An Interactive Introduction to OpenGL Programming

Course # 29

Dave Shreiner

Ed Angel

Vicki Shreiner

Table of Contents

Introduction

"An Interactive Introduction to OpenGL Programming" provides an overview of the OpenGL Application Programming Interface (API), a library of subroutines for drawing three-dimensional objects and images on a computer. After the completion of the course, a programmer able to write simple programs in the "C" language will be able to create an OpenGL application that has moving 3D objects that look like they are being lit by lights in the scene and by specifying colors or images that should be used to color those objects. Additionally, the viewpoint of the scene can be controlled by the mouse and keyboard, and can be updated interactively. Finally, the course provides references for exploring more of the capabilities of OpenGL that aren't covered in the class.

Course Prerequisites

You need very little experience with computer graphics or with programming to become successful using OpenGL. Our course does expect you to have a reading knowledge of a procedural language (all of our examples are in "C", but don't use any advanced concepts). We also try to explain the background of each concept as well as demonstrate how to accomplish the technique in OpenGL. A bibliography is included to aid you in finding more information or clarifying points that didn't make sense the first time around.

OpenGL and Window Systems

The OpenGL library is platform independent with implementations available on almost every operating system: Microsoft Windows; Apple Computer's MAC O/S; and most version of UNIX, including Linux. Although the code that you write using the OpenGL API is easily moved between platforms, OpenGL relies on the native windowing system of the computer you're running the program on. Each windowing system has unique methods for opening windows, processing keyboard and mouse input, and enabling windows to be able to be drawn into by OpenGL. In order to make this process simpler, this course uses the GLUT library (OpenGL Utility Toolkit, authored by Mark Kilgard) to hide the specifics required for different operating systems.

Topics

Our course covers a number of topics that enable the creation of interesting OpenGL applications:

- *3D object modeling* – how to combine vertices to create the three geometric primitives: points, lines, and polygons. We'll also discuss how to construct objects by assembling geometric primitives. By far, modeling objects is the most laborious task in 3D graphics. For all but the simplest shapes, or shapes derived from mathematical formulas (i.e., circles, spheres, cones, etc.), most objects are created using a modeling program (e.g., Maya, 3D Studio Max, Houdini, etc.). The GLUT library contains routines for creating some common shapes as well, which we briefly discuss.
- *Transformations* – computer graphics relies heavily on the use of 4×4 matrices for mapping our virtual three-dimensional world to the two-dimensional screen.

OpenGL takes care of doing all the math, and simplifies the specification and use of these matrices. We'll find that using OpenGL, we can easily control:

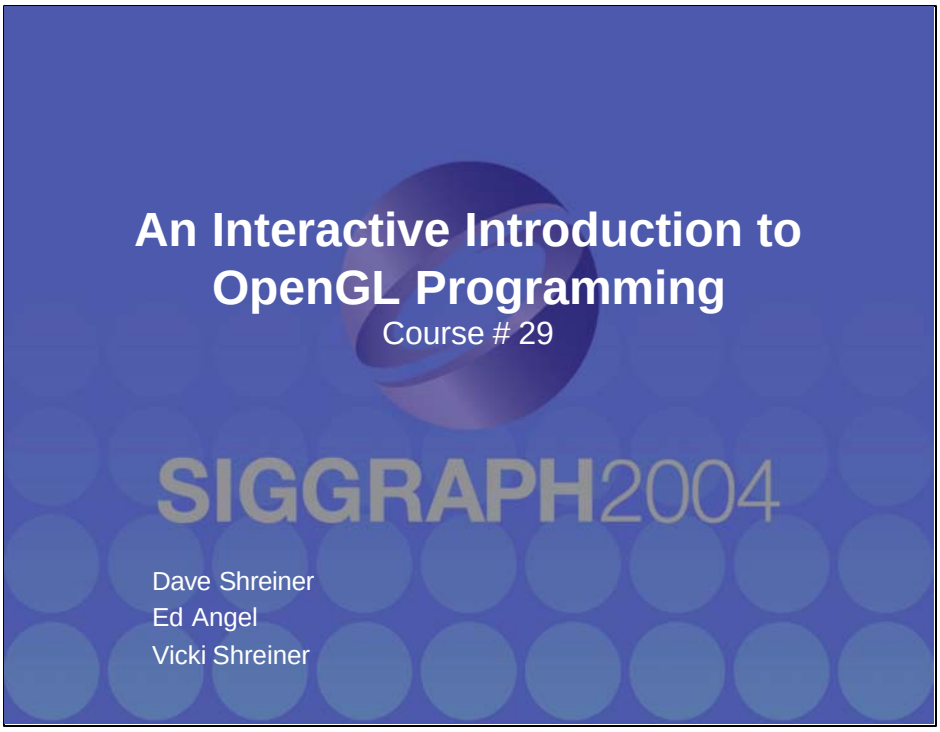
  - how our virtual eye views our scene
  - position, size, and orientation of the objects in our scene
  - the creation of a complex model from the hierarchical placement of its components and suitable transformations (think about a car; each wheel is basically the same, just positioned at different points on the chassis. We'll use transformations to put all the things in the right places, and make the entire car move as a single unit).

- *Lighting* – simulate how light illuminates the surface of our objects in our scene. In nature, what we see is the result of light reflecting off of our surroundings, and entering our eye. These interactions are quite complicated, and for an interactive program, are too computationally intensive to be completely accurate. OpenGL uses a simplified lighting model to create reasonable lighting effects that usually suffice for interactive applications. One point that generally surprises novices to OpenGL is that shadows are not supported. Shadows require considerable knowledge of the scene and the placement of the objects, which is data that's not generally available to OpenGL while it's drawing. This may seem counter-intuitive; however, OpenGL processes each primitive in isolation. Techniques that add shadows into an OpenGL scene have been developed. Any of the texts in the bibliography will contain information on the topic.
- *Depth buffering* – determines which geometric primitives are closest to the eye. We take for granted that when an object is behind another object, the one farthest from our eye is obscured. Since OpenGL doesn't enforce a rendering order for the primitives you ask it to render, depth buffering is used to determine visibility of objects in your scene.
- *Double buffering* – One of the principle goals of the course is to help you develop interactive graphics programs. To move objects around in your scene, you will have to draw the scene multiple times, moving objects the appropriate amount each time. In general this approach starts with a "clean slate" for each frame, which requires OpenGL to initialize its framebuffer at the start of each frame. This initialization is generally done by setting all of the pixels to the same color; however, doing this causes the animation to "flicker." Double buffering is a technique to remove the flickering from our sequence of frames, and provide a smooth interactive experience.

- *Texture mapping* – this technique allows for geometric models, perhaps composed of just a few polygons, to have much higher color fidelity. This is accomplished by determining the color of the pixels filled by our geometric primitives not only by the color of that primitive, but from a texture image. The texture contains much more color information, and OpenGL knows how to extract the colors to produce a much richer picture than the standard method for shading a polygon. Texture mapping enables a wide variety of techniques that would otherwise be prohibitively complex to do.

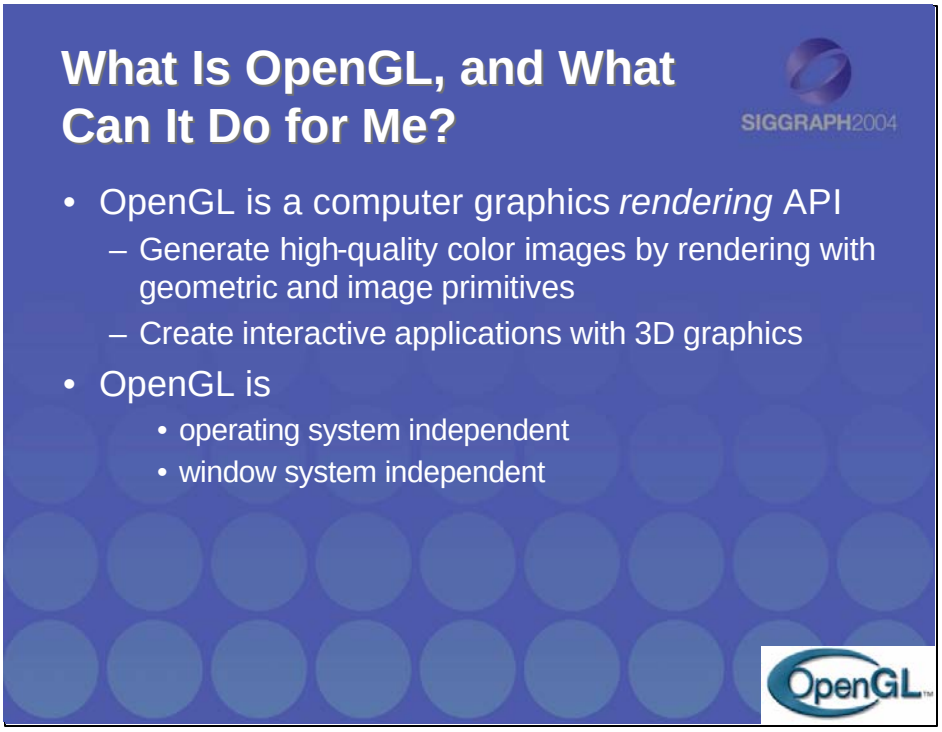
Presentation Course Notes

The following pages include the slides presented at SIGGRAPH 2004, as well as notes to aid in use of the slides after the talk.


An Interactive Introduction to
OpenGL Programming
Course # 29
SIGGRAPH2004
Dave Shreiner
Ed Angel
Vicki Shreiner

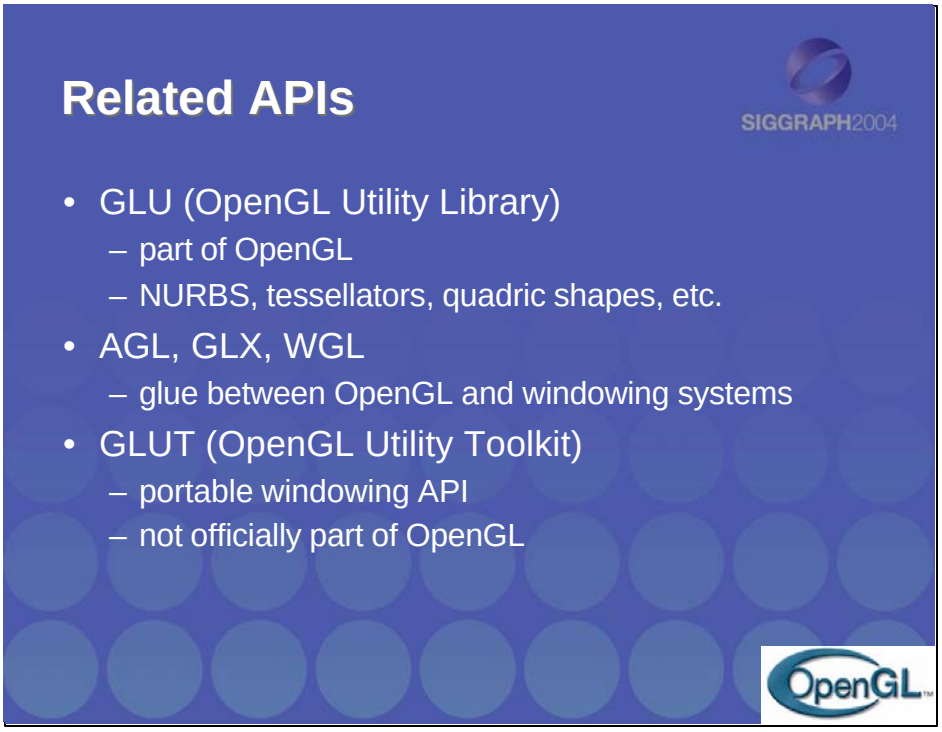

SIGGRAPH2004
Welcome


Welcome
SIGGRAPH2004
• Today's Goals and Agenda
– Describe OpenGL and its uses
– Demonstrate and describe OpenGL's capabilities and features
– Enable you to write an interactive, 3-D computer graphics program in OpenGL
OpenGL



OpenGL is a library for drawing, or *rendering*, computer graphics. By using OpenGL, 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 that 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.

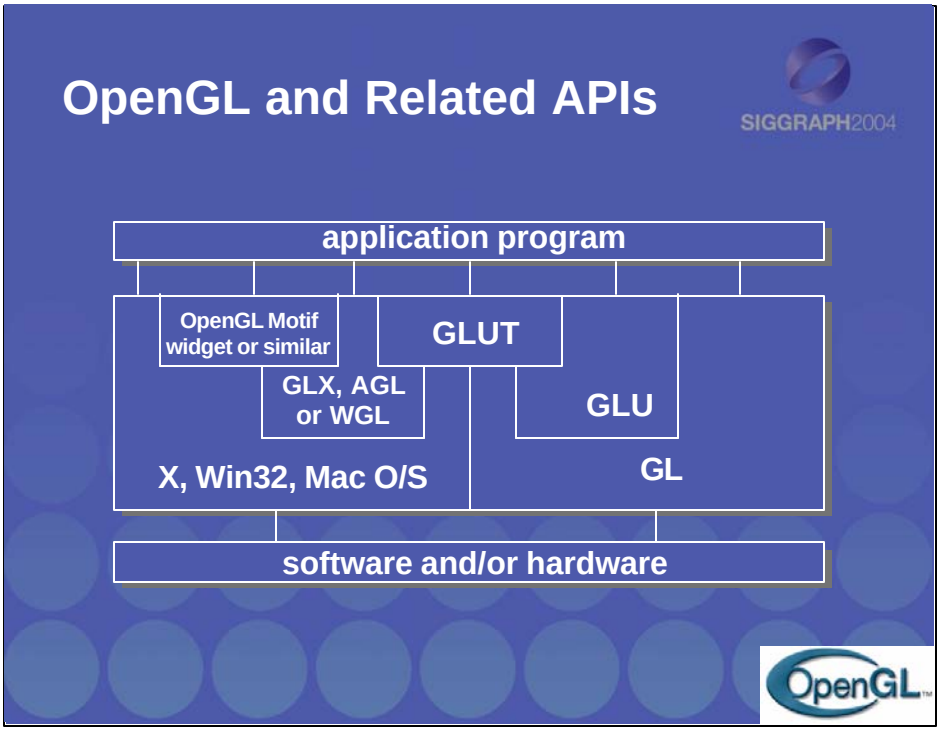


As mentioned, OpenGL is window and operating system independent. To integrate it into various window systems, additional libraries are used to modify a native window into an OpenGL capable window. Every window system has its own unique library and functions to do this. Some examples are:

- `GLX` for the X Windows system, common on Unix platforms
- `AGL` 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.

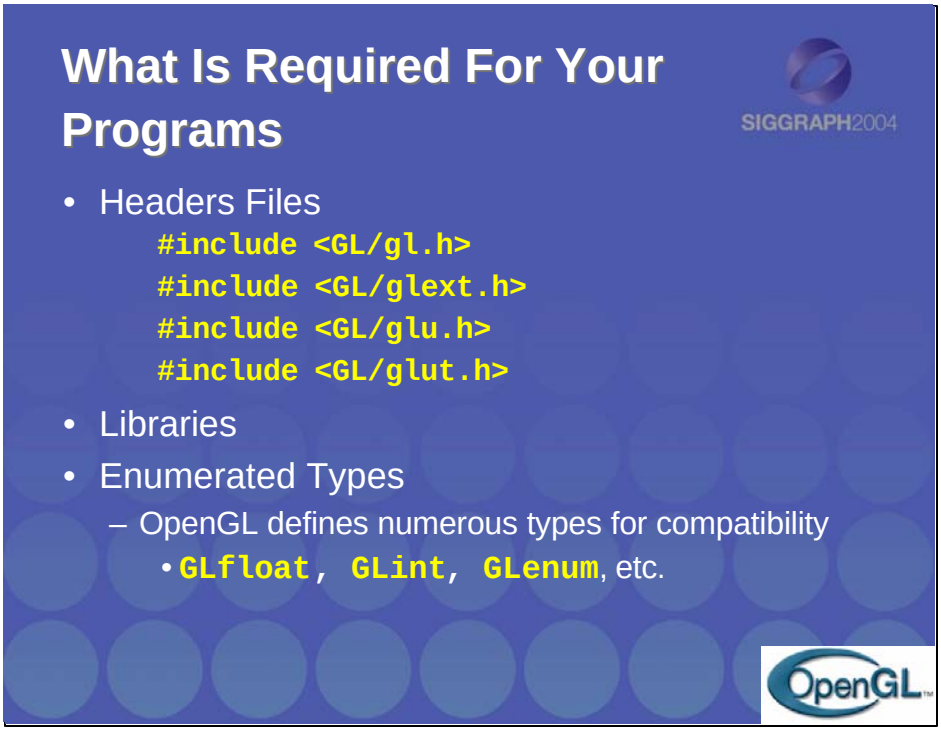
Finally to simplify programming and window system dependence, we will be using the freeware library, GLUT. GLUT, written by Mark Kilgard, is a public domain window system independent toolkit for making simple OpenGL applications. GLUT simplifies the process of creating windows, working with events in the window system and handling animation.



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.



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

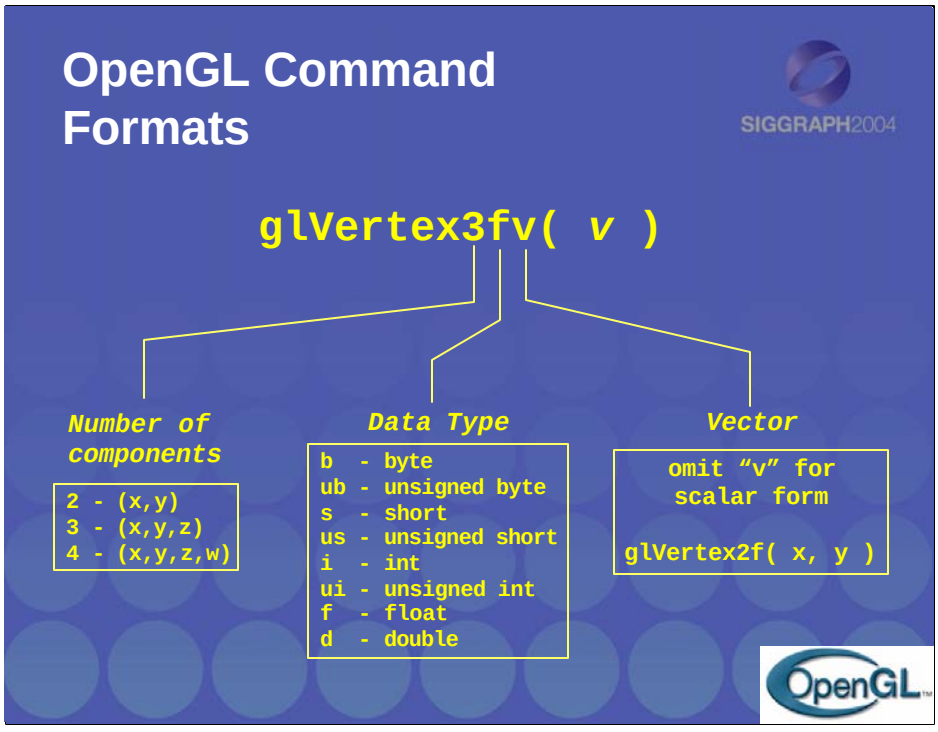
For C, there are a few required elements which an application must do:

- *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.

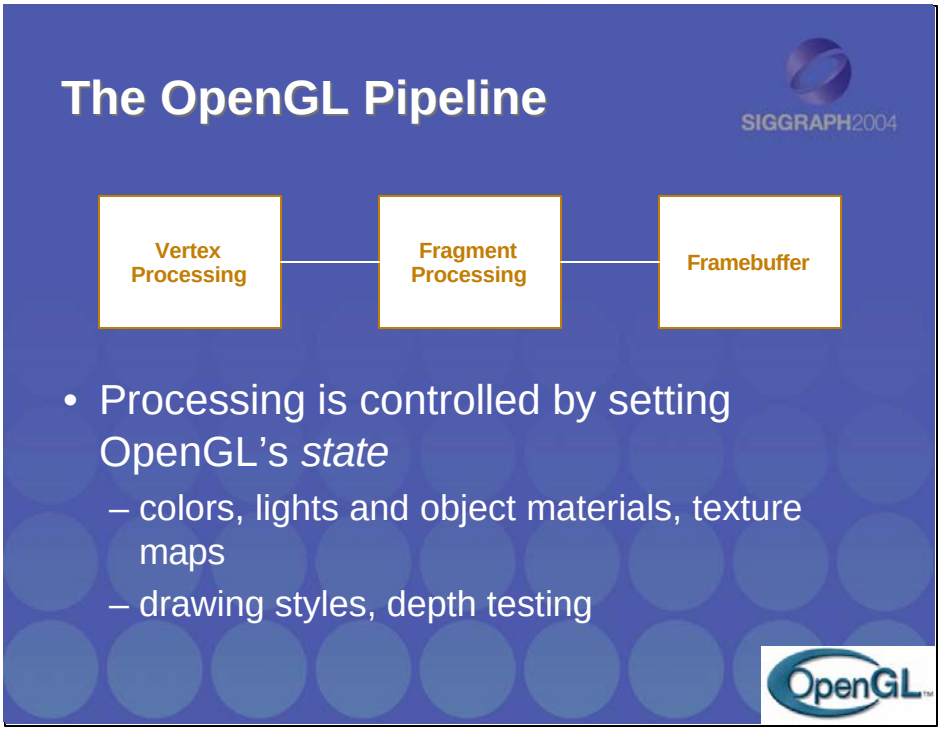


The OpenGL API calls are designed to accept almost any basic data type, which is reflected in the calls name. Knowing how the call names are structured makes it easy to determine which call should be used for a particular data format and size.

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`*`)`.

OpenGL considers all points to be 3D. Even if you're drawing a simple 2D line plot, OpenGL considers each vertex to have an *x*-, *y*-, and a *z*-coordinate. In fact, OpenGL really uses *homogenous coordinates*, which are a set of four numbers (a 4-tuple) that is usually written as (*x*, *y*, *z*, *w*). The *w*-coordinate is there to simplify the matrix multiplication that we'll discuss in the Transformations section. You can safely ignore *w* for now.

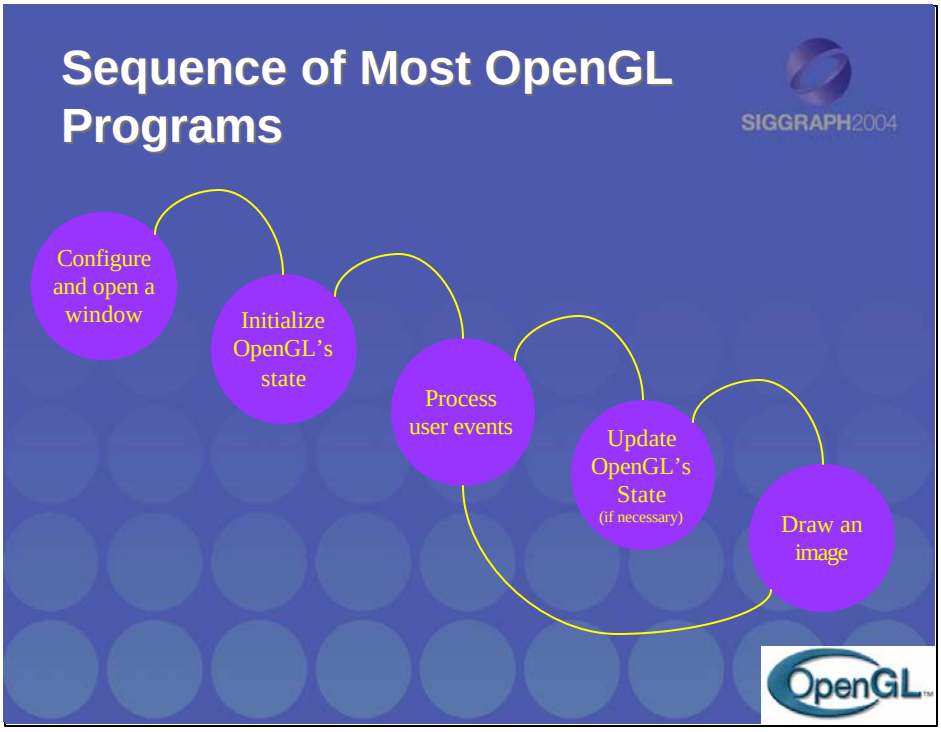
For `glVertex*()` calls which do not specify all the coordinates (i.e. `glVertex2f()`), OpenGL will default *z = 0.0*, and *w = 1.0* .



OpenGL is a *pipelined architecture*, which means that the order of operations is fixed. In general, OpenGL operations can be partitioned into two "processing units": vertex operations, and fragment operations.

The operation of each pipeline step is controlled by what's commonly referred to as *state*. State is just the collection of variables that OpenGL keeps track of internally. They include colors, positions, texture maps, etc. We'll discuss many of these state groups during the course. Setting state comprises about 80% of the OpenGL functions in the library.


SIGGRAPH2004
An Example OpenGL Program



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 OpenGL application 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 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.

An OpenGL Program

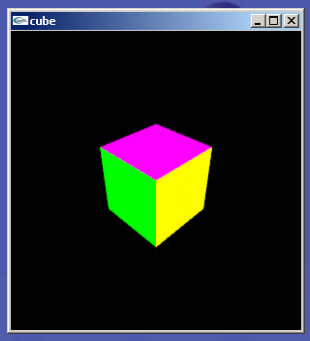
```
#include <GL/glut.h>
#include "cube.h"

void main( int argc, char *argv[] )
{
  glutInit( &argc, argv );
  glutInitDisplayMode( GLUT_RGBA |
                       GLUT_DEPTH );
  glutCreateWindow( "cube" );

  init();

  glutDisplayFunc( display );
  glutReshapeFunc( reshape );

  glutMainLoop();
}
```



The main part of the program. *GLUT* is used to open the OpenGL window, and handle input from the user.

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.

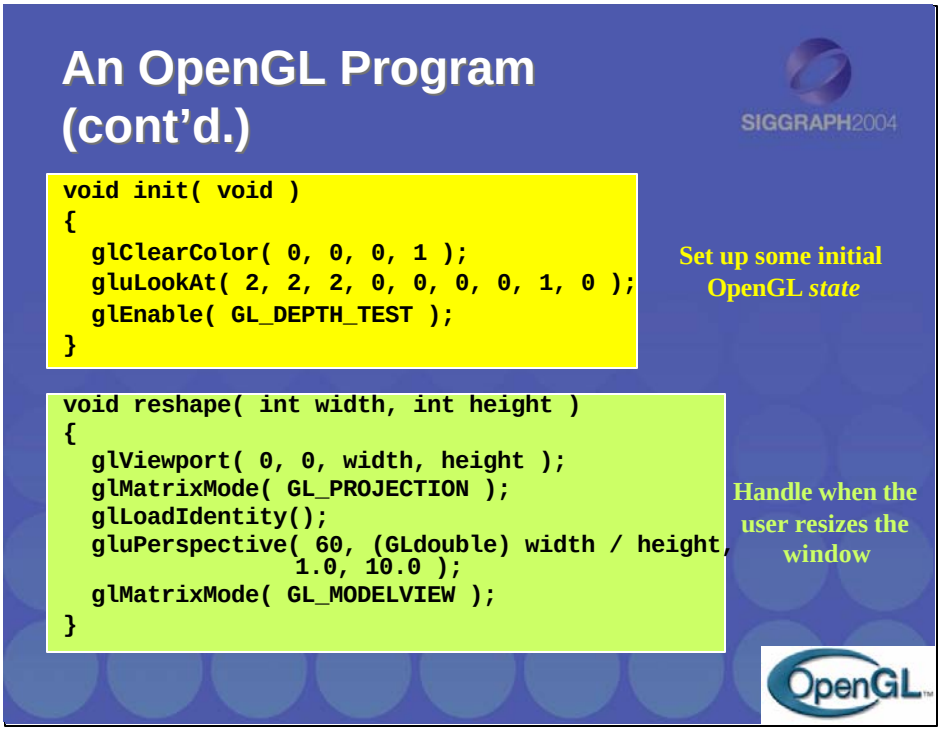
All GLUT-based OpenGL programs begin with configuring the GLUT window to be opened.

Next, in the routine `init()` (detailed on the following slide), we make OpenGL calls to set parameters that we'll use later in the `display()` function. These parameters, commonly called *state* , are values that OpenGL uses to determine how it will draw. There's nothing special about the `init()` routine, we just use it to logically separate the state that we need to set up only once (as compared to every frame).

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.

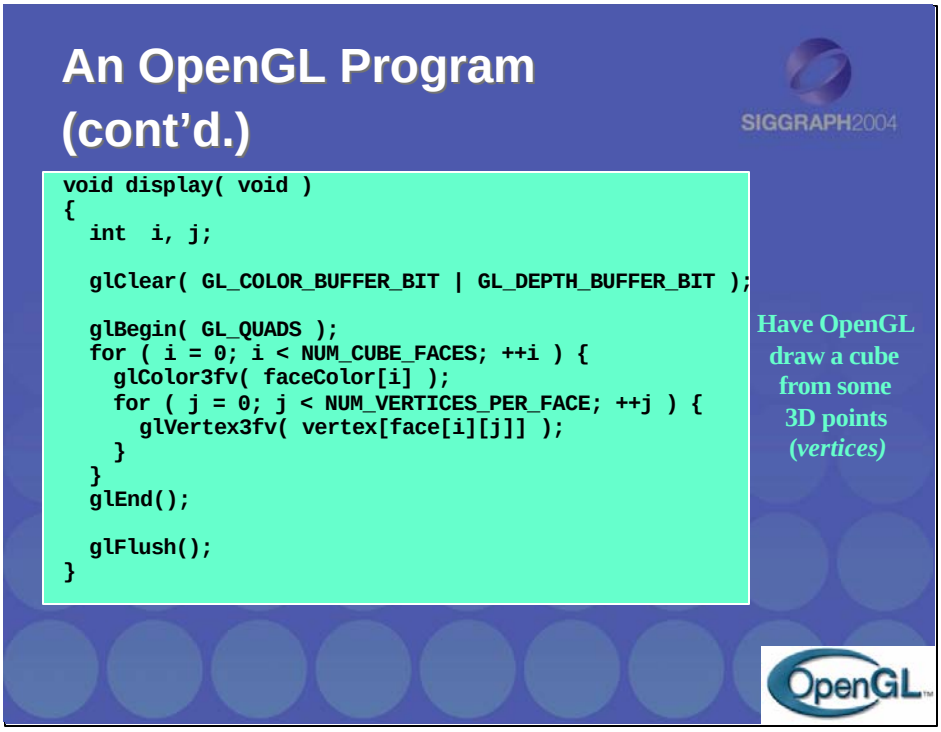
Finally, as with all interactive programs, the event loop is entered. For GLUT-based programs, this is done by calling `glutMainLoop()`. As `glutMainLoop()` never exits (it is essentially an infinite loop), any program statements that follow `glutMainLoop()` will never be executed.

The header file "`cube.h`" contains the geometric data (vertices, colors, etc.) for the cube model. Cubes are a very popular shape to render in computer graphics (along with teapots … we'll explain that one later), and is a nice example to work through to get a feel for *modeling* computer graphics objects. After the class, try to figure out the geometry for a cube. We've included our "`cube.h`" in the appendices of the notes for you to compare yours to.



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.



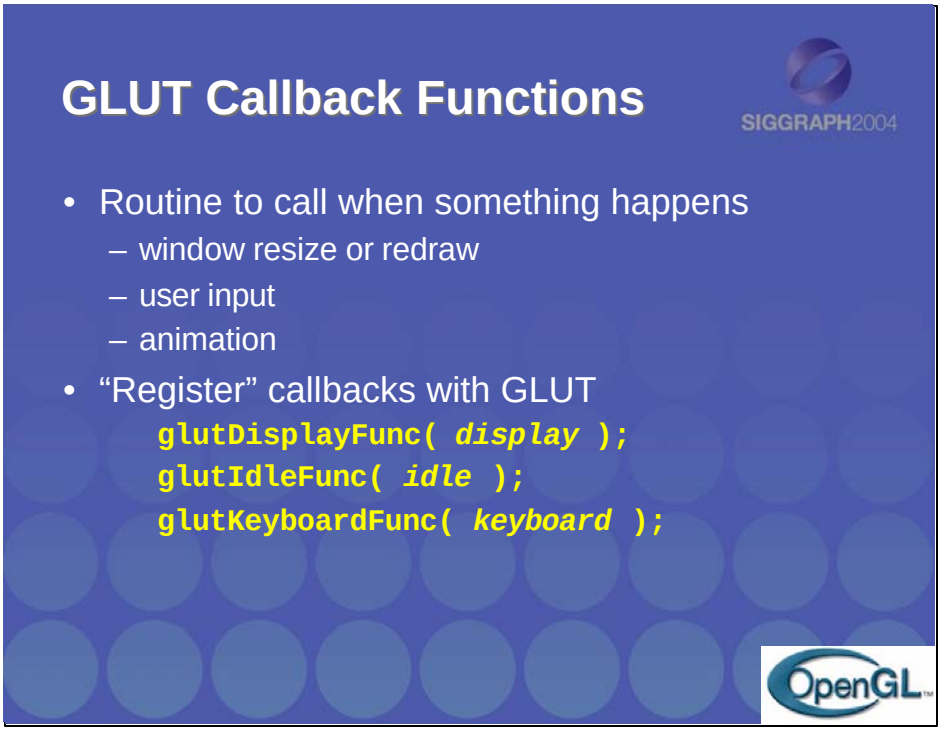
Finally, we see the `display()` routine which is used by GLUT to call our OpenGL calls to make our image. Almost all of your OpenGL drawing code should be called from `display()` (or routines that `display()` calls).

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.

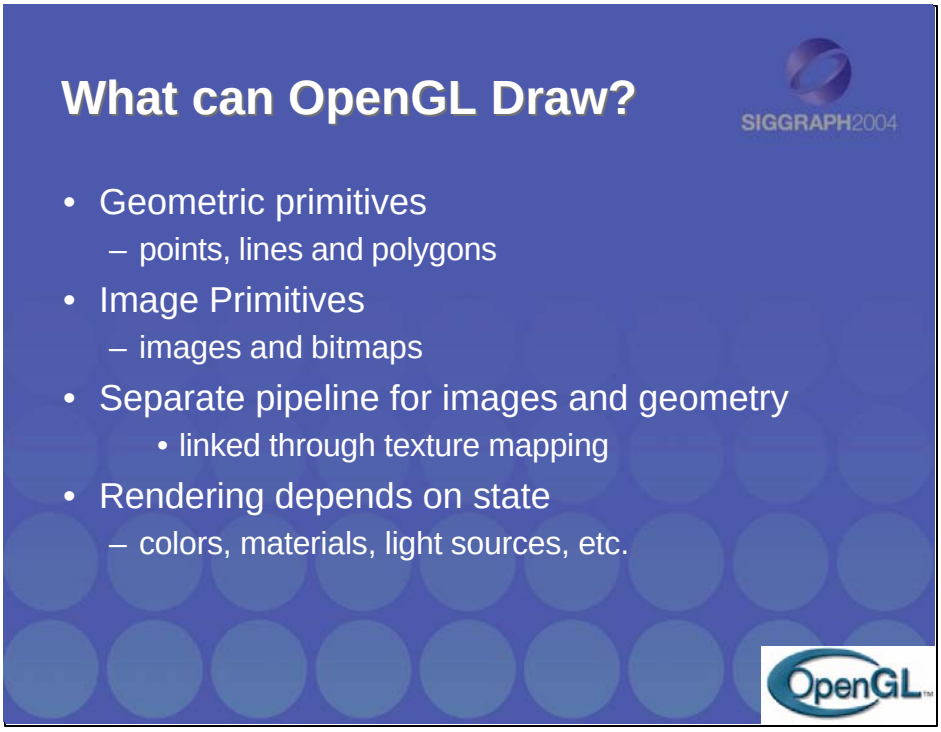


GLUT uses a *callback mechanism* to do its event processing. Callbacks simplify event processing for the application developer. As compared to more traditional event driven programming, where the author must receive and process each event, and call whatever actions are necessary, callbacks simplify the 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.


SIGGRAPH2004
Drawing with OpenGL

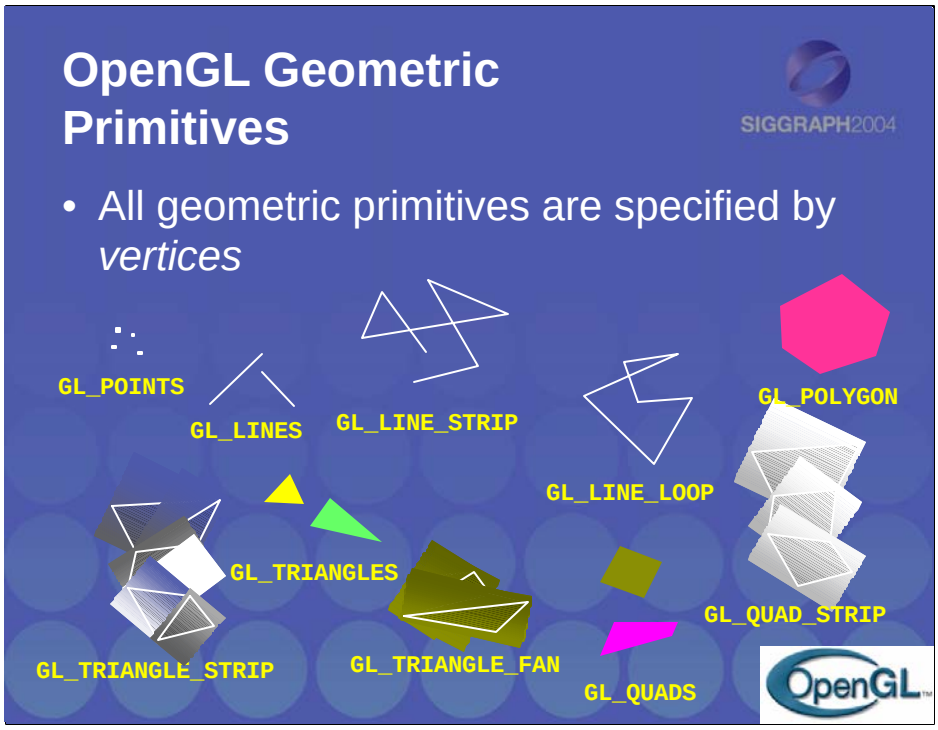


As mentioned, OpenGL is a library for rendering computer graphics. Generally, there are two operations that you do with OpenGL:

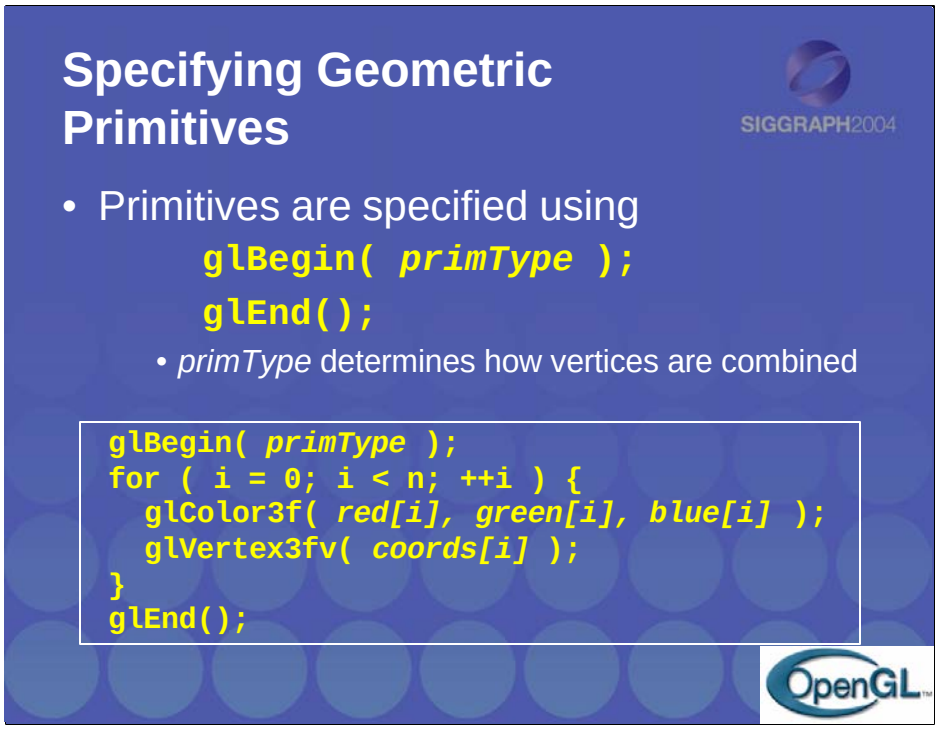
- draw something
- change the state of how OpenGL draws

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.



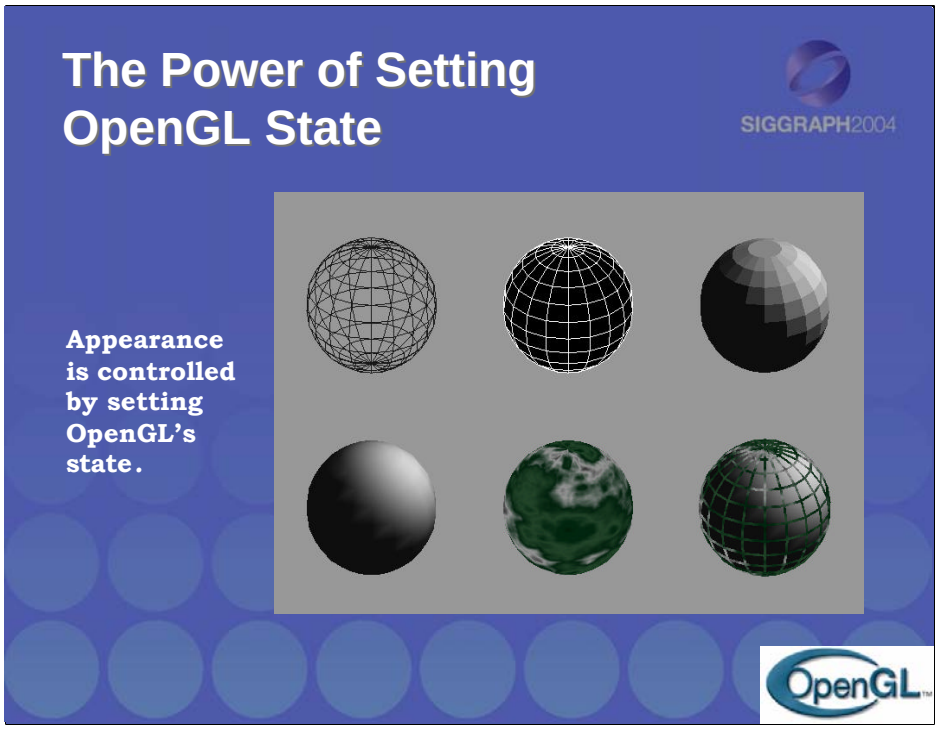
Every OpenGL geometric primitive is specified by its vertices, which are *homogenous coordinates*. Homogenous coordinates are of the form (x, y, z, w). Depending on how vertices are organized, OpenGL can render any of the above primitives.



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`

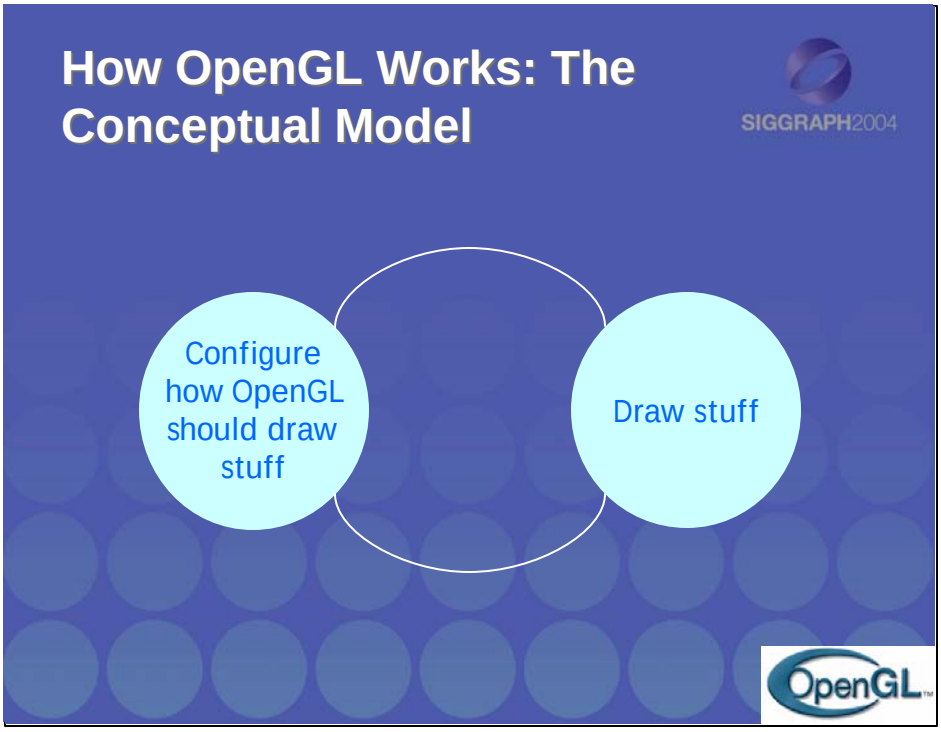
We also see an example of setting OpenGL's state, which is the topic of the next few slides, and most of the course. In this case, the color that our primitive is going to be drawn is set using the `glColor()` call.



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 from left to right 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.

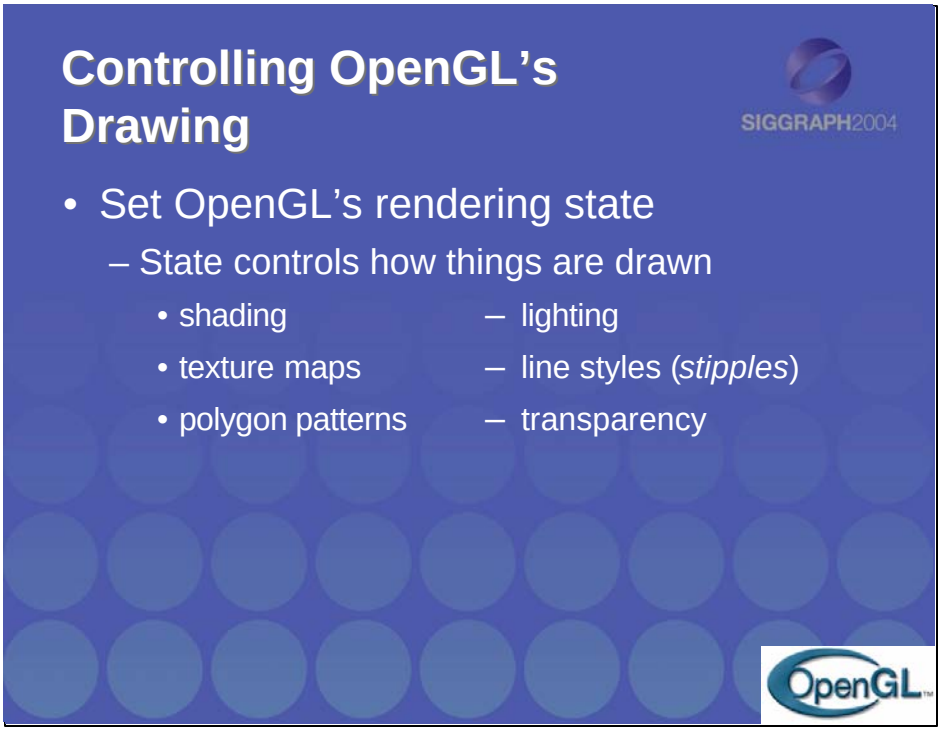
For the bottom row (left to right), the first image is the same sphere, only this time, *gouraud- (or smooth-) shaded*. The only difference in the programs between the top-row right, and bottom-row left is a single line of OpenGL code. The middle sphere was generated using texture mapping. The final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



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.

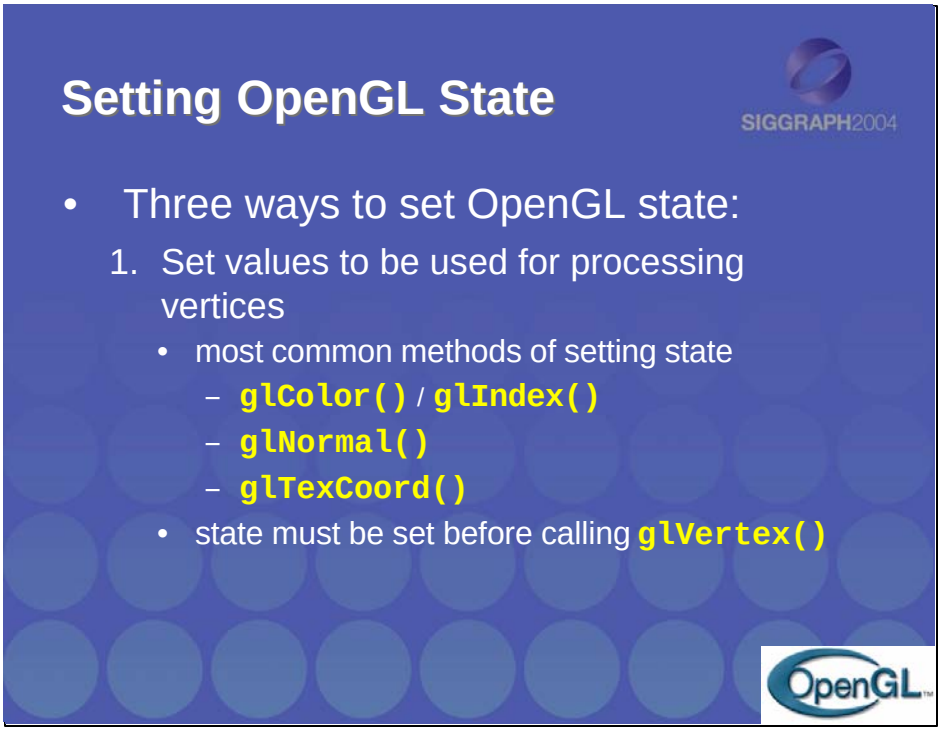
Your application will consist of cycles of setting state, and rendering using the state that you just set.



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.

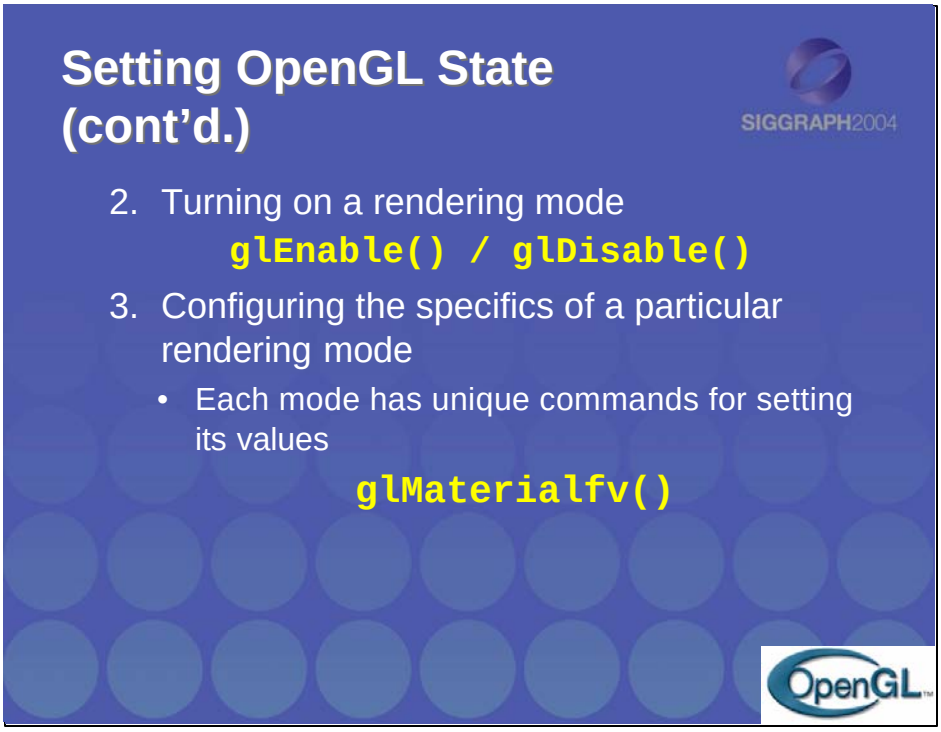
OpenGL has over 400 function calls in it, most of which are concerned with setting the rendering state. Among the things that state controls are:

- 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



There are three ways to set OpenGL state.

The first, as detailed here, is to directly set parameters that OpenGL will use in processing vertices. This includes setting colors, lighting normals, and texture coordinates. These values will not change (under most circumstances) until the next time you specify data. In some cases, this every vertex will have its own unique set of these values, and the data will change with each vertex. In other cases, values may remain constant across the entire execution of a program.

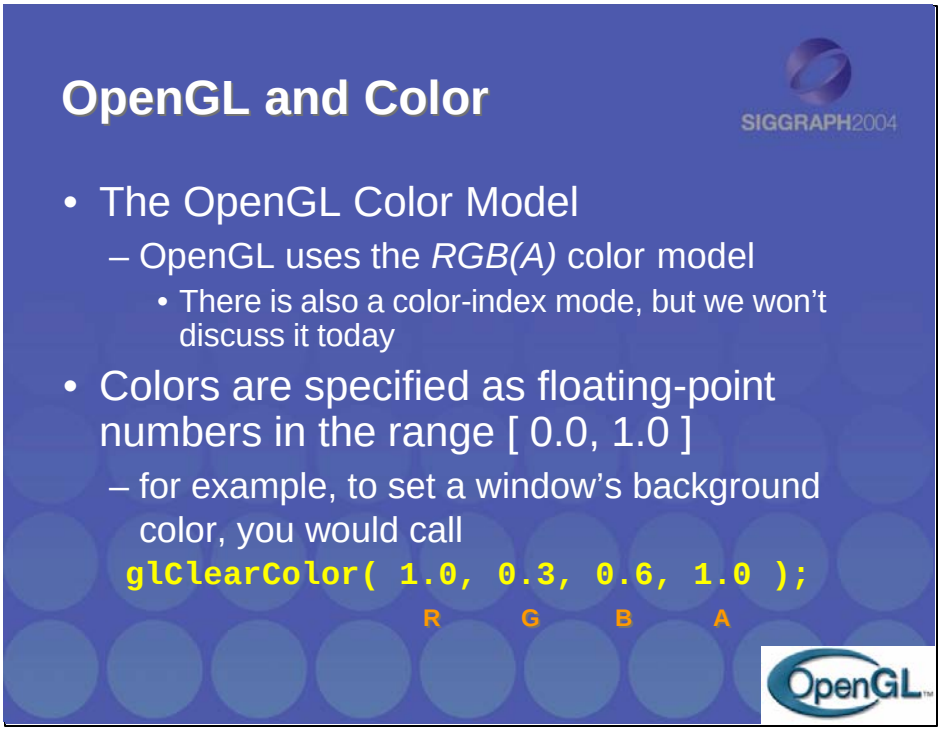


There are two actions that are required to control how OpenGL renders.

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.

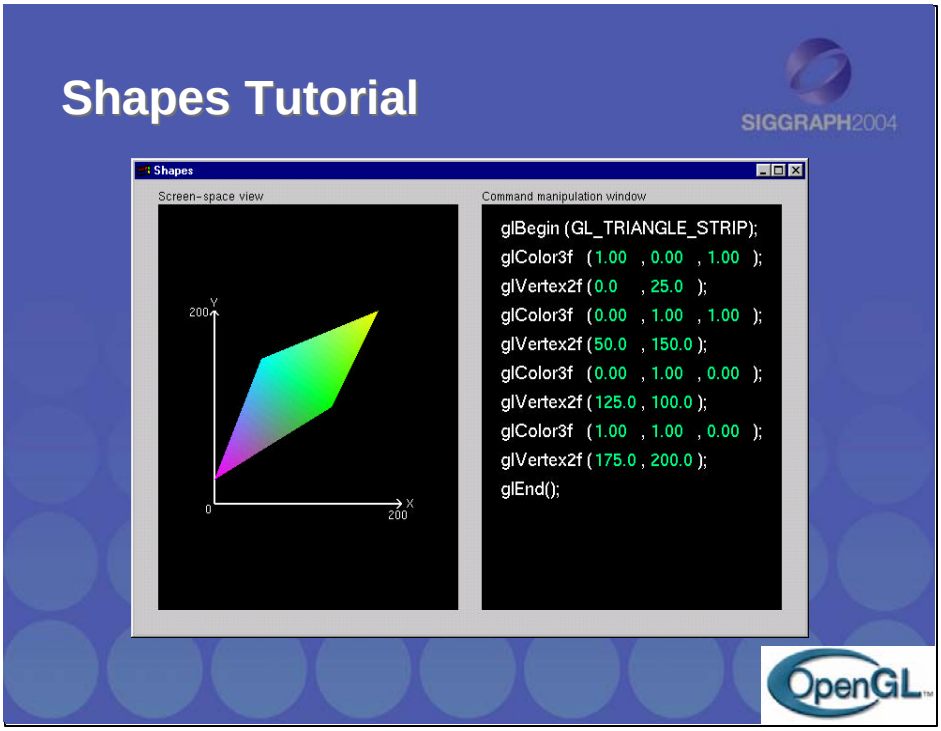
It's important to note that initial state is identical for every OpenGL implementation, regardless of which operating system, or which hardware system you are working on.



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.



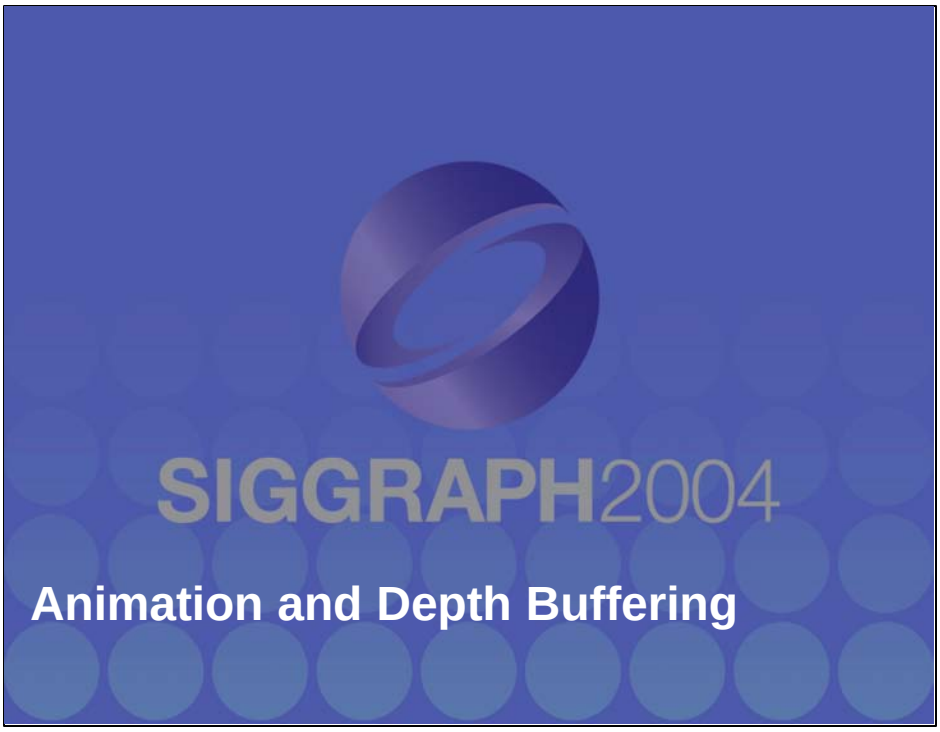
This is the first of the series of Nate Robins' tutorials. This tutorial illustrates the principles of rendering geometry, specifying both colors and vertices.

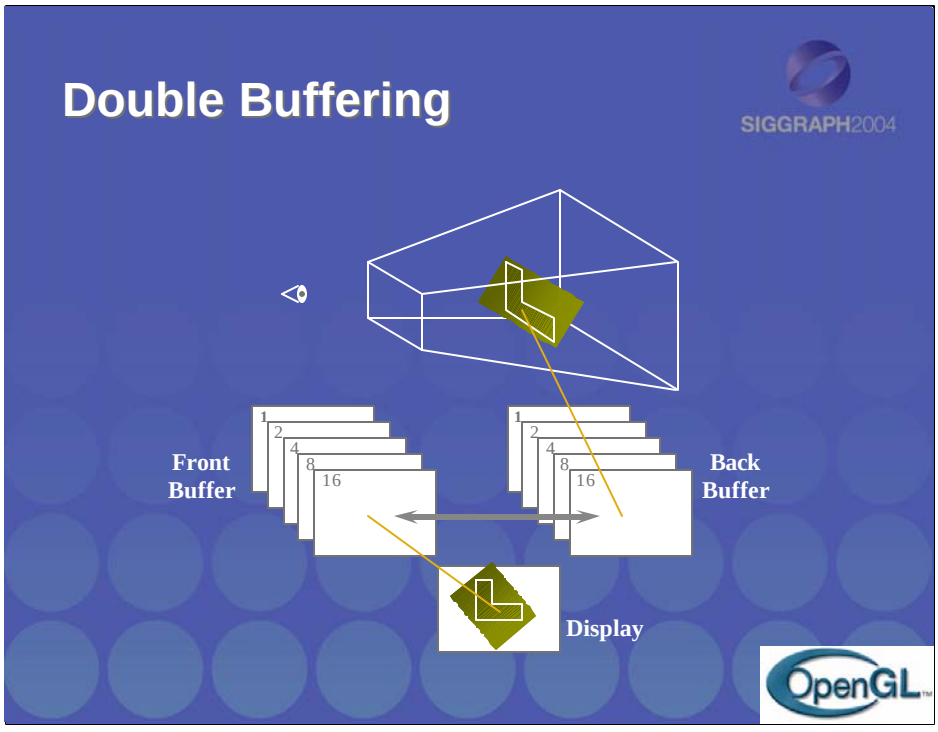
The shapes tutorial has two views: a screen-space window and a command manipulation window.

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.

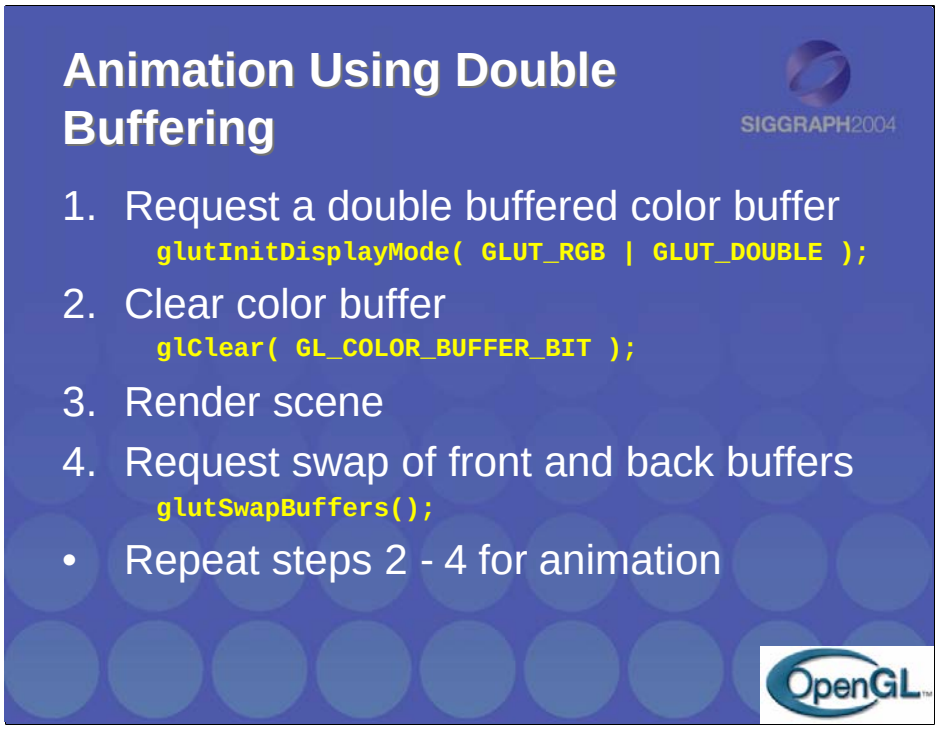
The left and right mouse buttons will do similar operations in the other tutorials.


SIGGRAPH2004
Animation and Depth Buffering



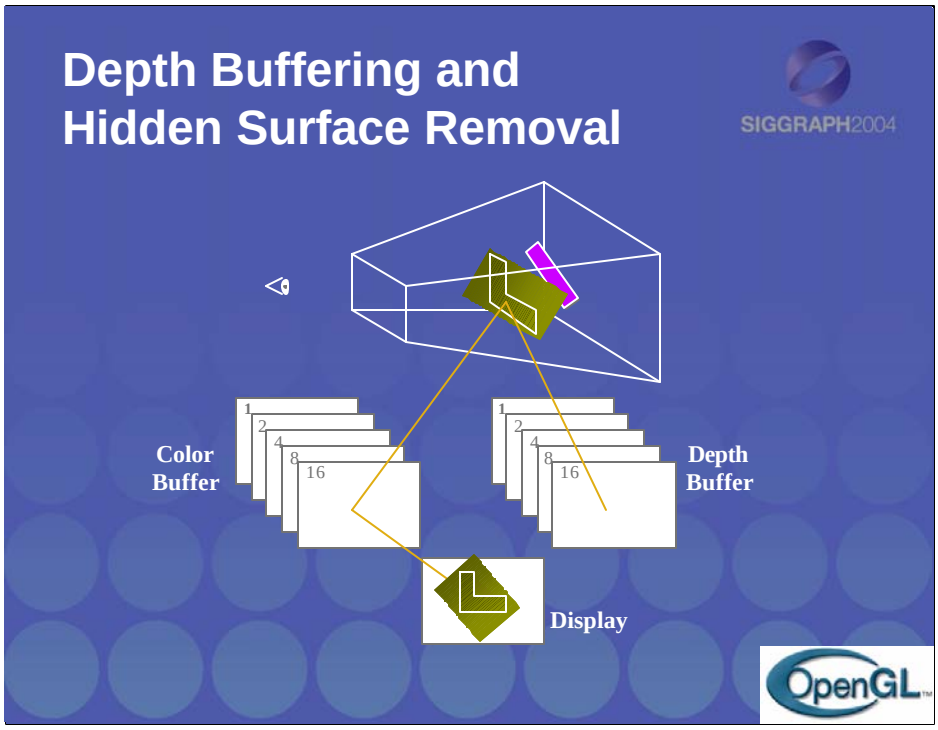
Double buffer is a technique for tricking the eye into seeing smooth animation of rendered scenes. The color buffer is usually divided into two equal halves, called the *front buffer* and the *back buffer*.

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.



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. The glutSwapBuffers() call is part of the GLUT library; if you use your operating system's native windowing system to do OpenGL, you will use a different function than glutSwapBuffers() to do a buffer swap. Each windowing system has it's own call for doing a swap buffers operation (for GLX, glXSwapBuffers; for WGL, wglSwapBuffers; etc.)

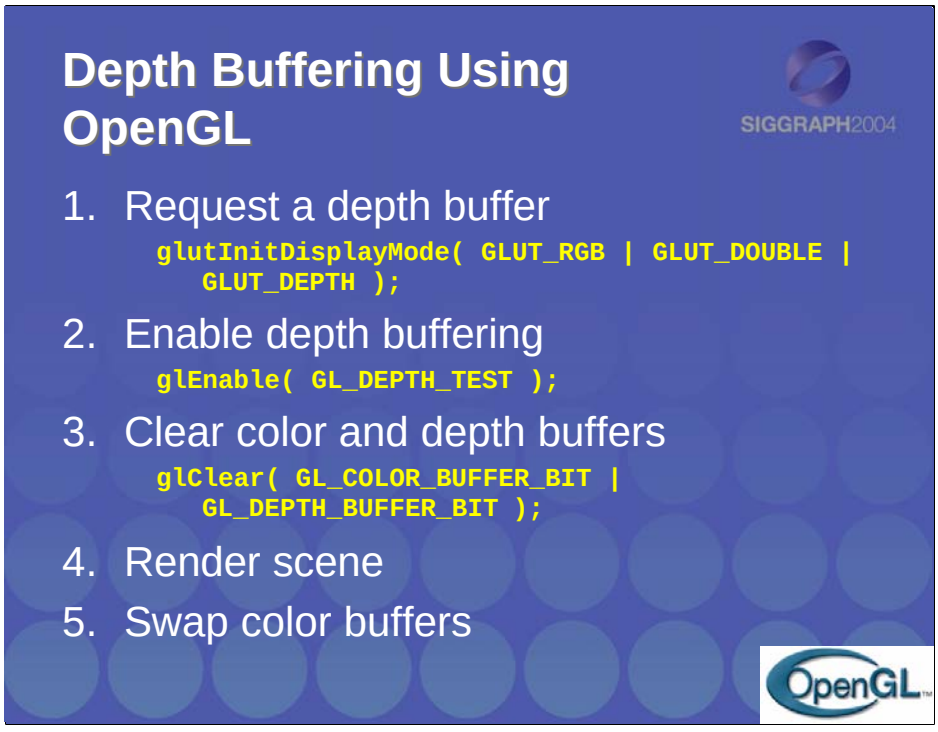


Depth buffering is a technique to determine which primitives in your scene are occluded by other primitives. As each pixel in a primitive is rasterized, its distance from the eyepoint (depth value), is compared with the values stored in the depth buffer. If the pixel's depth value is less than the stored 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.

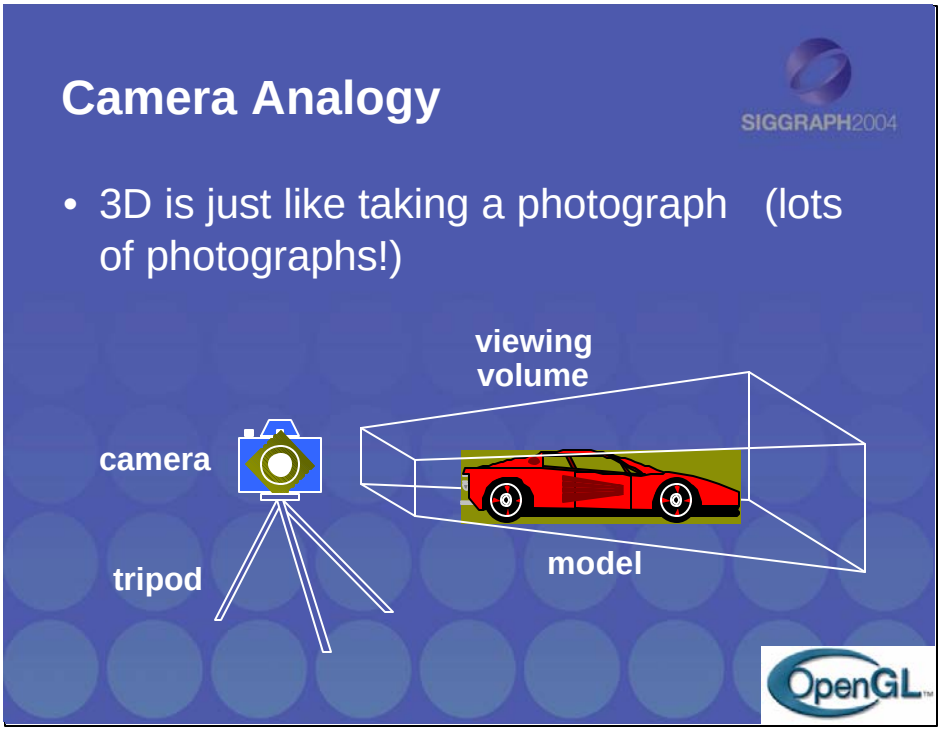


Enabling depth testing in OpenGL is very straightforward.

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

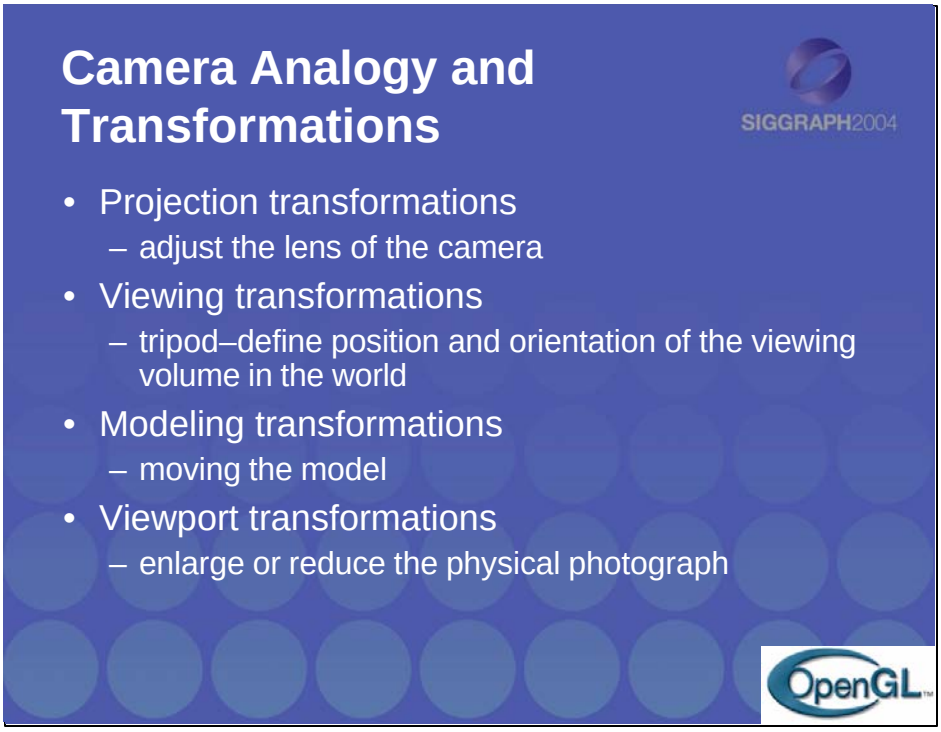
Once the window is created, the depth test is enabled using `glEnable(GL_DEPTH_TEST)`.


SIGGRAPH2004
Transformations

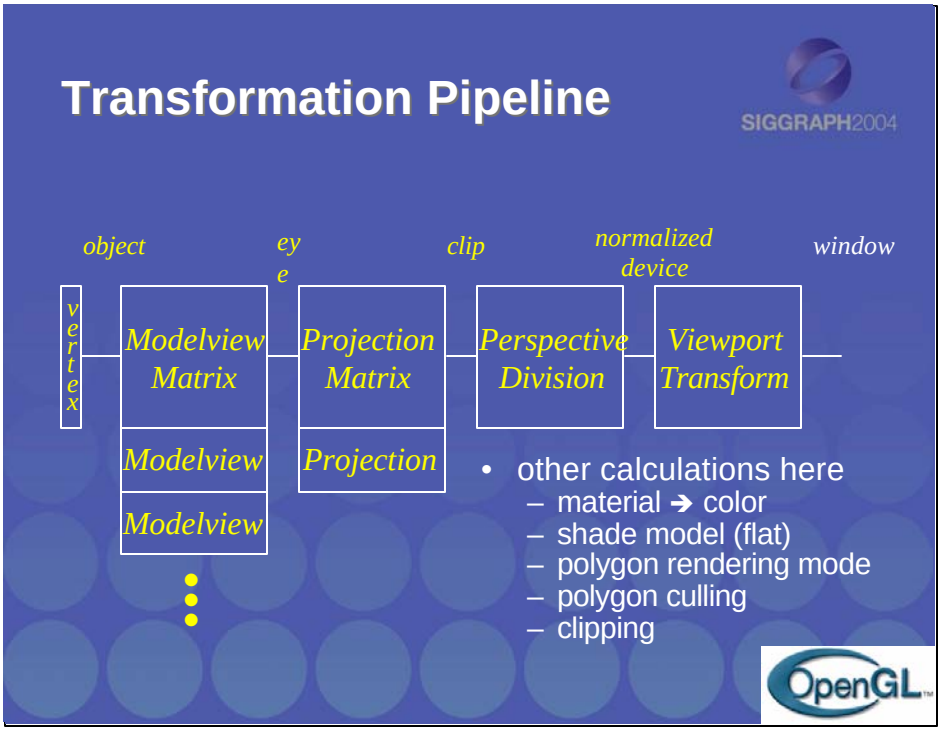


This model has become known as the “synthetic camera model”.

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



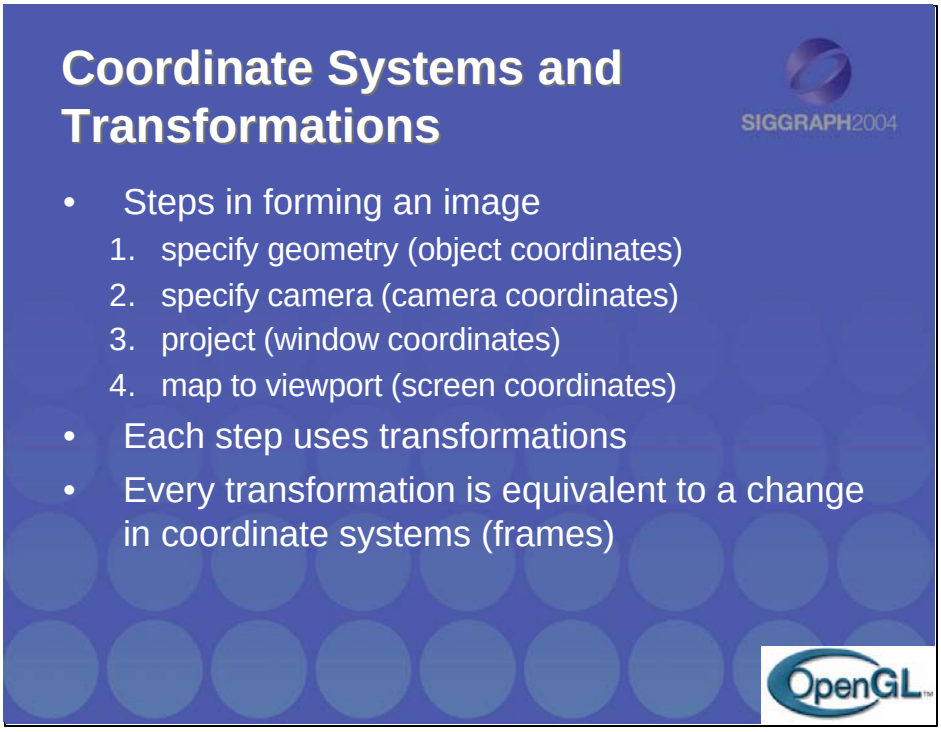
Note that human vision and a camera lens have cone-shaped viewing volumes. OpenGL (and almost all computer graphics APIs) describe a pyramid-shaped viewing volume. Therefore, the computer will "see" differently from the natural viewpoints, especially along the edges of viewing volumes. This is particularly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



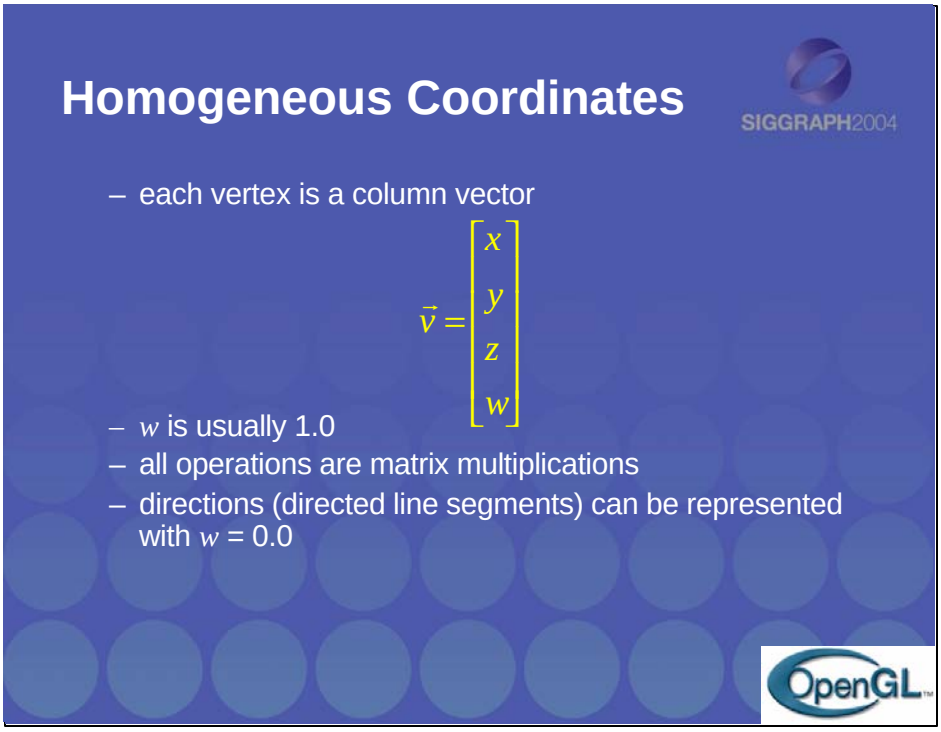
The depth of matrix stacks are implementation-dependent, but the model-view matrix stack is guaranteed to be at least 32 matrices deep, and the Projection matrix stack is guaranteed to be at least 2 matrices deep.

The material-to-color, flat-shading, and clipping calculations take place after the model-view 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.



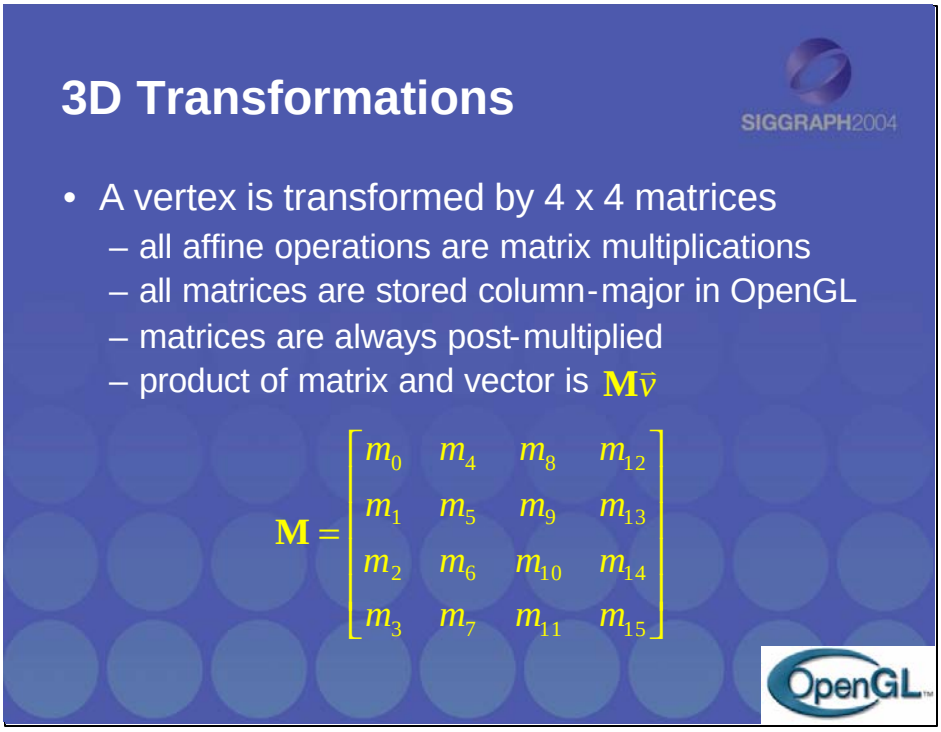
Every transformation can be thought of as changing the representation of a vertex from one coordinate system or frame to another. Thus, initially vertices are specified in object coordinates. However, to view them, OpenGL must convert these representations to ones in the reference system of the camera. 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.



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.

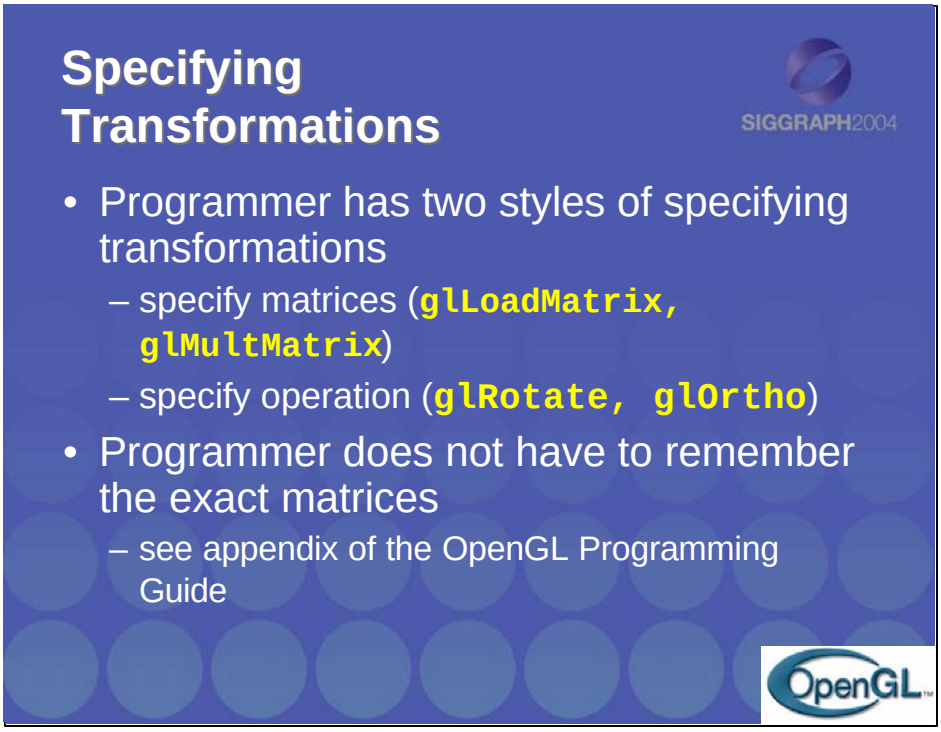


Perspective projection and translation require the 4th row and column, or operations would require addition, as well as multiplication.

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

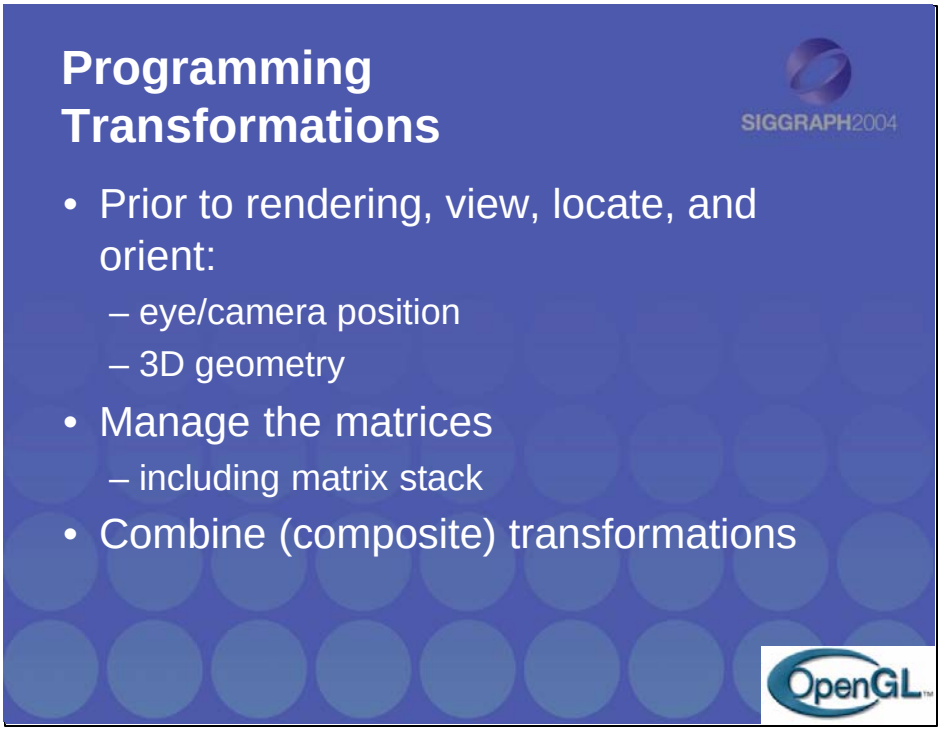
Because OpenGL only multiplies a matrix on the right, the programmer must remember that the last matrix specified is the first applied.

Recall that matrix multiplication is not commutative (i.e., for matrices *A*, and *B*, *AB* is not the same as *BA*). This holds true for OpenGL matrix operations. In particular, when you specify matrix transformations in OpenGL, the matrices accumulate in such a way that the last transformation you specify in your program is the first operation applied to a vertex. For example, if you specify a rotation and then a translation, the vertex is first rotated around the axis specified by the angle specified in your rotation transformation, and then translated by your translate operation. If you issued the rotation and translation in the reverse order, the translation would modify the vertex first, and then the rotation.



Generally, a programmer can obtain the desired matrix by a sequence of simple transformations that can be concatenated together, e.g. `glRotatef()`, `glTranslatef()`, and `glScalef()`.

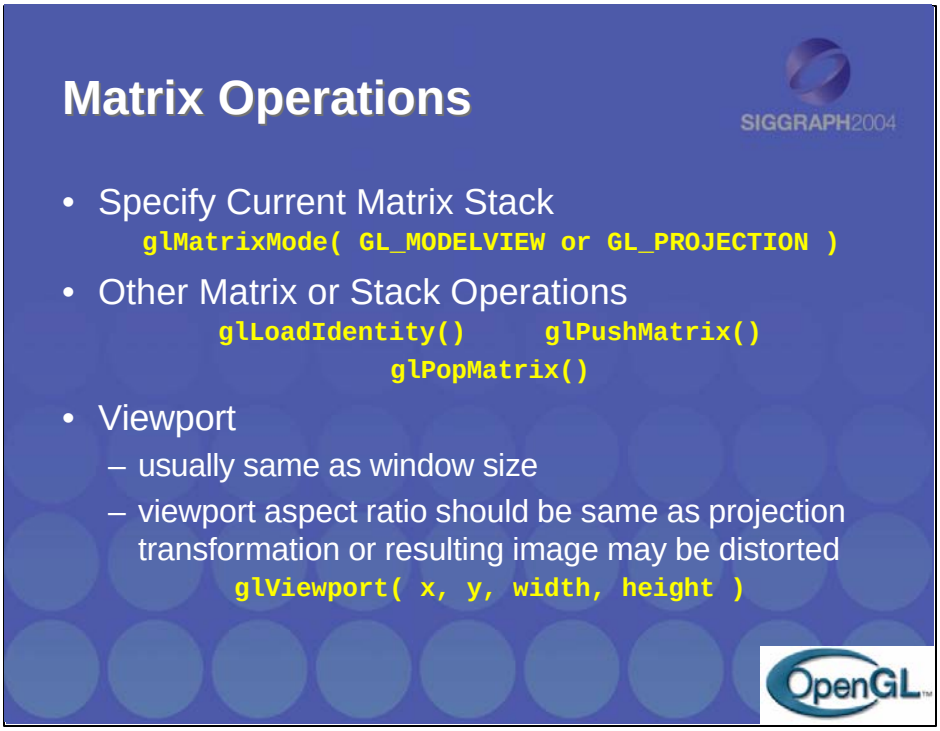
For the basic viewing transformations, OpenGL and the Utility library have supporting functions.



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.

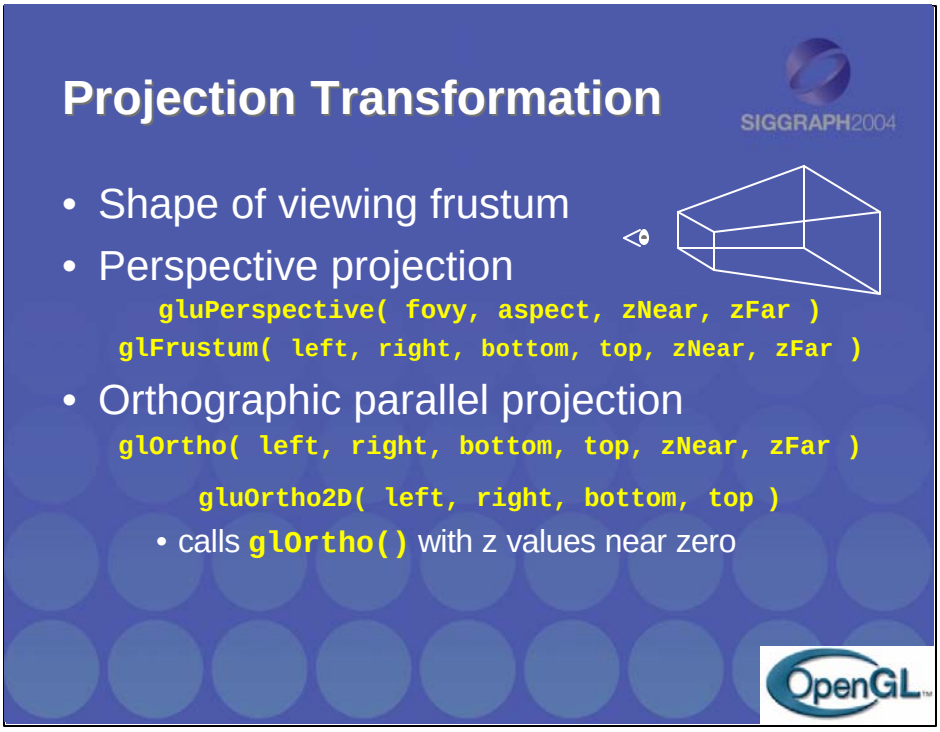


`glLoadMatrix*()` replaces the matrix on the top of the current matrix stack. `glMultMatrix*()`, post-multiples the matrix on the top of the current matrix stack. The matrix argument is a column-major 4 × 4 double or single precision floating point matrix.

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.



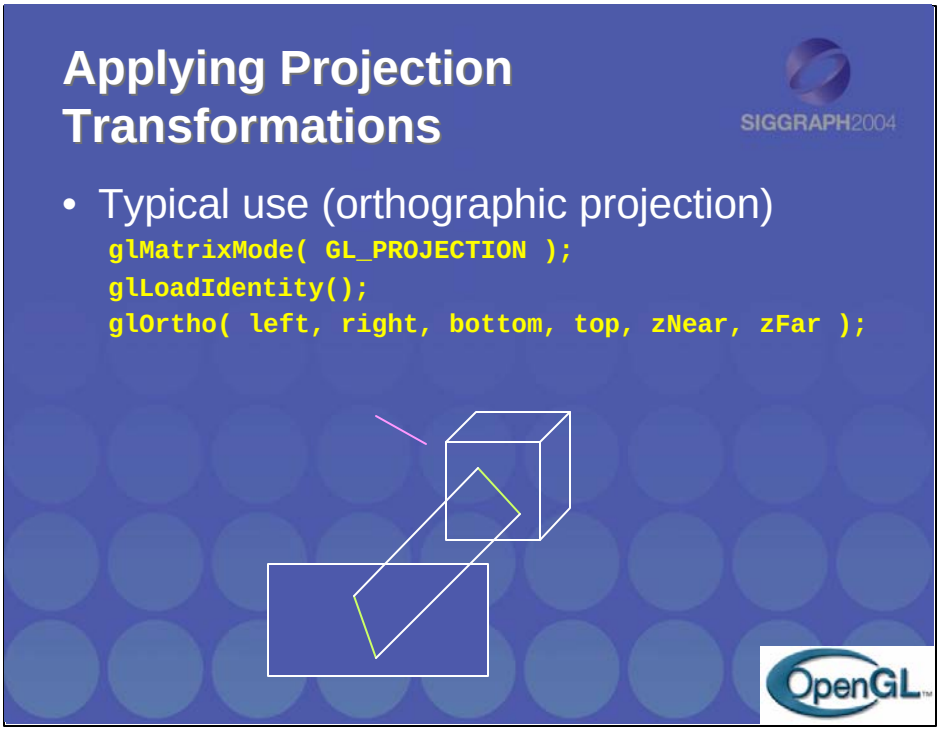
For perspective projections, the viewing volume is shaped like a truncated pyramid (frustum). There is a distinct camera (eye) position, and vertices of objects are "projected" to camera. Objects which are further from the camera appear smaller. The default camera position at (0, 0, 0), looks down the *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.

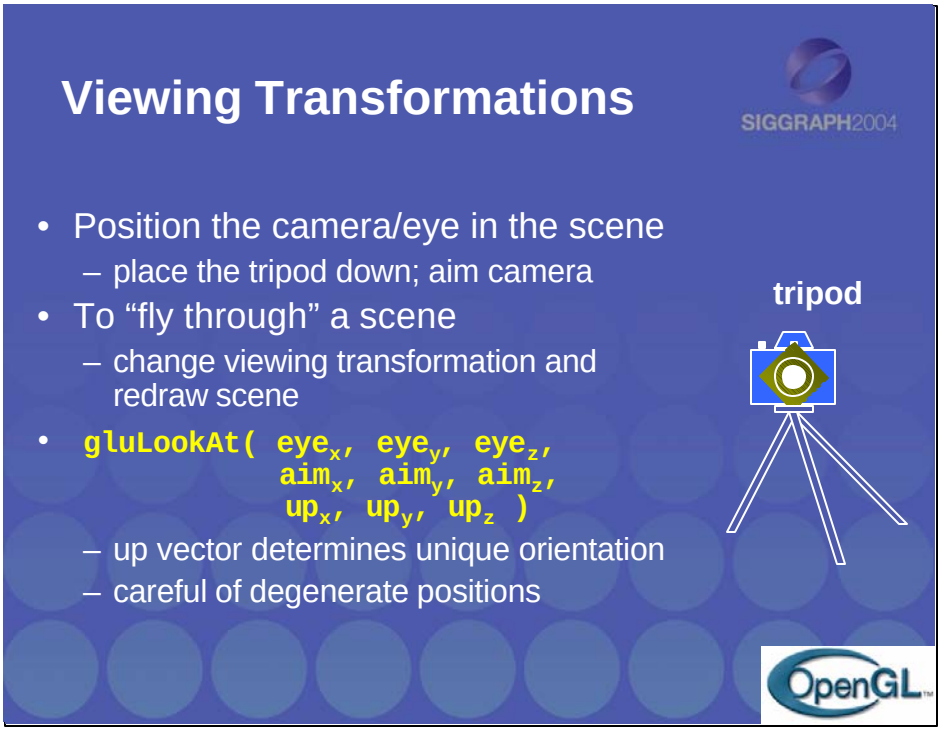
Warning: for `gluPerspective()` or `glFrustum()`, do not use zero for `zNear`!

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).



Many users would follow the demonstrated sequence of commands with a `glMatrixMode(GL_MODELVIEW)` call to return to model-view stack.

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

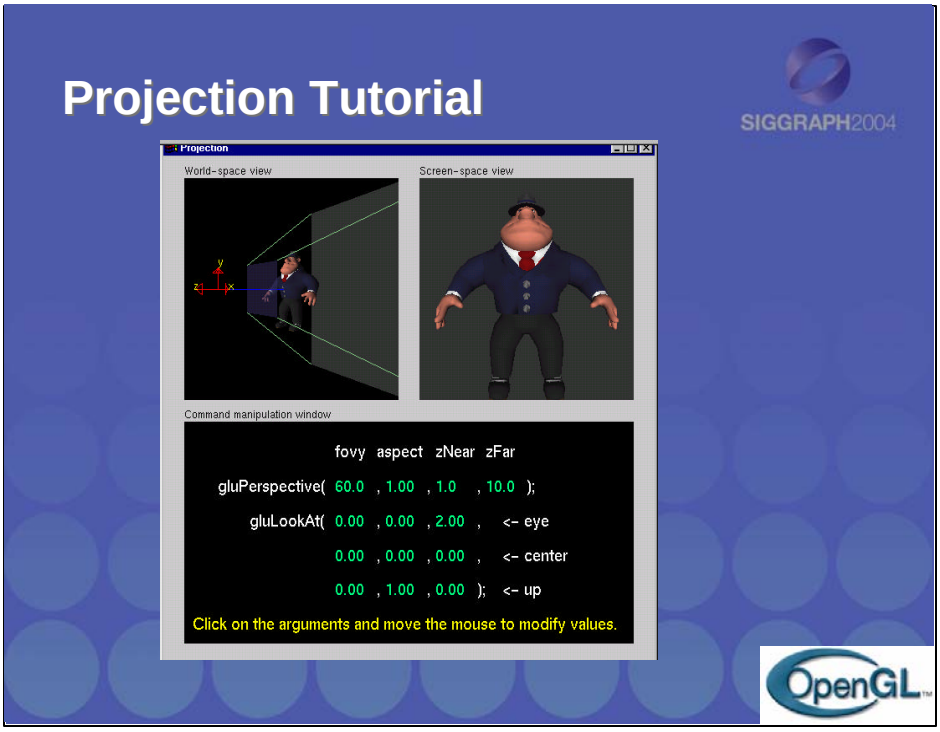


`gluLookAt()` multiplies itself onto the current matrix, so it usually comes after `glMatrixMode(GL_MODELVIEW)` and `glLoadIdentity()`.

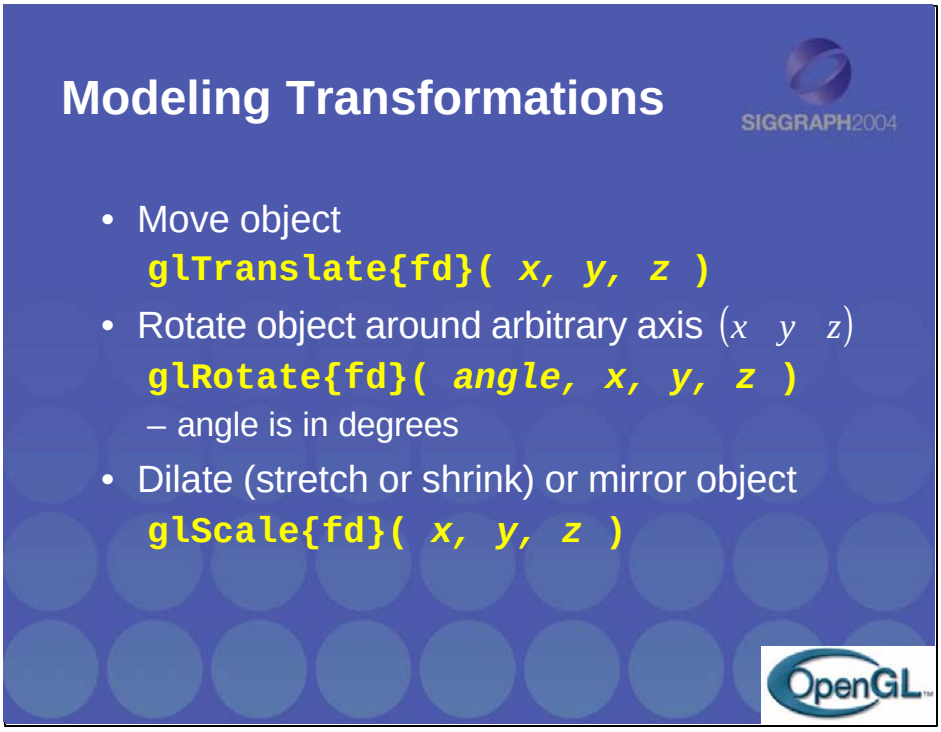
Because of degenerate positions, `gluLookAt()` is not recommended for most animated fly-over applications.

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

Note: that the name model-view 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.

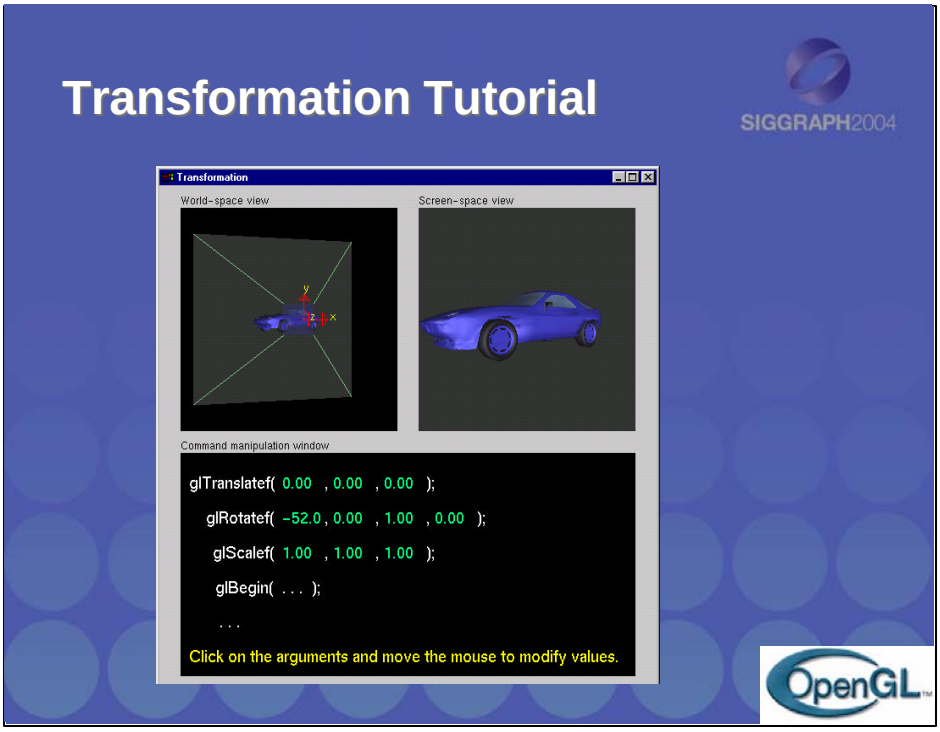


The RIGHT mouse button controls different menus. The screen-space view menu allows you to choose different models. The command-manipulation menu allows you to select different projection commands (including glOrtho and glFrustum).



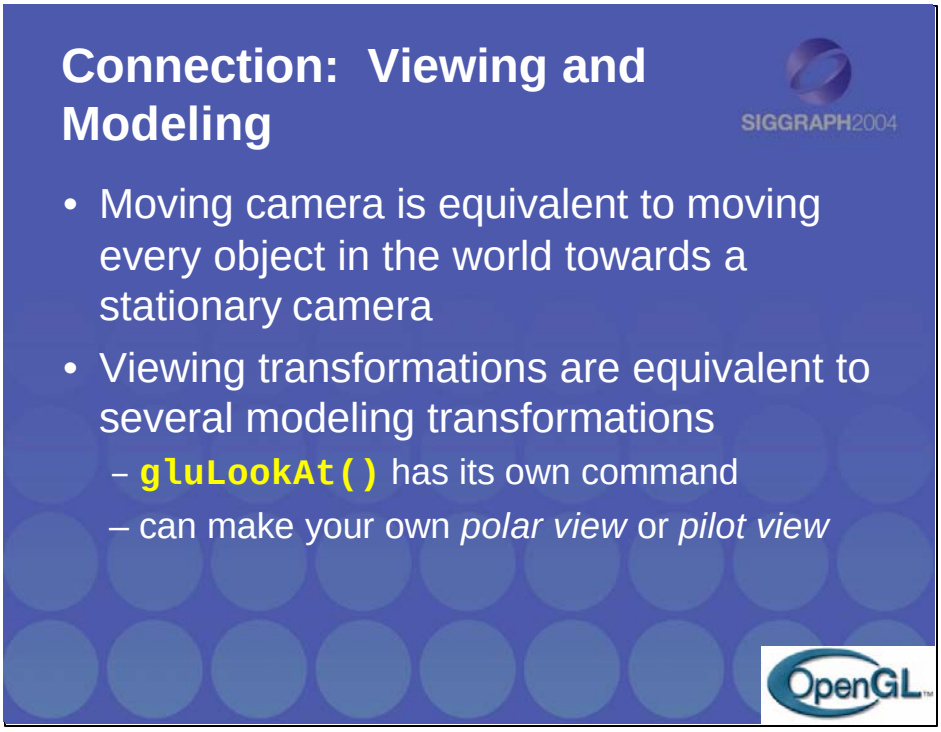
`glTranslate()`, `glRotate()`, and `glScale()` multiplies itself onto the current matrix, so it usually comes after `glMatrixMode(GL_MODELVIEW)`. There are many situations where the modeling transformation is multiplied onto a non-identity matrix.

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.



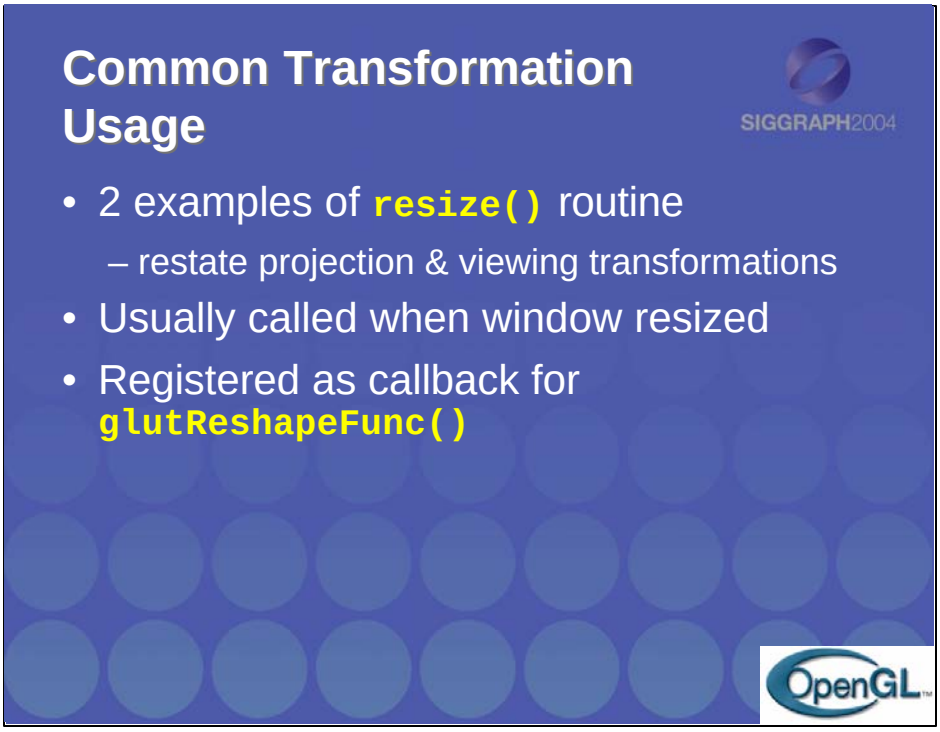
Instead of `gluLookAt()`, one can use the following combinations of `glTranslate()` and `glRotate()` to achieve a viewing transformation. Like `gluLookAt()`, these transformations should be multiplied onto the model-view matrix, which should have an initial identity matrix.

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)
```

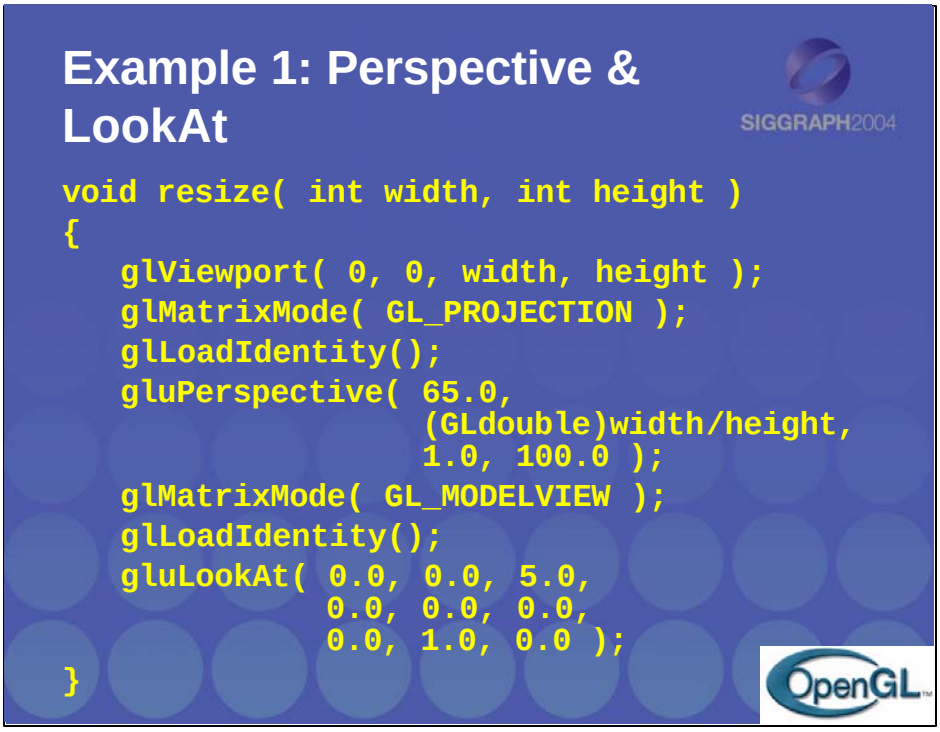


Example: Suppose the user resizes the window. Do we see the same objects?

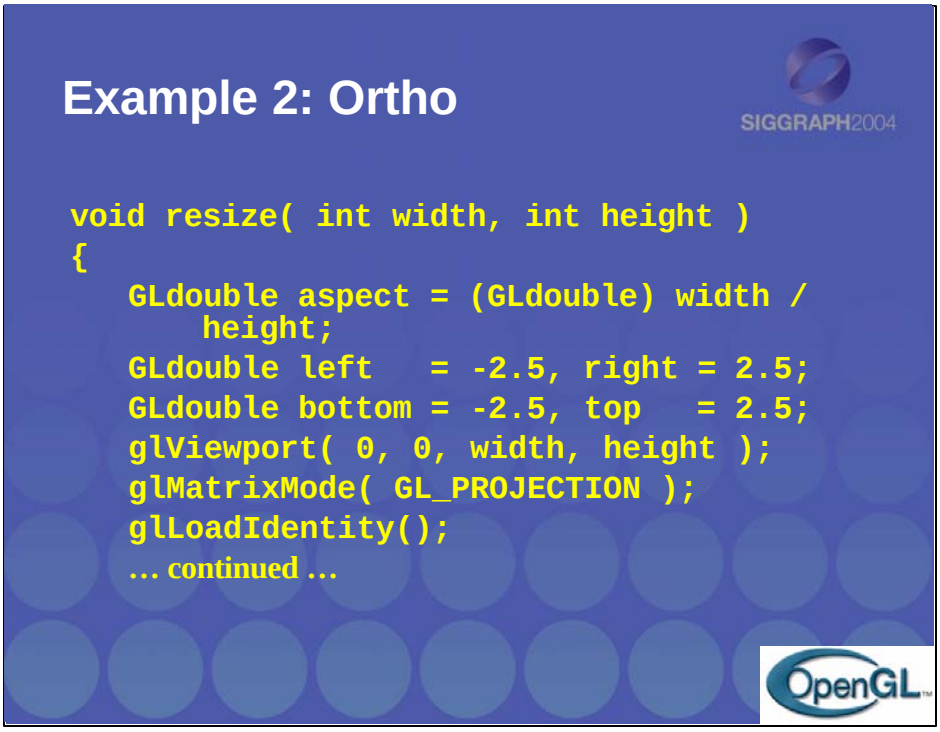
What if the new aspect ratio is different from the original? Can we avoid distortion of objects?

What we should do is application dependent. Hence users should write their own reshape callbacks.

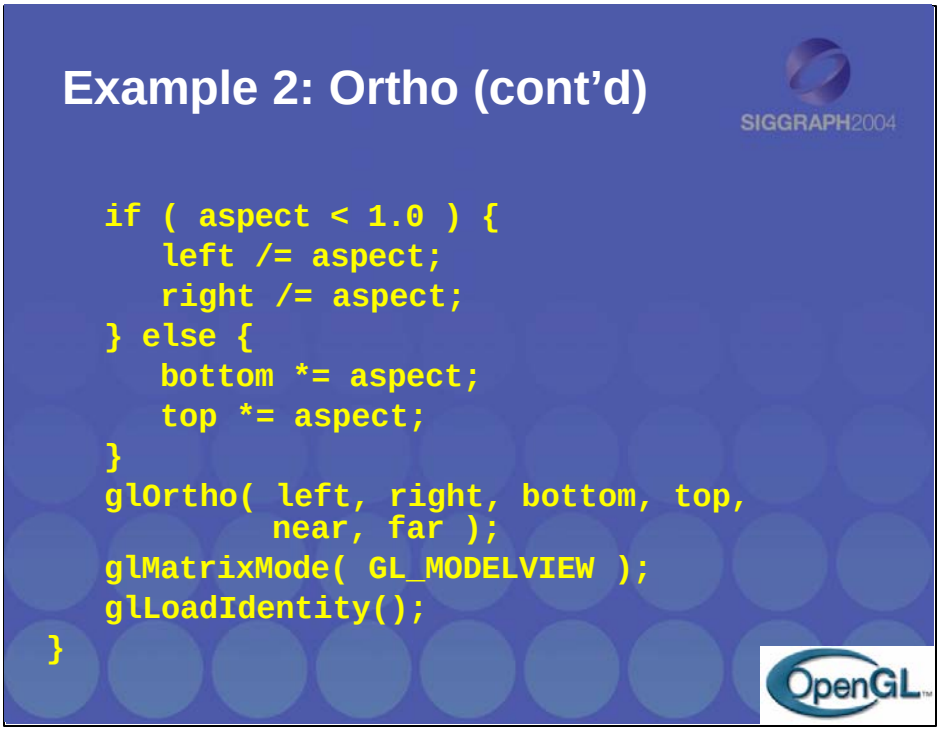
Typical reshape callbacks alter the projection matrix or the viewport.



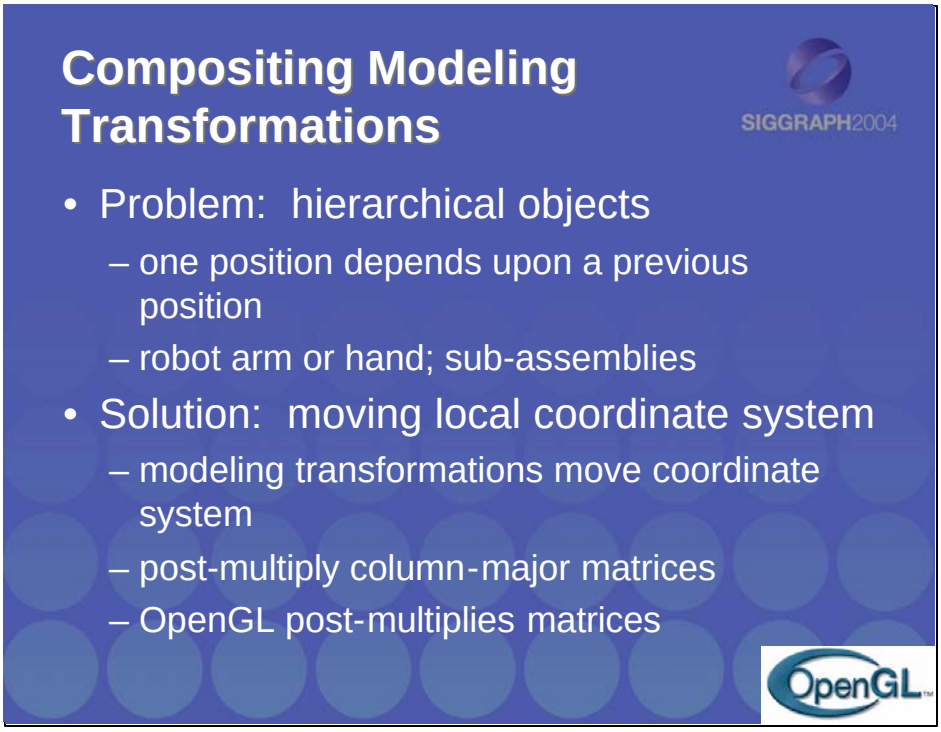
Example one of `resize()` uses the viewport's width and height values as the aspect ratio for `gluPerspective()` which eliminates distortion.



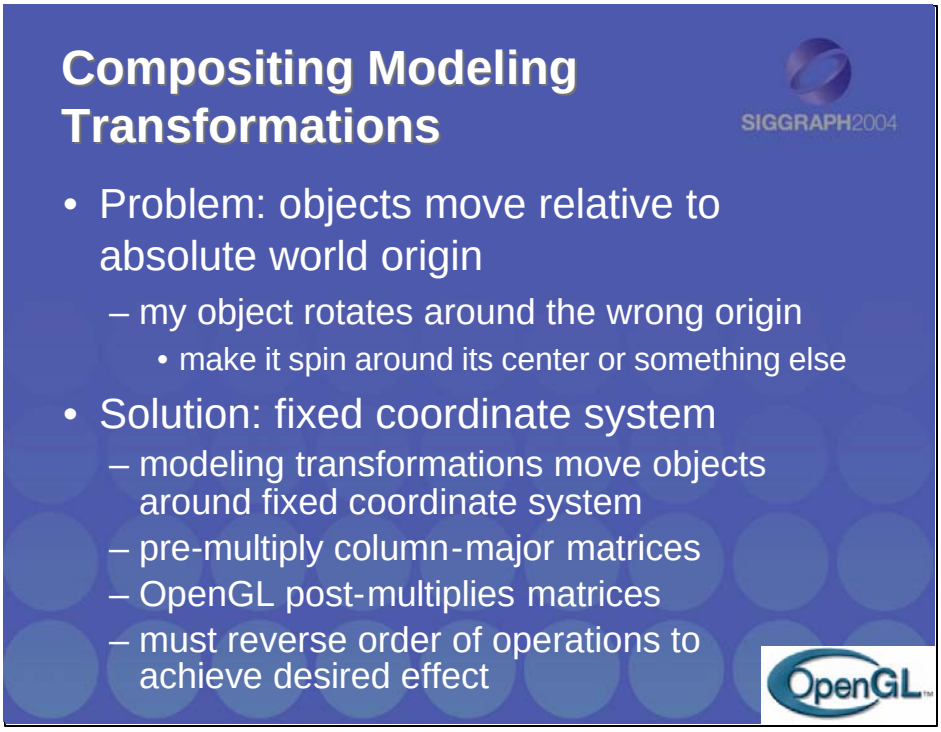
In example two of `resize()`, we first compute the aspect ratio (`aspect`) of the new viewing area. Then we will use this value to modify the world space values (`left`, `right`, `bottom`, `top`) of the viewing frustum depending on the new shape of the viewing volume



Continuing from the previous page, we determine how to modify the viewing volume based on the computed aspect ratio. After we compute the new world space values, we call `glOrtho()` to modify the viewing volume.



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.



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();
```

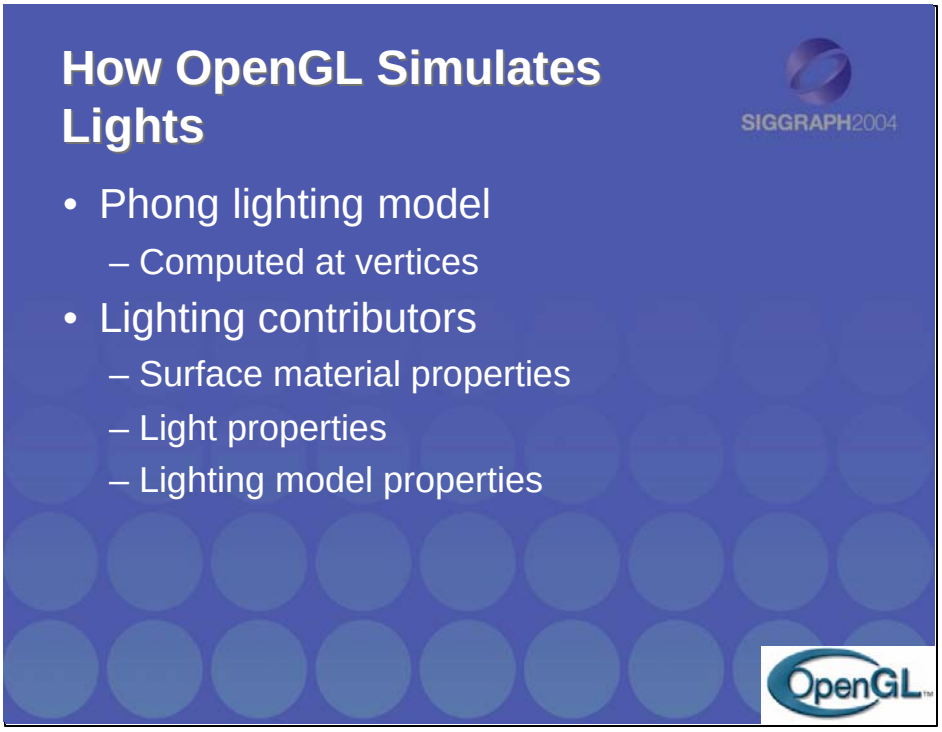
Here (*x*, *y*, *z*) is the fixed point. We first (last transformation in code) move it to the origin. Then we rotate about the axis (*ax*, *ay*, *az*) and finally move fixed point back.


SIGGRAPH2004
Lighting

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.

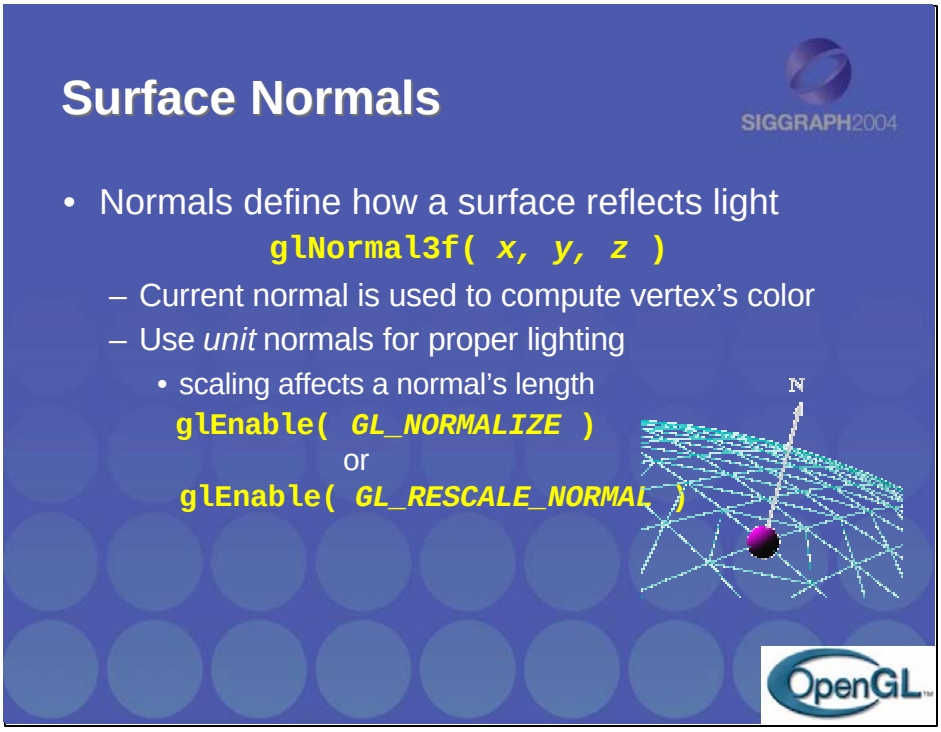
Lighting is available in both RGBA mode and color index mode. RGBA is more flexible and less restrictive than color index mode lighting.



OpenGL 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)

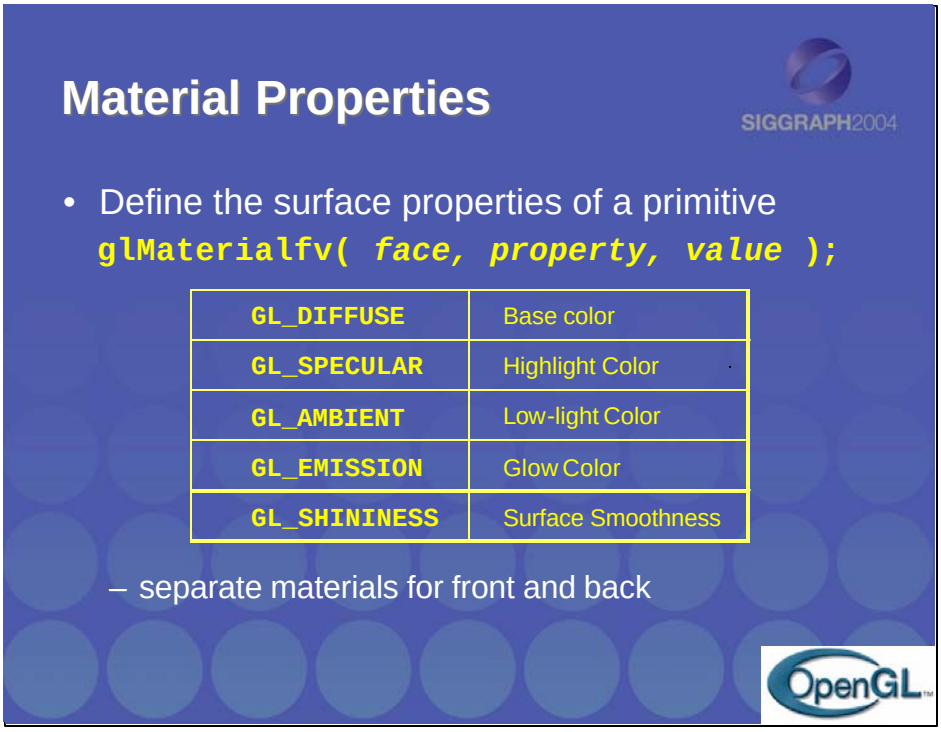


The lighting normal tells OpenGL how the object reflects light around a vertex. If you imagine that there is a small mirror at the vertex, the lighting normal describes how the mirror is oriented, and consequently how light is reflected.

`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 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:

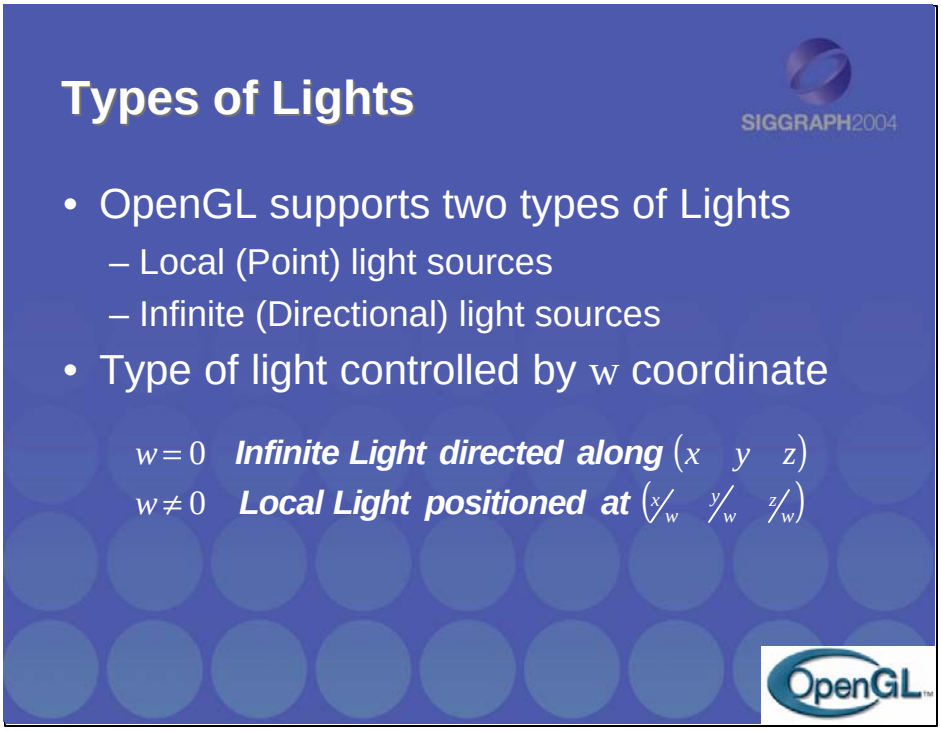
- `GL_DIFFUSE` - base color of object
- `GL_SPECULAR` - color of highlights on object
- `GL_AMBIENT` - color of object when not directly illuminated
- `GL_EMISSION` - color emitted from the object (think of a firefly)
- `GL_SHININESS` - concentration of highlights on objects. Values range from 0 (very 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`.

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.

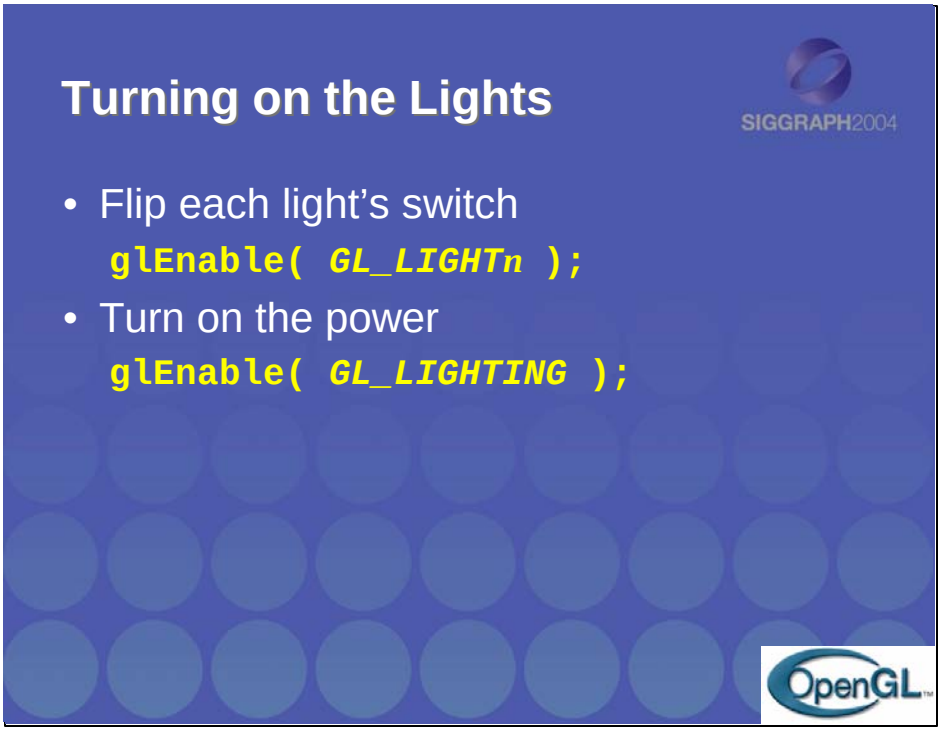
OpenGL lights can emit different colors for each of a materials properties. For example, a light's `GL_AMBIENT` color is combined with a material's `GL_AMBIENT` color to produce the ambient contribution to the color - Likewise for the diffuse and specular colors.



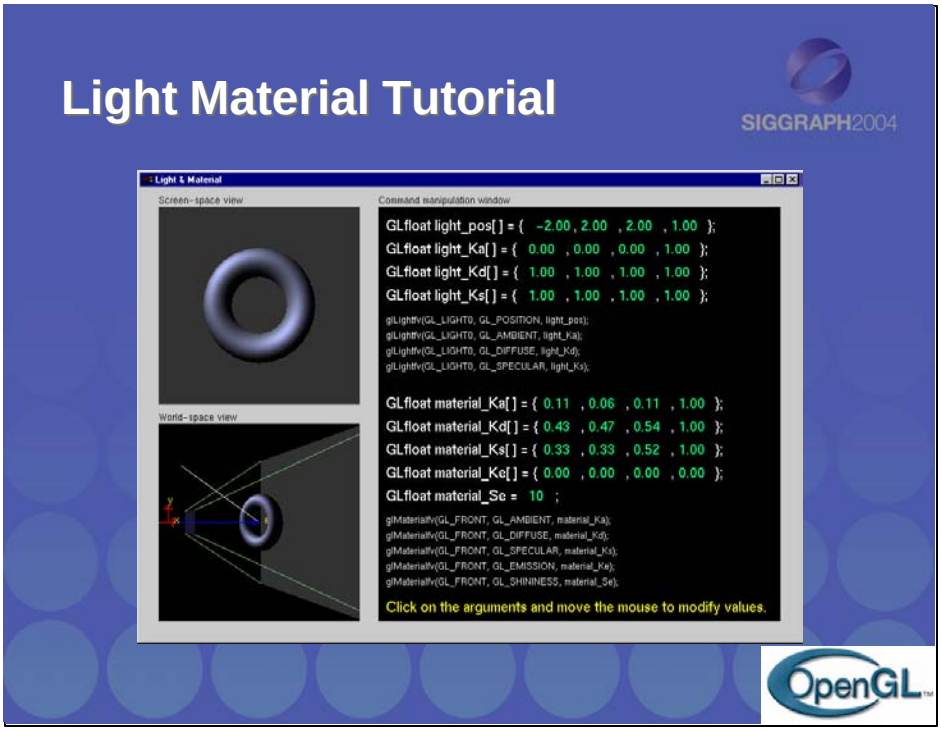
OpenGL supports two types of lights: infinite (directional) and local (point) light sources. The type of light is determined by the *w* coordinate of the light's position.

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

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

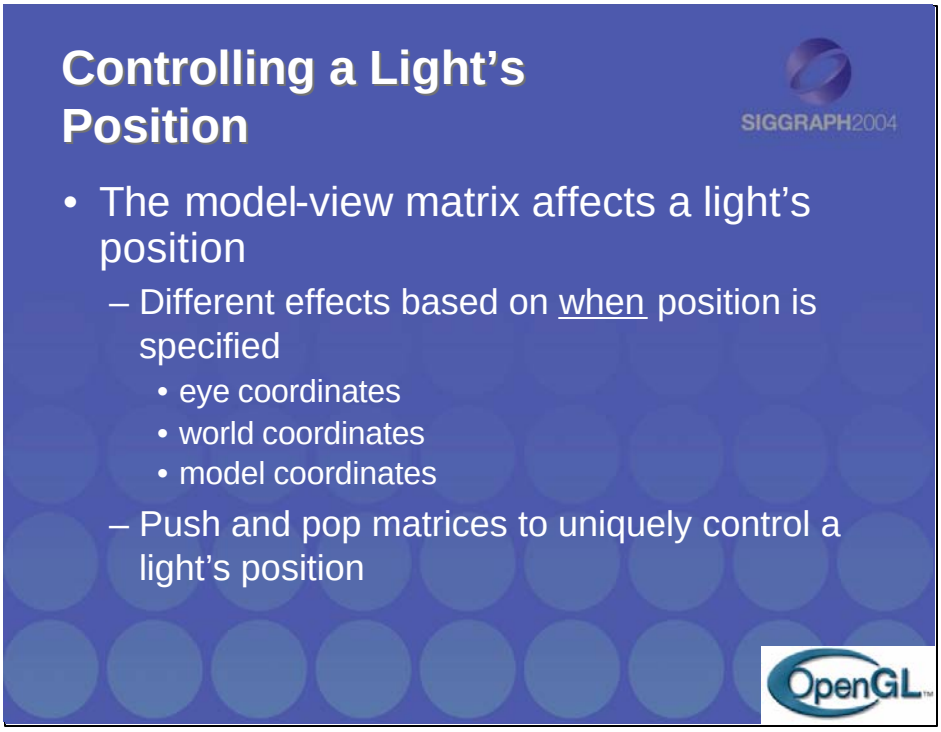


Each OpenGL light is controllable separately, using `glEnable()` and the respective light constant `GL_LIGHT`*n*. Additionally, global control over whether lighting will be used to compute primitive colors is controlled by passing `GL_LIGHTING` to `glEnable()`. This provides a handy way to enable and 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.



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

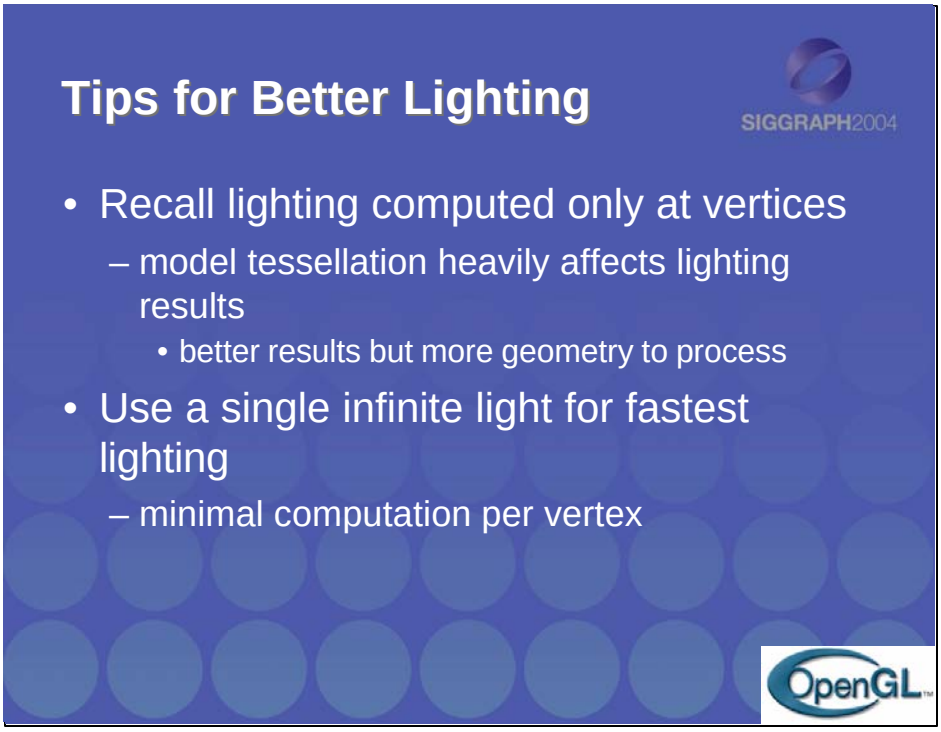
In general, there are three coordinate systems where you can specify a light's position/direction

1) *Eye coordinates* - which is represented by an identity matrix in the model-view. 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 model-view 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 model-view matrix. This method allows arbitrary, and even animated, position of a light using modeling transformations.

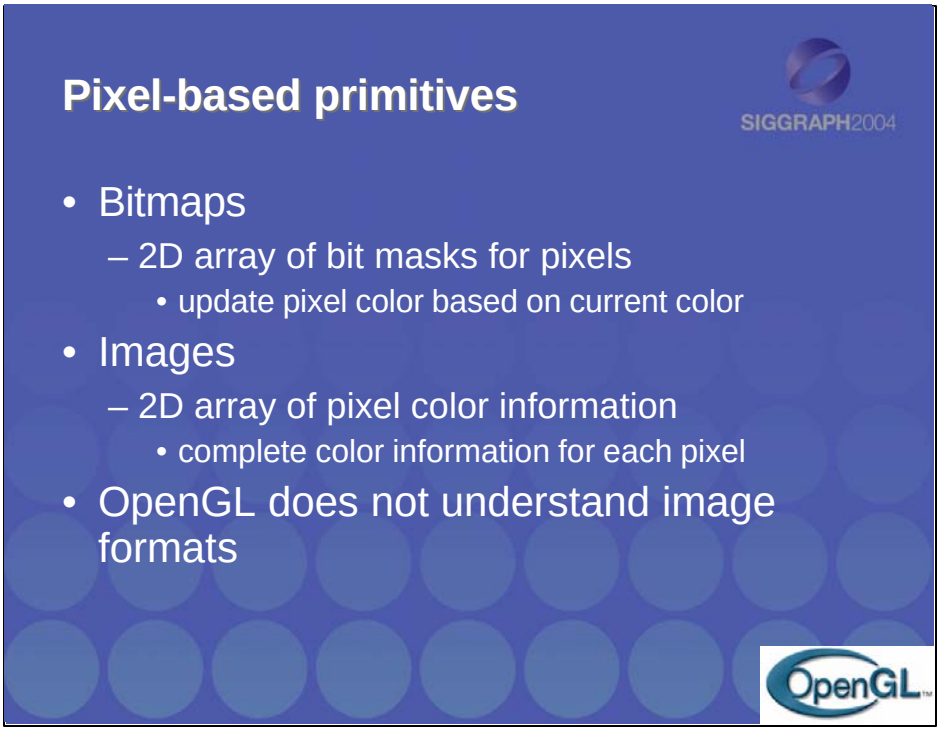
This tutorial demonstrates the different lighting affects of specifying a light's position in eye and world coordinates. Experiment with how highlights and illuminated areas change under the different lighting position specifications.



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.

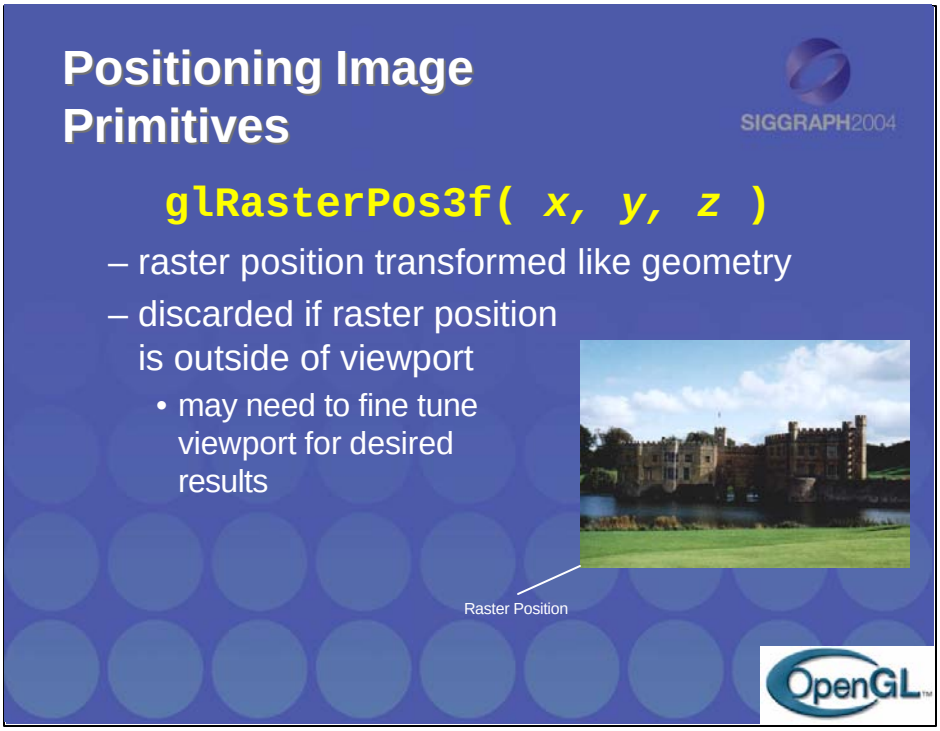

SIGGRAPH2004
Texture Mapping



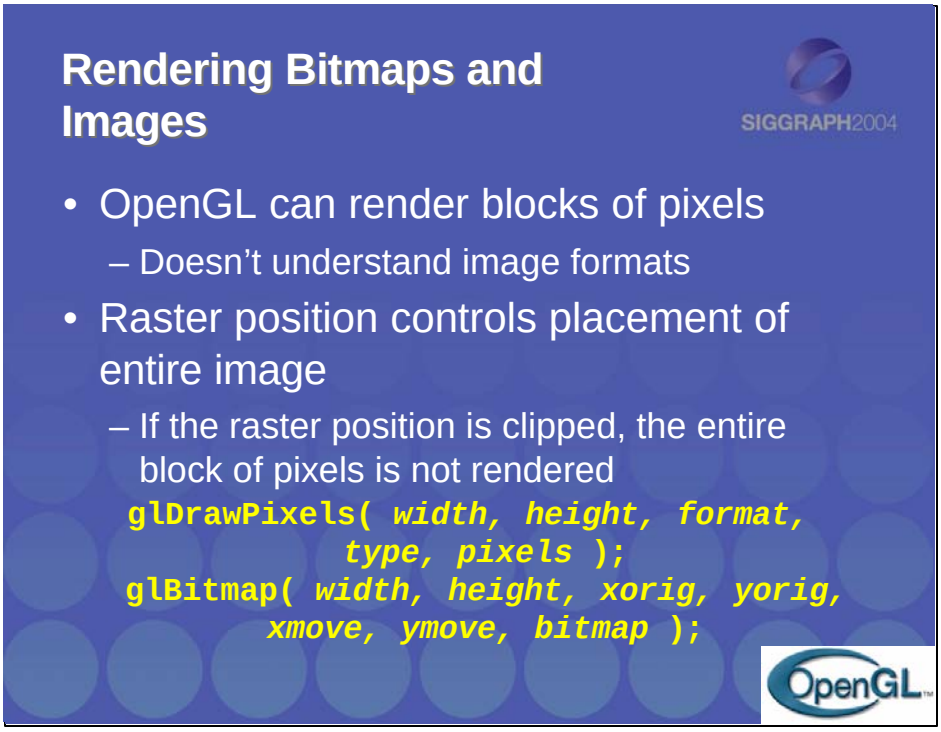
In addition to geometric primitives, OpenGL also supports *pixel-based primitives*. These primitives contain explicit color information for each pixel that they contain. They come in two types:

> *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.



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.

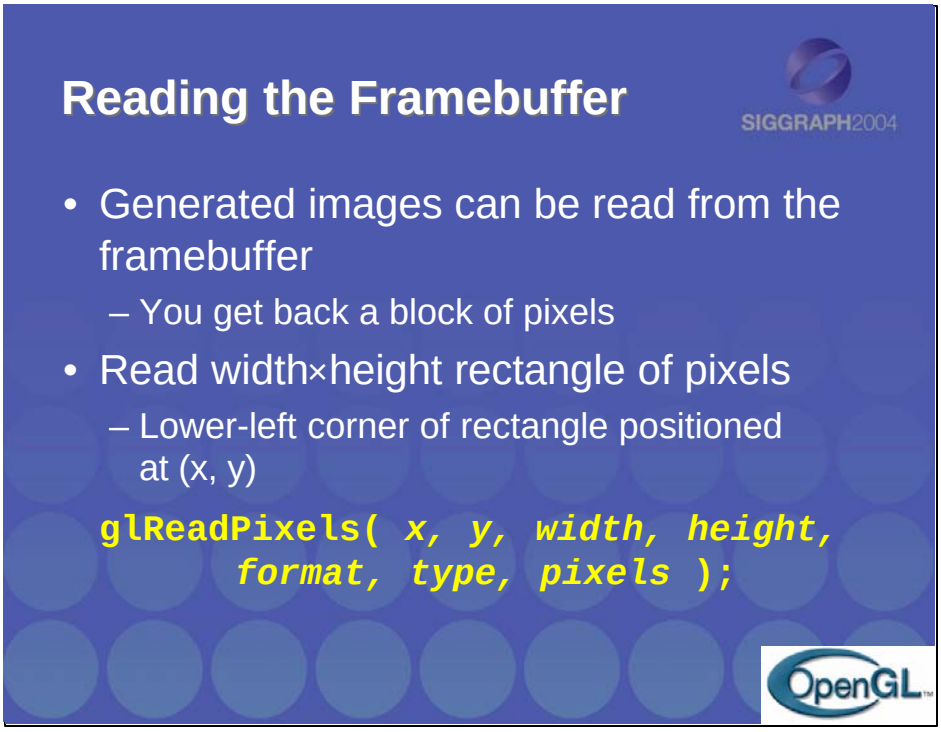


In addition to rendering geometric primitives, OpenGL can also render imaging primitives. `glDrawPixels`() will render a rectangle of color values with the lower-left corner of rectangle positioned at the current raster position (set with `glRasterPos*()`). `glBitmap()` will use the provided *bitmap*, which is a rectangle with a single bit representing whether a pixel should be colored or not. The bitmap's lower-left corner will also be positioned at the current raster position, and for any bits in the bitmap, the corresponding pixels will be shaded the current color when `glRasterPos*()` was called. (This means that if you call

```
glColor3fv( color1 );
glRasterPos2fv( pos );
glColor3fv( color2 );
glBitmap( … );
```

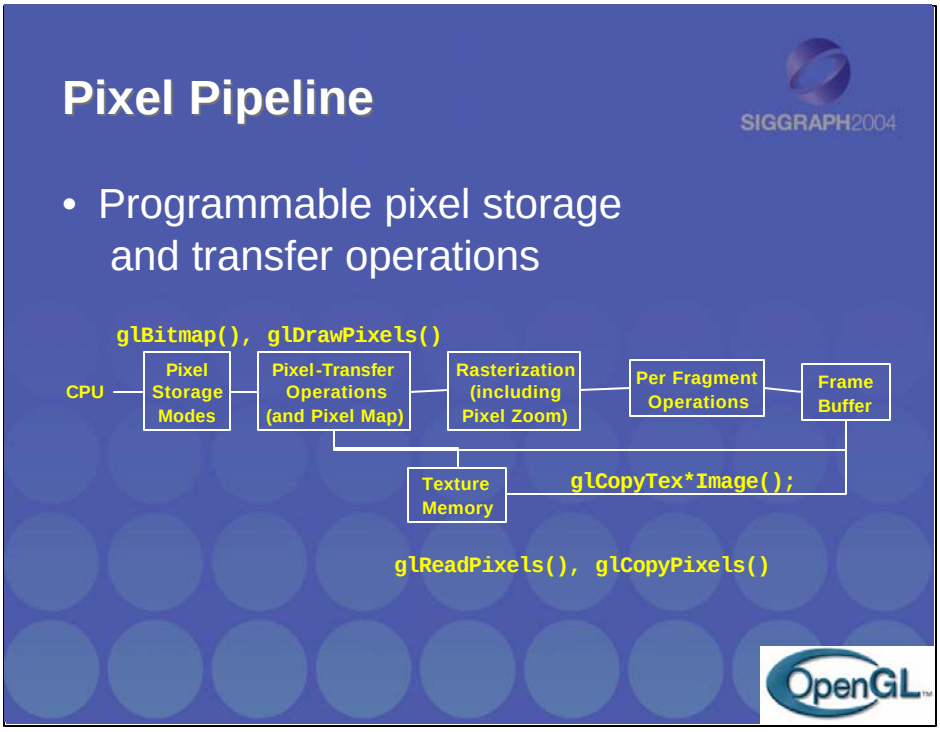
The color used to shade pixels will be color1, and not color2. Just one of those things to look out for.

OpenGL doesn't understand image formats (e.g., GIF, JPEG, TIFF, etc.), nor how to render an image in one of those formats. You need to read the image (probably using an image library, unless you're ambitious0, and extract the rectangle of pixel colors, which should then be passed into `glDrawPixels()`.



OpenGL can also read the rendered pixels from the framebuffer and return those values back to you. You might use them in an image file (e.g., a GIF, JPEG, TIFF, etc.), as a frame in a movie, or whatever. In addition to reading the color buffer, you can also read the depth buffer, stencil buffer, and others. The *format* parameter controls what type of pixel values or color components you want to render from the framebuffer.

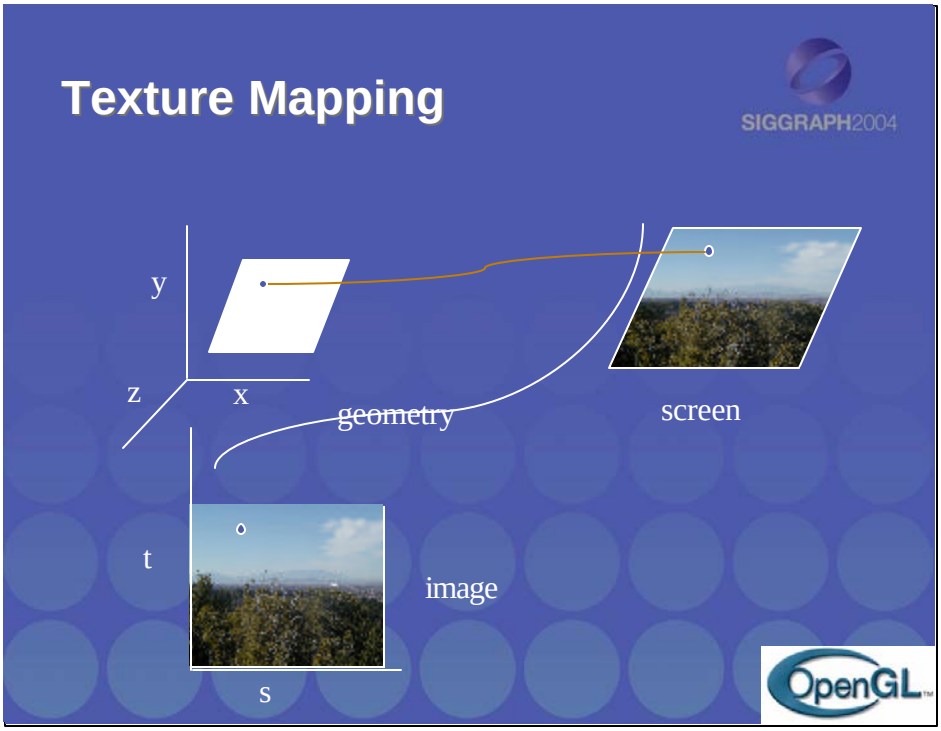
A by-product of calling `glReadPixels()` is that it will flush all outstanding OpenGL calls before the framebuffer is read.



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.

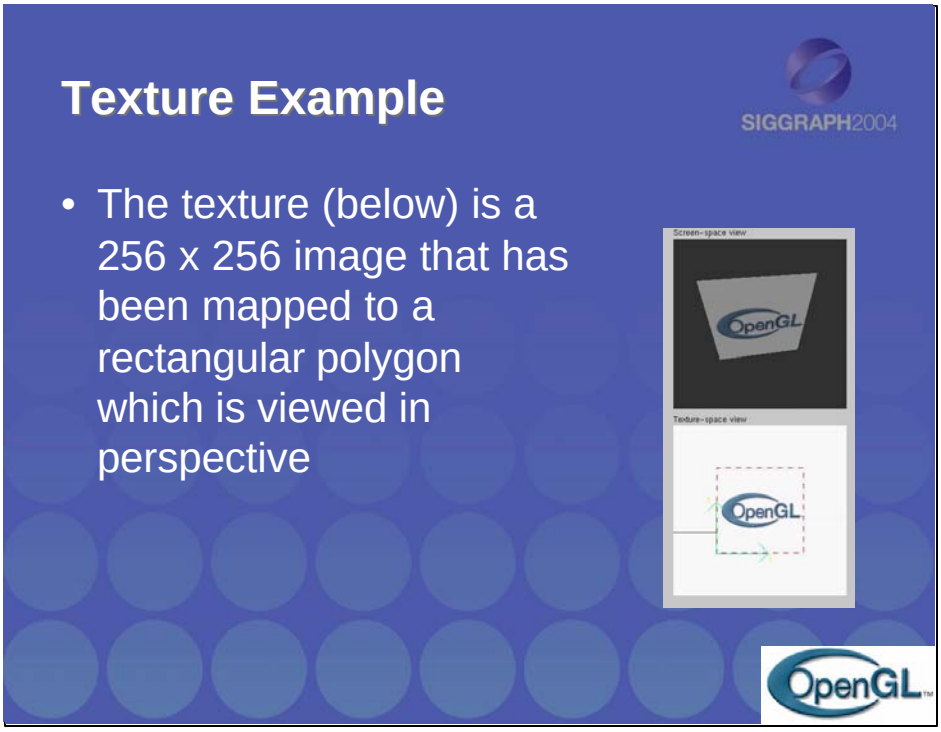
In addition to rendering into the framebuffer, pixels can be copied from the framebuffer back into host memory, or transferred into texture mapping memory.

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.



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 three-dimensional geometric object in (*x*, *y*, *z*) world coordinates maps a point in (*s*, *t*) space to a corresponding point on the screen.



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.

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.

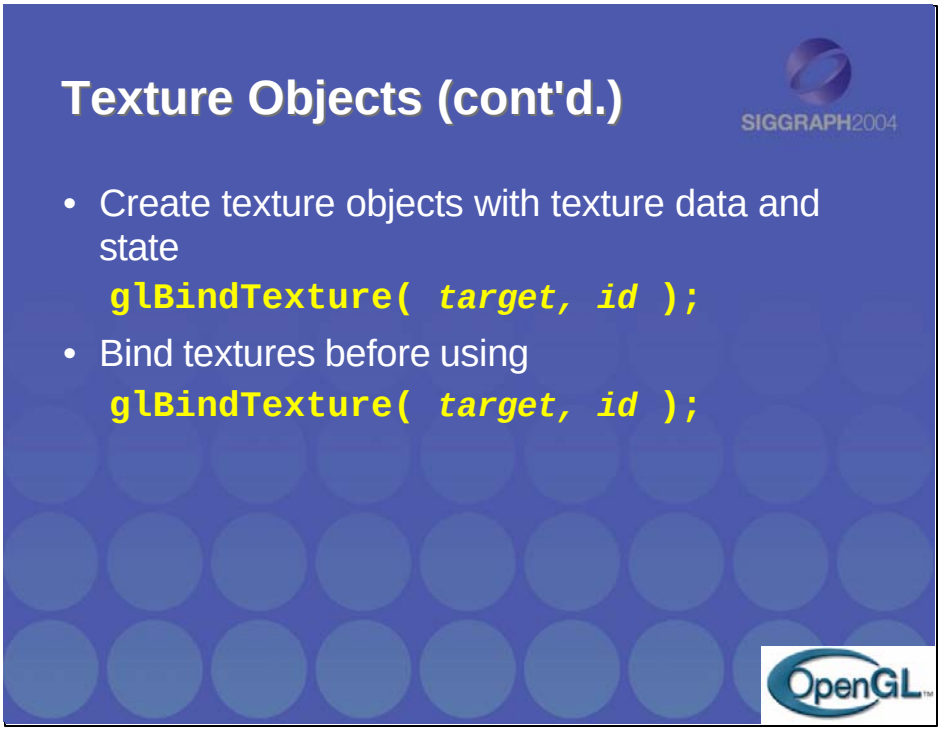
Texture memory is a limited resource and having only a single active texture can lead to inefficient code.

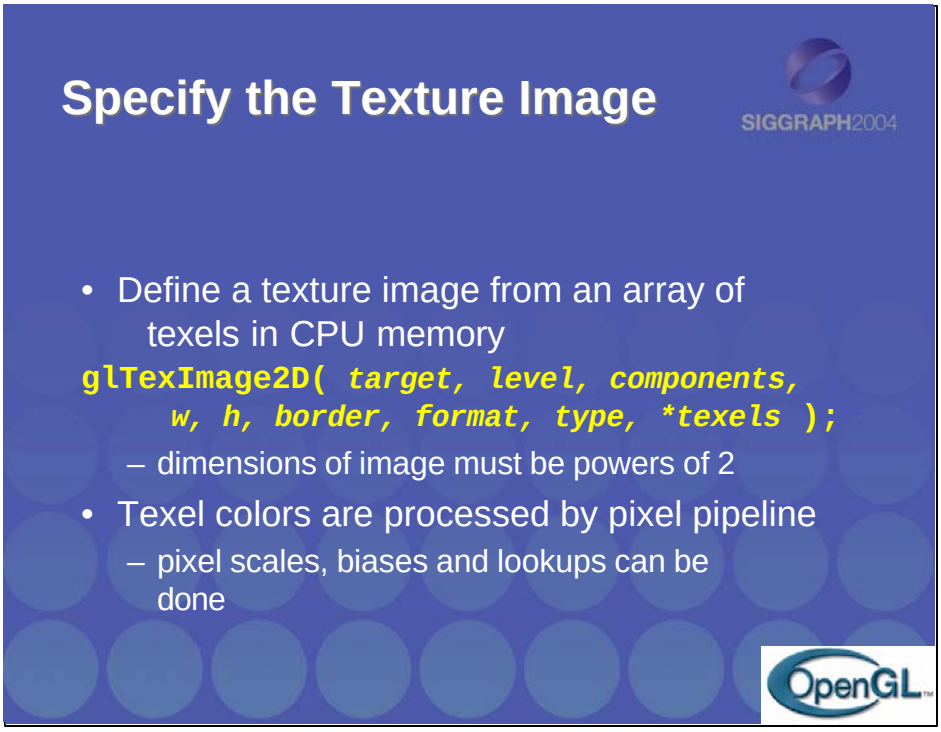
The first step in creating texture objects is to have OpenGL reserve some indices for your objects. `glGenTextures()` will request *n* texture ids and return those values back to you in `texIds`.

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.

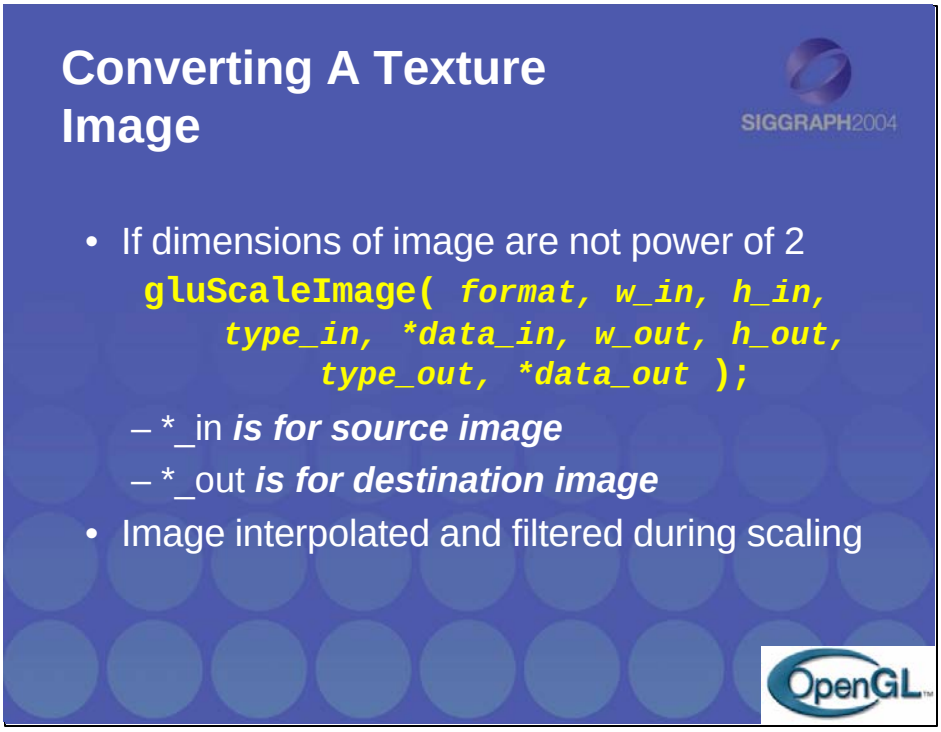

Texture Objects (cont'd.)
SIGGRAPH2004
• Create texture objects with texture data and state
glBindTexture(target, id);
• Bind textures before using
glBindTexture(target, id);
OpenGL



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 sent to OpenGL with `glTexImage*()` must be a power of two in both directions. An optional one texel wide border may be added around the image. This is useful for certain wrapping modes.

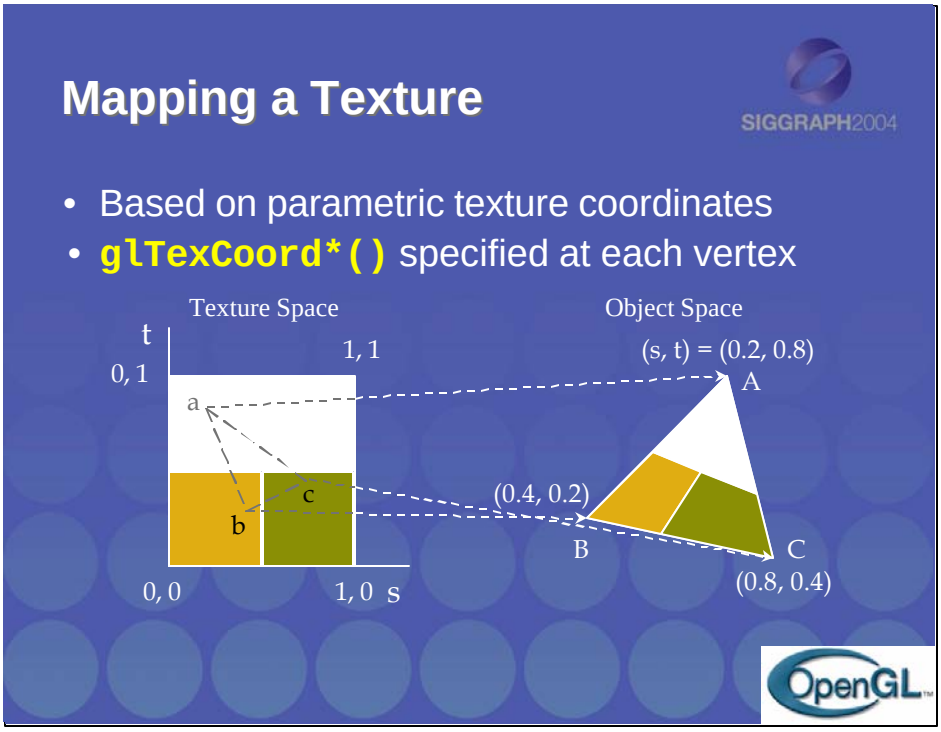
The level parameter is used for defining how OpenGL should use this image when mapping texels to pixels. Generally, you'll set the level to 0, unless you are using a texturing technique called mipmapping, which we will discuss in the next section.



If your image does not meet the power of two requirement for a dimension, the `gluScaleImage()` call will resample an image to a particular size. It uses a simple box filter to interpolate the new images pixels from the source image.

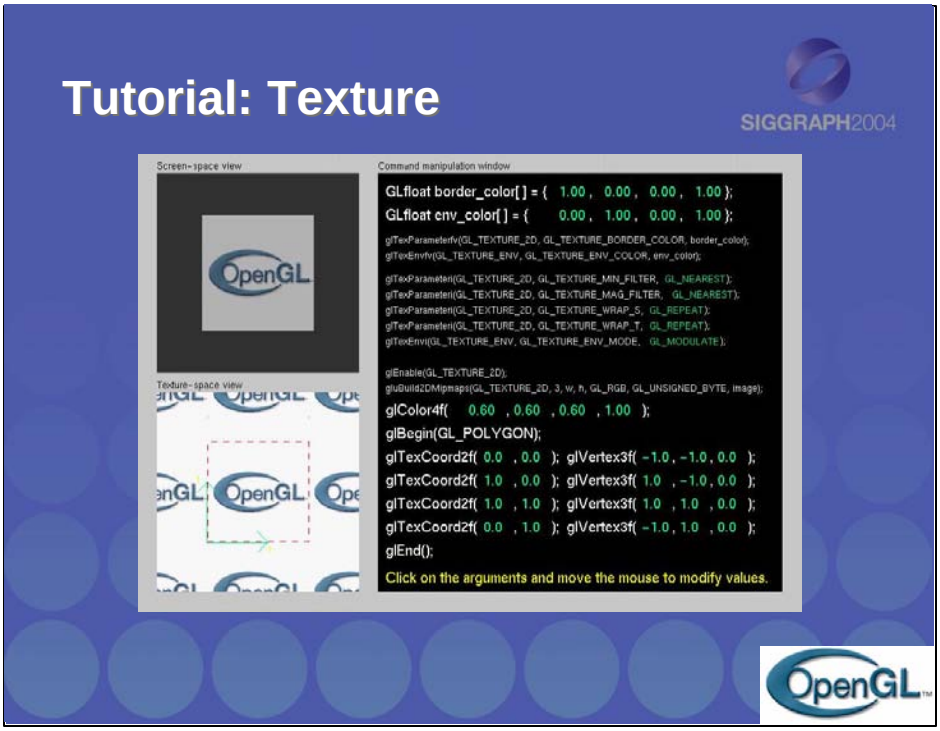
Additionally, `gluScaleImage()` can be used to convert from one data type (i.e. `GL_FLOAT`) to another type, which may better match the internal format in which OpenGL stores your texture.

Note that use of `gluScaleImage()` can also save memory.



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.

Tutorial: Texture



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.

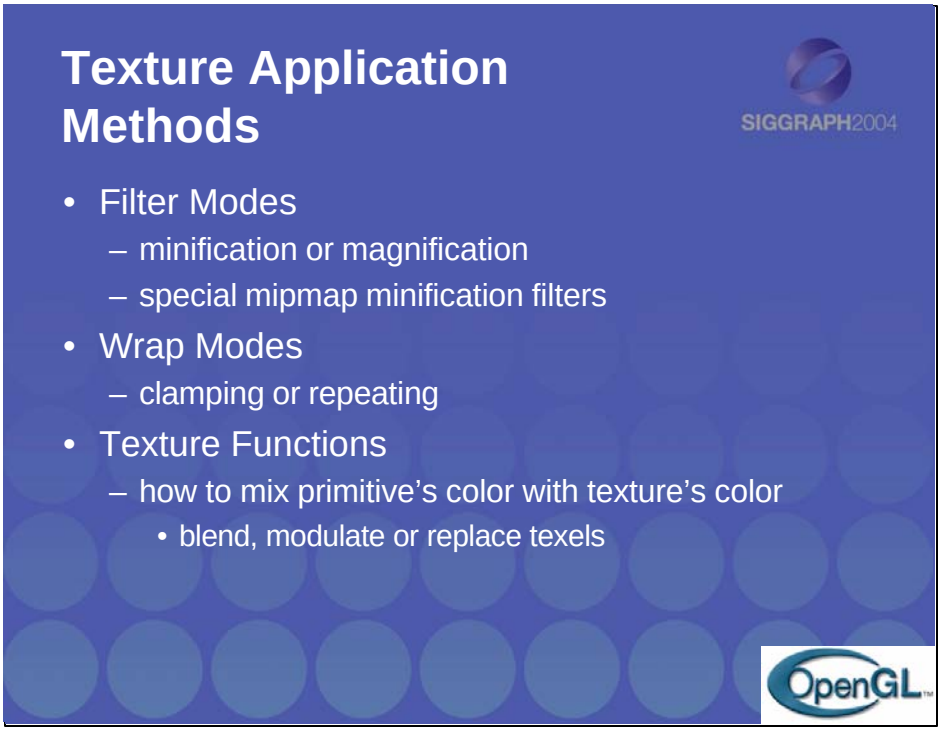
As with any other OpenGL state, texture mapping requires that `glEnable()` be called. The tokens for texturing are:

`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.



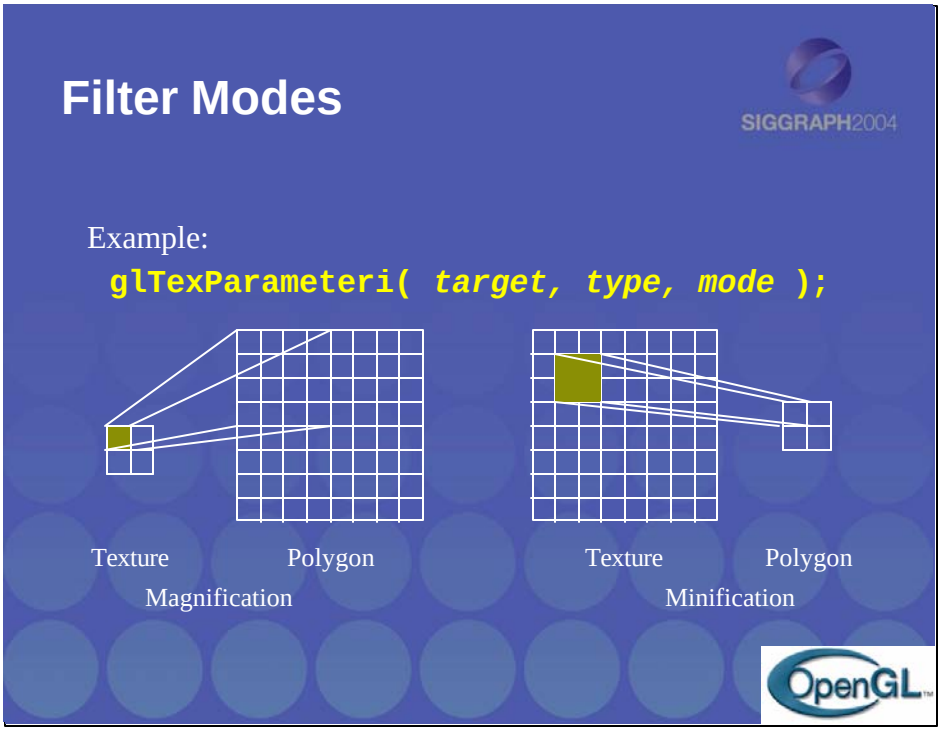
Textures and the objects being textured are rarely the same size (in pixels). Filter modes determine the methods used by how texels should be expanded (magnification), or shrunk (minification) to match a pixel's size. An additional technique, called mipmapping is a special instance of a minification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



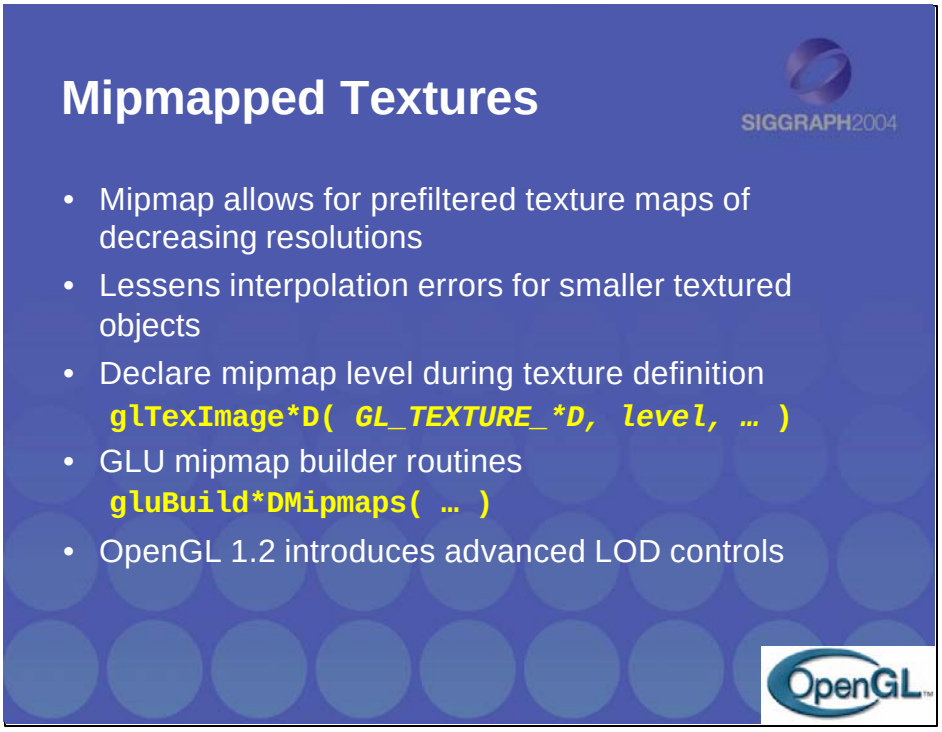
Filter modes control how pixels are minified or magnified. Generally a color is computed using the nearest texel or by a linear average of several texels.

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.

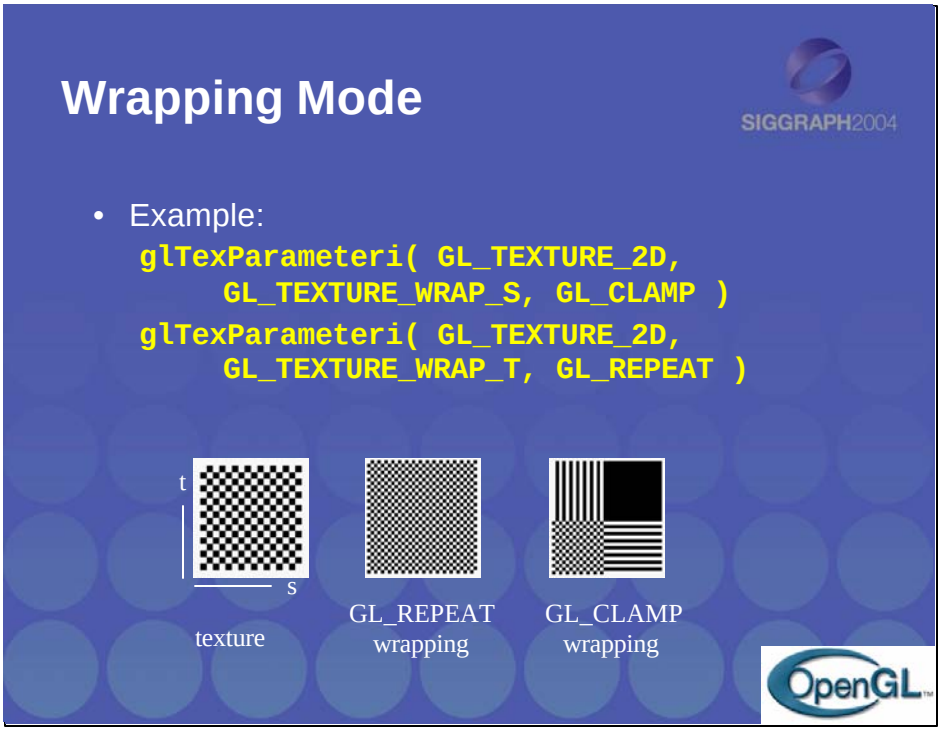


As primitives become smaller in screen space, a texture may appear to shimmer as the minification filters creates rougher approximations. Mipmapping is an attempt to reduce the shimmer effect by creating several approximations of the original image at lower resolutions.

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.

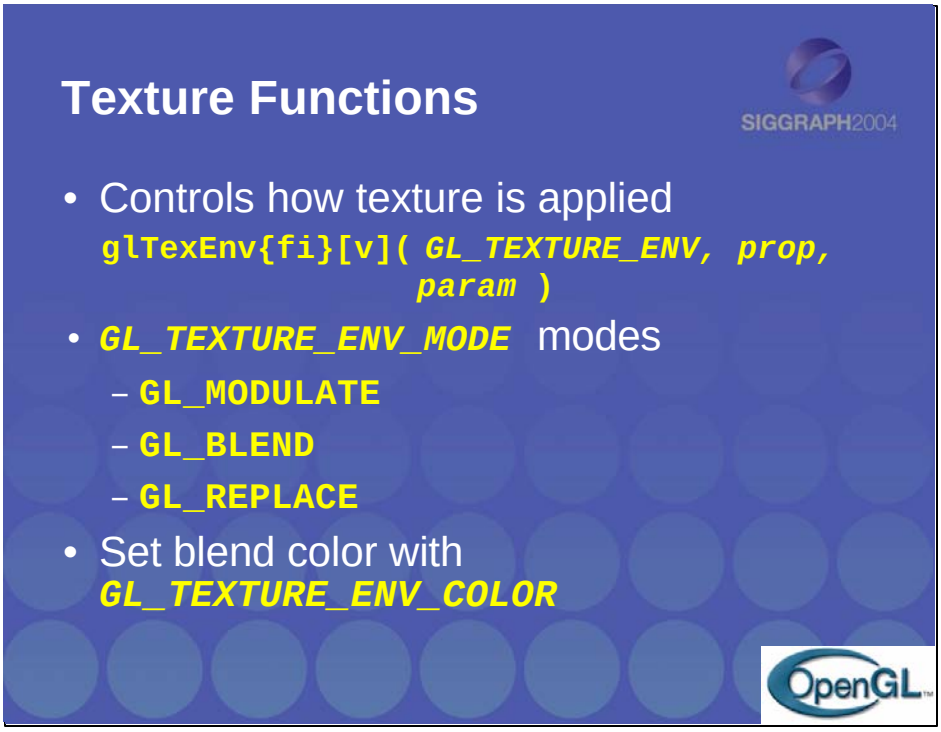
The `gluBuild*Dmipmaps()` routines will automatically generate each mipmap image, and call `glTexImage*D()` with the appropriate level value.

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).



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.



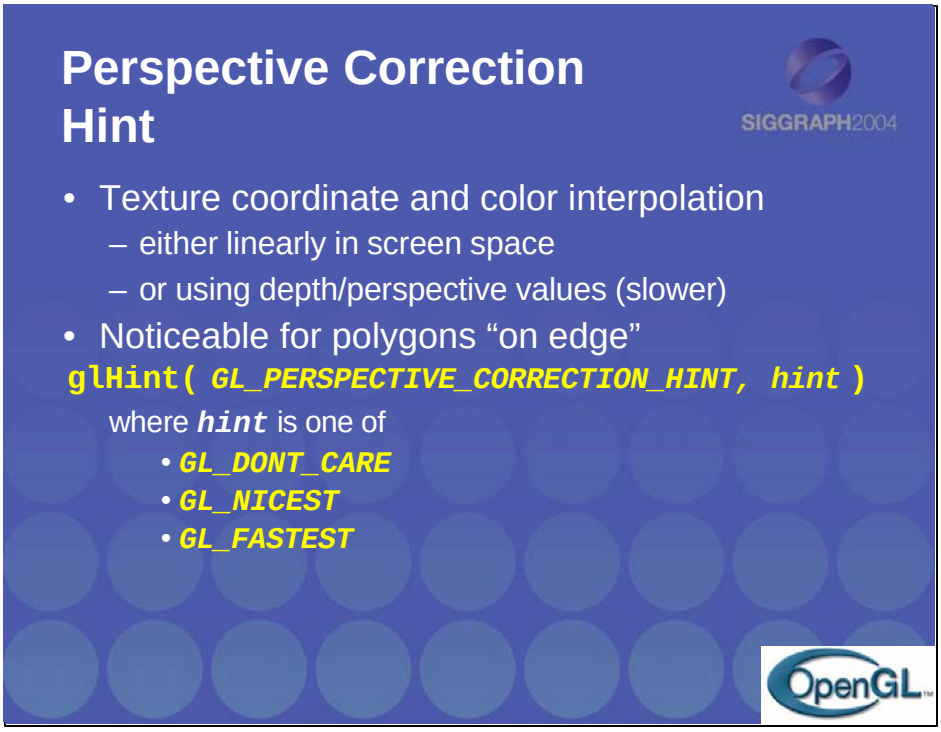
The texture mode determines how texels and fragment colors are combined. The most common modes are:

`GL_MODULATE` - multiply texel and fragment color

`GL_BLEND` - linearly blend texel, fragment, env color

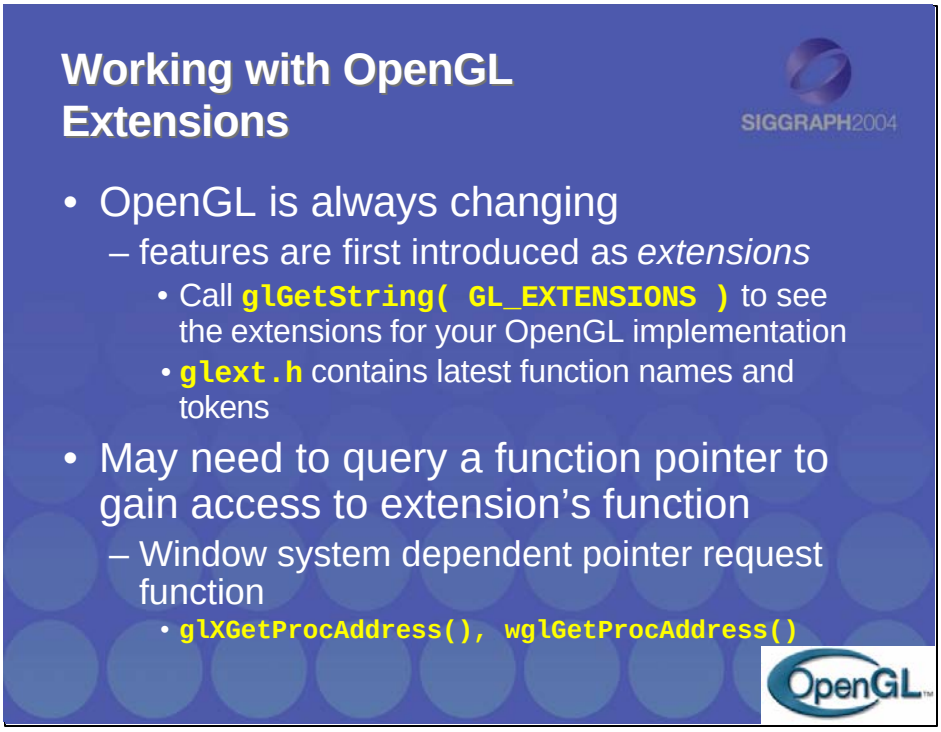
`GL_REPLACE` - replace fragment's color with texel

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.



An OpenGL implementation may chose to ignore hints.


SIGGRAPH2004
Advanced OpenGL Topics

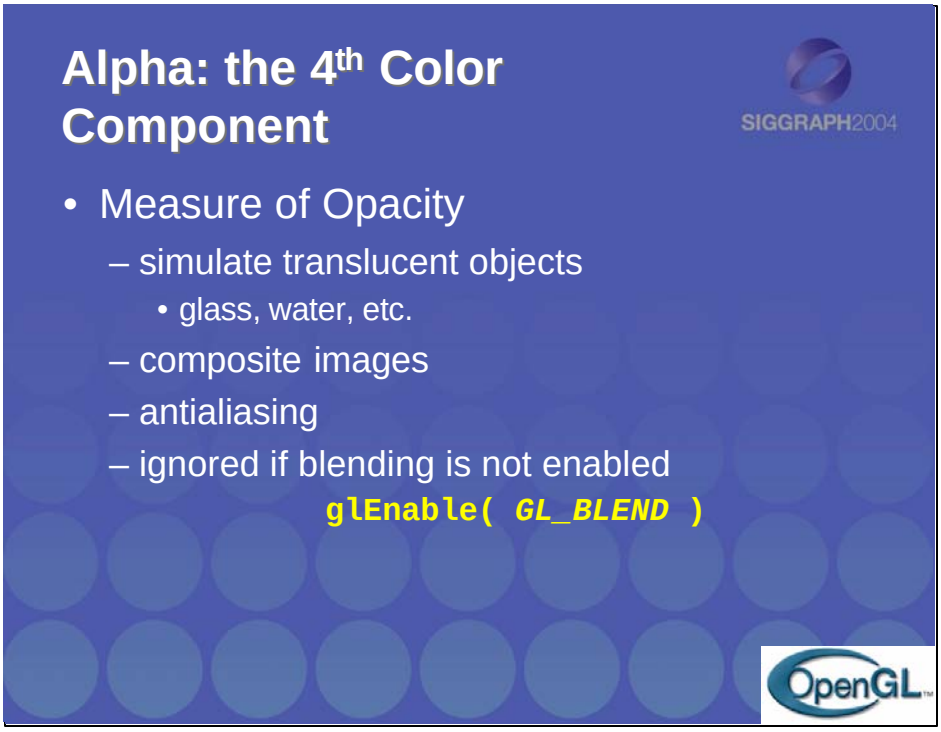


OpenGL is continually adding new features. These new functions and operations are first introduced as *extensions* to OpenGL. If they are adopted by the OpenGL community, they may be added into the core of OpenGL when OpenGL's revision changes.

To access extensions, you should first check that the implementation you're working with supports the extensions that you need to use. Calling `glGetString(GL_EXTENSIONS)` will return back the list of extensions that your OpenGL implementation supports.

Once you know that your extension is supported, you may need to request a pointer in order to be able to call the function (this has little to do with OpenGL itself, but more with the variations in how operating systems and drivers relate; however, this technique will work on almost all OpenGL implementations). In order to obtain the function pointer, you need to query the windowing system for that function. Since this operation is window-system dependent, the function used varies from system to system (this functionality hasn't been implemented in GLUT at the time of this writing). For the X Window System, you would call `glXGetProcAddress()`, and for Microsoft Windows, you'd call `wglGetProcAddresss()`. Either function will either return NULL if the function's not available (i.e., the extension's not supported), or a pointer to the function.

The file `glext.h`, which is maintained at the `www.opengl.org` website, contains the latest list of OpenGL extensions with their functions and tokens.

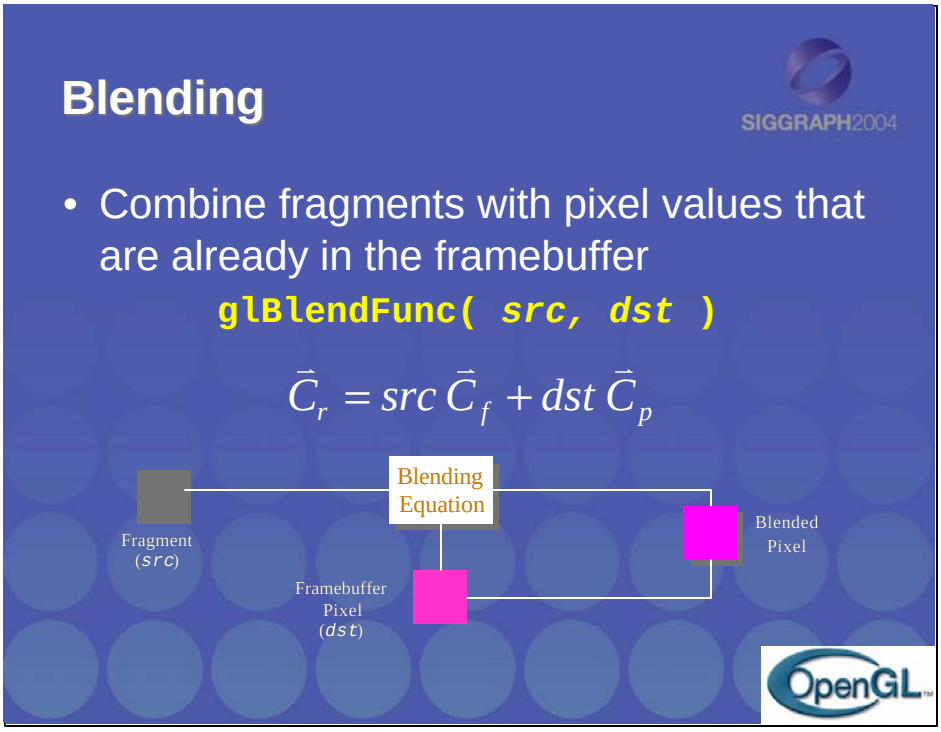


The alpha component for a color is a measure of the fragment's opacity. As with other OpenGL color components, its value ranges from 0.0 (which represents completely transparent) to 1.0 (completely opaque).

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 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.

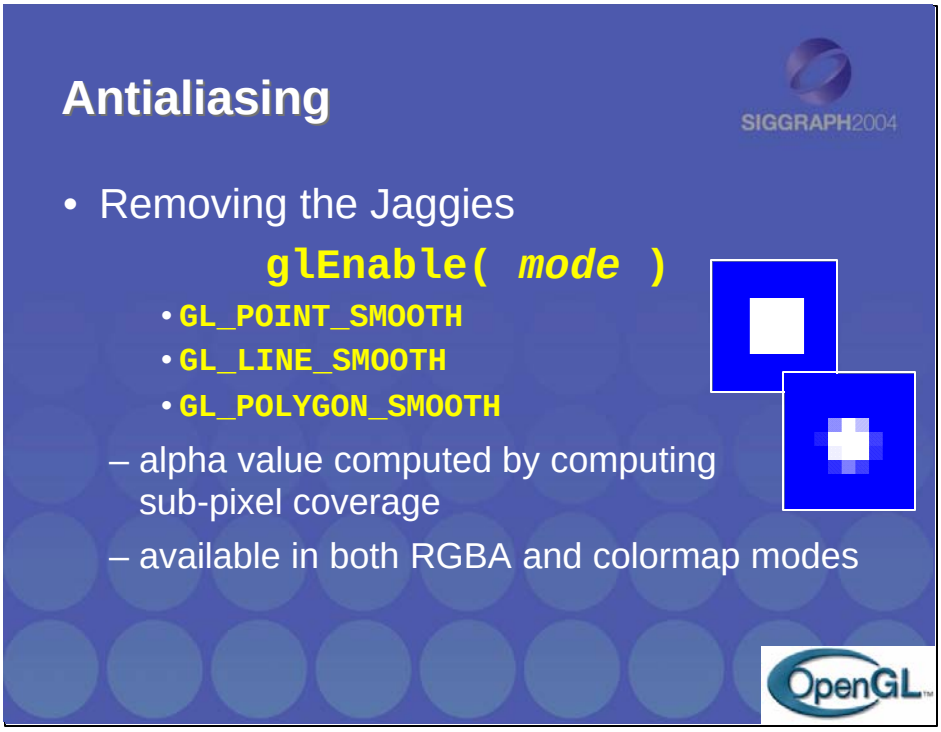
The fragment and pixel each have a factor which controls their contribution to the final pixel color. These *blending factors* are set using `glBlendFunc()`, which sets the source factor, which is used to scale the incoming fragment color, and the destination blending factor, which scales the pixel read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



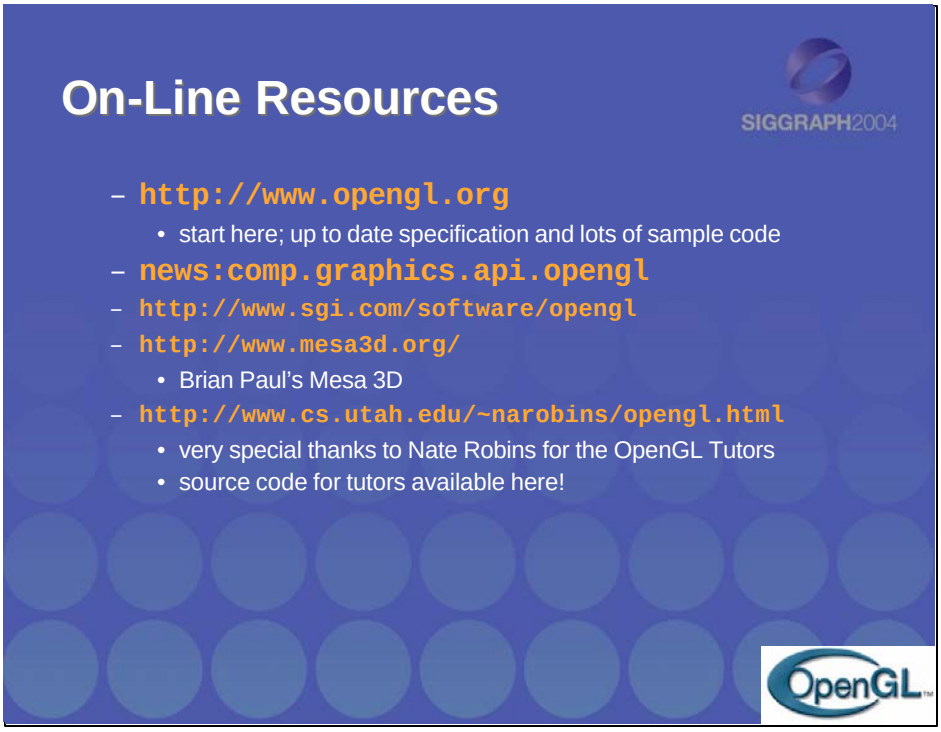
Antialiasing is a process to remove the *jaggies* which is the common name for jagged edges of rasterized geometric primitives. OpenGL supports antialiasing of all geometric primitives by enabling both `GL_BLEND` and one of the constants listed above.

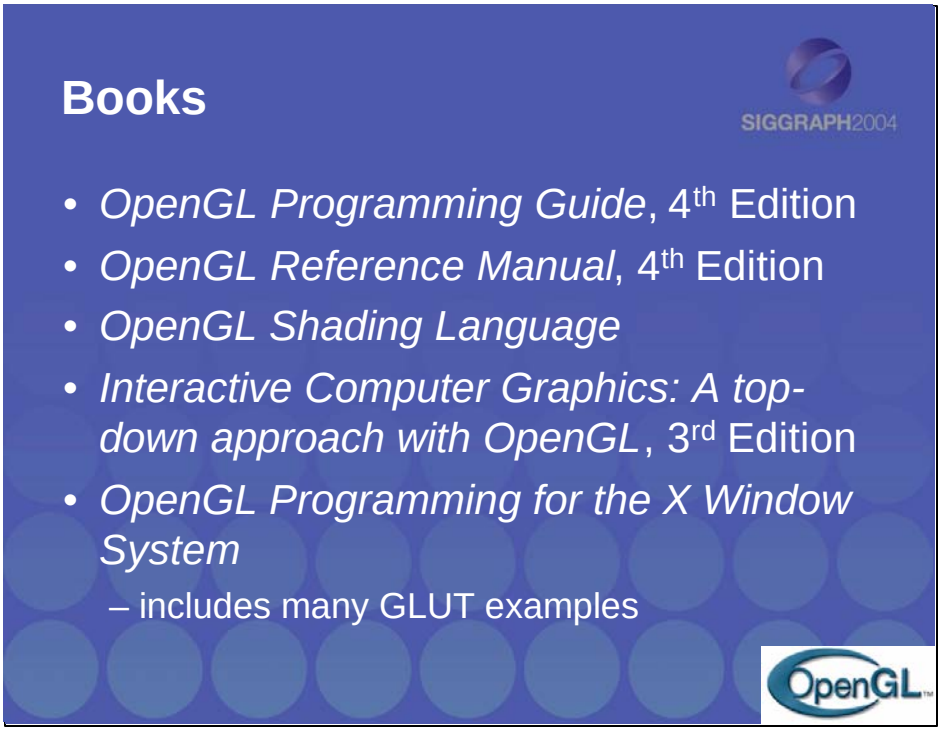
Antialiasing is accomplished in RGBA mode by computing an alpha value for each pixel that the primitive touches. This value is computed by subdividing the pixel into *subpixels* and determining the ratio used subpixels to total subpixels for that pixel. Using the computed alpha value, the fragment's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


SIGGRAPH2004
Summary / Q & A


On-Line Resources
SIGGRAPH2004
– http://www.opengl.org
• start here; up to date specification and lots of sample code
– news:comp.graphics.api.opengl
– http://www.sgi.com/software/opengl
– http://www.mesa3d.org/
• Brian Paul's Mesa 3D
– http://www.cs.utah.edu/~narobins/opengl.html
• very special thanks to Nate Robins for the OpenGL Tutors
• source code for tutors available here!
OpenGL



The OpenGL Programming Guide is often referred to as the "Red Book" due to the color of its cover. Likewise, *The OpenGL Reference Manual* is also called the "Blue Book."

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."

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


Thanks for Coming
SIGGRAPH2004
• Questions and Answers
Dave Shreiner shreiner@sgi.com
Ed Angel angel@cs.unm.edu
Vicki Shreiner vshreiner@sgi.com
OpenGL

Bibliography

- *The OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.4, 4th Edition*
  The OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis.
  Addison-Wesley / November 2003
  ISBN: 0321173481
- *The OpenGL Reference Manual, Version 1.4, 4th Edition*
  The OpenGL Architecture Review Board; Edited by Dave Shreiner
  Addison-Wesley / February 2004
  ISBN: 032117383X
- *Interactive Computer Graphics: A Top-Down Approach with OpenGL, 3rd Edition*
  Ed Angel
  Addison-Wesley / July 2002
  ISBN: 0201773430
- *OpenGL: A Primer, 2nd Edition*
  Ed Angel
  Addison-Wesley / June 2001
  ISBN: 0321237625
- *OpenGL Programming for the X Window System*
  Mark Kilgard
  Addison-Wesley / August 1996
  ISBN: 0201483599
- *OpenGL Programming for Windows 95 and Windows NT*
  Ron Fosner
  Addison-Wesley / October 1996
  ISBN: 0201407094
- *The OpenGL Shading Language*
  Randi Rost
  Addison-Wesley / February 2004
  ISBN: 0321197895
- *The OpenGL Extensions Guide*
  Eric Lengyel
  Charles River Media / July 2003
  ISBN: 1584502940

Glossary

antialiasing

A technique to reduce the visual artifacts (commonly called the *jaggies)* that result from rasterization of geometric primitives into the framebuffer. Most often the technique employs alpha-blending, or usage of a *nulti-sampled* framebuffer.

back buffer

The non-visible rendering buffer where images are rendered before a buffer swap. When a swap buffer occurs (e.g., when the applications calls `glutSwapBuffers()`) the *front buffer* and the back buffer are exchanged.

callback functions

A function that is called when a certain event occurs. The GLUT library uses callback functions as it's principle means of allowing you to control the response to various user input (e.g., pressed keys, moving the mouse, resizing the window, etc.)

enumerated types

Specific types that OpenGL defines to help with cross-platform compatibility.

eye coordinates

The three-dimensional coordinate system used by OpenGL. *World coordinates* are transformed into eye-coordinates by the application of the model-view matrix.

flat shading

Using the same color for all fragments that are defined by a geometric primitive.

frame

A completed rendering. An animation, for example, is a sequence of frames.

front buffer

The visible buffer in double-buffering mode. This buffer is displayed while rendering is directed to the *back buffer*.

geometric primitives

Points, lines, or polygons: the only rendering primitives available in core OpenGL.

gouraud- (or smooth-) shaded

Interpolating colors across a geometric primitive.

homogenous coordinates

Four-dimensional coordinates, (x, y, z, w) used for representing vertices. The reason they're called "homogenous" is that all transformations reduce to a 4×4 matrix multiplication operation.

image primitives

Pixel rectangles or bitmaps that can be rendered directly into the framebuffer.

jaggies

The stair-stepping effect that occurs when rendering

	diagonal edges in the framebuffer. *Antialiasing* is used to reduce the effects of the jaggies.
model coordinates	The three-dimensional coordinate system where your models are defined. Model coordinates are the one that's you pass into OpenGL using the `glVertex*()` functions.
multi-sampled buffer	A framebuffer where each pixel is represented by a collection of sub-pixels. When a primitive is rendered, each sub-pixel is colored, and when the processing of all sub-pixels is completed, the results of the sub-pixels are combined to form the final pixel color in the framebuffer. This allows the system to better antialias primitives.
pipelined architecture	A model for specifying the operations that OpenGL executes in processing geometric and image primitives.
rendering	The process of drawing in computer graphics.
setting state	Modifying OpenGL's internal state to change how will render primitives.
state	OpenGL's internal values that are used when rendering. Most OpenGL functions are for setting state.
stipples	Patterns applied to lines and polygons. Stipples differ from texture maps as they determine whether a pixel is to be rendered or not, as compared to a texture's modification of a pixel's color.
texture mapping	A process of coloring a pixel based on looking up colors in an image. Texture mapping allows you to specify considerable more detail to a geometric primitive than is possible using just gouraud shading alone.
texture object	An OpenGL object that manages the state for a texture map. Using texture objects helps to manage OpenGL's resources more efficiently to increase performance.
world coordinates	The coordinate system where all of your models live and are position in. In general if you only specify a viewing transformation, and never any modeling transformations, then all of the coordinates you pass into OpenGL will be world coordinates.