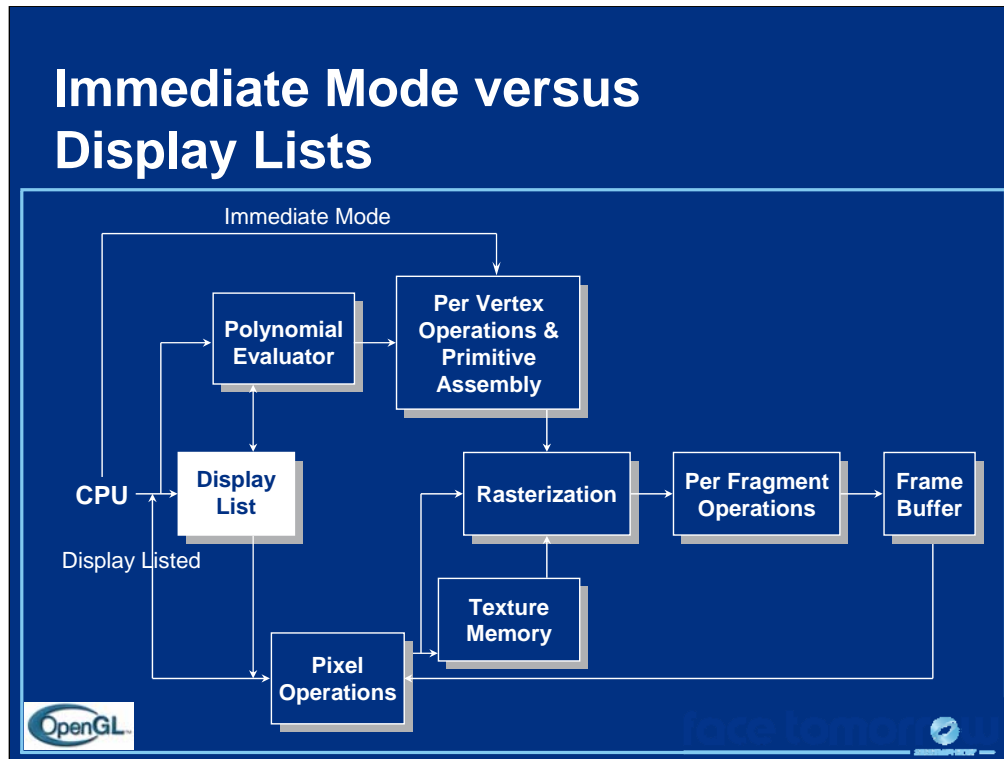
In immediate mode, primitives (vertices, pixels) flow through the system and produce images. These data are lost. New images are created by reexecuting the display function and regenerating the primitives.

In retained mode, the primitives are stored in a display list (in "compiled" form). Images can be recreated by "executing" the display list. Even without a network between the server and client, display lists should be more efficient than repeated executions of the display function.

Display Lists

- Creating a display list

```
GLuint id;
void init( void )
{
    id = glGenLists( 1 );
    glNewList( id, GL_COMPILE );
    /* other OpenGL routines */
    glEndList();
}
```

- Call a created list

```
void display( void )
{
    glCallList( id );
}
```

Instead of GL_COMPILE, glNewList also accepts the constant GL_COMPILE_AND_EXECUTE, which both creates and executes a display list.

If a new list is created with the same identifying number as an existing display list, the old list is replaced with the new calls. No error occurs.

Display Lists (cont'd.)

- Not all OpenGL routines can be stored in display lists
- State changes persist, even after a display list is finished
- Display lists can call other display lists
- Display lists are not editable, but you can fake it
  - make a list (A) which calls other lists (B, C, and D)
  - delete and replace B, C, and D, as needed

OpenGL

Some routines cannot be stored in a display list. Here are some of them:

all `glGet*` routines

`glIs*` routines (e.g., `glIsEnabled`, `glIsList`, `glIsTexture`)

`glGenLists`   `glDeleteLists`   `glFeedbackBuffer`

`glSelectBuffer`   `glRenderMode`   `glVertexPointer`

`glNormalPointer` `glColorPointer`   `glIndexPointer`

`glReadPixels`   `glPixelStore`   `glGenTextures`

`glTexCoordPointer`   `glEdgeFlagPointer`

`glEnableClientState`   `glDisableClientState`

`glDeleteTextures`   `glAreTexturesResident`

`glFlush`   `glFinish`

If there is an attempt to store any of these routines in a display list, the routine is executed in immediate mode. No error occurs.

Display Lists and Hierarchy

- 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();
```
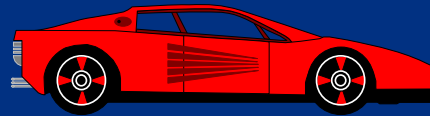
Advanced Primitives

- Vertex Arrays

- Bernstein Polynomial Evaluators

  - basis for GLU NURBS

    - NURBS (Non-Uniform Rational B-Splines)

- GLU Quadric Objects

  - sphere

  - cylinder (or cone)

  - disk (circle)

In addition to specifying vertices one at a time using `glVertex*()`, OpenGL supports the use of arrays, which allows you to pass an array of vertices, lighting normals, colors, edge flags, or texture coordinates. This is very useful for systems where function calls are computationally expensive. Additionally, the OpenGL implementation may be able to optimize the processing of arrays.

OpenGL evaluators, which automate the evaluation of the Bernstein polynomials, allow curves and surfaces to be expressed algebraically. They are the underlying implementation of the OpenGL Utility Library's NURBS implementation.

Finally, the OpenGL Utility Library also has calls for generating polygonal representation of quadric objects. The calls can also generate lighting normals and texture coordinates for the quadric objects.

Alpha: the 4th Color Component

- Measure of Opacity
  - simulate translucent objects
    - glass, water, etc.
  - composite images
  - antialiasing
  - ignored if blending is not enabled

    `glEnable( GL_BLEND )`

OpenGL

The alpha component for a color is a measure of the fragment's opacity. As with other OpenGL color components, its value ranges from 0.0 (which represents completely transparent) to 1.0 (completely opaque).

Alpha values are important for a number of uses:

- simulating translucent objects like glass, water, etc.
- blending and compositing images
- antialiasing geometric primitives

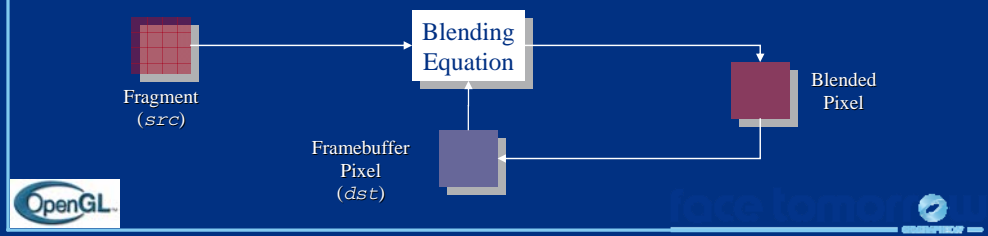Blending can be enabled using `glEnable(GL_BLEND)`.

Blending

- Combine fragments with pixel values that are already in the framebuffer

  glBlendFunc(*src, dst*)

  $$\vec{C}_r = src\ \vec{C}_f + dst\ \vec{C}_p$$

Blending combines fragments with pixels to produce a new pixel color. If a fragment makes it to the blending stage, the pixel is read from the framebuffer's position, combined with the fragment's color and then written back to the position.

The fragment and pixel each have a factor which controls their contribution to the final pixel color. These *blending factors* are set using `glBlendFunc()`, which sets the source factor, which is used to scale the incoming fragment color, and the destination blending factor, which scales the pixel read from the framebuffer. Common OpenGL blending factors are:

GL_ONE	GL_ZERO
GL_SRC_ALPHA	GL_ONE_MINUS_SRC_ALPHA

They are then combined using the *blending equation*, which is addition by default.

Blending is enabled using `glEnable(GL_BLEND)`

Note: If your OpenGL implementation supports the `GL_ARB_imaging` extension, you can modify the blending equation as well.

Fog

glFog{if}(*property, value*)

- Depth Cueing
  - Specify a range for a linear fog ramp
    - **GL_FOG_LINEAR**
- Environmental effects
  - Simulate more realistic fog
    - **GL_FOG_EXP**
    - **GL_FOG_EXP2**

OpenGL

Fog works in two modes:

Linear fog mode is used for depth cueing affects. In this mode, you provide OpenGL with a starting and ending distance from the eye, and between those distances, the fog color is blended into the primitive in a linear manner based on distance from the eye.

In this mode, the fog coefficient is computed as
$$f = \frac{z - start}{end - start}$$

Here's a code snippet for setting up linear fog:

```
glFogf(GL_FOG_MODE, GL_FOG_LINEAR);
glFogf(GL_FOG_START, fogStart);
glFogf(GL_FOG_END, fogEnd);
glFogfv(GL_FOG_COLOR, fogColor);
glEnable(GL_FOG);
```

Exponential fog mode is used for more natural environmental affects like fog, smog and smoke. In this mode, the fog's density increases exponentially with the distance from the eye. For these modes, the coefficient is computed as

$$f = \begin{cases} e^{-density \cdot z} & \texttt{GL\_FOG\_EXP} \\ e^{-density \cdot z^2} & \texttt{GL\_FOG\_EXP2} \end{cases}$$
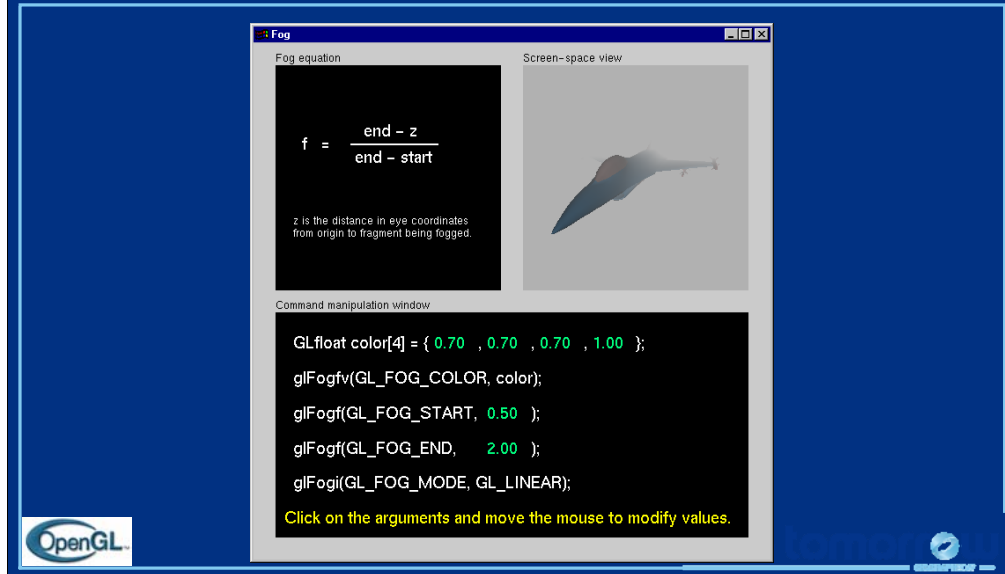
In this tutorial, experiment with the different fog modes, and in particular, the parameters which control either the fog density (for exponential mode) and the start and end distances (for linear mode).

Multi-pass Rendering

- Blending allows results from multiple drawing passes to be combined together
  - enables more complex rendering algorithms

Example of bump-mapping done with a multi-pass OpenGL algorithm

OpenGL blending enables techniques which may require accumulating multiple images of the same geometry with different rendering parameters to be done.
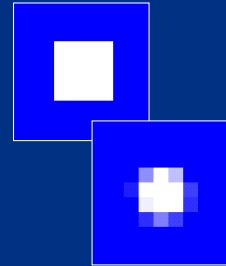
Antialiasing

- Removing the Jaggies

  glEnable(*mode*)

  - **GL_POINT_SMOOTH**
  - **GL_LINE_SMOOTH**
  - **GL_POLYGON_SMOOTH**
  - alpha value computed by computing sub-pixel coverage
 - available in both RGBA and colormap modes

Antialiasing is a process to remove the *jaggies* which is the common name for jagged edges of rasterized geometric primitives. OpenGL supports antialiasing of all geometric primitives by enabling both GL_BLEND and one of the constants listed above.

Antialiasing is accomplished in RGBA mode by computing an alpha value for each pixel that the primitive touches. This value is computed by subdividing the pixel into *subpixels* and determining the ratio used subpixels to total subpixels for that pixel. Using the computed alpha value, the fragment's colors are blended into the existing color in the framebuffer for that pixel.

Color index mode requires a ramp of colors in the colormap to simulate the different values for each of the pixel coverage ratios.

In certain cases, GL_POLYGON_SMOOTH may not provide sufficient results, particularly if polygons share edges. As such, using the accumulation buffer for full scene antialising may be a better solution.
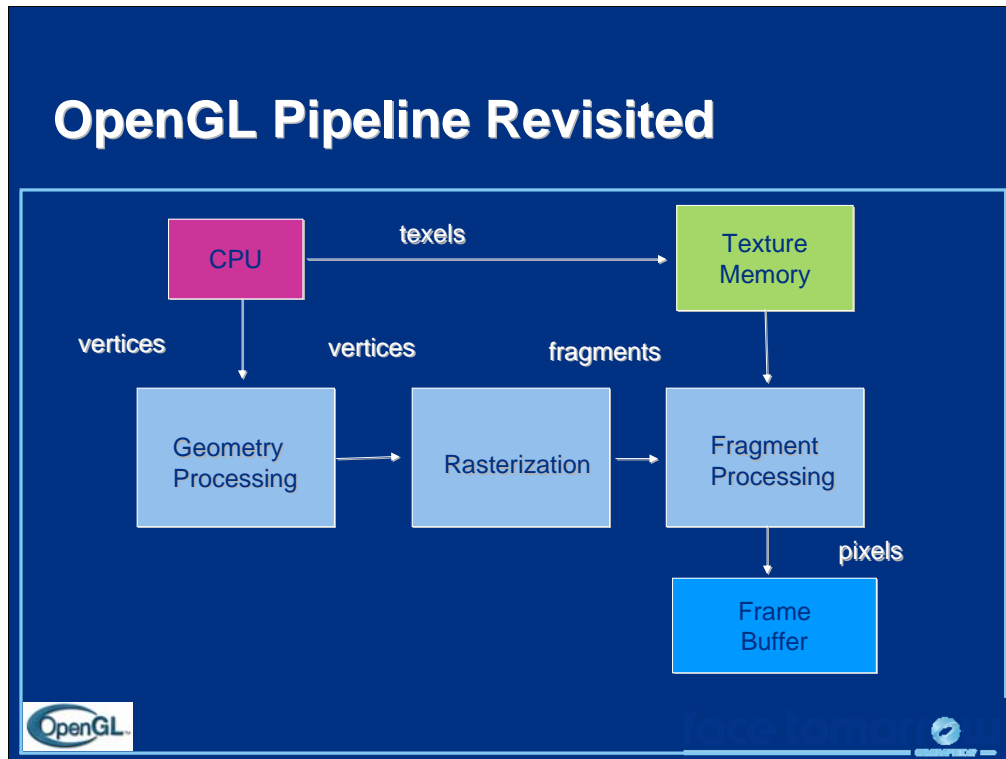
There are three types of memory that can be accessed: normal CPU memory, texture memory, and the frame buffer.

At the present, geometry processing and fragment processing can be altered by writing programs called shaders whereas the rasterizer is fixed.

Geometry Processing

- Coordinate Transformations
- Primitive Assembly
- Clipping
- Per-vertex lighting
- Programmable through a vertex program or vertex shader

Geometry processing works on vertices represented in four dimensional homogeneous coordinates.

In the fixed function pipeline (which is the default if no shader is loaded by the application), the geometric processing includes:

The modelview and projection transformations

The assembly of groups of vertices between a glBegin and a glEnd into primitives such as lines, points, and ploygons.

Clipping these primitives against the volume defined by glOrtho, gluPerspective, or glFrustum.

Computing the modified Phong model (if lighting is enabled) at each vertex to determine a vertex color.

Rasterizer

- Outputs fragments ("potential pixels") that are interior to primitives

- Interpolates colors, depth, texture coordinates, and other per vertex variables to obtain values for each fragment

- Not programmable

Each fragment corresponds to a location in the frame buffer and has attributes such as color, depth, and textures coordinates that are obtained by interpolating the corresponding values as the vertices. However, the final color of a pixel is determined by the color from the rasterizer, and other factors including hidden surface removal, compositing, and texture mapping which are done as part of fragment processing.

Fragment Processing

- Assigns color to each fragment and updates frame buffer

- Handles hidden surface removal, texture mapping, and blending

- Programmable through fragment program or fragment shader

OpenGL

Note that many of the operation that are carried out on a per vertex basis such as shading can be carried out on a per fragment basis through a fragment shader. For example, rather than using the vertex colors computed by the modified Phong model and then interpolating them across a primitive, the same model can be computed for each fragment because the rasterizer will interpolate normals and other quantities across each primitive. This method is called Gouraud or interpolated shading.

GLSL

- OpenGL Shading Language
- C like language for writing both vertex and fragment shaders
- Adds matrix and vector data types + overloading of operators
- OpenGL state variables available to shaders
- Variables can be passed among shaders and applications

Supported as a extension in earlier versions of OpenGL but is now part of the OpenGL standard.

Matrix and vector types are 2, 3, and 4 dimensional.

Although there are no pointers, we can pass matrices, vectors, and C-structs back and forth.

Although the language is the same for vertex and fragment shaders, each has a different execution model. Vertex shaders are invoked for each vertex produced by the application; fragment shaders for each fragment that is produced by the rasterizer. GLSL has type qualifiers to identify variables that may be local to a shader, pass from a vertex shader to a fragment shader, and to identify input and output variables.
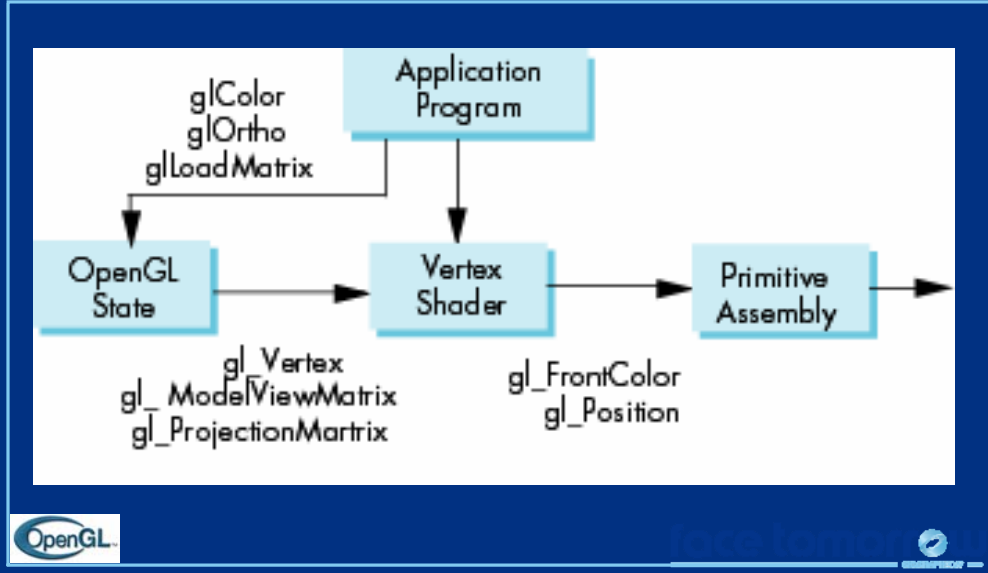
The vertex shader is executed for each vertex defined in the application. Minimally every vertex shader must output a vertex position (gl_Position) to the rasterizer. A user defined vertex shader must do all the functions that the fixed geometric processor does. Hence, most vertex shaders must do the coordinate system changes usually done by the modelview and projection matrices on the application on the input vertex position (gl_Vertex). Since OpenGL state variables are available to the shader, we often see lines of code such as

gl_Position = gl_ProjectionMatrix*gl_ModeViewMatrix*gl_Vertex;

in the shader. Most shaders also produce an output color (gl_FrontColor, gl_BackColor) for each vertex. Here is a trival vertex shader that colors every vertex red and does the standard coordinate changes.

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
void main(void)
{
 gl_Position =
  gl_ModelViewProjectionMatrix*gl_Vertex;
 gl_FrontColor = red;
}
```

Simple Vertex Shader

```
uniform float time;


void main()
{
    gl_Position = gl_ModelViewProjectionMatrix *
                  gl_Vertex;

    gl_Position.xyz = (1.0+0.1*sin(0.001*time)) *
                  gl_Position.xyz;

    gl_FrontColor = gl_Color;

}
```

In this simple shader, a variable time is provided by the application. The product of the model-view and projection matrices (gl_ModelViewProjectionMatrix) is provided by the state and is used to convert the position each vertex from object coordinates (gl_Vertex) to clip coordinates (gl_Position). The time variable is used to sinsusoidally vary the y component of the vertex. Finally the color set by the application (gl_Color) is sent as the front color (gl_FrontColor) along with the new position to the rasterizer.

Vertex Shader Applications

- General Vertex Lighting
- Off loading CPU
- Dynamic Meshes
- Morphing
- Additional Application Specific Vertex Attributes

Some basic fragment shader applications include:

Vertex Lighting: Use models that either more physically realistic than the modified Phong model or are non photorealistic such as cartoon shading.

Many operations can be done either in the CPU or the GPU. Often we can offload the CPU by pushing operations to the GPU. One example is creating waves on a mesh by sending a time parameter to the shader which controls mesh vertex heights sinusoidally thus creating waves. Particle system calculations are another example.

Morphing involves interpolation between two sets of vertices. The interpolation can be done on the GPU.

We can pass additional information on a per vertex basis to the GPU. For example, in a simulation we might have temperature or flow data at each vertex.
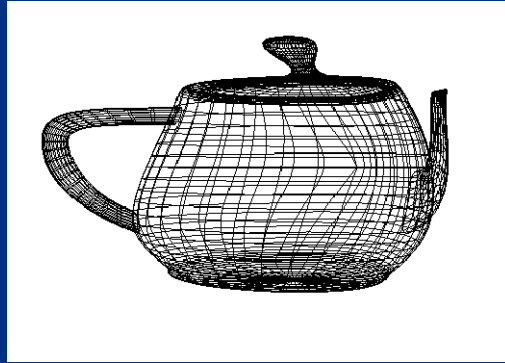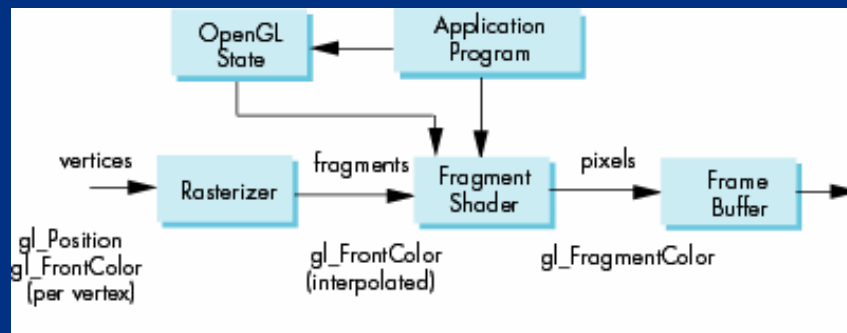
In this example each vertex is rotated in the about the y axis by an angle dependent on its distance from the axis (twisting).

Although we could do this operation in the application, by doing it in the shader we offload the CPU.

Fragment Shader Execution

The fragment shader is executed for each fragment output from the rasterizer. Every fragment program must produce a fragment color (gl_FragmentColor). Since each fragment has a location in the frame buffer, fragment locations cannot be altered. Vertex attributes, either bultin (gl_FrontColor) or application defined are available to the fragment shader with values that have been interpolated across the primitive. Tests such as depth are done after the fragment shader.

Below is a simple fragment shader that passes through a color defined and output by the vertex shader (color_out) for each vertex that is interpolated across the primitive.

```
varying vec3 color_out;
void main(void)
{
  gl_FragColor = color_out;
}
```

Simple Fragment Shader

```
uniform float time;
void main()
{
    float d = length(gl_FragCoord.xy);
    gl_FragColor.r = 0.5*(1.0+sin(0.001*time)) *
            gl_FragCoord.x/d;
    gl_FragColor.g = 0.5*(1.0+cos(0.001*time)) *
            gl_FragCoord.y/d;
    gl_FragColor.b = gl_FragCoord.z;
    gl_FragColor.a = 1.0;
}
```

In this simple fragment shader the time variable provided by the application is used to vary the red and green components of each fragment's color sinusoidally. The location each fragment in the frame buffer (gl_FragCoord) is available from the state. Every fragment program must output a fragment color (gl_FragColor).

Fragment Shader Applications

- Per Fragment Lighting
- Bump Maps
- Environment Maps
- Texture Manipulation
- Shadow Maps

OpenGL

Fragment shaders are more powerful than vertex shaders. With a fragment shader you can apply the lighting model to each fragment generated by the rasterizer rather than using interpolated values from each vertex. One example is bump mapping where the normal is altered at each fragment allowing the rendering of surfaces that are not smooth. Fragment shader also have access to textures so that multipass techniques such as environment maps can be carried out in real time.

In this example, a diffuse color was computed for each fragment by using the interpolating normals and light vectors ouput by the rasterizer.

A texture was also applied to each fragment. Per fragment lighting using the same model as a vertex shader or the default modified Phong model should result in a much smoother image.

Linking with Application

- In OpenGL application, we must:
  - Read shaders
  - Compile shaders
  - Define variables
  - Link everything together
- OpenGL 2.0 contains a set of functions for each of these steps

Similar to the initialization of other OpenGL application programs, a set of operations must be carried out to set up user written shaders and link them with an OpenGL application program. Shaders are placed in program objects. A program object can contain multiple shaders of each type. Just as with any program, a shader must be compiled and linked with other program entities. The linking stage sets up internal tables that allow the application to tie together variables in the shaders with variables in the application program.

Accumulation Buffer

- Problems of compositing into color buffers
  - limited color resolution
    - clamping
    - loss of accuracy
  - Accumulation buffer acts as a "floating point" color buffer
    - accumulate into accumulation buffer
    - transfer results to frame buffer

Since most graphics hardware represents colors in the framebuffer as integer numbers, we can run into problems if we want to accumulate multiple images together.

Suppose the framebuffer has 8 bits per color component. If we want to prevent any possible overflow adding 256 8 bit per color images, we would have to divide each color component by 256 thus reducing us to 0 bits of resolution.

Many OpenGL implementations support the accumulation in software only, and as such, using the accumulation buffer may cause some slowness in rendering.

High end graphics cards now use floating point frame buffers which allow us to do many of the operations such as compositing without and faster without loss of accuracy directly in the frame buffer and without use of an accumulation buffer.

Accumulation Buffer Applications

- Compositing
- Full Scene Antialiasing
- Depth of Field
- Filtering
- Motion Blur

Compositing, which combines several images into a single image, done with the accumulation buffer generally gives better results than blending multiple passes into the framebuffer.

Full scene antialiasing utilizes compositing in the accumulation buffer to smooth the jagged edges of all objects in the scene. *Depth of field*, simulates how a camera lens can focus on a single object while other objects in the view may be out of focus.

Filtering techniques, such as convolutions and blurs (from image processing) can be done easily in the accumulation buffer by rendering the same image multiple times with slight pixel offsets.

Motion blur, a technique often used in Saturday morning cartoons, simulates motion in a stationary object. We can do with the accumulation buffer by rendering the same scene multiple times, and varying the position of the object we want to appear as moving for each render pass. Compositing the results will give the impression of the object moving.

Unlike other buffers, we do not draw into the stencil buffer. We set its values with the stencil functions. However, the rendering can alter the values in the stencil buffer depending on whether a fragment passes or fails the stencil test.

Getting to the Framebuffer
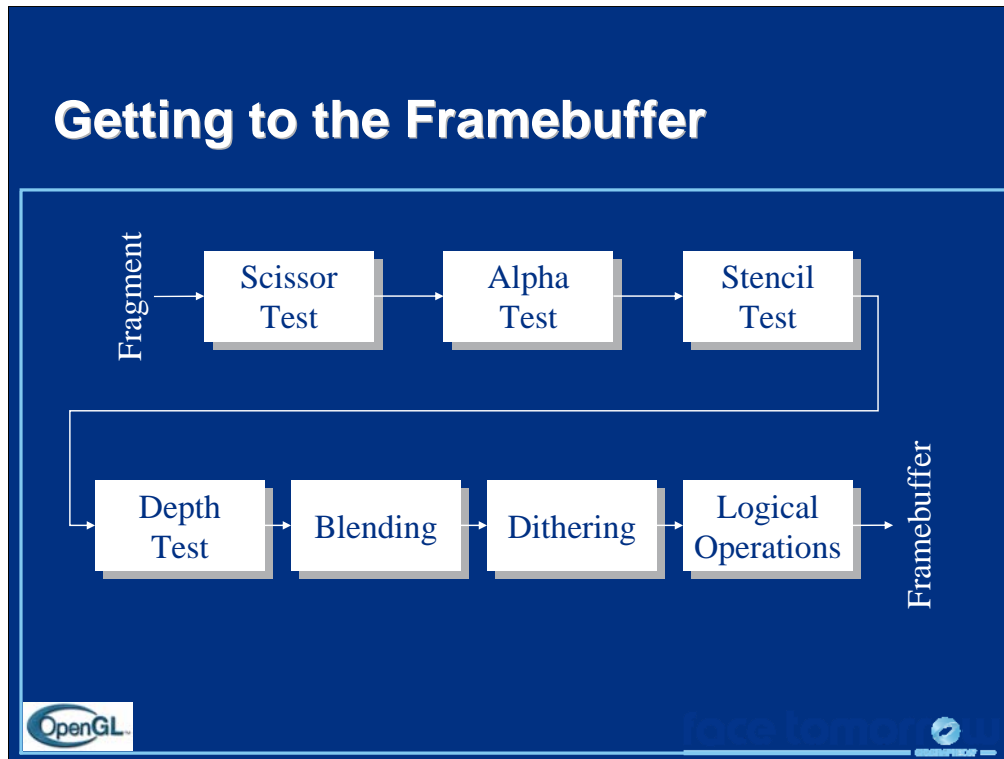
In order for a fragment to make it to the frame buffer, it has a number of testing stages and pixel combination modes to go through.

The tests that a fragment must pass are:

- *scissor test* - an additional clipping test
- *alpha test* - a filtering test based on the alpha color component
- *stencil test* - a pixel mask test
- *depth test* - fragment occlusion test

Each of these tests is controlled by a `glEnable()` capability.

If a fragment passes all enabled tests, it is then blended, dithered and/or logically combined with pixels in the framebuffer. Each of these operations can be enabled and disabled.

Alpha Test

- Reject pixels based on their alpha value

    glAlphaFunc(*func, value*)

    glEnable(*GL_ALPHA_TEST*)

    – use alpha as a mask in textures

Alpha values can also be used for fragment testing. glAlphaFunc() sets a value which, if glEnable(GL_ALPHA_TEST) has been called, will test every fragment's alpha against the value set, and if the test fails, the fragment is discarded.

The functions which glAlphaFunc() can use are:

GL_NEVER	GL_LESS
GL_EQUAL	GL_LEQUAL
GL_GREATER	GL_NOTEQUAL
GL_GEUQAL	GL_ALWAYS

The default is GL_ALWAYS, which always passes fragments.

Alpha testing is particularly useful when combined with texture mapping with textures which have an alpha component. This allows your texture map to act as a localized pixel mask. This technique is commonly used for objects like trees or fences, where modeling the objects (and all of its holes) becomes prohibitive.

GPUs and GLSL

- Over the past few years, graphical processing units (GPUs) have become more powerful and now are programmable

- Support first through OpenGL extensions and OpenGL Shading Language (GLSL)

- Incorporated in OpenGL 2.0

By most measures, GPUs are more powerful than the CPUs in workstations. However, the architecture of a GPU is that of a stream processor.

GPUs can also be programmed using Nvidia's Cg (C for Graphics) language which is almost identical to Microsoft's High Level Shading Language (HLSL). Hence shaders written in Cg will run under both OpenGL and DirectX on Windows platforms. Cg and GLSL are very similar. However, the advantage of GLSL is that, like the rest of OpenGL, it is platform independent. In addition, because GLSL is tied to OpenGL it is easier for the programmer to access OpenGL state variables.
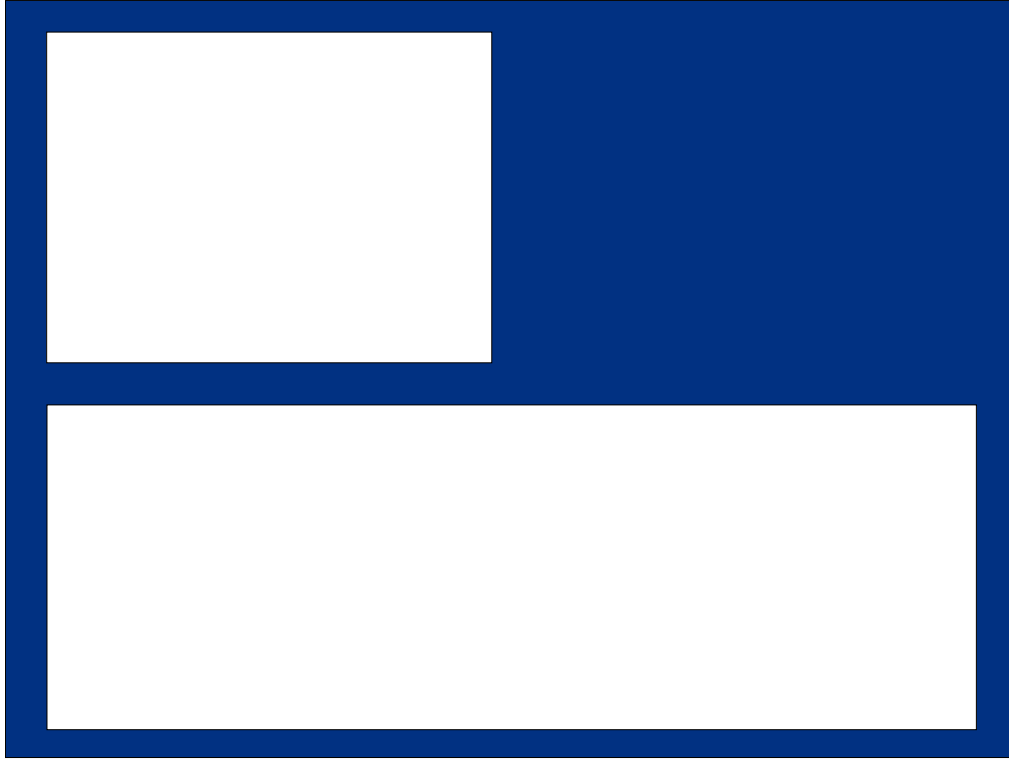
On-Line Resources

- **http://www.opengl.org**
  - start here; up to date specification and lots of sample code
  - online "man pages" for all OpenGL functions
- **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, 6th Edition
- The OpenGL Shading Language, 2nd Edition
- Interactive Computer Graphics: A top-down approach with OpenGL, 4th Edition
- OpenGL Programming for the X Window System
- OpenGL: A Primer 3rd Edition
- OpenGL Distilled

The OpenGL Programming Guide – often referred to as the "Red Book" due to the color of its cover – discusses all aspects of OpenGL programming, discussing all of the features of OpenGL in detail.

Mark Kilgard's *OpenGL Programming for the X Window System*, is the "Green Book", and Ron Fosner's *OpenGL Programming for Microsoft Windows*, which has a white cover is sometimes called the "Alpha Book." *The OpenGL Shading Language*, by Randi Rost, Barthold Litchenbelt, and John Kessenich, is the "Orange Book."

All of the OpenGL programming series books, along with *Interactive Computer Graphics: A top-down approach with OpenGL*, *OpenGL: A Primer, and OpenGL Distilled* are published by Addison Wesley Publishers.

Thanks for Coming

Questions and Answers

Ed Angel	`angel@cs.unm.edu`
Dave Shreiner	`shreiner@siggraph.org`
Vicki Shreiner	`vshreiner@sgi.com`