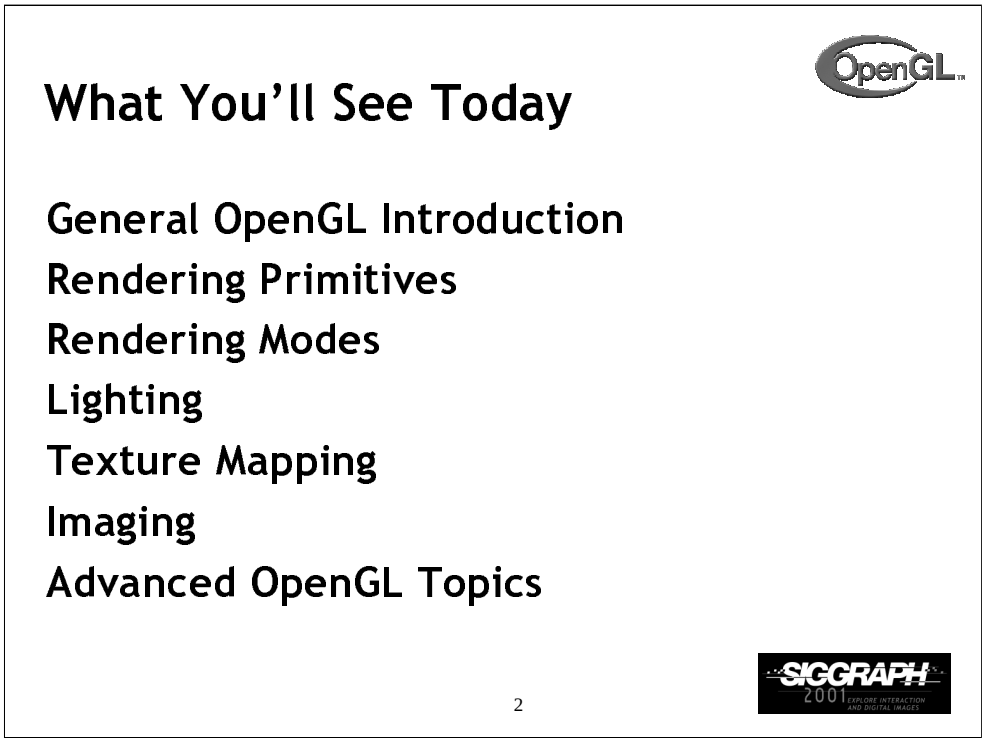
An Interactive Introduction to OpenGL Programming

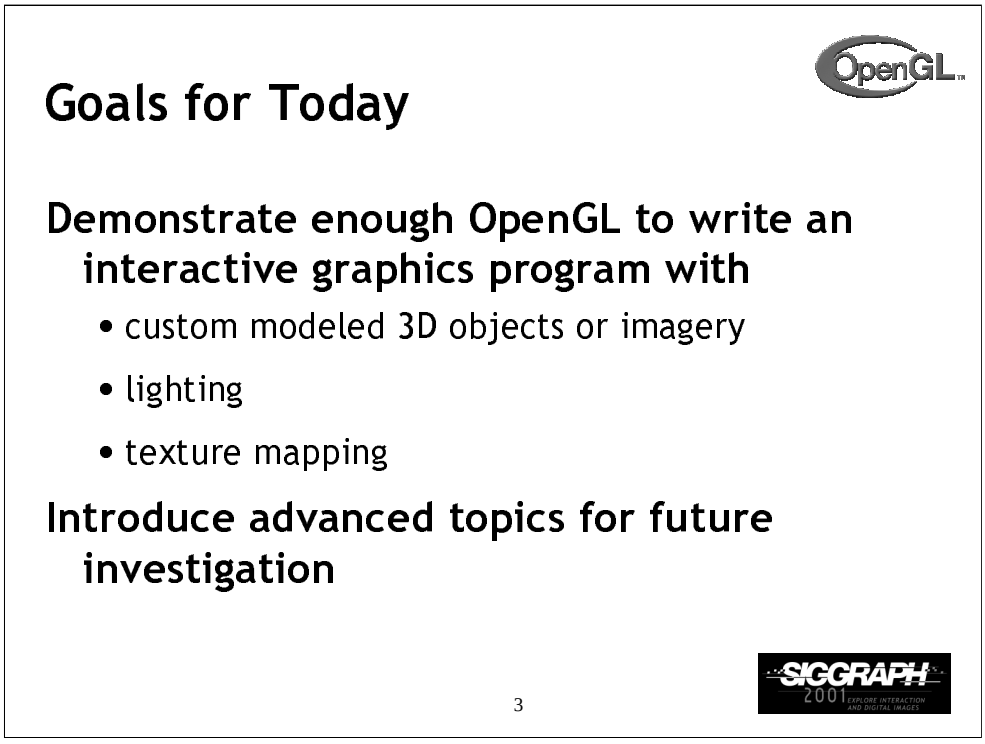
Dave Shreiner
Ed Angel
Vicki Shreiner

1



This course provides a general introduction and overview to the OpenGL API (Application Programming Interface) and its features. OpenGL is a rendering library available on almost any computer which supports a graphics monitor.

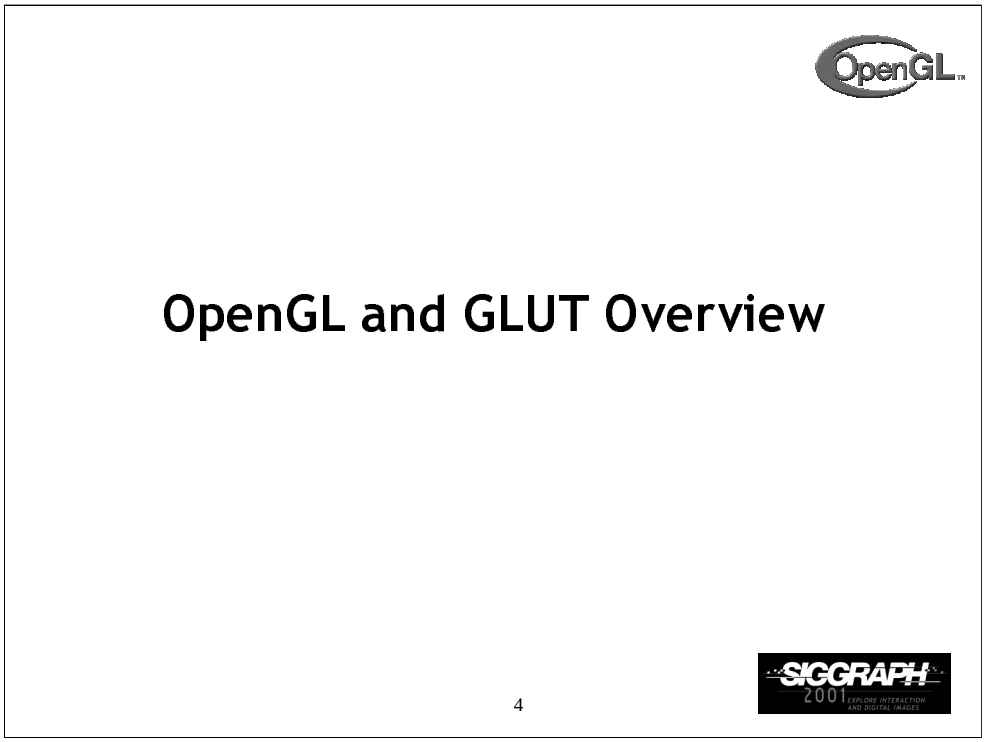
Today, we'll discuss the basic elements of OpenGL: rendering points, lines, polygons and images, as well as more advanced features as lighting and texture mapping.

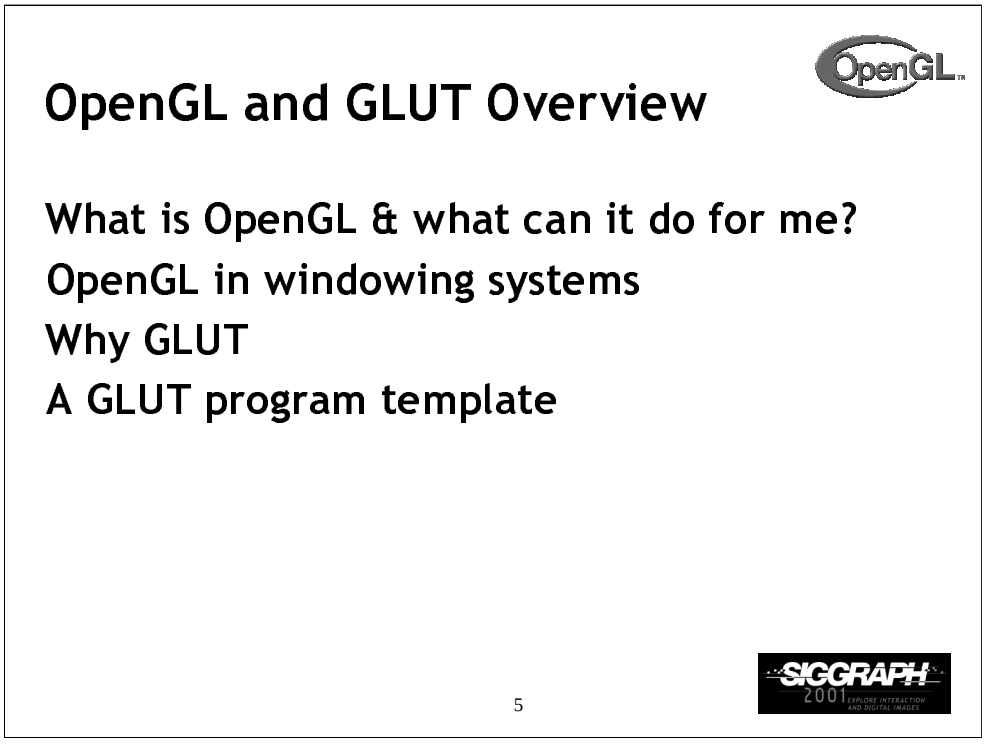


Today we hope to demonstrate the capabilities and flexibility of OpenGL such that you'll be able to author your own programs which can display 3D objects with lighting effects, shading, and custom texture maps.

Additionally, we'll introduce more advanced OpenGL topics for further personal investigation.

One of OpenGL's strengths is that its interface is easy to use for the novice, yet powerful enough to satisfy the requirement of professional applications, whether they be for flight simulation, animation, computer aided design, or scientific visualization.


OpenGL and GLUT Overview
4
SIGGRAPH
2001
EXPLORE INTERACTION
AND DIGITAL IMAGES

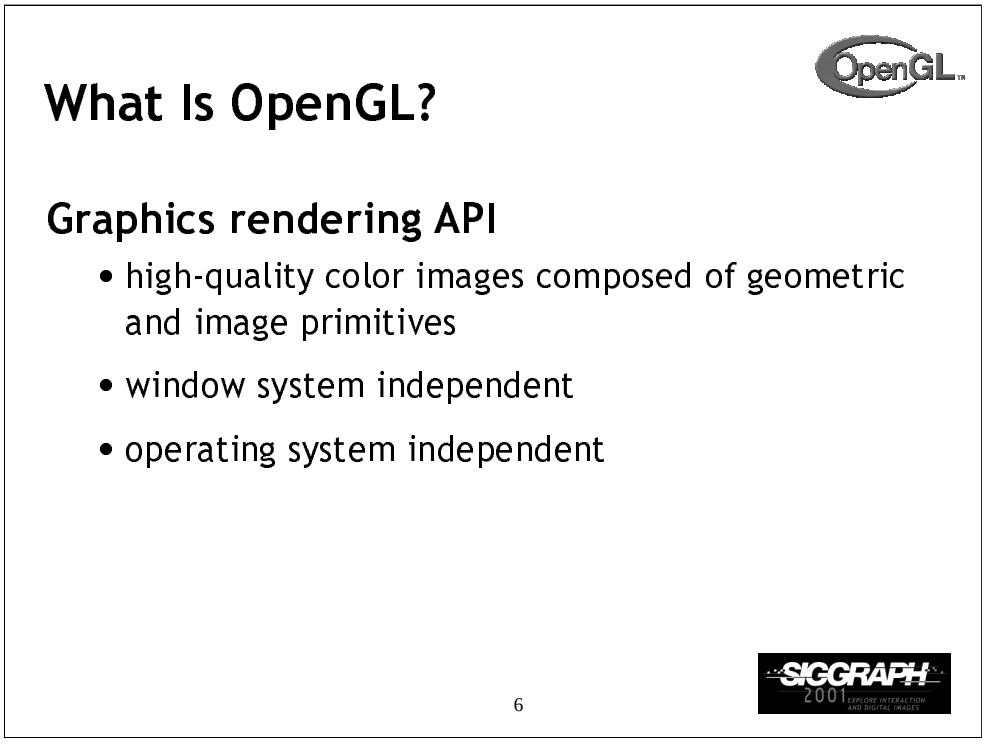


In this section, we discuss what the OpenGL API (Application Programming Interface) is, and some of its capabilities.

As OpenGL is platform independent, we need some way to integrate OpenGL into each windowing system. Every windowing system where OpenGL is supported has additional API calls for managing OpenGL windows, colormaps, and other features. These additional APIs are platform dependent.

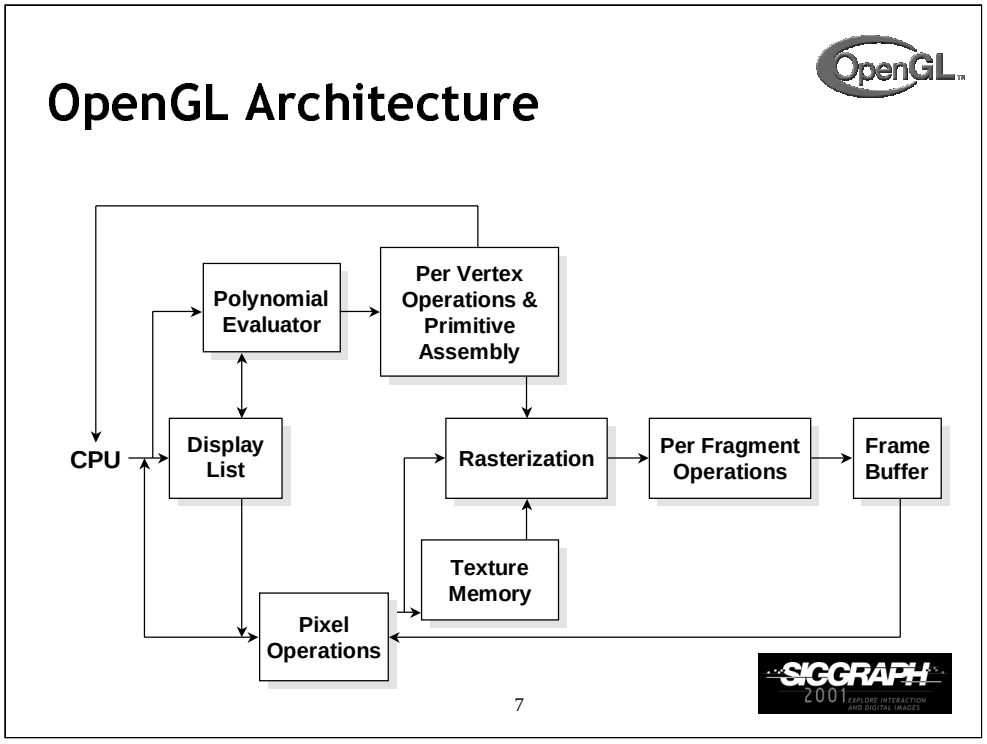
For the sake of simplicity, we'll use an additional freeware library for simplifying interacting with windowing systems, GLUT. GLUT, the OpenGL Utility Toolkit is a library to make writing OpenGL programs regardless of windowing systems much easier. It'll be the base of all of our examples in the class.

We conclude the section with a basic program template for an OpenGL program using GLUT.



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

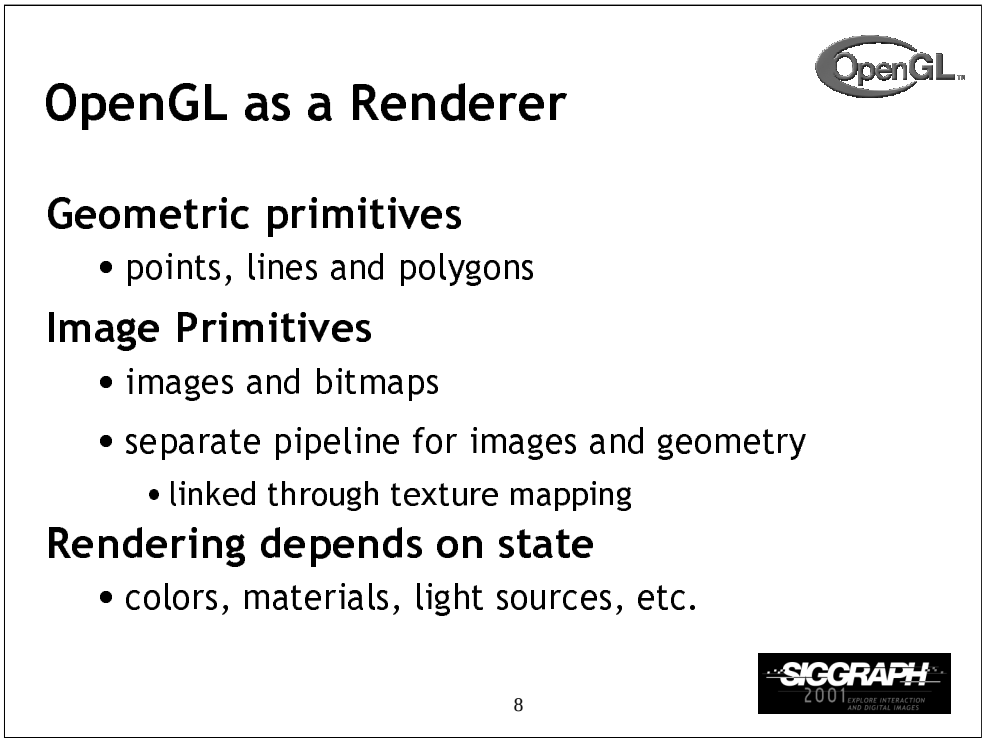
OpenGL is window and operating system independent. As such, the part of your application which does rendering is platform independent. However, in order for OpenGL to be able to render, it needs a window to draw into. Generally, this is controlled by the windowing system on whatever platform you're working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



This is the most important diagram you will see today, representing the flow of graphical information, as it is processed from CPU to the frame buffer.

There are two pipelines of data flow. The upper pipeline is for geometric, vertex-based primitives. The lower pipeline is for pixel-based, image primitives. Texturing combines the two types of primitives together.

There is a pull-out poster in the back of the OpenGL Reference Manual ("Blue Book"), which shows this diagram in more detail.

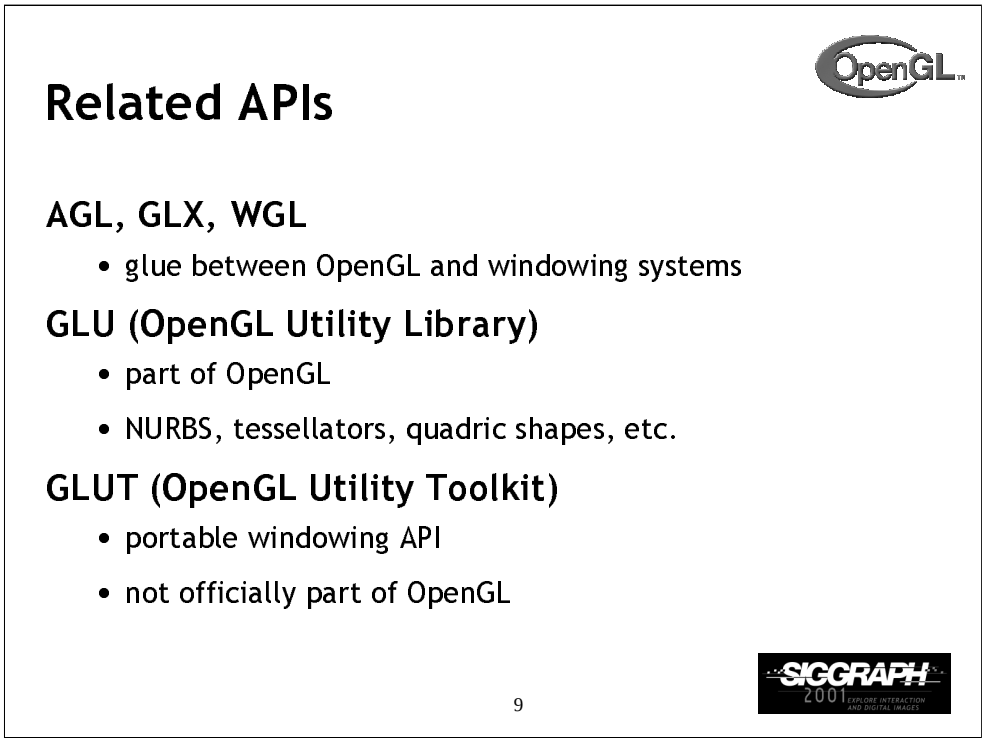


As mentioned, OpenGL is a library for rendering computer graphics. Generally, there are two operations that you do with OpenGL:

- 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've read it into your program.) Additionally, OpenGL links image and geometric primitives together using *texture mapping*, which is an advanced topic we'll 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 color that you want a vertex to be, to initializing multiple mipmap levels for texture mapping.

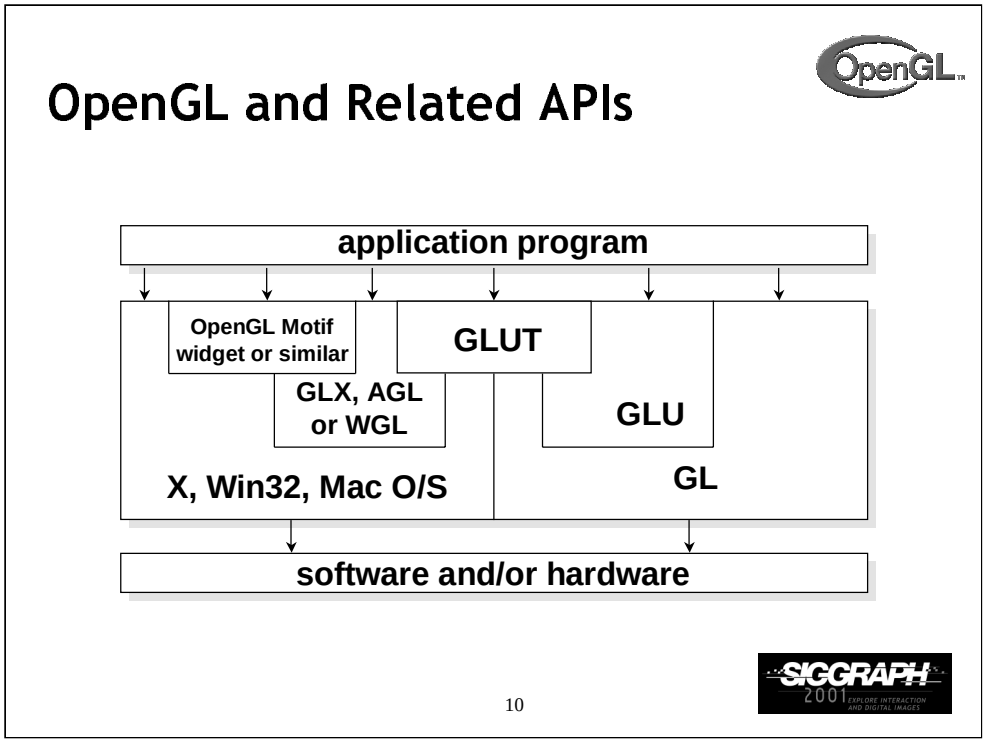


As mentioned, OpenGL is window and operating system independent. To integrate it into various window systems, additional libraries are used to modify a native window into an OpenGL capable window. Every window system has its own unique library and functions to do this. Some examples are:

- 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.

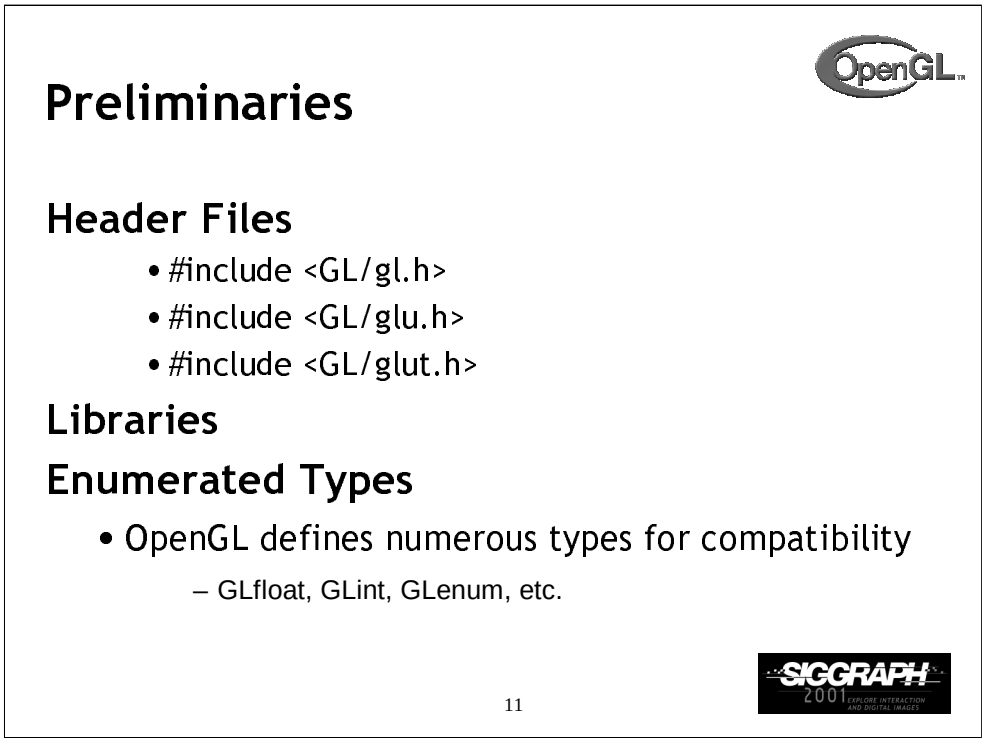
Finally to simplify programming and window system dependence, we'll be using the freeware library, GLUT. GLUT, written by Mark Kilgard, is a public domain window system independent toolkit for making simple OpenGL applications. It simplifies the process of creating windows, working with events in the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



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 that don't 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.

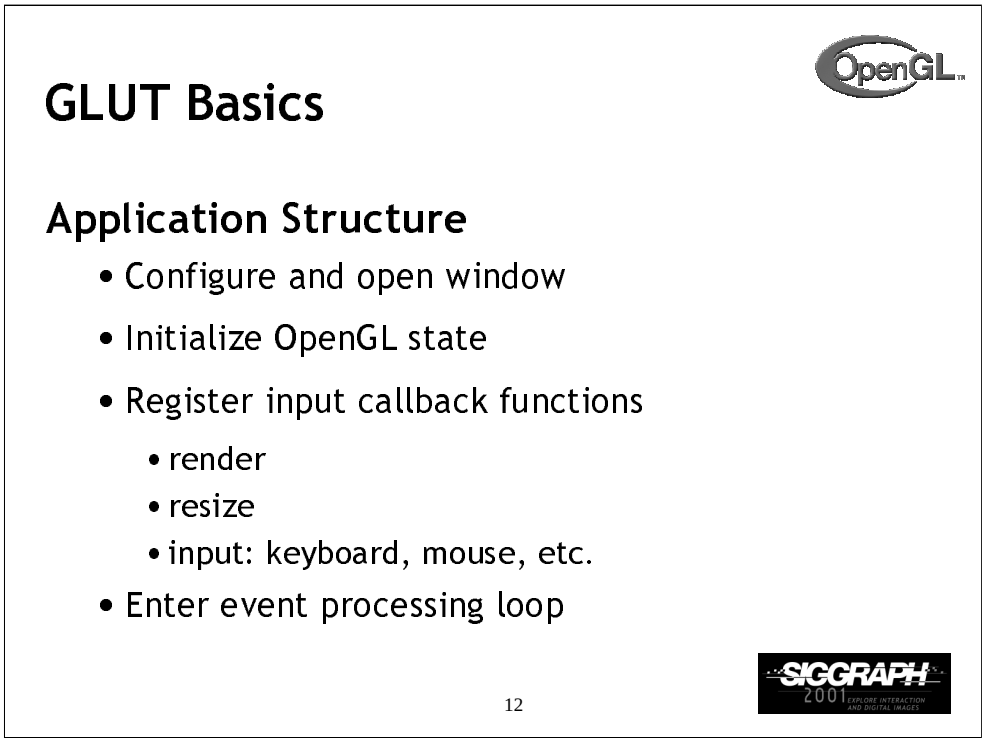
For C, there are a few required elements which an application must do:

- *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're using. Each operating system has its own set of libraries. For Unix systems, the OpenGL library is commonly named `libGL.so` and for Microsoft Windows, it's 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.



Here's the basic structure that we'll be using in our applications. This is generally what you'd do in your own OpenGL applications.

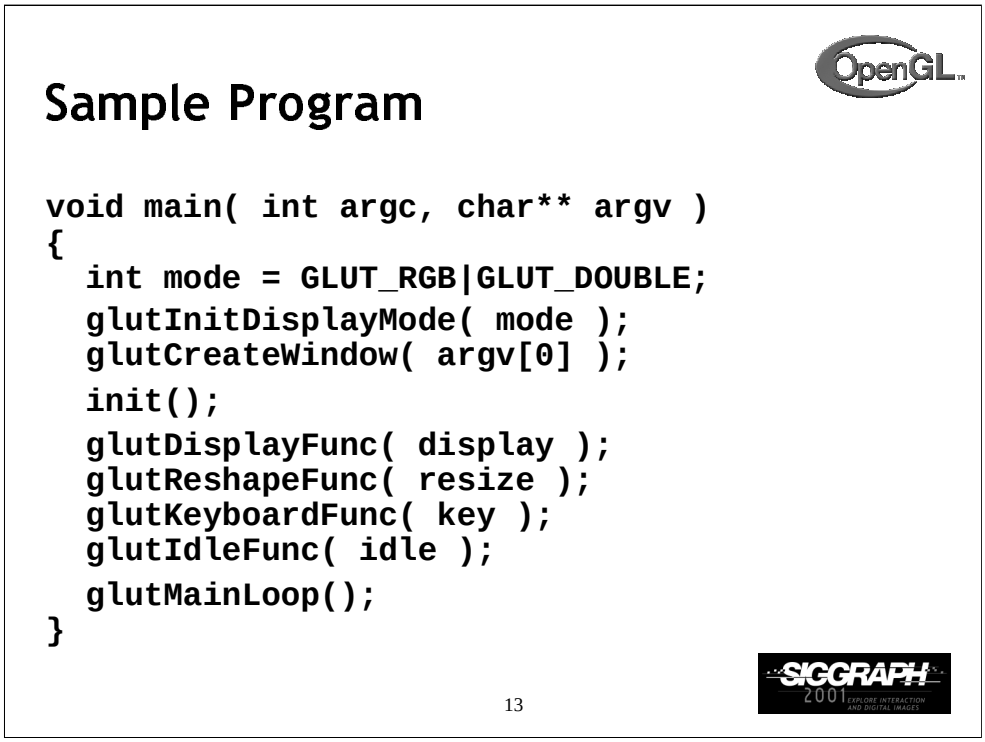
The steps are:

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

2) Initialize any OpenGL state that you don't 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'll 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'll 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.



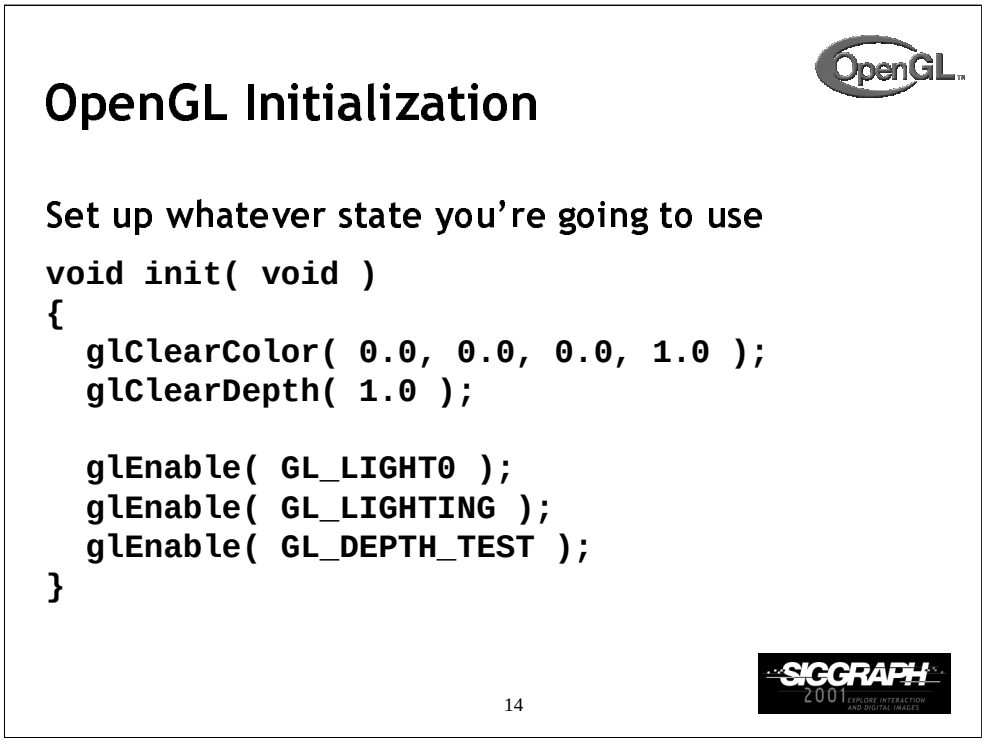
Here's an example of the main part of a GLUT based OpenGL application. This is the model that we'll use for most of our programs in the course.

The `glutInitDisplayMode()` and `glutCreateWindow()` functions compose the window configuration step.

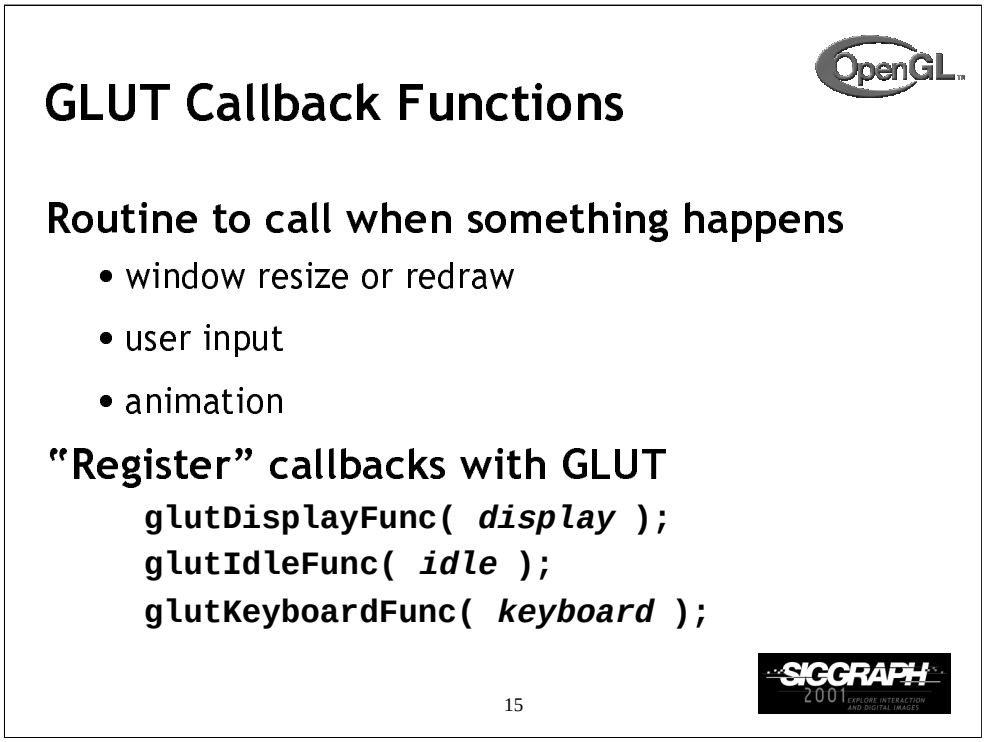
We then call the `init()` routine, which contains our one-time initialization. Here we initialize any OpenGL state and other program variables that we might need to use during our program that remain constant throughout the program's execution.

Next, we register the callback routines that we're going to use during our program.

Finally, we enter the event processing loop, which interprets events and calls our respective callback routines.



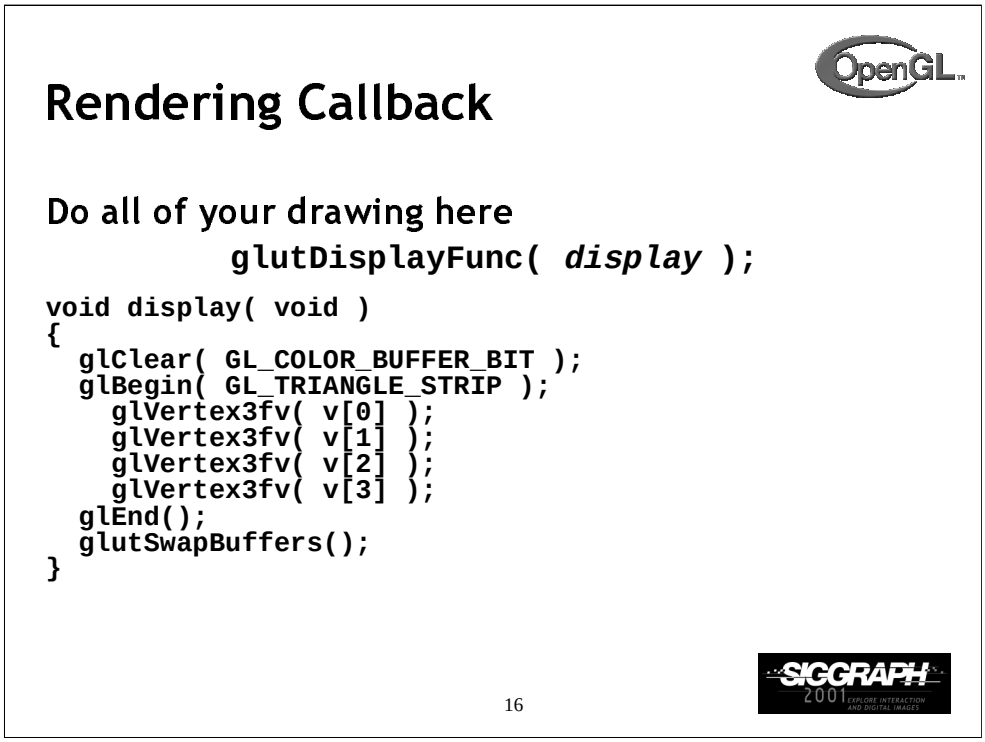
Here's the internals of our initialization routine, `init()`. Over the course of the day, you'll learn what each of the above OpenGL calls do.



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.

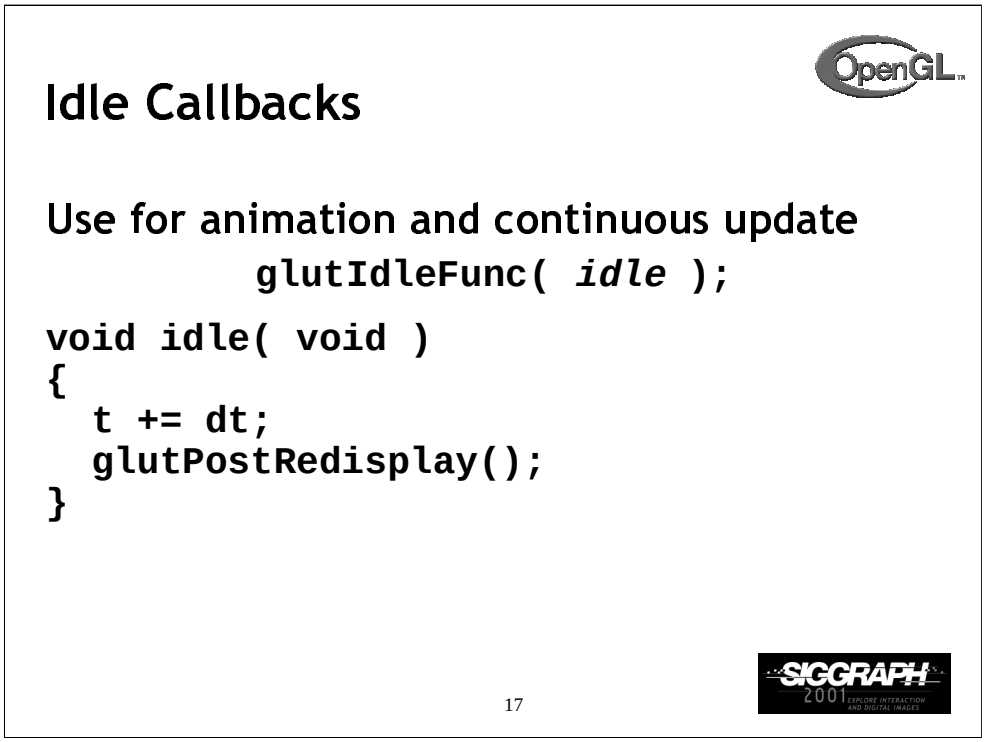
GLUT supports many different callback actions, including:

- `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.



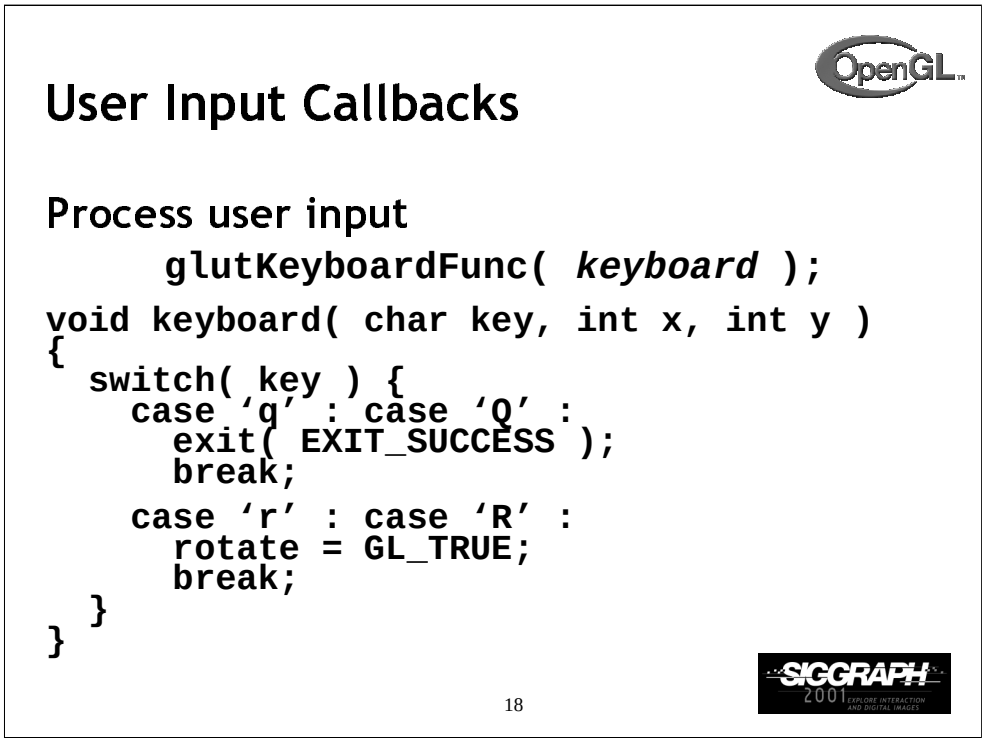
One of the most important callbacks is the `glutDisplayFunc()` callback. This callback is called when the window needs to be refreshed. It's here that you'd do all of your OpenGL rendering.

The above routine merely clears the window, and renders a triangle strip and then swaps the buffers for smooth animation transition. You'll learn more about what each of these calls do during the day.

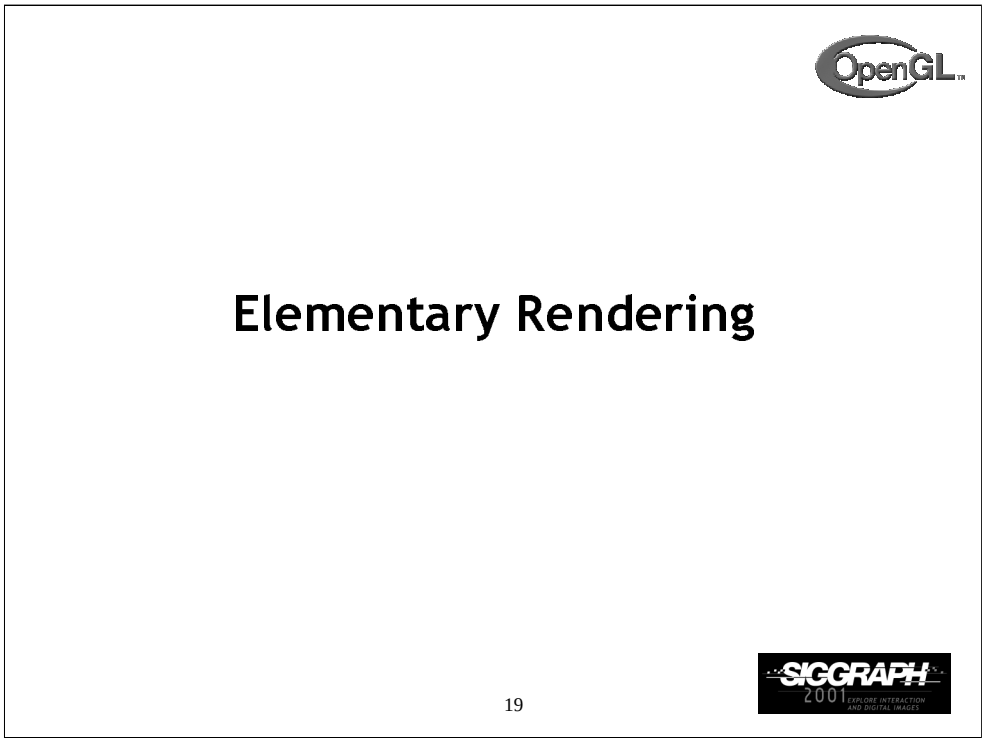


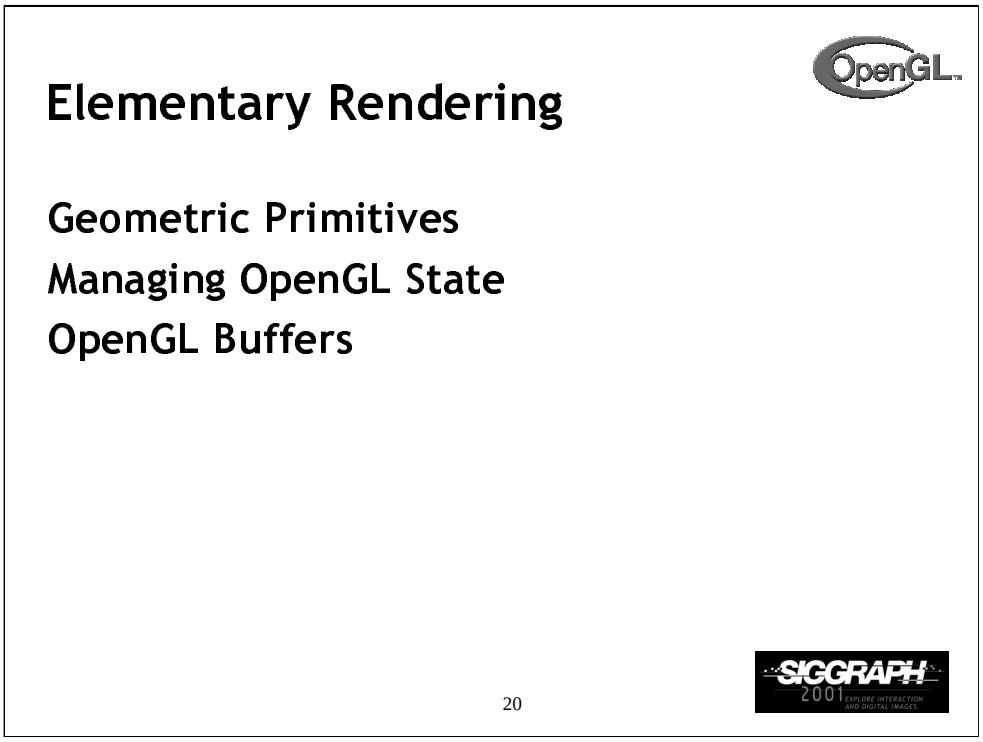
Animation requires the ability to draw a sequence of images. The `glutIdleFunc()` is the mechanism for doing animation. You register a routine which updates your *motion variables* (usually global variables in your program which control how things move) and then requests that the scene be updated.

`glutPostRedisplay()` requests that the callback registered with `glutDisplayFunc()` be called as soon as possible. This is preferred over calling your rendering routine directly, since the user may have interacted with your application and user input events need to be processed.



Above is a simple example of a user input callback. In this case, the routine was registered to receive keyboard input. GLUT supports user input through a number of devices including the keyboard, mouse, dial and button boxes and spaceballs.

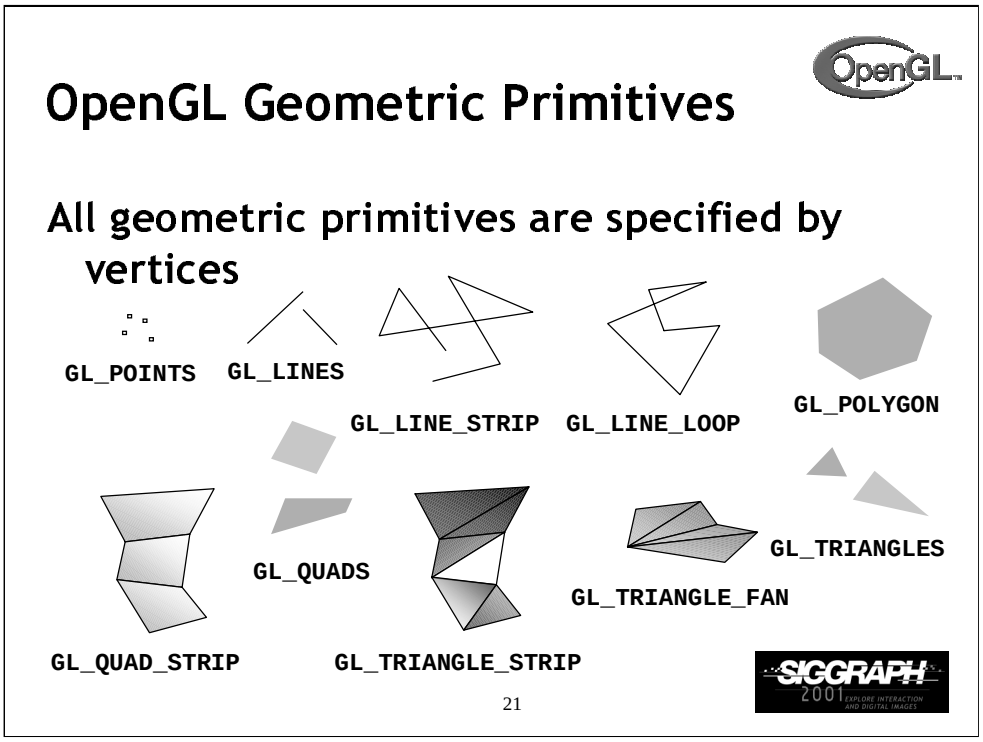

OpenGL
Elementary Rendering
19
SIGGRAPH
2001
EXPLORE INTERACTION
AND DIGITAL IMAGES



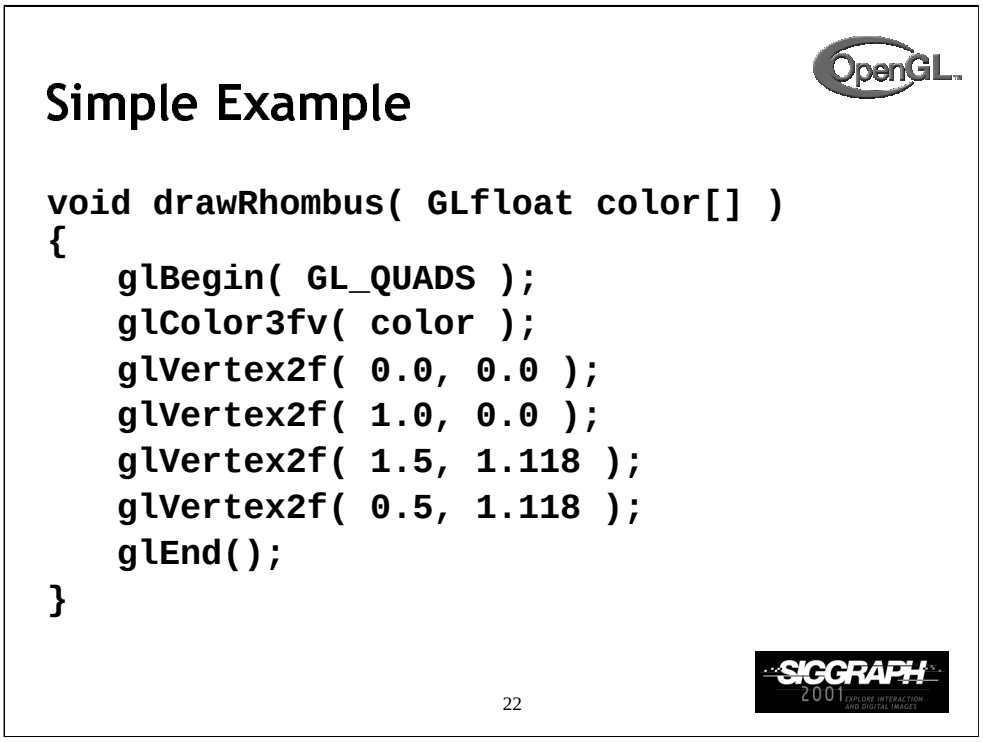
In this section, we'll be discussing the basic geometric primitives that OpenGL uses for rendering, as well as how to manage the OpenGL state which controls the appearance of those primitives.

OpenGL also supports the rendering of bitmaps and images, which is discussed in a later section.

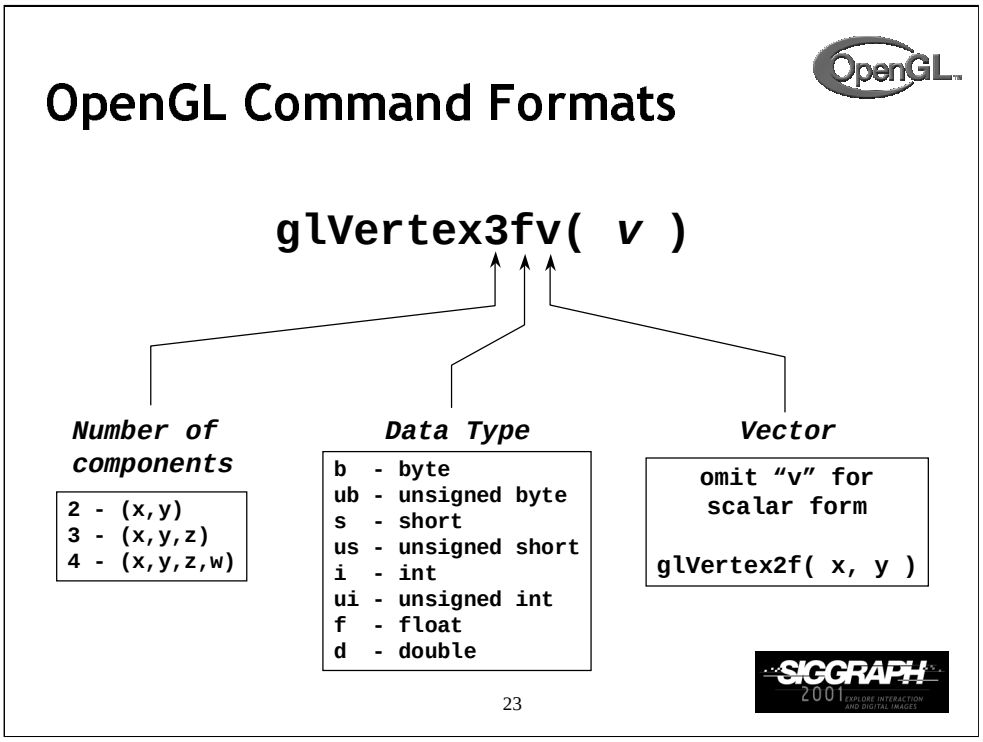
Additionally, we'll discuss the different types of OpenGL buffers, and what each can be used for.



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.



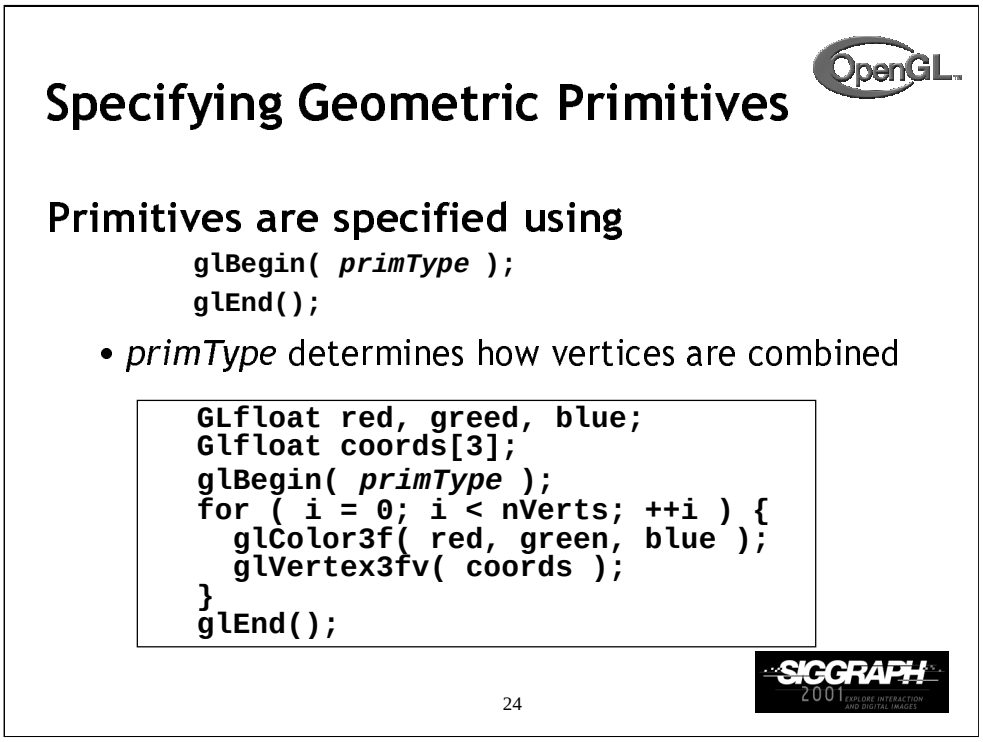
The `drawRhombus()` routine causes OpenGL to render a single quadrilateral in a single color. The rhombus is planar, since the *z* value is automatically set to 0.0 by `glVertex2f()`.



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.

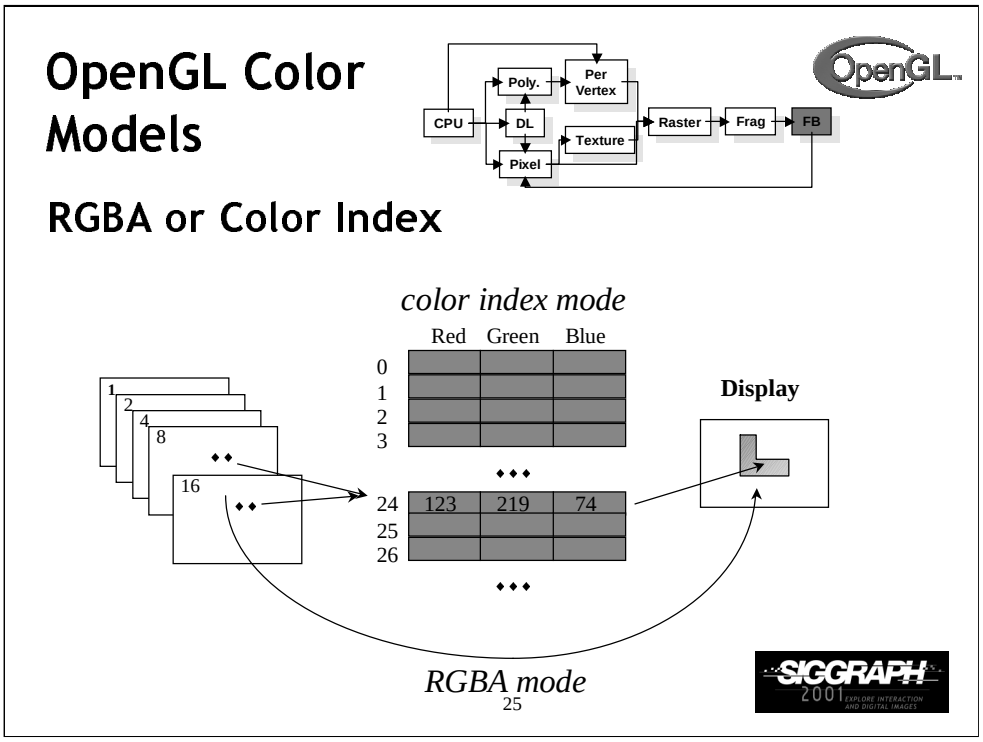
For instance, vertices from most commercial models are stored as three component floating point vectors. As such, the appropriate OpenGL command to use is `glVertex3fv`(coords).

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



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

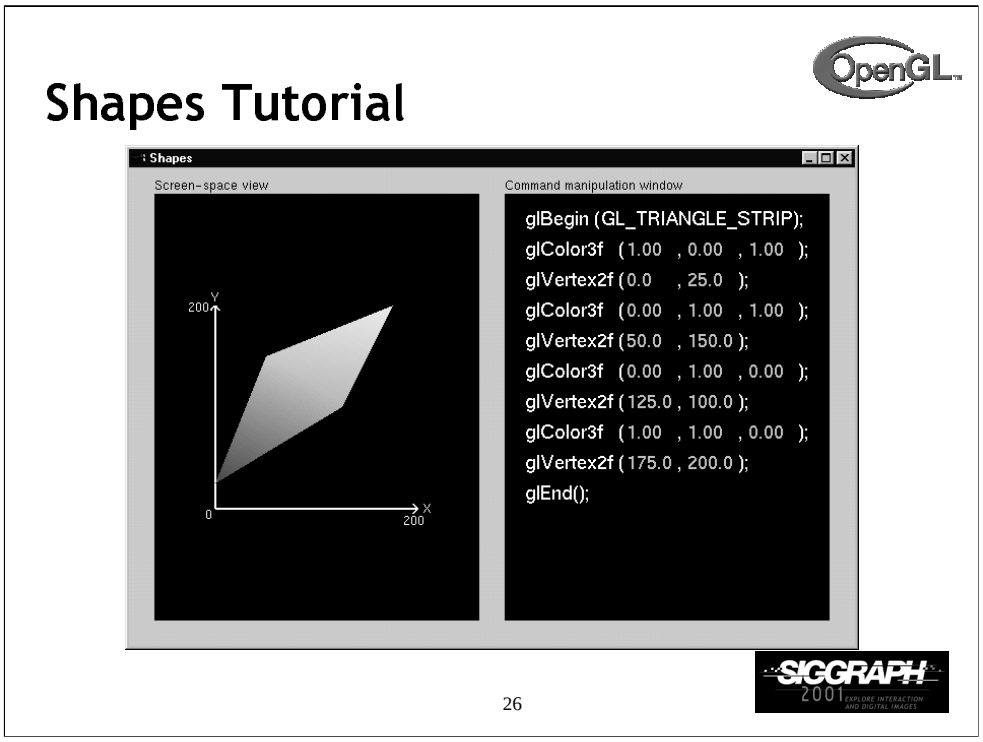


Every OpenGL implementation must support rendering in both RGBA mode, (sometimes described as *TrueColor* mode) and color index (or *colormap*) mode.

For RGBA rendering, vertex colors are specified using the `glColor*()` call.

For color index rendering, the vertex's index is specified with `glIndex*()`.

The type of window color model is requested from the windowing system. Using GLUT, the `glutInitDisplayMode()` call is used to specify either an RGBA window (using `GLUT_RGBA`), or a color indexed window (using `GLUT_INDEX`).



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.

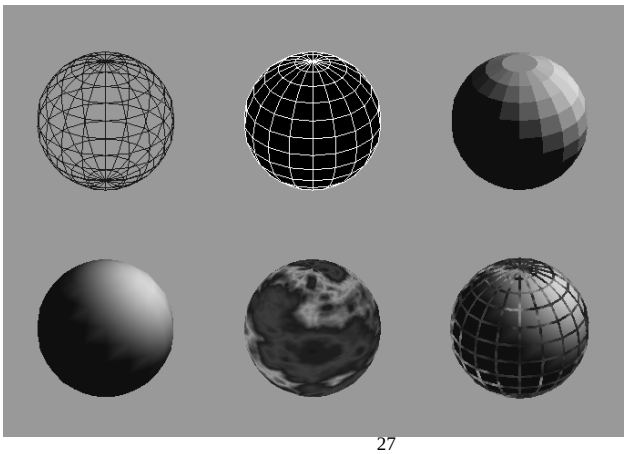
In the command manipulation window, pressing the LEFT mouse while the pointer is over the green parameter numbers allows you to move the mouse in the y-direction (up and down) and change their values. With this action, you can change the appearance of the geometric primitive in the other window. With 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 don't 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.

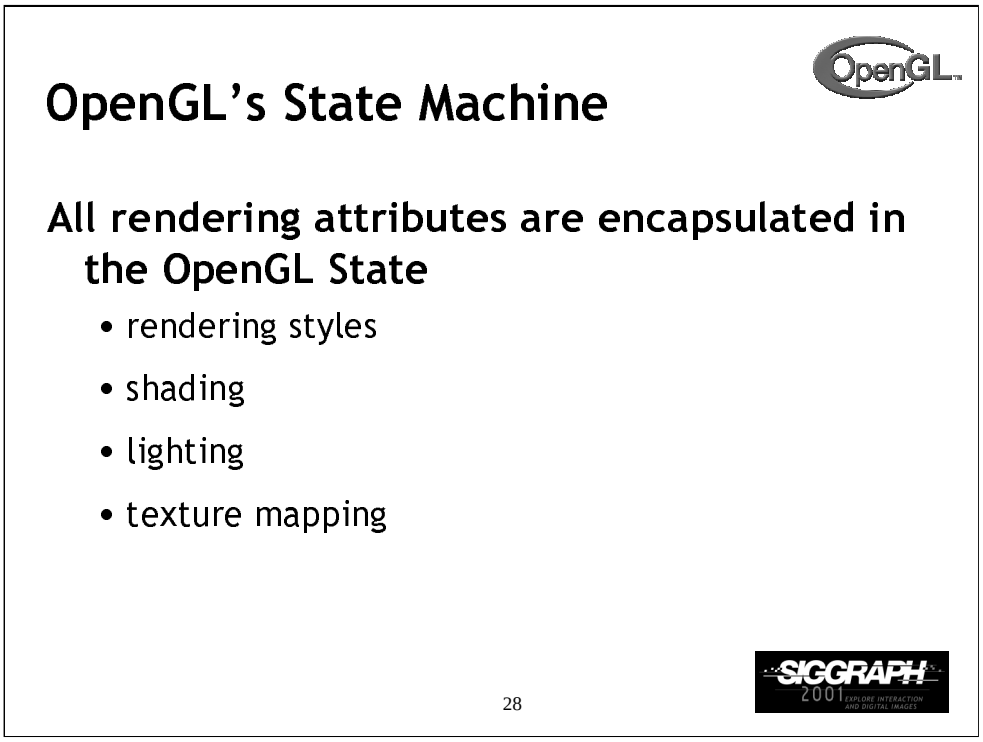
Controlling Rendering Appearance

From Wireframe to Texture Mapped

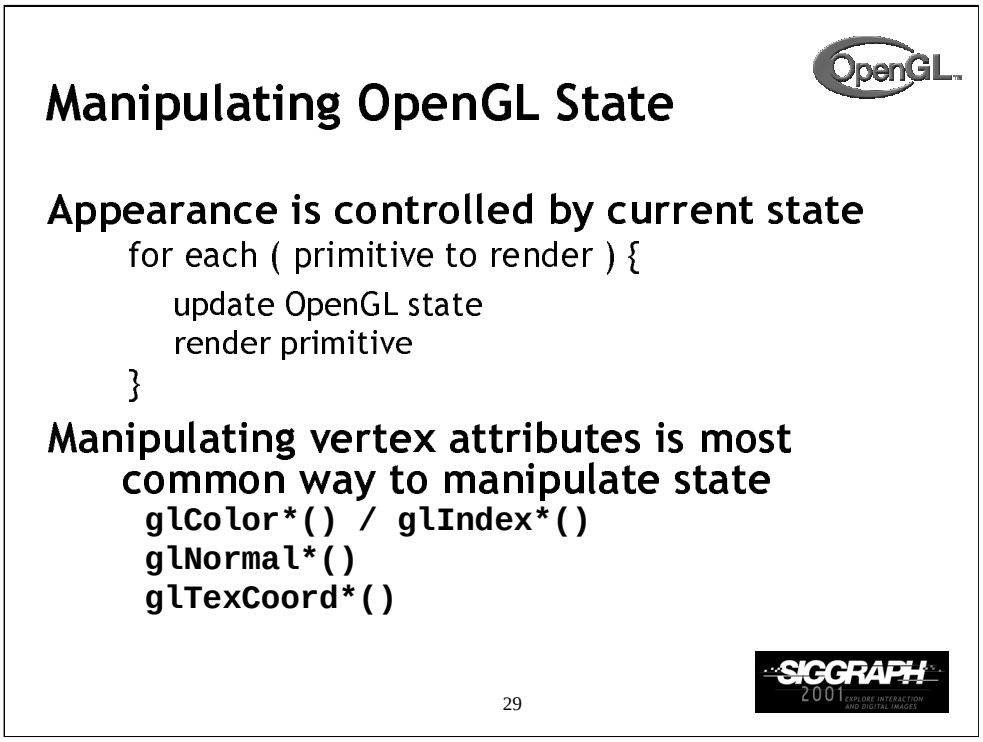


27

OpenGL can render from a simple line-based wireframe to complex multi-pass texturing algorithms to simulate bump mapping or Phong lighting.

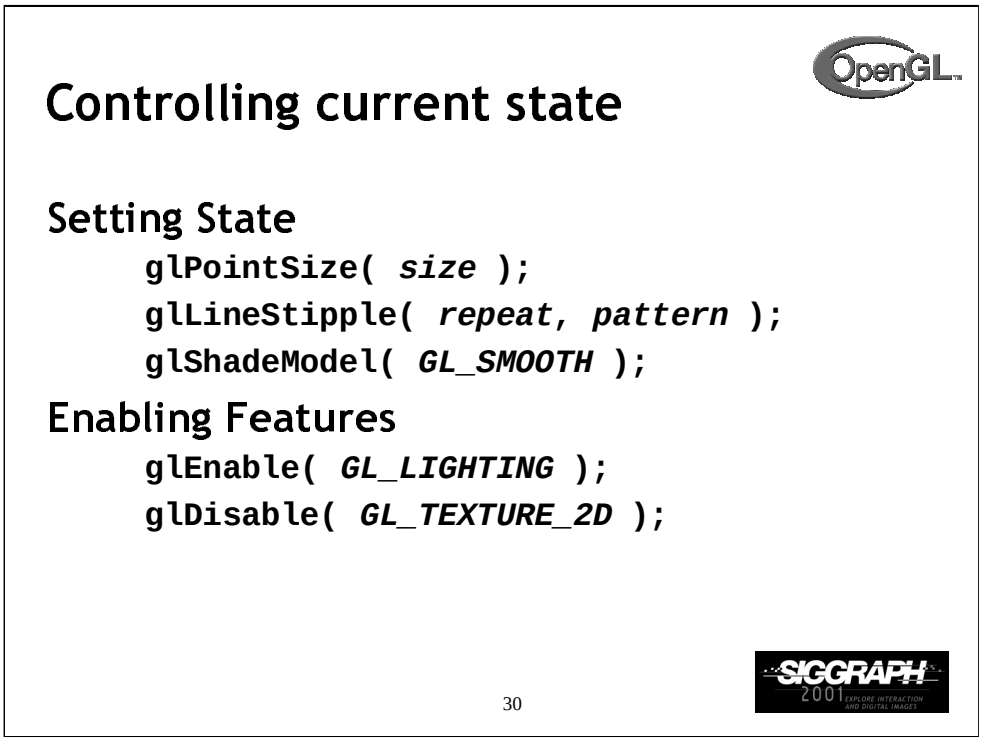


Each time OpenGL processes a vertex, it uses data stored in its internal state tables to determine how the vertex should be transformed, lit, textured or any of OpenGL's other modes.



The general flow of any OpenGL rendering is to set up the required state, then pass the primitive to be rendered, and repeat for the next primitive.

In general, the most common way to manipulate OpenGL state is by setting vertex attributes, which include color, lighting normals, and texturing coordinates.

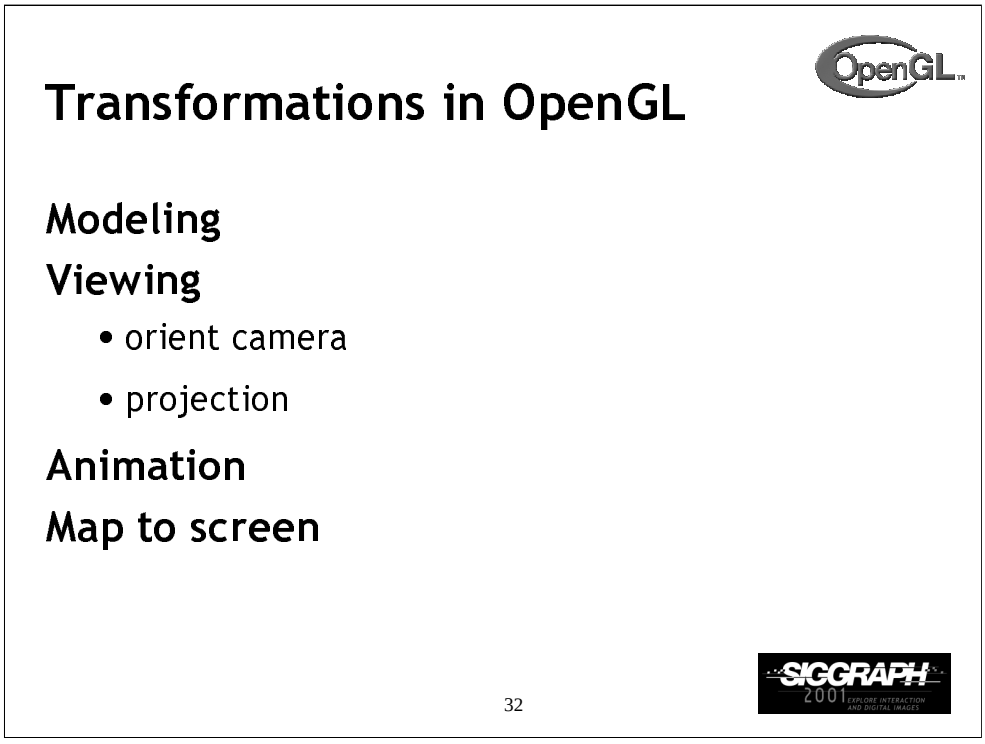


Setting OpenGL state usually includes modifying the rendering attribute, such as loading a texture map, or setting the line width. Also for some state changes, setting the OpenGL state also enables that feature (like setting the point size or line width).

Other features need to be turned on. This is done using `glEnable()`, and passing the token for the feature, like `GL_LIGHT0` or `GL_POLYGON_STIPPLE`.

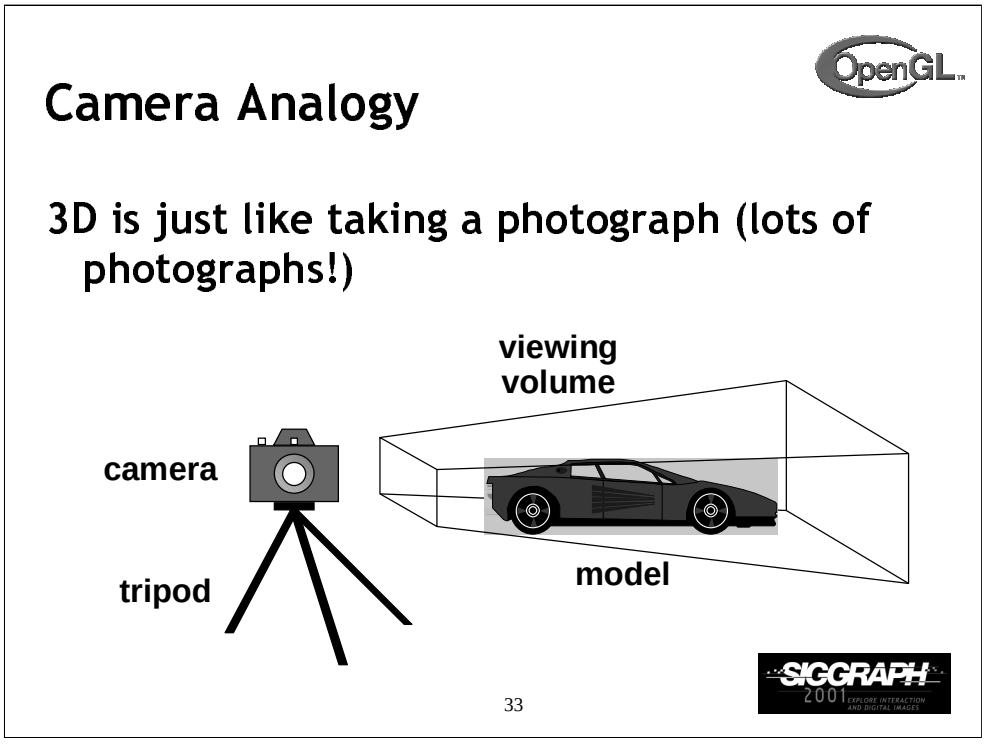
Transformations

31



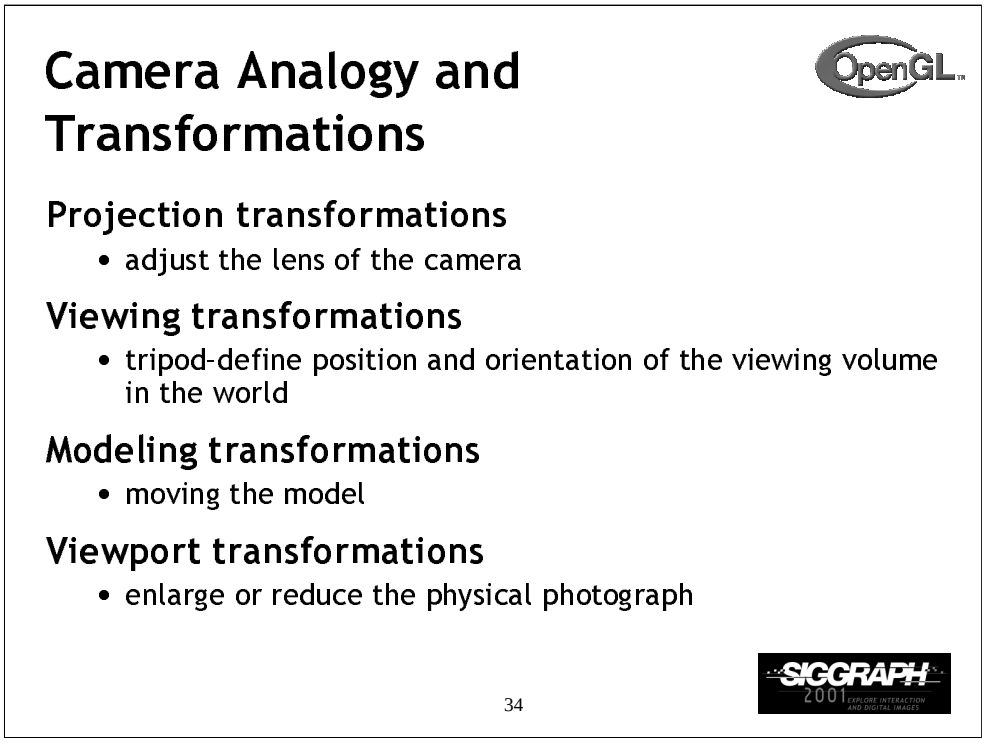
Transformations are used both by the applications programmer to move and orient objects (either statically or dynamically) and by OpenGL to implement the viewing pipeline.

Three transformations (ModelView, perspective, texture) are part of the state. Their matrices can be set by application programs but the operations are carried out within the viewing pipeline.

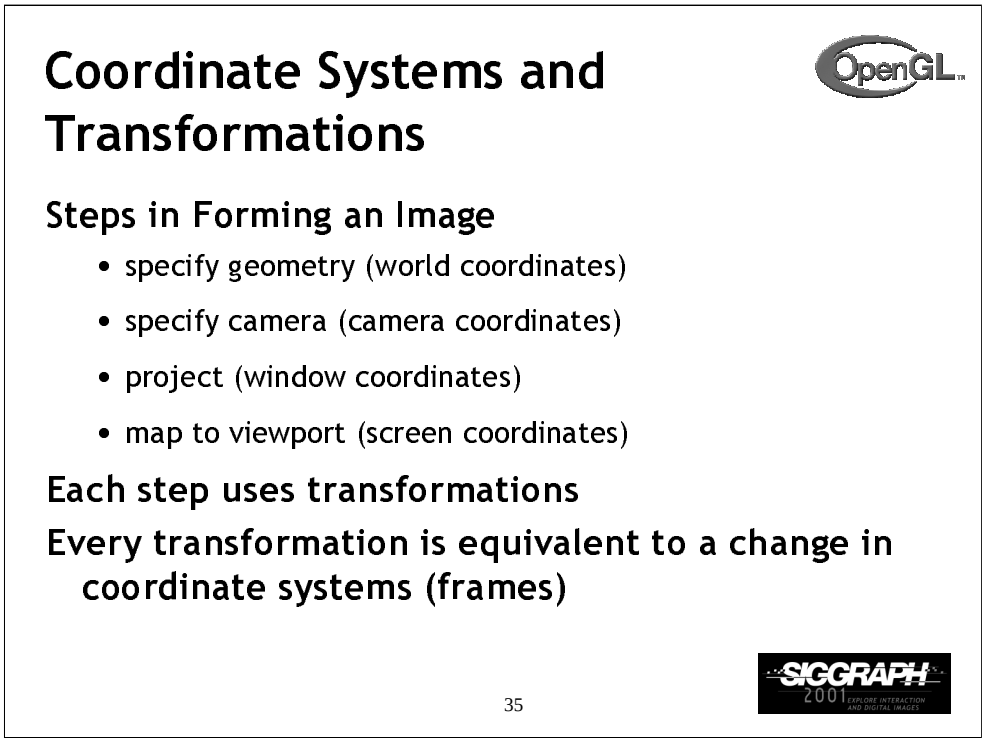


This model has become know as the synthetic camera model.

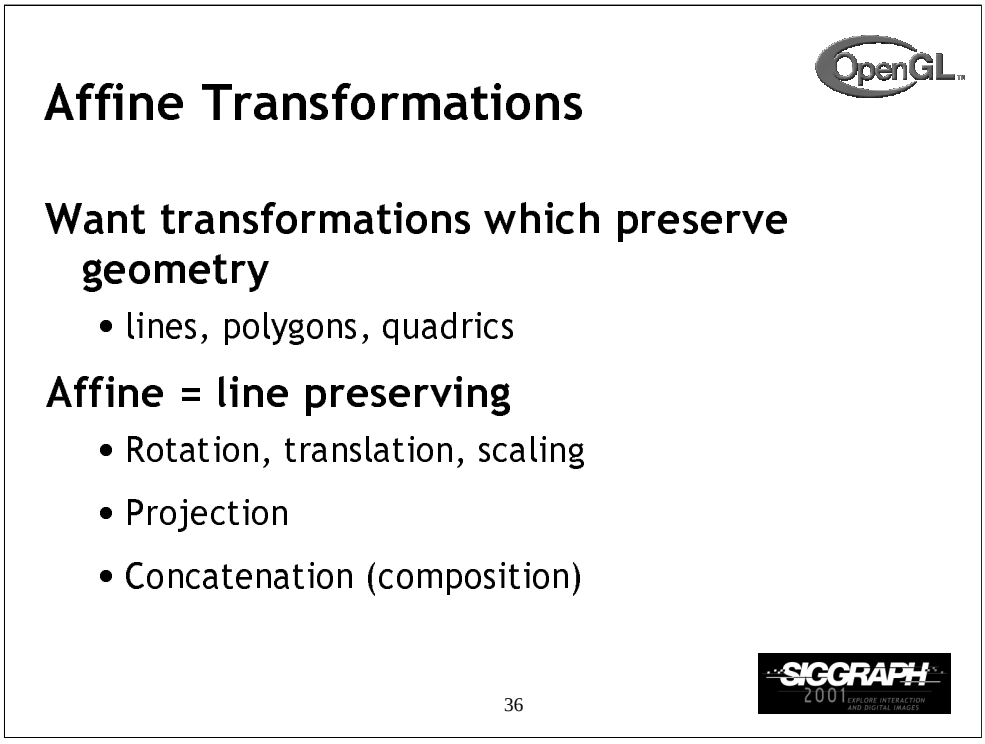
Note that both the objects to be viewed and the camera are three-dimensional while the resulting image is two dimensional.



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.



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 ModelView matrix). Similarly, the projection matrix converts from camera coordinates to window coordinates.



The transformations supported by OpenGL are a special class that is important for graphical applications and for problems in science and engineering. In particular, affine transformations will not alter the type of object. A transformed line (polygon, quadric) is still a line (polygon, quadric).

Any composition of affine transformations is still affine. For example, a rotation followed by a translation followed by a projection preserves lines and polygons.

Homogeneous Coordinates

- each vertex is a column vector

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

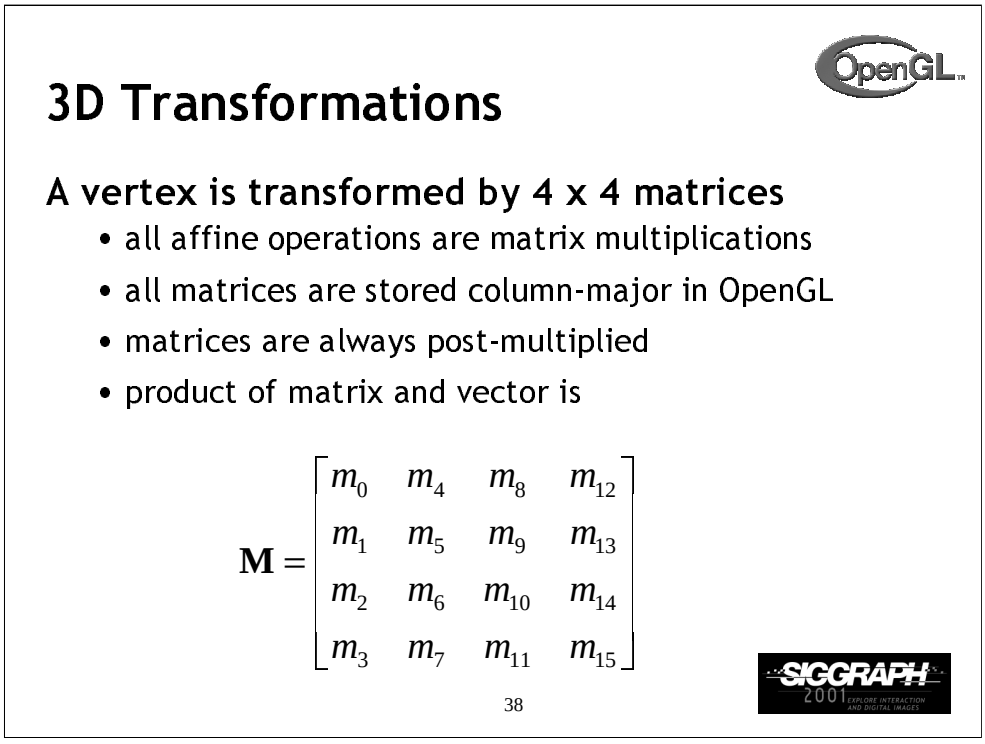
- w is usually 1.0
- all operations are matrix multiplications
- directions (directed line segments) can be represented with $w = 0.0$

37

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.

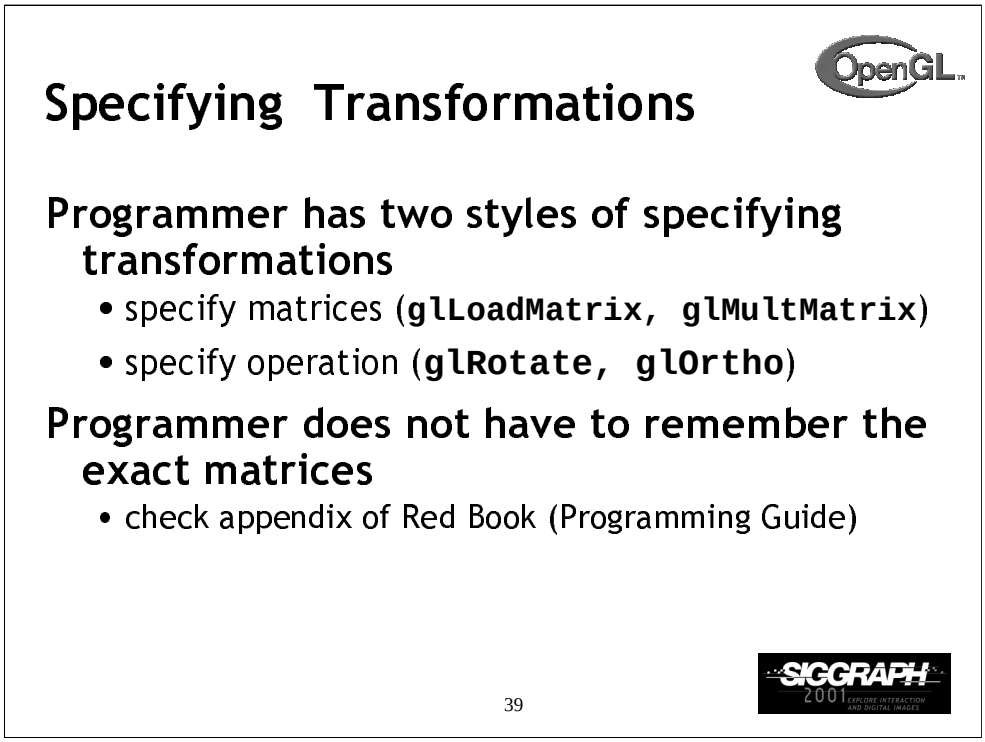
If w is changed from 1.0, we can recover x, y and z by division by w. Generally, only perspective transformations change w and require this perspective division in the pipeline.



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

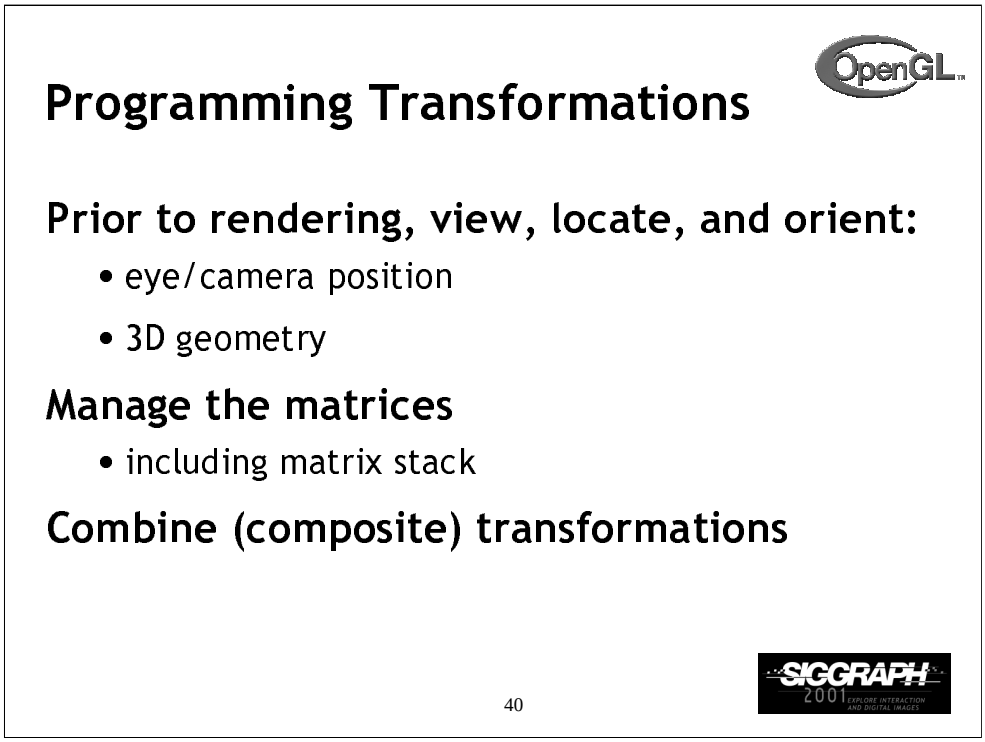
For operations other than perspective projection, the fourth row is always (0, 0, 0, 1) which leaves w unchanged..

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



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()`.

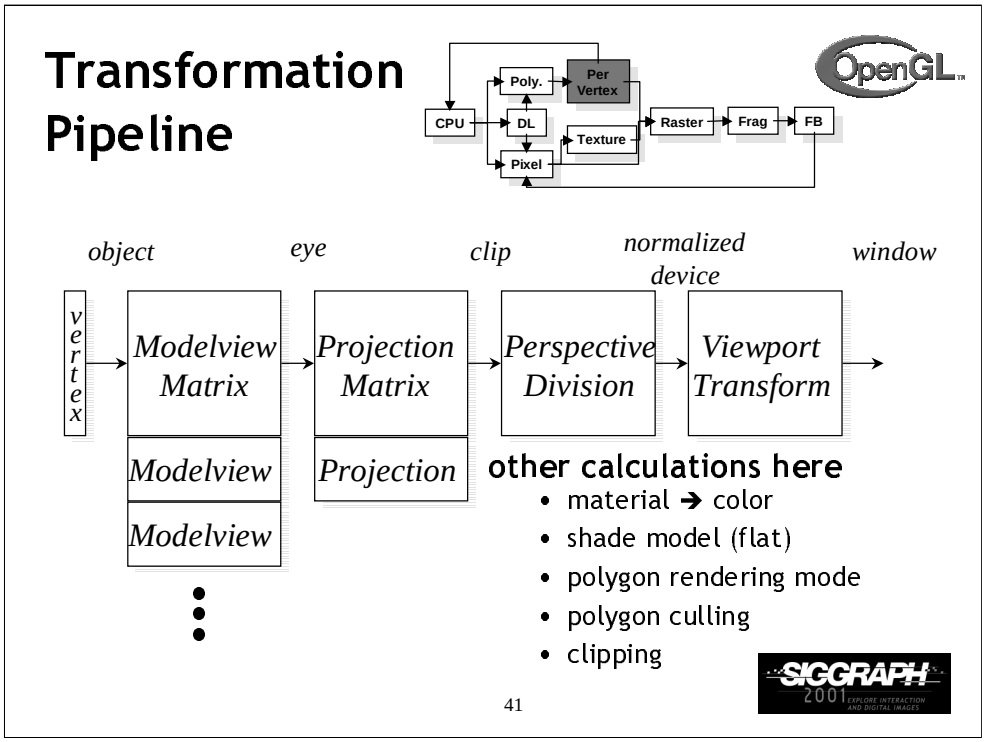
For the basic viewing transformations, OpenGL and the Utility library have supporting functions.



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.

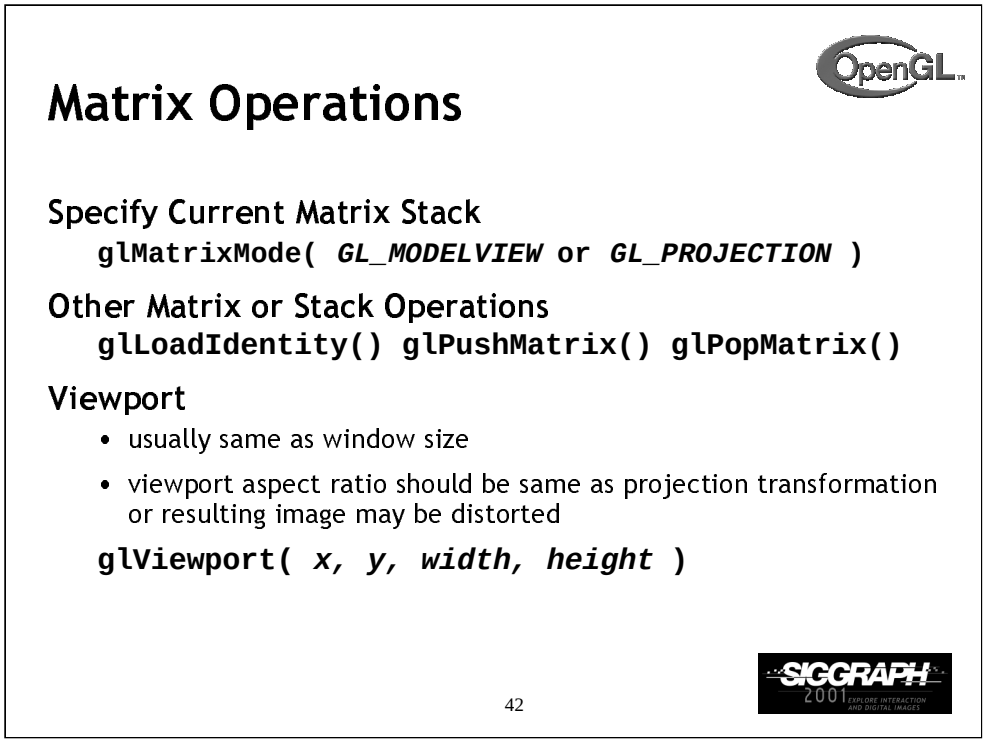
OpenGL provides matrix stacks for each type of supported matrix (ModelView, projection, texture) to store matrices.



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.



`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.

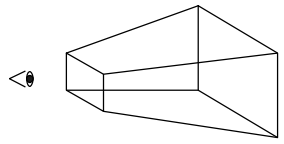
Matrix stacks are used because it is more efficient to save and restore matrices than to calculate and multiply new matrices. Popping a matrix stack can be said to "jump back" to a previous location or orientation.

`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 doesn't. 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

43

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 negative *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 same as the viewport to avoid distortion. `zNear` and `zFar` define the distance to the near and far clipping planes.

`glFrustum()` is rarely used.

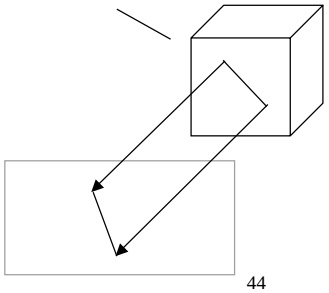
Warning: for `gluPerspective()` or `glFrustum()`, don't use zero for `zNear`!

For `glOrtho()`, the viewing volume is shaped like a rectangular parallelepiped (a box). Vertexes of an object are "projected" towards infinity. Distance does not change the apparent size of an object. 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 );
```



44

Many users would follow the demonstrated sequence of commands with a `glMatrixMode(GL_MODELVIEW)` call to return to ModelView matrix 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

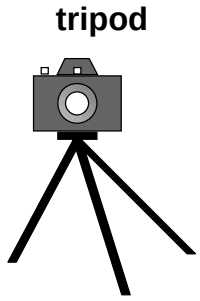
- place the tripod down; aim camera

To "fly through" a scene

- change viewing transformation and redraw scene

```
gluLookAt( eyex, eyey, eyez,
           aimx, aimy, aimz,
           upx, upy, upz )
```

- up vector determines unique orientation
- careful of degenerate positions



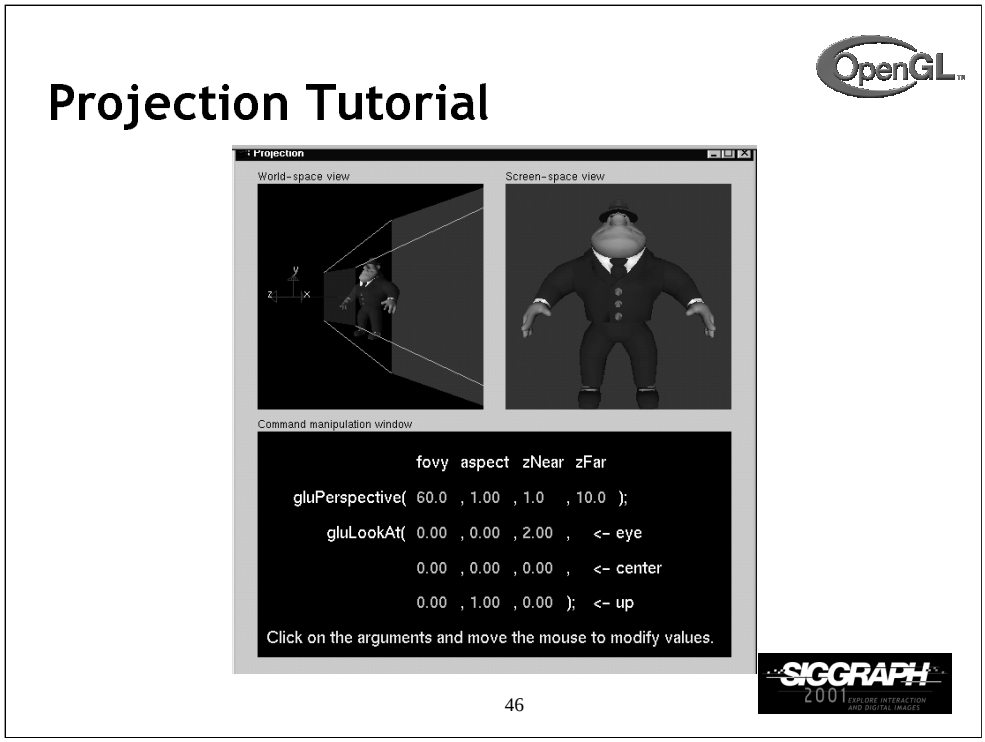
45

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

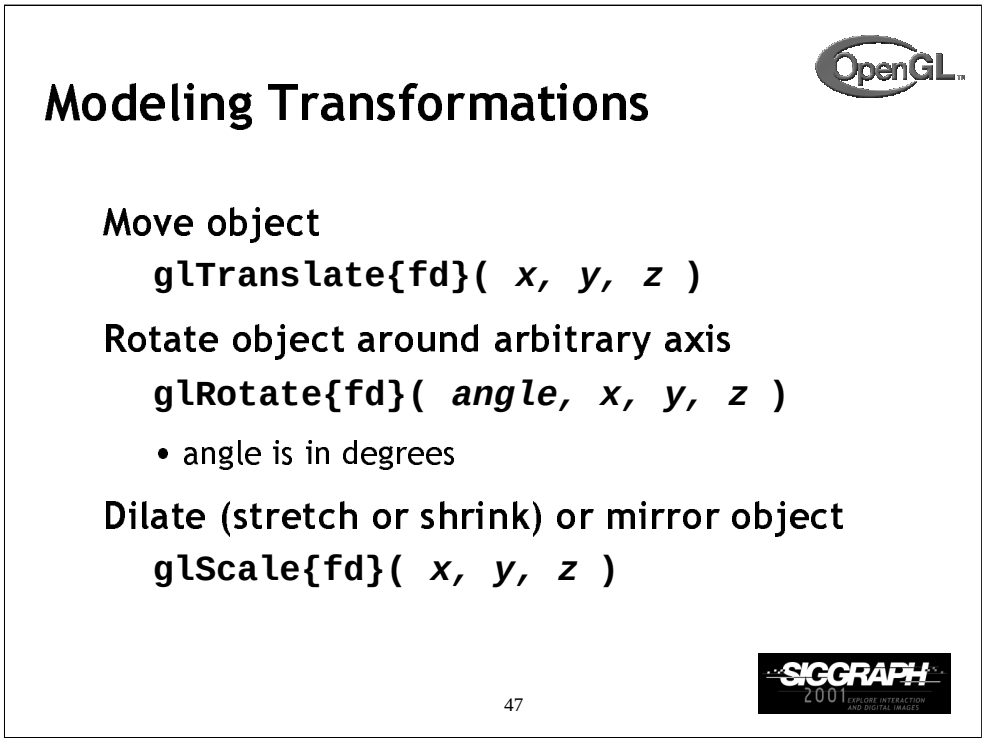
Because of degenerate positions, `gluLookAt()` is not recommended for most animated fly-over applications.

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.

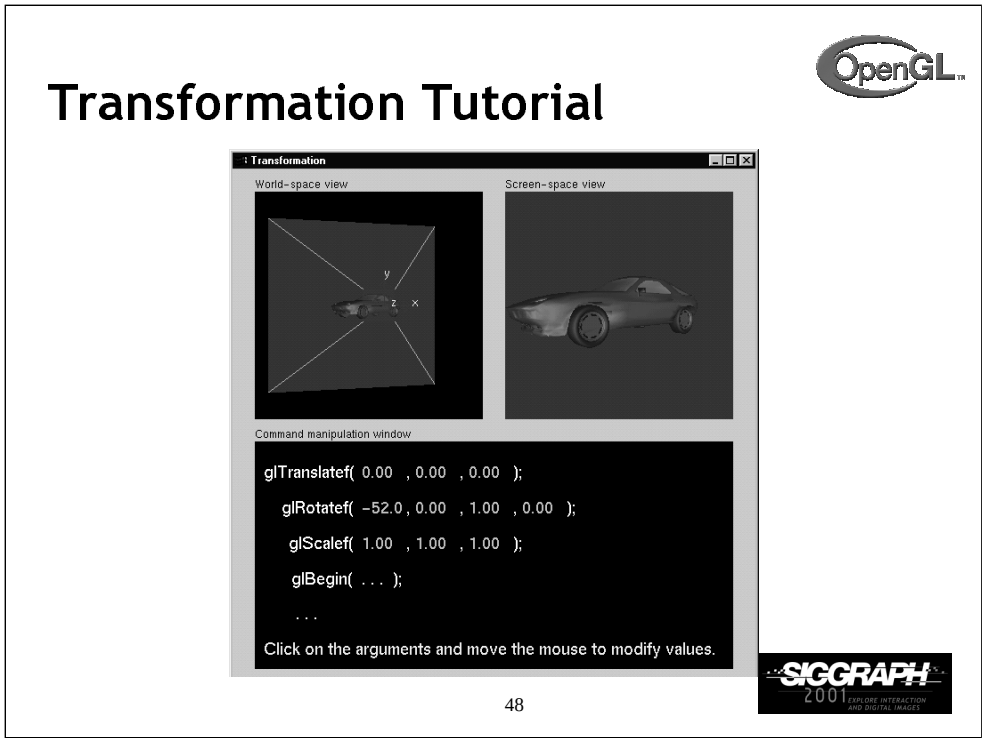


The RIGHT mouse button controls different menus. The screen-space view menu allows you to choose different models. The command-manipulation menu allows you to select different projection commands (including glOrtho and glFrustum).



`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*

49

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"):

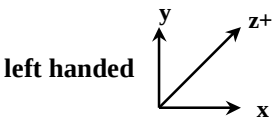
```
glRotated(roll, 0, 0, 1)
glRotated(pitch, 0, 1, 0)
glRotated(heading, 1, 0, 0)
glTranslated(-x, -y, -z)
```

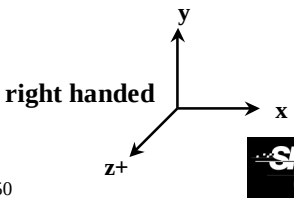
Projection is left handed

Projection transformations (`gluPerspective`, `glOrtho`) are left handed

- think of ***zNear*** and ***zFar*** as distance from view point

Everything else is right handed, including the vertexes to be rendered

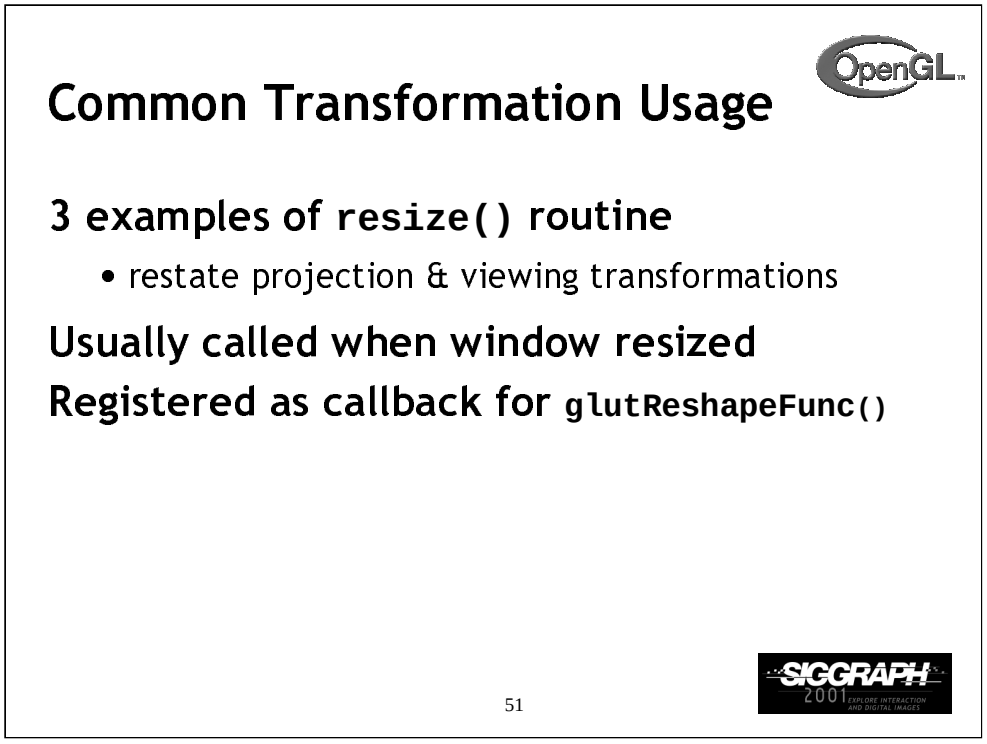




50

If you get this wrong, you may see nothing in your image. Switching from right to left handed coordinates is equivalent to rotating the camera 180 degrees!

One way to think of problem is that the viewing system is left-handed so distances from the camera are measured from the camera to the object.



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.

resize(): Perspective & LookAt

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

52

Using the viewport width and height as the aspect ratio for `gluPerspective` eliminates distortion.

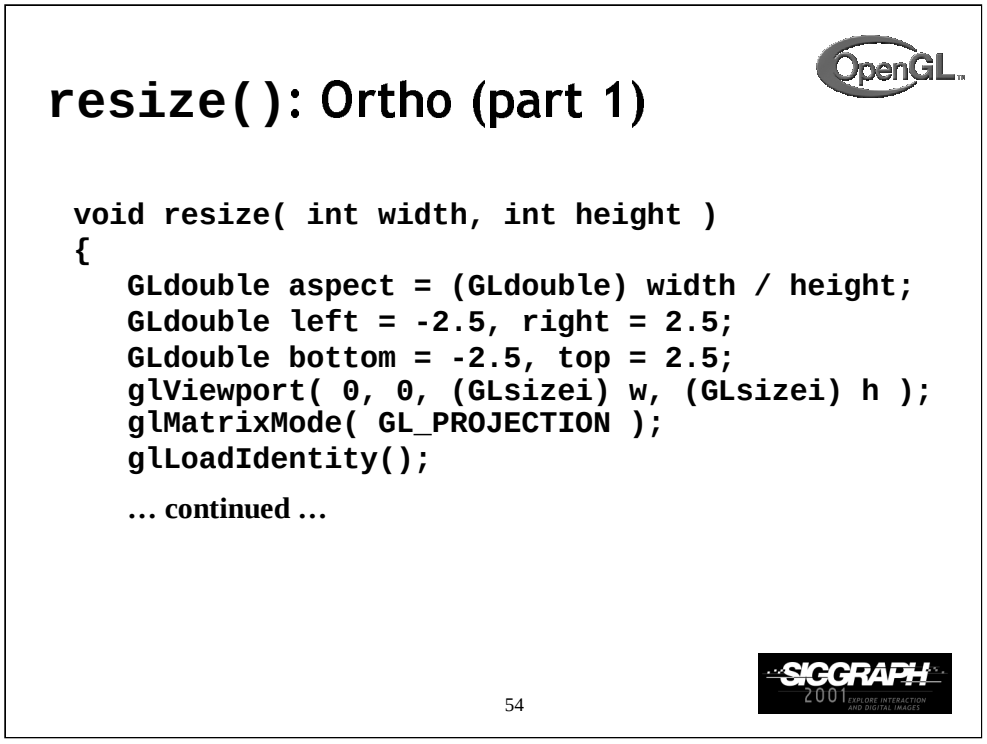
resize(): Perspective & Translate

Same effect as previous LookAt

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

53

Moving all objects in the world five units away from you is mathematically the same as “backing up” the camera five units.



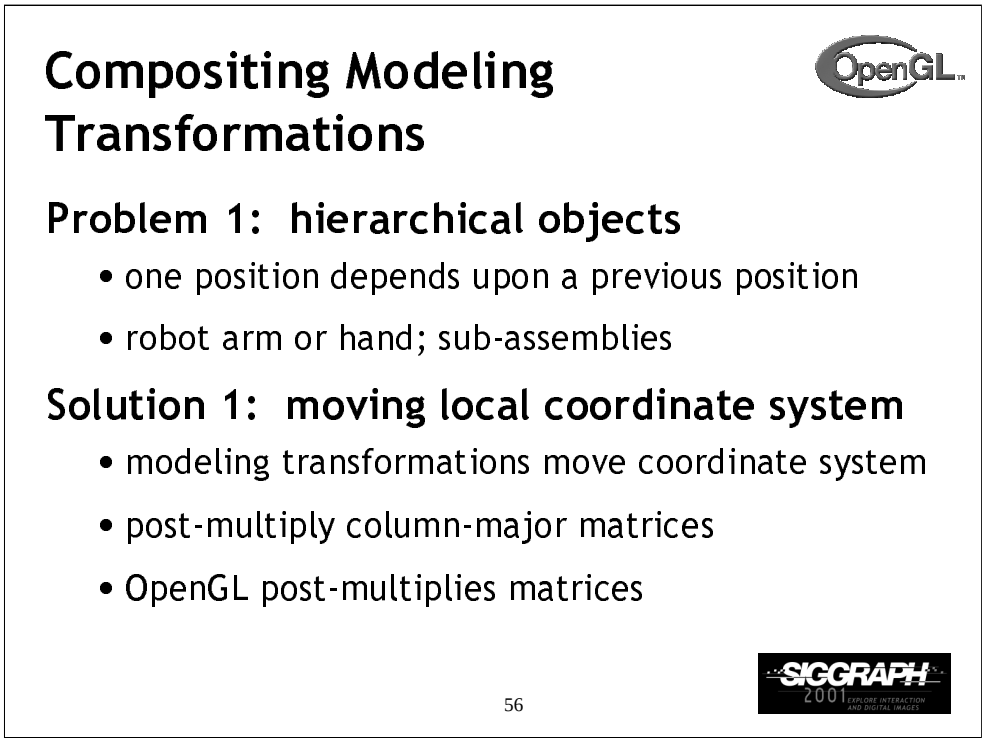
In this routine, we first compute the aspect ratio (`aspect`) of the new viewing area. We'll use this value to modify the world space values (`left`, `right`, `bottom`, `top`) of the viewing frustum depending on the new shape of the viewing volume

resize(): Ortho (part 2)

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

55

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.

Compositing Modeling Transformations

Problem 2: objects move relative to absolute world origin

- my object rotates around the wrong origin
  - make it spin around its center or something else

Solution 2: fixed coordinate system

- modeling transformations move objects around fixed coordinate system
- pre-multiply column-major matrices
- OpenGL post-multiplies matrices
- must reverse order of operations to achieve desired effect

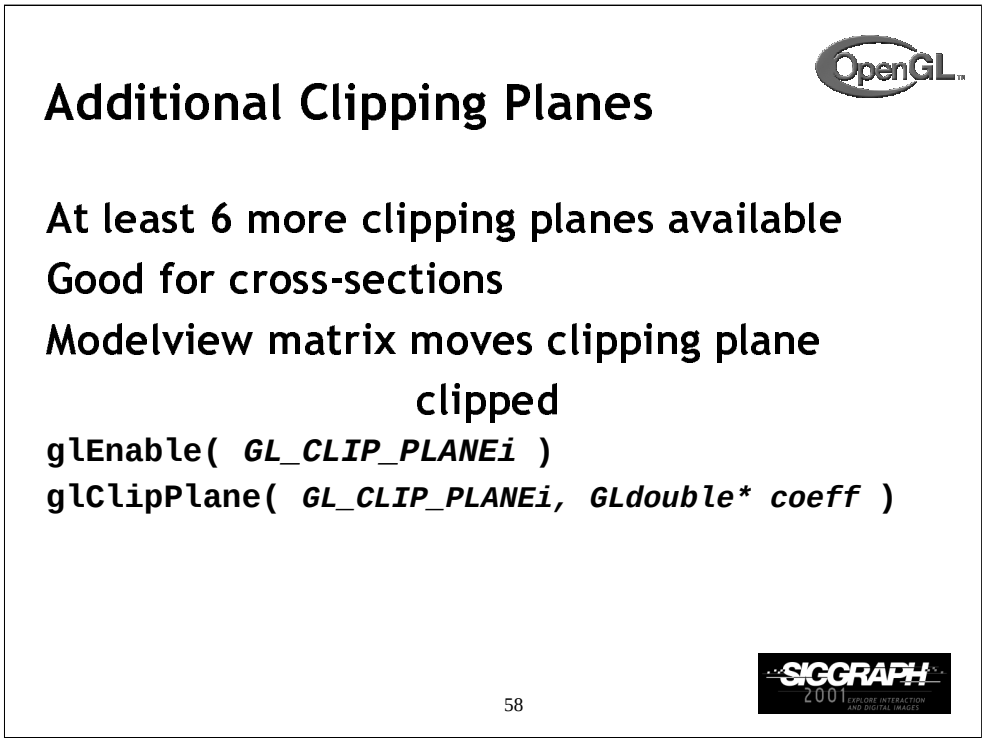
57

You'll adjust to reading a lot of code backwards!

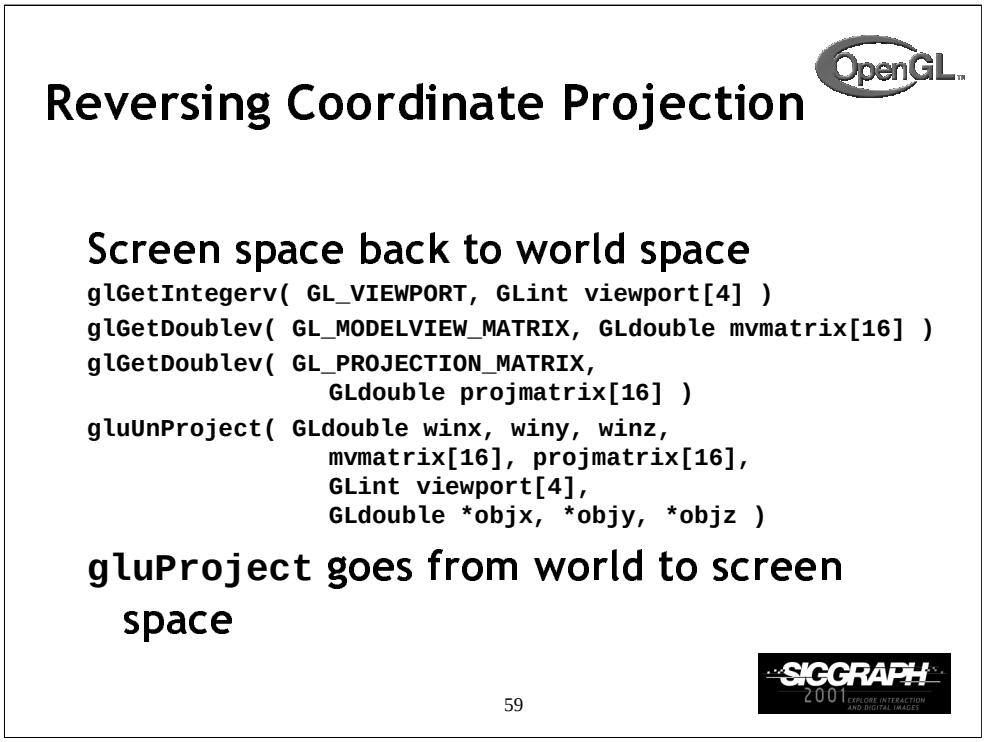
Typical sequence

```
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.

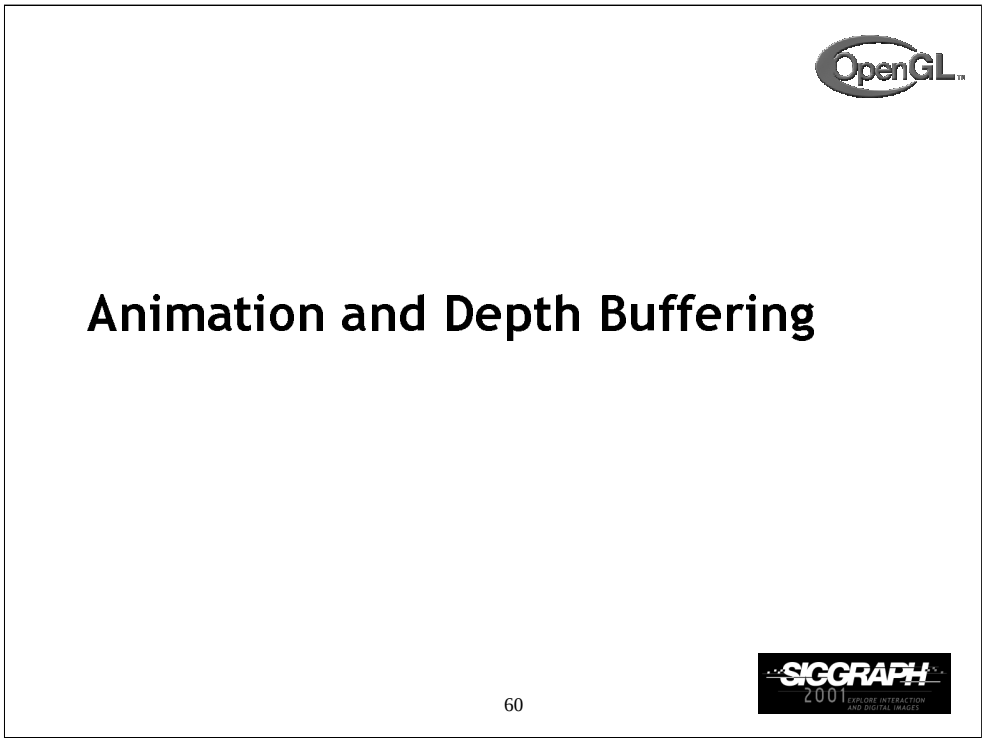


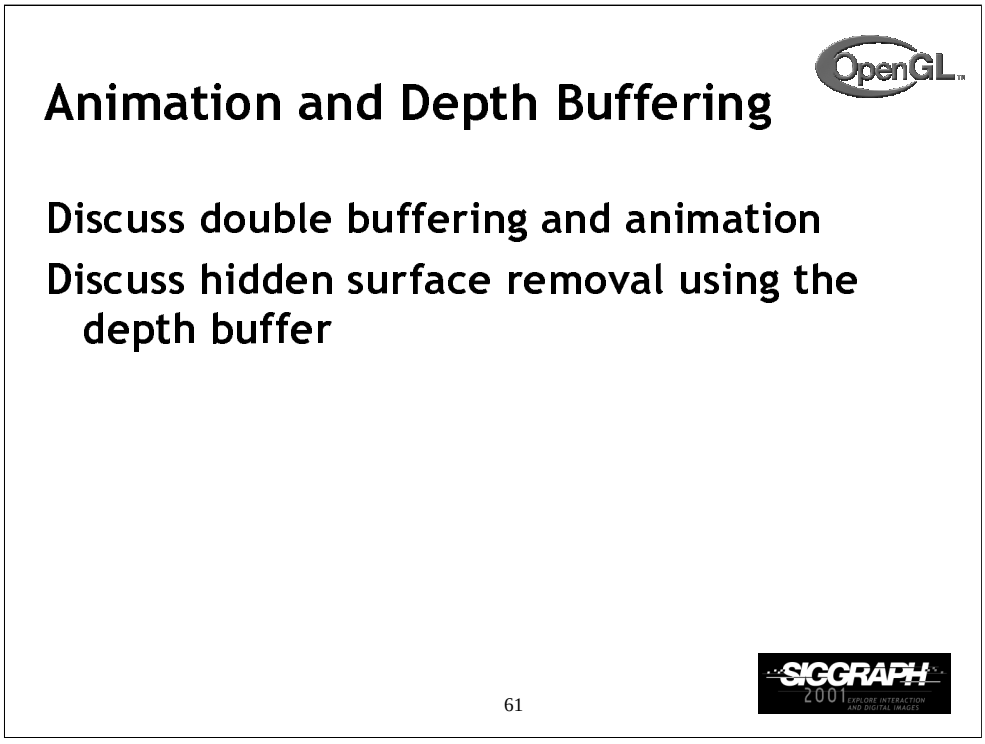
Use of additional clipping places may slow rendering as they are usually implemented in software.



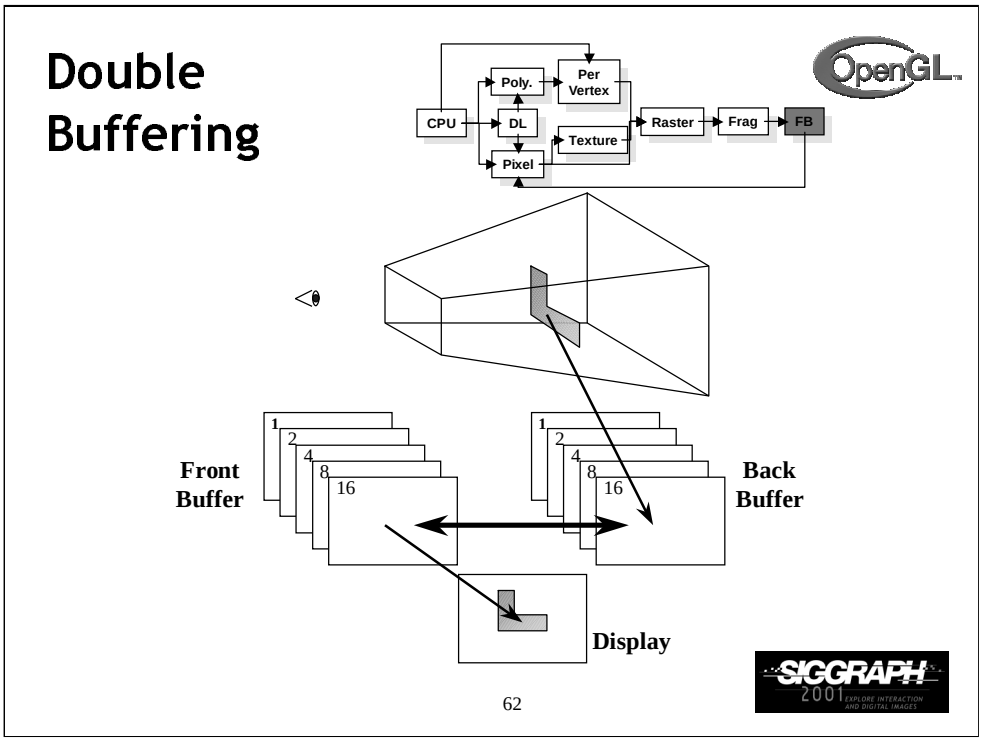
Generally, OpenGL projects 3D data onto a 2D screen. Sometimes, you need to use a 2D screen position (such as a mouse location) and figure out where in 3D it came from. If you use `gluUnProject()` with *winz = 0* and *winz = 1*, you can find the 3D point at the near and far clipping planes. Then you can draw a line between those points, and you know that some point on that line was projected to your screen position.

OpenGL Release 1.2 also introduced `gluUnProject4()`, which also returns the transformed world-space *w* coordinate.


OpenGL
Animation and Depth Buffering
60
SIGGRAPH
2001
EXPLORE INTERACTION
AND DIGITAL IMAGES



In this section we talk about adding the necessary steps for producing smooth interactive animations with OpenGL using double buffering. Additionally, we discuss hidden surface removal using 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.

Animation Using Double Buffering

Request a double buffered color buffer

```
glutInitDisplayMode( GLUT_RGB | GLUT_DOUBLE );
```

Clear color buffer

```
glClear( GL_COLOR_BUFFER_BIT );
```

Render scene

Request swap of front and back buffers

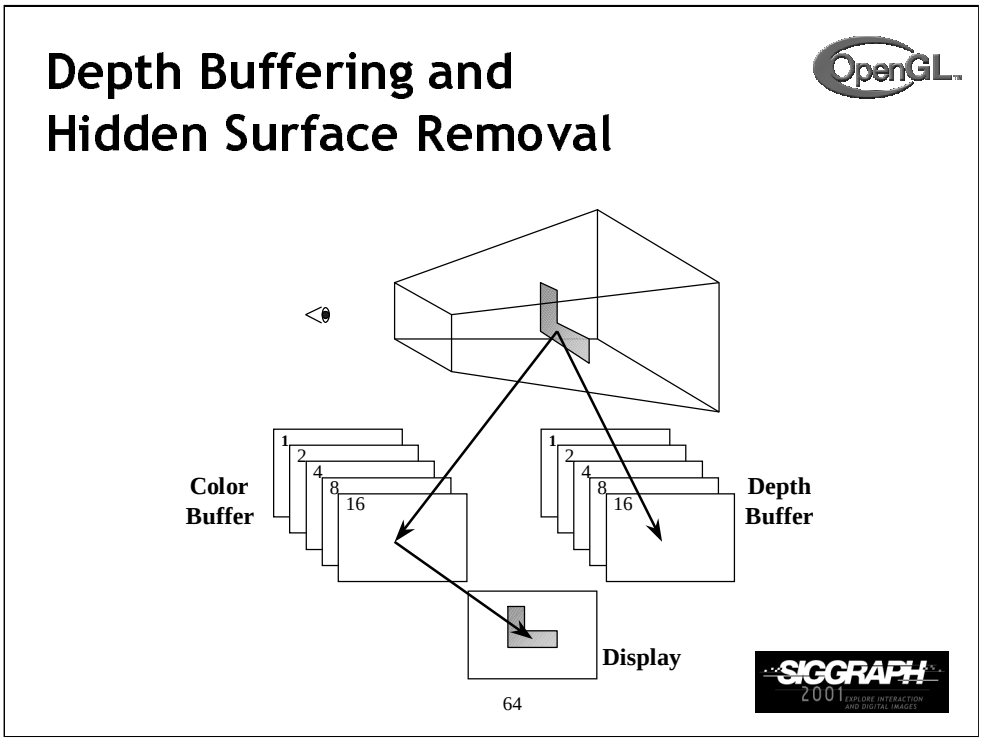
```
glutSwapBuffers();
```

Repeat steps 2 - 4 for animation

63

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.

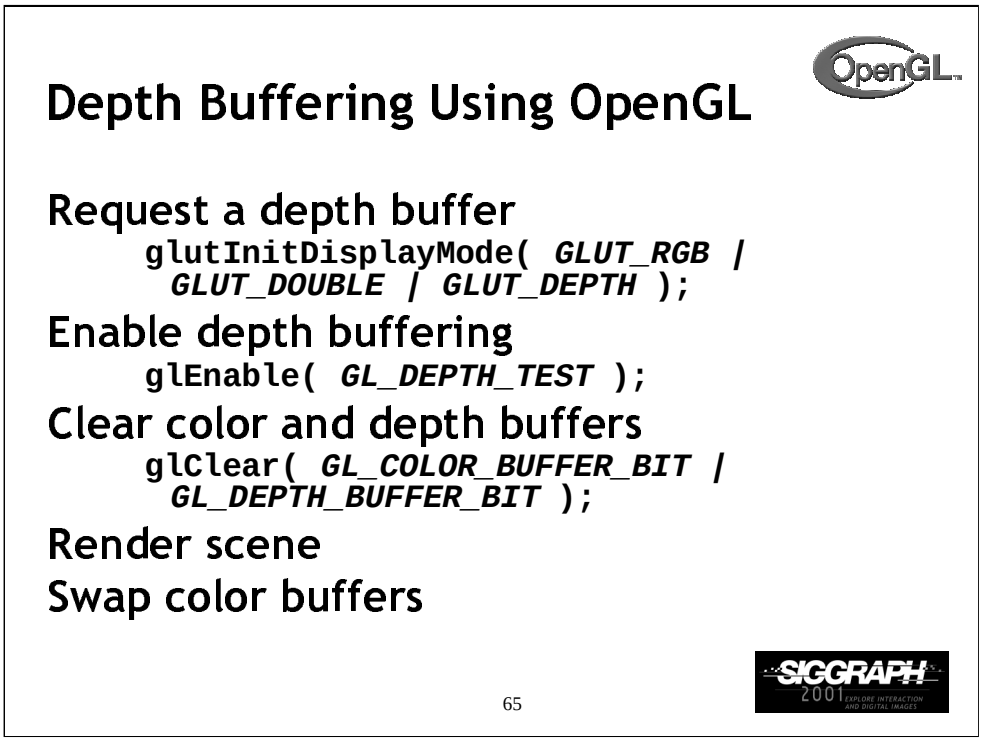


Depth buffering is a technique to determine which primitives in your scene are occluded by other primitives. As each pixel in a primitive is rasterized, its distance from the eyepoint (depth value), is compared with the values stored in the depth buffer. If the pixel's depth value is less than the stored 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, 1], with one being essentially infinitely far from the eyepoint. Generally, the depth buffer is cleared to one 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 )`.

An Updated Program Template

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

66

In `main()`,

1) GLUT initializes and creates a window named "Tetrahedron"
2) set OpenGL state which is enabled through the entire life of the program in `init()`
3) set GLUT's idle function, which is called when there are no user events to process.
4) enter the main event processing loop of the program.

An Updated Program Template (cont.)

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


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

67

In `init()` the basic OpenGL state to be used throughout the program is initialized. For this simple program, all we do is set the background (clear color) for the color buffer. In this case, we've decided that instead of the default black background, we want a blue background.

Additionally, our `glutIdleFunc()`, which we've set to the function `idle()` above, requests that GLUT re-render our scene again. The function `glutPostRedisplay()` requests that GLUT call our display function (this was set with `glutDisplayFunc()`) at the next convenient time. This method provides better event processing response from the application.

An Updated Program Template (cont.)

```
void drawScene( void )
{
     GLfloat vertices[] = { … };
     GLfloat colors[] = { … };
     glClear( GL_COLOR_BUFFER_BIT |
       GL_DEPTH_BUFFER_BIT );
     glBegin( GL_TRIANGLE_STRIP );
   /* calls to glColor*() and glVertex*() */
     glEnd();
     glutSwapBuffers();
}
```

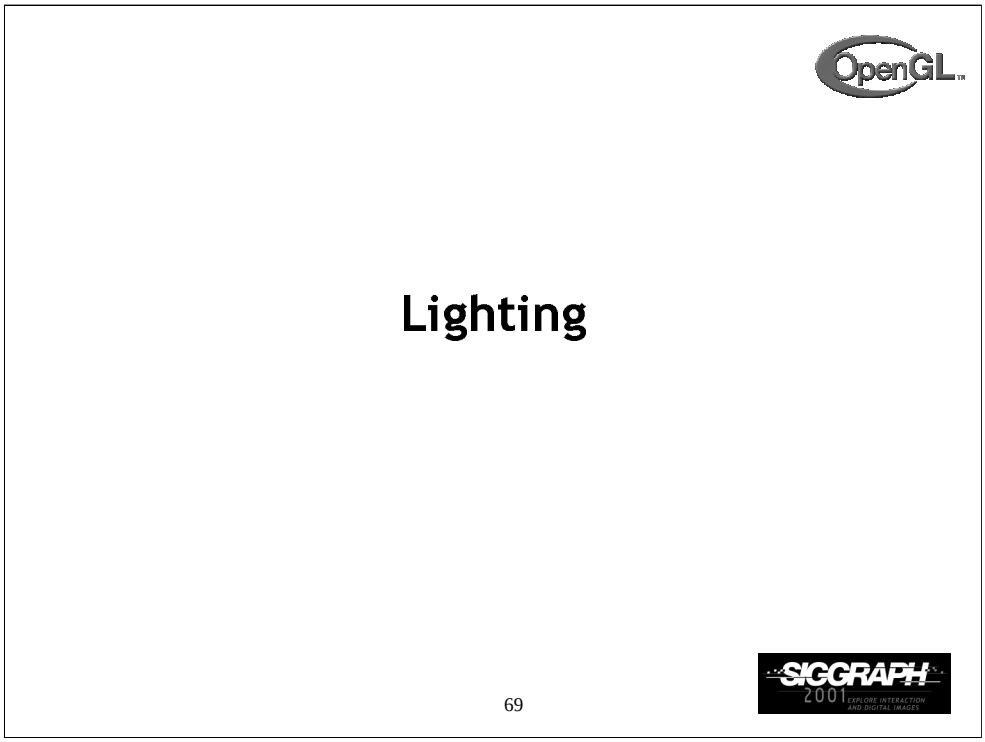
68

In `drawScene()`,

1) the color buffer is cleared to the background color

2) a triangle strip is rendered to create a tetrahedron (use your imagination for the details!)

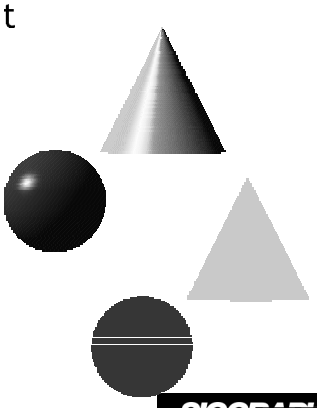
3) the front and back buffers are swapped.


OpenGL
Lighting
69
SIGGRAPH
2001
EXPLORE INTERACTION
AND DIGITAL IMAGES

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

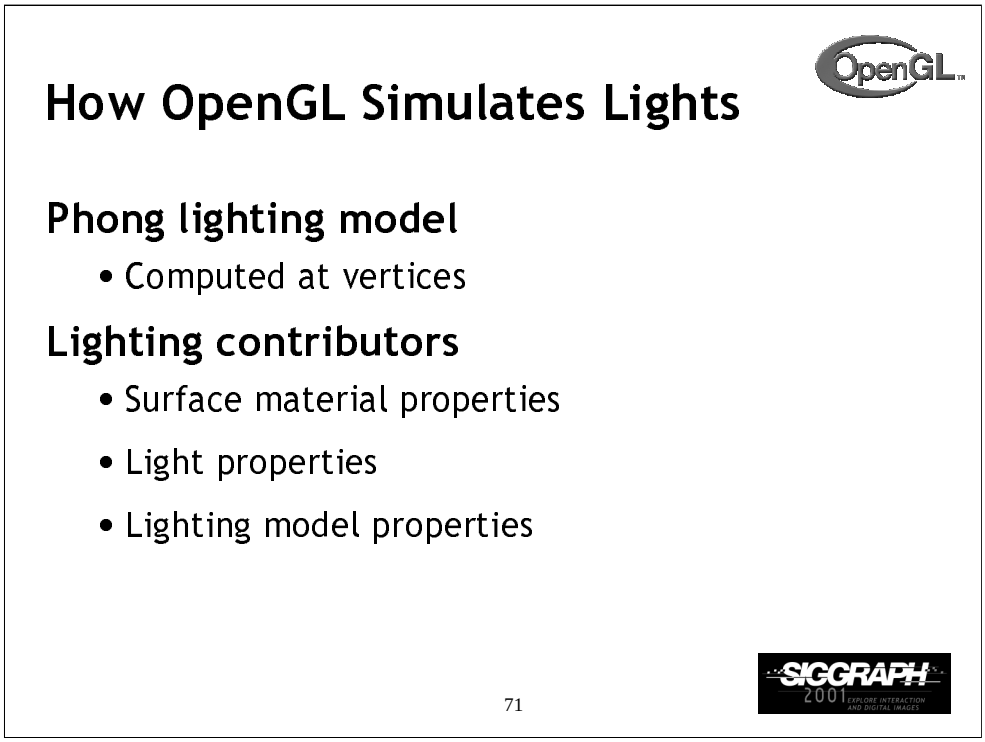


70

Lighting is an important technique in computer graphics. Without lighting, objects tend to look like they're 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.

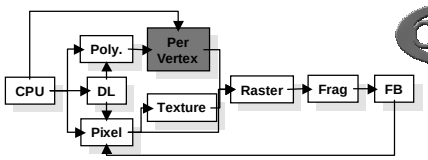


OpenGL lighting is based on the Phong lighting model. At each vertex in the primitive, a color is computed using that primitives material properties along with the light settings.

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)

Surface Normals



Normals define how a surface reflects light

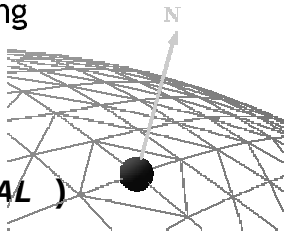
`glNormal3f( x, y, z )`

- Current normal is used to compute vertex's color
- Use *unit* normals for proper lighting
  - scaling affects a normal's length

`glEnable( GL_NORMALIZE )`

or

`glEnable( GL_RESCALE_NORMAL )`



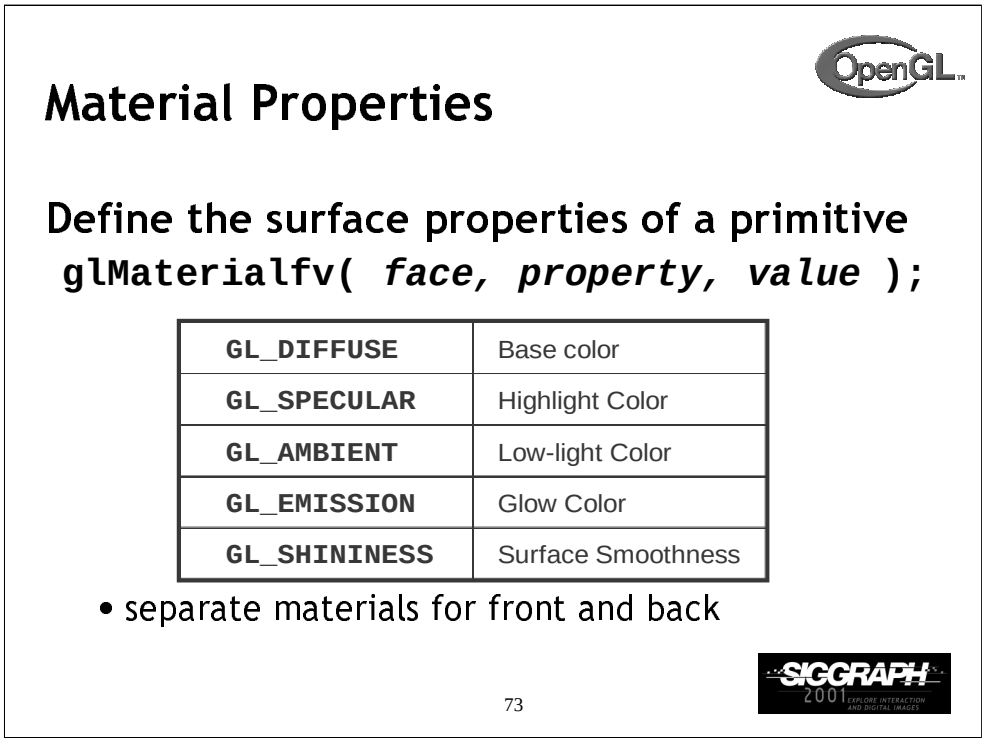
72

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.

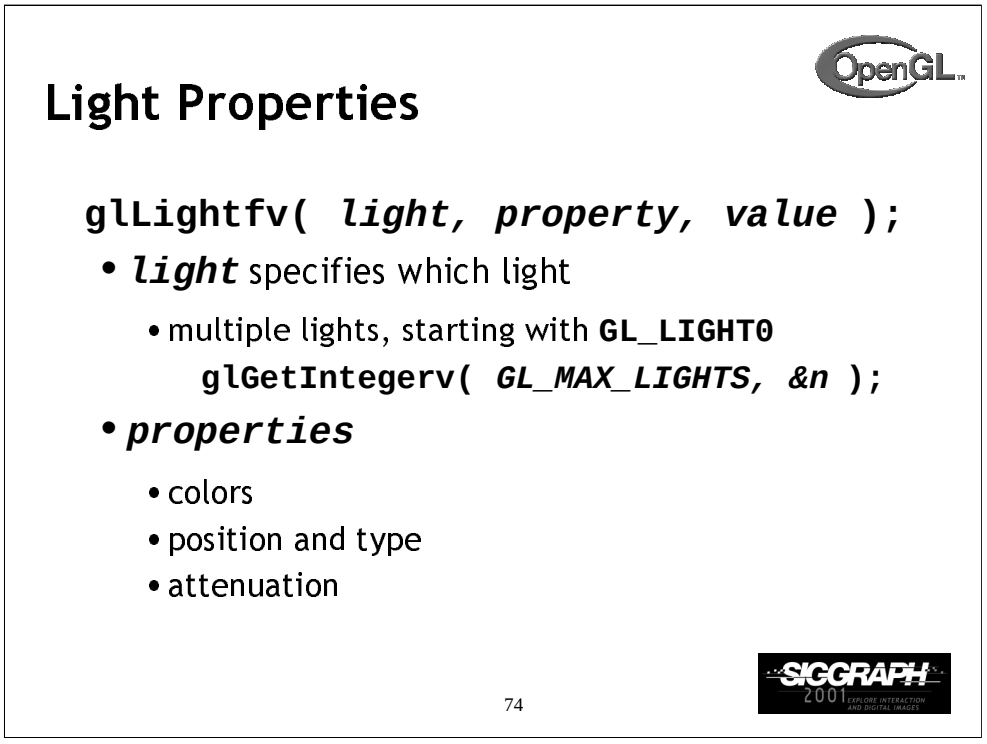


Material properties describe the color and surface properties of a material (dull, shiny, etc.). OpenGL supports material properties for both the front and back of objects, as described by their vertex winding.

The OpenGL material properties are:

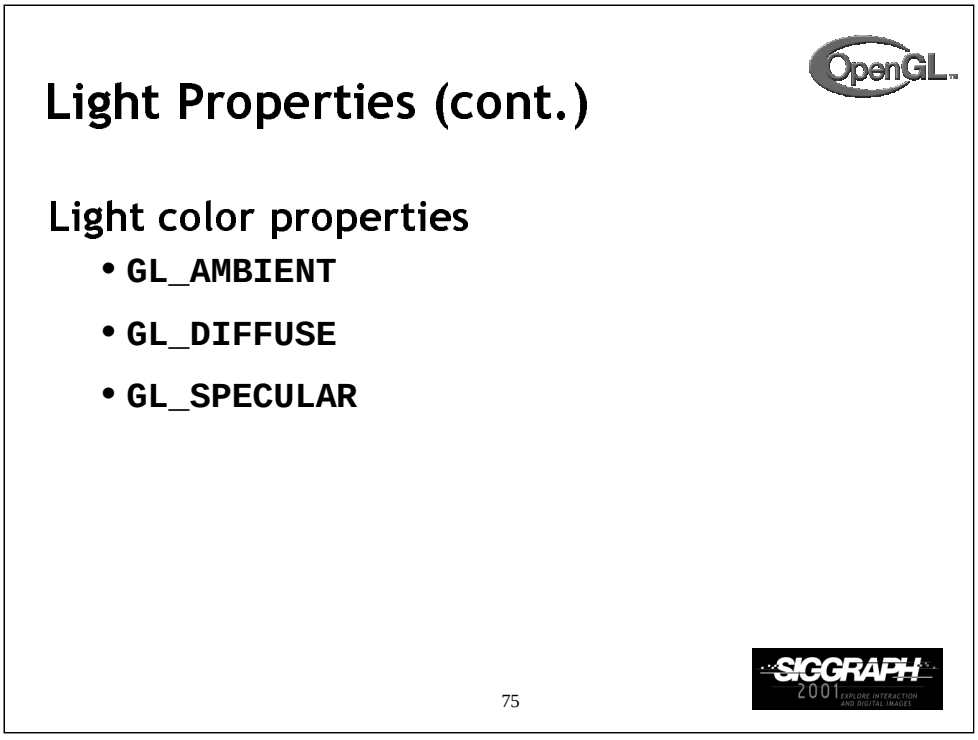
- `GL_DIFFUSE` - base color of object
- `GL_SPECULAR` - color of highlights on object
- `GL_AMBIENT` - color of object when not directly illuminated
- `GL_EMISSION` - color emitted from the object (think of a firefly)
- `GL_SHININESS` - concentration of highlights on objects. Values range from 0 (very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

Types of Lights

OpenGL supports two types of Lights

- Local (Point) light sources
- Infinite (Directional) light sources

Type of light controlled by w coordinate

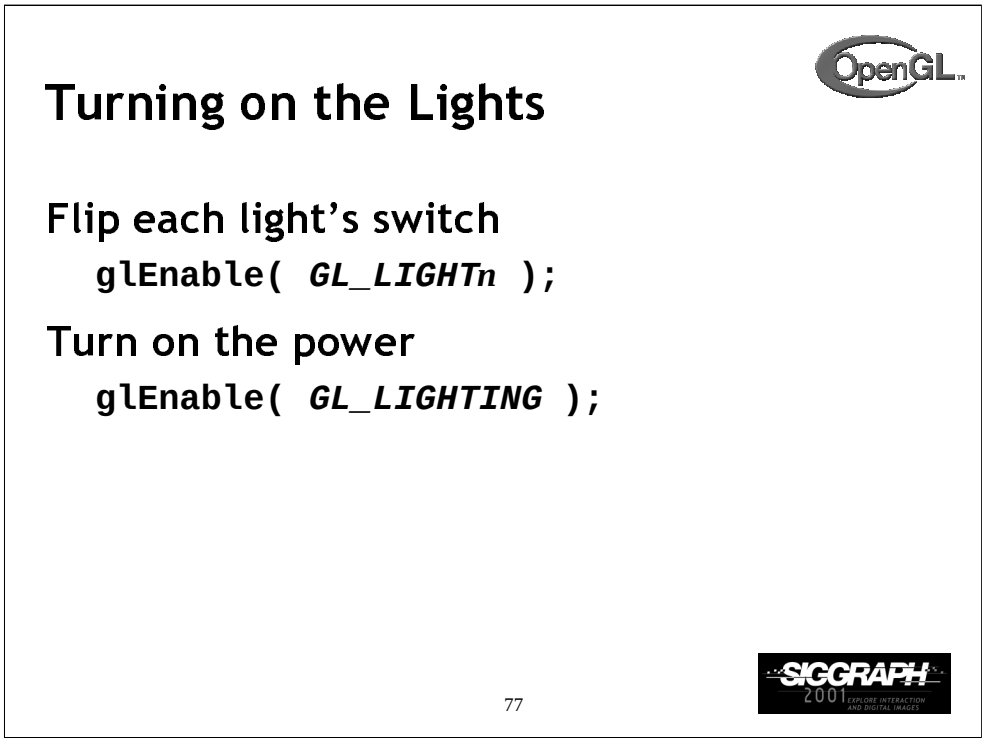
$w = 0$ ***Infinite Light directed along*** $(x \quad y \quad z)$

$w \neq 0$ ***Local Light positioned at*** $\left(x/w \quad y/w \quad z/w\right)$

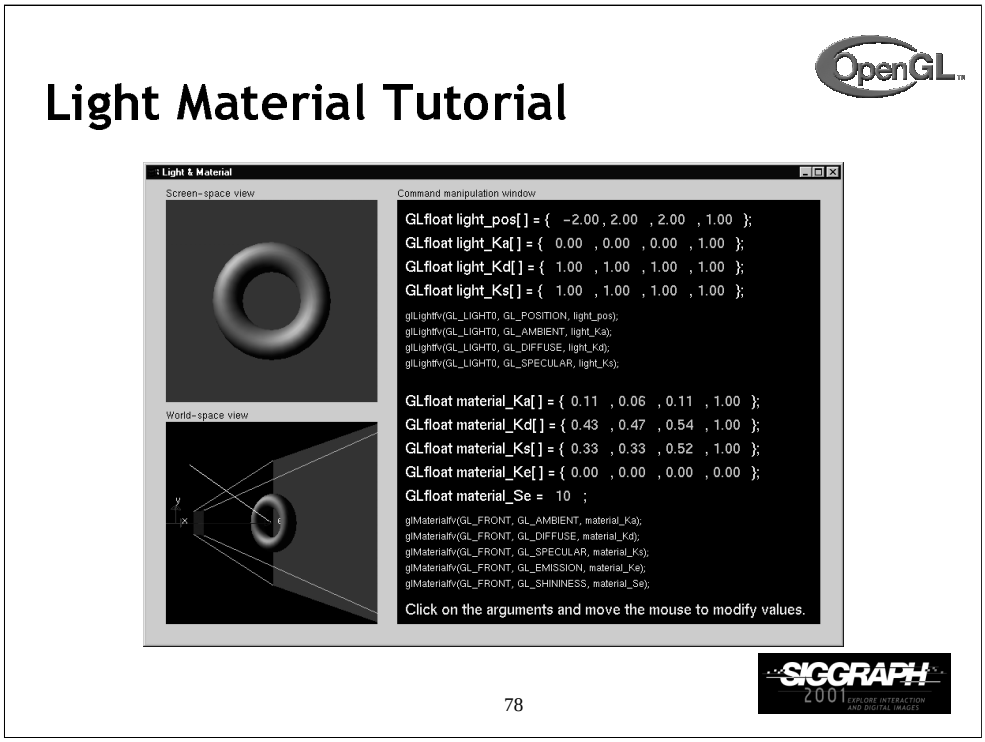
76

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

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.

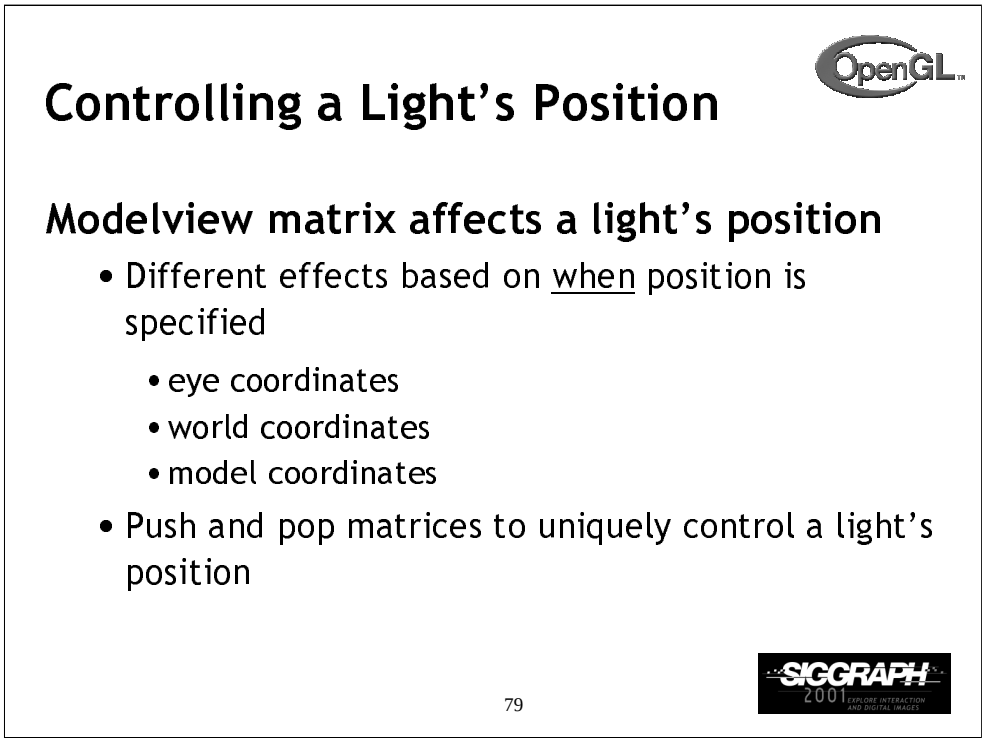


Each OpenGL light is controllable separately, using `glEnable()` and the respective light constant `GL_LIGHT`*n*. Additionally, global control over whether lighting will be used to compute primitive colors is controlled by passing `GL_LIGHTING` to `glEnable()`. This provides a handy way to enable and disable lighting without turning on or off all of the separate components.



In this tutorial, concentrate on noticing the effects 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.



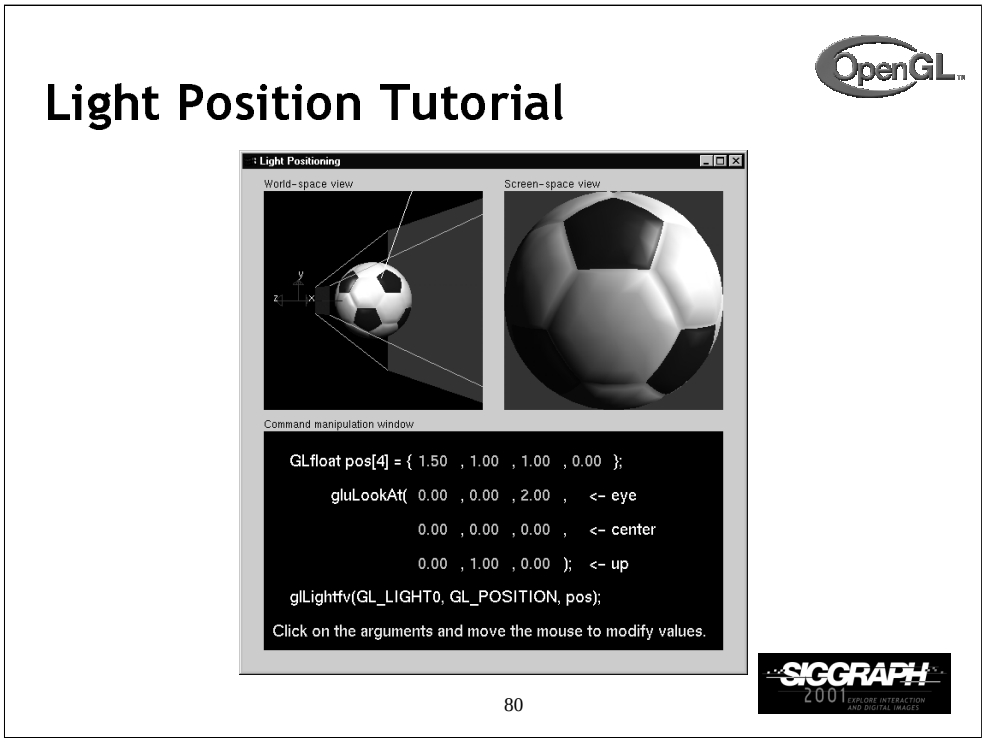
As mentioned previously, a light's position is transformed by the current ModelView matrix when it is specified. As such, depending on when you specify the light's position, and what's in the ModelView matrix, you can obtain different lighting effects.

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

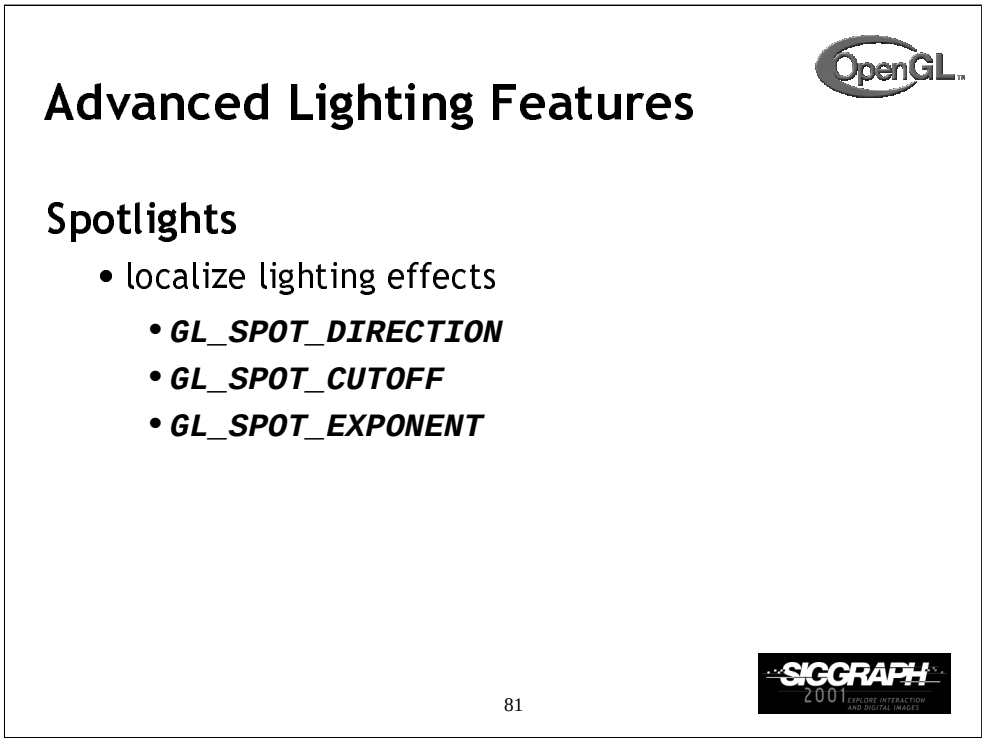
1) *Eye coordinates* - which is represented by an identity matrix in the ModelView. In this case, when the light's position/direction is specified, it remains fixed to the imaging plane. As such, regardless of how the objects are manipulated, the highlights remain in the same location relative to the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



A local light can also be converted into a spotlight. By setting the `GL_SPOT_DIRECTION`, `GL_SPOT_CUTOFF`, and `GL_SPOT_EXPONENT`, the local light will shine in a direction and its light will be limited to a cone centered around that direction vector.

Advanced Lighting Features

Light attenuation

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

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

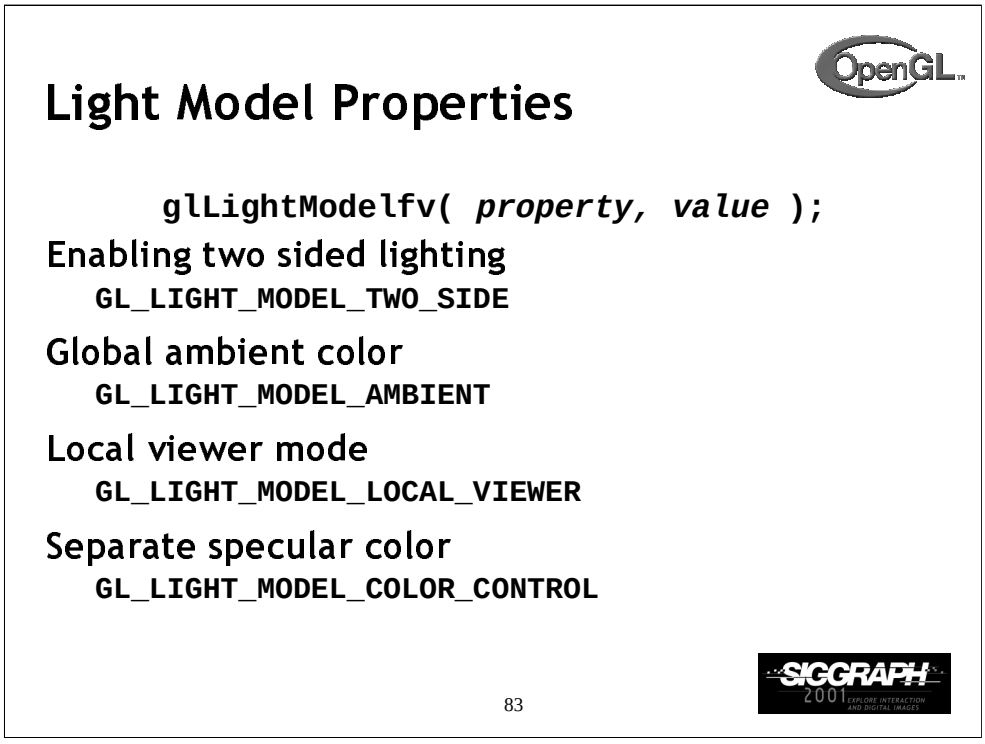
82

Each OpenGL light source supports attenuation, which describes how light diminishes with distance. The OpenGL model supports quadratic attenuation, and utilizes the following attenuation factor, f_i, where d is the distance from the eyepoint to the vertex being lit:

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

where:

- k_c is the `GL_CONSTANT_ATTENUATION` term
- k_l is the `GL_LINEAR_ATTENUATION` term
- k_q is the `GL_QUADRATIC_ATTENUATION` term



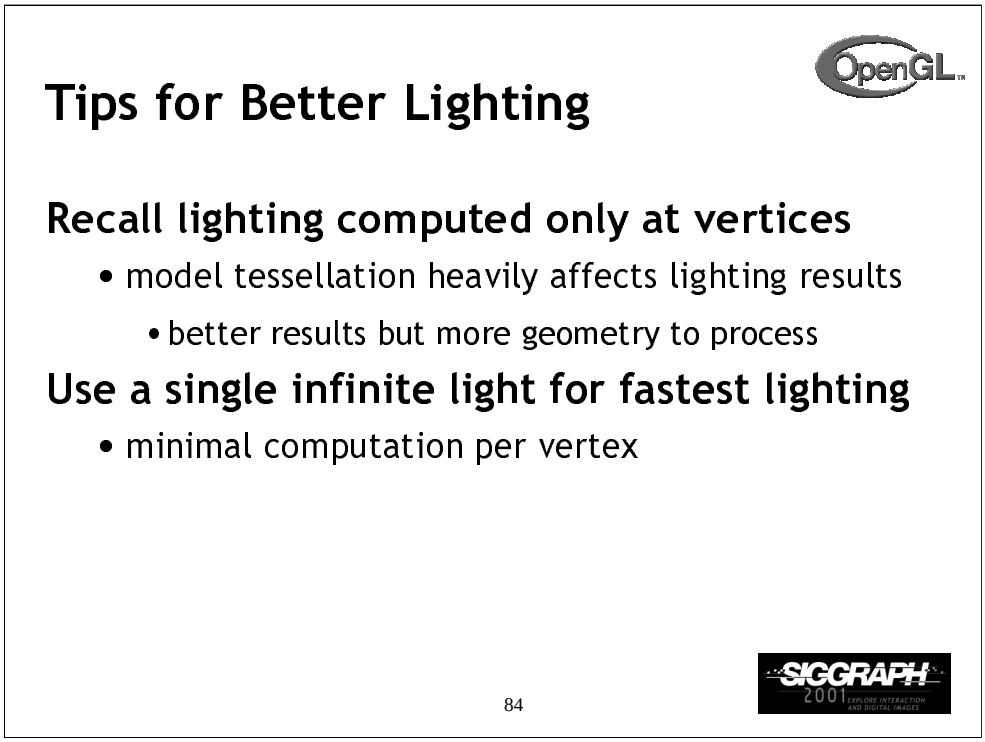
Properties which aren't directly connected with materials or lights are grouped into *light model properties.* With OpenGL 1.2, there are four properties associated with the lighting model:

1) *Two-sided lighting* uses the front and back material properties for illuminating a primitive.

2) *Global ambient color* initializes the global ambient contribution of the lighting equation.

3) *Local viewer mode* disables an optimization which provides faster lighting computations. With local viewer mode on, you get better light results at a slight performance penalty.

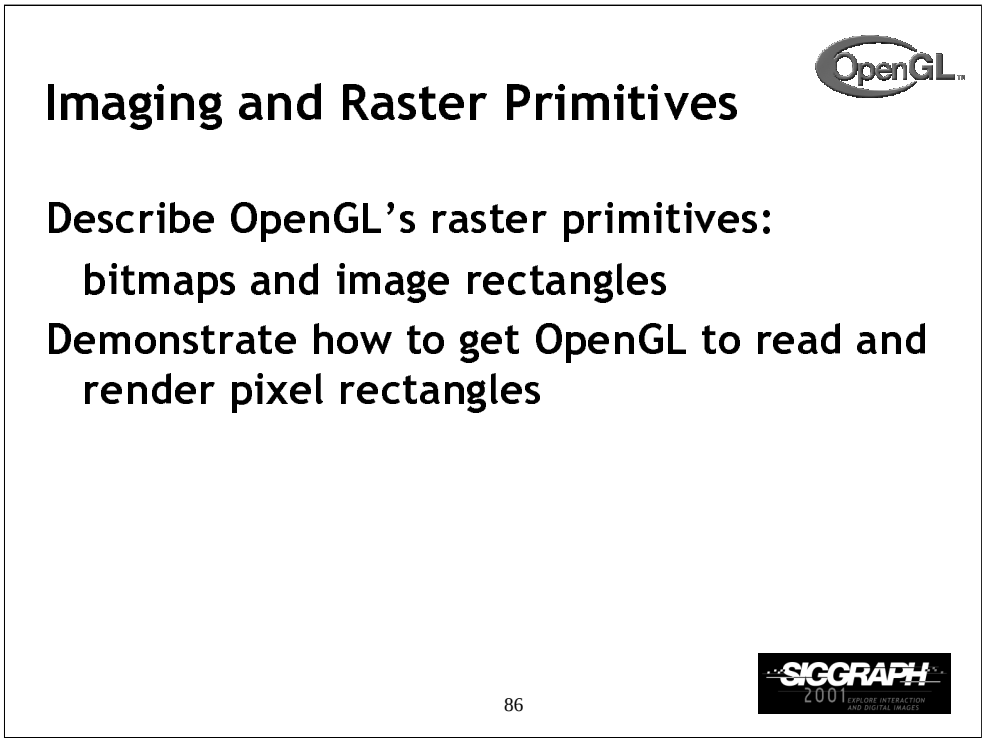
4) *Separate specular color* is a mode for maintaining better specular highlights in certain texture mapped conditions. This is a new feature for OpenGL 1.2.



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.

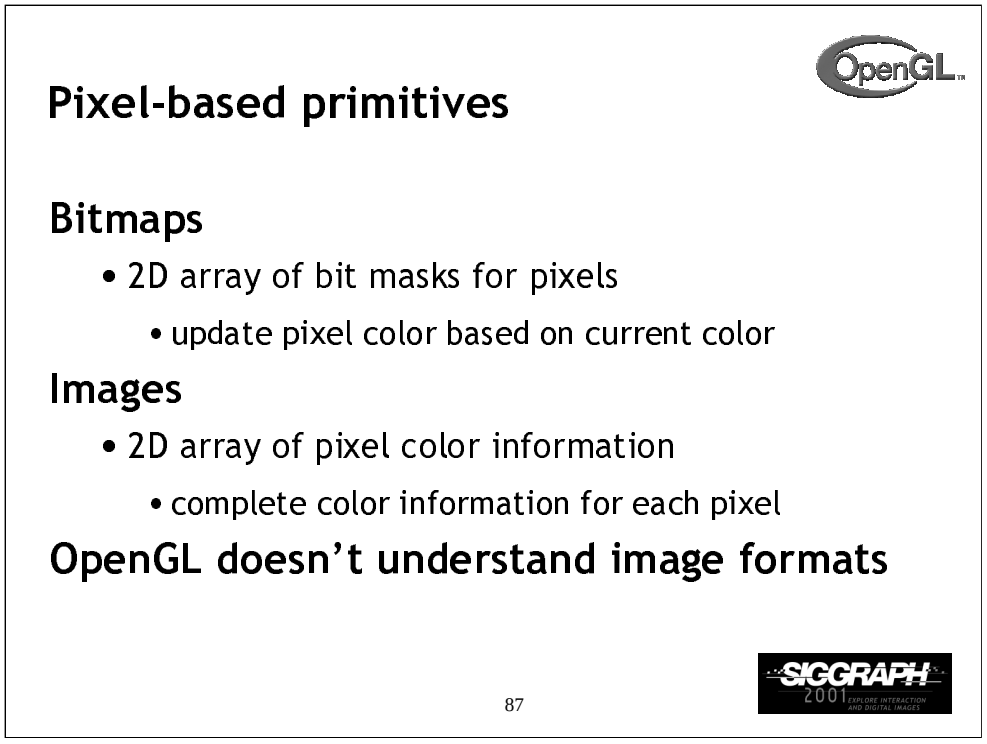
To achieve maximum performance for lighting in your applications, use a single infinite light source. This minimizes the amount of work that OpenGL has to do to light every vertex.

Imaging and Raster Primitives



OpenGL is not only a complete interface for 3D rendering, it's also a very capable image processing engine. In this section we discuss some of the basic functions of OpenGL for rendering color-pixel rectangles and single-bit bitmaps, as well as how to read color information from the framebuffer.

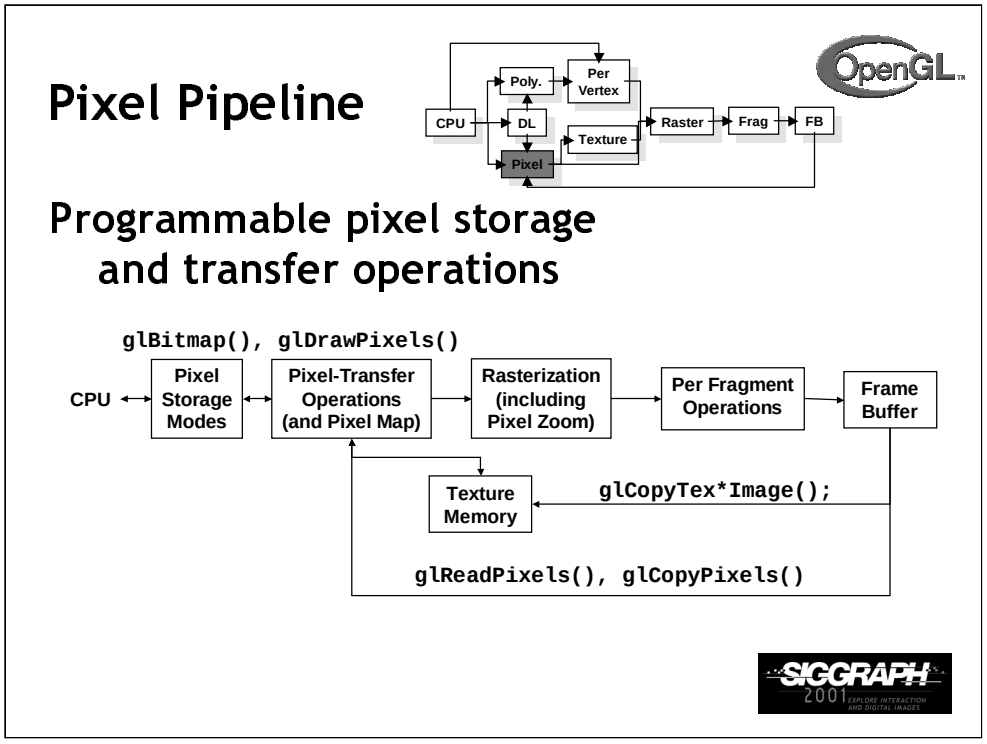
OpenGL doesn't render images, per se, since images usually are stored in some file with an image format associated with it (for example, JPEG images). OpenGL only knows how to render rectangles of pixels, not decode image files.



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, doesn't 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.



Just as there's 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 from the image is processed by the fragment operations discussed in the last section, and finally 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, for 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 )`

- raster position transformed like geometry
- discarded if raster position is outside of viewport
  - may need to fine tune viewport for desired results

Raster Position

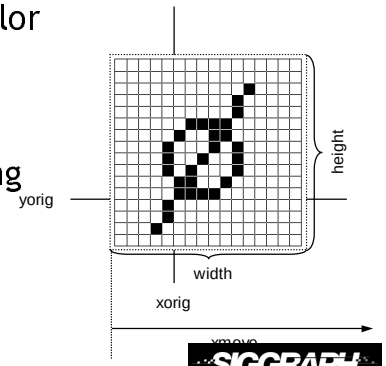
89

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 Bitmaps

glBitmap(*width, height, xorig, yorig, xmove, ymove, bitmap*)

- render bitmap in current color at
- advance raster position by after rendering

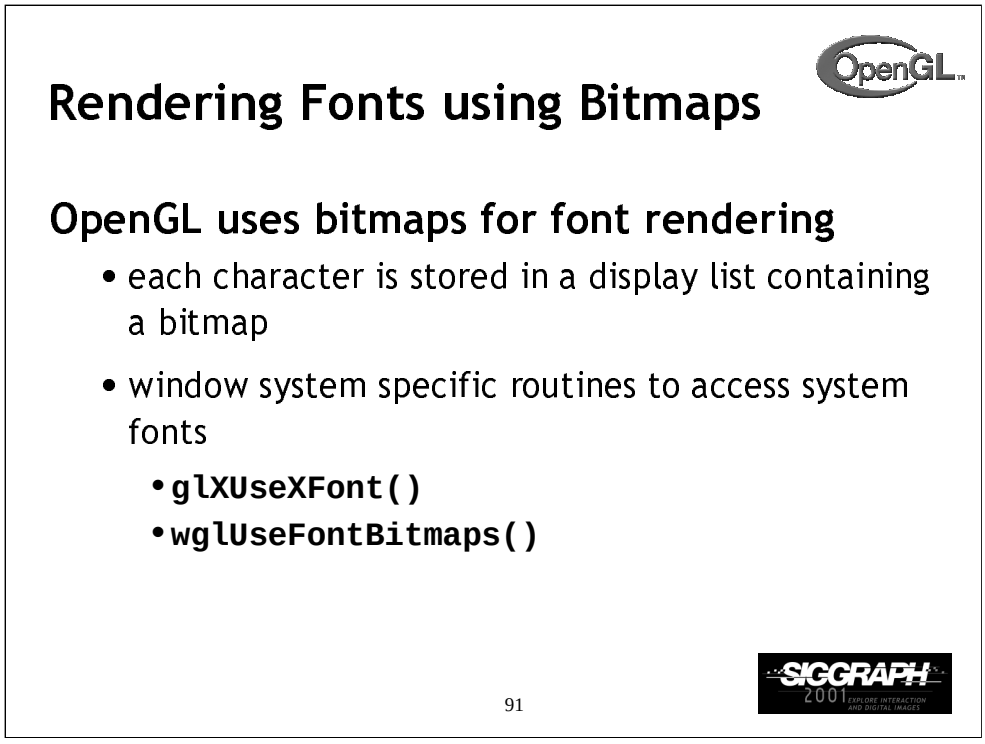


90

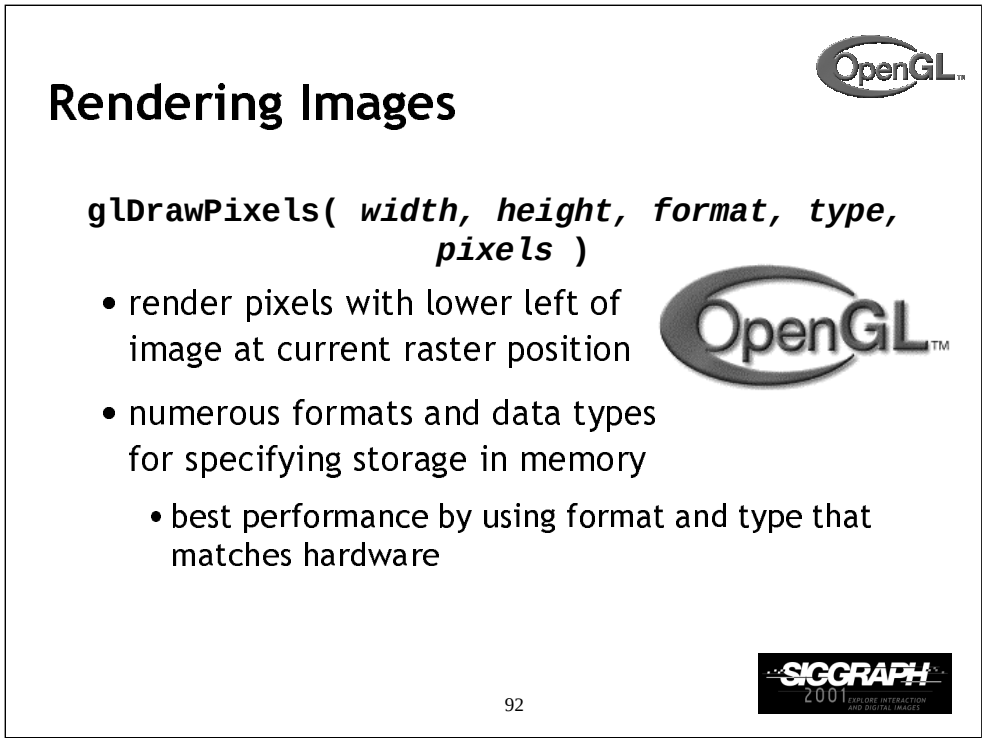
Bitmaps are used as a mask to determine which pixels to update. A bitmap is specified as a packed array of bits in a byte array. For each value of one in the bitmap, a fragment is generated in the currently set color and processed by the fragment operations.

Bitmaps can have their own origin, which provides a relative position to the current raster position. Additionally, after the bitmap is rendered, the raster position is automatically updated by the offset provided in (*xmove*, *ymove)*.

Bitmaps are particularly useful for rendering bitmapped text, which we'll discuss in a moment.



OpenGL uses bitmaps to do font rendering. The window system specific routines process system native font files, and create bitmaps for the different glyphs in the font. Each character is stored in a display list that is part of a set that is created when the font is processed.

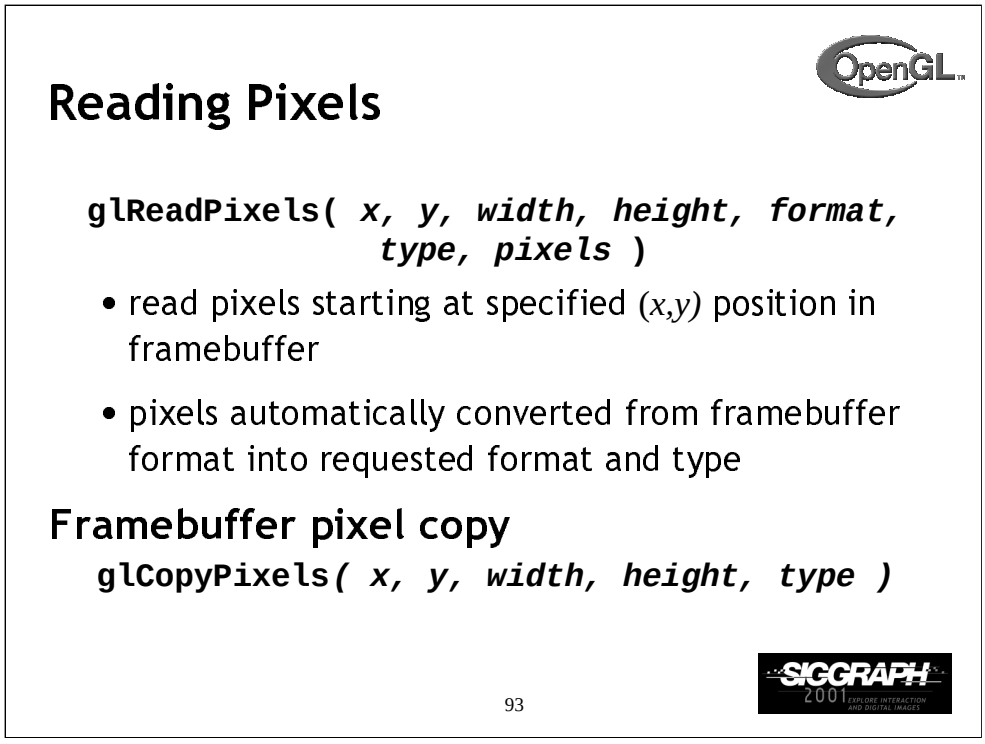


Rendering images is done with the `glDrawPixels()`command. A block of pixels from host CPU memory is passed into OpenGL with a format and data type specified. For each pixel in the image, a fragment is generated using the color retrieved from the image, and further processed.

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`.



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.

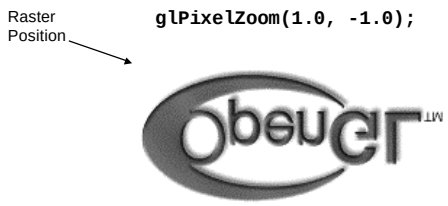
Pixels read from the framebuffer are processed by the pixel storage and transfer modes, as well as converting them into the format and type requested, and placing them in host memory.

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.

Pixel Zoom

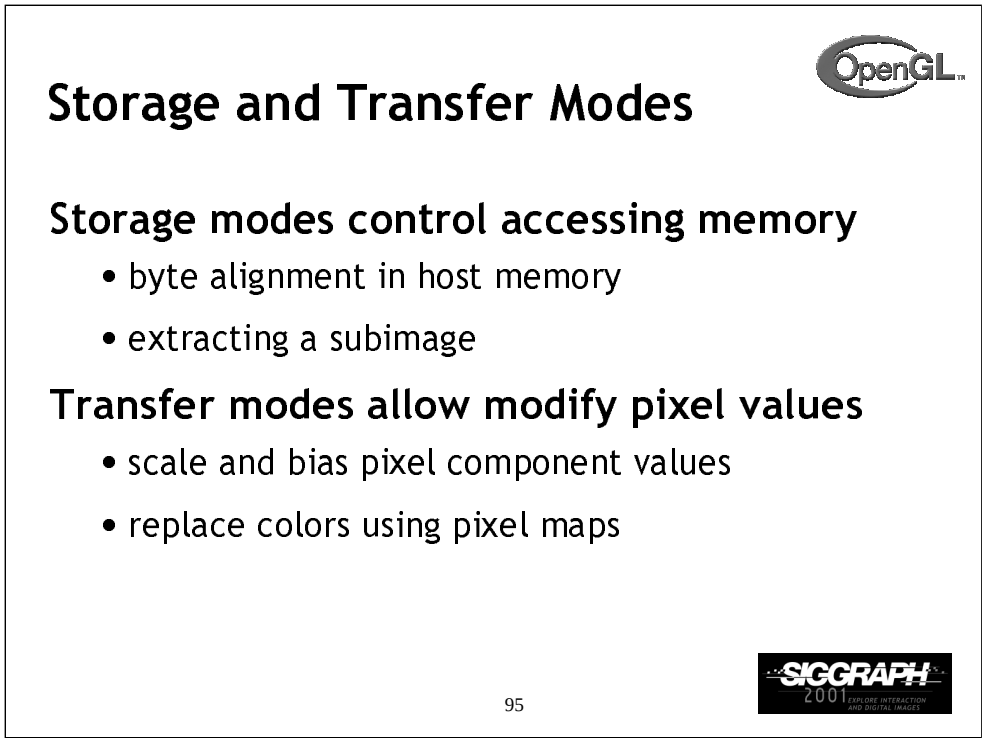
glPixelZoom(*x, y*)

- expand, shrink or reflect pixels around current raster position
- fractional zoom supported



94

OpenGL can also scale pixels as they are being rendered. `glPixelZoom()` will scale or shrink pixels as well as reflect them around the current raster position.



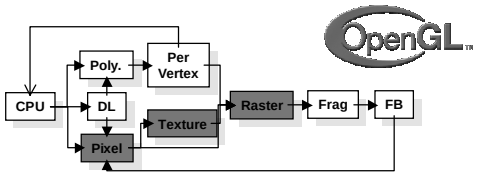
When pixels are read from or written to host memory, pixels can be modified by *storage and transfer* modes.

Storage modes control how host memory is accessed and written to, including byte swapping and addressing, and modifying how memory is accessed to read only a small subset of the original image.

Transfer modes allow pixel modifications as they are processed. This includes scaling and biasing the color component values, as well as replacing color values using color lookup tables.


OpenGL
Texture Mapping
96
SIGGRAPH
2001
EXPLORE INTERACTION AND DIGITAL IMAGES

Texture Mapping



Apply a 1D, 2D, or 3D image to geometric primitives

Uses of Texturing

- simulating materials
- reducing geometric complexity
- image warping
- reflections

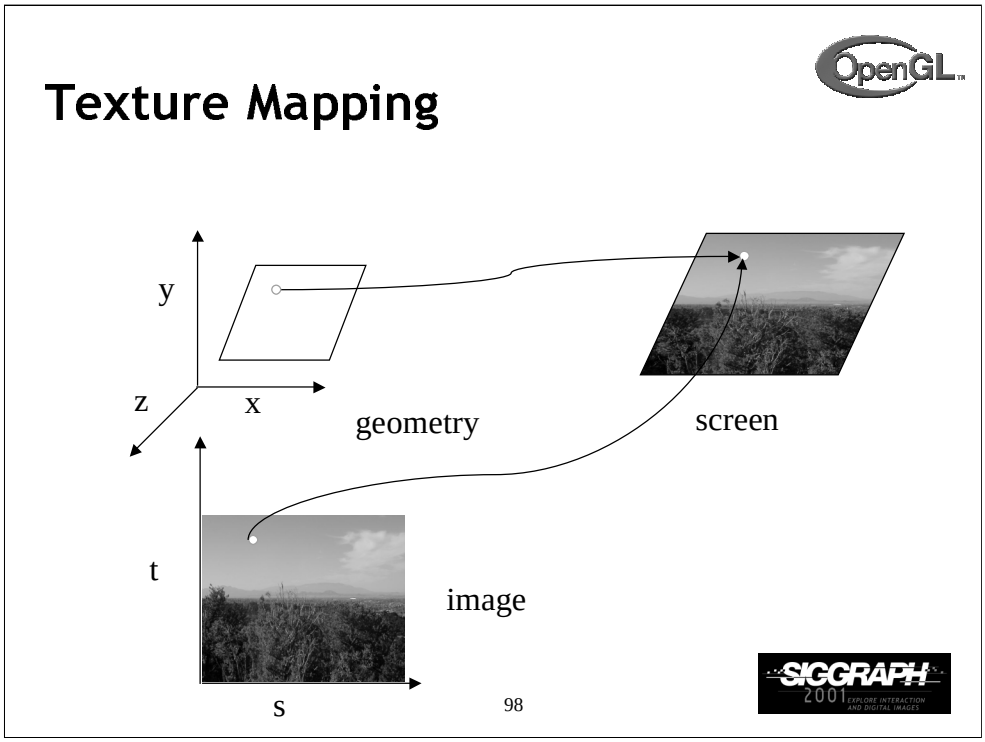
97

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

In this section, we'll discuss *texture* (sometimes also called *image*) mapping. Texture mapping augments the colors specified for a geometric primitive with the colors stored in an image. An image can be a 1D, 2D, or 3D set of colors called *texels*. 2D textures will be used throughout the section for demonstrations, however, the processes described are identical for 1D and 3D textures.

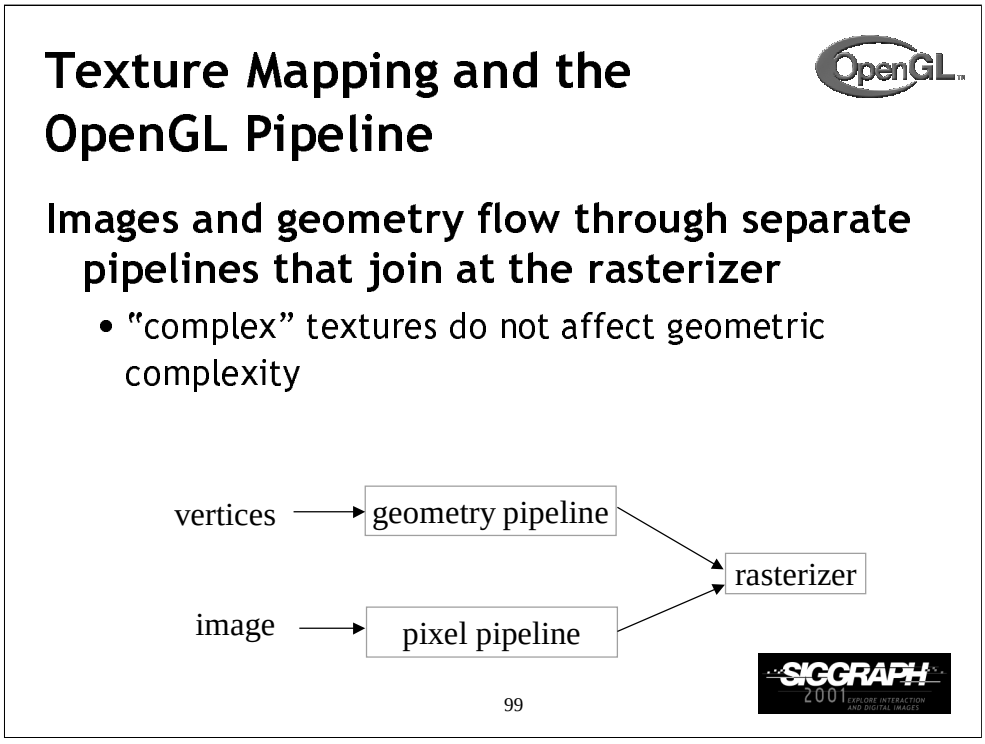
Some of the many uses of texture mapping include:

- simulating materials like wood, bricks or granite
- reducing the complexity (number of polygons) of a geometric object
- image processing techniques like image warping and rectification, rotation and scaling
- simulating reflective surfaces like mirrors or polished floors



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.

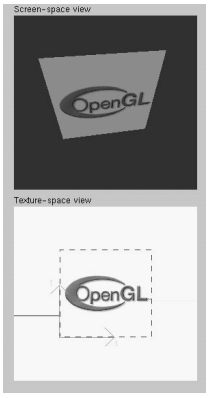
A texture map for a two-dimensional geometric object in (*x*, *y*, *z*) world coordinates maps a point in (*s*, *t*) space to a corresponding point on the screen.



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 x 256 image that has been mapped to a rectangular polygon which is viewed in perspective

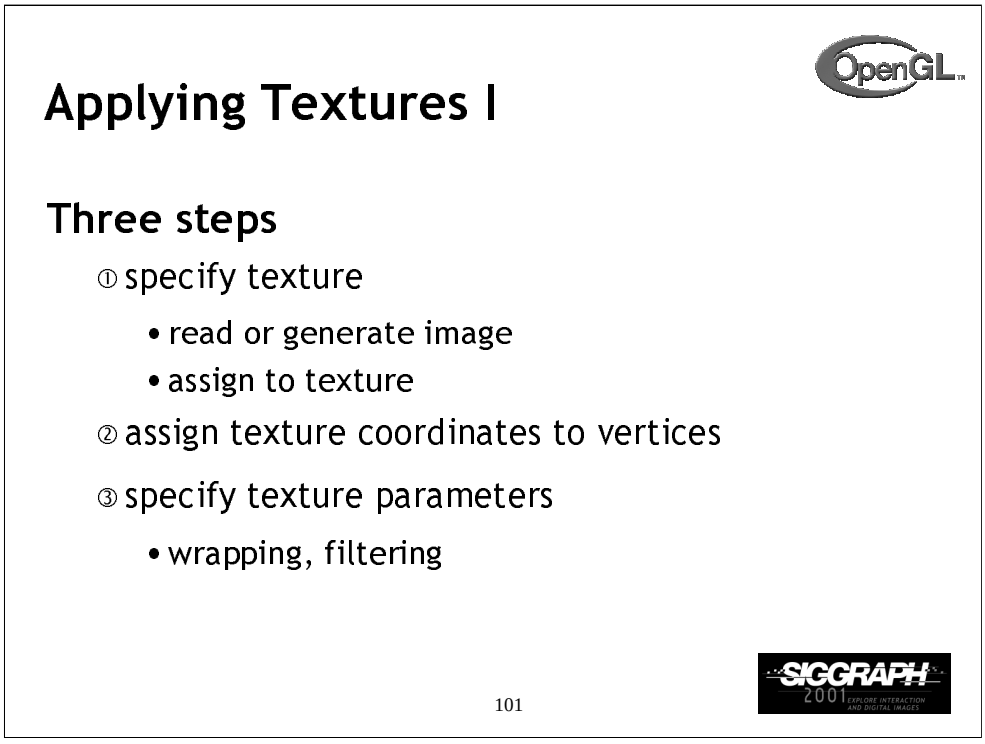


100

This example is from the 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.



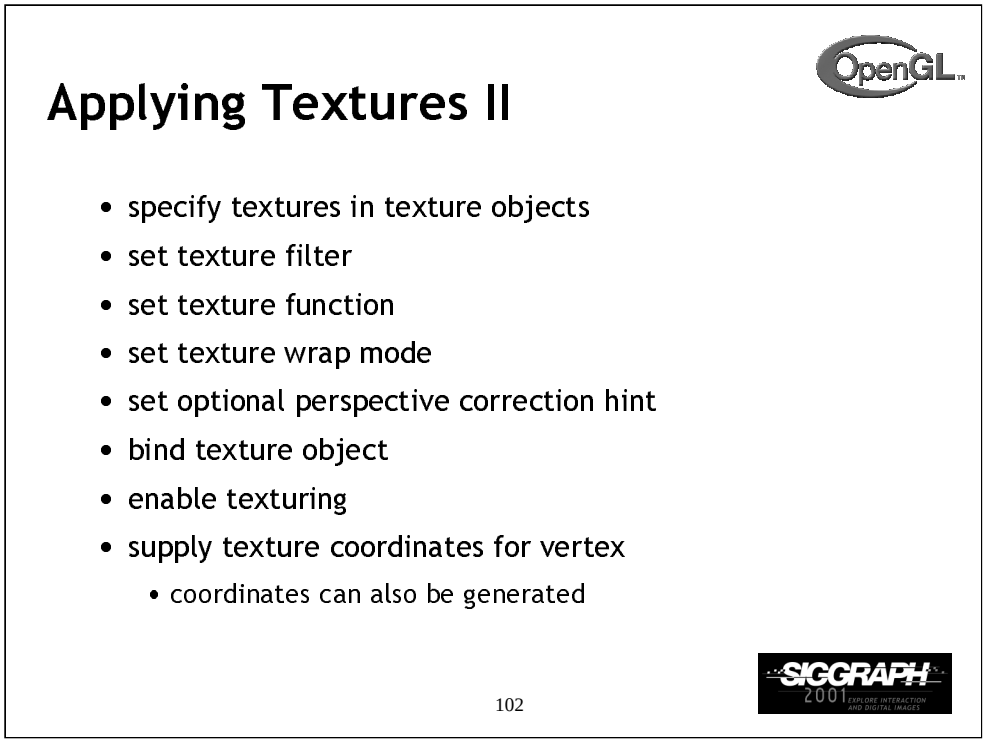
In the simplest approach, we must perform these three steps.

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 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:

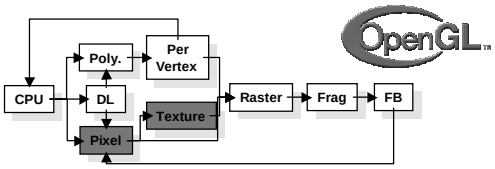
`GL_TEXTURE_1D` - one-dimensional texturing

`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.

Specify Texture Image



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

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

- dimensions of image must be powers of 2

Texel colors are processed by pixel pipeline

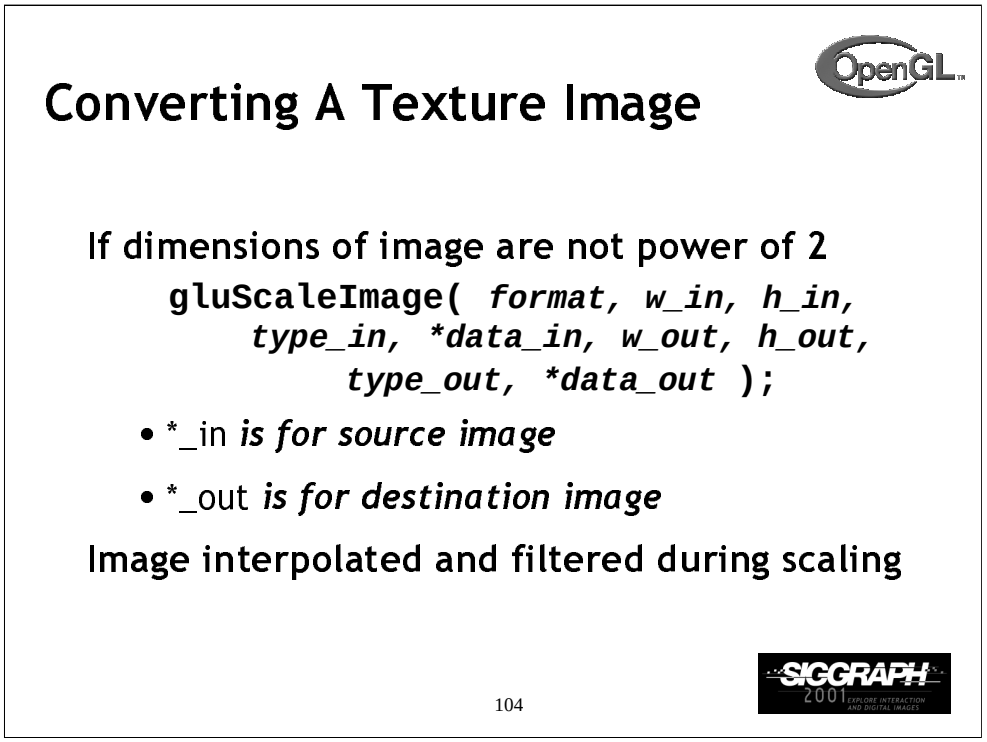
- pixel scales, biases and lookups can be done

103

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're using a texturing technique called mipmapping, which we'll discuss in a few slides.



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 (which computes the weighted average of adjacent pixels to generate the new pixel values) 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. Converting types using `gluScaleImage()` can help your application save memory.

Specifying a Texture: Other Methods

Use frame buffer as source of texture image

- uses current buffer as source image

glCopyTexImage2D(...)

glCopyTexImage1D(...)

Modify part of a defined texture

glTexSubImage2D(...)

glTexSubImage1D(...)

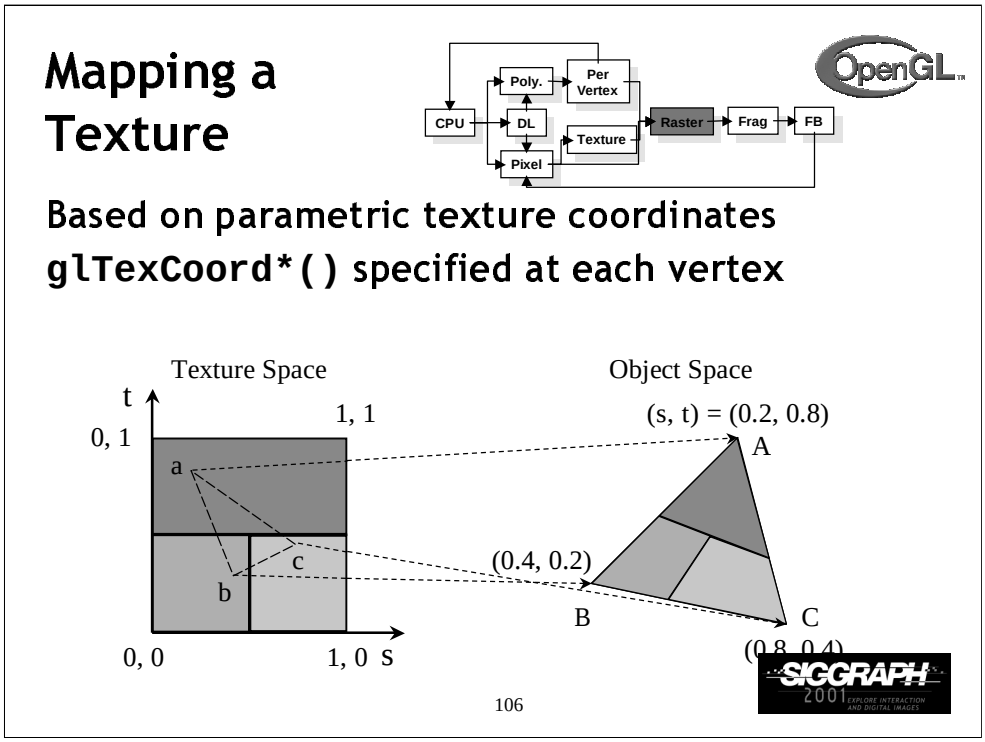
Do both with **glCopyTexSubImage2D(...)**, etc.

105

`glCopyTexImage*()` allows textures to be defined by rendering into any of OpenGL's buffers. The source buffer is selected using the `glReadBuffer()` command.

Using `glTexSubImage*()` to replace all or part of an existing texture often outperforms using `glTexImage*()` to allocate and define a new one. This can be useful for creating a "texture movie" (sequence of textures which changes appearance on an object's surface).

There are some advanced techniques using `glTexSubImage*()` which include loading an image which doesn't meet the power of two requirement. Additionally, several small images can be "packed" into one larger image (which was originally created with `glTexImage*()`), and loaded individually with `glTexSubImage*()`. Both of these techniques require the manipulation of the texture transform matrix, which is outside the scope of this course.



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

- generate texture coordinates based upon distance from plane

$$Ax + By + Cz + D = 0$$

generation modes

- **`GL_OBJECT_LINEAR`**
- **`GL_EYE_LINEAR`**
- **`GL_SPHERE_MAP`**

107

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.

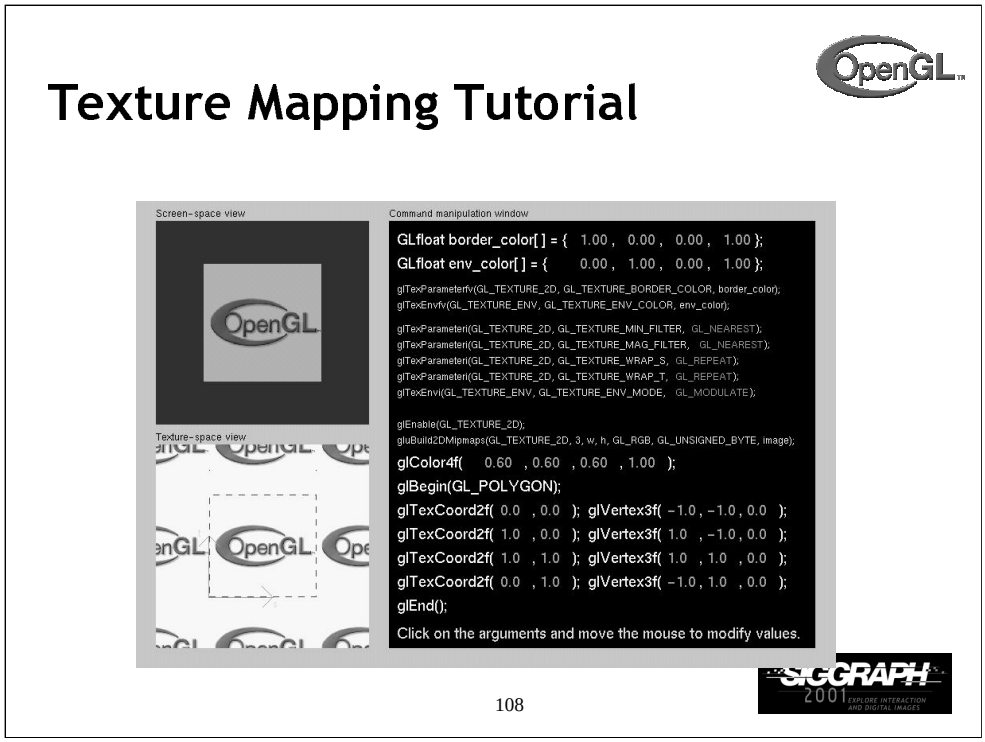
As with lighting positions, texture generation planes are transformed by the ModelView matrix, which allows different results based upon when the `glTexGen()` is issued.

There are three ways in which texture coordinates are generated:

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

`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)

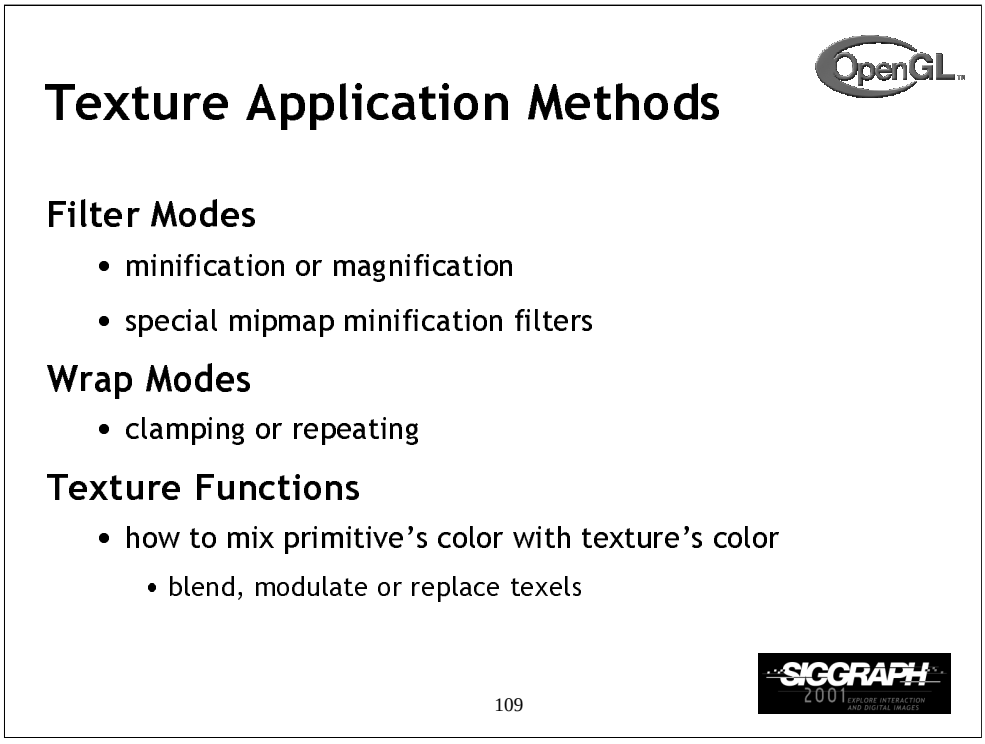


This tutorial demonstrates the power and ease of texture mapping. Begin by merely manipulating the texture and vertex coordinates; notice their relationship.

Next, change the texture filter parameters with the `glTexParameter*()` function. Notice the quality decrease as the mode's changed to `GL_NEAREST`. This mode is the fastest for rendering with texture maps, but has the least quality. As you try other texture filter modes, note the changes in visual quality. Make sure to change the size of the texture quad (or other object) in the screen-space window.

Continue to experiment by changing the wrap modes for the s- and t-coordinates with the `glTexParameter*()` function. Note the effects that changing the wrap modes has when the texture coordinates outside the range [0,1].

Finally, experiment with the texture environment utilizing the `glTexEnv*()` call. The most common texture environment modes are `GL_MODULATE`, and `GL_REPLACE`; the other's are used less, but none less useful.



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

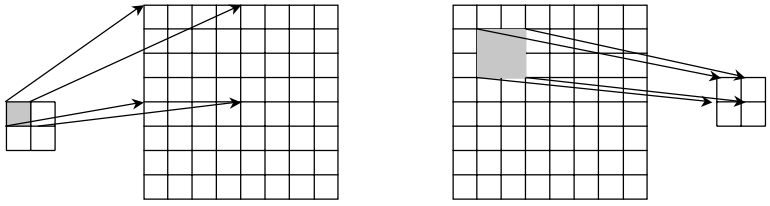
GL_REPEAT - use only the fractional part of the texture coordinate, causing the texture to repeat across an object

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 );
```



110

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 have values of:

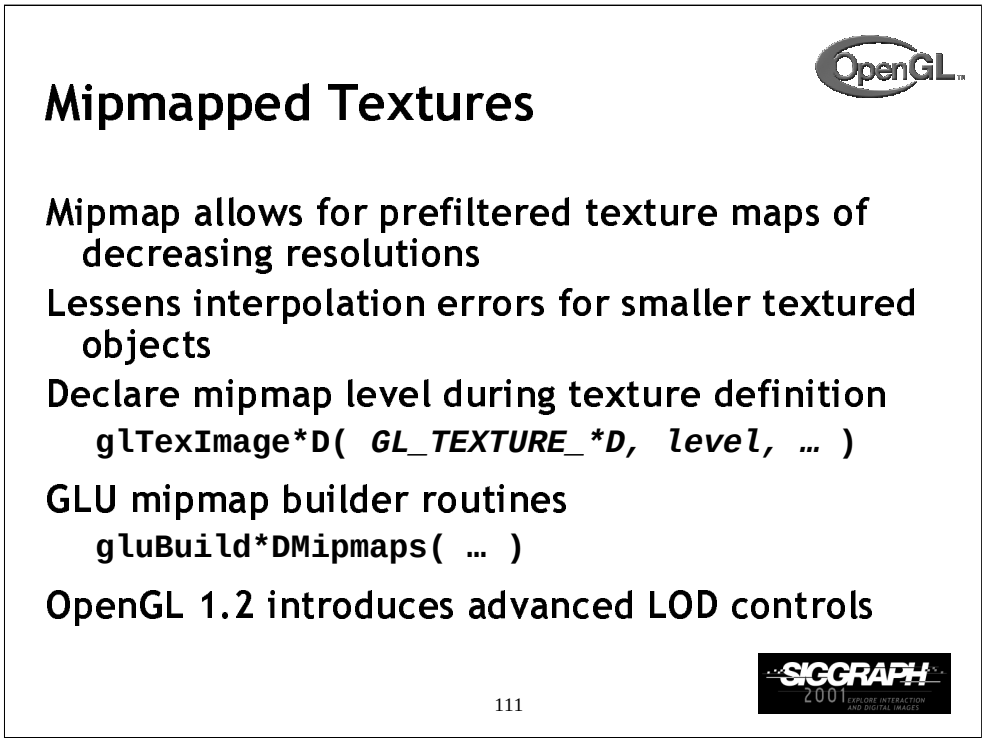
`GL_NEAREST_MIPMAP_NEAREST`

`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.

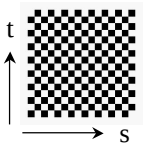
The `gluBuild*Dmipmaps()` routines will automatically generate each mipmap image, and call `glTexImage*D()` with the appropriate level value.

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

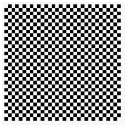
Wrapping Mode

Example:

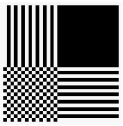
```
glTexParameteri( GL_TEXTURE_2D,
     GL_TEXTURE_WRAP_S, GL_CLAMP )
glTexParameteri( GL_TEXTURE_2D,
     GL_TEXTURE_WRAP_T, GL_REPEAT )
```



texture



GL_REPEAT wrapping

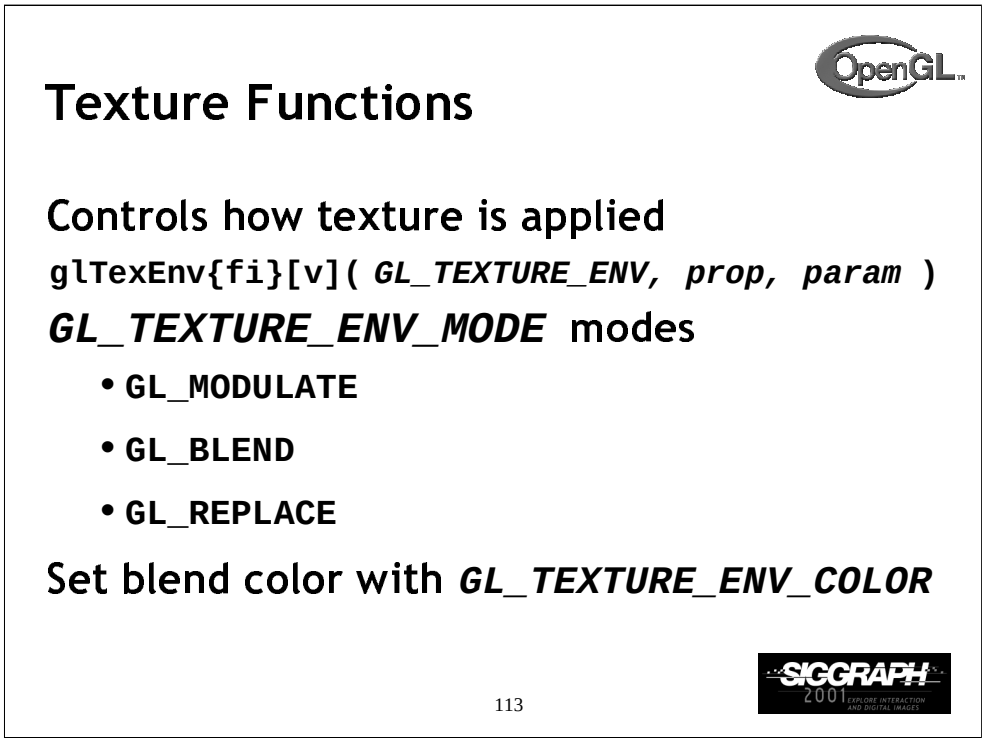


GL_CLAMP wrap

112

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.



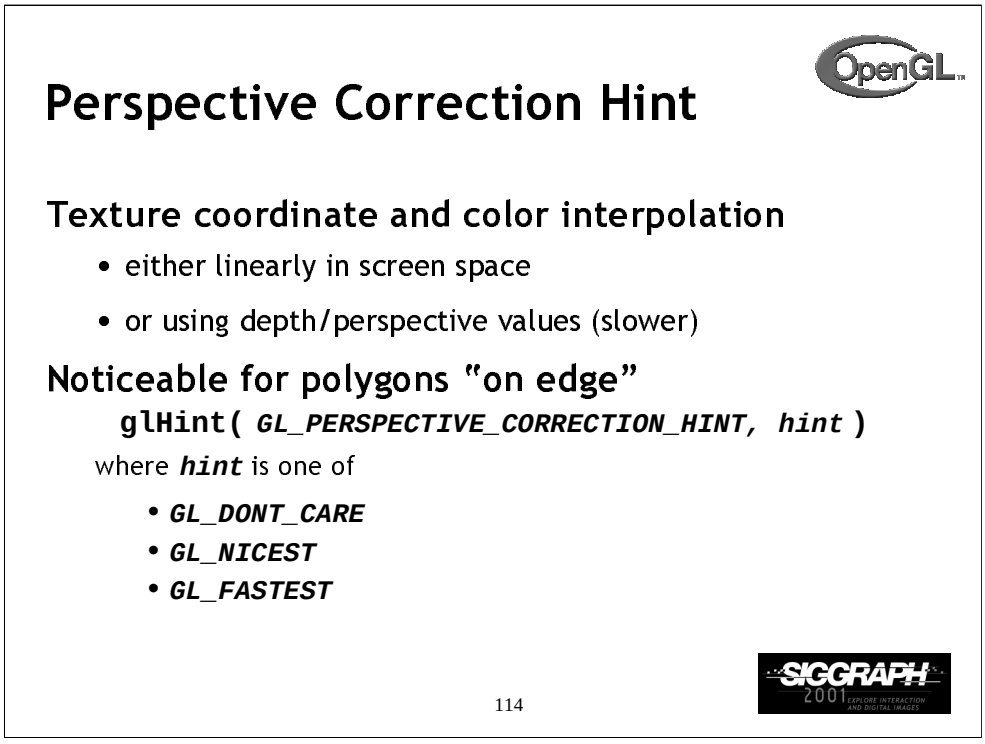
The texture mode determines how texels and fragment colors are combined. The most common modes are:

GL_MODULATE - multiply texel and fragment color

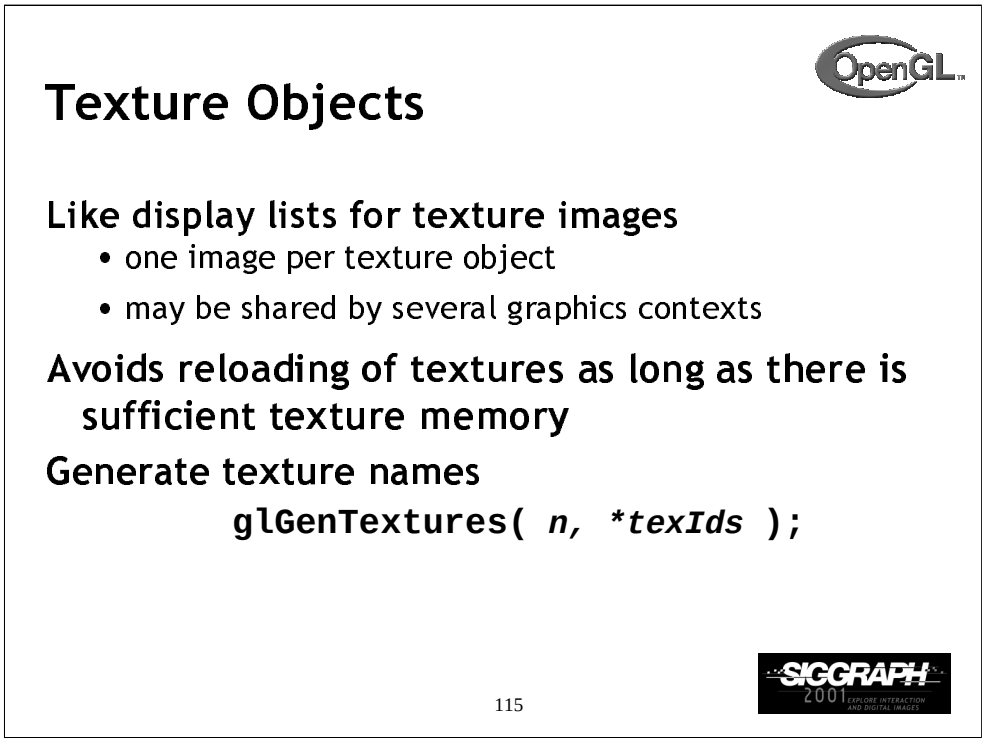
GL_BLEND - linearly blend texel, fragment, env color

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.



When an OpenGL implementation fills the pixels of a texture-mapped, geometric primitive, it is permissible that it doesn't adjust how it samples the texels in the texture map to account for perspective projection. In order to request OpenGL attempt to do perspective-correct texture interpolation, use `glHint()` with the `GL_PERSPECTIVE_CORRECTION_HINT`. An OpenGL implementation may choose to ignore this hint (in the case that the implementation cannot correctly render perspectively rendered primitives), in which case you will still see an incorrect generated image.

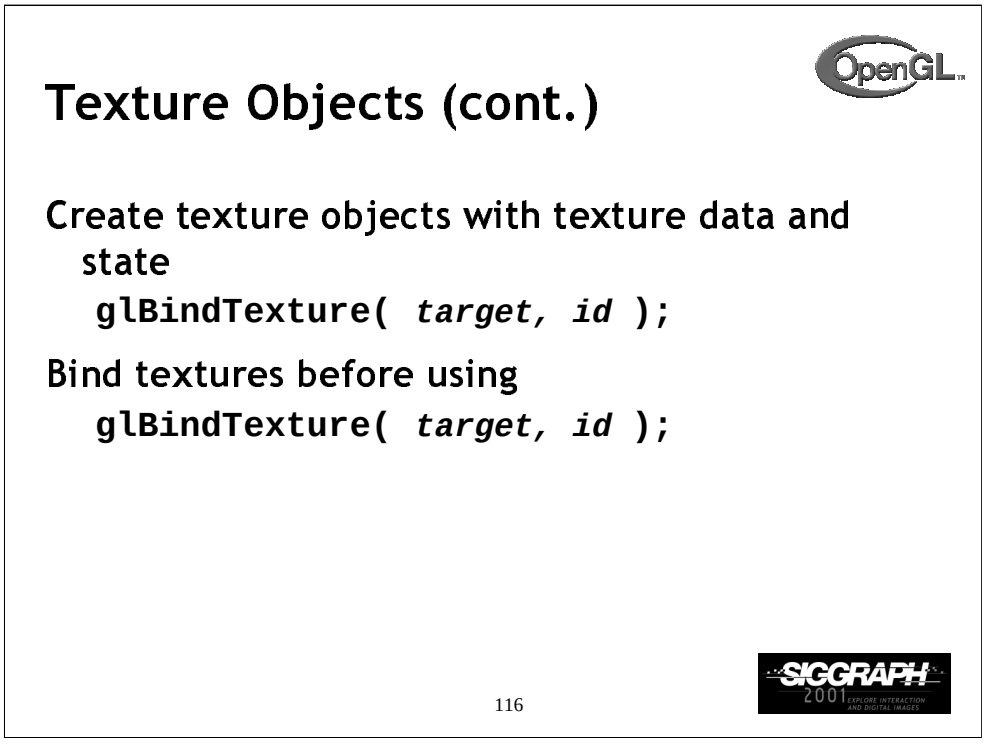


The first step in creating texture objects is to have OpenGL reserve some indices for your objects. `glGenTextures()` will request *n* texture ids and return those values back to you in `texIds`.

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.

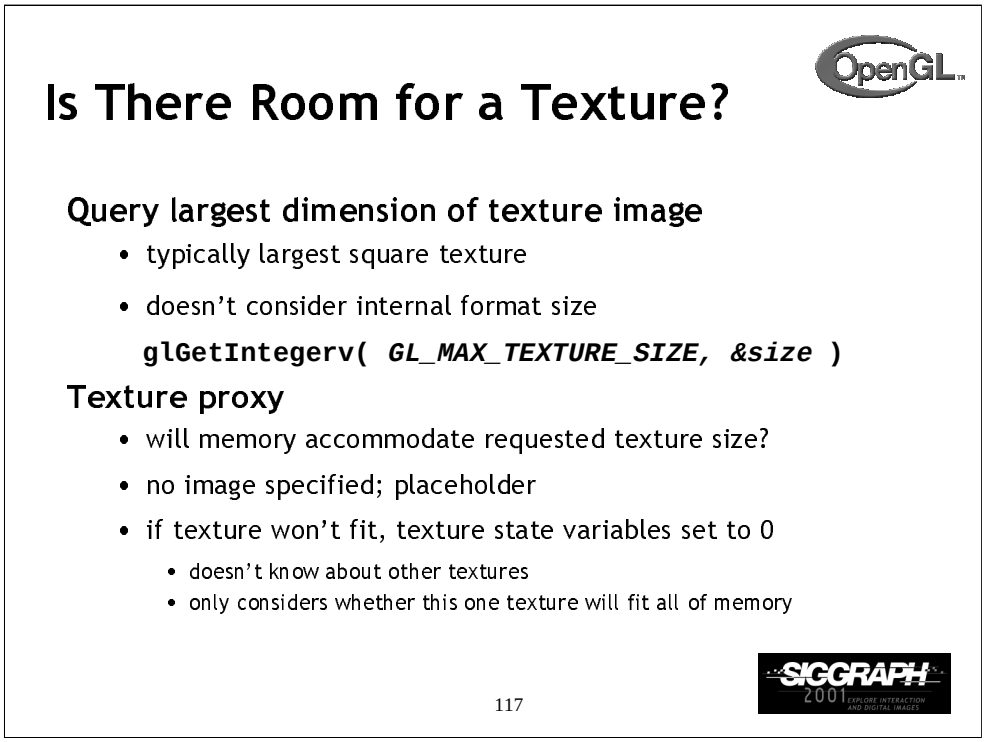
To have OpenGL use a particular texture object, call `glBindTexture()` with the target and id of the object you want to be active.

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



If you utilize more than one texture map in your application, you will generally call `glBindTexture()` more than one time per image. The first time will be to "open" the texture object for creation; when you load the texture and set its parameters. You'll do this one for each texture map you will use, most often in the initialization section of your application.

The second call to `glBindTexture()` should occur before you wish to render object using the texture map stored in that texture object. This will probably occur once per rendered frame (assuming that you need to render object that use this texture).



OpenGL implementations cannot support infinitely sized texture; there's a maximum size that can be supported. Initially, the only information that OpenGL provided on texture sizes was a single number representing the maximum number of texels that could be stored in any one dimension. Unfortunately, this single number didn't account for many of the different parameters that are needed to describe a texture (number of components, bit-size of each component, etc.)

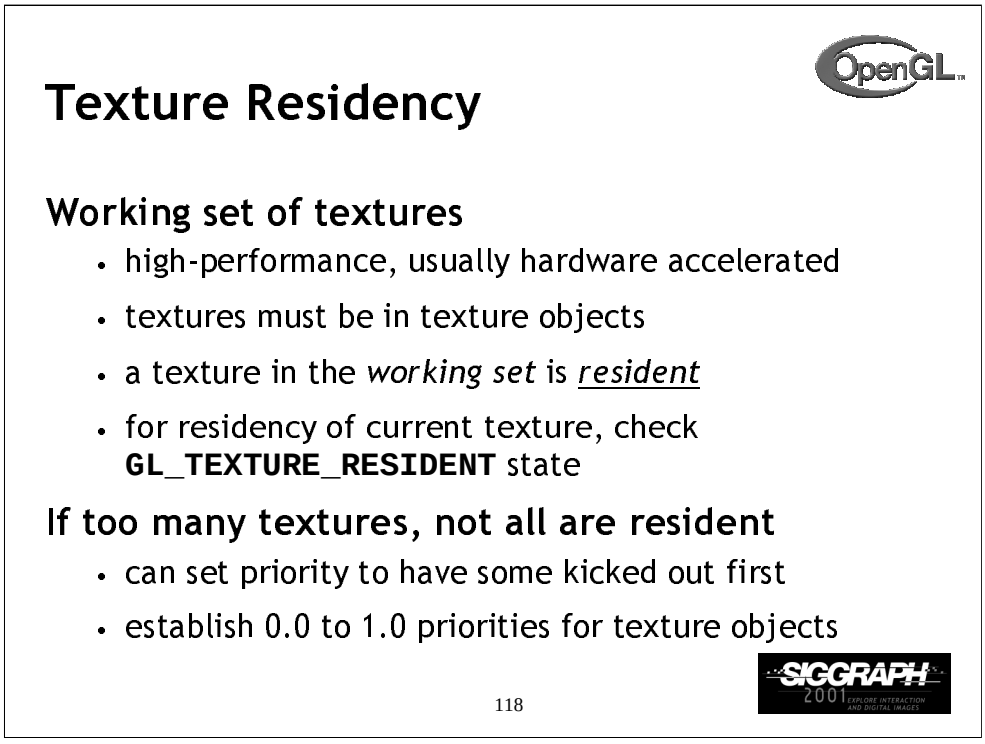
In OpenGL 1.1, *texture proxies* were added that allowed a program to test to see if a particular combination of texture attributes would fit into memory. To query using a texture proxy, make a `glTexImage*D()` call with all the appropriate parameter, except pass NULL in for the texture data, as in:

```
glTexImage2D(GL_PROXY_TEXTURE_2D, 0, GL_RGBA8, 64,
64, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
```

OpenGL will then set up the texture state, which can be queried with `glTexParameter()`. If OpenGL cannot support the requested size of texture, it will set all of the texture state to zero. For example, calling

```
glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D, 0,
GL_TEXTURE_COMPONENTS, &proxyComponents);
```

For a texture that doesn't fill with return `proxyComponents` equal to zero.



Query for residency of an array of texture objects:

```
GLboolean glAreTexturesResident(GLsizei n,
   Gluint *texNums, GLboolean *residences)
```

Set priority numbers for an array of texture objects:

```
glPrioritizeTextures(GLsizei n, GLuint *texNums,
   GLclampf *priorities)
```

Lower priority numbers mean that, in a crunch, these texture objects will be more likely to be moved out of the working set.

One common strategy is avoid prioritization, because many implementations will automatically implement an LRU (least recently used) scheme, when removing textures from the working set.

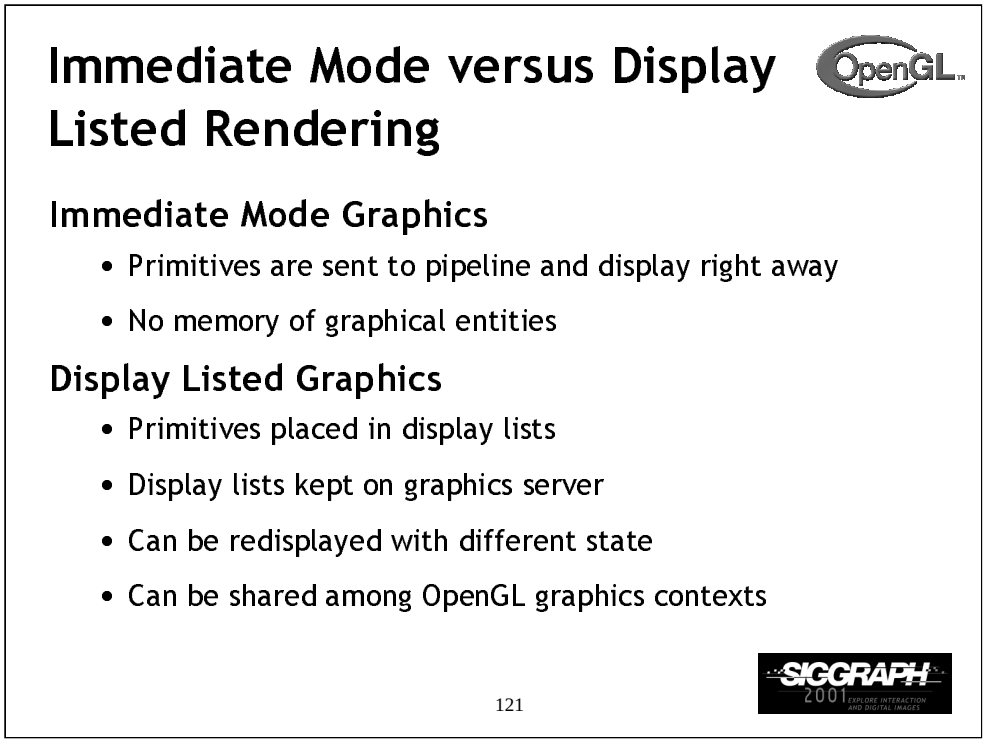
If there is no high-performance working set, then all texture objects are considered to be resident.

Advanced OpenGL Topics

Advanced OpenGL Topics

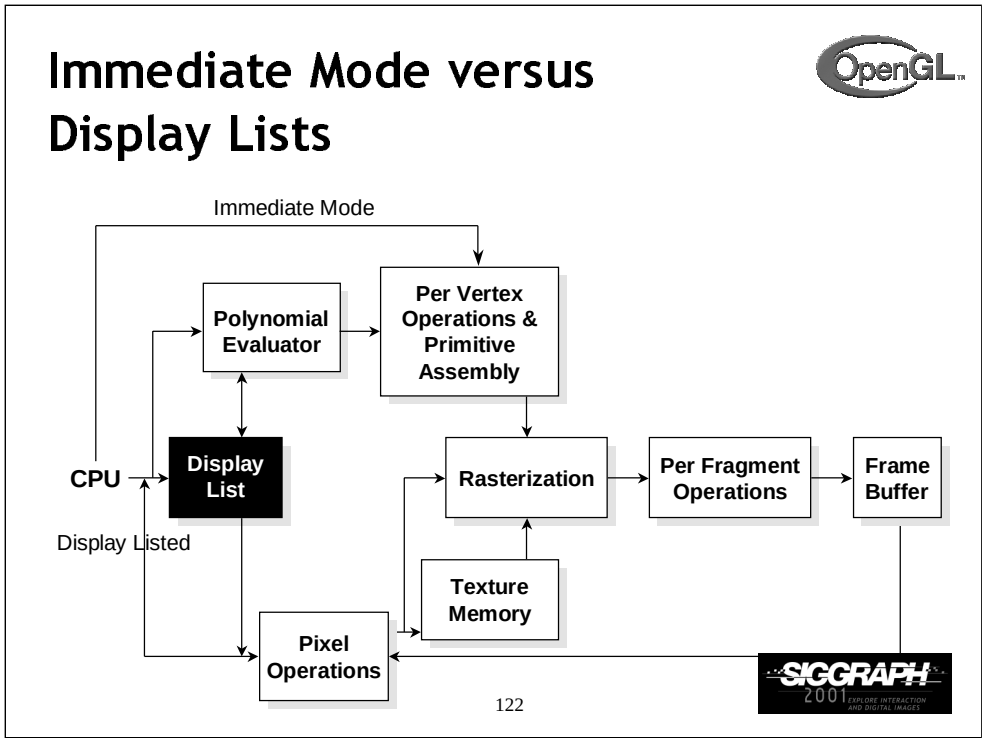
Display Lists and Vertex Arrays
Alpha Blending and Antialiasing
Using the Accumulation Buffer
Fog
Feedback & Selection
Fragment Tests and Operations
Using the Stencil Buffer

120



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

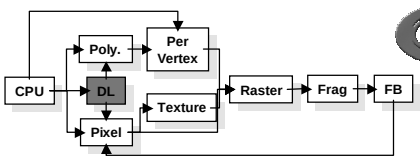
To share display lists among graphics contexts in the X Window System, use the `glXCreateContext()` routine.



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();
}
```

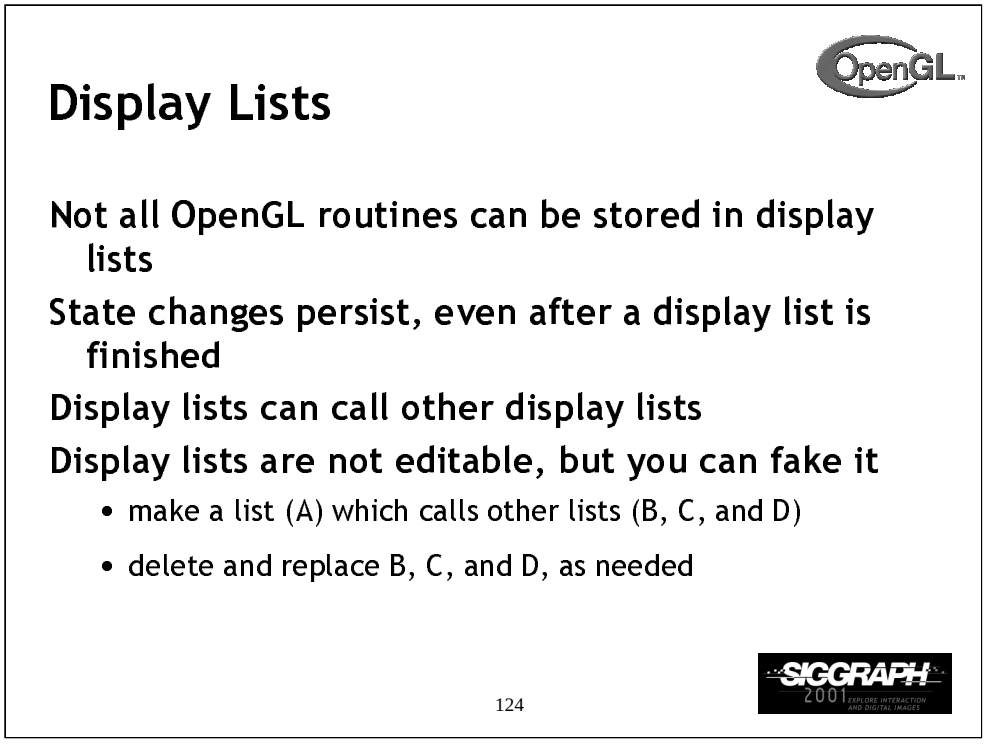
Call a created list

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

123

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.



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`	

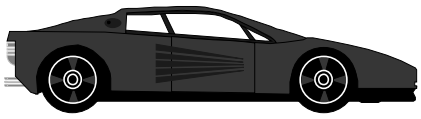
If there is an attempt to store any of these routines in a display list, the routine is executed in immediate mode. No error occurs.

Display Lists and Hierarchy

Consider model of a car

- Create display list for chassis
- Create display list for wheel

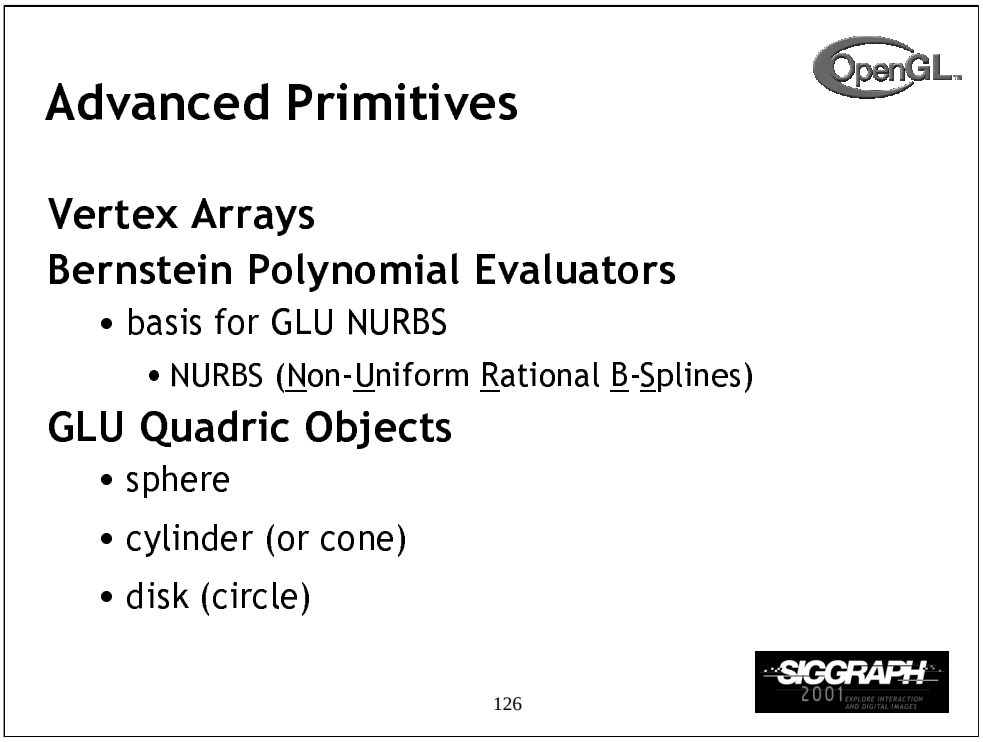
```
glNewList( CAR, GL_COMPILE );
  glCallList( CHASSIS );
  glTranslatef( … );
  glCallList( WHEEL );
  glTranslatef( … );
  glCallList( WHEEL );
     …
glEndList();
```



125

One of the most convenient uses of display lists is repetition of the same graphical object. This technique is sometimes called *instancing*, carried over from the same concept in object-oriented programming.

In the above case, the wheels of the car are all drawn using the same display lists, with the only difference being different modeling transformations used to position each of the tires in the appropriate place.

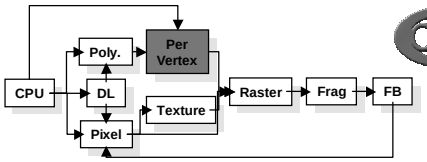


In addition to specifying vertices one at a time using `glVertex*()`, OpenGL supports the use of arrays, which allows you to pass an array of vertices, lighting normals, colors, edge flags, or texture coordinates. This is very useful for systems where function calls are computationally expensive. Additionally,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

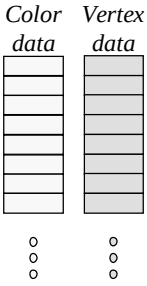
OpenGL evaluators, which automate the evaluation of the Bernstein polynomials, allow curves and surfaces to be expressed algebraically. They are the underlying implementation of the OpenGL Utility Library's NURBS implementation.

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.

Vertex Arrays



Pass arrays of vertices, colors, etc. to OpenGL in a large chunk




```
glVertexPointer( 3, GL_FLOAT, 0, coords )
glColorPointer( 4, GL_FLOAT, 0, colors )
glEnableClientState( GL_VERTEX_ARRAY )
glEnableClientState( GL_COLOR_ARRAY )
glDrawArrays( GL_TRIANGLE_STRIP, 0, numVerts );
```

All active arrays are used in rendering

127

Vertex Arrays allow vertices, and their attributes to be specified in chunks, which reduces the need for sending single vertices and their attributes one call at a time. This is a useful optimization technique, as well as usually simplifying storage of polygonal models.

`glInterleavedArrays()` is a specialized command which substitutes for both calls to `gl*Pointer()` and `glEnableClientState(*).`

When OpenGL processes the arrays, any enabled array is used for rendering. There are three methods for rendering using vertex arrays:

One way is the `glDrawArrays()` routine, which will render the specified primitive type by processing *numVerts* consecutive data elements from the enabled arrays.

A second way is `glDrawElements(),` which allows indirect indexing of data elements in the enabled arrays. This allows shared data elements to be specified only once in the arrays, but be accessed numerous times.

Another way is `glArrayElement()`, which processes a single set of data elements from all activated arrays. As compared to the previous two commands above, `glArrayElement()` must appear between a `glBegin()` / `glEnd()` pair.

For more information on vertex arrays, see chapter 2 of the OpenGL Programming Guide.

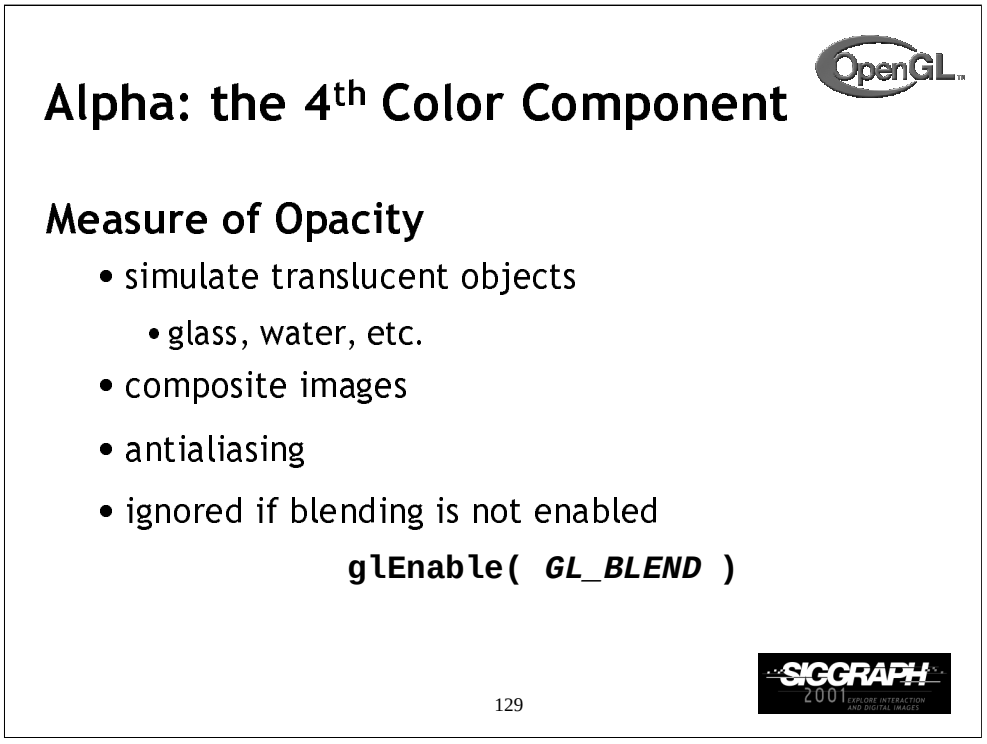
Why use Display Lists or Vertex Arrays?

- May provide better performance than immediate mode rendering
- Display lists can be shared between multiple OpenGL context
  - reduce memory usage for multi-context applications
- Vertex arrays may format data for better memory access

128

Display lists and vertex arrays are principally performance enhancements. On some systems, they may provide better OpenGL performance than immediate mode because of reduced function call overhead or better data organization.

Display lists can also be used to group similar sets of OpenGL commands, like multiple calls to `glMaterial()` to set up the parameters for a particular object. In addition for applications which have multiple OpenGL contexts, display lists can be shared across contexts for less memory usage.

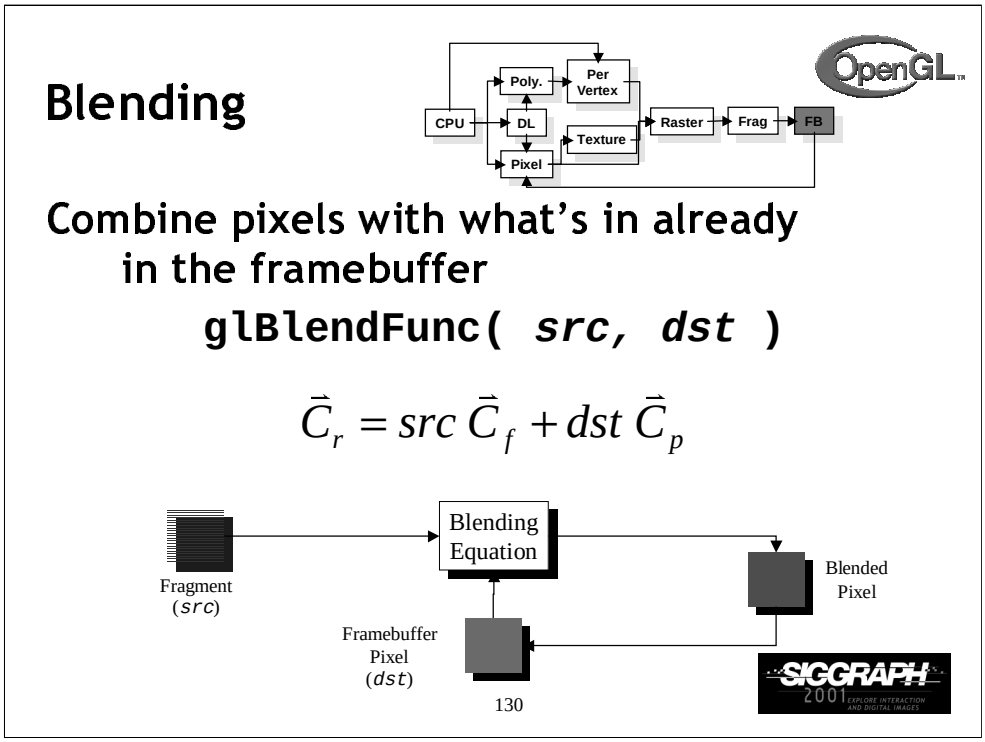


The alpha component for a color is a measure of the fragment's opacity. As with other OpenGL color components, its value ranges from 0.0 (which represents completely transparent) to 1.0 (completely opaque).

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.

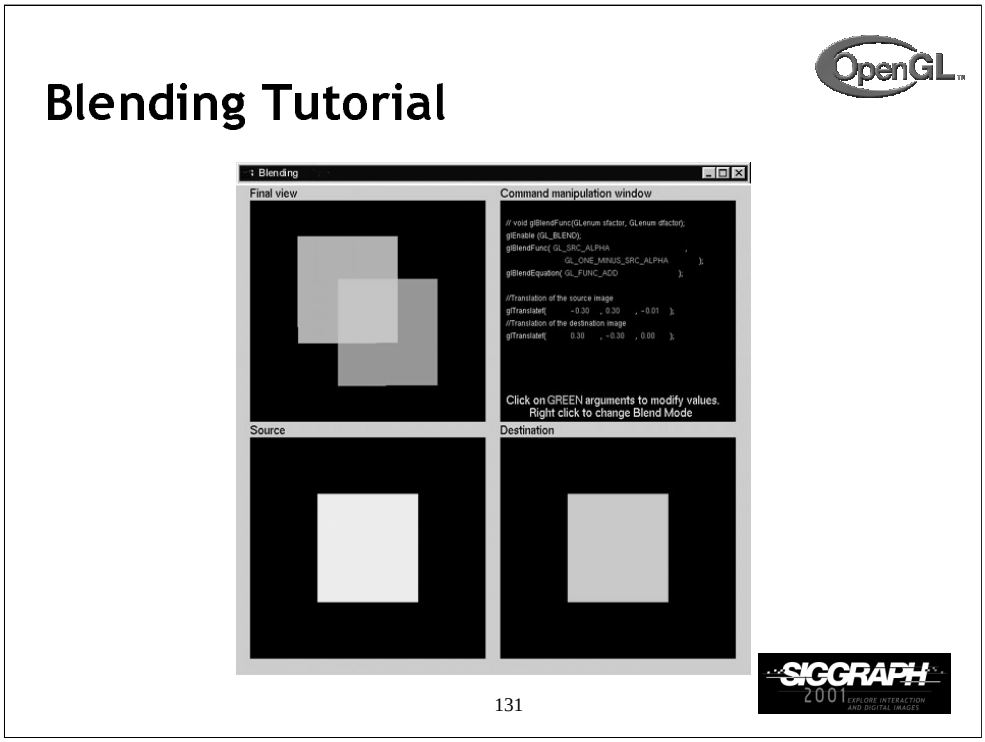
The fragment and pixel each have a factor which controls their contribution to the final pixel color. These *blending factors* are set using `glBlendFunc()`, which sets the source factor, which is used to scale the incoming fragment color, and the destination blending factor, which scales the pixel read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

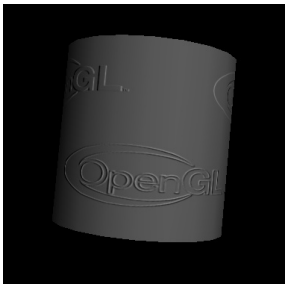


The blending tutorial provides an opportunity to explore how pixel blending can be used to create interest effects. The *source* and *destination* colors are scaled in the manner specified by the parameters set with `glBlendFunc()`. The pixels are combined mathematically using the operation specified in `glBlendEquation()`. `glBlendEquation()`, along with several of the blending modes, are only supported if the `GL_imaging` extension is supported.

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

132

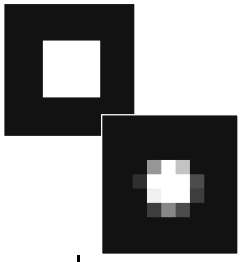
OpenGL blending enables techniques which may require accumulating multiple images of the same geometry with different rendering parameters to be done.

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

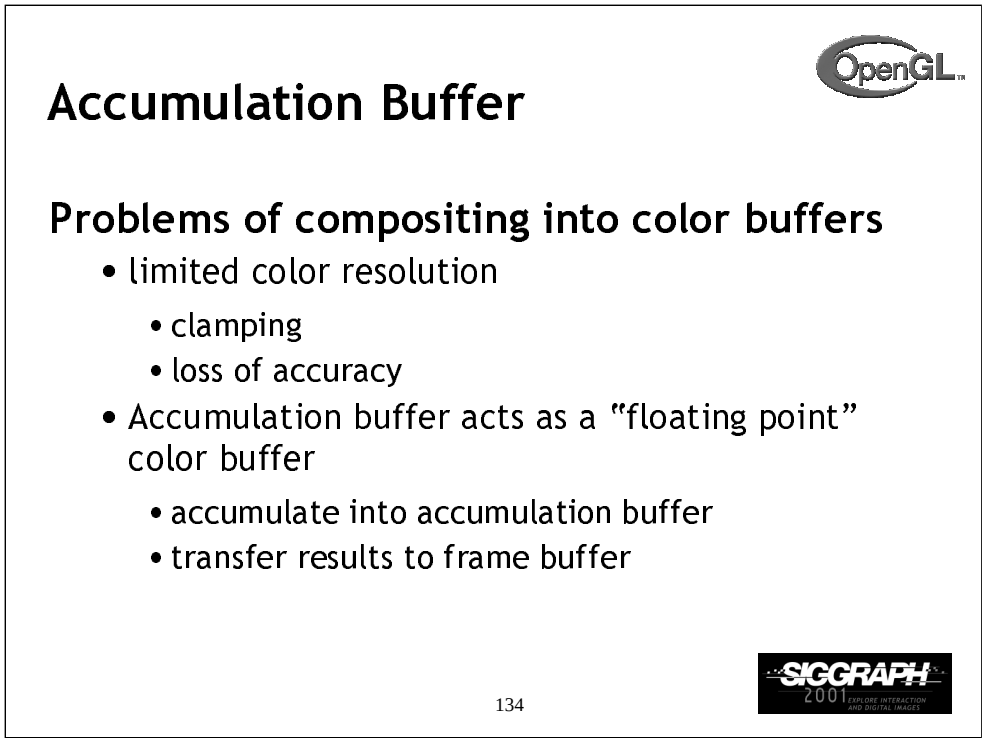
133

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.



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.

Accessing Accumulation Buffer

glAccum(*op, value*)

- operations
  - within the accumulation buffer: ***GL_ADD, GL_MULT***
  - from read buffer: ***GL_ACCUM, GL_LOAD***
  - transfer back to write buffer: ***GL_RETURN***
- **glAccum(*GL_ACCUM, 0.5*)** multiplies each value in write buffer by 0.5 and adds to accumulation buffer

135

If we want to average n images, we can add in each with a value of 1 and read the result with a factor of 1/*n*. Equivalently, we can accumulate each with a factor of 1/*n* and read back with a factor of 1.

Accumulation Buffer Applications

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

136

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.

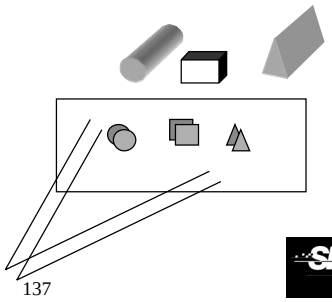
Filtering techniques, such as convolutions and blurs (from image processing) can be done easily in the accumulation buffer by rendering the same image multiple times with slight pixel offsets.

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.

Full Scene Antialiasing : *Jittering the view*

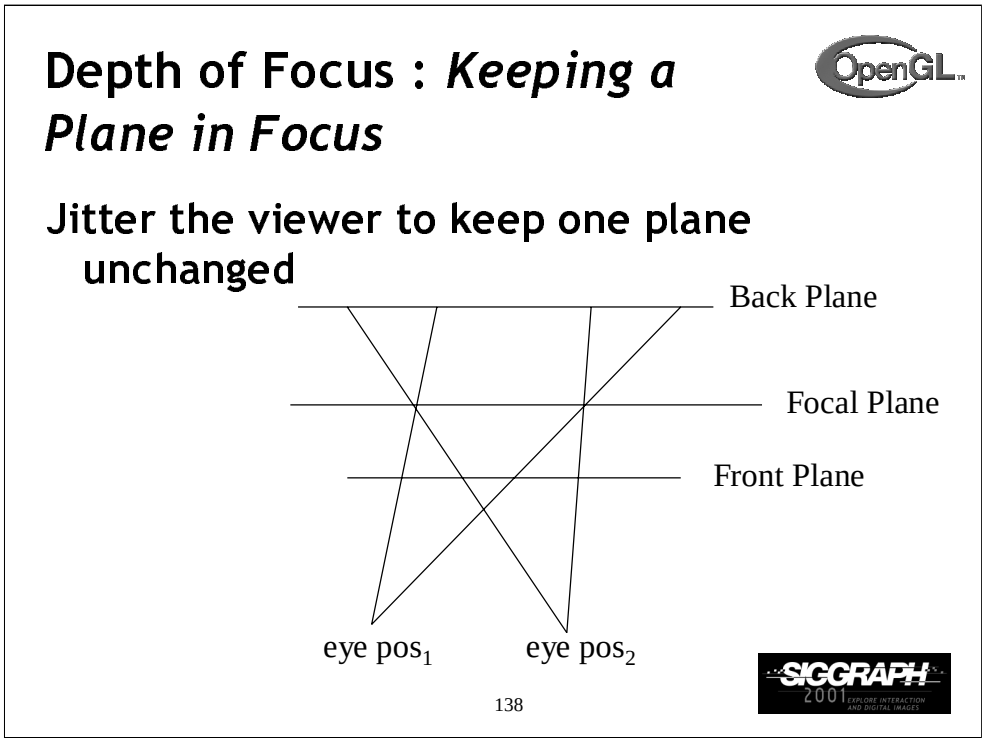
Each time we move the viewer, the image shifts

- Different aliasing artifacts in each image
- Averaging images using accumulation buffer averages out these artifacts

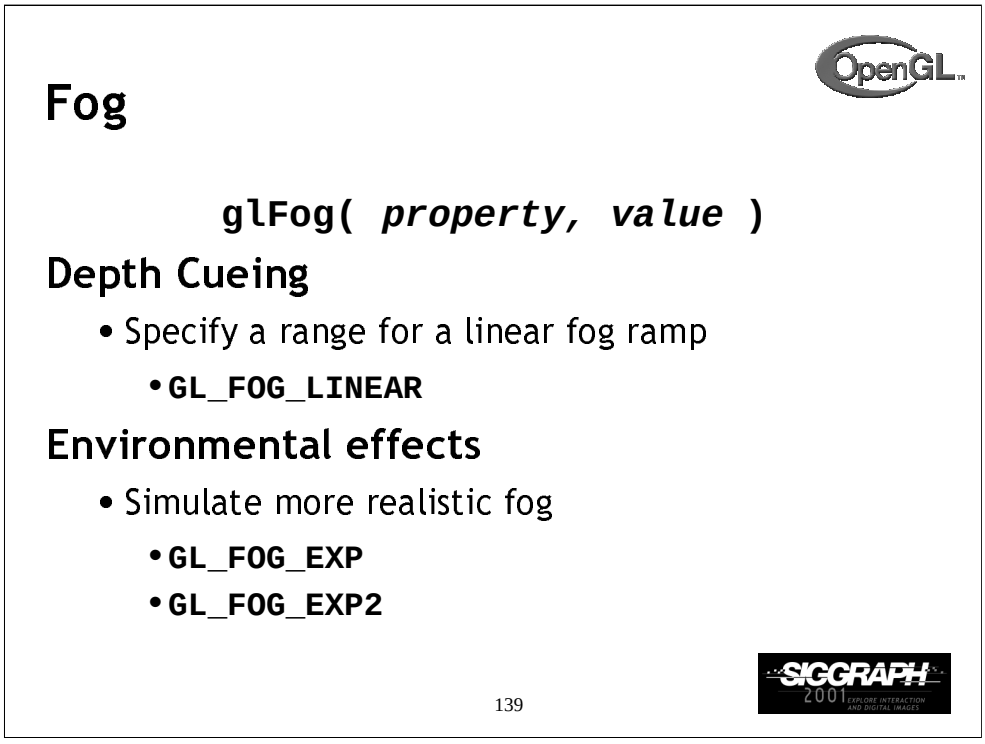


137

Full scene antialiasing, as mentioned, reduces the aliasing artifacts of objects in the scene by combining several renderings of the same scene, with each rendering done from a slightly different viewpoint. Since the viewpoint is only changed a little for each rendering pass, most of the scene looks very similar, but when all the images are composited together, the hard edges are averaged away.



Depth of field images can be produced by shifting the eyepoint around in the same parallel plane as to the focal plane. By compositing the resulting images together, objects near the center of the viewing frustum are kept in focus, while objects farther from the focal plane are composited to be a little blurry.



Fog works in two modes:

Linear fog mode is used for depth cueing effects. 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.

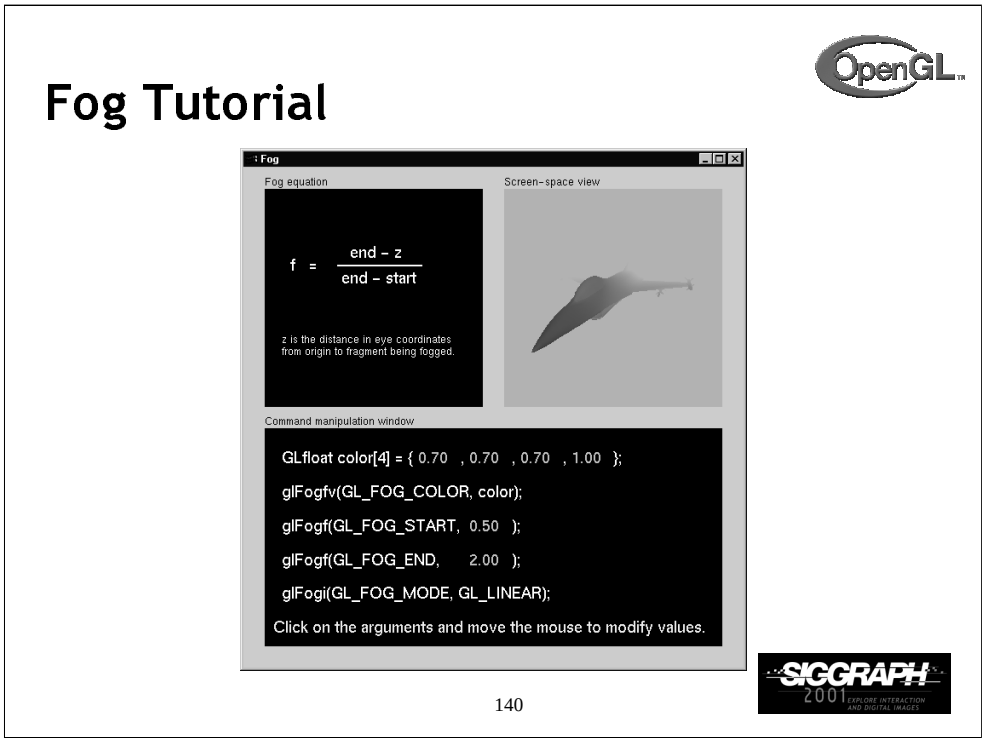
In this mode, the fog coefficient is computed as $f = \frac{z - start}{end - start}$

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

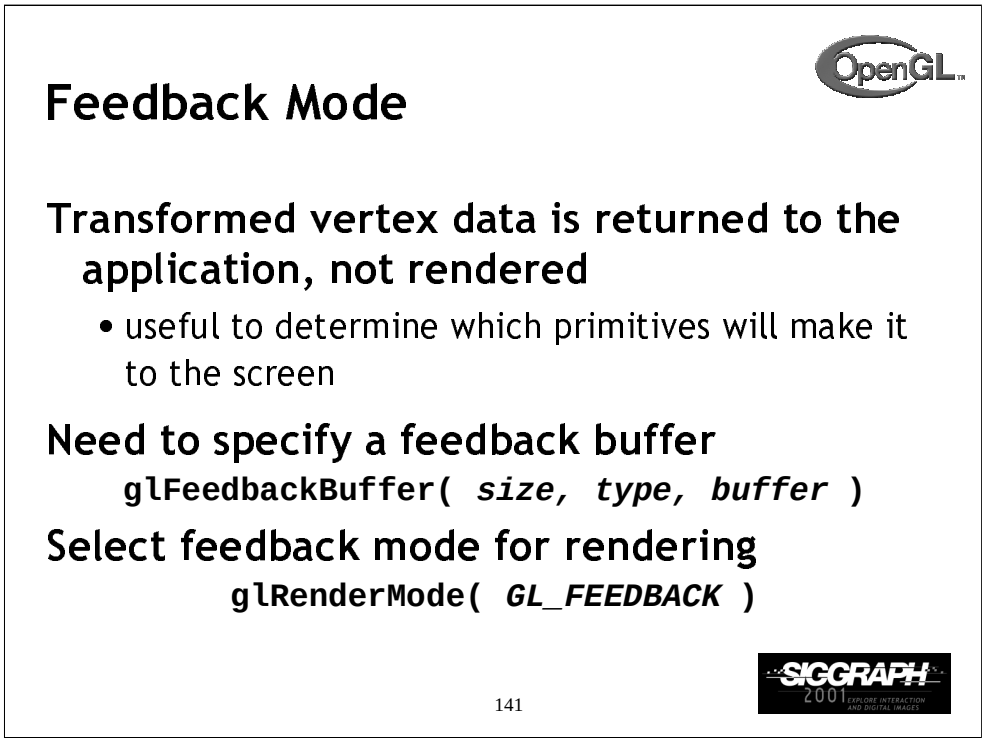
```
glFogf(GL_FOG_MODE, GL_FOG_LINEAR);
glFogf(GL_FOG_START, fogStart);
glFogf(GL_FOG_END, fogEnd);
glFogfv(GL_FOG_COLOR, fogColor);
glEnable(GL_FOG);
```

Exponential fog mode is used for more natural environmental effects 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).

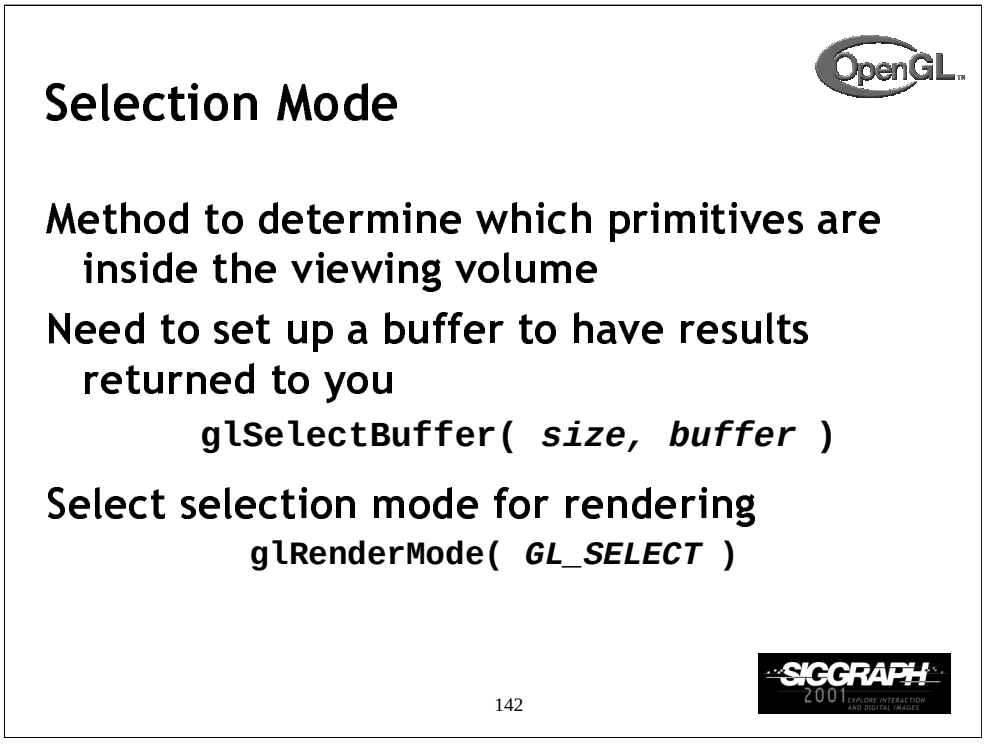


Feedback mode is useful for determining which primitives will eventually be rendered, after transformation and clipping. The data returned back from feedback mode is dependant on what `type` of data was requested.

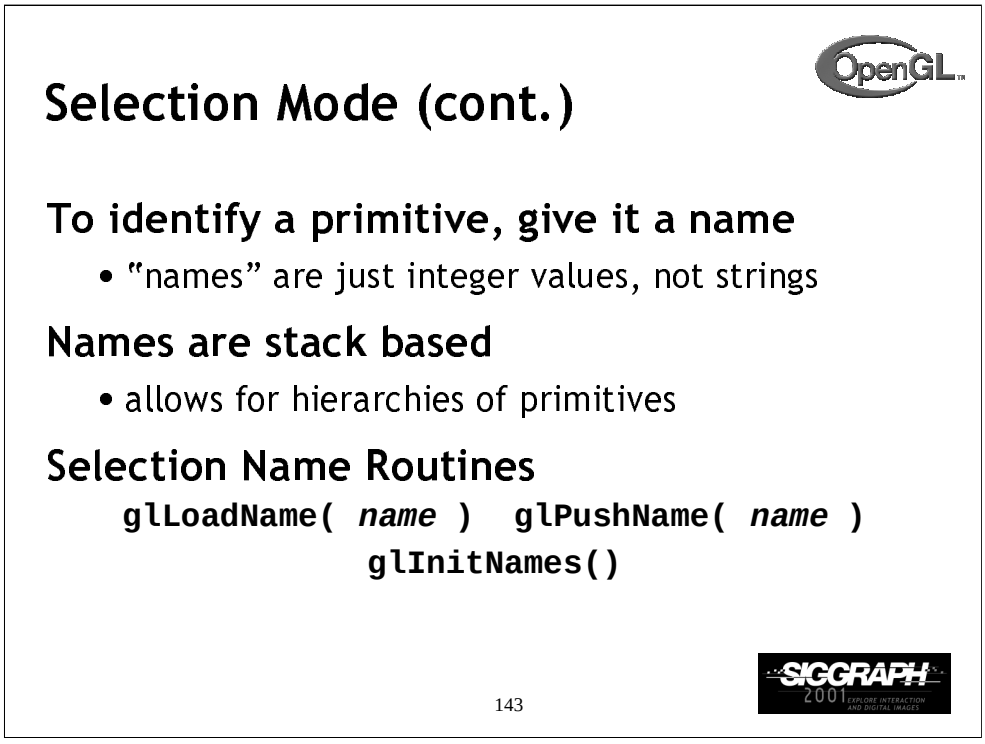
Possible types of values which can be returned are:

- 2D or 3D vertex values
- 3D vertex data with color
- 4D vertex data with color and texture

Each set of vertex data returned is delineated with a token representing what type of primitive was rendered (`GL_POINT_TOKEN, GL_LINE_TOKEN, GL_POLYGON_TOKEN, GL_BITMAP_TOKEN, GL_DRAW_PIXELS_TOKEN, GL_PASS_THROUGH_TOKEN`) , followed by the requested data.



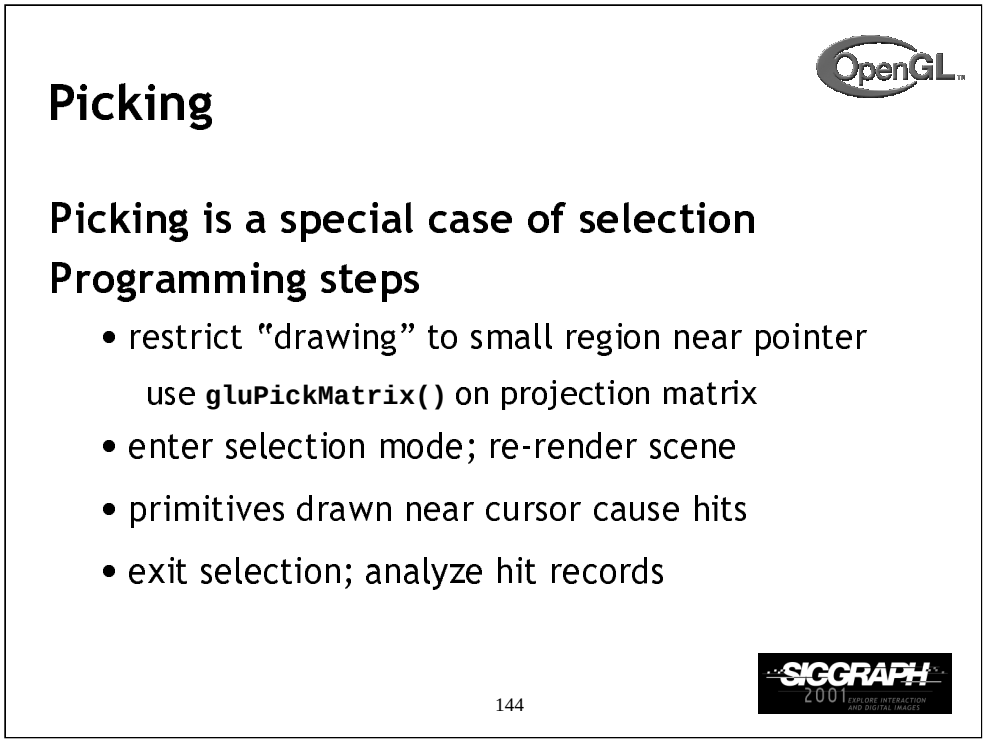
Selection mode is a way to determine which primitives fall within the viewing volume. As compared to feedback mode, where all the vertex data for a primitive is returned to you, selection mode only returns back a “name” which you assign for the primitive.



Selection mode uses *names* to identify primitives that pass the selection test. Any number of primitives can share the same name, allowing groups of primitives to be identified as a logical object.

After specifying the selection buffer, it must be initialized first by calling `glPushName()`. A hierarchy of names can be set up by calling `glPushName()` to obtain a new level in the hierarchy, and `glLoadName()` to uniquely name each node in the hierarchy.

`glInitNames()` can be used to completely clear out an existing name hierarchy.



The picking region is usually specified in a piece of code like this:

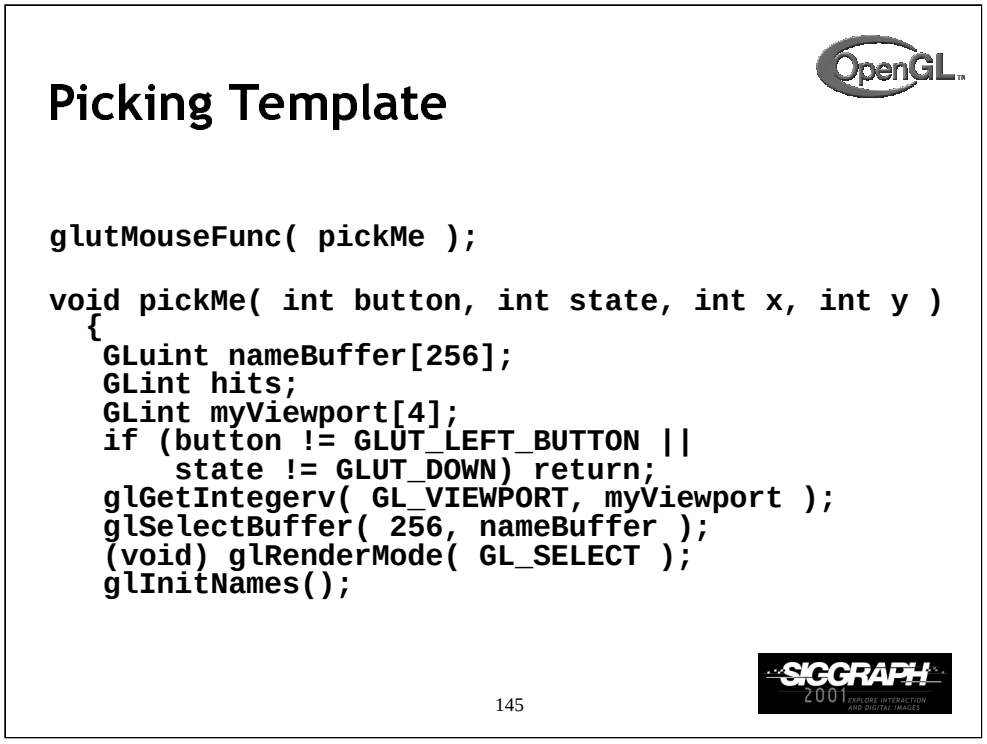
```
glMatrixMode (GL_PROJECTION);
glLoadIdentity();
gluPickMatrix(x, y, width, height, viewport);
```

`gluPerspective(...)` or `glOrtho(...)`

The picking matrix is the rare situation where the standard projection matrix (perspective or ortho) is multiplied onto a non-identity matrix.

Each *hit record* contains:

- number of names per hit
- smallest and largest depth values
- all the names



In this example, we specify a function to be called when a mouse button is pressed.

The routine which is called specifies the selection buffer, and switches into selection mode for retrieving which objects fall within the picking region.

The `glInitNames()` function resets the selection buffer to its default state.

Picture Template (cont.)

```
   glMatrixMode( GL_PROJECTION );
   glPushMatrix();
   glLoadIdentity();
   gluPickMatrix( (GLdouble) x, (GLdouble)
      (myViewport[3]-y), 5.0, 5.0, myViewport );
/*   gluPerspective or glOrtho or other projection  */
   glPushName( 1 );
/*   draw something  */
   glLoadName( 2 );
/*   draw something else … continue …  */
```

146

We continue the example by specifying the picking region using `gluPickMatrix()` and then specifying our normal projection transformation. Continuing, we initialize the name stack by calling `glPushName()` (remember, you need to do a `glPushName()`, and not a `glLoadName()` first).

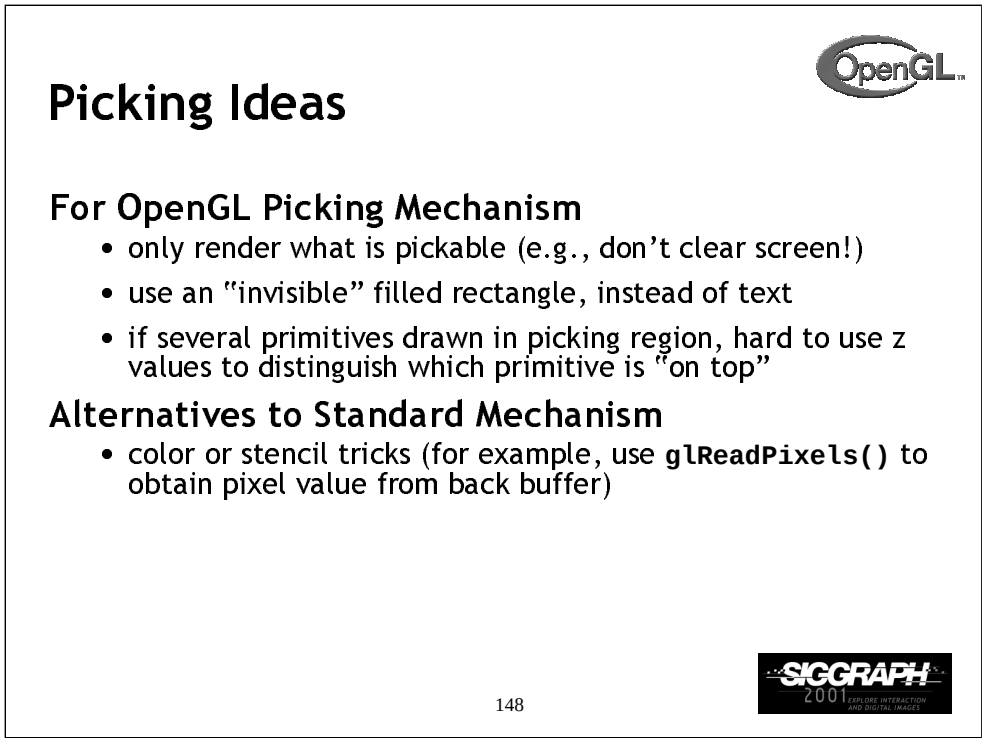
Finally, we render all the objects in our scene, providing new names for the selection buffer as necessary.

Picking Template (cont.)

```
   glMatrixMode( GL_PROJECTION );
   glPopMatrix();
   hits = glRenderMode( GL_RENDER );
/*   process nameBuffer  */
}
```

147

Completing our example, we restore the projection matrix to its pre-pick matrix mode, and process our hits with the data returned back to us in the selection buffer provided previously.

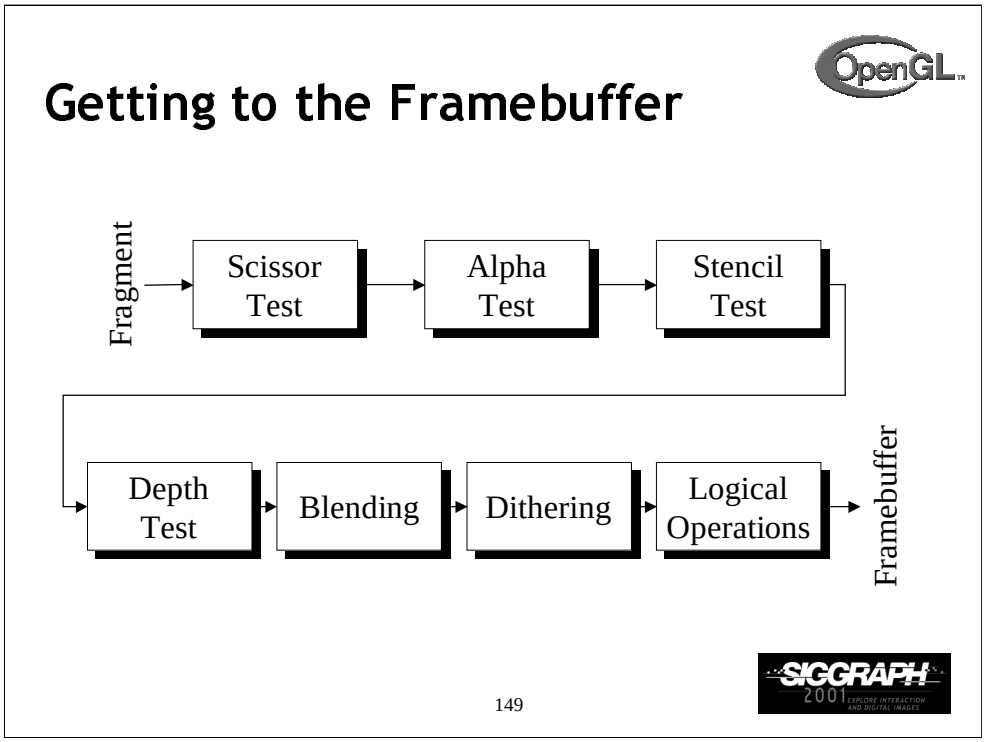


There are a few tricks that make picking more useful and simpler to use:

- In order to make picking as fast as possible, only render what's pickable.
- Try to use simple geometry to simulate a more complex object. For example, use a filled rectangle as compared to a text string, or the bounding sphere of a complicated polygonal object.

The selection mechanism returns the depth values of objects, which can be used to sort objects based on their distance from the eyepoint.

OpenGL selection and picking methods aren't the only ways to determine what primitives are on the screen. In some cases, it may be faster and easier to use unique colors for each object, render the scene into the back-buffer, and read the pixel or pixels which are of interest (like the hot spot on a cursor). Looking up the color may be much faster and more direct than parsing the hit list returned from selection.



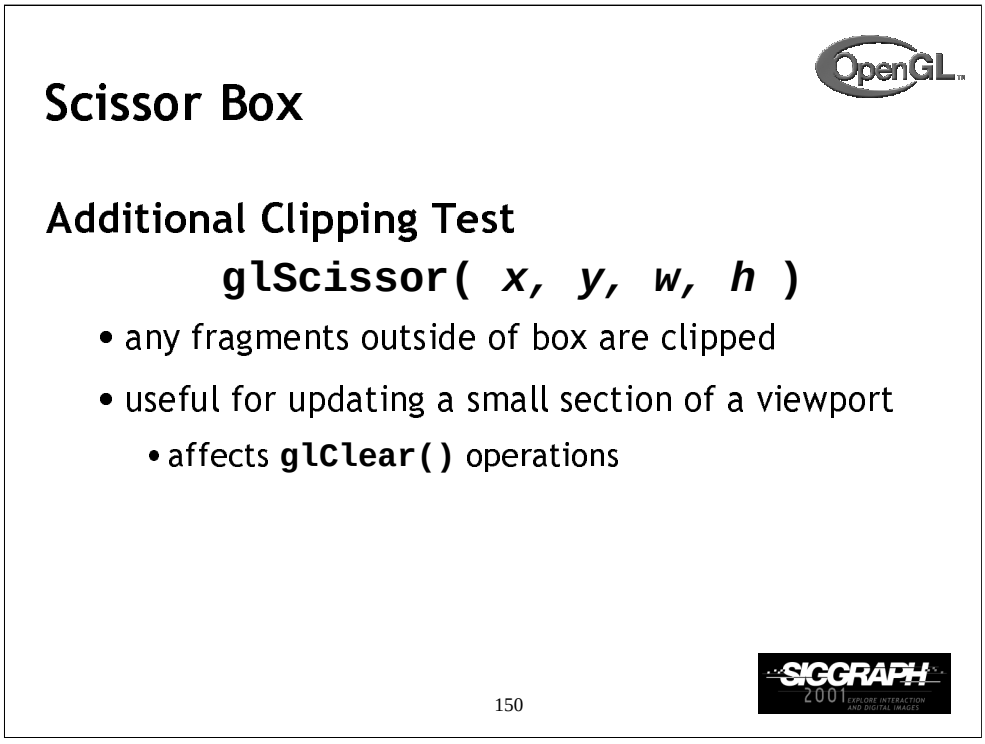
In order for a fragment to make it to the frame buffer, it has a number of testing stages and pixel combination modes to go through.

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.

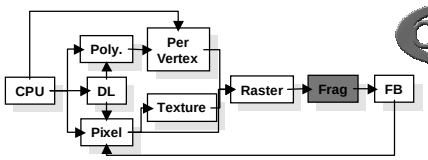
If a fragment passes all enabled tests, it is then blended, dithered and/or logically combined with pixels in the framebuffer. Each of these operations can be enabled and disabled.



The *scissor test* provides an additional rectangular clipping test in addition to clipping to the viewport. This is useful for clearing only particular parts of the viewport (`glClear()` is not bounded by the viewport clipping operation), and restricting pixel updates to a small region of the viewport.

The scissor test can be enabled with `glEnable(GL_SCISSOR_TEST);`

Alpha Test

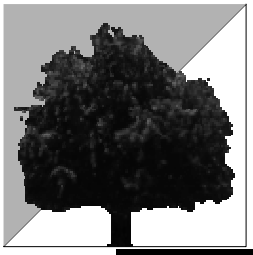


Reject pixels based on their alpha value

`glAlphaFunc( func, value )`

`glEnable( GL_ALPHA_TEST )`

- use alpha as a mask in textures



151

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.

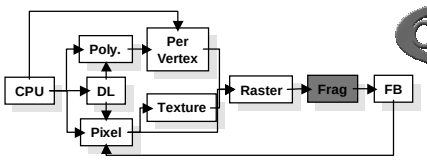
The functions which `glAlphaFunc()` can use are:

`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.

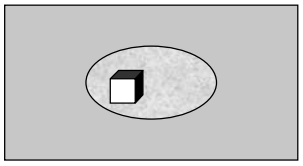
Alpha testing is particularly useful when combined with texture mapping with textures which have an alpha component. This allows your texture map to act as a localized pixel mask. This technique is commonly used for objects like trees or fences, where modeling the objects (and all of its holes) becomes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

Stencil Buffer



Used to control drawing based on values in the stencil buffer

- Fragments that fail the stencil test are not drawn
- Example: create a mask in stencil buffer and draw only objects not in mask area



152

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.

Controlling Stencil Buffer

glStencilFunc(*func, ref, mask*)

- compare value in buffer with **ref** using **func**
- only applied for bits in **mask** which are 1
- **func** is one of standard comparison functions

glStencilOp(*fail, zfail, zpass*)

- Allows changes in stencil buffer based on passing or failing stencil and depth tests: **GL_KEEP, GL_INCR**

153

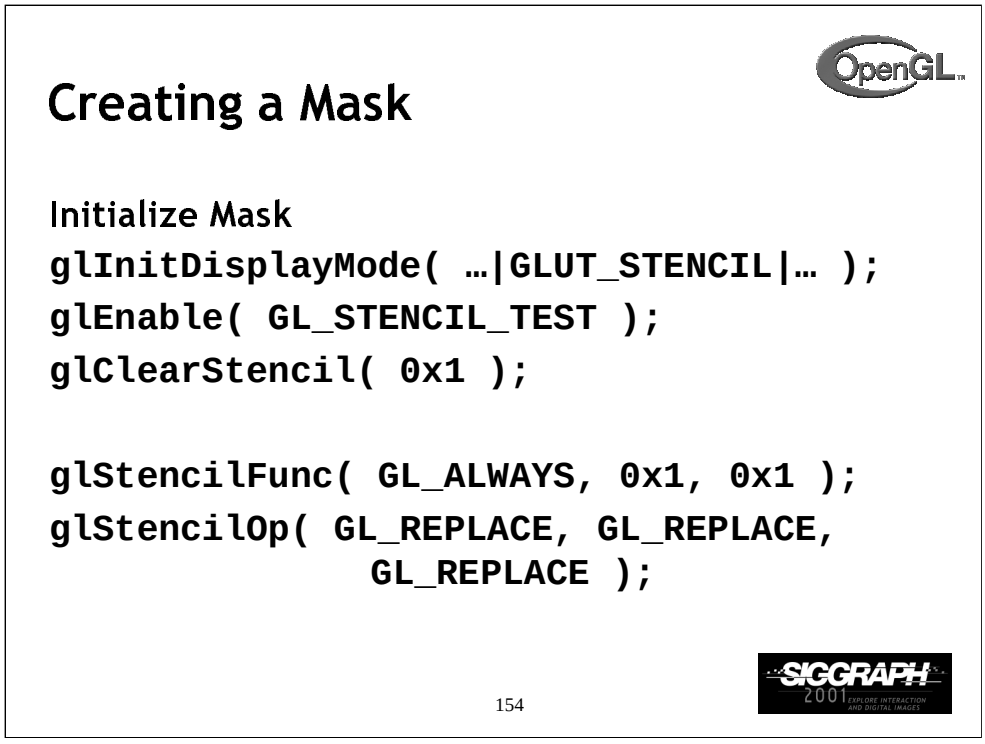
The two principal functions for using the stencil buffer are `glStencilFunc()` which controls how the bits in the stencil buffer are used to determine if a particular pixel in the framebuffer is writable.

`glStencilOp()` controls how the stencil buffer values are updated, based on three tests:

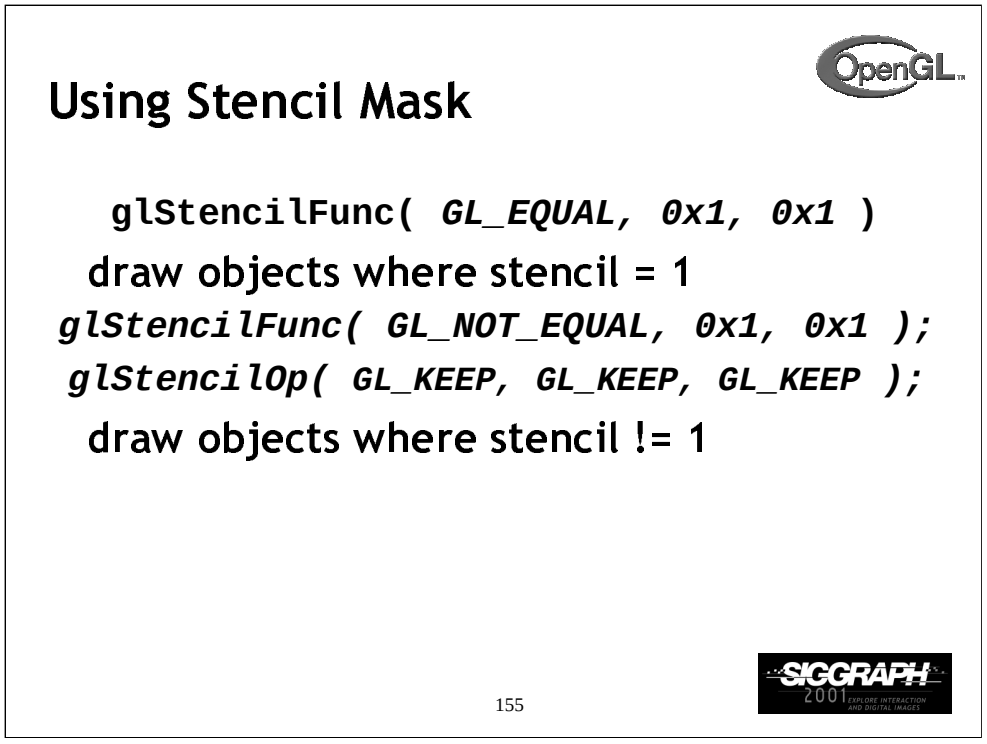
1) did the pixel pass the stencil test specified with `glStencilFunc()`

2) did the pixel fail the depth test for that pixel.

3) did the pixel pass the depth test for that pixel. This would mean that the pixel in question would have appeared in the image.

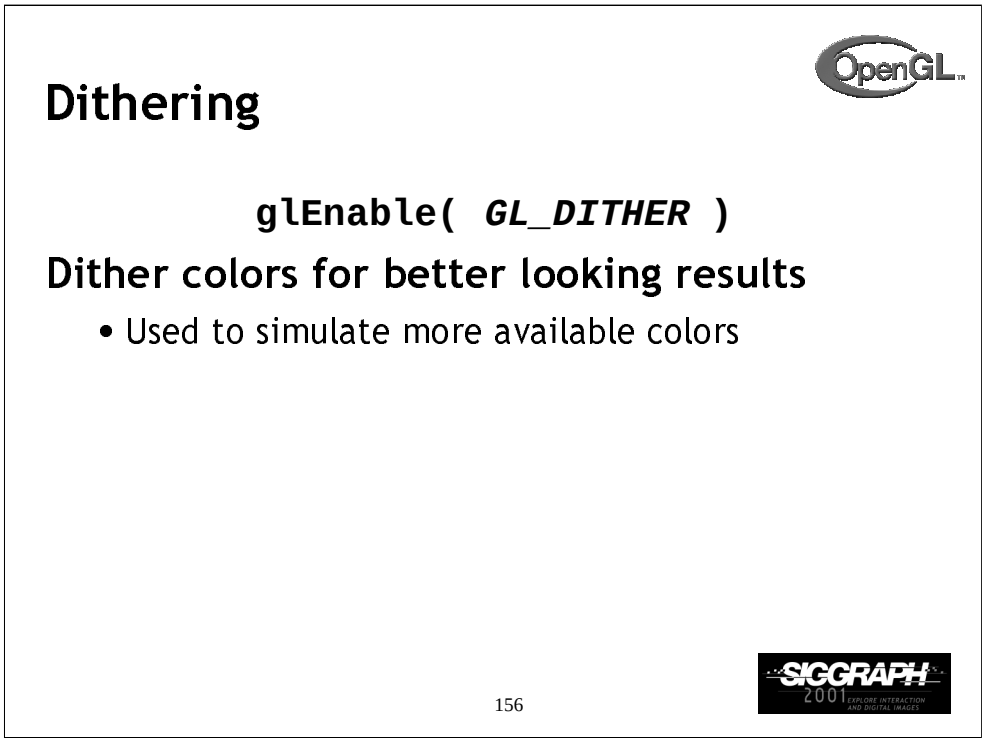


In this example, we specify a simple stencil mask. We do this by specifying that regardless of which tests the pixel passes or fails, we replace its value in the stencil buffer with the value 0x1. This permits us to render the shape of the pixel mask we want directly into the stencil buffer (in a manner of speaking).

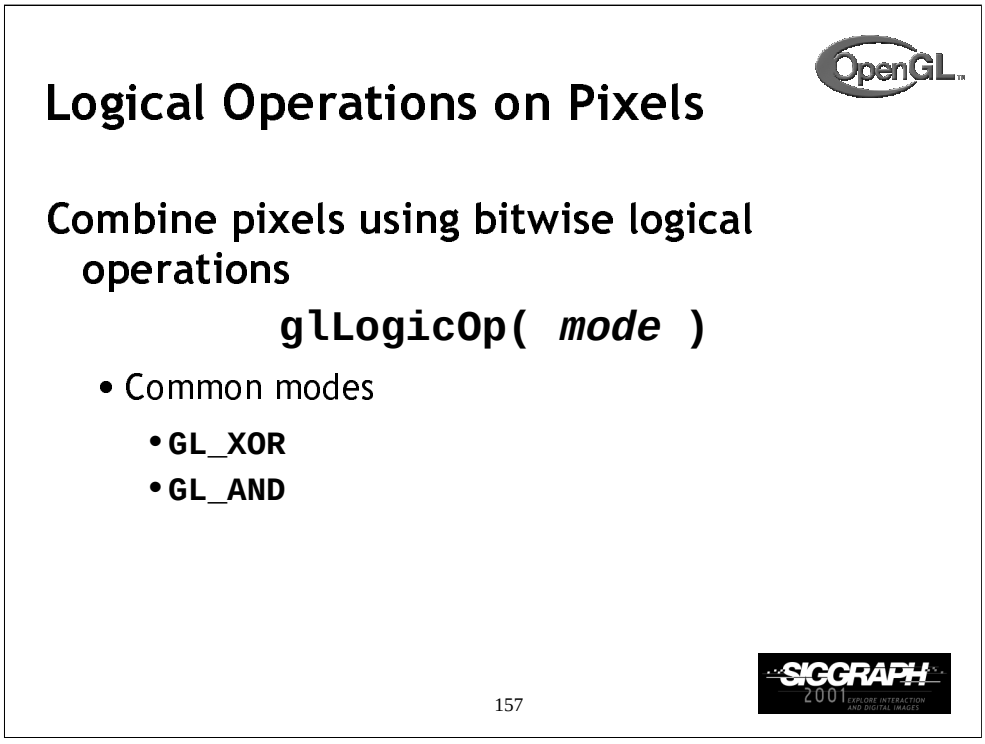


After the stencil mask is specified, we can use the mask to selectively update pixels. With the first set of commands, we only update the pixels where the stencil buffer is set to 0x01 in the stencil buffer.

In the second example, we set the stencil state up to render only to pixels where the stencil value is not 0x01.



Dithering is a technique to trick the eye into seeing a smoother color when only a few colors are available. Newspaper's use this trick to make images look better. OpenGL will modify a fragment's color value with a dithering table before it is written into the framebuffer.

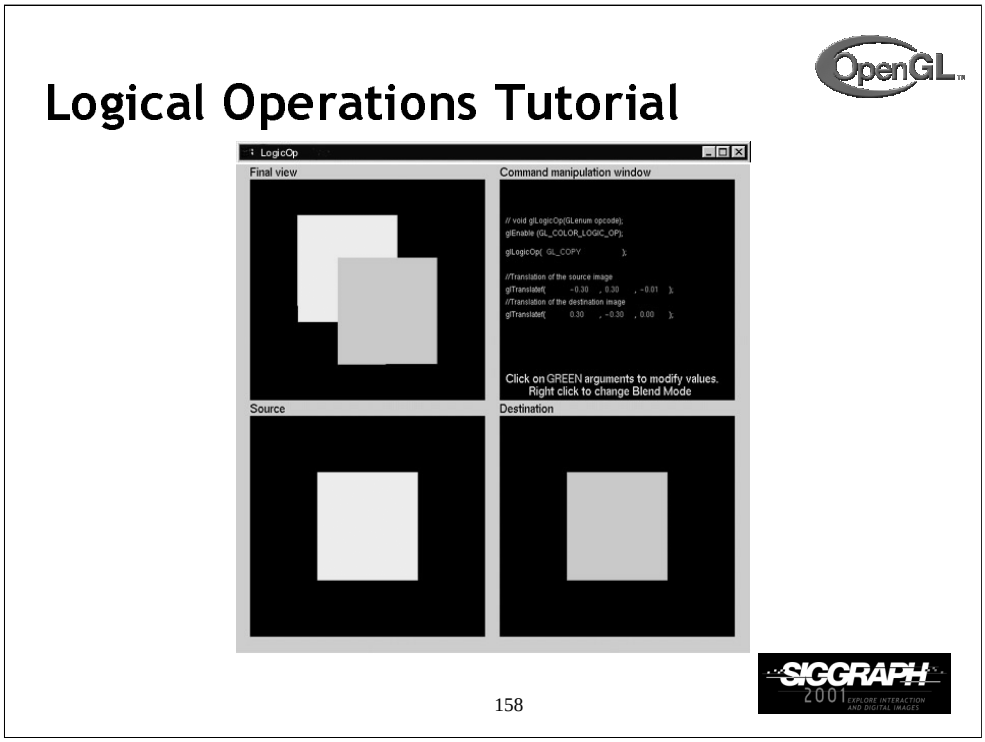


Logical operations allows pixels to be combined with bitwise logical operations, like logical ands, ors and nots. The fragment's color bits are combined with the pixel's color bits using the logical operation, and then written into the framebuffer.

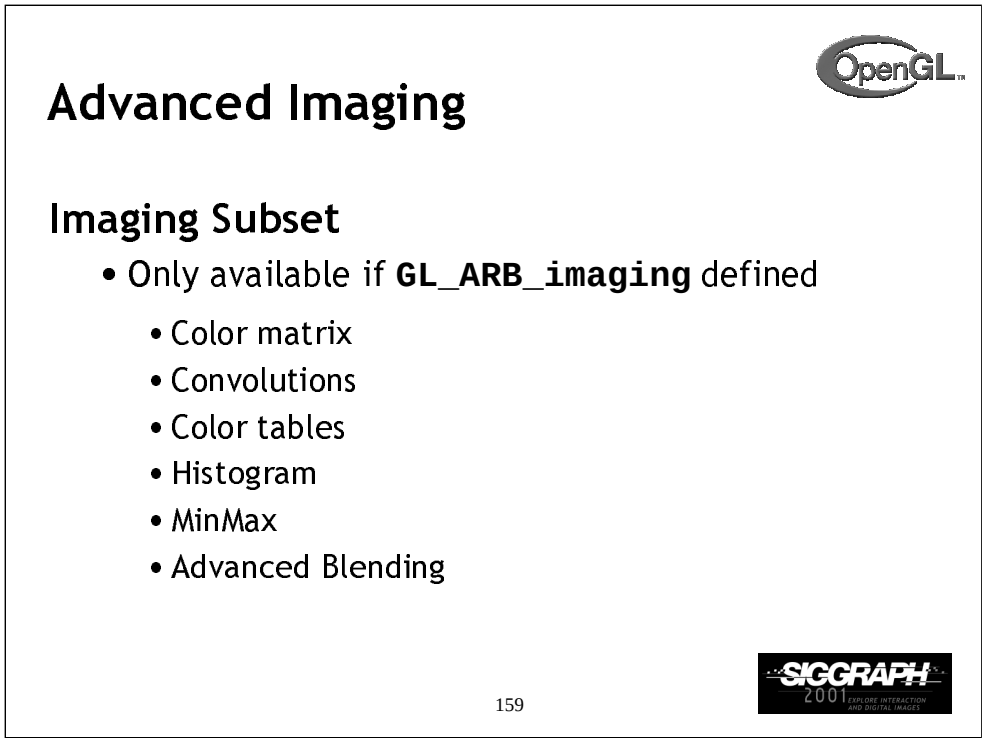
GL_XOR is a useful logical operation for creating "rubber banding" type techniques, where you only momentarily want to modify a pixel, and then return back to its original value.

There are several OpenGL logical operation modes:

```
GL_CLEAR          GL_SET     GL_COPY,
GL_COPY_INVERTED  GL_NOOP    GL_INVERT
GL_AND            GL_NAND    GL_OR
GL_NOR            GL_XOR     GL_AND_INVERTED
GL_AND_REVERSE    GL_EQUIV   GL_OR_REVERSE
GL_OR_INVERTED
```



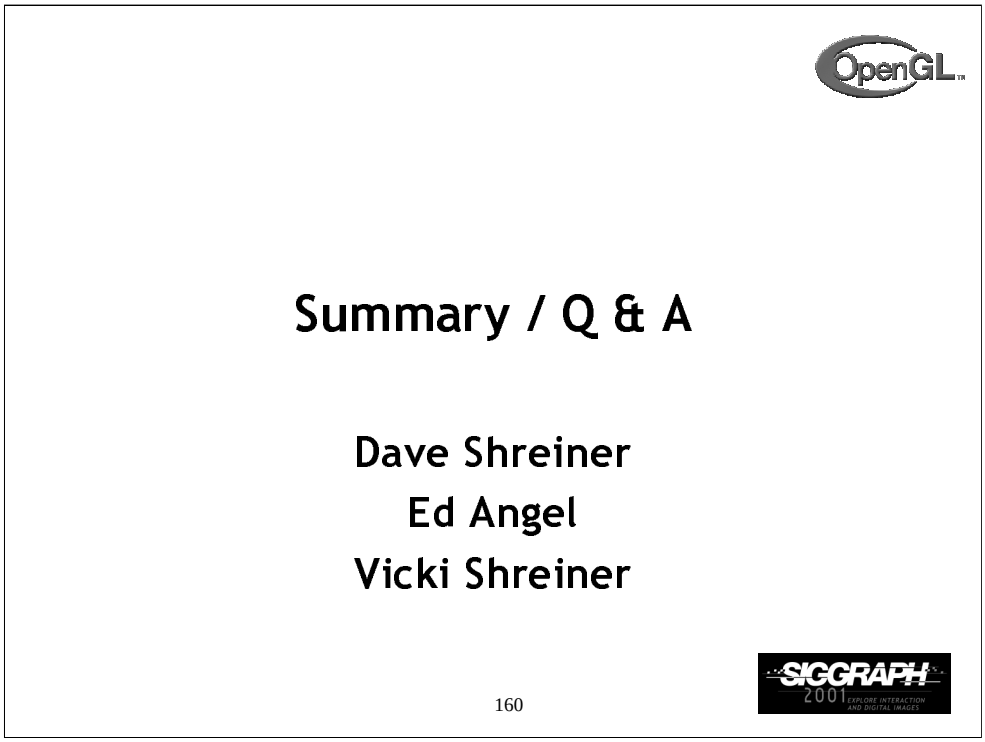
This tutorial provides an opportunity to experiment with OpenGL's pixel logical operations. The *source* and *destination* colors are combined using the logical operation that's set with the `glLogicOp()` function.



OpenGL may also contain an advanced set of functionality referred to as the *Imaging subset*. This functionality is only present if your implementation supports the `GL_ARB_imaging` extension.

Some of the functionality included in the imaging subset is:

- using a *color matrix* to apply linear transformation to color components
- computing image *convolutions*
- replacing colors using *color tables*
- *histogramming* and computing the minimum and maximum pixel values (*minmax*)
- advanced pixel blending modes


OpenGL
Summary / Q & A
Dave Shreiner
Ed Angel
Vicki Shreiner
160
SIGGRAPH
2001
EXPLORE INTERACTION AND DIGITAL IMAGES

On-Line Resources

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

Books

OpenGL Programming Guide, 3rd Edition

OpenGL Reference Manual, 3rd Edition

OpenGL Programming for the X Window System

- includes many GLUT examples

Interactive Computer Graphics: A top-down approach with OpenGL, 2nd Edition

162

Thanks for Coming

Questions and Answers

Dave Shreiner	**shreiner@sgi.com**
Ed Angel	**angel@cs.unm.edu**
Vicki Shreiner	**vshreiner@sgi.com**