An Introduction to OpenGL Programming

Ed Angel University of New Mexico
Dave Shreiner ARM, Inc.

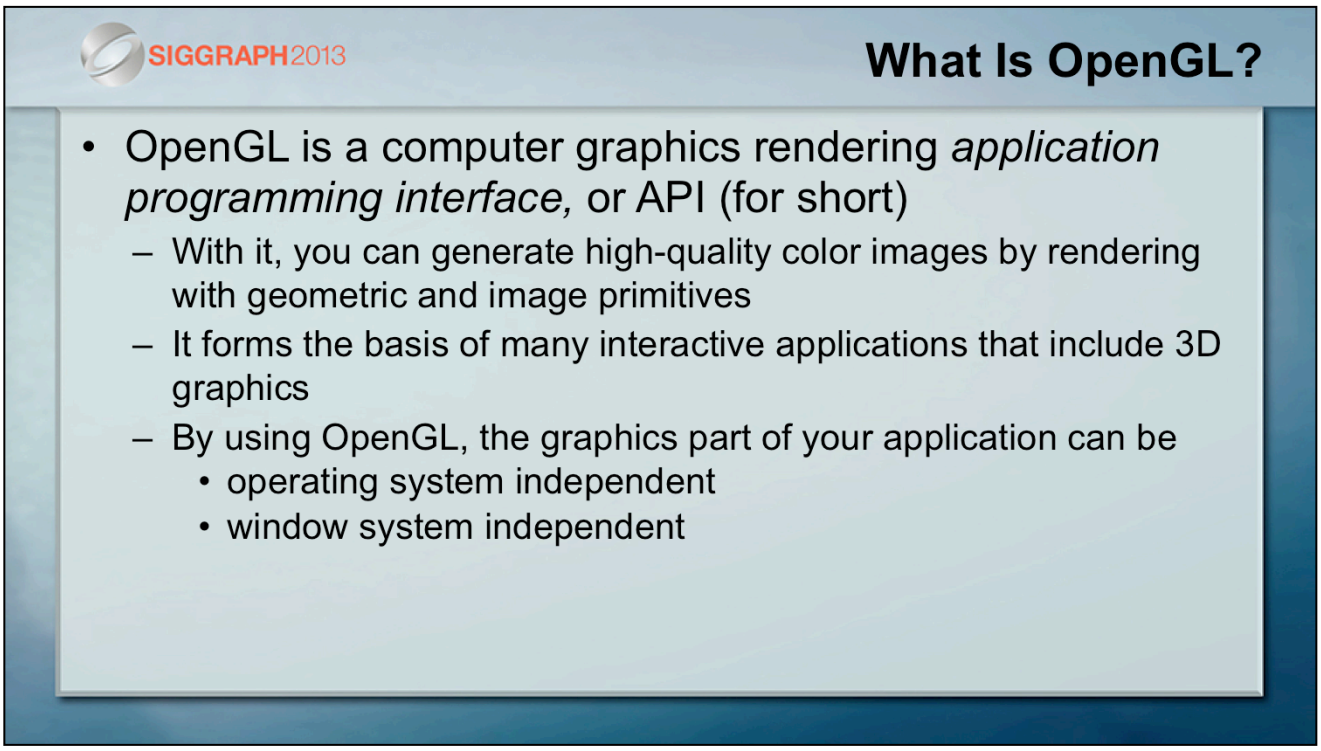
Presented at SIGGRAPH 2013
Sunday, July 21st, 2013
Anaheim, CA, USA

Contents

An Introduction to OpenGL Programming

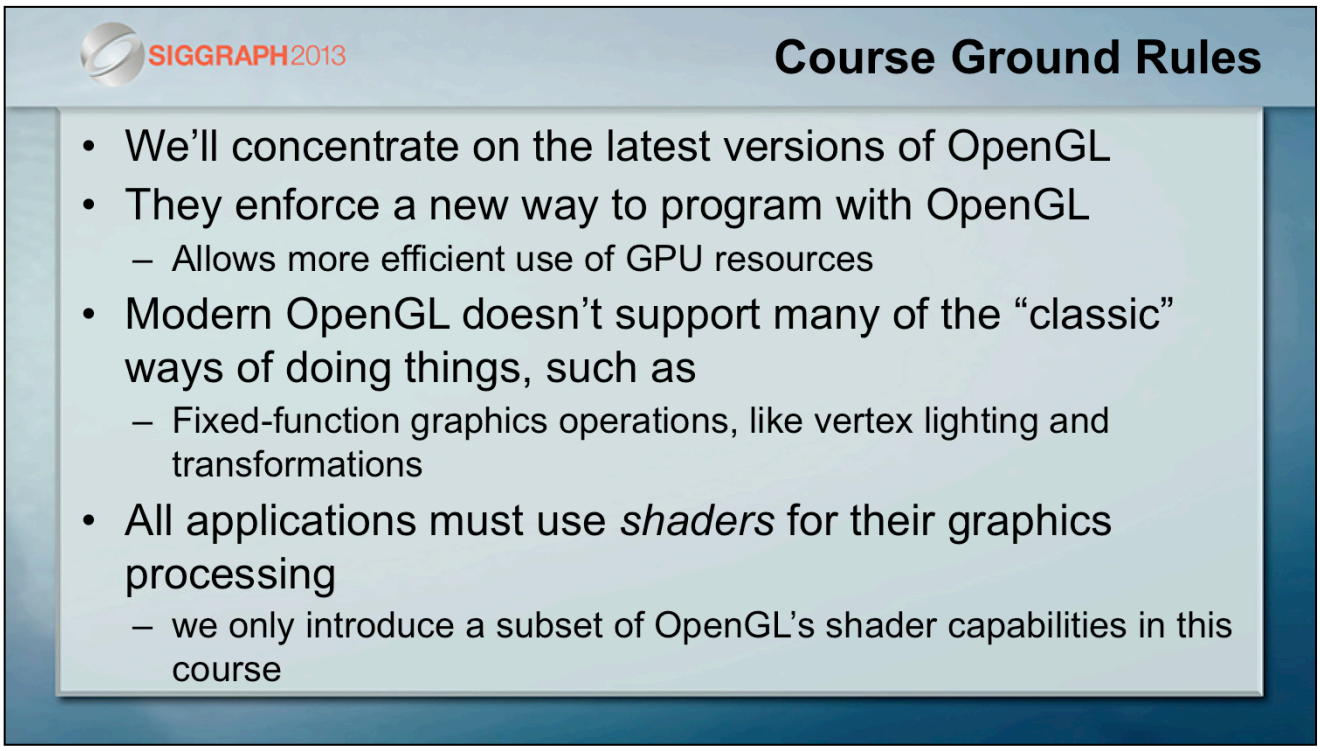
Ed Angel University of New Mexico
Dave Shreiner ARM, Inc.

SIGGRAPH 2013
The 40th International Conference and Exhibition on Computer Graphics and Interactive Techniques



OpenGL is a library of function calls for doing computer graphics. With it, you can create interactive applications that render high-quality color images composed of 2D and 3D geometric objects and images.

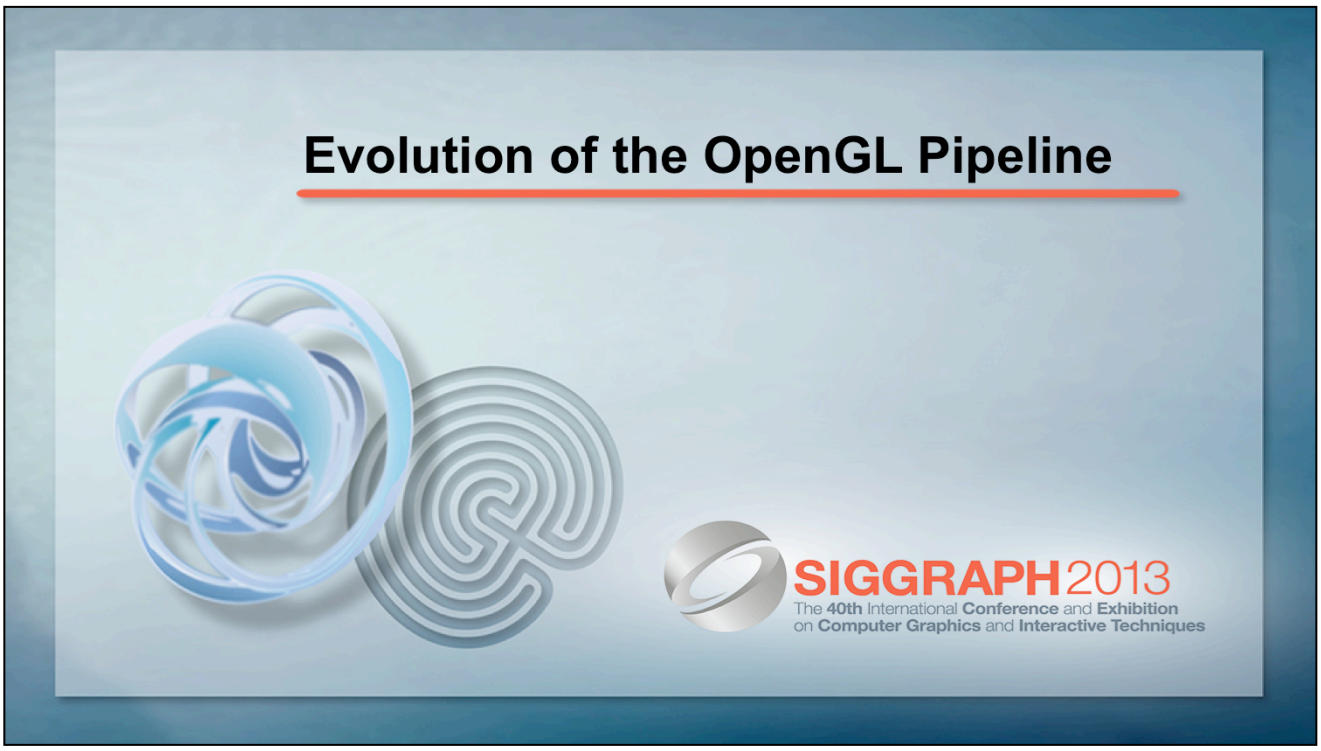
Additionally, the OpenGL API is independent of all operating systems, and their associated windowing systems. That means that the part of your application that draws can be platform independent. However, in order for OpenGL to be able to render, it needs a window to draw into. Generally, this is controlled by the windowing system on whatever platform you are working on.

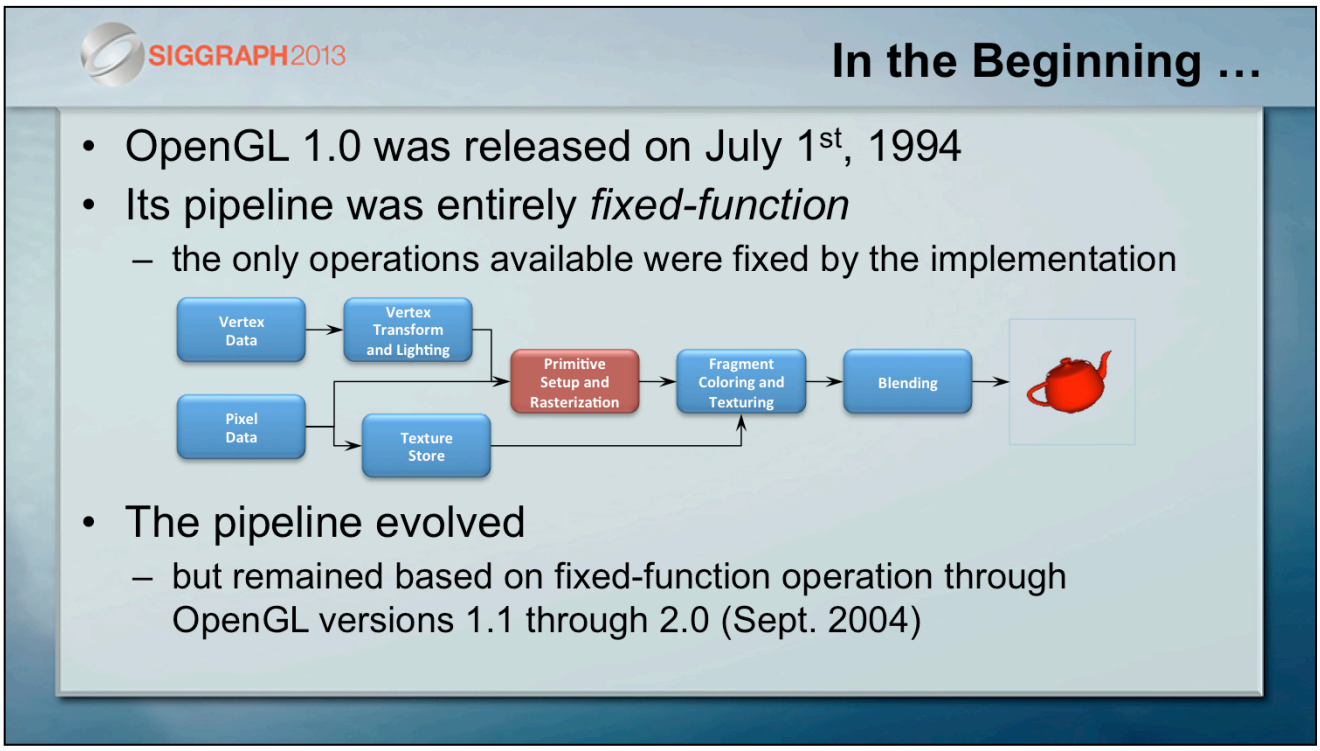


While OpenGL has been around for over 20 years, a lot of changes have occurred since it was created. This course concentrates on the latest versions of OpenGL – version 4.3, although we don't have time to discuss all the features available. In these modern versions of OpenGL (which we define as versions starting with version 3.1), OpenGL applications are entirely *shader* based. In fact, most of this course will discuss shaders and the operations they support.

If you're familiar with previous versions of OpenGL, or other *rasterization-based* graphics pipelines in general that may have included *fixed-function* processing, we won't be covering those techniques since these functions have been deprecated. Instead, we'll concentrate on showing how we can implement those techniques on a modern, shader-based graphics pipeline.

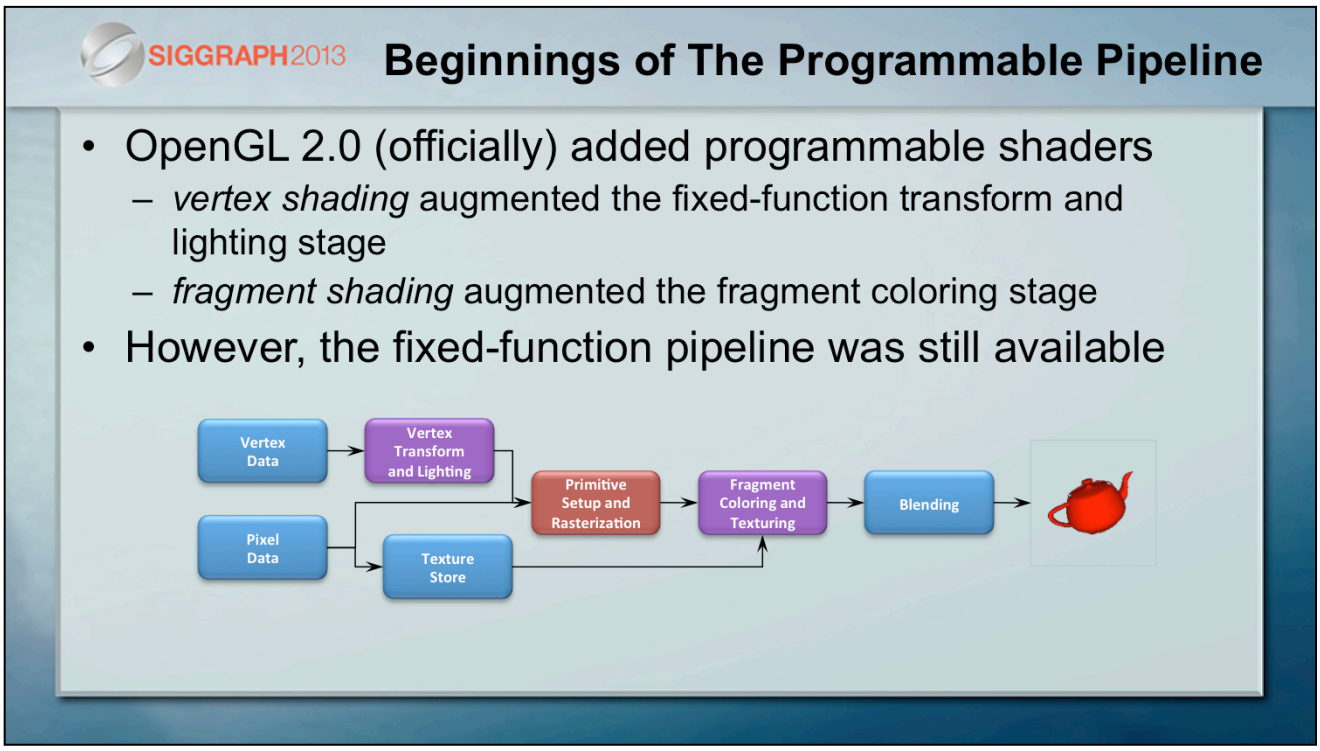
In this modern world of OpenGL, all applications will need to provide shaders, and as such, providing some perspective on how the pipeline evolved and its phases will be illustrative. We'll discuss this next.


Evolution of the OpenGL Pipeline
SIGGRAPH2013
The 40th International Conference and Exhibition
on Computer Graphics and Interactive Techniques



The initial version of OpenGL was announced in July of 1994. That version of OpenGL implemented what's called a *fixed-function pipeline*, which means that all of the operations that OpenGL supported were fully-defined, and an application could only modify their operation by changing a set of input values (like colors or positions). The other point of a fixed-function pipeline is that the order of operations was always the same – that is, you can't reorder the sequence operations occur.

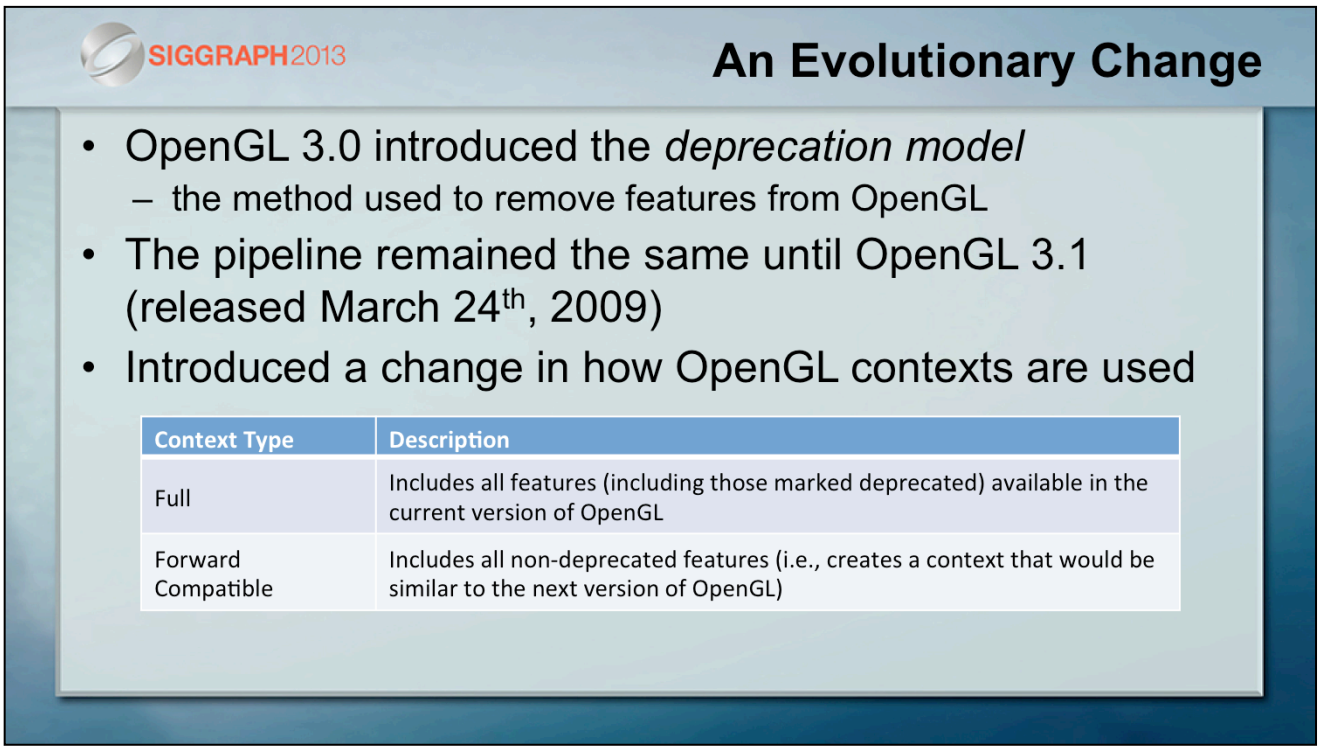
This pipeline was the basis of many versions of OpenGL and expanded in many ways, and is still available for use. However, modern GPUs and their features have diverged from this pipeline, and support of these previous versions of OpenGL are for supporting current applications. If you're developing a new application, we strongly recommend using the techniques that we'll discuss. Those techniques can be more flexible, and will likely preform better than using one of these early versions of OpenGL since they can take advantage of the capabilities of recent Graphics Processing Units (GPUs).



While many features and improvements were added into the fixed-function OpenGL pipeline, designs of GPUs were exposing more features than could be added into OpenGL. To allow applications to gain access to these new GPU features, OpenGL version 2.0 officially added *programmable shaders* into the graphics pipeline. This version of the pipeline allowed an application to create small programs, called *shaders*, that were responsible for implementing the features required by the application. In the 2.0 version of the pipeline, two programmable stages were made available:

- *vertex shading* enabled the application full control over manipulation of the 3D geometry provided by the application
- *fragment shading* provided the application capabilities for *shading* pixels (the terms classically used for determining a pixel's color).

OpenGL 2.0 also fully supported OpenGL 1.X's pipeline, allowing the application to use both version of the pipeline: fixed-function, and *programmable*.

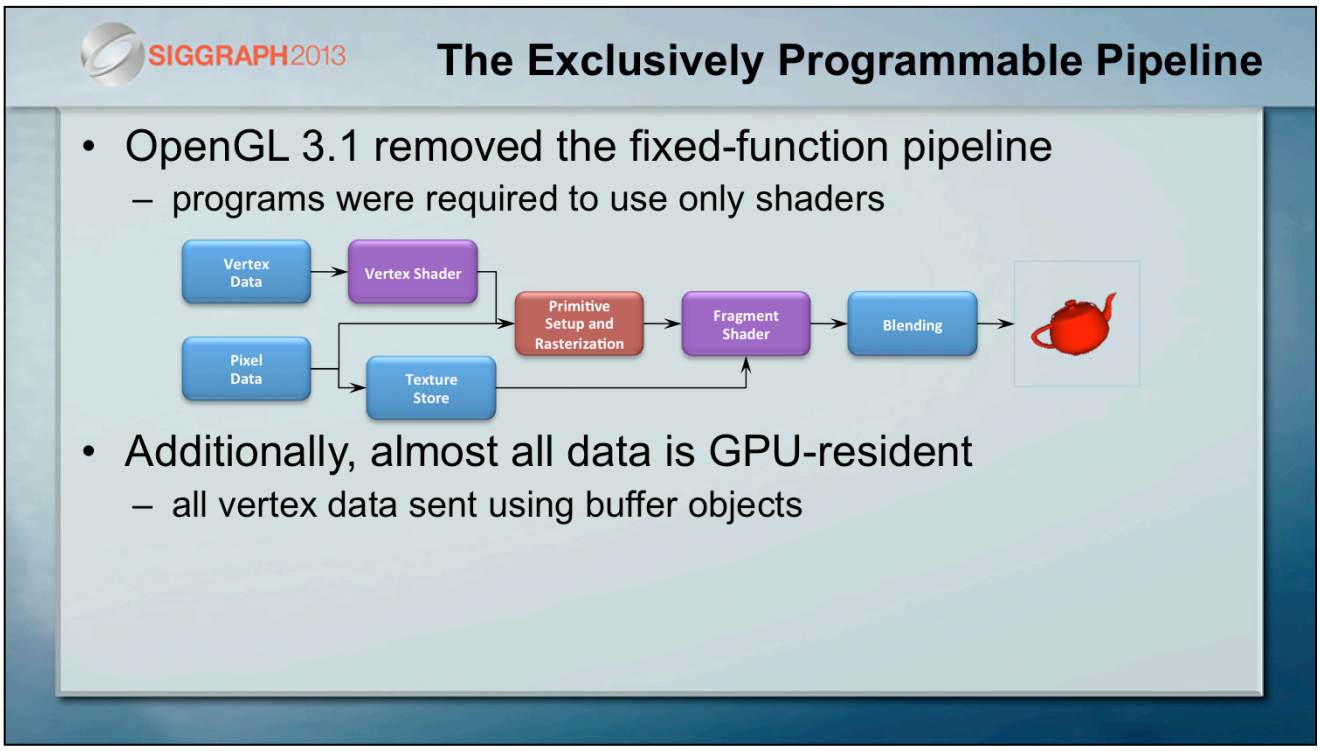


Until OpenGL 3.0, features have only been added (but never removed) from OpenGL, providing a lot of application backwards compatibility (up to the use of extensions). OpenGL version 3.0 introduced the mechanisms for removing features from OpenGL, called the *deprecation model.* It defines how the OpenGL design committee (the OpenGL Architecture Review Board (ARB) of the Khronos Group) will advertise of which and how functionality is removed from OpenGL.

You might ask: why remove features from OpenGL? Over the 15 years to OpenGL 3.0, GPU features and capabilities expanded and some of the methods used in older versions of OpenGL were not as efficient as modern methods. While removing them could break support for older applications, it also simplified and optimized the GPUs allowing better performance.

Within an OpenGL application, OpenGL uses an opaque data structure called a *context*, which OpenGL uses to store shaders and other data. Contexts come in two flavors:

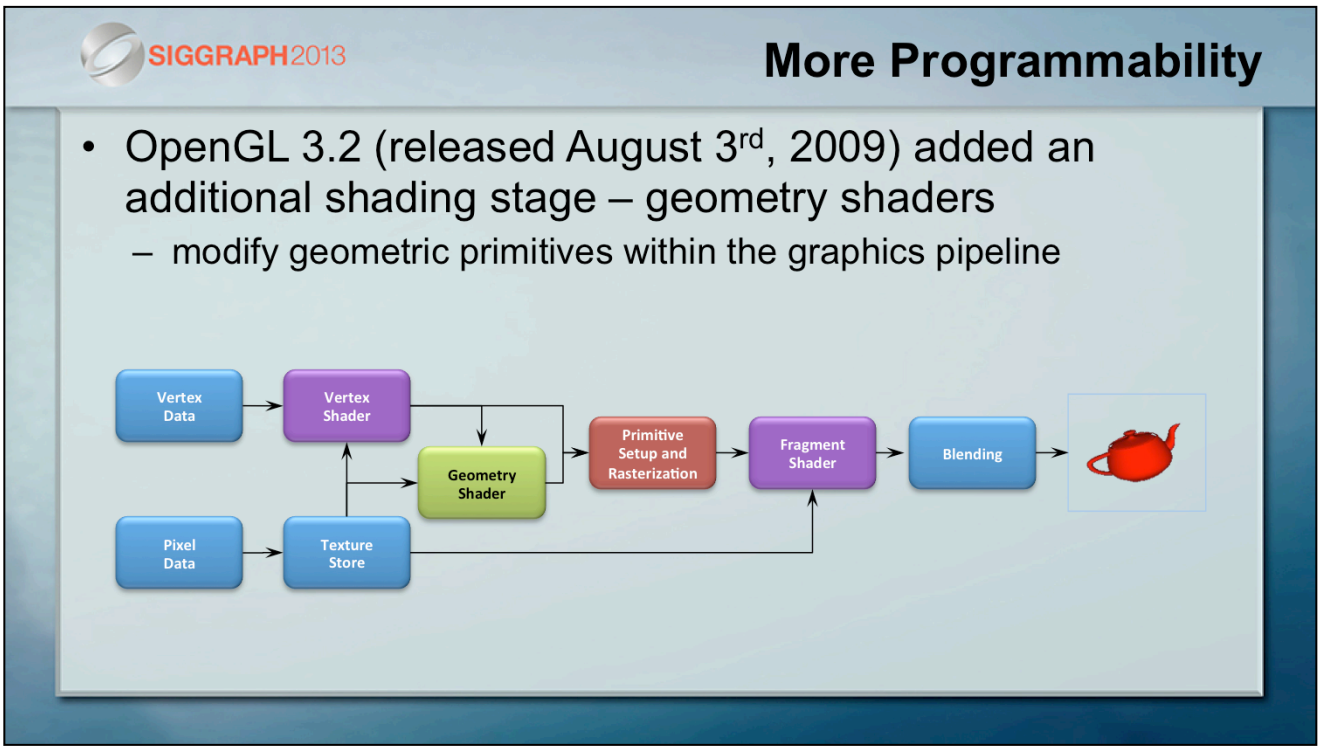
- *full* contexts expose all the features of the current version of OpenGL, including features that are marked deprecated.
- *forward-compatible* contexts enable only the features that will be available in the next version of OpenGL (i.e., deprecated features pretend to be



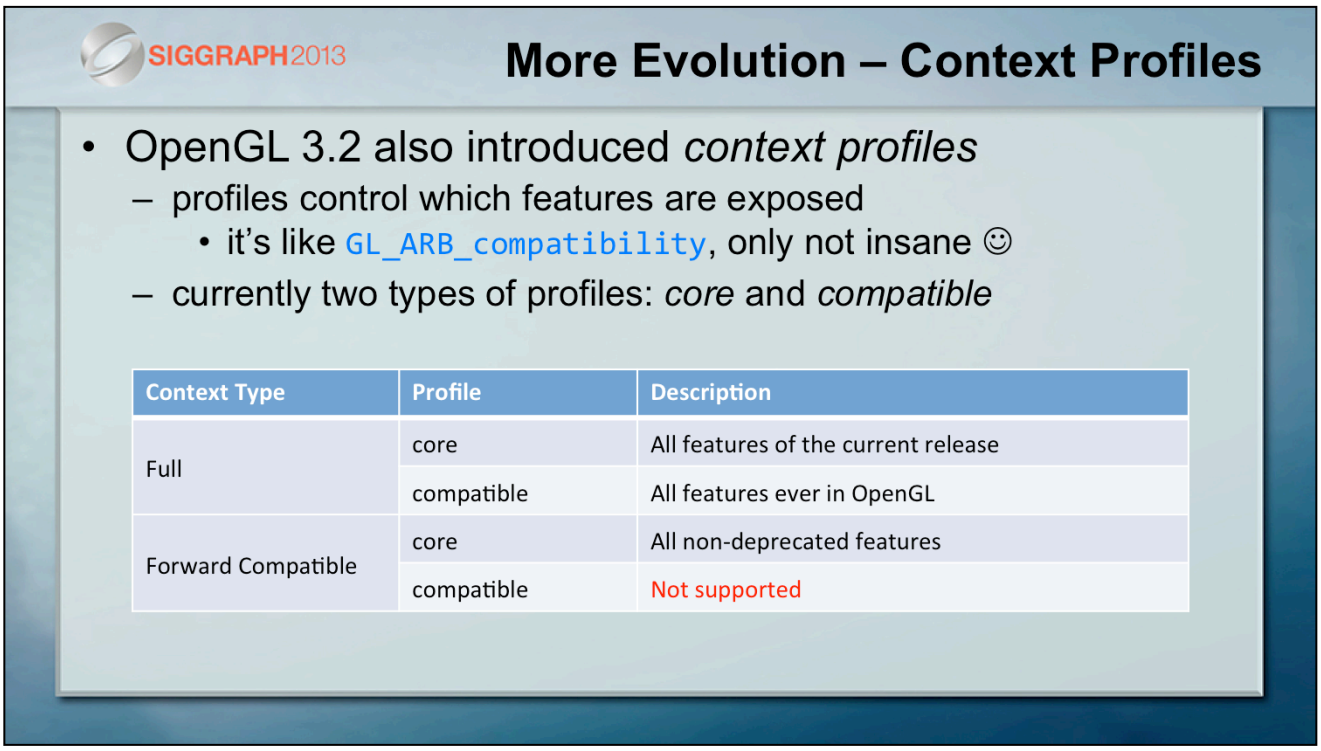
OpenGL version 3.1 was the first version to remove deprecated features, and break backwards compatibility with previous versions of OpenGL. The features removed from included the old-style fixed-function pipeline, among other lesser features.

One major refinement introduced in 3.1 was requiring all data to be placed in GPU-resident *buffer objects*, which help reduce the impacts of various computer system architecture limitations related to GPUs.

While many features were removed from OpenGL 3.1, the OpenGL ARB realized that to make it easy for application developers to transition their products, they introduced an OpenGL extensions, `GL_ARB_compatibility`, that allowed access to the removed features.

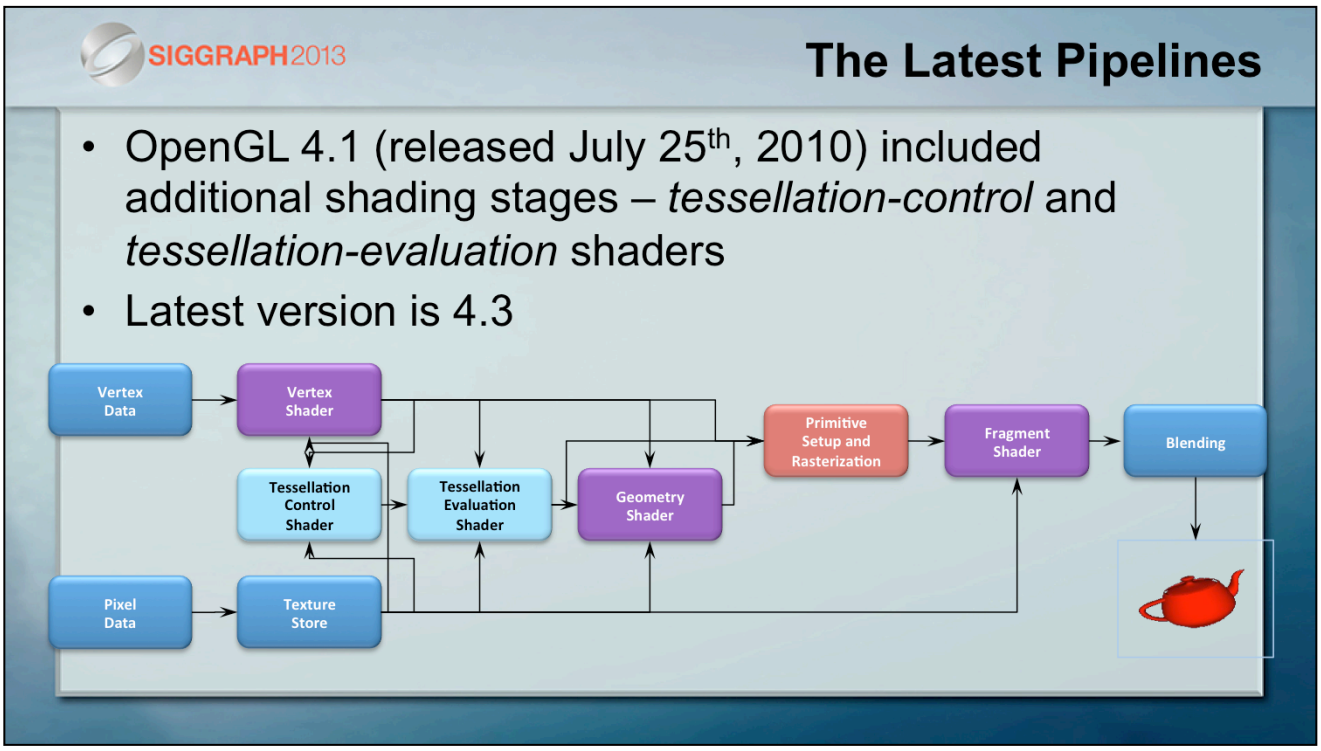


Until OpenGL 3.2, the number of *shader stages* in the OpenGL pipeline remained the same, with only vertex and fragment shaders being supported. OpenGL version 3.2 added a new shader stage called *geometry shading* which allows the modification (and generation) of geometry within the OpenGL pipeline.



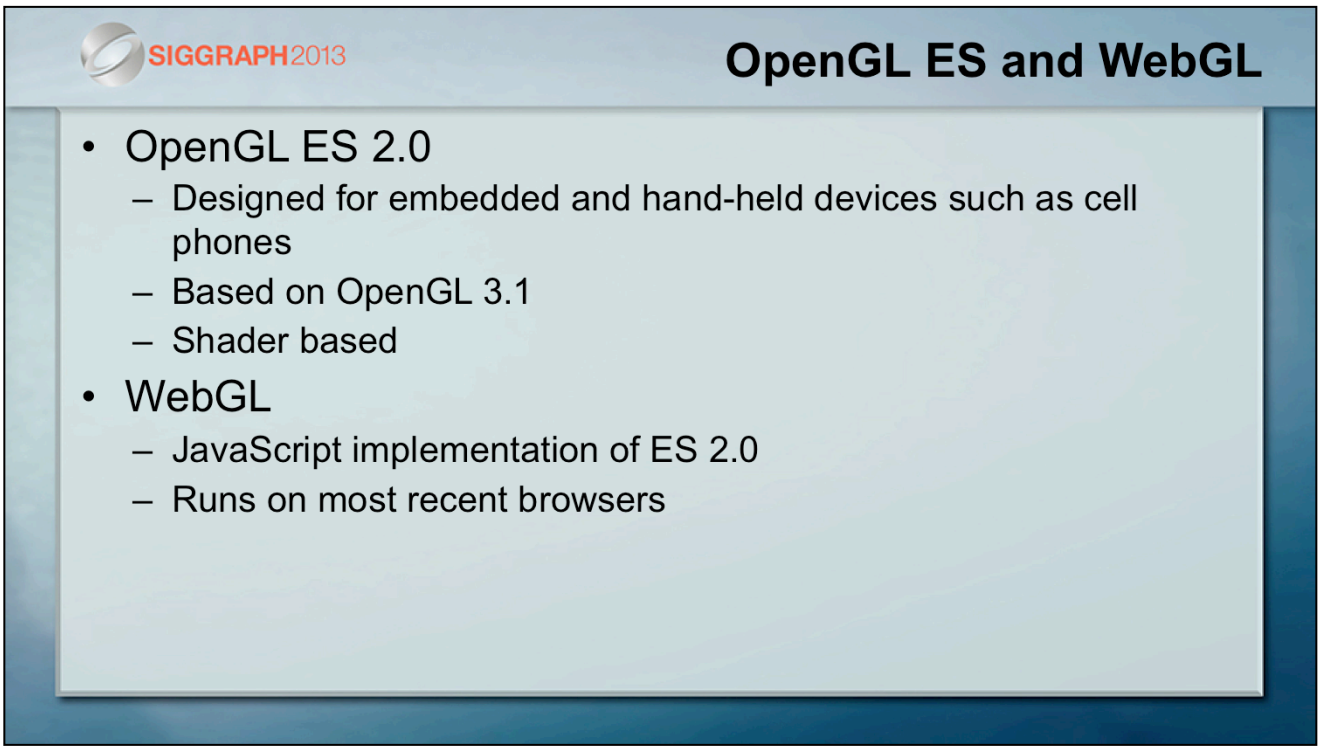
In order to make it easier for developers to choose the set of features they want to use in their application, OpenGL 3.2 also introduced *profiles* which allow further selection of OpenGL contexts.

The *core* profile is the modern, trimmed-down version of OpenGL that includes the latest features. You can request a core profile for a Full or Forward-compatible profile. Conversely, you could request a *compatible* profile, which includes all functionality (supported by the OpenGL driver on your system) in all versions of OpenGL up to, and including, the version you've requested.



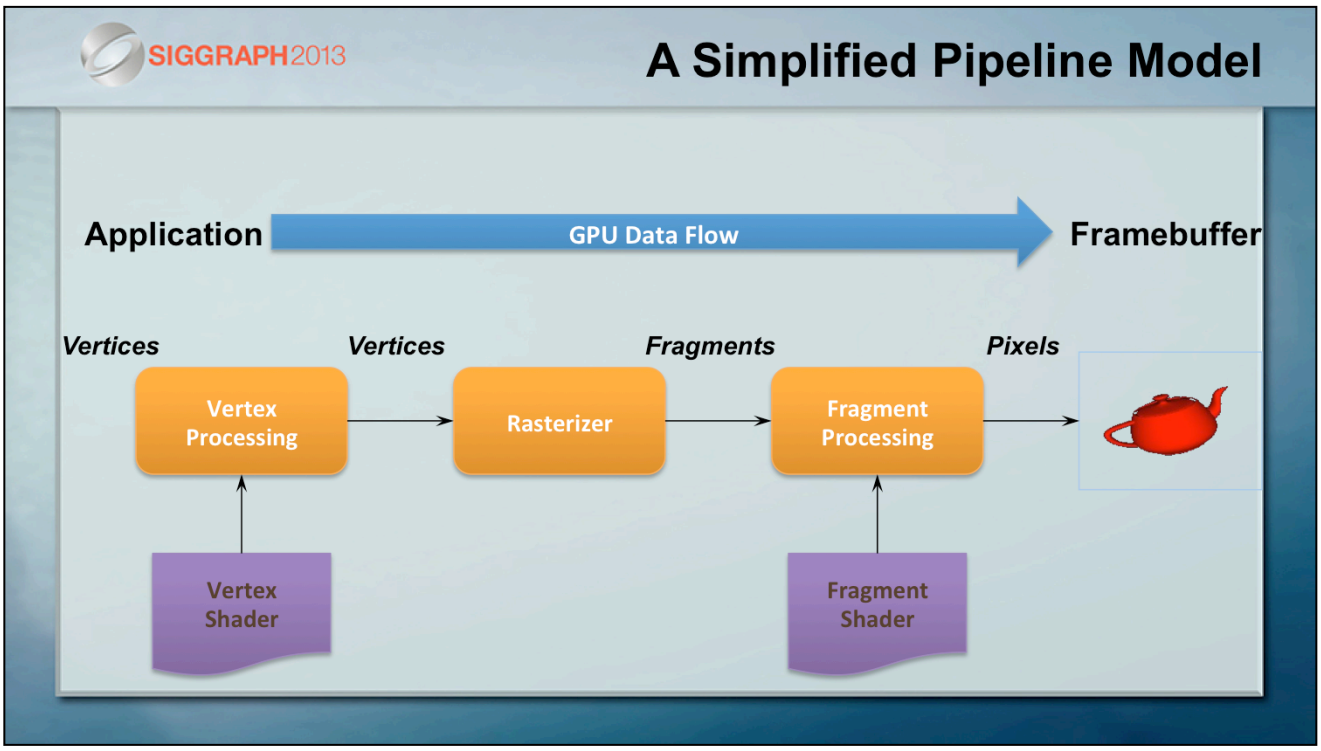
The OpenGL 4.X pipeline added another pair of shaders (which work in tandem, so we consider it a single stage) for supporting dynamic tessellation in the GPU. *Tessellation control* and *tessellation evaluation* shaders were added to OpenGL version 4.0.

The current version of OpenGL is 4.3, which includes some additional features over the 4.0 pipeline, but no new shading stages.



WebGL is becoming increasingly more important because it is supported by all browsers except Internet Explorer (and even that appears to be changing). Besides the advantage of being able to run without recompilation across platforms, it can easily be integrated with other Web applications and make use of a variety of portable packages available over the Web.


OpenGL Application Development
SIGGRAPH2013
The 40th International Conference and Exhibition
on Computer Graphics and Interactive Techniques



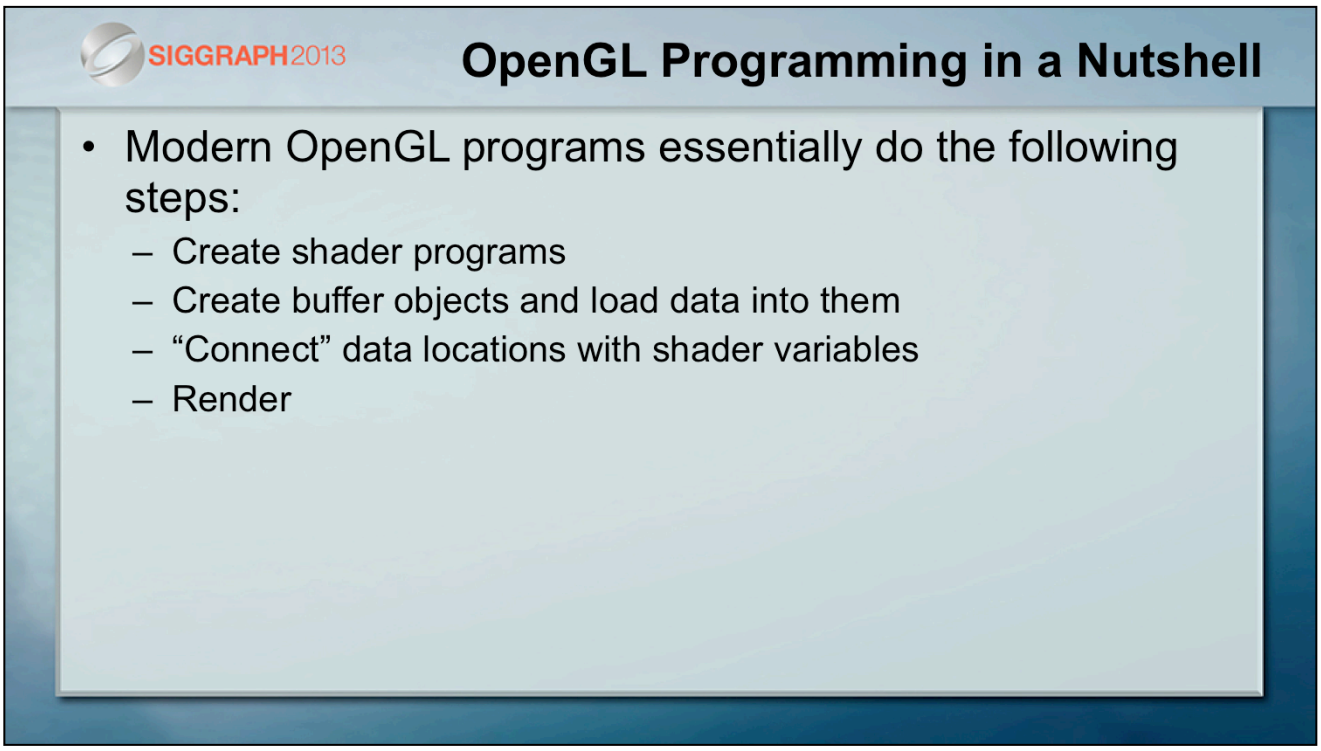
To begin, let us introduce a simplified model of the OpenGL pipeline. Generally speaking, data flows from your application through the GPU to generate an image in the *frame buffer*. Your application will provide *vertices*, which are collections of data that are composed to form geometric objects, to the OpenGL pipeline. The *vertex processing* stage uses a vertex shader to process each vertex, doing any computations necessary to determine where in the frame buffer each piece of geometry should go. The other shading stages we mentioned – tessellation and geometry shading – are also used for vertex processing, but we're trying to keep this simple.

After all the vertices for a piece of geometry are processed, the *rasterizer* determines which pixels in the frame buffer are affected by the geometry, and for each pixel, the *fragment processing* stage is employed, where the *fragment shader* runs to determine the final color of the pixel.

In your OpenGL applications, you'll usually need to do the following tasks:

- specify the vertices for your geometry
- load vertex and fragment shaders (and other shaders, if you're using them as well)
- issue your geometry to engage the OpenGL pipeline for processing

Of course, OpenGL is capable of many other operations as well, many of

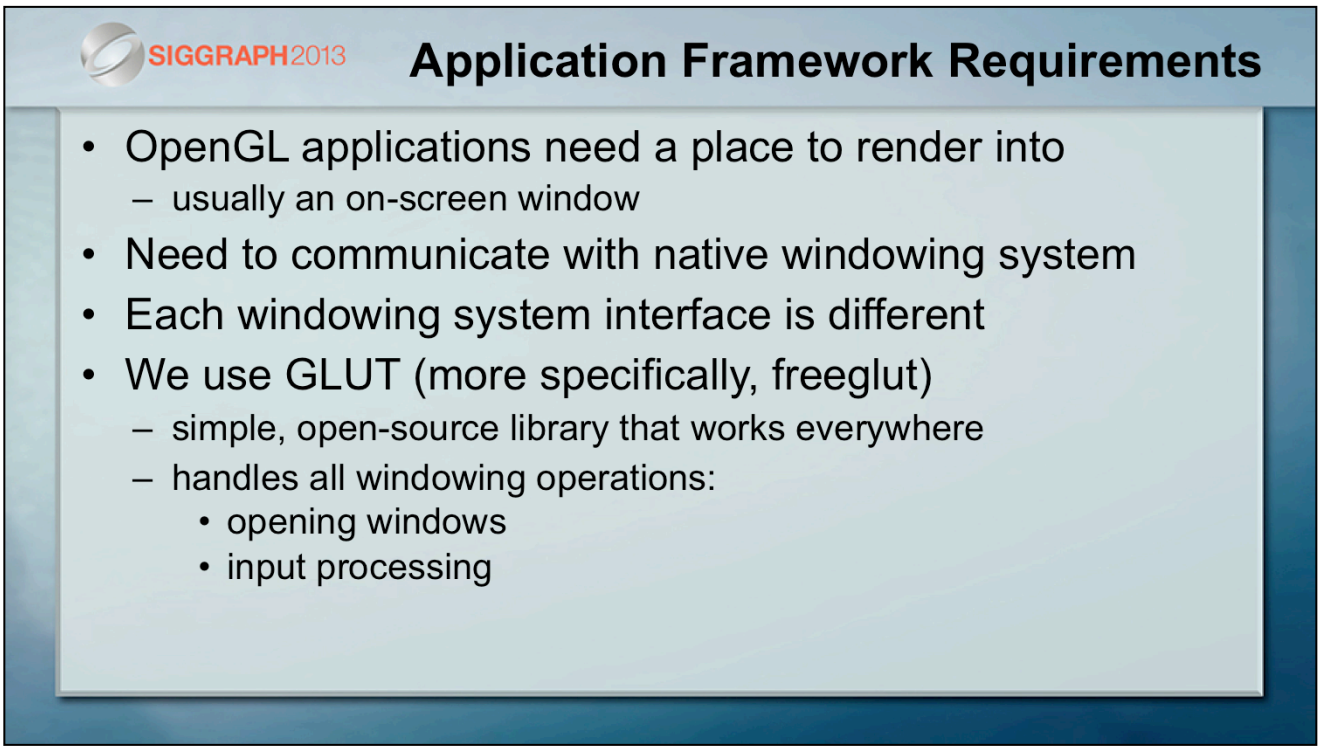


You'll find that a few techniques for programming with modern OpenGL goes a long way. In fact, most programs – in terms of OpenGL activity – are very repetitive. Differences usually occur in how objects are rendered, and that's mostly handled in your shaders.

There four steps you'll use for rendering a geometric object are as follows:

1. First, you'll load and create OpenGL *shader programs* from shader source programs you create
2. Next, you will need to load the data for your objects into OpenGL's memory. You do this by creating *buffer objects* and loading data into them.
3. Continuing, OpenGL needs to be told how to interpret the data in your buffer objects and associate that data with variables that you'll use in your shaders. We call this *shader plumbing*.
4. Finally, with your data initialized and shaders set up, you'll render your objects

We'll expand on those steps more through the course, but you'll find that most applications will merely iterate through those steps.

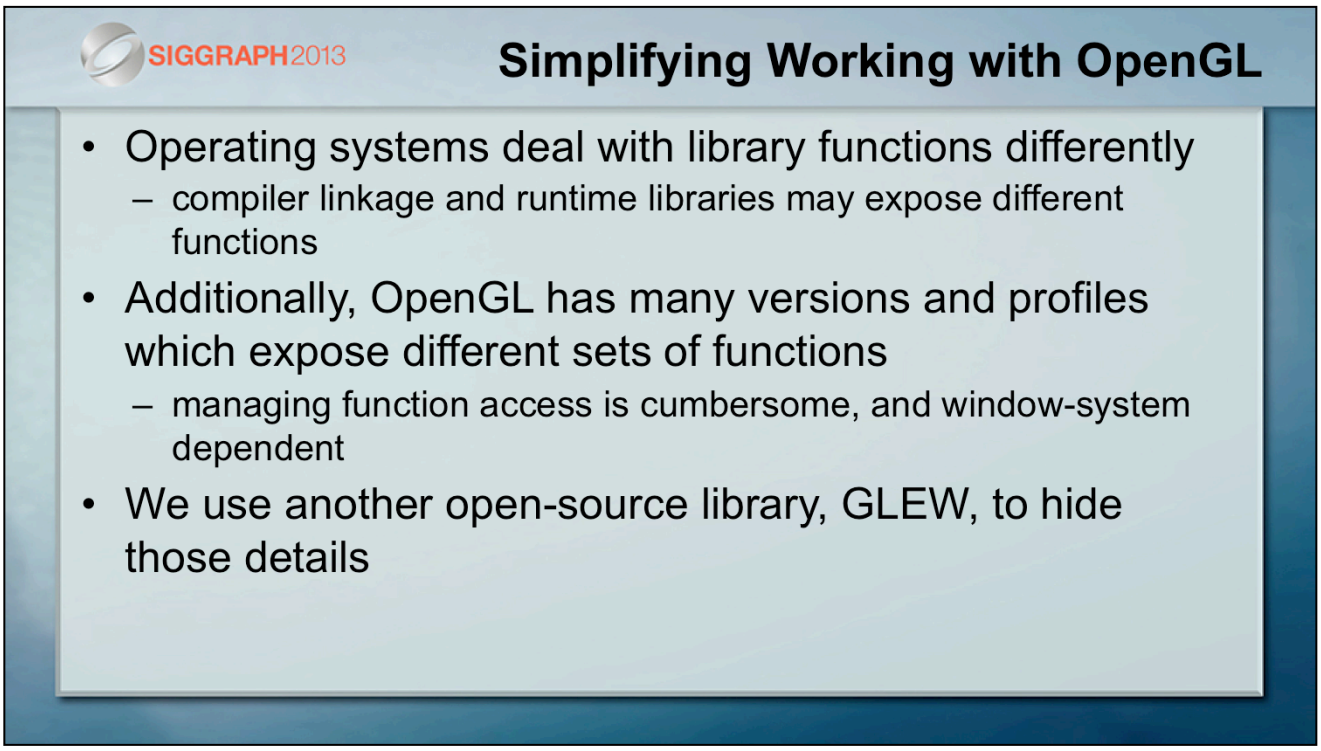


While OpenGL will take care of filling the pixels in your application's output window or image, it has no mechanisms for creating that *rendering surface*. Instead, OpenGL relies on the native windowing system of your operating system to create a window, and make it available for OpenGL to render into. For each windowing system (like Microsoft Windows, or the X Window System on Linux [and other Unixes]), there's a *binding library* that lets mediates between OpenGL and the native windowing system.

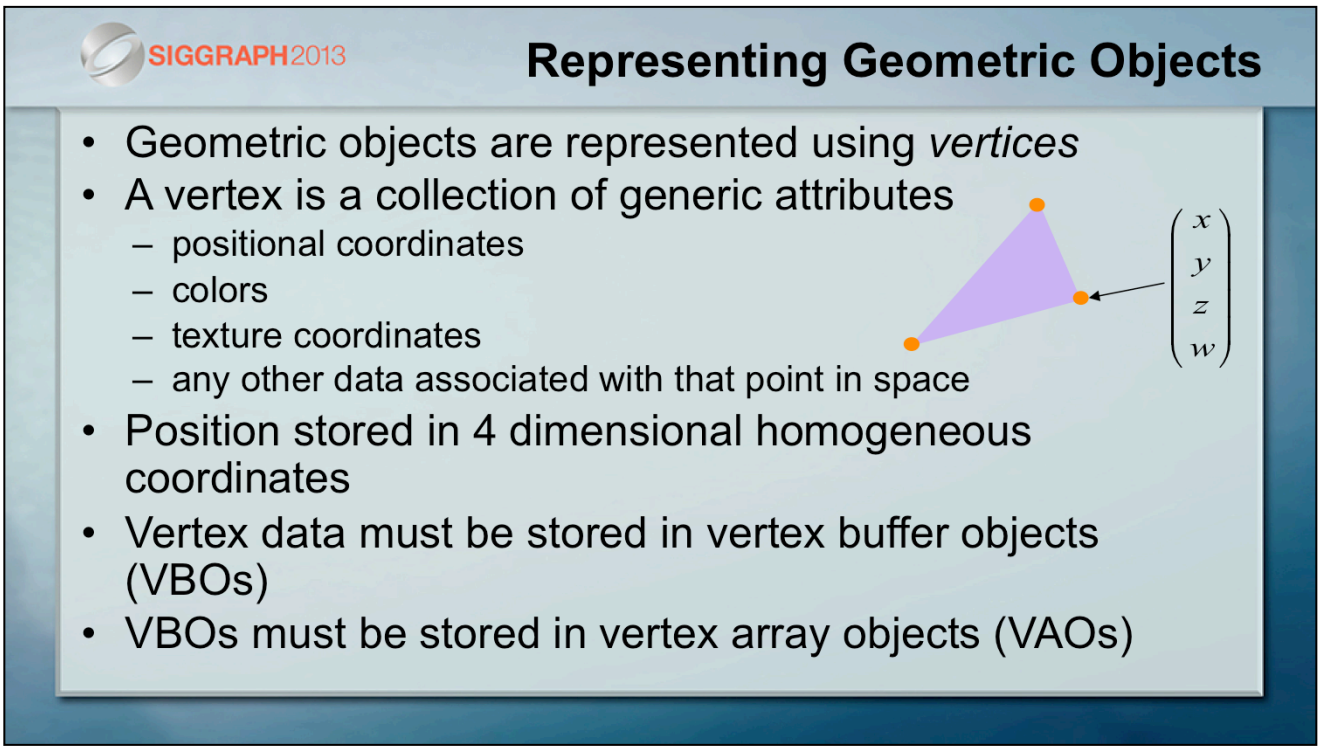
Since each windowing system has different semantics for creating windows and binding OpenGL to them, discussing each one is outside of the scope of this course. Instead, we use an open-source library named **freeglut** that abstracts each windowing system's specifics into a simple library. freeglut is a derivative of an older implementation called GLUT, and we'll use those names interchangeably. GLUT will help us in creating windows, dealing with user input and input devices, and other window-system activities.

You can find out more about freeglut at its website:
`http://freeglut.sourceforge.net`

Both GLUT and freeglut use deprecated functions and should not work with a core profile. One alternative is GLFW which runs on Windows, Linux and Mac OS X.



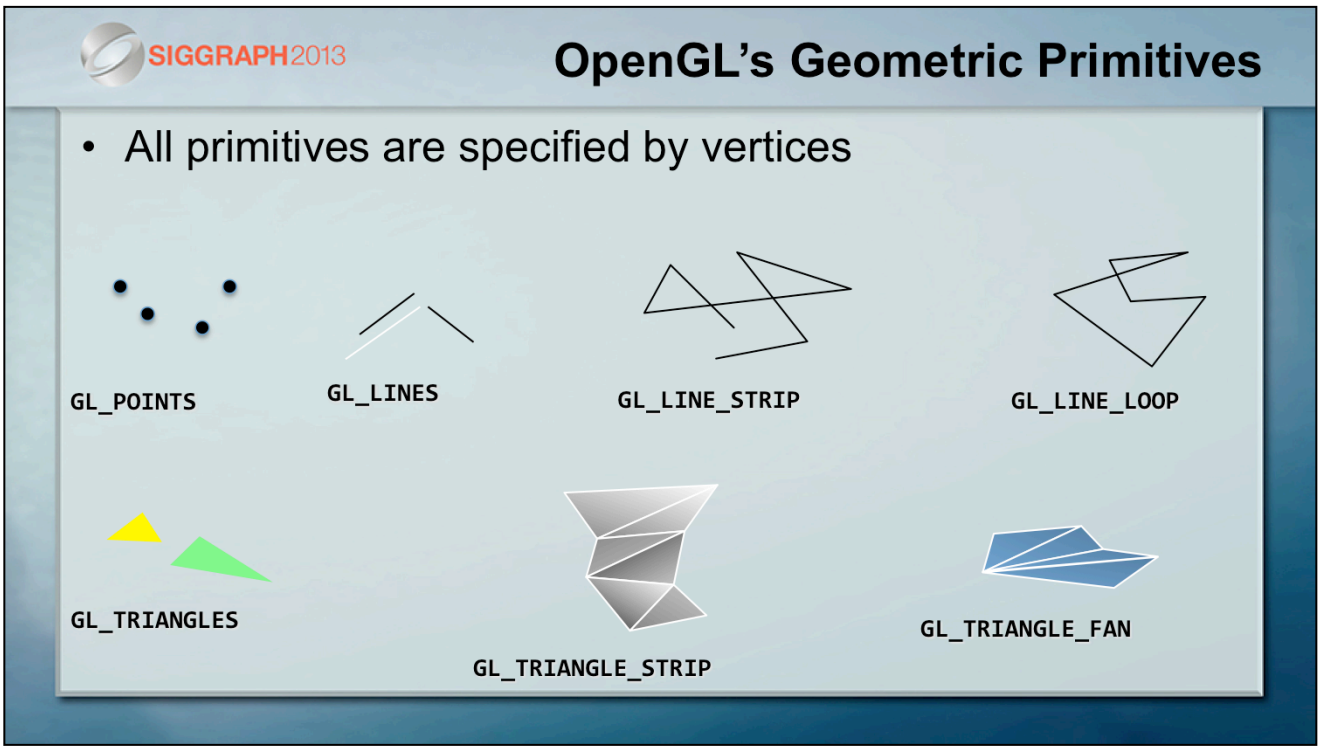
Just like window systems, operating systems have different ways of working with libraries. In some cases, the library you link your application exposes different functions than the library you execute your program with. Microsoft Windows is a notable example where you compile your application with a `.lib` library, but use a `.dll` at runtime for finding function definitions. As such, your application would generally need to use operating-system specific methods to access functions. In general, this is troublesome and a lot of work. Fortunately, another open-source library comes to our aid, GLEW, the OpenGL Extension Wrangler library. It removes all the complexity of accessing OpenGL functions, and working with OpenGL extensions. We use GLEW in our examples to simplify the code. You can find details about GLEW at its website: `http://glew.sourceforge.net`



In OpenGL, as in other graphics libraries, objects in the scene are composed of *geometric primitives*, which themselves are described by *vertices*. A vertex in modern OpenGL is a collection of data values associated with a location in space. Those data values might include colors, reflection information for lighting, or additional coordinates for use in texture mapping. Locations can be specified on 2, 3 or 4 dimensions but are stored in 4 dimensional *homogeneous coordinates*.

Vertices must be organized in OpenGL server-side objects called *vertex buffer objects* (also known as *VBOs*), which need to contain all of the vertex information for all of the primitives that you want to draw at one time. VBOs can store vertex information in almost any format (i.e., an array-of-structures (AoS) each containing a single vertex's information, or a structure-of-arrays (SoA) where all of the same "type" of data for a vertex is stored in a contiguous array, and the structure stores arrays for each attribute that a vertex can have). The data within a VBO needs to be contiguous in memory, but doesn't need to be tightly packed (i.e., data elements may be separated by any number of bytes, as long as the number of bytes between attributes is consistent).

VBOs are further required to be stored in *vertex array objects* (known as *VAOs*). Since it may be the case that numerous VBOs are associated with a single object, VAOs simplify the management of the collection of VBOs.

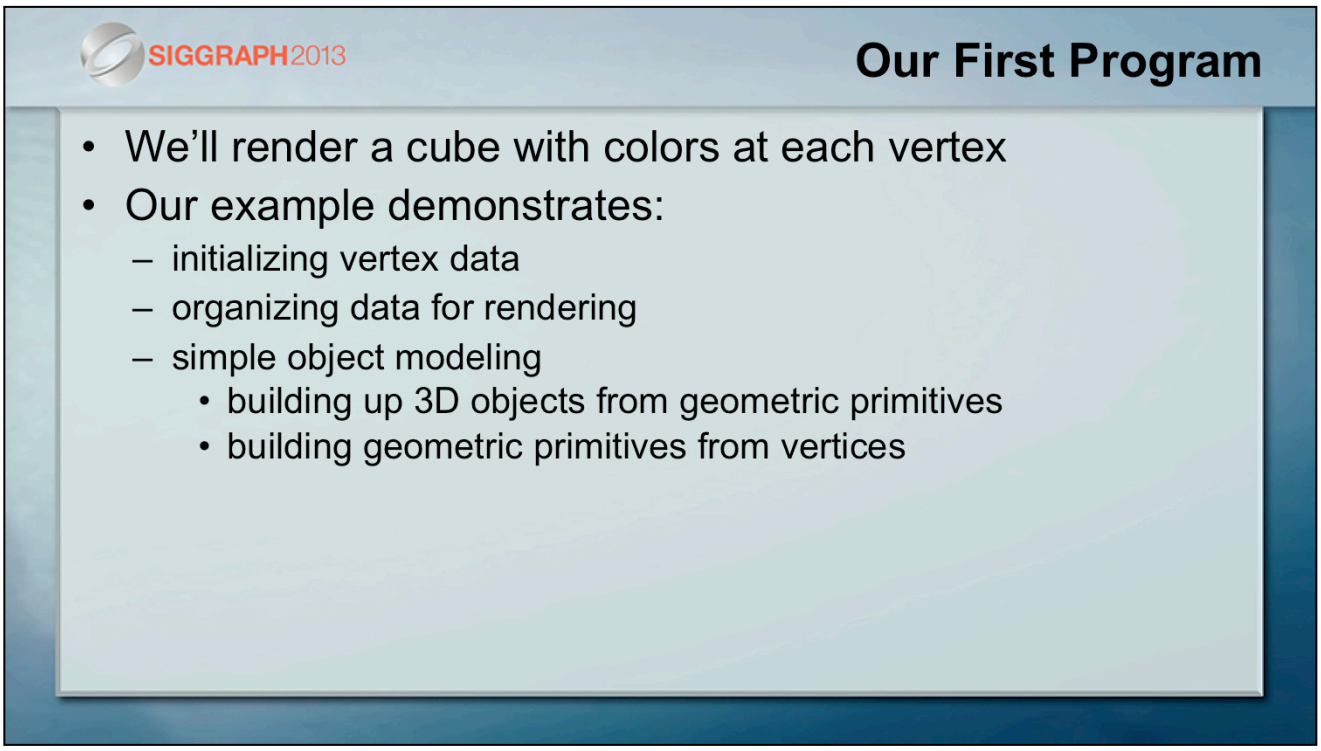


To form 3D geometric objects, you need to decompose them into geometric primitives that OpenGL can draw. OpenGL only knows how to draw three things: points, lines, and triangles, but can use collections of the same type of primitive to optimize rendering.

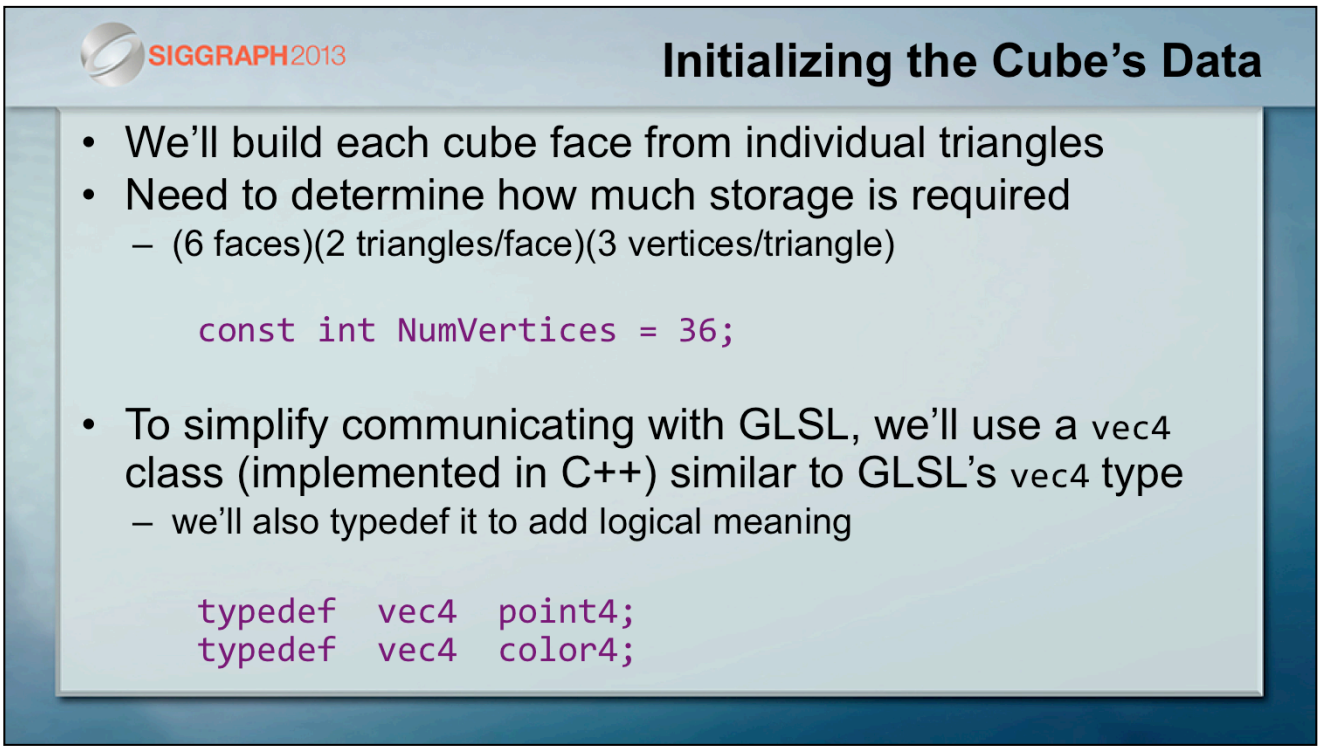
OpenGL Primitive	Description	Total Vertices for *n* Primitives
GL_POINTS	Render a single point per vertex (points may be larger than a single pixel)	n
GL_LINES	Connect each pair of vertices with a single line segment.	2n
GL_LINE_STRIP	Connect each successive vertex to the previous one with a line segment.	n+1
GL_LINE_LOOP	Connect all vertices in a loop of line segments.	n
GL_TRIANGLES	Render a triangle for each triple of vertices.	3n
GL_TRIANGLE_STRIP	Render a triangle from the first three vertices in the list, and then create a new triangle with the last two rendered vertices, and the new vertex.	n+2
GL_TRIANGLE_FAN	Create triangles by using the first vertex in the list, and pairs of successive vertices.	n+2

A First Program

SIGGRAPH 2013
The 40th International Conference and Exhibition
on Computer Graphics and Interactive Techniques



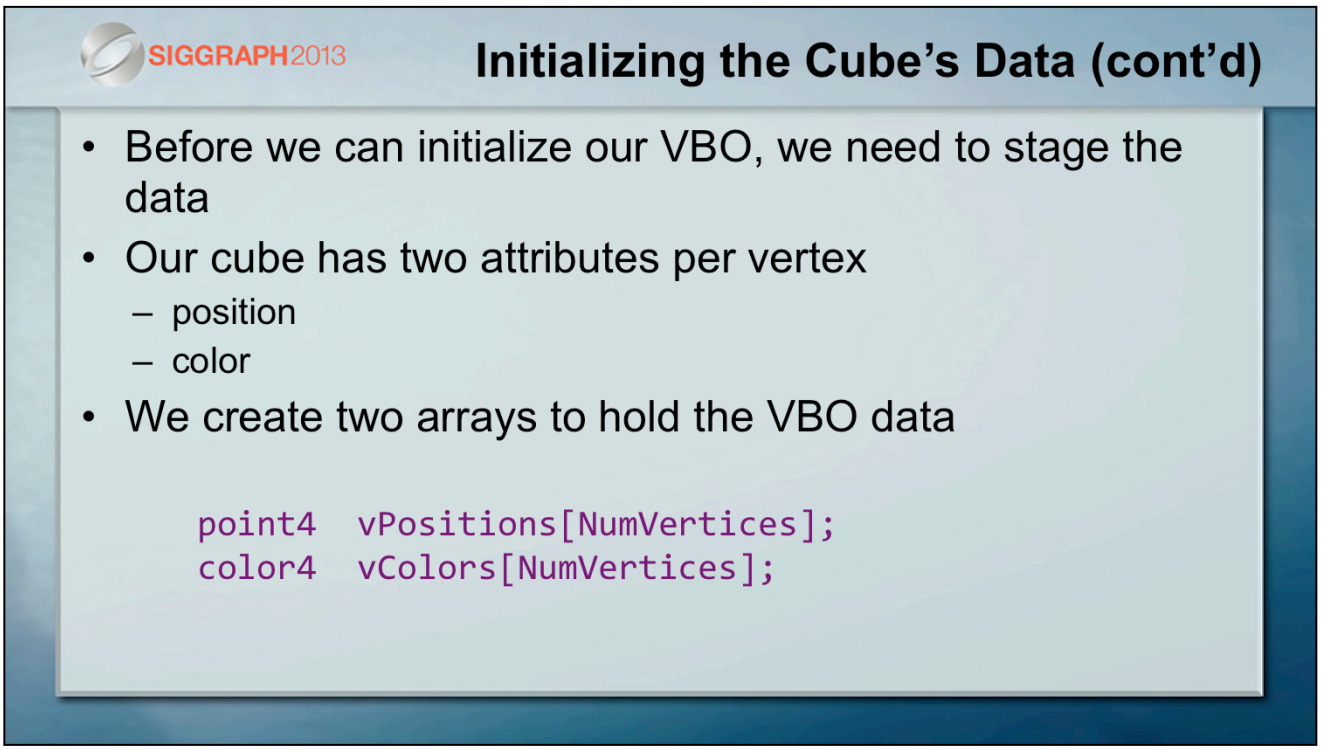
The next few slides will introduce our first example program, one which simply displays a cube with different colors at each vertex. We aim for simplicity in this example, focusing on the OpenGL techniques, and not on optimal performance.



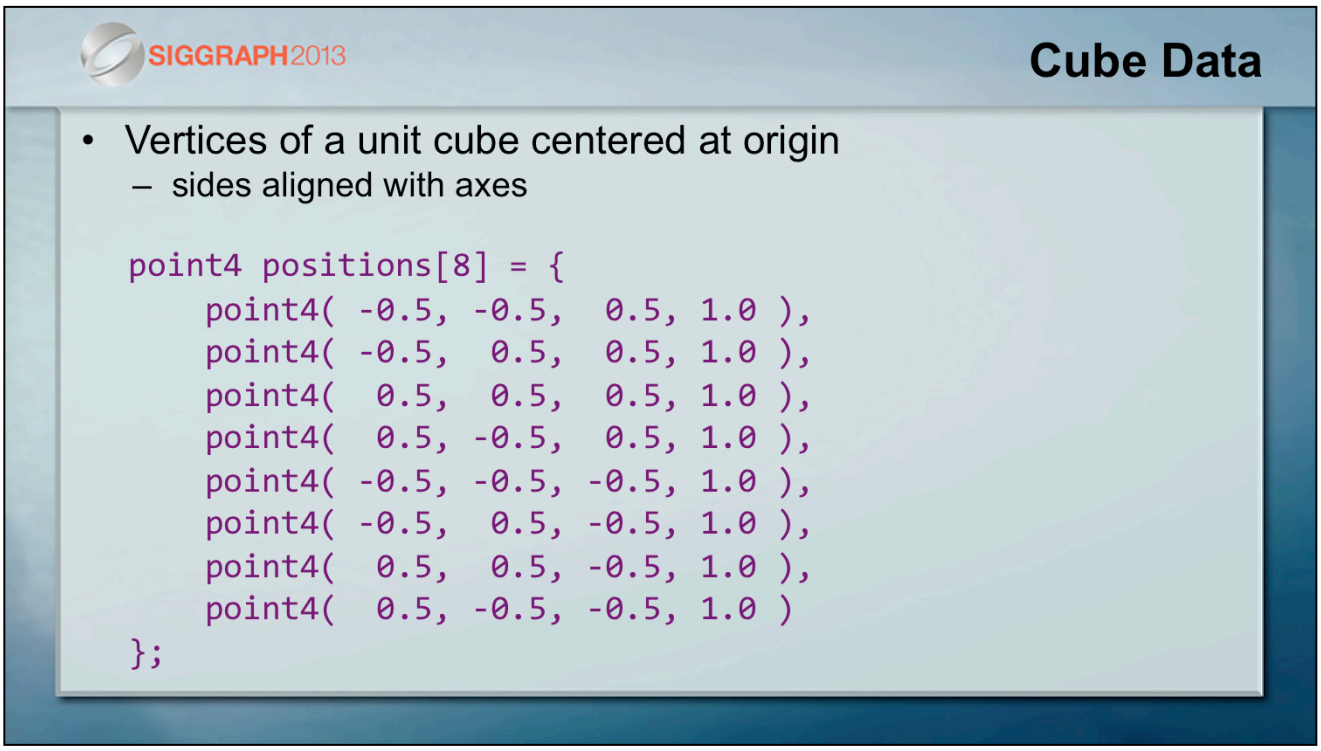
In order to simplify our application development, we define a few types and constants to make our code more readable and organized.

Our cube, like any other cube, has six square faces, each of which we'll draw as two triangles. In order to sizes memory arrays to hold the necessary vertex data, we define the constant `NumVertices`.

Additionally, as we'll see in our first shader, the OpenGL shading language, GLSL, has a built-in type called vec4, which represents a vector of four floating-point values. We define a C++ class for our application that has the same semantics as that GLSL type. Additionally, to logically associate a type for our data with what we intend to do with it, we leverage C++ typedefs to create aliases for colors and positions.



In order to provide data for OpenGL to use, we need to stage it so that we can load it into the VBOs that our application will use. In your applications, you might load these data from a file, or generate them on the fly. For each vertex, we want to use two bits of data – *vertex attributes* in OpenGL speak – to help process each vertex to draw the cube. In our case, each vertex has a position in space, and an associated color. To store those values for later use in our VBOs, we create two arrays to hold the per vertex data. Note that we can organize our data in other ways such as with a single array with interleaved positions and colors.

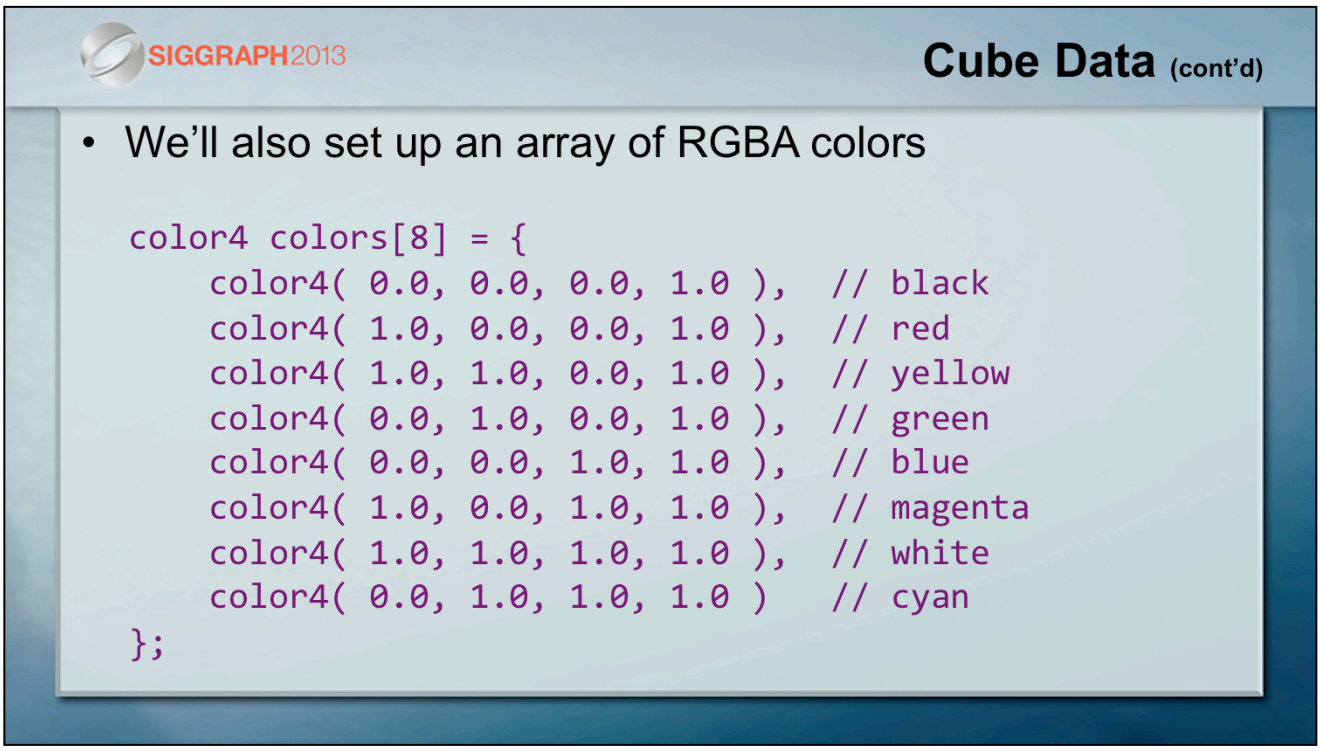


In our example we'll copy the coordinates of our cube model into a VBO for OpenGL to use. Here we set up an array of eight coordinates for the corners of a unit cube centered at the origin.

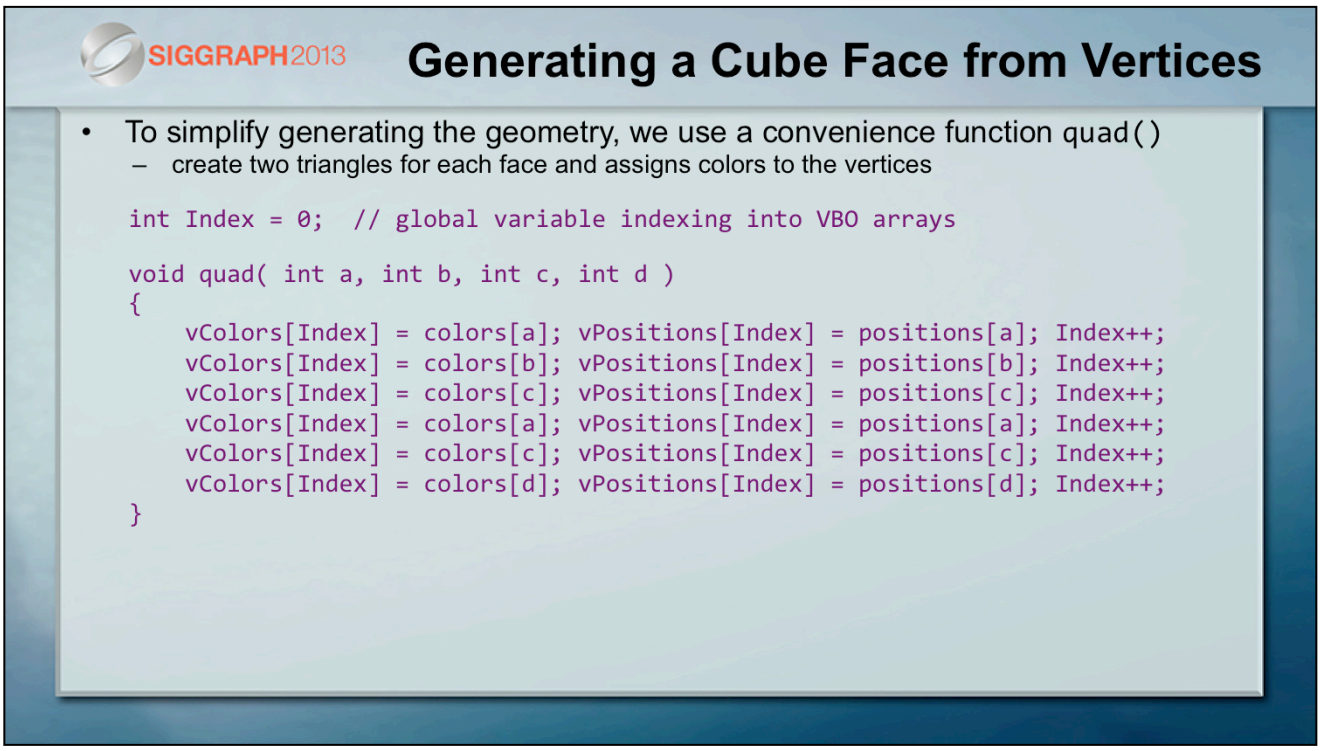
You may be asking yourself: "Why do we have four coordinates for 3D data?" The answer is that in computer graphics, it's often useful to include a fourth coordinate to represent three-dimensional coordinates, as it allows numerous mathematical techniques that are common operations in graphics to be done in the same way. In fact, this four-dimensional coordinate has a proper name, a *homogenous coordinate*. We could also use a point3 type, i.e.

point2(-0.5, -0.5, 0.5)

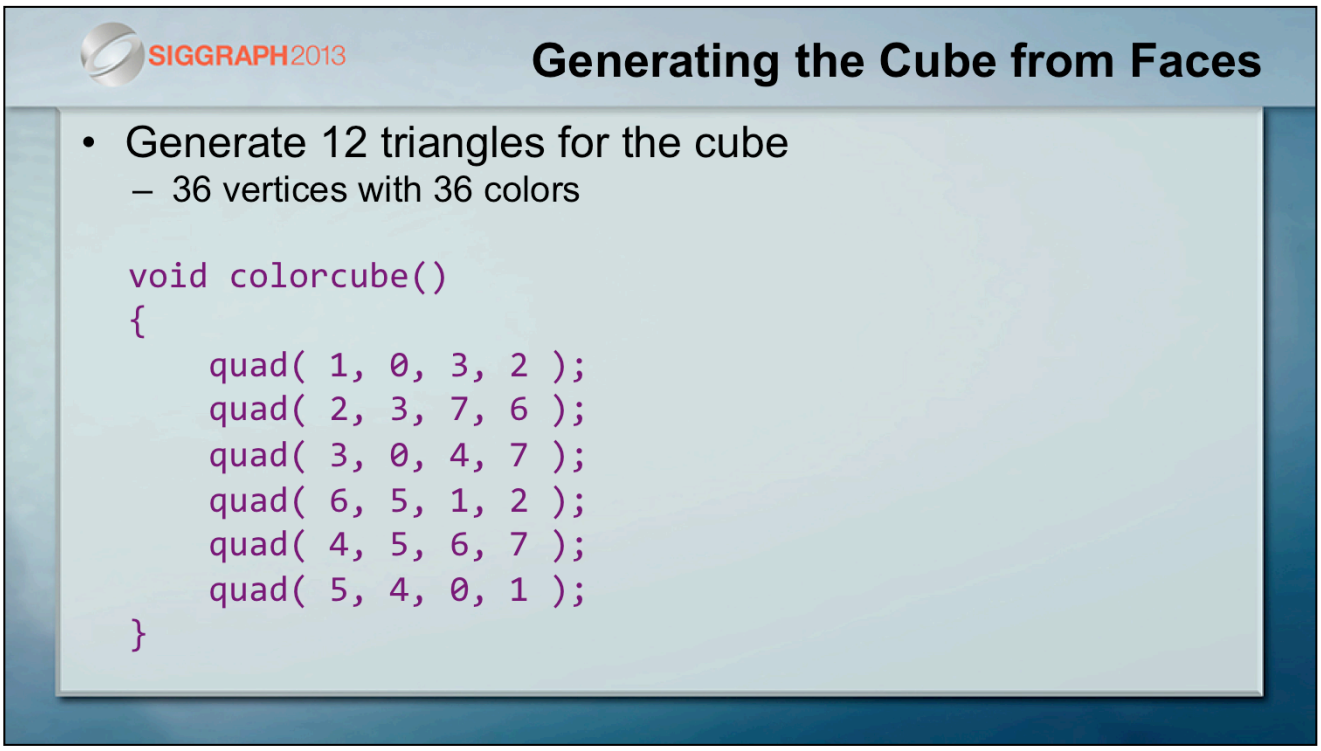
which will be stored in 4 dimensions on the GPU.



Just like our positional data, we'll set up a matching set of colors for each of the model's vertices, which we'll later copy into our VBO. Here we set up eight RGBA colors. In OpenGL, colors are processed in the pipeline as floating-point values in the range [0.0, 1.0]. Your input data can take any for; for example, image data from a digital photograph usually has values between [0, 255]. OpenGL will (if you request it), automatically convert those values into [0.0, 1.0], a process called *normalizing* values.

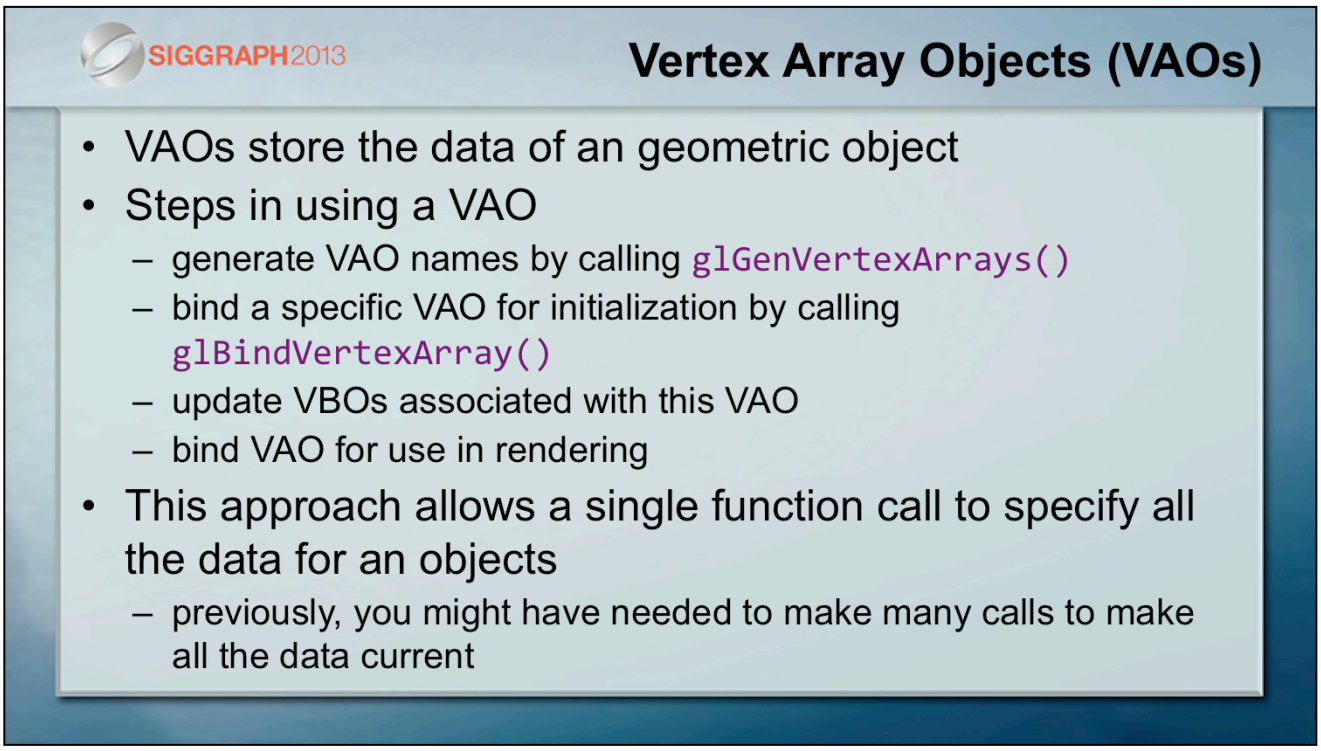


As our cube is constructed from square cube faces, we create a small function, `quad()`, which takes the indices into the original vertex color and position arrays, and copies the data into the VBO staging arrays. If you were to use this method (and we'll see better ways in a moment), you would need to remember to reset the `Index` value between setting up your VBO arrays.



Here we complete the generation of our cube's VBO data by specifying the six faces using index values into our original `positions` and `colors` arrays. It's worth noting that the order that we choose our vertex indices is important, as it will affect something called *backface culling* later.

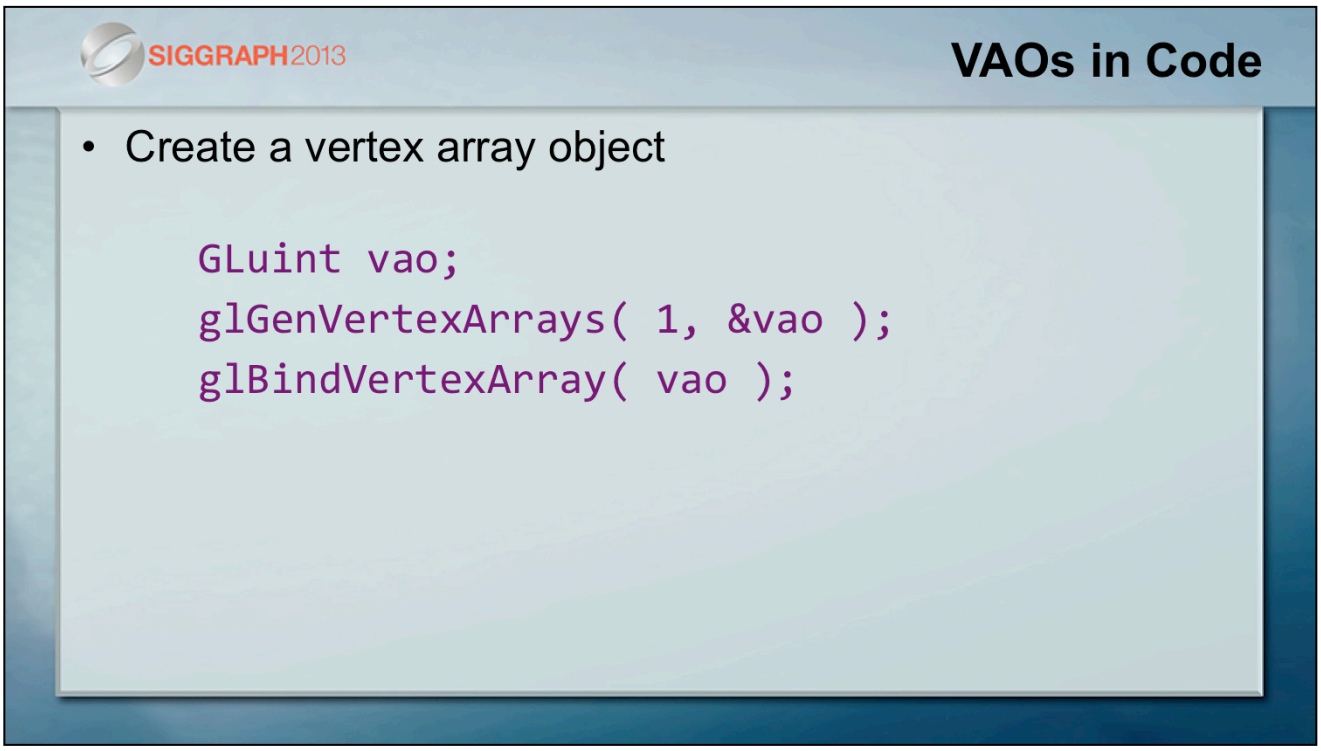
We'll see later that instead of creating the cube by copying lots of data, we can use our original vertex data along with just the indices we passed into `quad()` here to accomplish the same effect. That technique is very common, and something you'll use a lot. We chose this to introduce the technique in this manner to simplify the OpenGL concepts for loading VBO data.



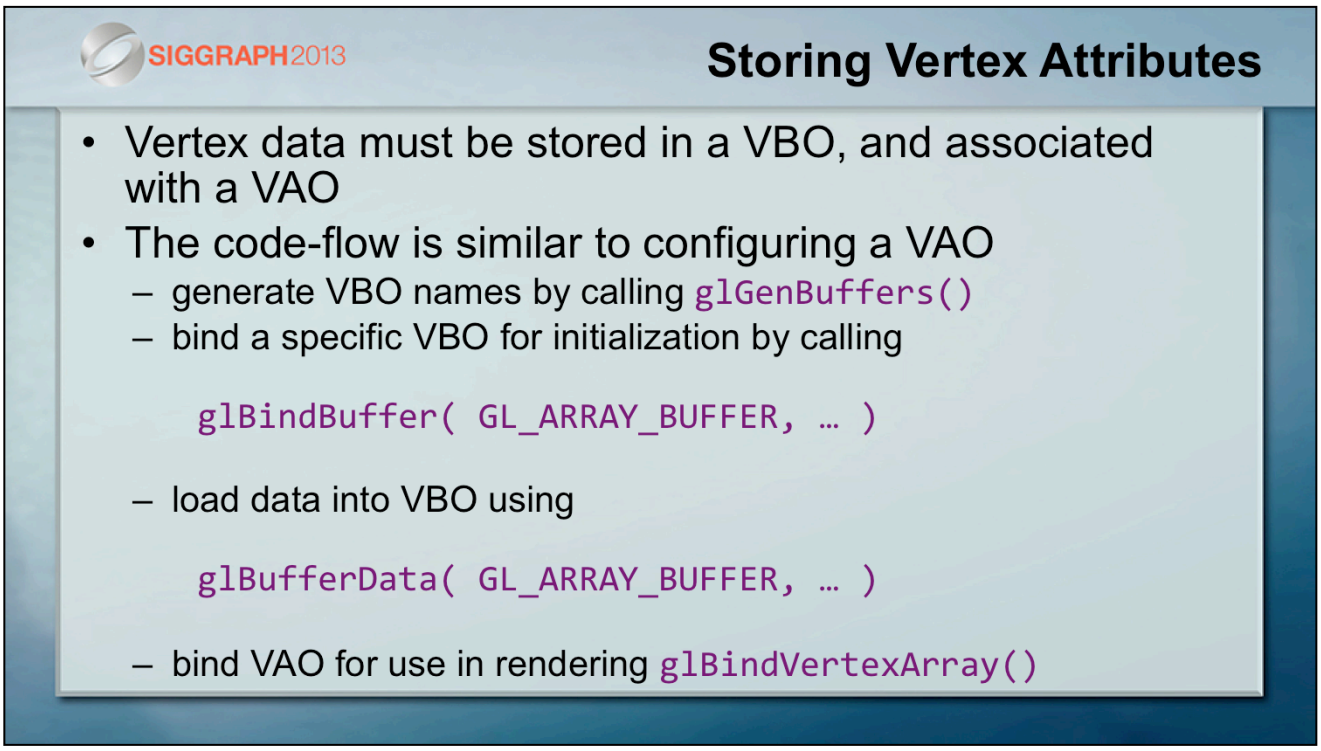
Similarly to VBOs, *vertex array objects* (VAOs) encapsulate all of the VBO data for an object. This allows much easier switching of data when rendering multiple objects (provided the data's been set up in multiple VAOs).

The process for initializing a VAO is similar to that of a VBO, except a little less involved.

1. First, generate a name VAO name by calling `glGenVertexArrays()`
2. Next, make the VAO "current" by calling `glBindVertexArray()`. Similar to what was described for VBOs, you'll call this every time you want to use or update the VBOs contained within this VAO.



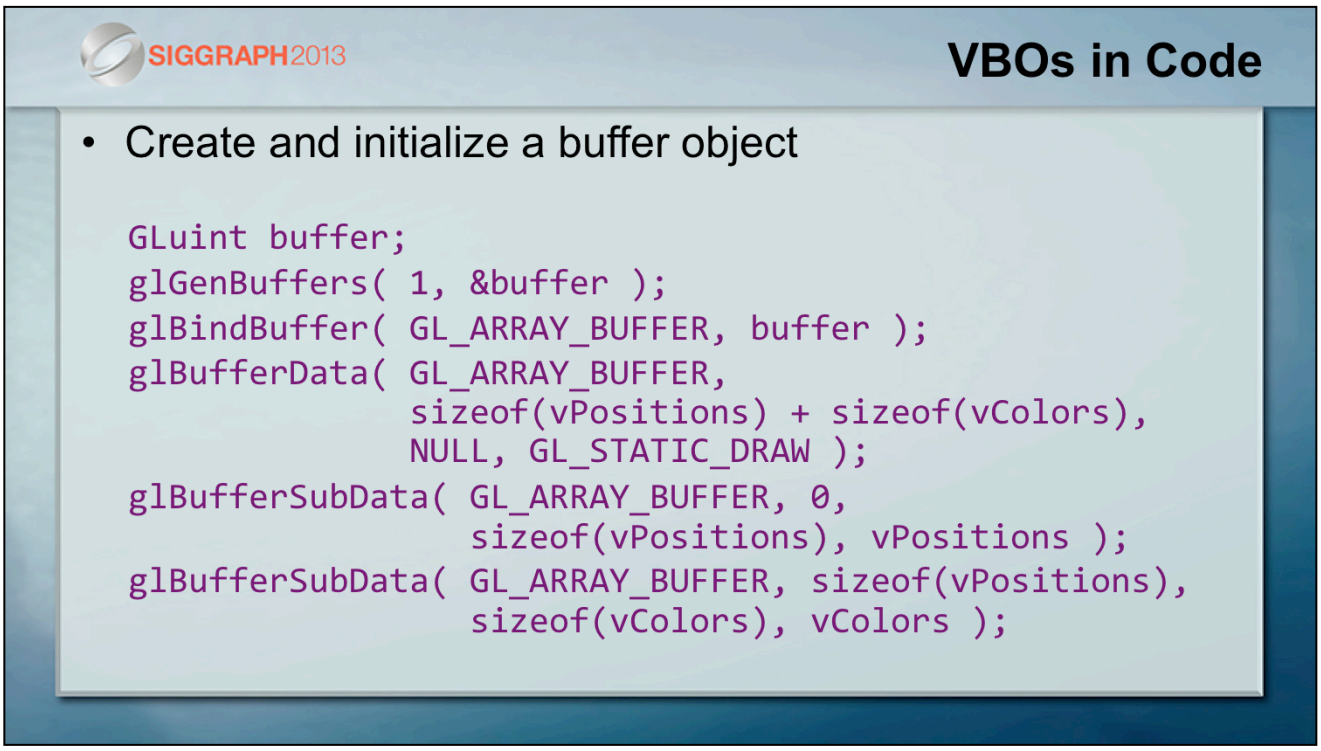
The above sequence calls shows how to create and bind a VAO. Since all geometric data in OpenGL must be stored in VAOs, you'll use this code idiom often.



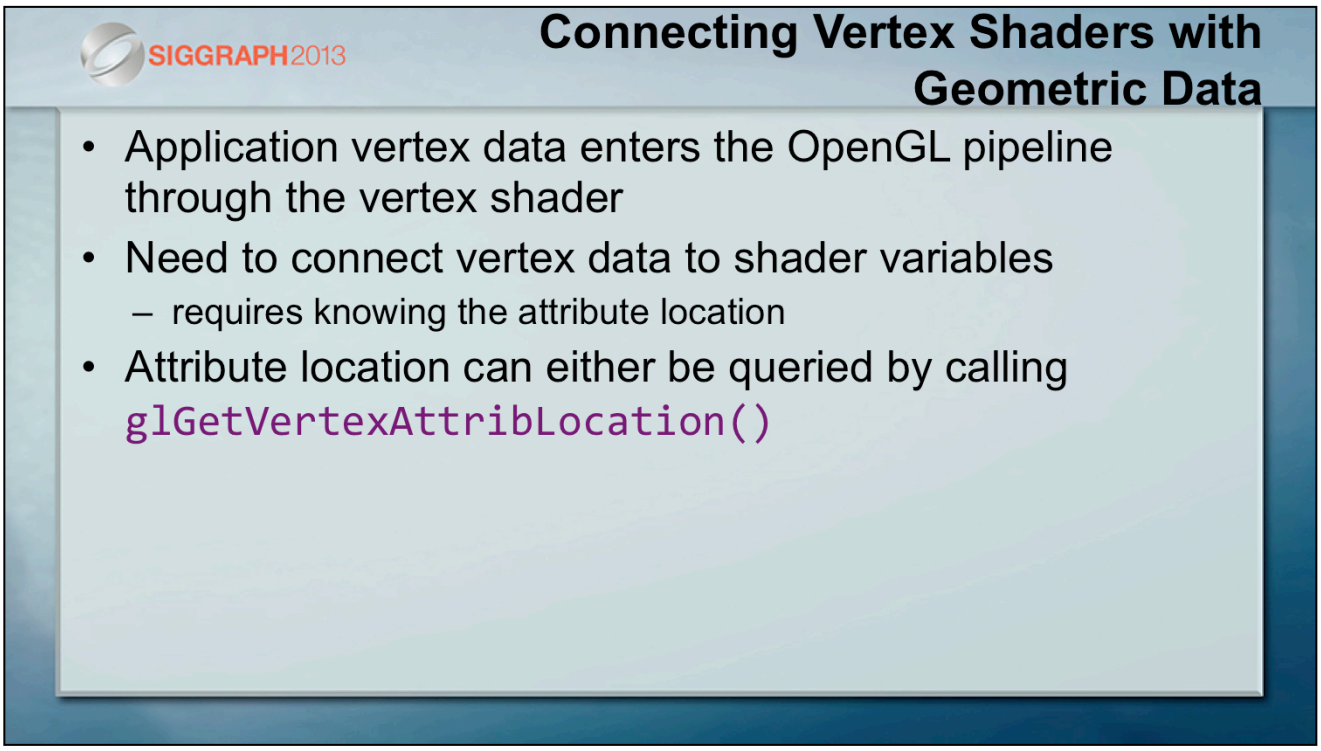
While we've talked a lot about VBOs, we haven't detailed how one goes about creating them. Vertex buffer objects, like all (memory) objects in OpenGL (as compared to geometric objects) are created in the same way, using the same set of functions. In fact, you'll see that the pattern of calls we make here are similar to other sequences of calls for doing other OpenGL operations.

In the case of vertex buffer objects, you'll do the following sequence of function calls:

1. Generate a buffer's name by calling `glGenBuffers()`
2. Next, you'll make that buffer the "current" buffer, which means it's the selected buffer for reading or writing data values by calling `glBindBuffer()`, with a type of `GL_ARRAY_BUFFER`. There are different types of buffer objects, with an array buffer being the one used for storing geometric data.
3. To initialize a buffer, you'll call `glBufferData()`, which will copy data from your application into the GPU's memory. You would do the same operation if you also wanted to update data in the buffer
4. Finally, when it comes time to render using the data in the buffer, you'll once again call `glBindVertexArray()` to make it and its VBOs current again. In fact, if you have multiple objects, each with their own VAO, you'll likely call `glBindVertexArray()` once per frame for each object.

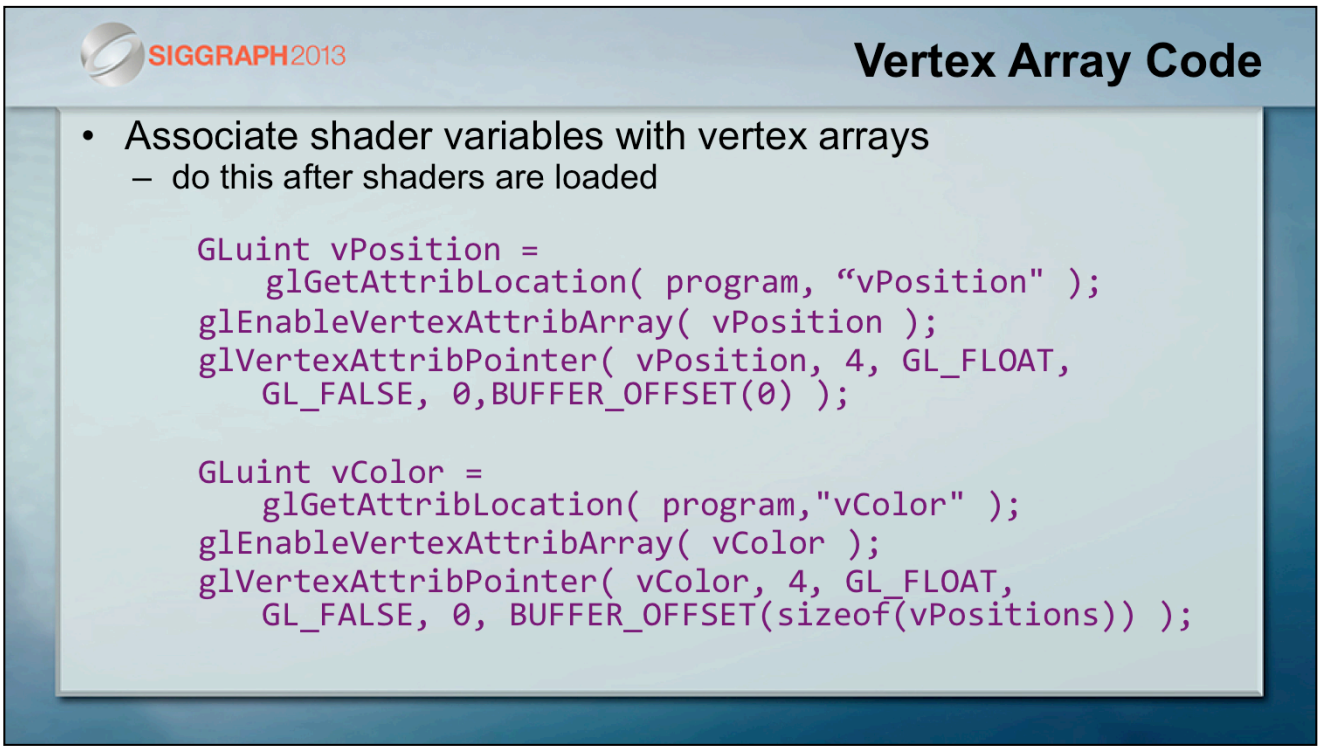


The above sequence of calls illustrates generating, binding, and initializing a VBO with data. In this example, we use a technique permitting data to be loaded into two steps, which we need as our data values are in two separate arrays. It's noteworthy to look at the `glBufferData()` call; in this call, we basically have OpenGL allocate an array sized to our needs (the combined size of our point and color arrays), but don't transfer any data with the call, which is specified with the NULL value. This is akin to calling `malloc()` to create a buffer of uninitialized data. We later load that array with our calls to `glBufferSubData()`, which allows us to replace a subsection of our array. This technique is also useful if you need to update data inside of a VBO at some point in the execution of your application.



The final step in preparing you data for processing by OpenGL (i.e., sending it down for rendering) is to specify which vertex attributes you'd like issued to the graphics pipeline. While this might seem superfluous, it allows you to specify multiple collections of data, and choose which ones you'd like to use at any given time.

Each of the attributes that we enable must be associated with an "in" variable of the currently bound vertex shader. You retrieve vertex attribute locations was retrieved from the compiled shader by calling `glGetAttribLocation()`. We discuss this call in the shader section.



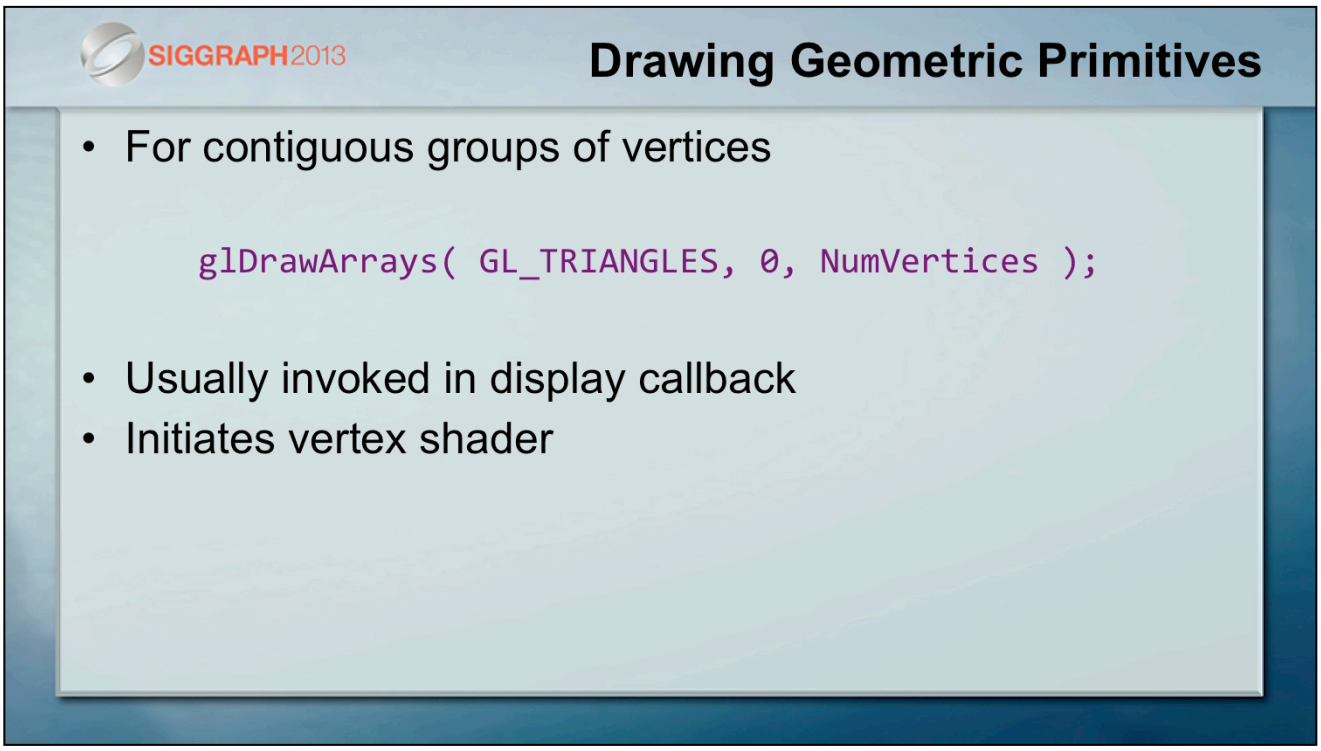
To complete the “plumbing” of associating our vertex data with variables in our shader programs, you need to tell OpenGL where in our buffer object to find the vertex data, and which shader variable to pass the data to when we draw. The above code snippet shows that process for our two data sources. In our shaders (which we’ll discuss in a moment), we have two variables: `vPosition`, and `vColor`, which we will associate with the data values in our VBOs that we copied form our vertex `positions` and `colors` arrays.

The calls to `glGetAttribLocation()` will return a compiler-generated index which we need to use to complete the connection from our data to the shader inputs. We also need to “turn the valve” on our data by enabling its attribute array by calling `glEnableVertexAttribArray()` with the selected attribute location.

This is the most flexible approach to this process, but depending on your OpenGL version, you may be able to use the `layout` construct, which allows you to specify the attribute location, as compared to having to retrieve it after compiling and linking your shaders. We’ll discuss that in our shader section later in the course.

BUFFER_OFFSET is a simple macro defined to make the code more readable

```
#define BUFFER_OFFSET( offset )   ((GLvoid*) (offset))
```

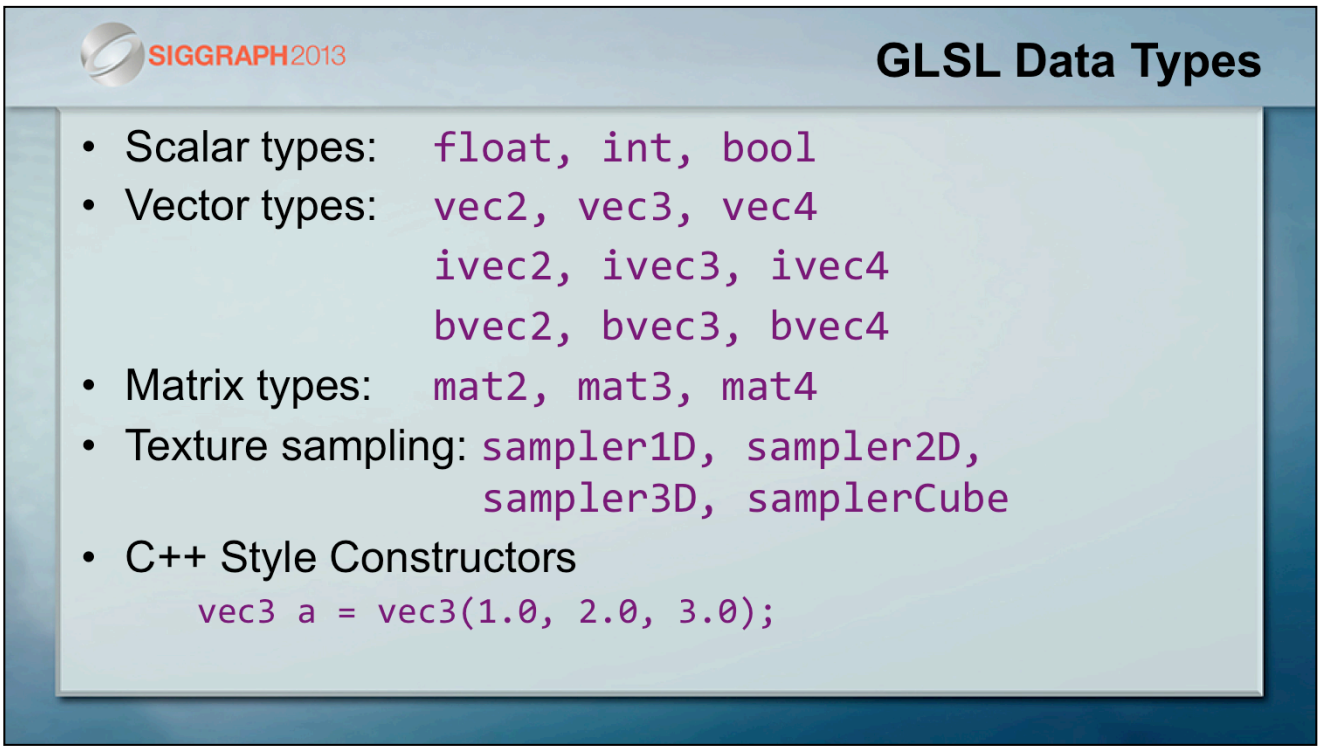


In order to initiate the rendering of primitives, you need to issue a drawing routine. While there are many routines for this in OpenGL, we'll discuss the most fundamental ones. The simplest routine is `glDrawArrays()`, to which you specify what type of graphics primitive you want to draw (e.g., here we're rending a triangle strip), which vertex in the enabled vertex attribute arrays to start with, and how many vertices to send.

This is the simplest way of rendering geometry in OpenGL Version 3.1. You merely need to store you vertex data in sequence, and then `glDrawArrays()` takes care of the rest. However, in some cases, this won't be the most memory efficient method of doing things. Many geometric objects share vertices between geometric primitives, and with this method, you need to replicate the data once for each vertex. We'll see a more flexible, in terms of memory storage and access in the next slides.

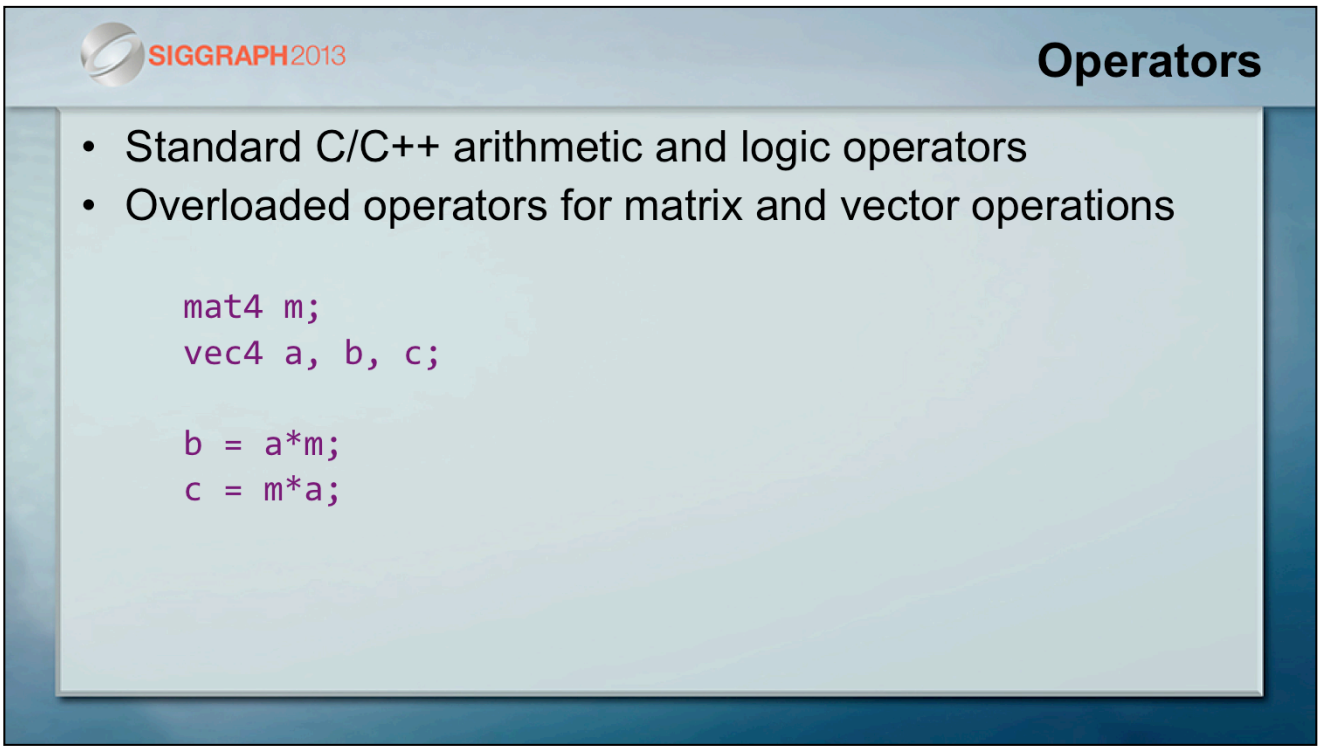
Shaders and GLSL

SIGGRAPH 2013
The 40th International Conference and Exhibition
on Computer Graphics and Interactive Techniques



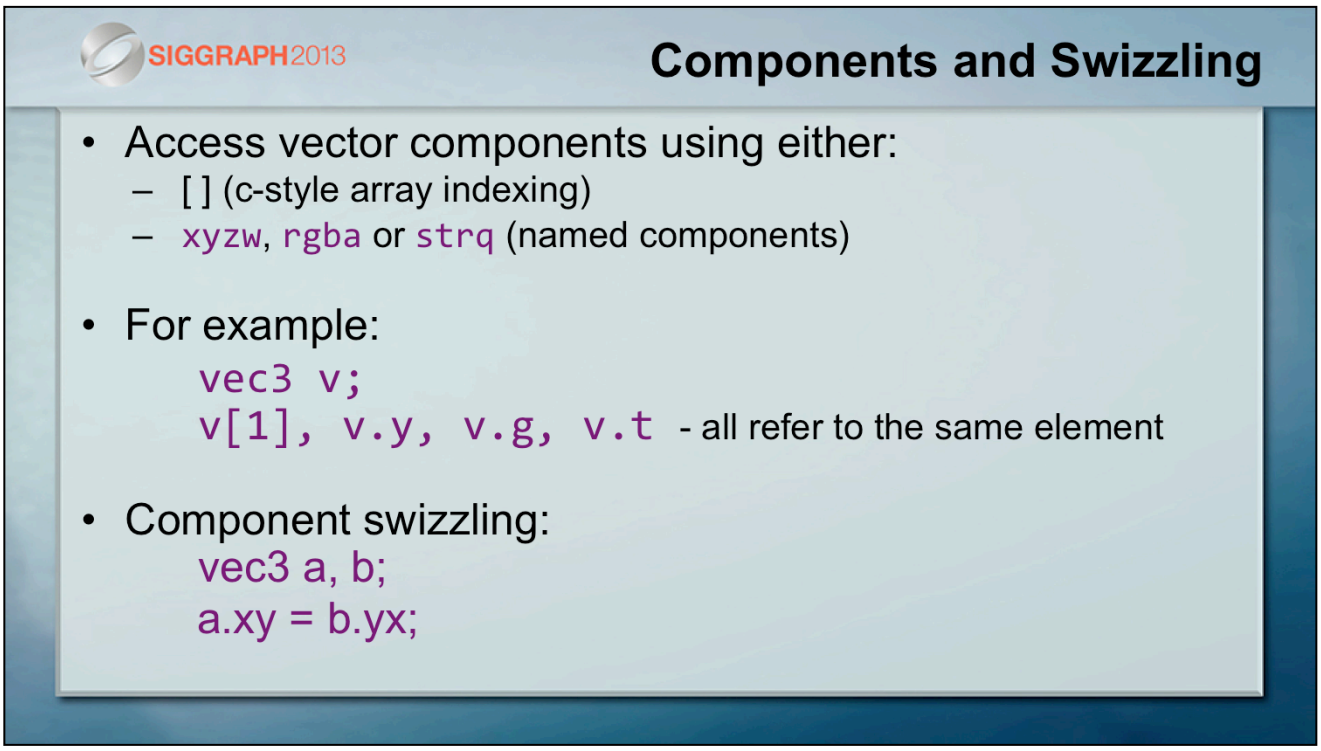
As with any programming language, GLSL has types for variables. However, it includes vector-, and matrix-based types to simplify the operations that occur often in computer graphics.

In addition to numerical types, other types like *texture samplers* are used to enable other OpenGL operations. We'll discuss texture samplers in the texture mapping section.



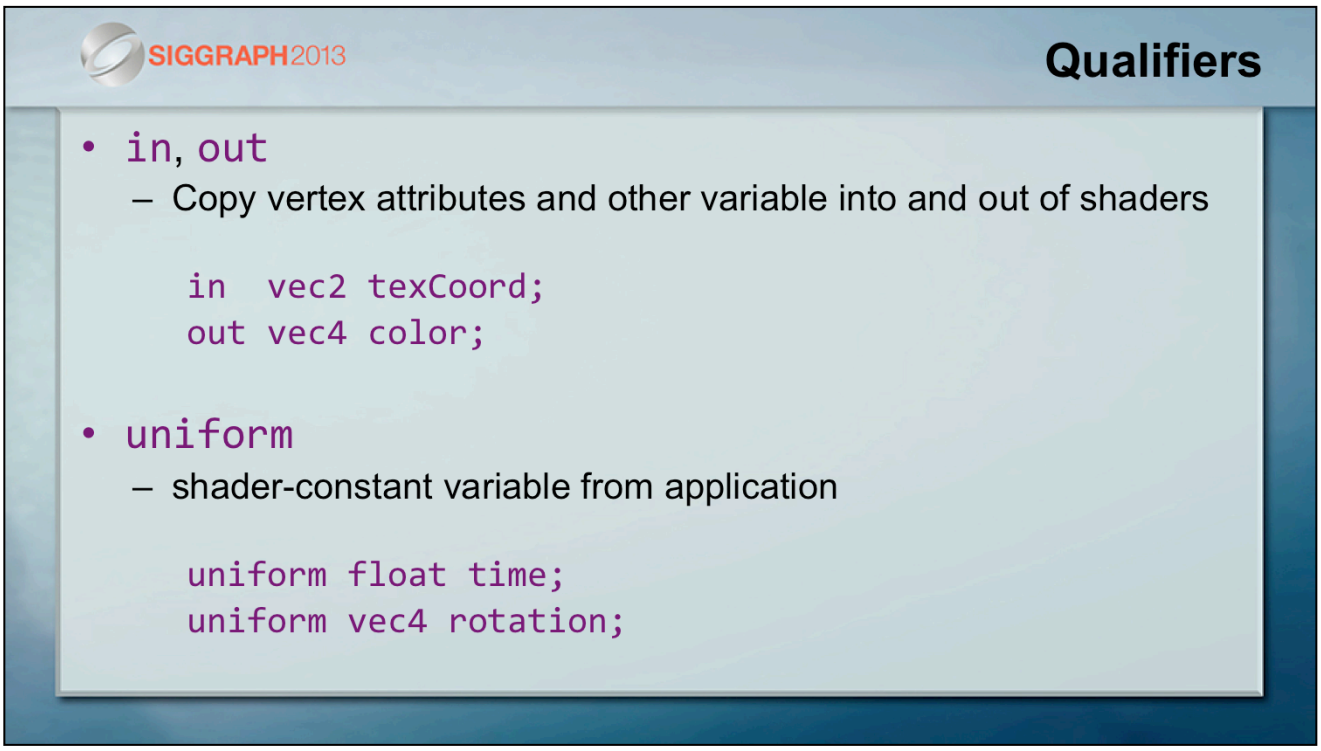
The vector and matrix classes of GLSL are first-class types, with arithmetic and logical operations well defined. This helps simplify your code, and prevent errors.

Note in the above example, overloading ensures that both a*m and m*a are defined although they will not in general produce the same result.



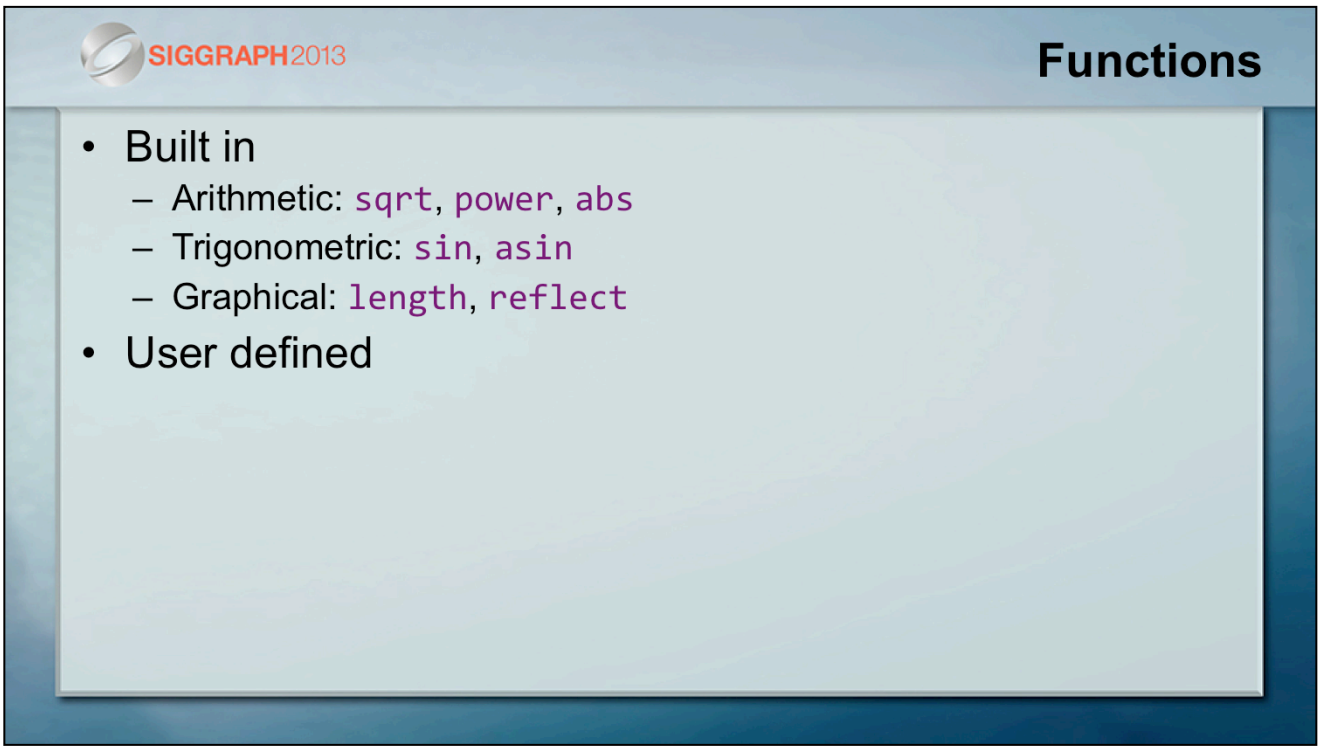
For GLSL's vector types, you'll find that often you may also want to access components within the vector, as well as operate on all of the vector's components at the same time. To support that, vectors and matrices (which are really a vector of vectors), support normal "C" vector accessing using the square-bracket notation (e.g., "[i]"), with zero-based indexing. Additionally, vectors (but not matrices) support *swizzling*, which provides a very powerful method for accessing and manipulating vector components.

Swizzles allow components within a vector to be accessed by name. For example, the first element in a vector – element 0 – can also be referenced by the names "x", "s", and "r". Why all the names – to clarify their usage. If you're working with a color, for example, it may be clearer in the code to use "r" to represent the red channel, as compared to "x", which make more sense as the x-positional coordinate

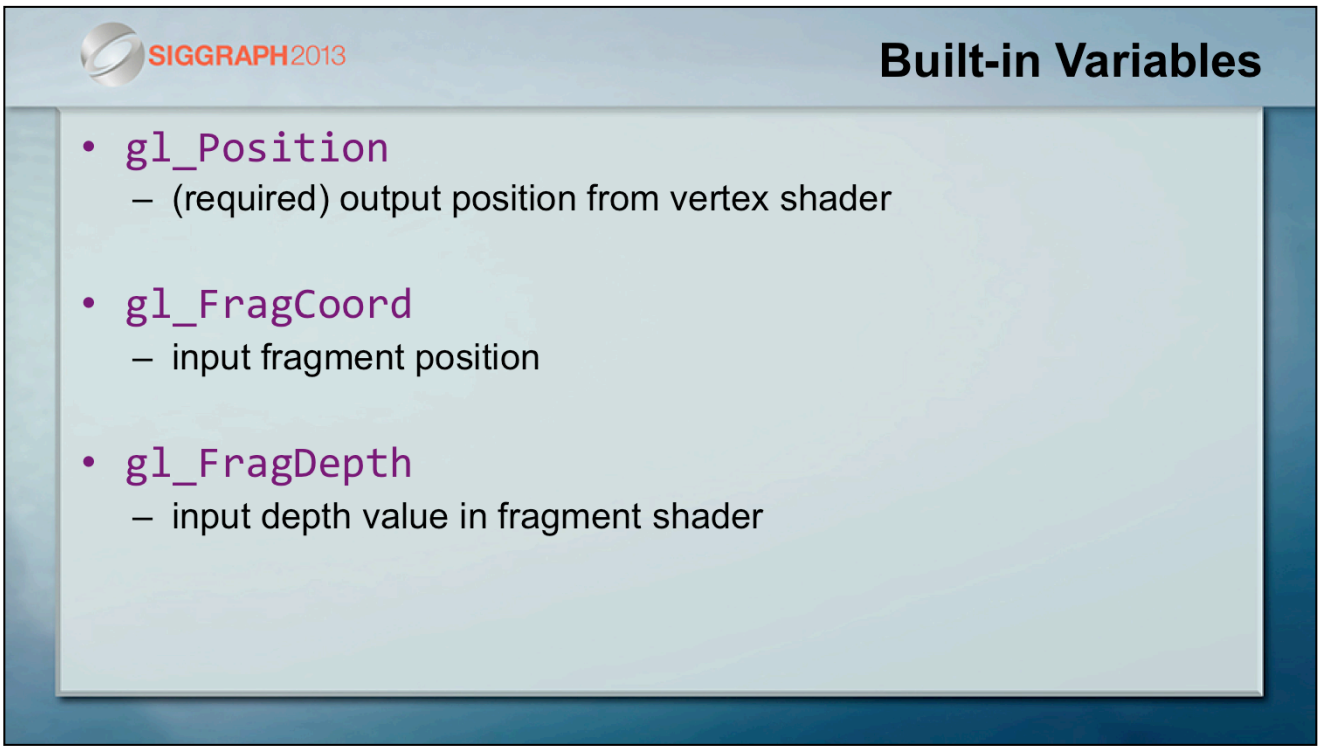


In addition to types, GLSL has numerous qualifiers to describe a variable usage. The most common of those are:

- `in` qualifiers that indicate the shader variable will receive data flowing into the shader, either from the application, or the previous shader stage.
- `out` qualifier which tag a variable as data output where data will flow to the next shader stage, or to the framebuffer
- `uniform` qualifiers for accessing data that doesn't change across a draw operation

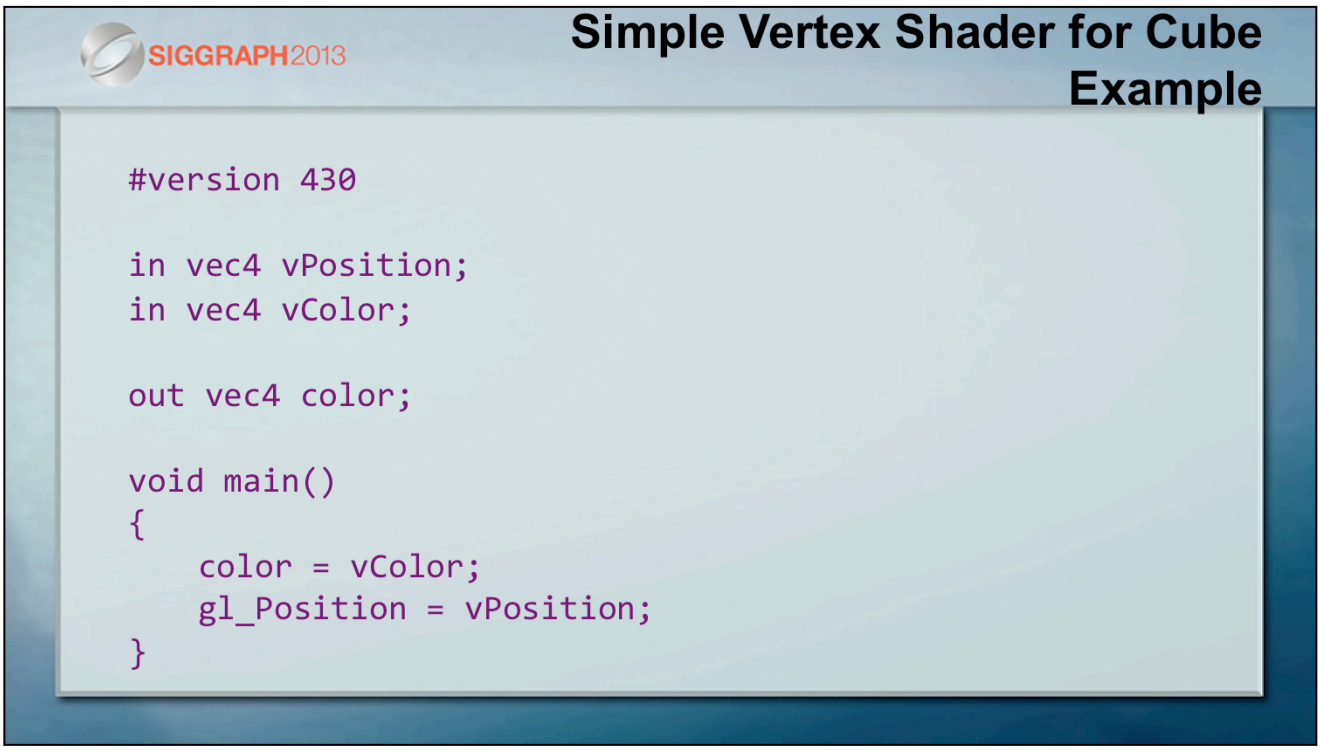


GLSL also provides a rich library of functions supporting common operations. While pretty much every vector- and matrix-related function available you can think of, along with the most common mathematical functions are built into GLSL, there's no support for operations like reading files or printing values. Shaders are really data-flow engines with data coming in, being processed, and sent on for further processing.

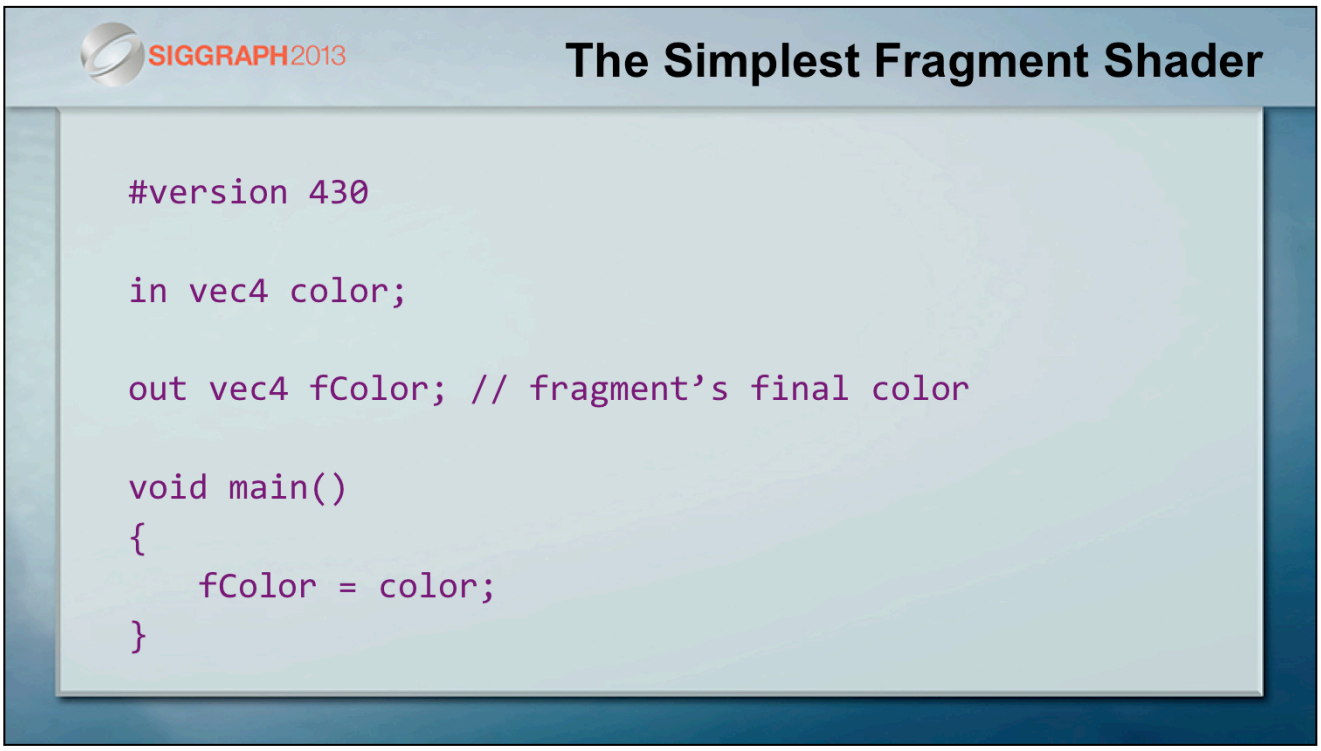


Fundamental to shader processing are a couple of built-in GLSL variable which are the terminus for operations. In particular, vertex data, which can be processed by up to four shader stages in OpenGL, are all ended by setting a positional value into the built-in variable, `gl_Position`.

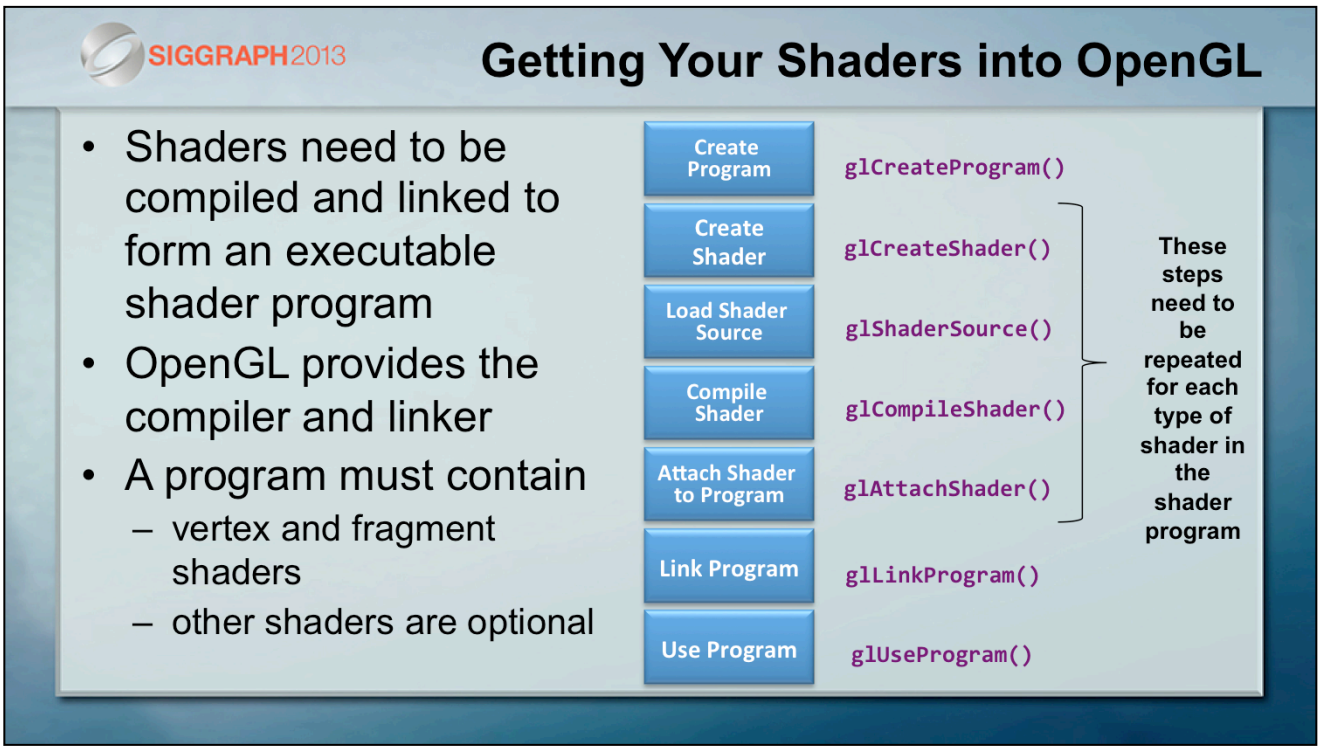
Additionally, fragment shaders provide a number of built-in variables. For example, `gl_FragCoord` is a read-only variable, while gl_FragDepth is a read-write variable. Later versions of OpenGL allow fragment shaders to output to other variables of the user's designation as well.



Here's the simple vertex shader we use in our cube rendering example. It accepts two vertex attributes as input: the vertex's position and color, and does very little processing on them; in fact, it merely copies the input into some output variables (with `gl_Position` being implicitly declared). The results of each vertex shader execution are passed further down the OpenGL pipeline, and ultimately end their processing in the fragment shader.

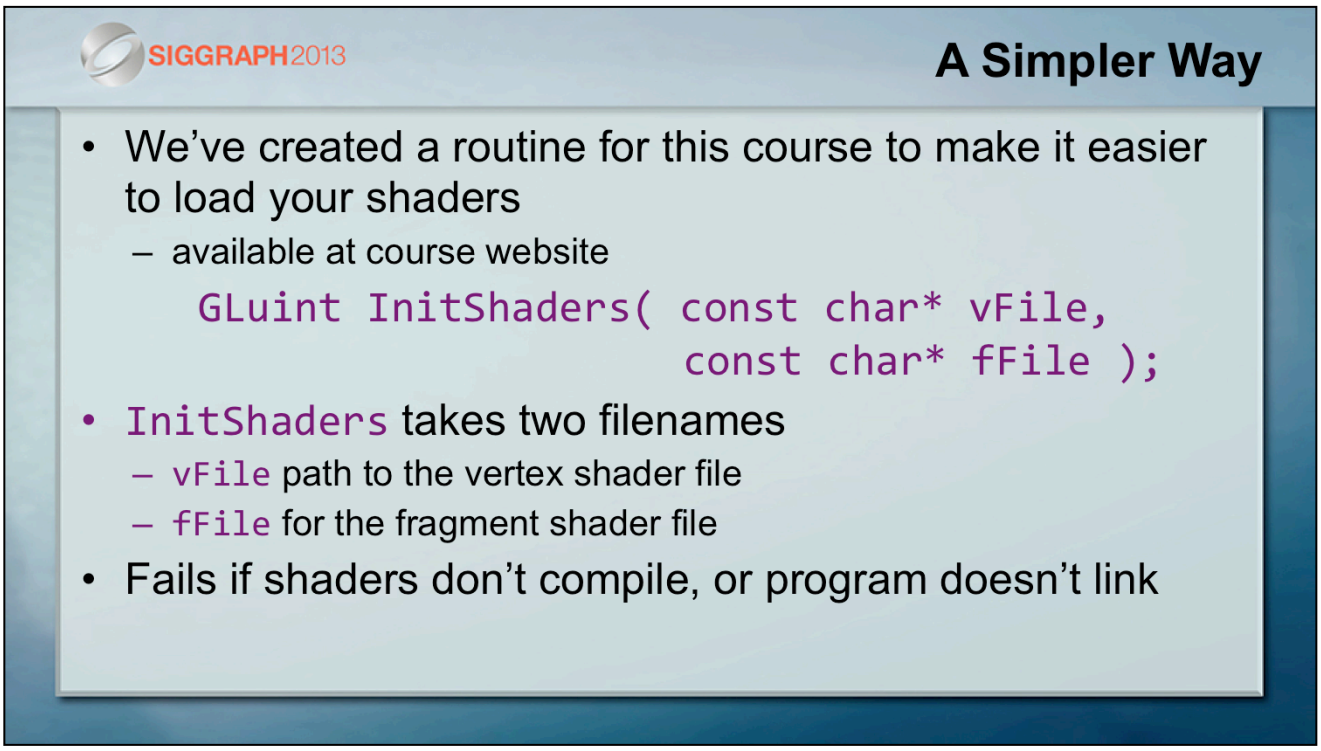


Here's the associated fragment shader that we use in our cube example. While this shader is as simple as they come – merely setting the fragment's color to the input color passed in, there's been a lot of processing to this point. In particular, every fragment that's shaded was generated by the rasterizer, which is a built-in, non-programmable (i.e., you don't write a shader to control its operation). What's magical about this process is that if the colors across the geometric primitive (for multi-vertex primitives: lines and triangles) is not the same, the rasterizer will interpolate those colors across the primitive, passing each iterated value into our `color` variable.



Shaders need to be compiled in order to be used in your program. As compared to C programs, the compiler and linker are implemented in the OpenGL driver, and accessible through function calls from within your program. The diagram illustrates the steps required to compile and link each type of shader into your shader program. A program can contain either a vertex shader (which replaces the fixed-function vertex processing), a fragment shader (which replaces the fragment coloring stages), or both. If a shader isn't present for a particular stage, the fixed-function part of the pipeline is used in its place.

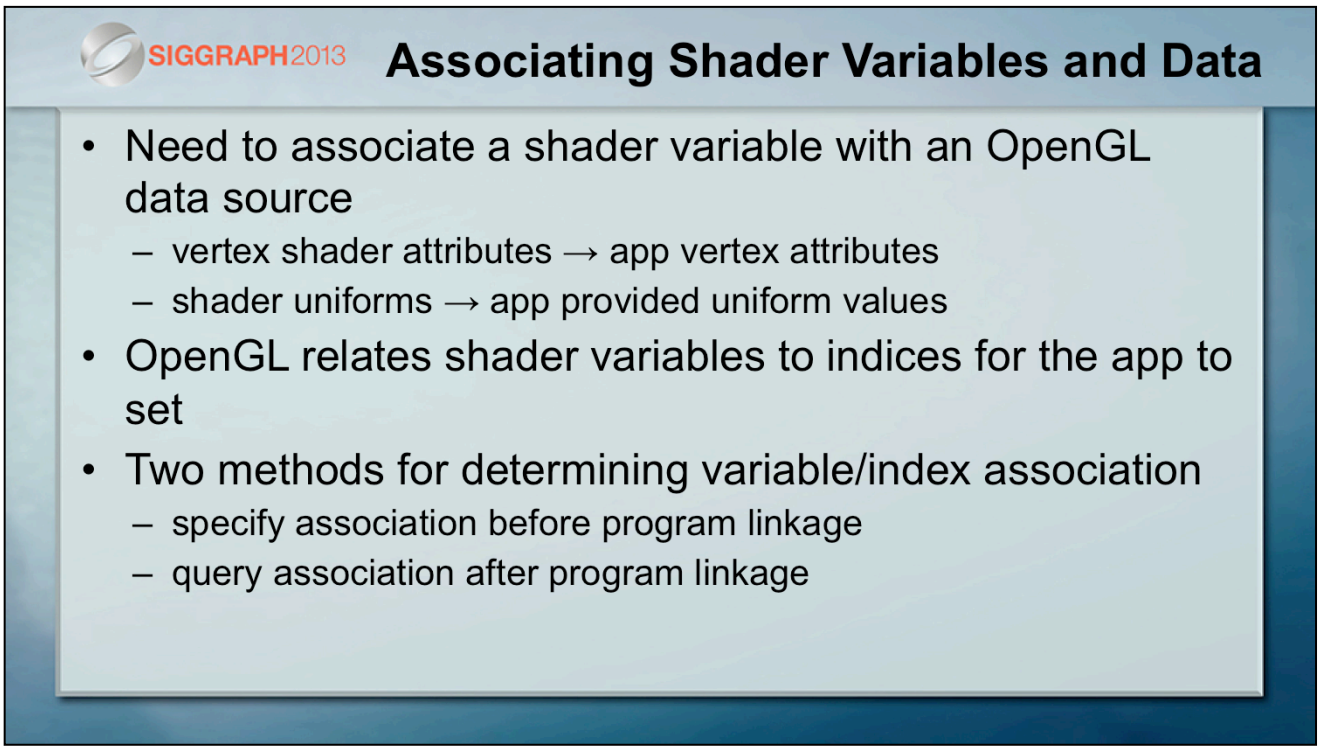
Just a with regular programs, a syntax error from the compilation stage, or a missing symbol from the linker stage could prevent the successful generation of an executable program. There are routines for verifying the results of the compilation and link stages of the compilation process, but are not shown here. Instead, we've provided a routine that makes this process much simpler, as demonstrated on the next slide.



To simplify our lives, we created a routine that simplifies loading, compiling, and linking shaders: `InitShaders()`. It implements the shader compilation and linking process shown on the previous slide. It also does full error checking, and will terminate your program if there's an error at some stage in the process (production applications might choose a less terminal solution to the problem, but it's useful in the classroom).

`InitShaders()` accepts two parameters, each a filename to be loaded as source for the vertex and fragment shader stages, respectively.

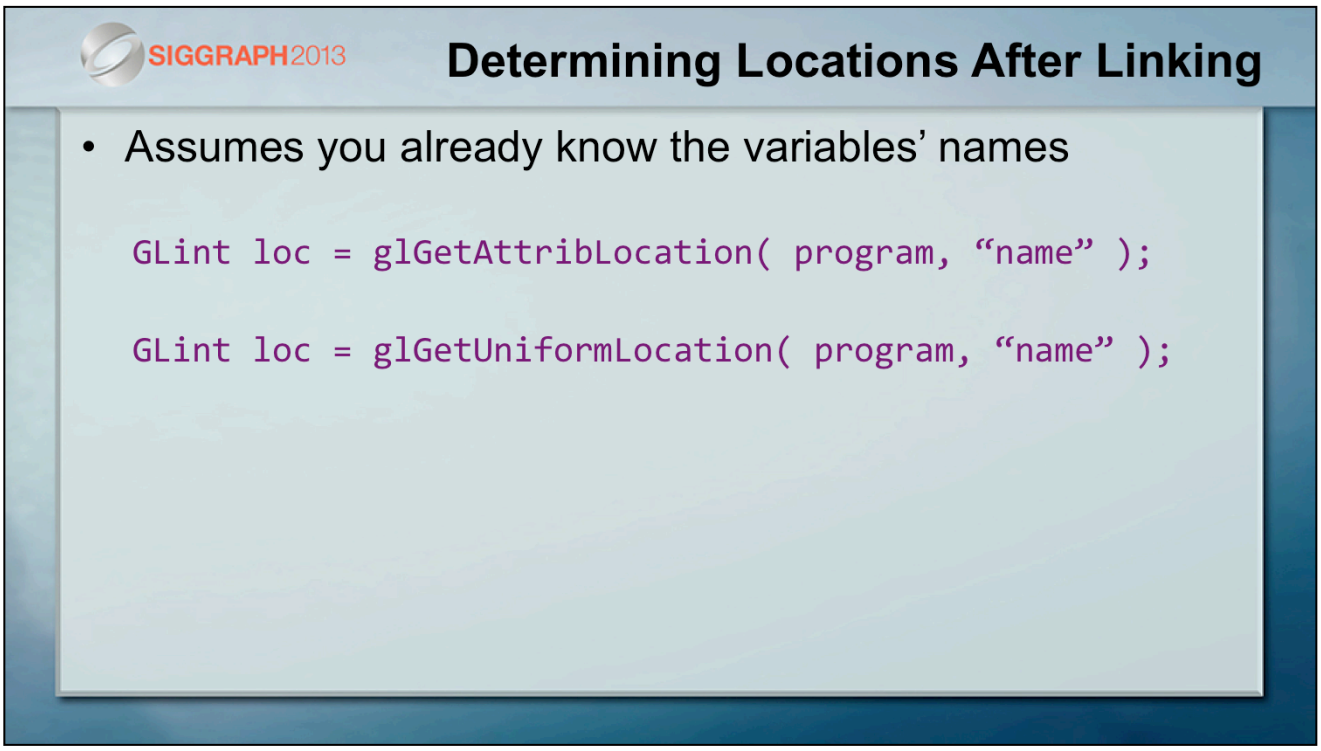
The value returned from `InitShaders()` will be a valid GLSL program id that you can pass into `glUseProgram()`.



OpenGL shaders, depending on which stage their associated with, process different types of data. Some data for a shader changes for each shader invocation. For example, each time a vertex shader executes, it's presented with new data for a single vertex; likewise for fragment, and the other shader stages in the pipeline. The number of executions of a particular shader rely on how much data was associated with the draw call that started the pipeline – if you call glDrawArrays() specifiying 100 vertices, your vertex shader will be called 100 times, each time with a different vertex.

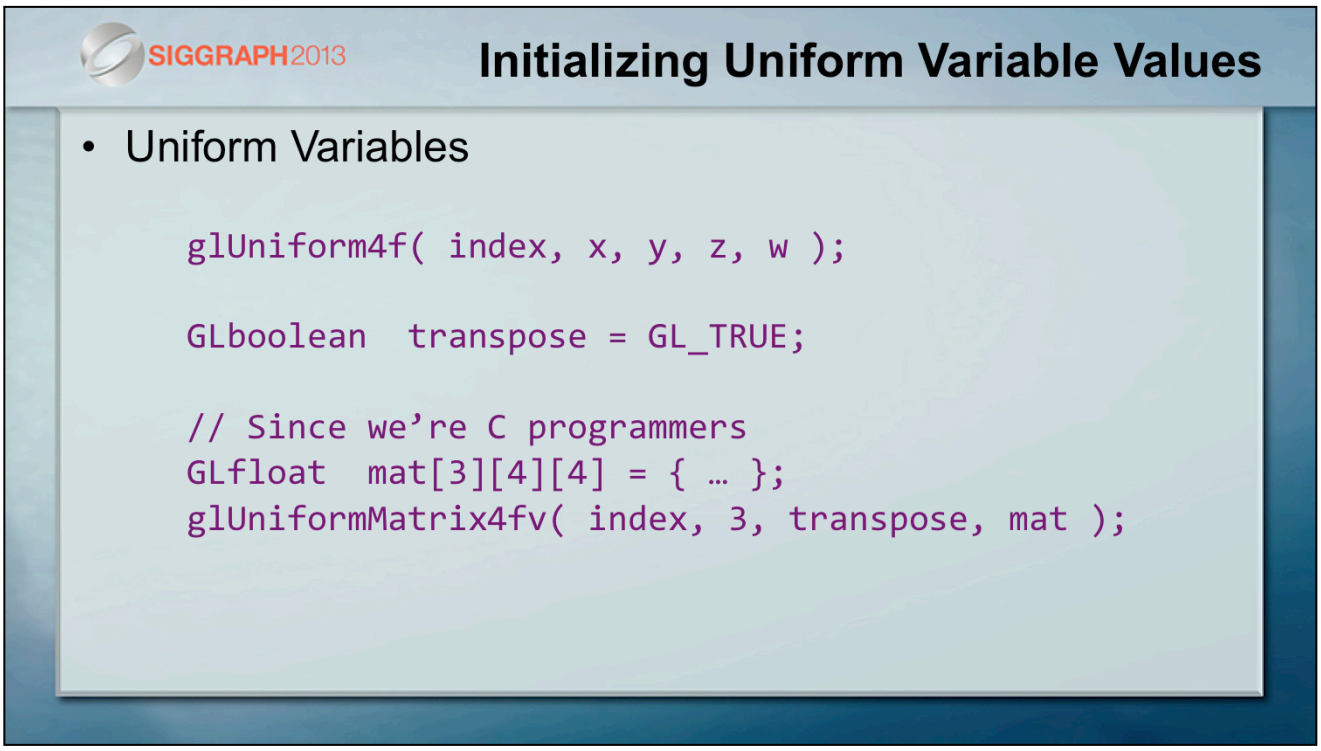
Other data that a shader may use in processing may be constant across a draw call, or even all the drawing calls for a frame. GLSL calls those *uniform* varialbes, since their value is uniform across the execution of all shaders for a single draw call.

Each of the shader's input data variables (ins and uniforms) needs to be connected to a data source in the application. We've already seen glGetAttribLocation() for retrieving information for connecting vertex data in a VBO to shader variable. You will also use the same process for uniform variables, as we'll describe shortly.

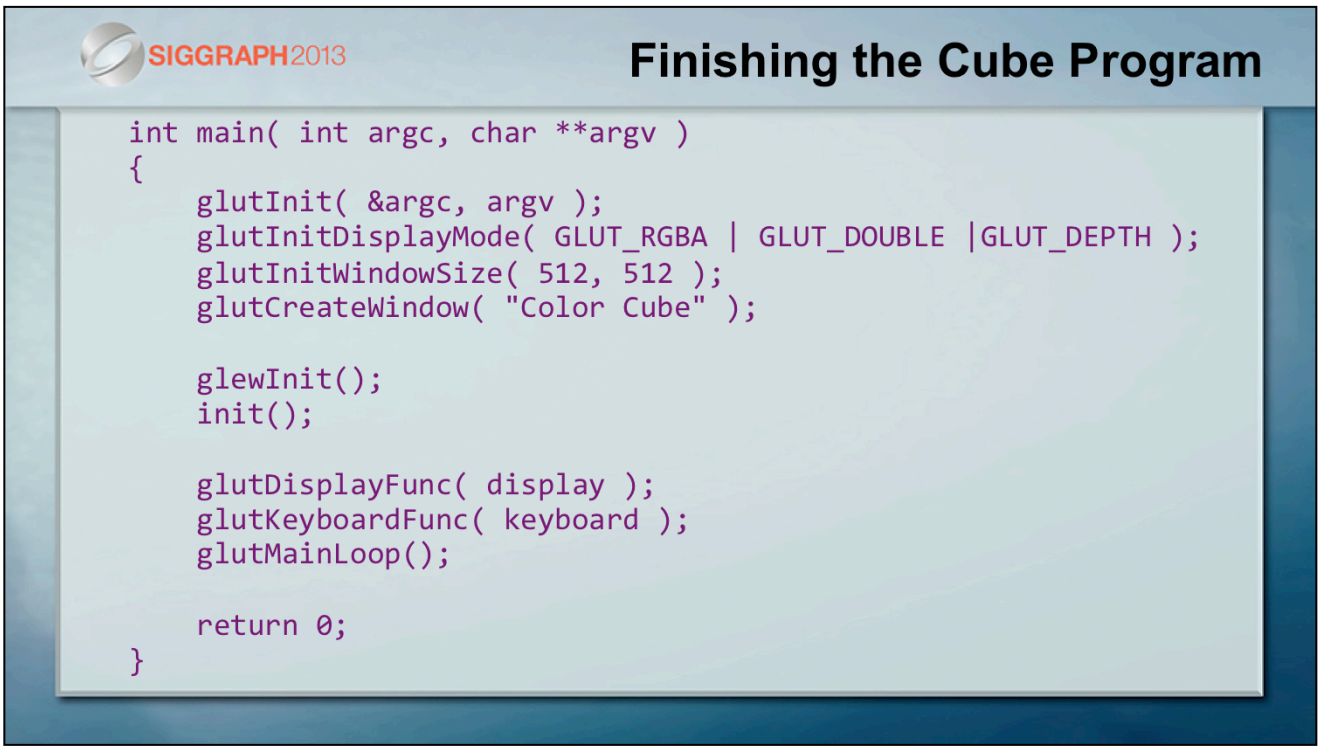


Once you know the names of variables in a shader – whether they're attributes or uniforms – you can determine their location using one of the glGet*Location() calls.

If you don't know the variables in a shader (if, for instance, you're writing a library that accepts shaders), you can find out all of the shader variables by using the glGetActiveAttrib() function.



You've already seen how one associates values with attributes by calling glVertexAttribPointer(). To specify a uniform's value, we use one of the glUniform*() functions. For setting a vector type, you'll use one of the glUniform*() variants, and for matrices you'll use a glUniformMatrix *() form.



You'll find that many OpenGL programs look very similar, particularly simple examples as we're showing in class. Above we demonstrate the basic initialization code for our examples. In our main() routine, you can see our use of the freeglut and GLEW libraries.

The main() has a number of tasks:

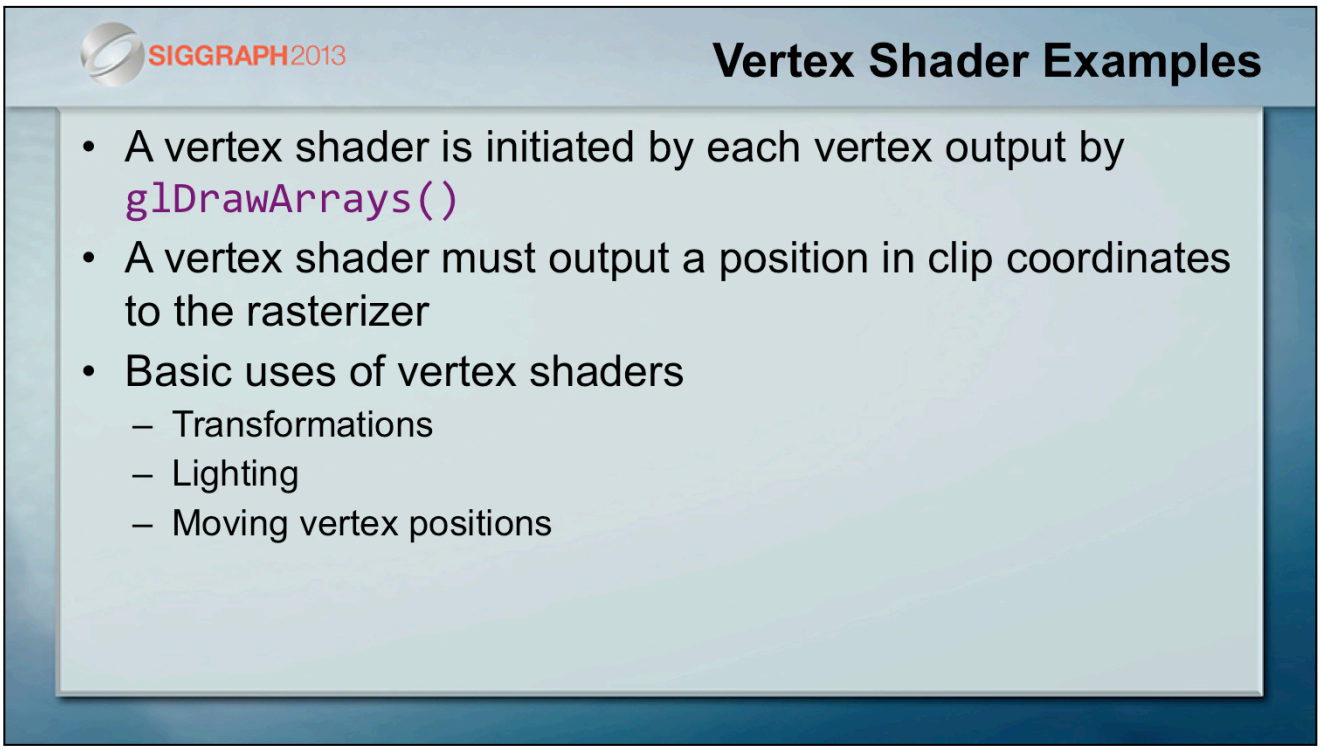
- Initialize and open a window
- Initialize the buffers and parameters by calling init()
- Specify the callback functions for events
- Enter an infinite event loop

Although callbacks aren't required by OpenGL, it is the standard method for developing interactive applications.

Cube Program's GLUT Callbacks

```
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
    glutSwapBuffers();
}

void keyboard( unsigned char key, int x, int y )
{
    switch( key ) {
        case 033: case 'q': case 'Q':
            exit( EXIT_SUCCESS );
            break;
    }
}
```

A display callback is required by freeglut. It is invoked whenever OpenGL determines a window has to be redrawn, i.e. when a window is first opened or the contents of a window are changed. In our example we use a keyboard callback to end the program.

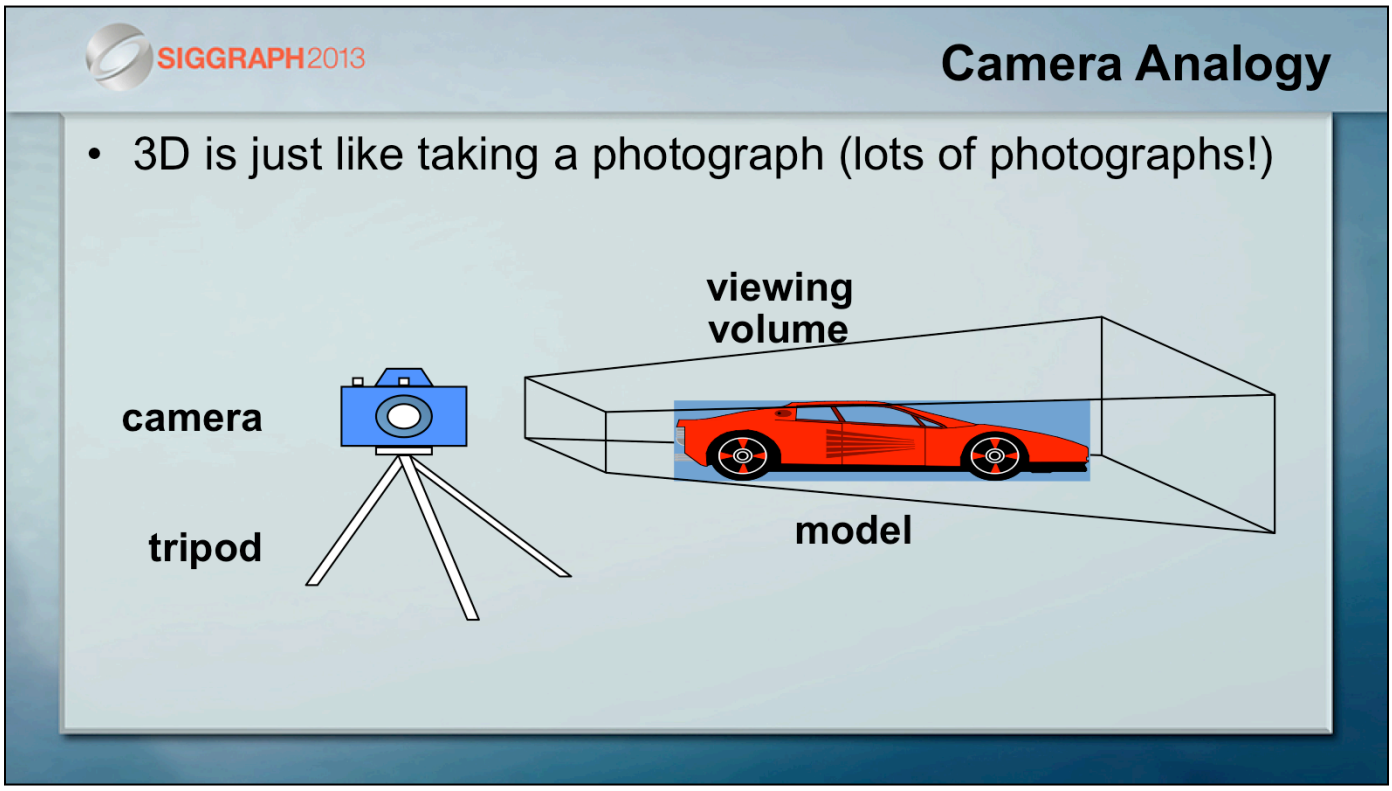


We begin delving into shader specifics by first taking a look at vertex shaders. As you've probably arrived at, vertex shaders are used to process vertices, and have the required responsibility of specifying the vertex's position in clip coordinates. This process usually involves numerous vertex transformations, which we'll discuss next. Additionally, a vertex shader may be responsible for determine additional information about a vertex for use by the rasterizer, including specifying colors.

To begin our discussion of vertex transformations, we'll first describe the *synthetic camera model*.

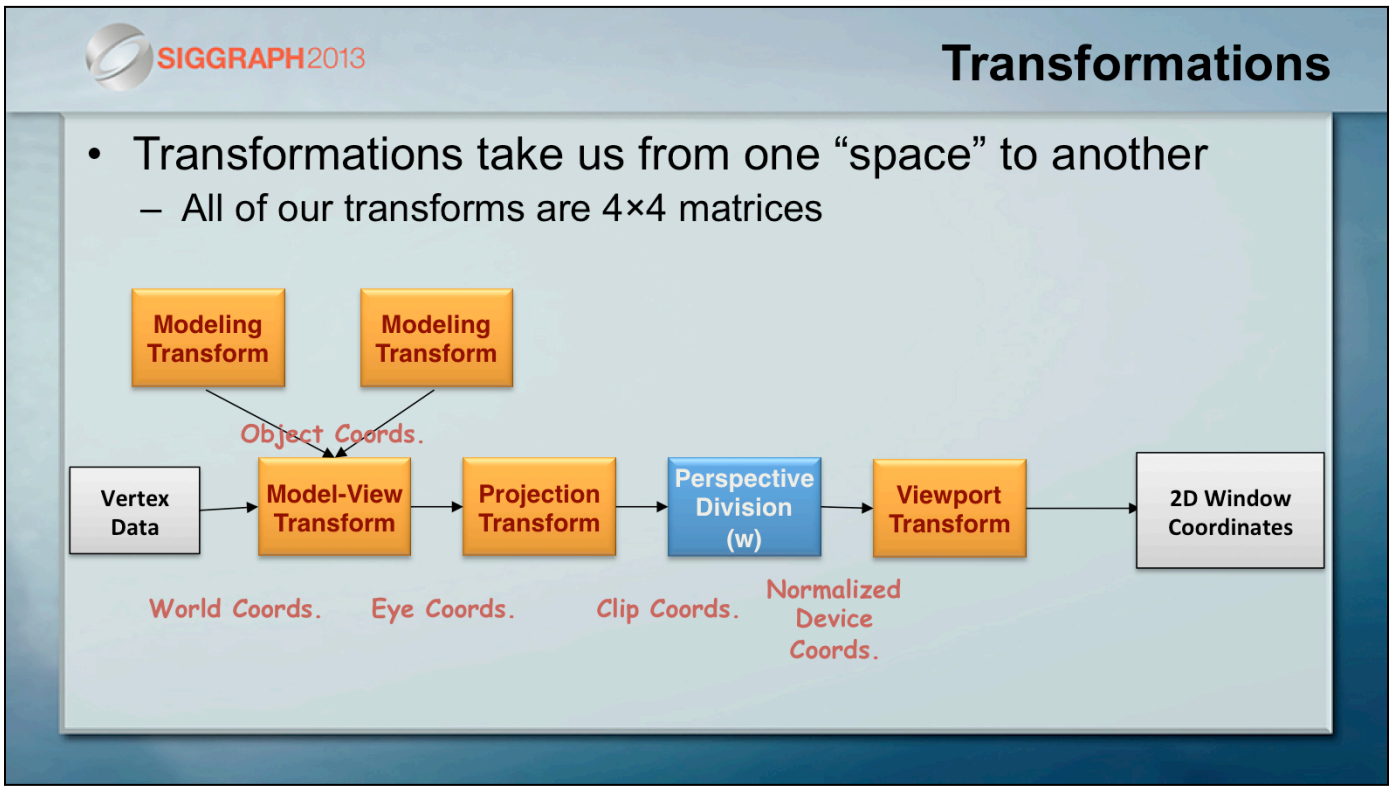
Transformations

SIGGRAPH 2013
The 40th International Conference and Exhibition
on Computer Graphics and Interactive Techniques



This model has become know as the synthetic camera model.

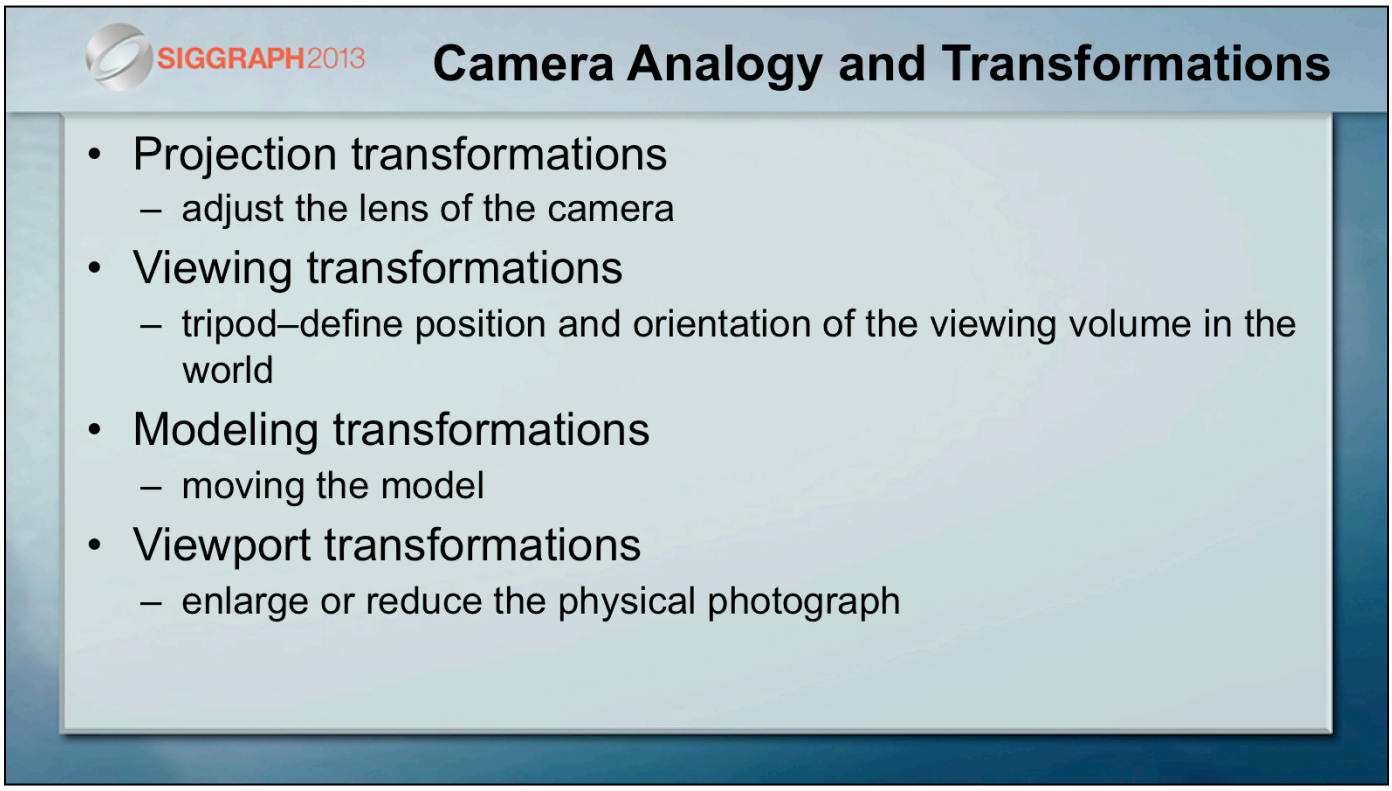
Note that both the objects to be viewed and the camera are three-dimensional while the resulting image is two dimensional.



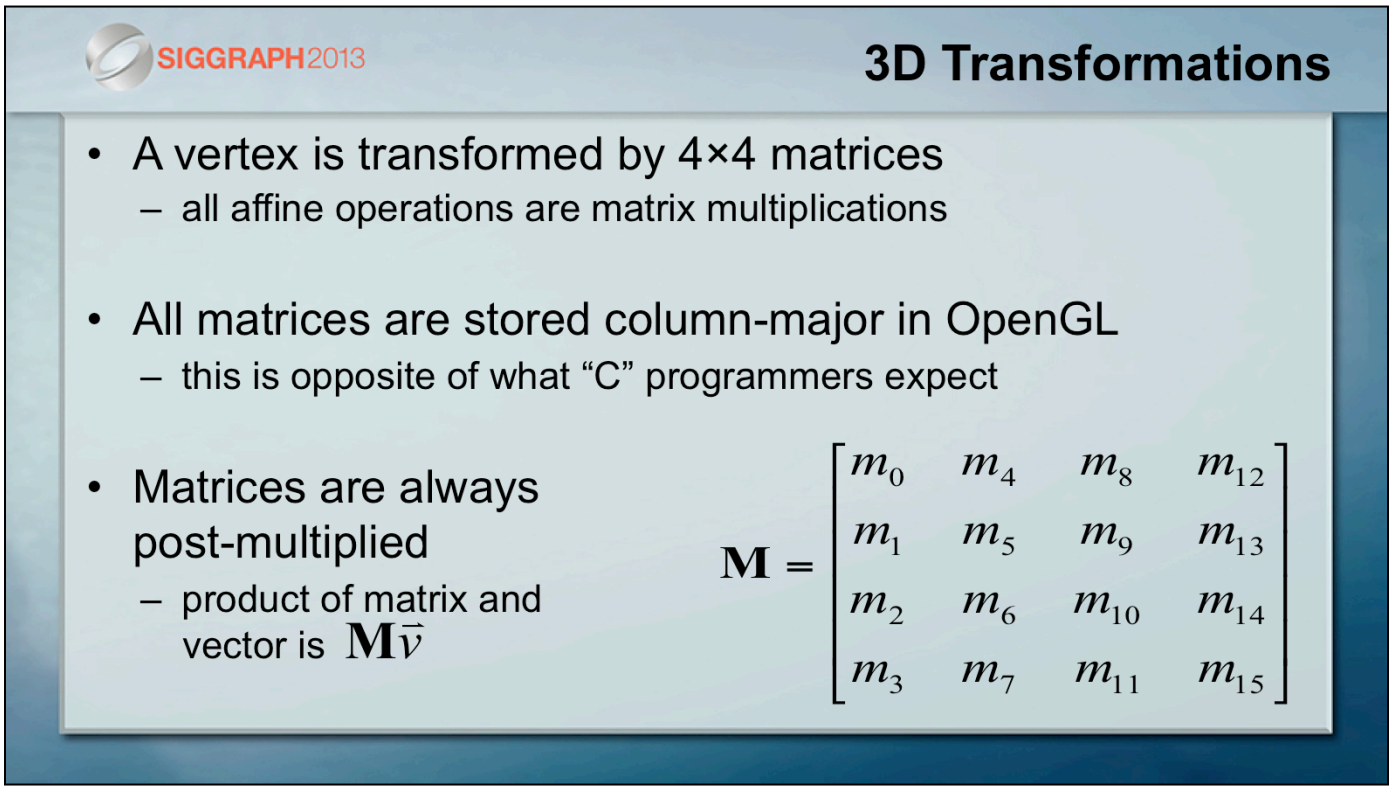
The processing required for converting a vertex from 3D or 4D space into a 2D window coordinate is done by the transform stage of the graphics pipeline. The operations in that stage are illustrated above. The purple boxes represent a matrix multiplication operation. In graphics, all of our matrices are 4×4 matrices (they're homogenous, hence the reason for homogenous coordinates).

When we want to draw an geometric object, like a chair for instance, we first determine all of the vertices that we want to associate with the chair. Next, we determine how those vertices should be grouped to form geometric primitives, and the order we're going to send them to the graphics subsystem. This process is called *modeling*. Quite often, we'll model an object in its own little 3D coordinate system. When we want to add that object into the scene we're developing, we need to determine its *world coordinates.* We do this by specifying a *modeling transformation*, which tells the system how to move from one coordinate system to another.

Modeling transformations, in combination with *viewing* transforms, which dictate where the viewing frustum is in world coordinates, are the first transformation that a vertex goes through. Next, the *projection transform* is applied which maps the vertex into another space called *clip coordinates,* which is where clipping occurs. After clipping, we divide by the *w* value of the



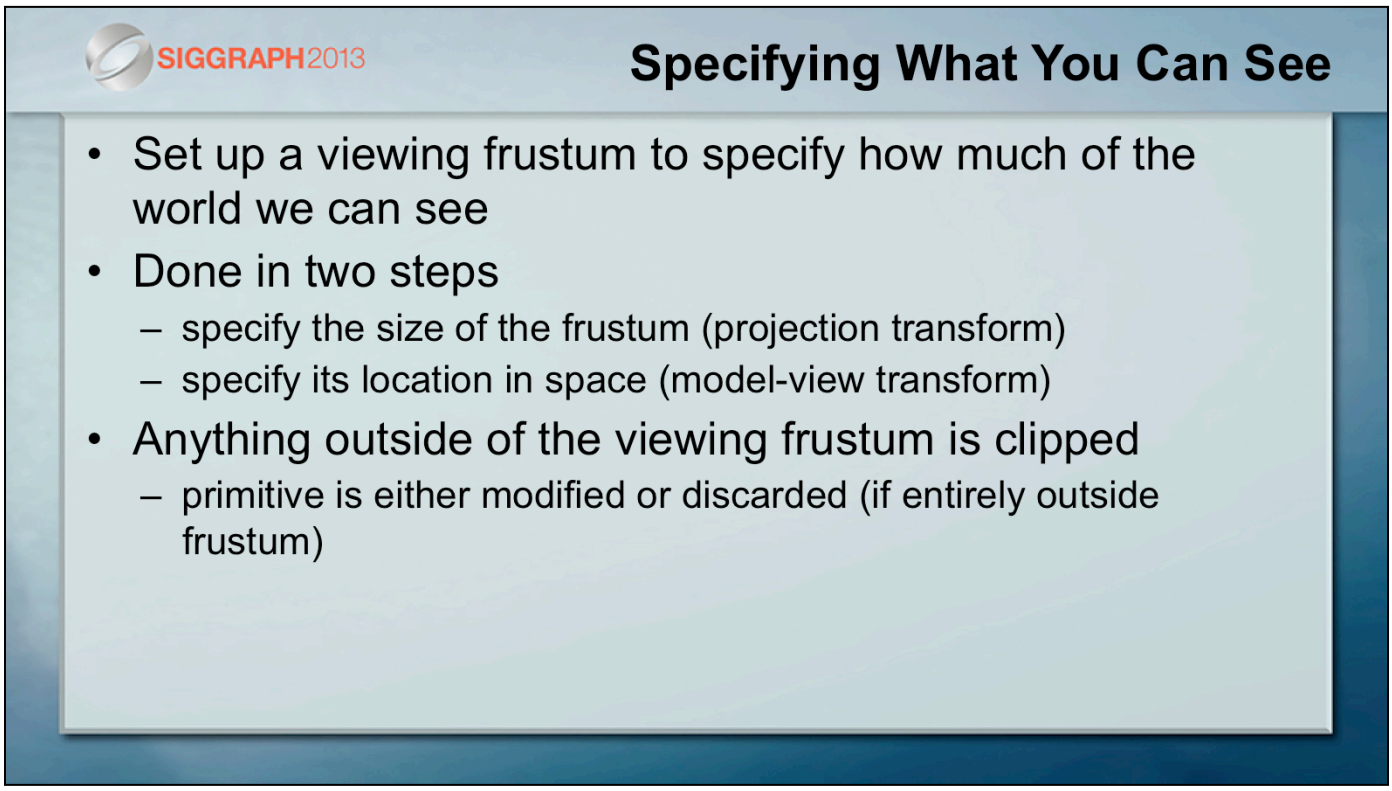
Note that human vision and a camera lens have cone-shaped viewing volumes. OpenGL (and almost all computer graphics APIs) describe a pyramid-shaped viewing volume. Therefore, the computer will “see” differently from the natural viewpoints, especially along the edges of viewing volumes. This is particularly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



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

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

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

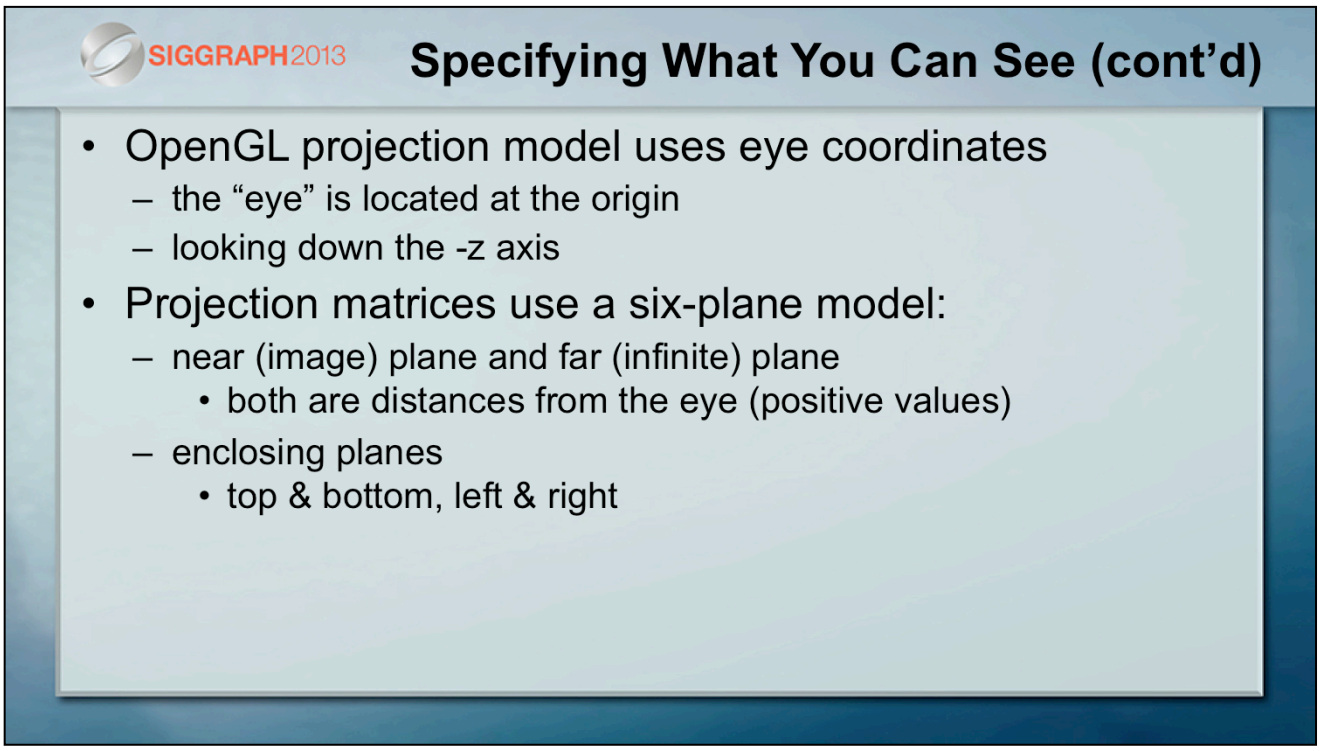


Another essential part of the graphics processing is setting up how much of the world we can see. We construct a *viewing frustum*, which defines the chunk of 3-space that we can see. There are two types of views: a *perspective view*, which you're familiar with as it's how your eye works, is used to generate frames that match your view of reality–things farther from your appear smaller. This is the type of view used for video games, simulations, and most graphics applications in general.

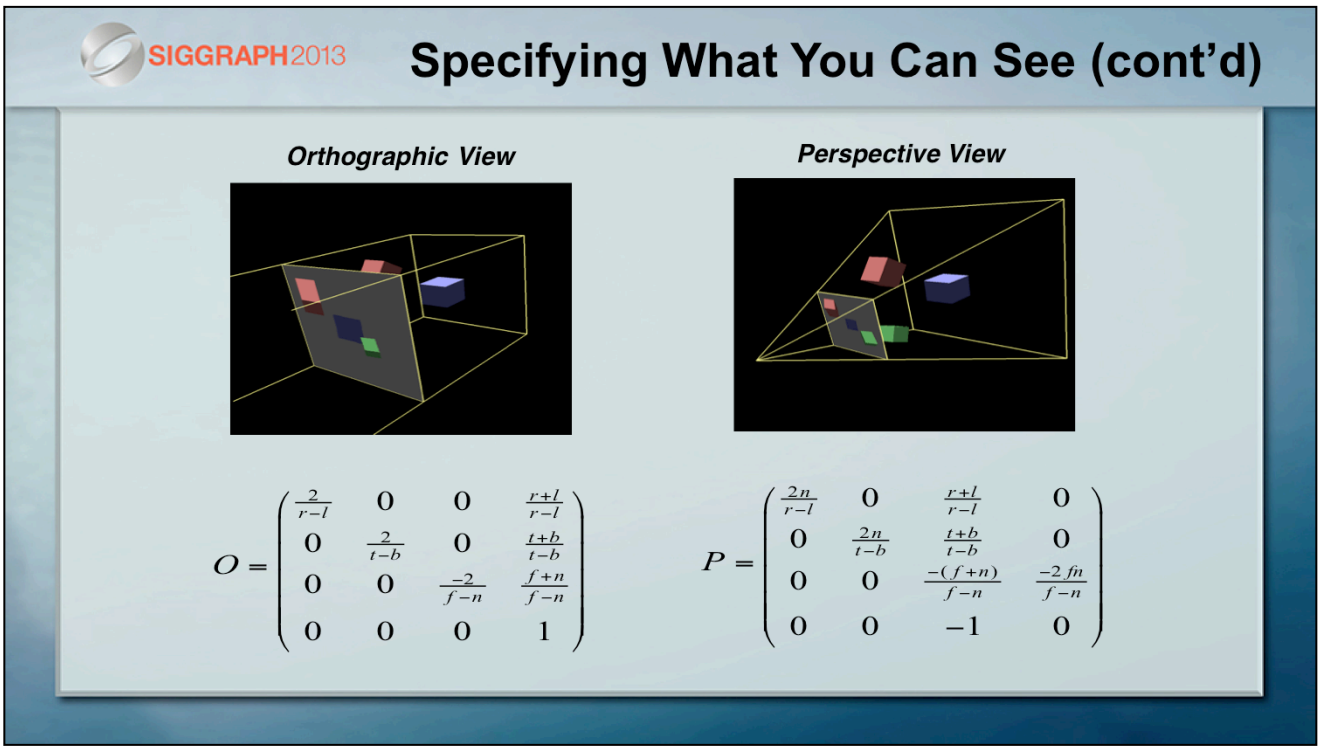
The other view, *orthographic*, is used principally for engineering and design situations, where relative lengths and angles need to be preserved.

For a perspective, we locate the eye at the apex of the frustum pyramid. We can see any objects which are between the two planes perpendicular to eye (they're called the *near* and *far* clipping planes, respectively). Any vertices between near and far, and inside the four planes that connect them will be rendered. Otherwise, those vertices are *clipped* out and discarded. In some cases a primitive will be entirely outside of the view, and the system will discard it for that frame. Other primitives might intersect the frustum, which we *clip* such that the part of them that's outside is discarded and we create new vertices for the modified primitive.

While the system can easily determine which primitive are inside the frustum, it's wasteful of system bandwidth to have lots of primitives discarded in this manner. We utilize a technique named *culling* to determine exactly which primitives need to be sent to the graphics processor, and send only those primitives to maximize its efficiency.



In OpenGL, the default viewing frusta are always configured in the same manner, which defines the orientation of our clip coordinates. Specifically, clip coordinates are defined with the "eye" located at the origin, looking down the –z axis. From there, we define two distances: our *near* and *far clip distances*, which specify the location of our near and far clipping planes. The viewing volume is then completely by specifying the positions of the enclosing planes that are parallel to the view direction .



The images above show the two types of projection transformations that are commonly used in computer graphics. The *orthographic view* preserves angles, and simulates having the viewer at an infinite distance from the scene. This mode is commonly used in used in engineering and design where it's important to preserve the sizes and angles of objects in relation to each other. Alternatively, the *perspective view* mimics the operation of the eye with objects seeming to shrink in size the farther from the viewer they are.

The each projection, the matrix that you would need to specify is provided. In those matrices, the six values for the positions of the left, right, bottom, top, near and far clipping planes are specified by the first letter of the plane's name. The only limitations on the values is for perspective projections, where the near and far values must be positive and non-zero, with near greater than far.

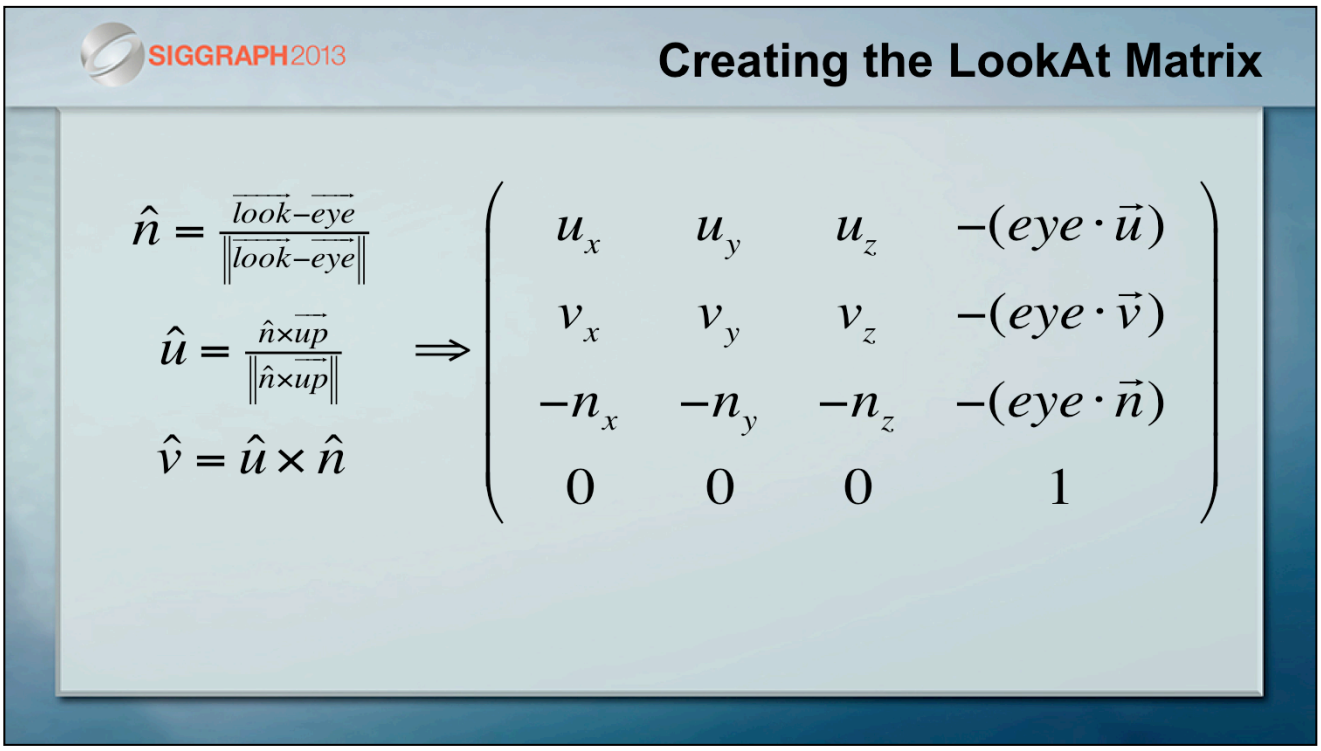
Viewing Transformations

- Position the camera/eye in the scene
  - place the tripod down; aim camera
- To "fly through" a scene
  - change viewing transformation and redraw scene
- `LookAt( eyex, eyey, eyez, lookx, looky, lookz, upx, upy, upz )`
  - up vector determines unique orientation
  - careful of degenerate positions

`LookAt()` generates a viewing matrix based on several points. `LookAt()` provides natrual semantics for modeling flight application, but care must be taken to avoid degenerate numerical situations, where the generated viewing matrix is undefined.

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.



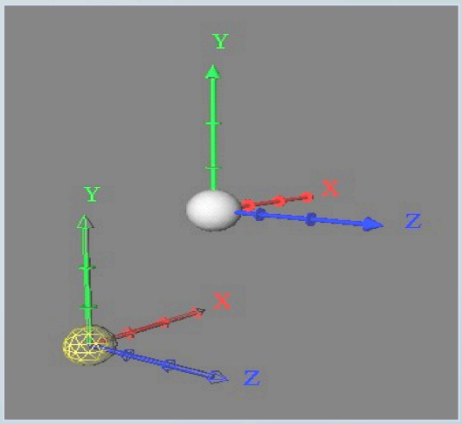
Using the values passed into the LookAt() call, the above matrix generates the corresponding viewing matrix.

SIGGRAPH2013

Translation

- Move the origin to a new location

$$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



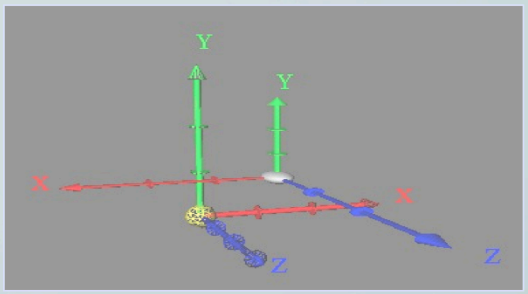
Here we show the construction of a translation matrix. Translations really move coordinate systems, and not individual objects.

SIGGRAPH2013

Scale

- Stretch, mirror or decimate a coordinate direction

$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



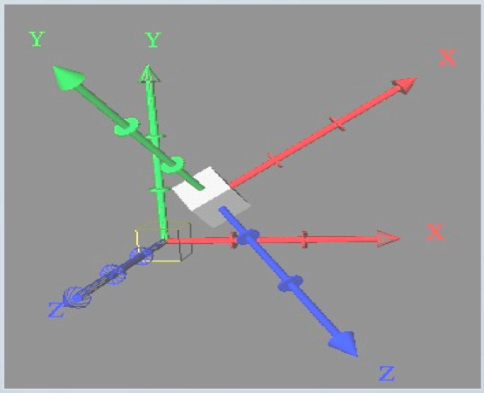
Note, there's a translation applied here to make things easier to see

Here we show the construction of a scale matrix, which is used to change the shape of space, but not move it (or more precisely, the origin). The above illustration has a translation to show how space was modified, but a simple scale matrix will not include such a translation.

SIGGRAPH2013

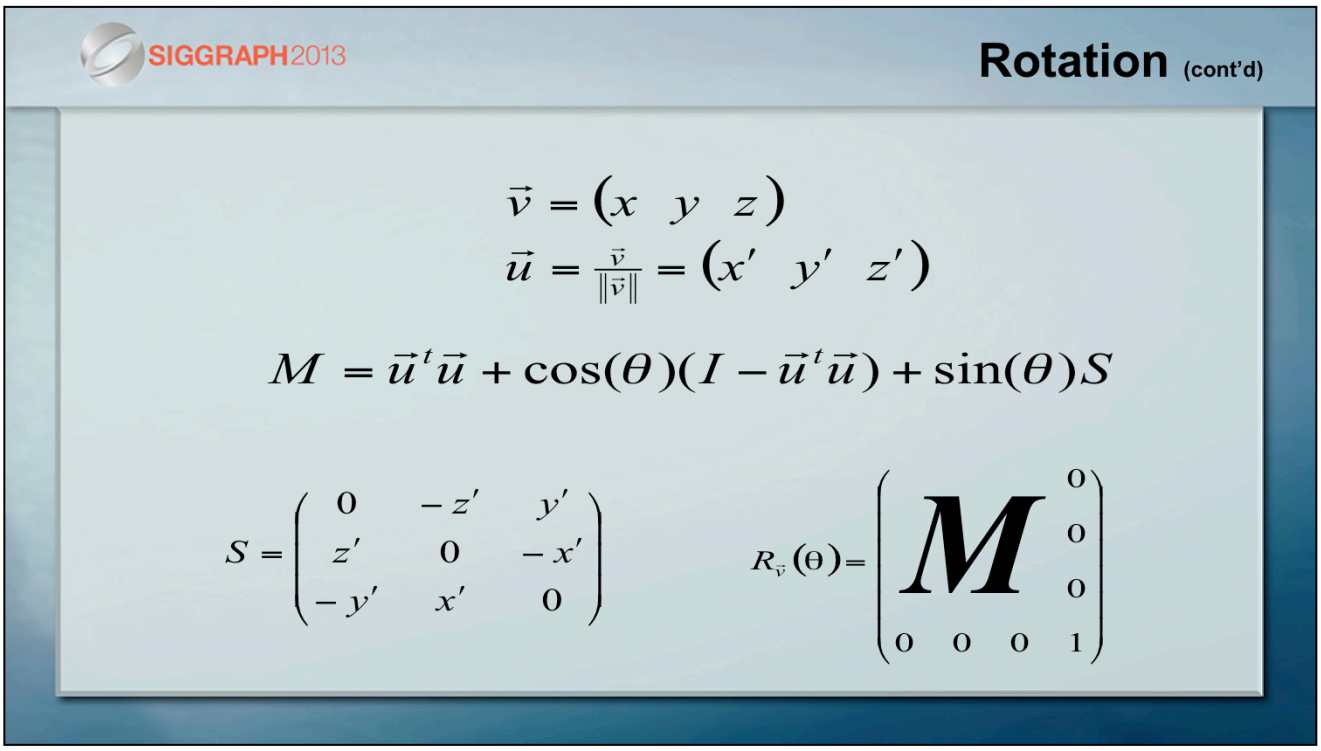
Rotation

- Rotate coordinate system about an axis in space



Note, there's a translation applied here to make things easier to see

Here we show the effects of a rotation matrix on space. Once again, a translation has been applied in the image to make it easier to see the rotation's affect.



The formula for generating a rotation matrix is a bit more complex that for scales and translations. Naming the axis of rotation v, we begin by normalizing v and storing the result in the vector u. From there, we create a 3 × 3 matrix M, which is composed of the sum of three terms.

1. The *outer product* of the vector u with its transpose u^t
2. The difference of the identity matrix, I, with u's outer product, scaled the by the cosine of the input angle θ
3. Finally, we scale the matrix S which is composed of the elements of the rotation matrix.

The complete rotation matrix is formed by composing M as the upper 3 × 3 matrix in R.

Vertex Shader for Rotation of Cube

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform vec3 theta;

void main()
{
    // Compute the sines and cosines of theta for
    // each of the three axes in one computation.
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
```

Here's an example vertex shader for rotating our cube. We generate the matrices in the shader (as compared to in the application), based on the input angle `theta`. It's useful to note that we can vectorize numerous computations. For example, we can generate a vectors of sines and cosines for the input angle, which we'll use in further computations.

Vertex Shader for Rotation of Cube (cont'd)

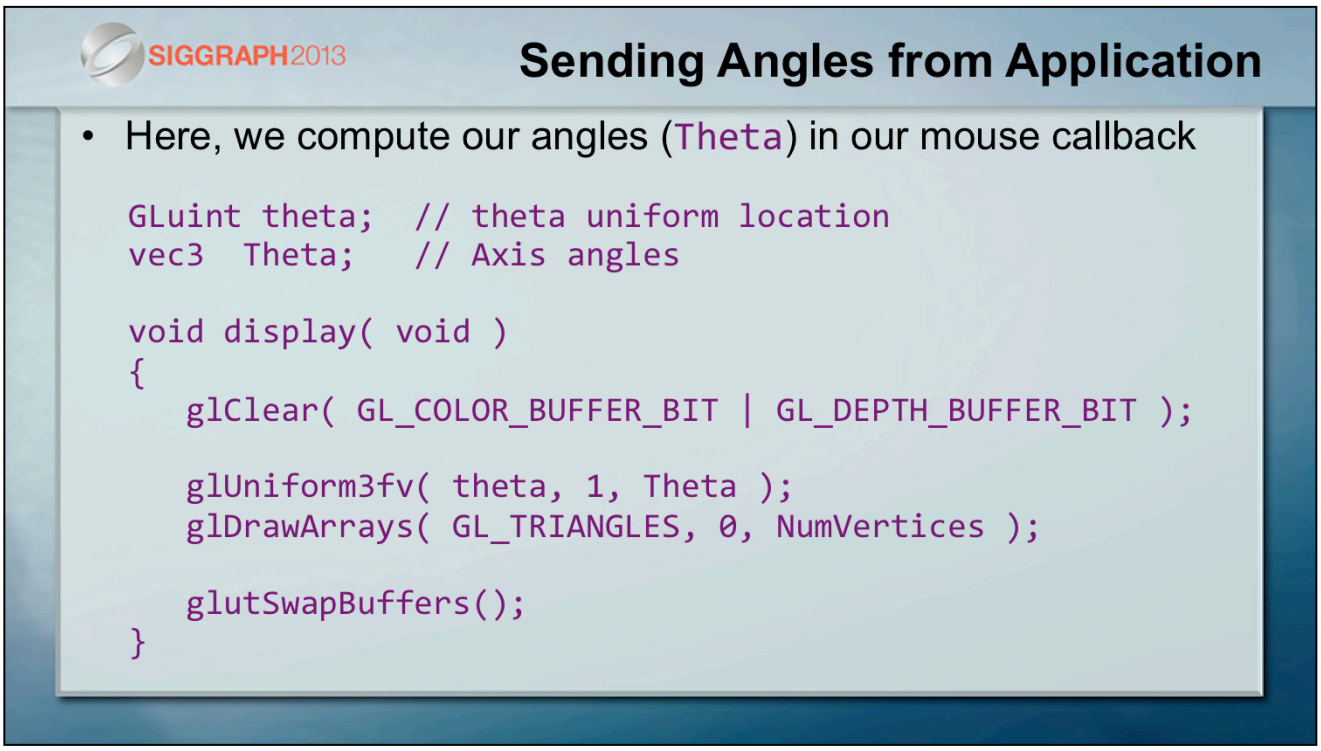
```
// Remember: these matrices are column-major

mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                0.0,  c.x,  s.x, 0.0,
                0.0, -s.x,  c.x, 0.0,
                0.0,  0.0,  0.0, 1.0 );

mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                0.0, 1.0,  0.0, 0.0,
                s.y, 0.0,  c.y, 0.0,
                0.0, 0.0,  0.0, 1.0 );
```

Completing our shader, we compose two of three rotation matrices (one around each axis). In generating our matrices, we use one of the many matrix constructor functions (in this case, specifying the 16 individual elements). It's important to note in this case, that our matrices are column-major, so we need to take care in the placement of the values in the constructor.

Vertex Shader for Rotation of Cube (cont'd)

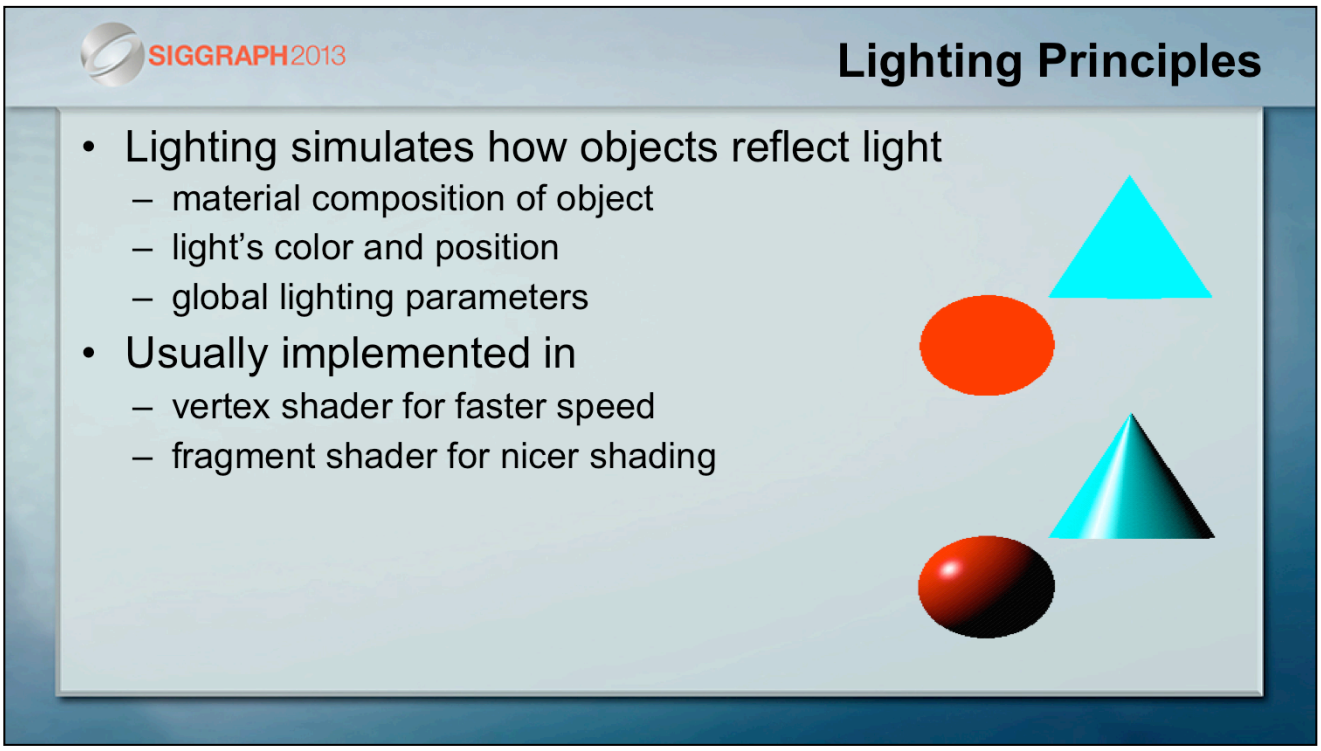
```
    mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,
                    s.z,  c.z, 0.0, 0.0,
                    0.0,  0.0, 1.0, 0.0,
                    0.0,  0.0, 0.0, 1.0 );

    color = vColor;
    gl_Position = rz * ry * rx * vPosition;
}
```

We complete our shader here by generating the last rotation matrix, and) and then use the composition of those matrices to transform the input vertex position. We also *pass-thru* the color values by assigning the input color to an output variable.



Finally, we merely need to supply the angle values into our shader through our uniform plumbing. In this case, we track each of the axes rotation angle, and store them in a `vec3` that matches the angle declaration in the shader. We also keep track of the uniform's location so we can easily update its value.

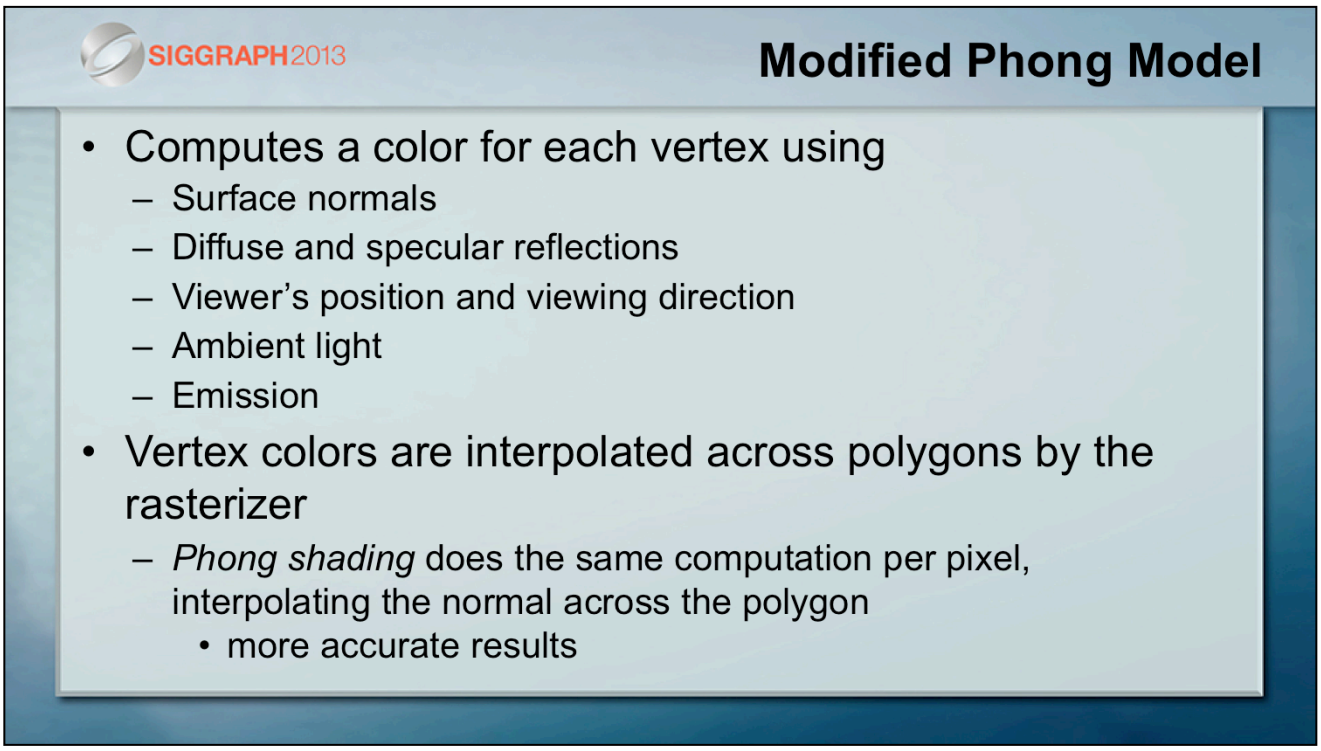

Lighting
SIGGRAPH2013
The 40th International Conference and Exhibition
on Computer Graphics and Interactive Techniques



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

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

While we'll discuss the mathematics of lighting in terms of computing illumination in a vertex shader, the almost identical computations can be done in a fragment shader to compute the lighting effects per-pixel, which yields much better results.



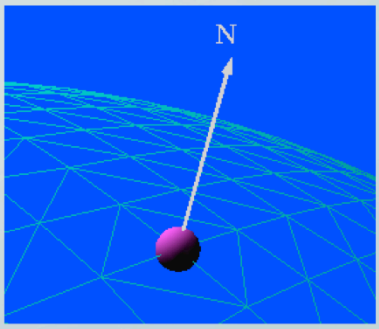
OpenGL can use the shade at one vertex to shade an entire polygon (constant shading) or interpolate the shades at the vertices across the polygon (smooth shading), the default.

The original lighting model that was supported in hardware and OpenGL was due to Phong and later modified by Blinn.

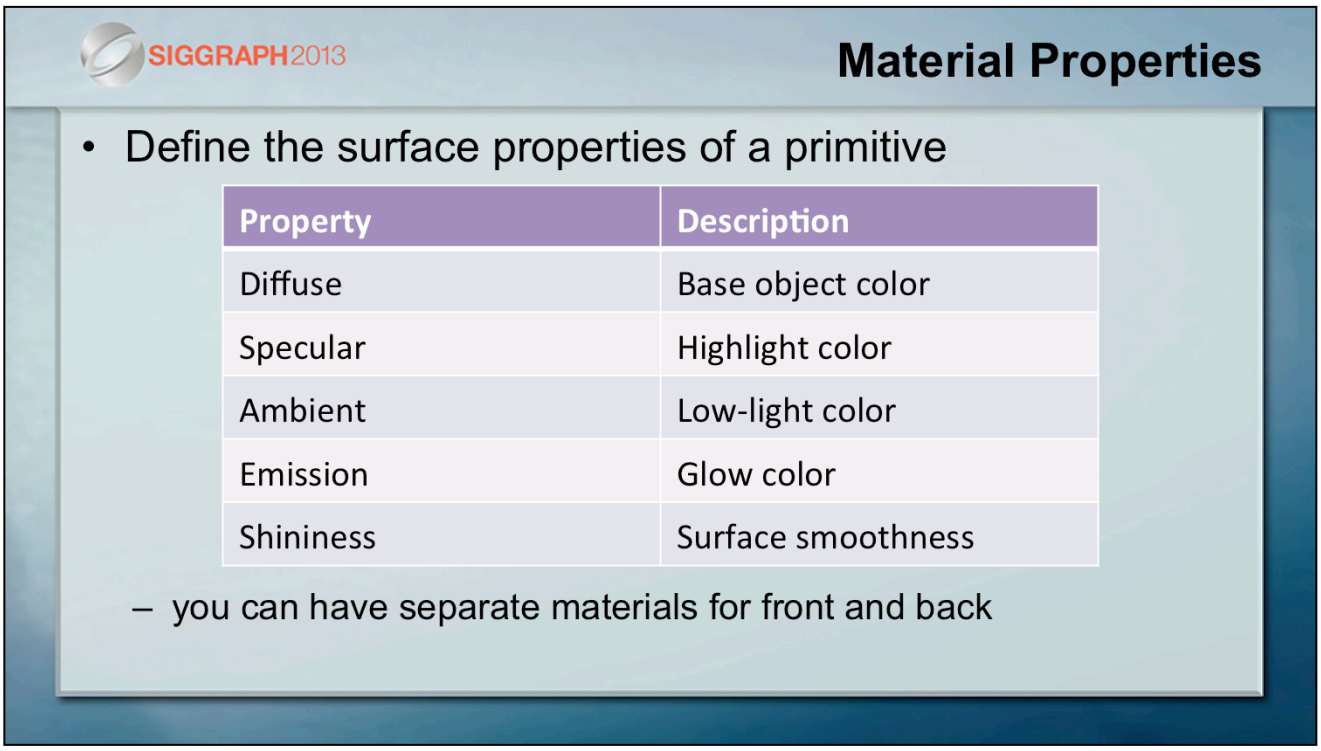
SIGGRAPH2013

Surface Normals

- Normals define how a surface reflects light
  - Application usually provides normals as a vertex atttribute
  - Current normal is used to compute vertex's color
  - Use unit normals for proper lighting
    - scaling affects a normal's length



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.



Material properties describe the color and surface properties of a material (dull, shiny, etc). The properties described above are components of the Phong lighting model, a simple model that yields reasonable results with little computation. Each of the material components would be passed into a vertex shader, for example, to be used in the lighting computation along with the vertex's position and lighting normal.

Adding Lighting to Cube

```
// vertex shader

in vec4 vPosition;
in vec3 vNormal;
out vec4 color;

uniform vec4
    AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;
```

Here we declare numerous variables that we'll use in computing a color using a simple lighting model. All of the uniform values are passed in from the application and describe the material and light properties being rendered.

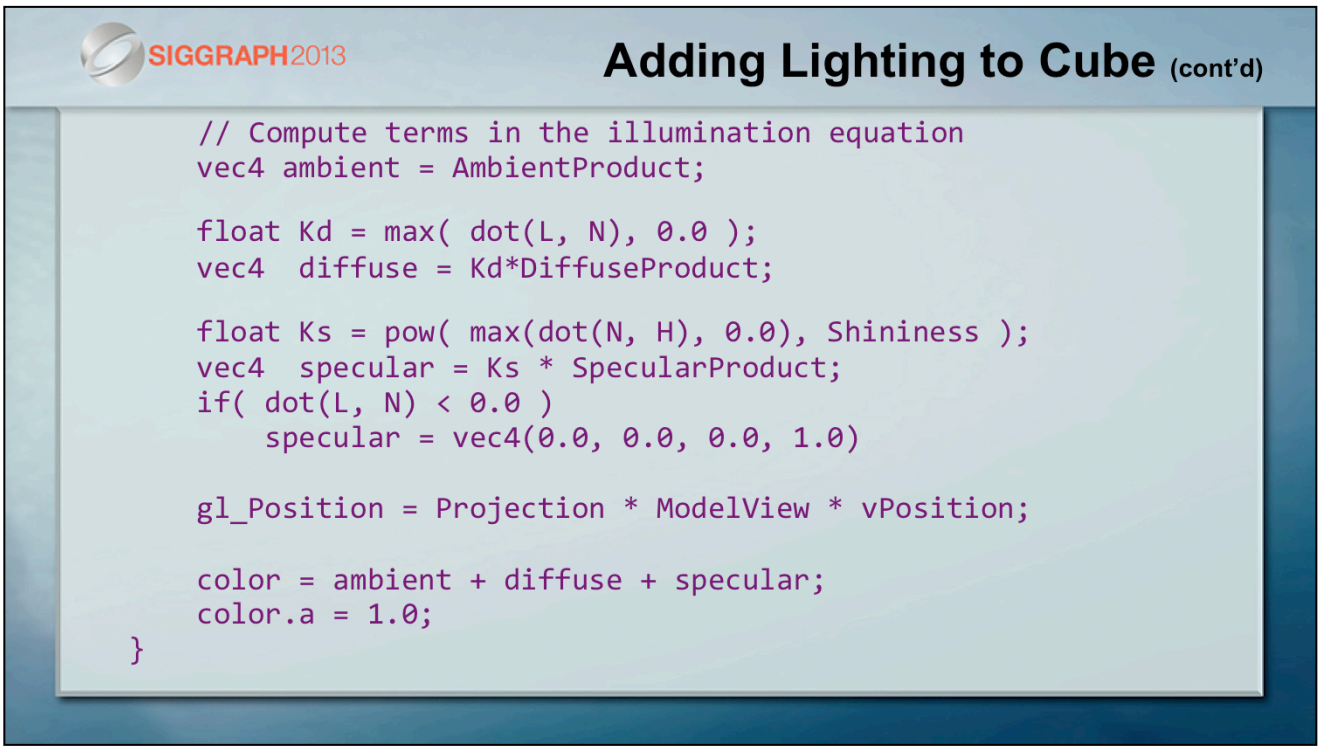
Adding Lighting to Cube (cont'd)

```
void main()
{
    // Transform vertex  position into eye coordinates
    vec3 pos = vec3(ModelView * vPosition);

    vec3 L = normalize(LightPosition.xyz - pos);
    vec3 E = normalize(-pos);
    vec3 H = normalize(L + E);

    // Transform vertex normal into eye coordinates
    vec3 N = normalize(vec3(ModelView * vNormal));
```

In the initial parts of our shader, we generate numerous vector quantities to be used in our lighting computation.

- pos represents the vertex's position in eye coordinates
- L represents the vector from the vertex to the light
- E represents the "eye" vector, which is the vector from the vertex's eye-space position to the origin
- H is the "half vector" which is the normalized vector half-way between the light and eye vectors
- N is the transformed vertex normal

Note that all of these quantities are vec3's, since we're dealing with vectors, as compared to homogenous coordinates. When we need to convert form a homogenous coordinate to a vector, we use a vector swizzle to extract the components we need.



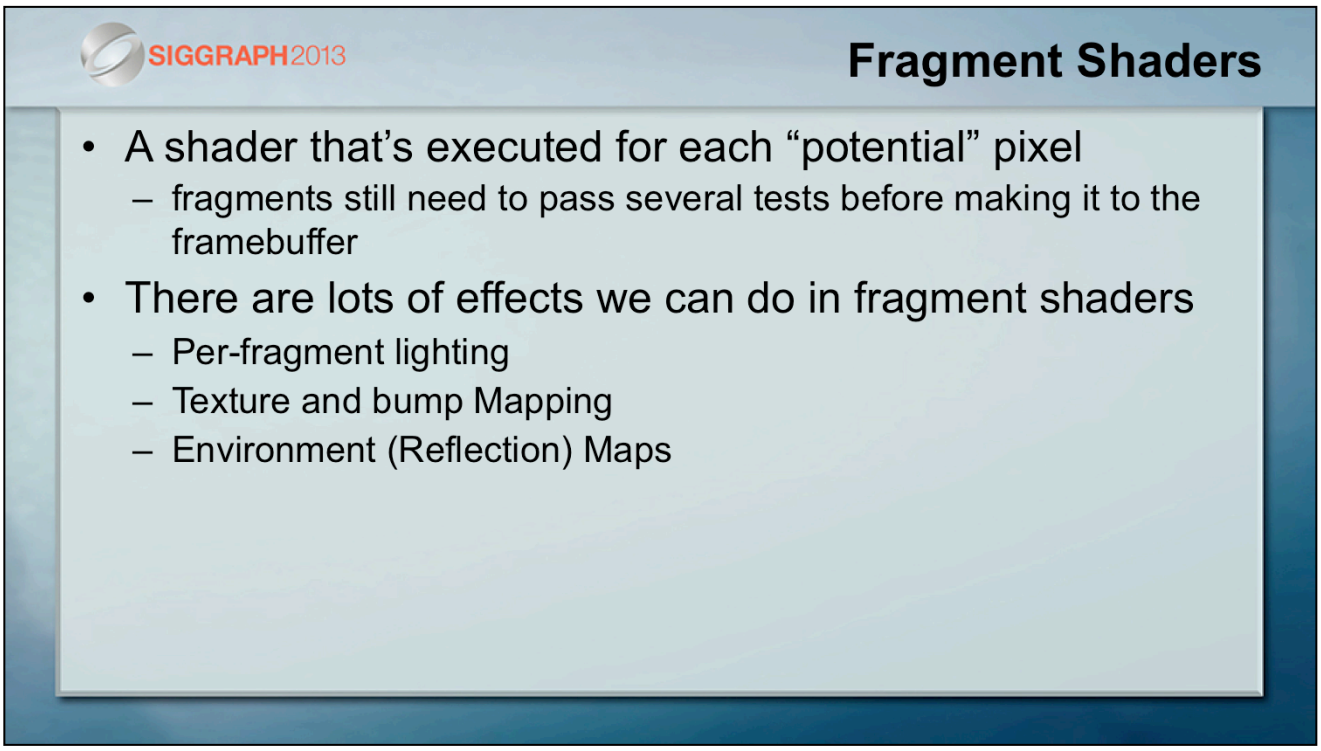
Here we complete our lighting computation. The Phong model, which this shader is based on, uses various material properties as we described before. Likewise, each light can contribute to those same properties. The combination of the material and light properties are represented as our "product" variables in this shader. The products are merely the component-wise products of the light and objects same material propreties. These values are computed in the application and passed into the shader.

In the Phong model, each material product is attenuated by the magnitude of the various vector products. Starting with the most influential component of lighting, the diffuse color, we use the dot product of the lighting normal and light vector, clamping the value if the dot product is negative (which physically means the light's behind the object). We continue by computing the specular component, which is computed as the dot product of the normal and the half-vector raised to the shininess value. Finally, if the light is behind the object, we correct the specular contribution.

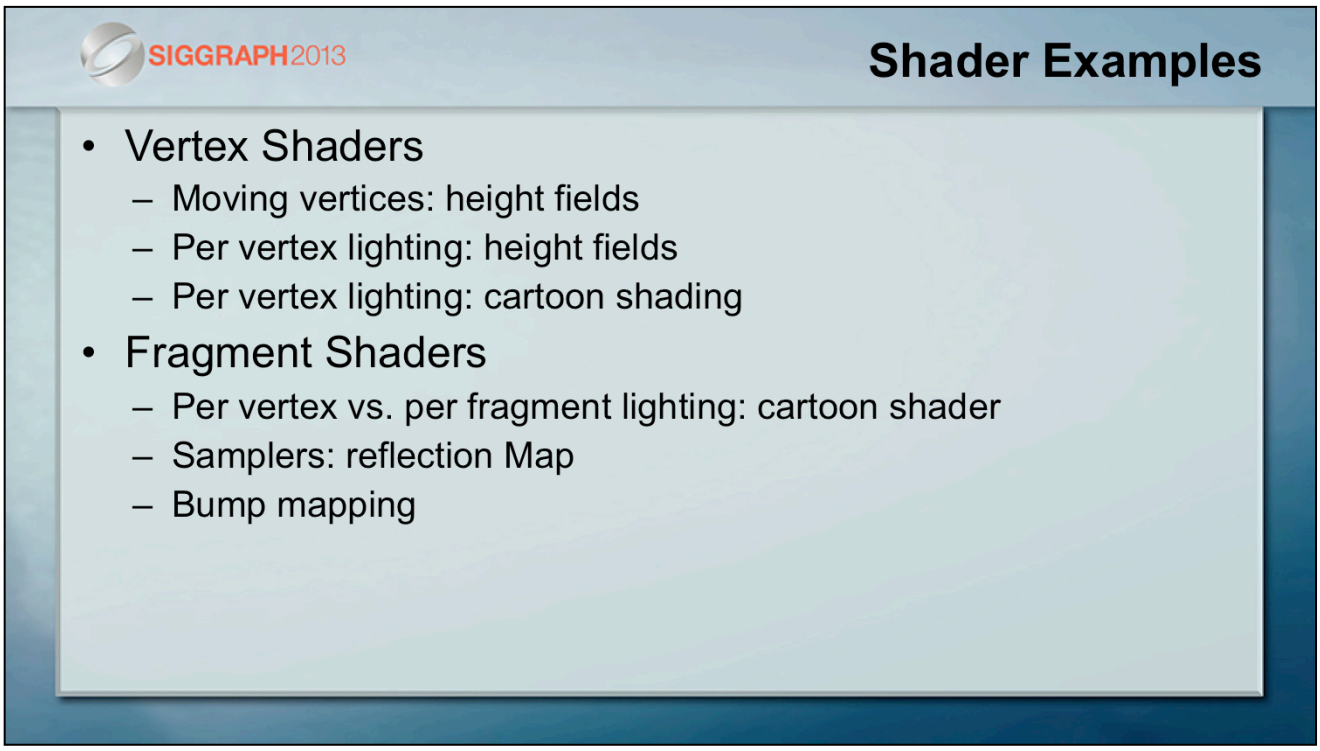
Finally, we compose the final vertex color as the sum of the computed ambient, diffuse, and specular colors, and update the transformed vertex position.

Fragment Shaders

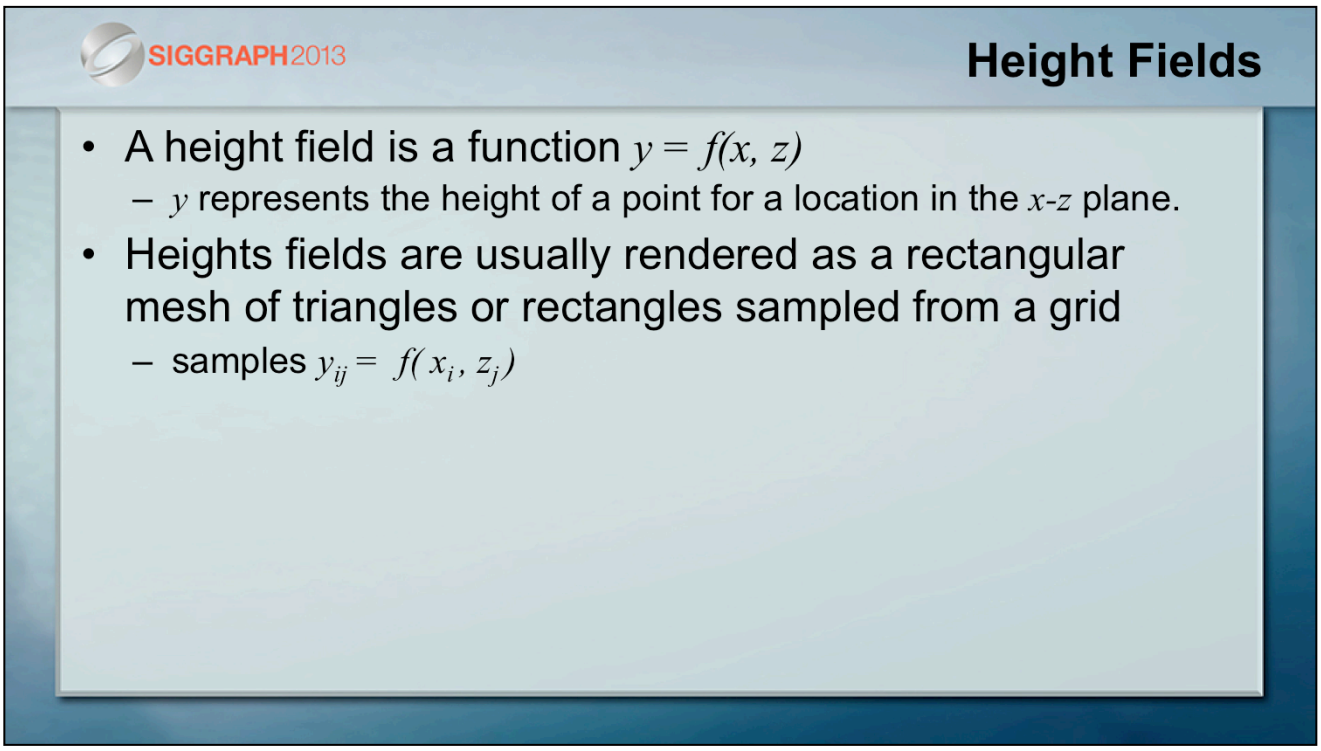
SIGGRAPH 2013
The 40th International Conference and Exhibition
on Computer Graphics and Interactive Techniques



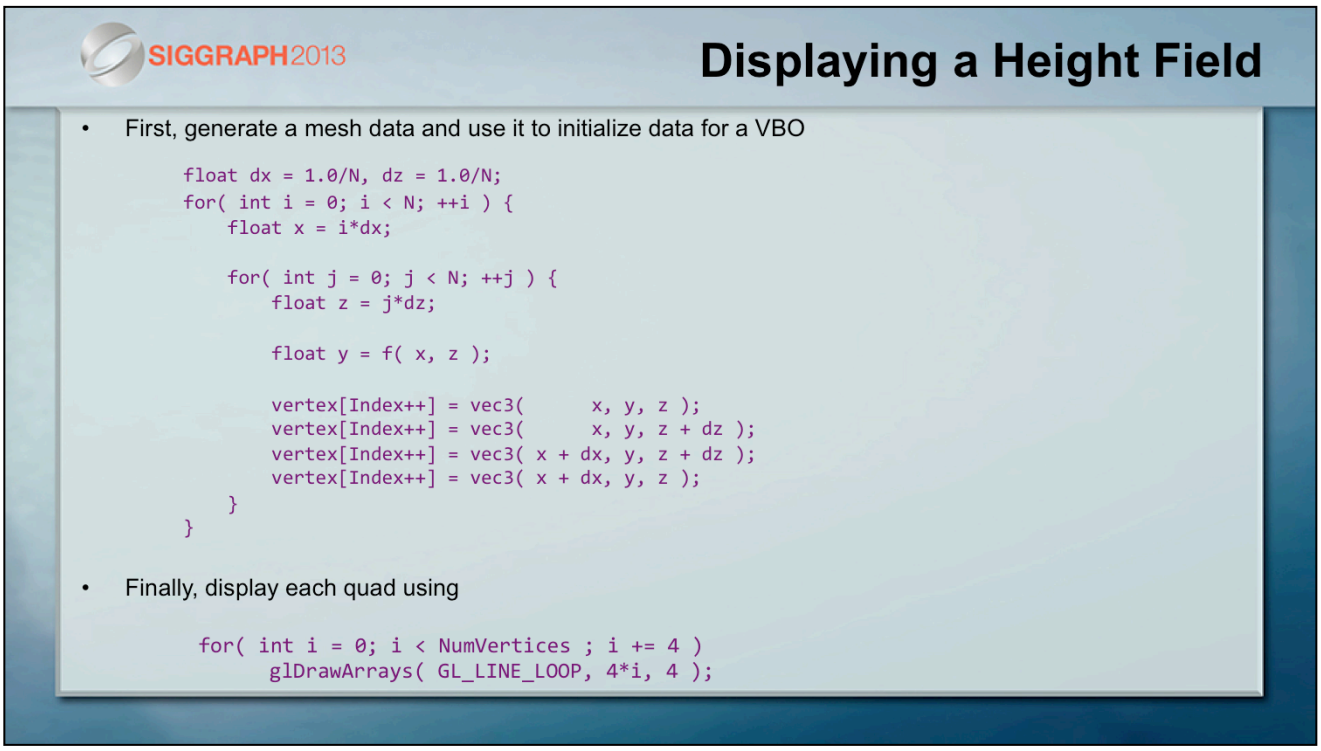
The final shading stage that OpenGL supports is *fragment shading* which allows an application per-pixel-location control over the color that may be written to that location. Fragments, which are on their way to the framebuffer, but still need to do some pass some additional processing to become pixels. However, the computational power available in shading fragments is a great asset to generating images. In a fragment shader, you can compute lighting values – similar to what we just discussed in vertex shading – per fragment, which gives much better results, or add bump mapping, which provides the illusion of greater surface detail. Likewise, we'll apply texture maps, which allow us to increase the detail for our models without increasing the geometric complexity.



We'll now analyze a few case studies from different applications.



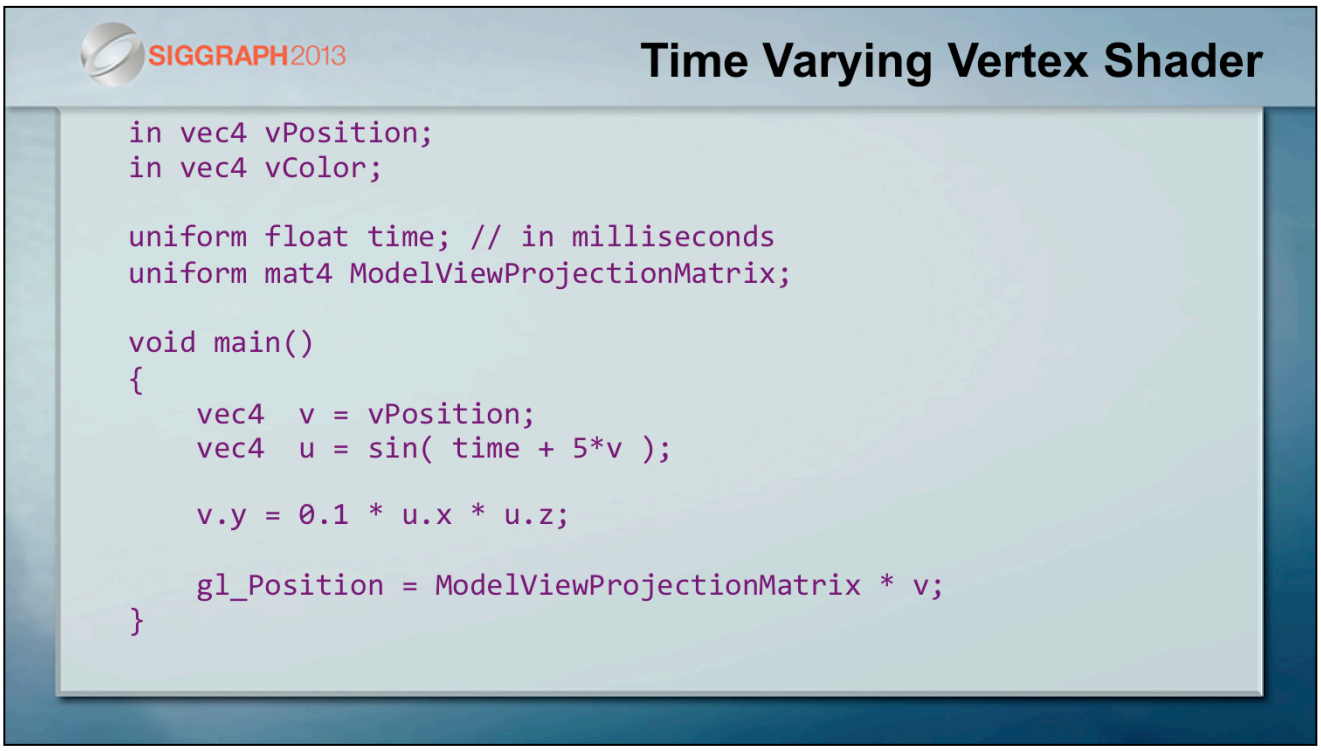
The first simple application we'll look at is rendering height fields, as you might do when rendering terrain in an outdoor game or flight simulator.

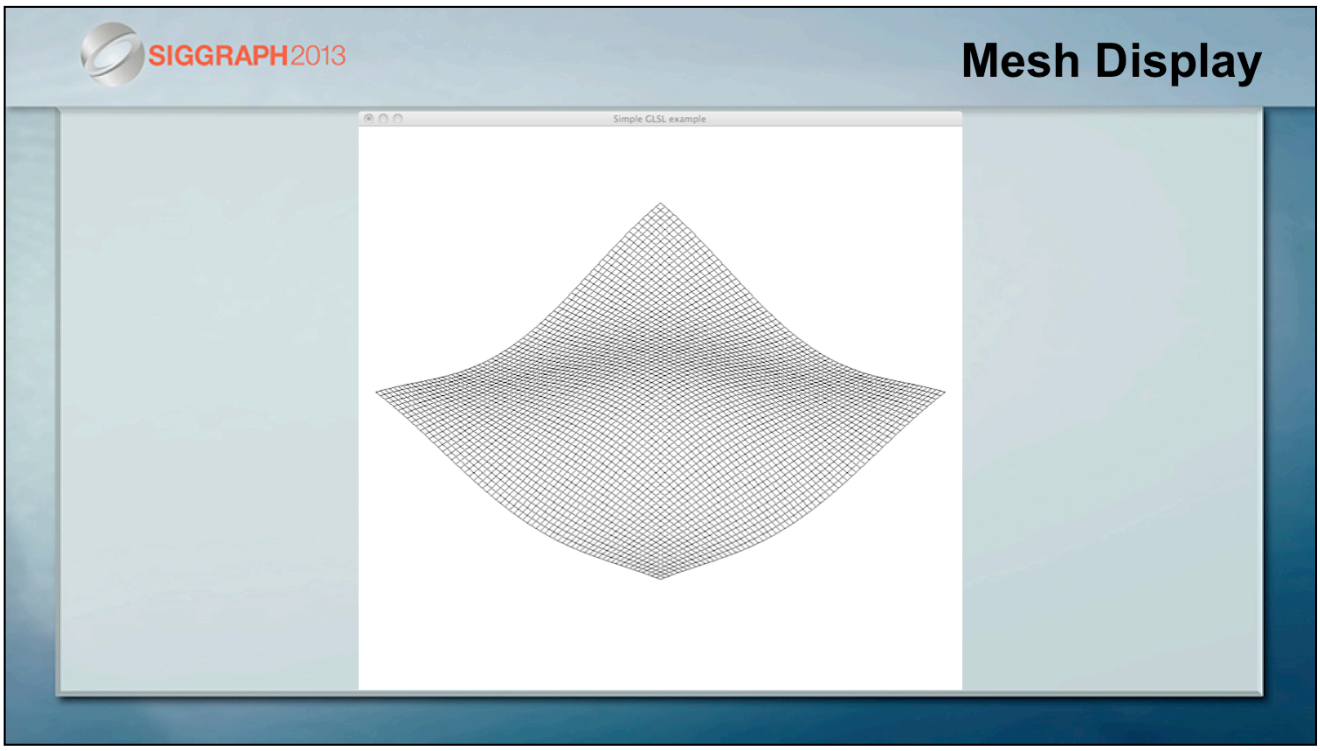


We'd first like to render a wire-frame version of our mesh, which we'll draw a individual line loops.

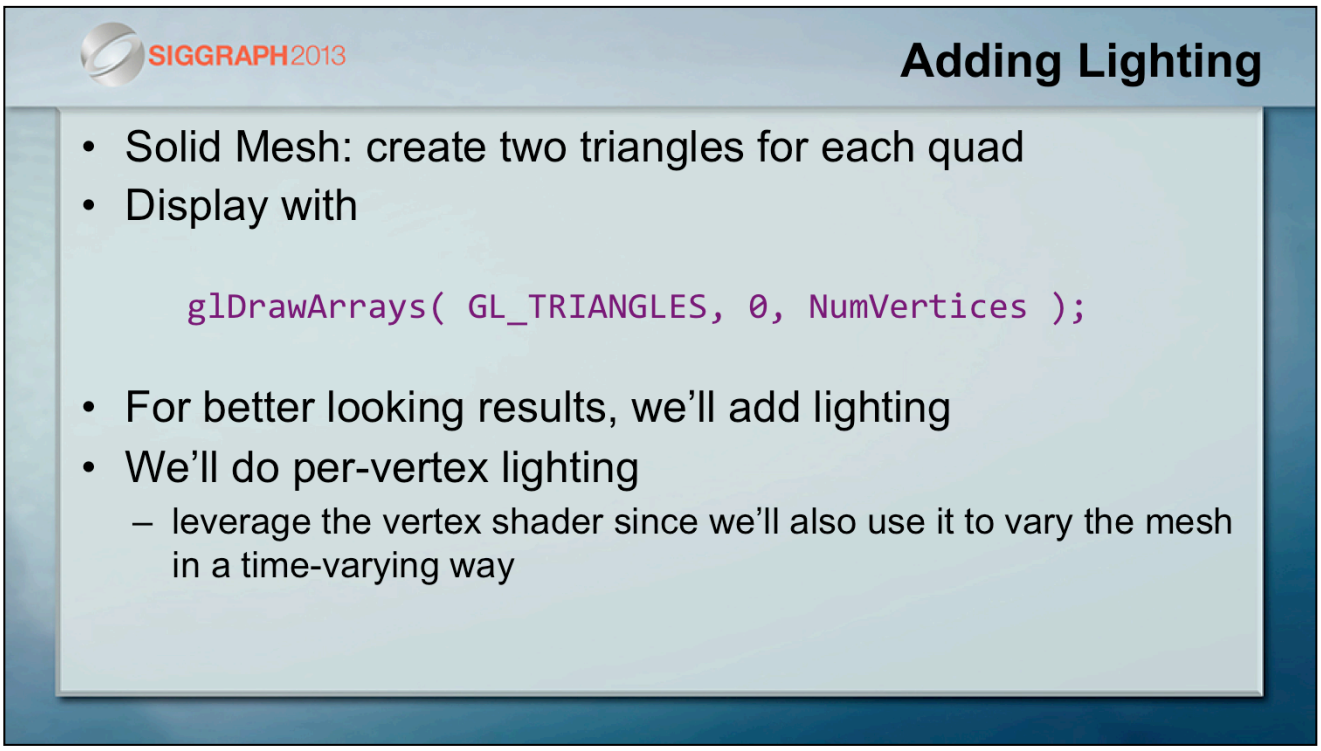
To begin, we build our data set by sampling the function f for a particular time across the domain of points. From there, we build our array of points to render. Once we have our data and have loaded into our VBOs we render it by drawing the individual wireframe quadrilaterals.

There are many ways to render a wireframe surface like this – give some thought of other methods.


SIGGRAPH2013
Time Varying Vertex Shader
in vec4 vPosition;
in vec4 vColor;
uniform float time; // in milliseconds
uniform mat4 ModelViewProjectionMatrix;
void main()
{
vec4 v = vPosition;
vec4 u = sin(time + 5*v);
v.y = 0.1 * u.x * u.z;
gl_Position = ModelViewProjectionMatrix * v;
}

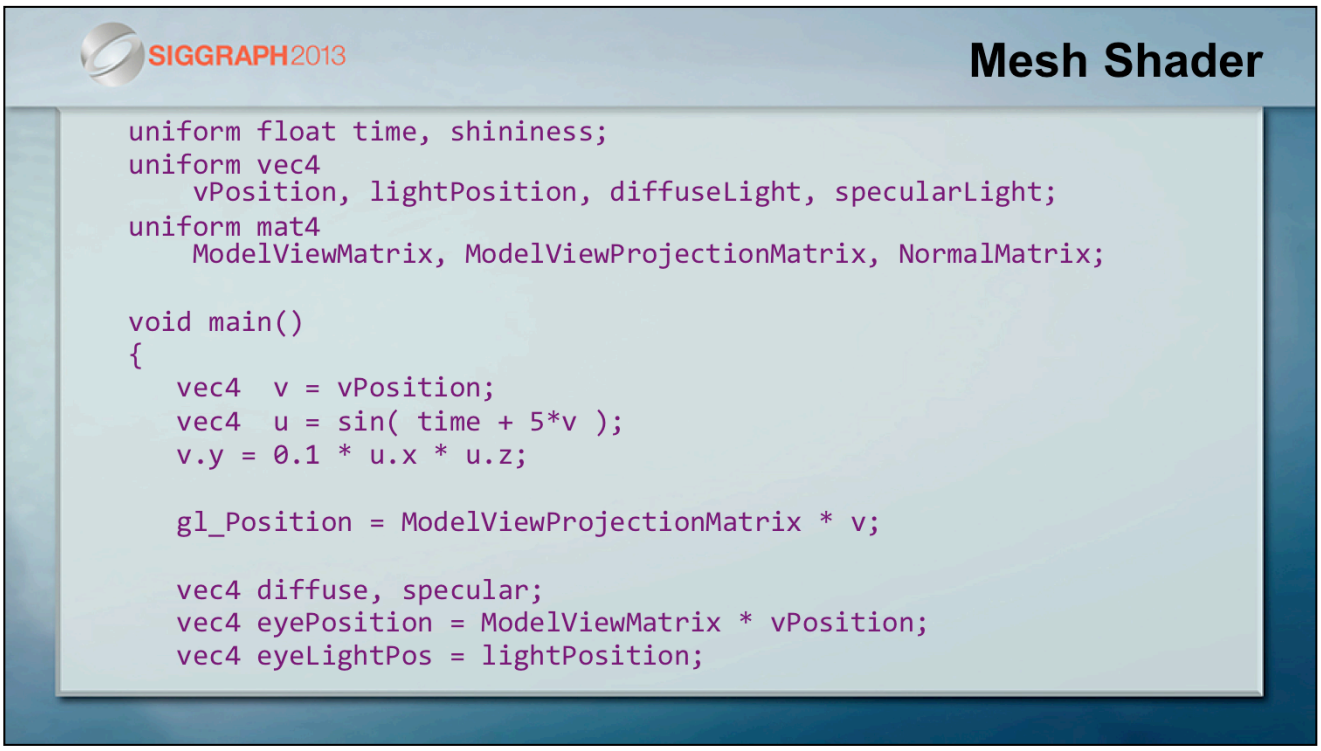


Here's a rendering of the mesh we just generated.



While the wireframe version is of some interest, we can create better looking meshes by adding a few more effects. We'll begin by creating a solid mesh by converting each wireframe quadrilateral into a solid quad composed of two separate triangles. Turns out with our pervious set of points, we can merely changed our `glDrawArrays()` call – or more specifically, the geometric primitive type – to render a solid surface.

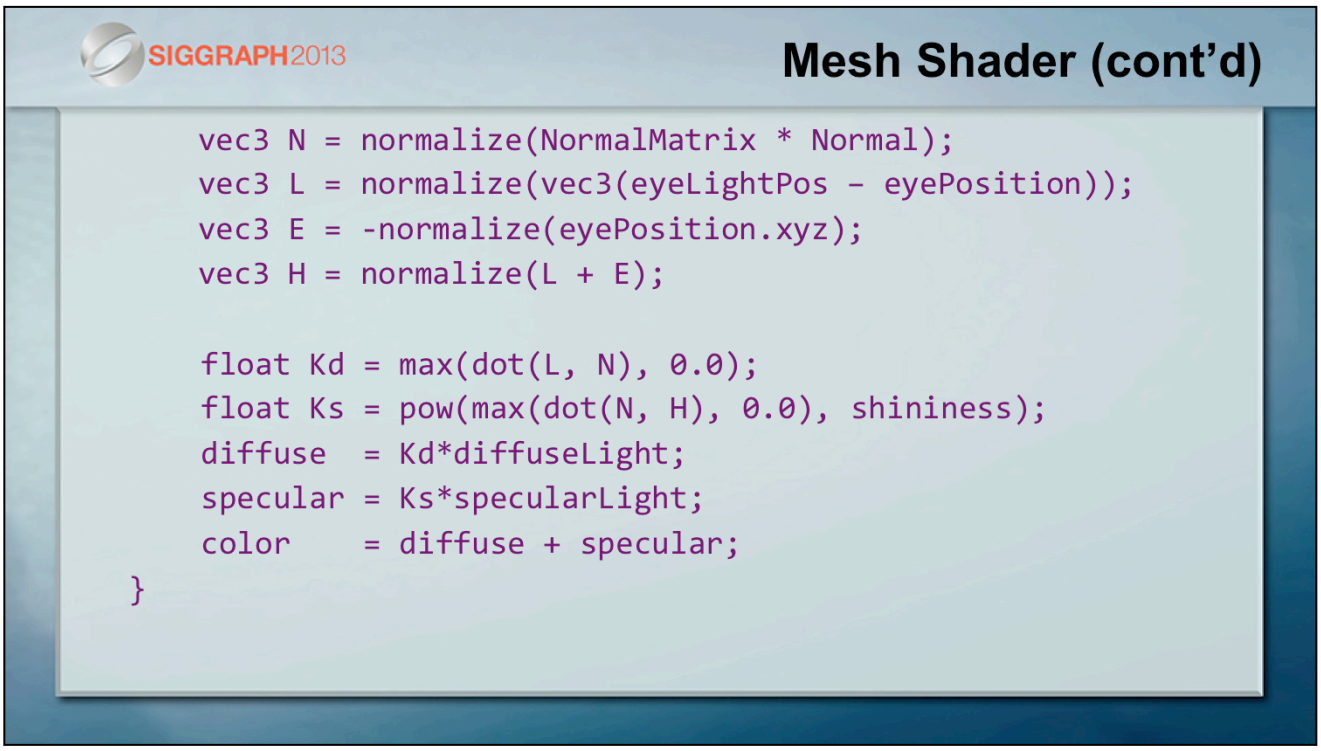
However, if we don't do some additional modification of one of our shaders, we'll get a large back blob. To produce a more useful rendering, we'll add lighting computations into our vertex shader, computing a lighting color for each vertex, which will be passed to the fragment shader.

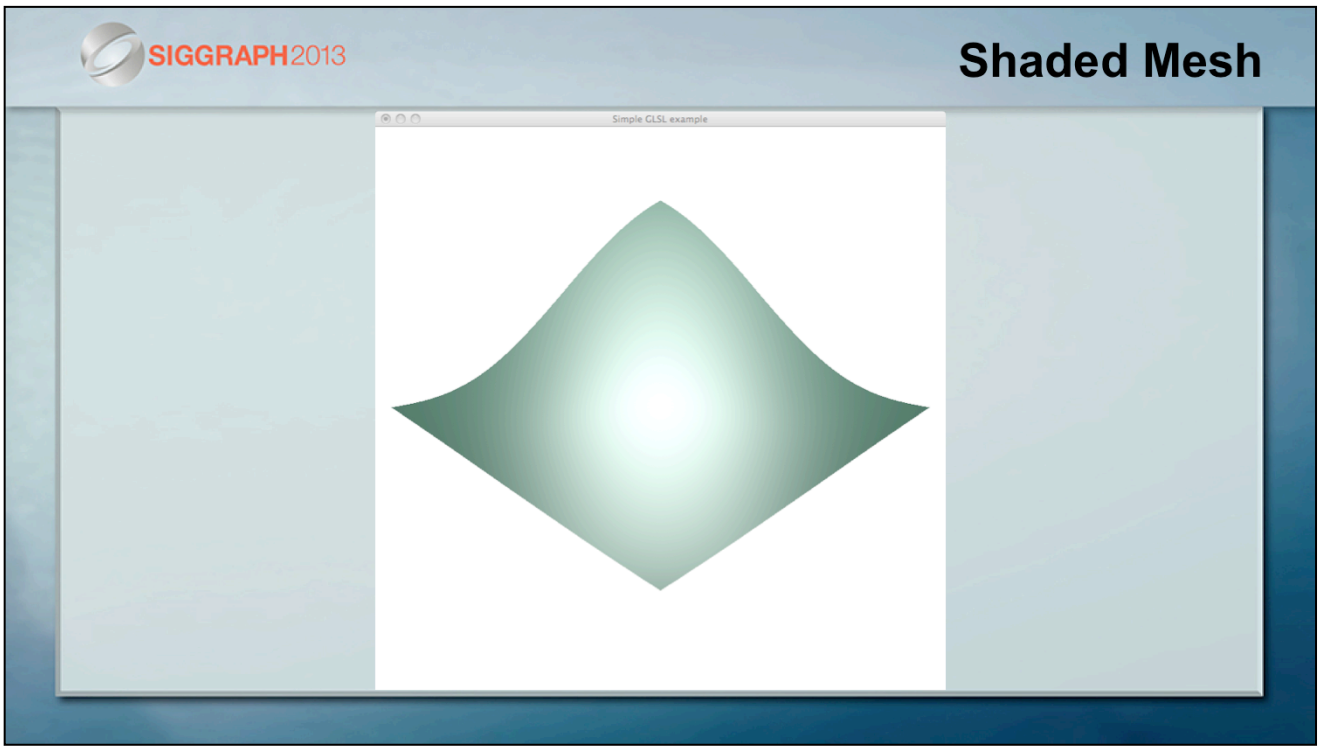


Details of lighting model are not important to here. The model includes the standard modified Phong diffuse and specular terms without distance.

Note that we do the lighting in eye coordinates and therefore must compute the eye position in this frame.

All the light and material properties are set in the application and are available through the OpenGL state.

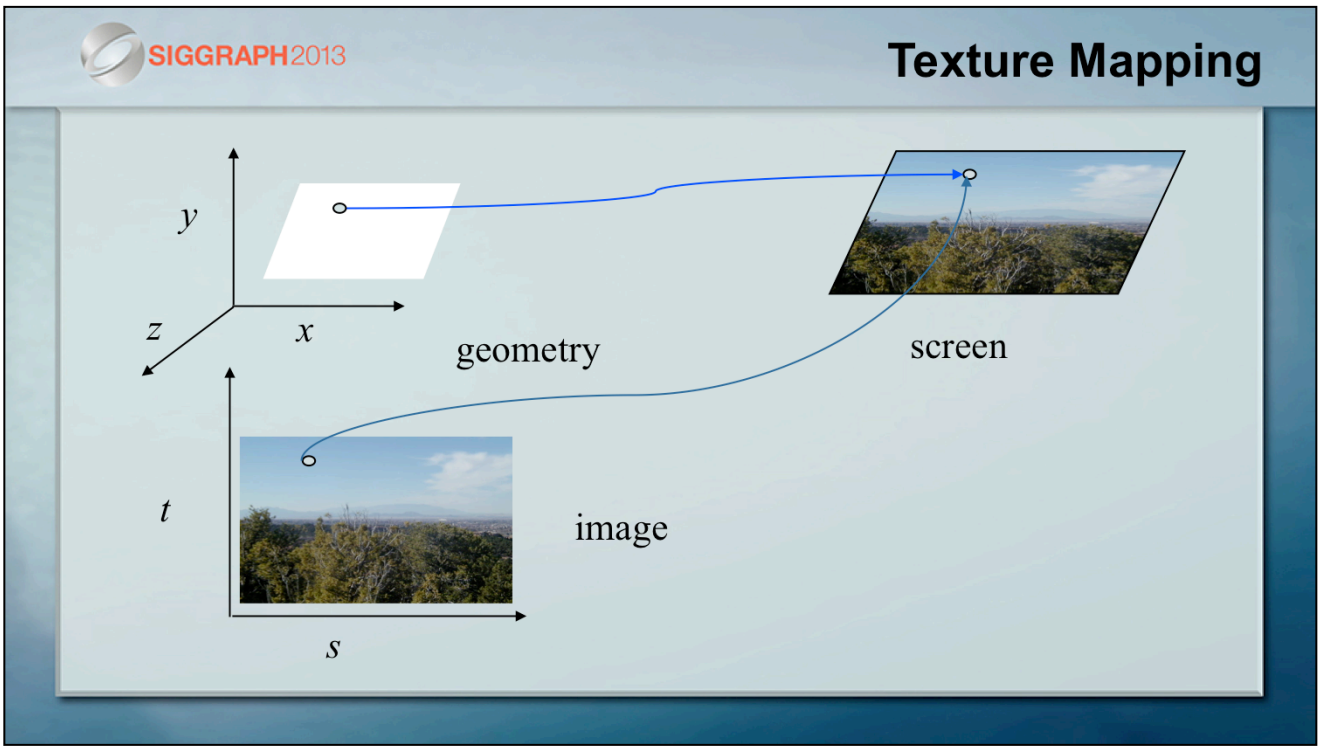

SIGGRAPH2013
Mesh Shader (cont'd)
vec3 N = normalize(NormalMatrix * Normal);
vec3 L = normalize(vec3(eyeLightPos – eyePosition));
vec3 E = -normalize(eyePosition.xyz);
vec3 H = normalize(L + E);
float Kd = max(dot(L, N), 0.0);
float Ks = pow(max(dot(N, H), 0.0), shininess);
diffuse = Kd*diffuseLight;
specular = Ks*specularLight;
color = diffuse + specular;
}



Here's a rendering of our shaded, solid mesh.

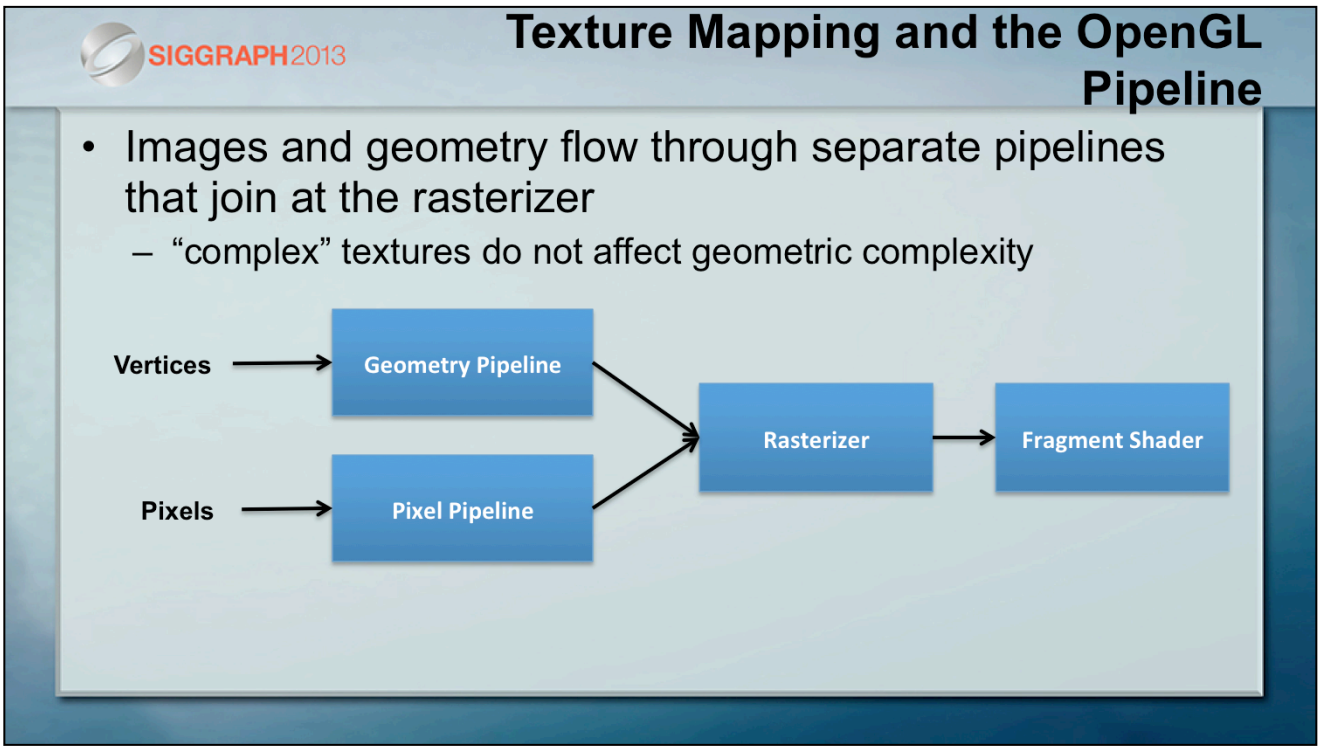
Texture Mapping

SIGGRAPH 2013
The 40th International Conference and Exhibition
on Computer Graphics and Interactive Techniques

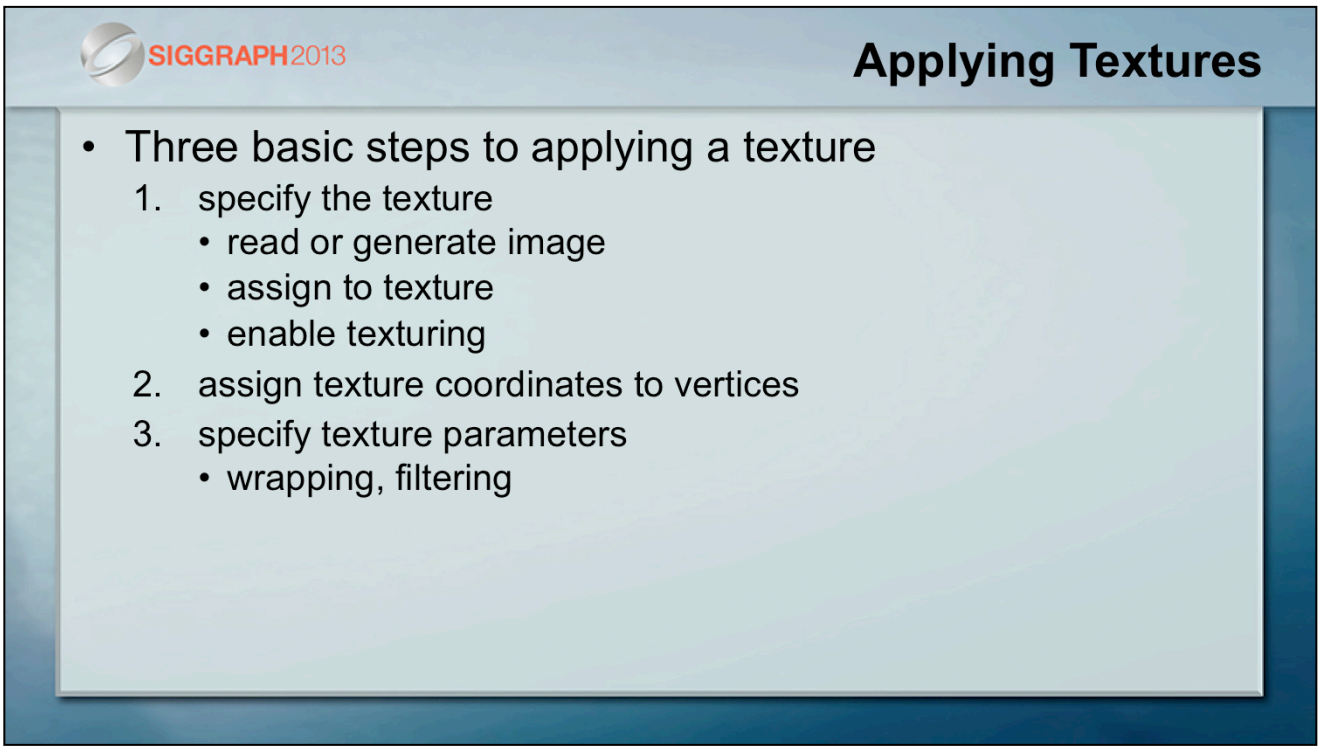


Textures are images that can be thought of as continuous and be one, two, three, or four dimensional. By convention, the coordinates of the image are *s*, *t*, *r* and *q*. Thus for the two dimensional image above, a point in the image is given by its (*s, t*) values with (*0, 0*) in the lower-left corner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



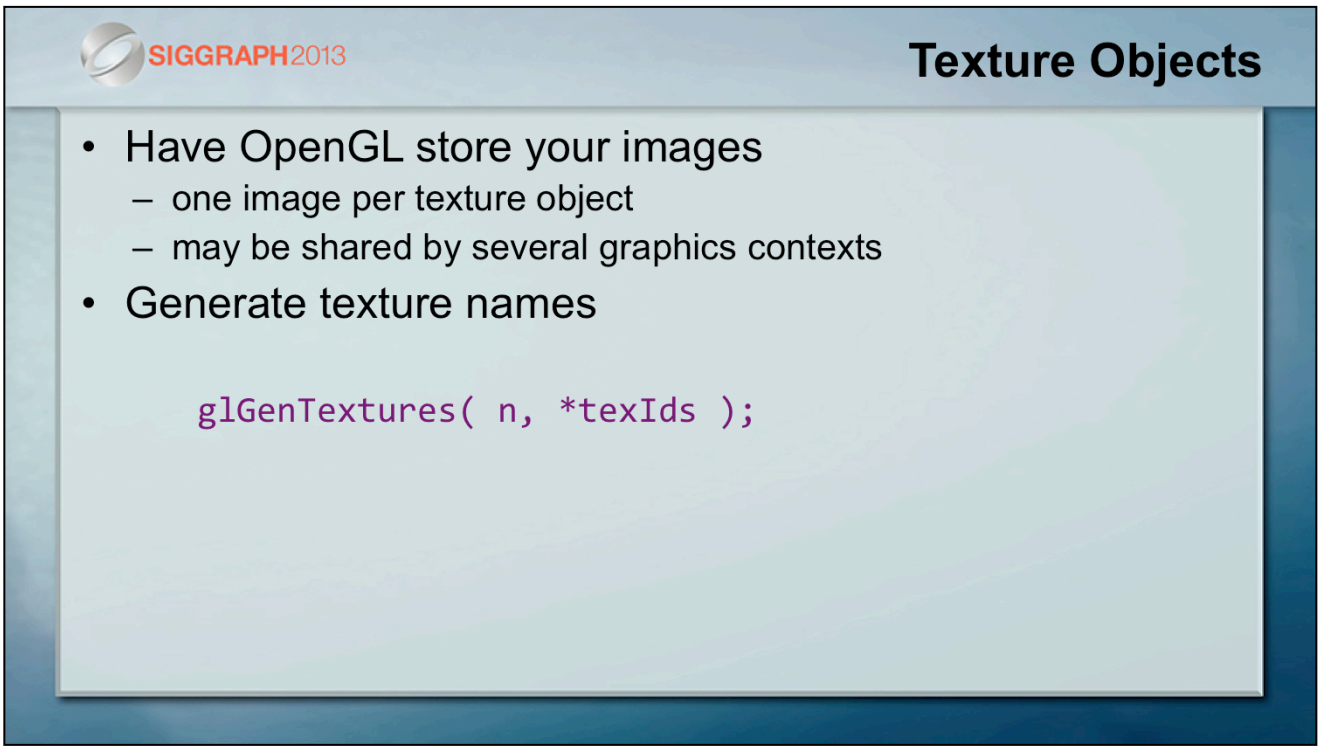
In the simplest approach, we must perform these three steps.

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

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

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

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

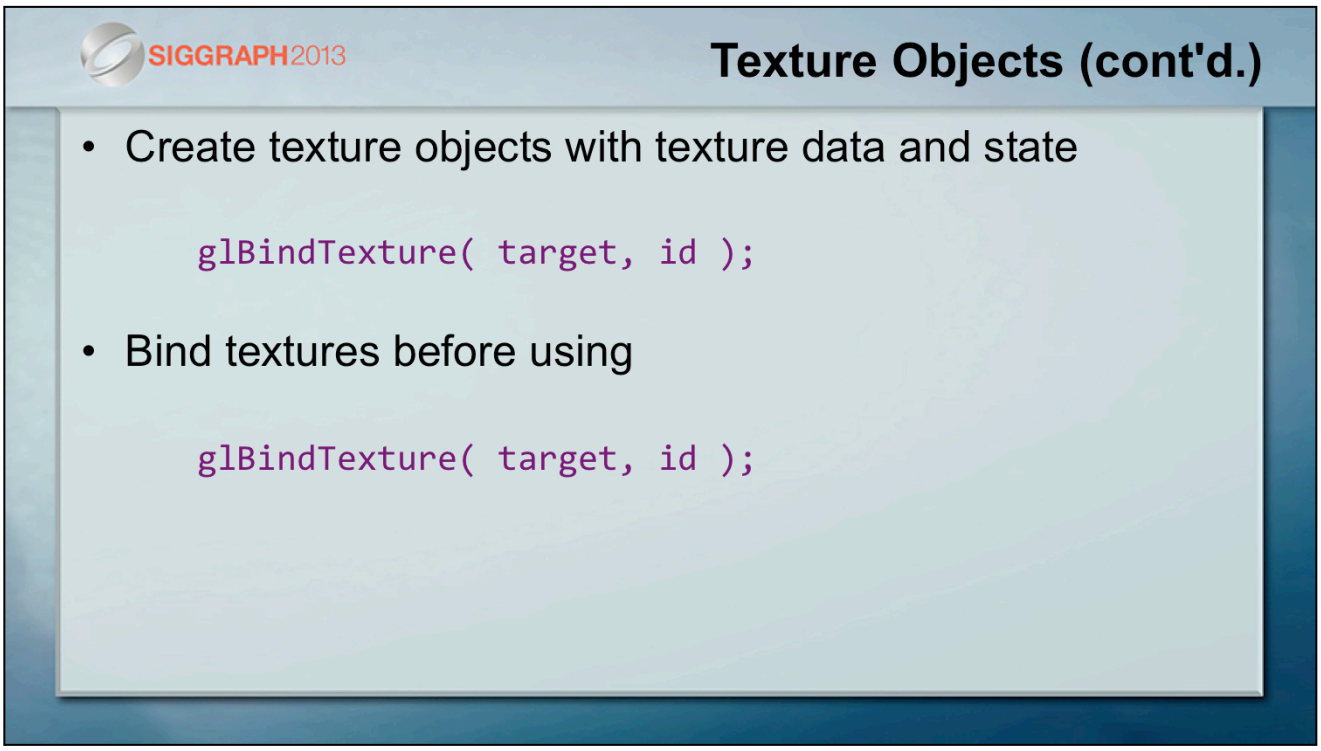


The first step in creating texture objects is to have OpenGL reserve some indices for your objects. `glGenTextures()` will request *n* texture ids and return those values back to you in `texIds`.

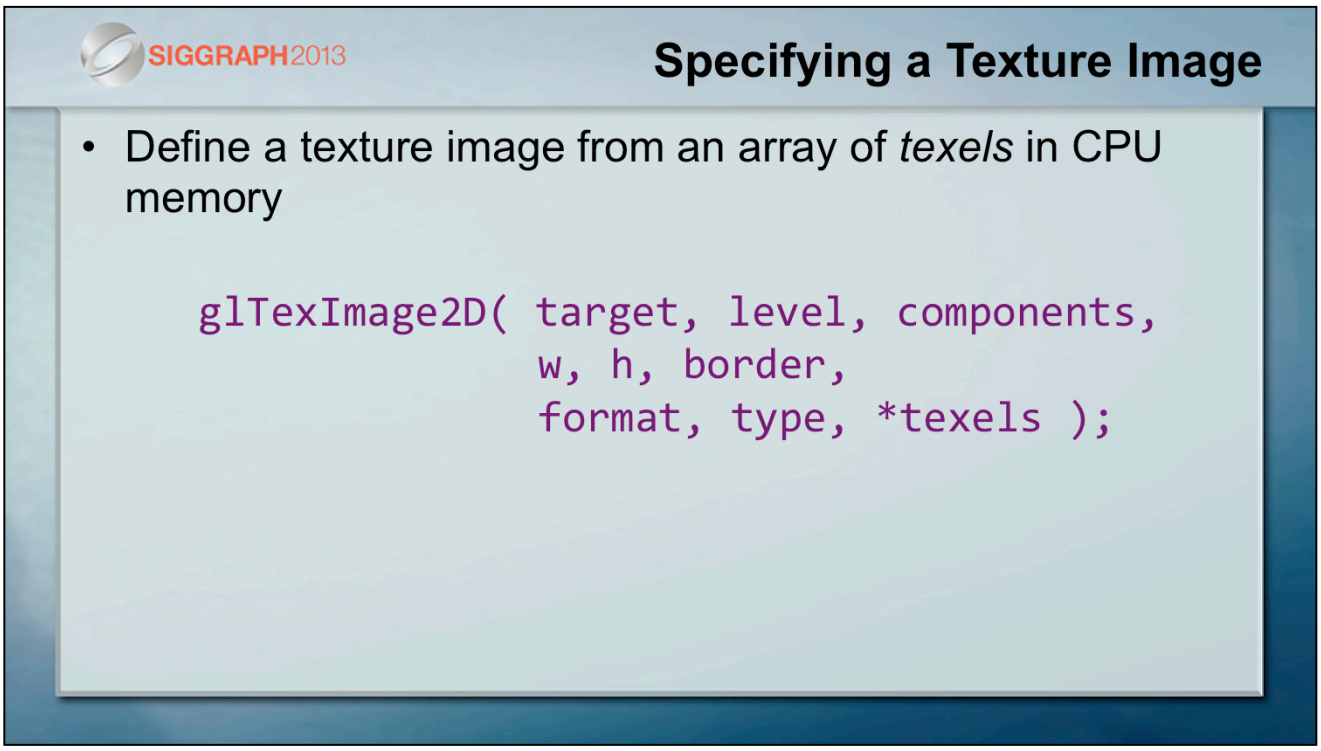
To begin defining a texture object, you call `glBindTexture()` with the id of the object you want to create. The target is one of `GL_TEXTURE_{123}D()`. All texturing calls become part of the object until the next `glBindTexture()` is called.

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

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

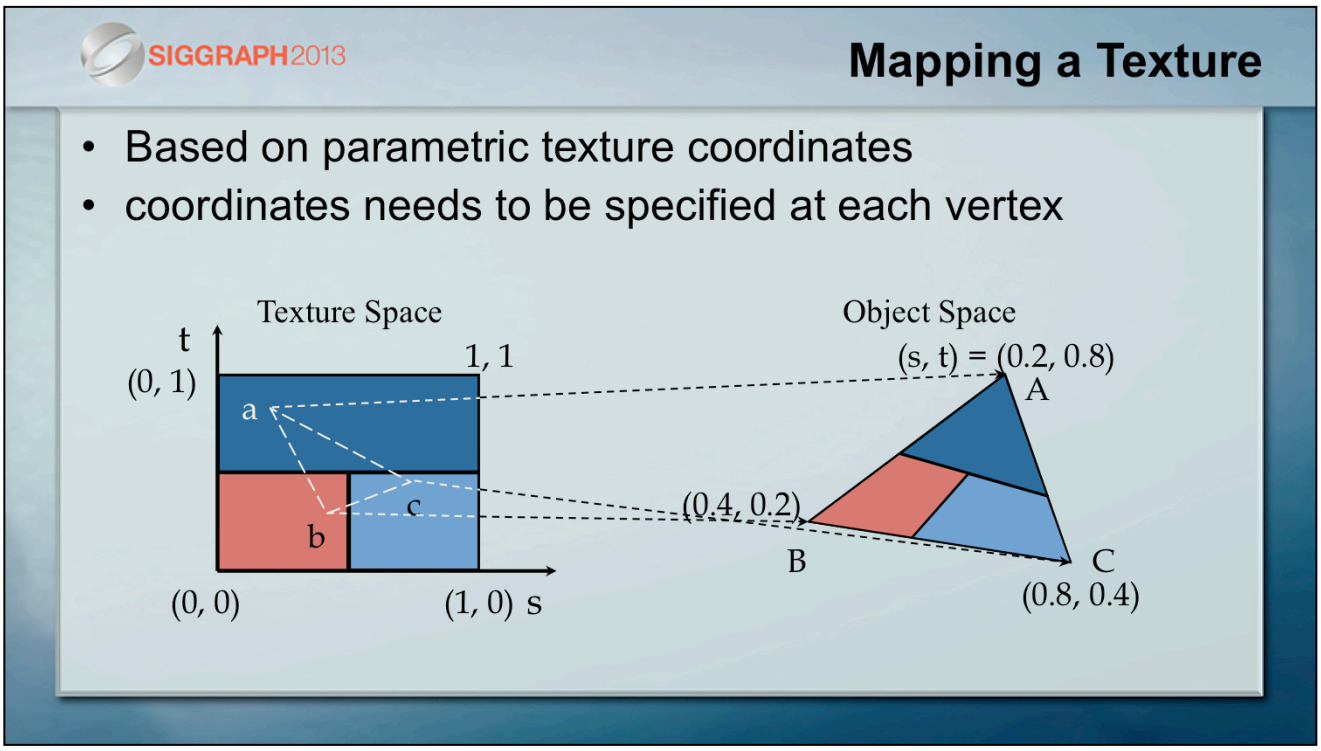


After creating a texture object, you'll need to bind to it to initialize or use the texture stored in the object. This operation is very similar to what you've seen when working with VAOs and VBOs.

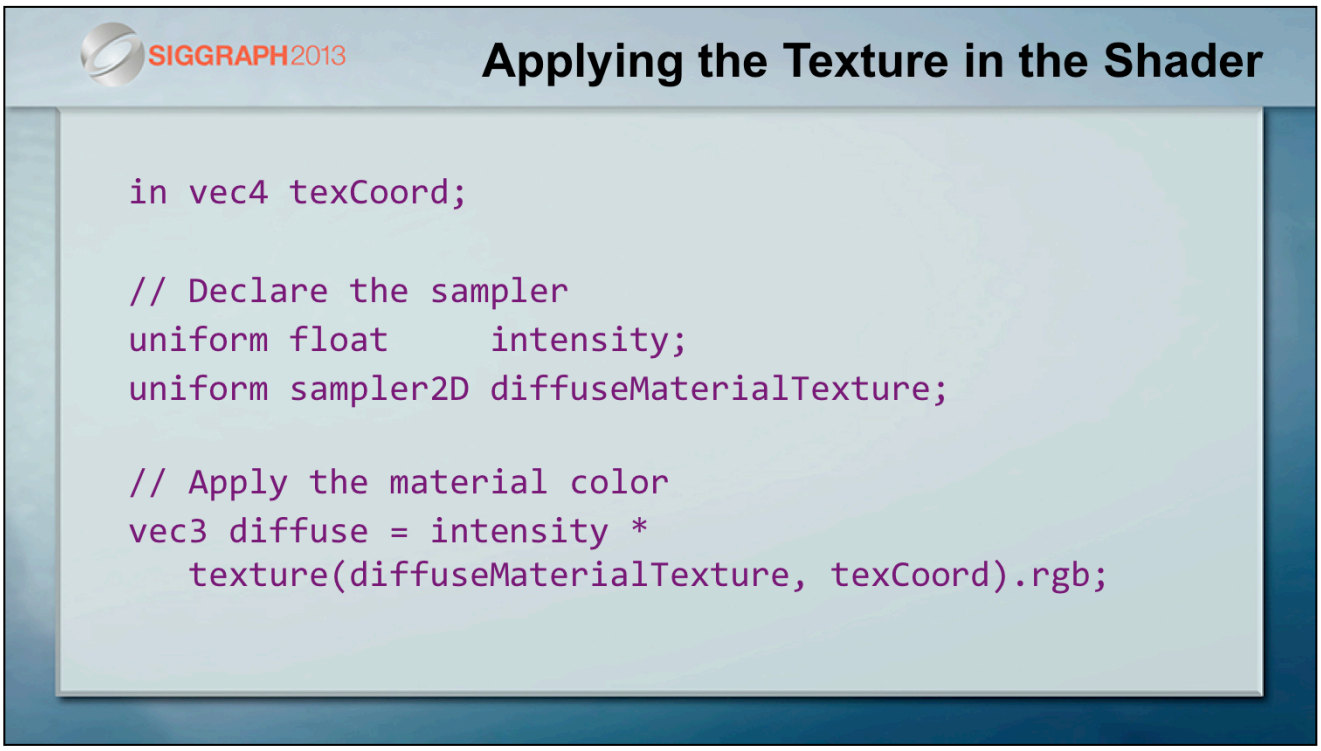


Specifying the texels for a texture is done using the `glTexImage{123}D()` call. This will transfer the texels in CPU memory to OpenGL, where they will be processed and converted into an internal format.

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



When you want to map a texture onto a geometric primitive, you need to provide texture coordinates. Valid texture coordinates are between 0 and 1, for each texture dimension, and usually manifest in shaders as vertex attributes. We'll see how to deal with texture coordinates outside the range [0, 1] in a moment.



Just like vertex attributes were associated with data in the application, so too with textures. In particular, you access a texture defined in your application using a *texture sampler* in your shader. The type of the sampler needs to match the type of the associated texture. For example, you would use a sampler2D to work with a two-dimensional texture created with glTexImage2D(GL_TEXTURE_2D, …);

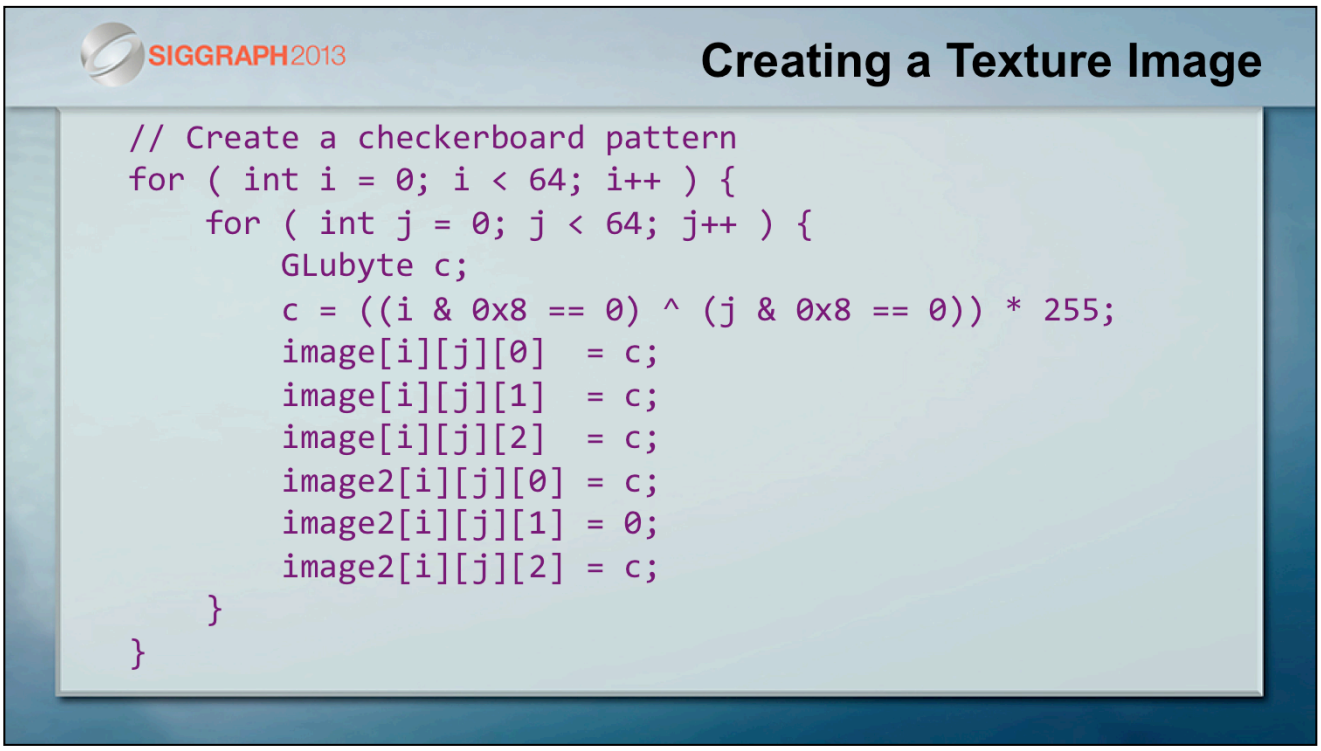
Within the shader, you use the texture() function to retrieve data values from the texture associated with your sampler. To the texture() function, you pass the sampler as well as the texture coordinates where you want to pull the data from.

Note: the overloaded texture() method was added into GLSL version 3.30. Prior to that release, there were special texture functions for each type of texture sampler (e.g., there was a texture2D() call for use with the sampler2D).

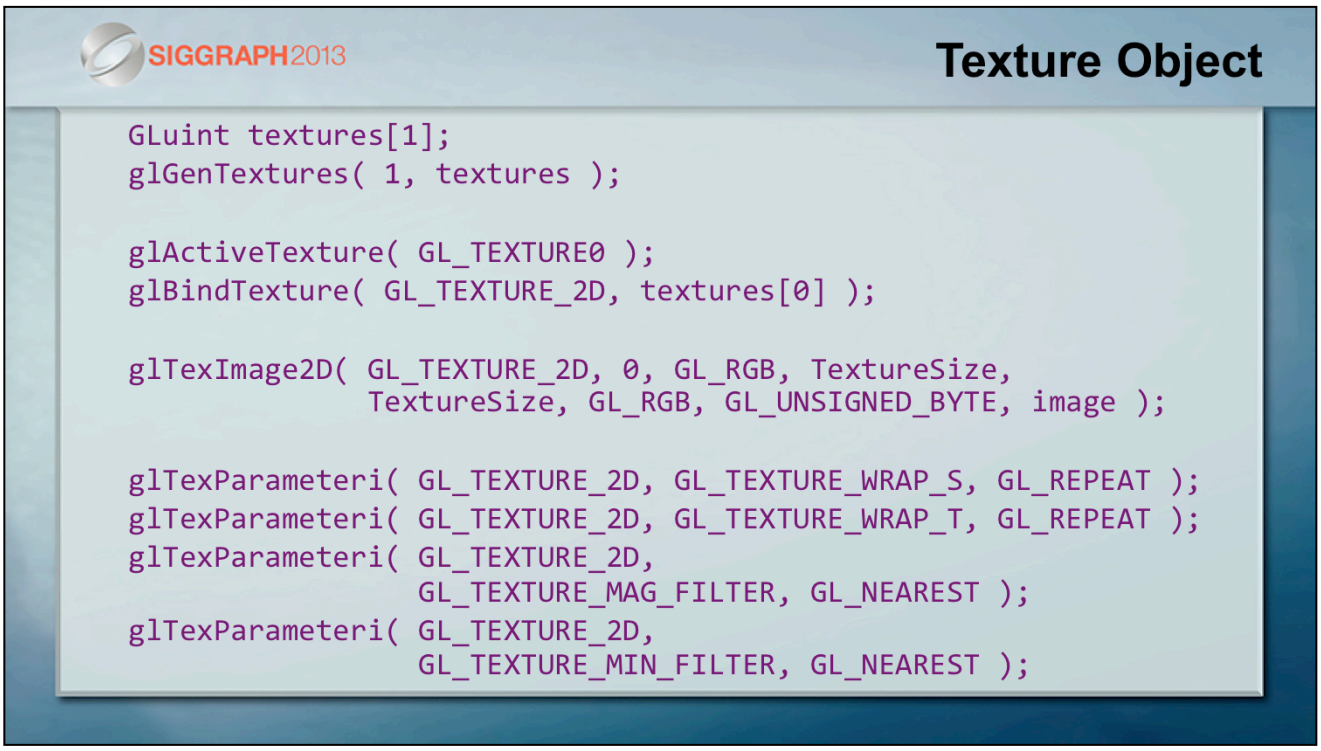
Applying Texture to Cube

```
// add texture coordinate attribute  to quad function

quad( int a, int b, int c, int d )
{
    vColors[Index] = colors[a];
    vPositions[Index] = positions[a];
    vTexCoords[Index] = vec2( 0.0, 0.0 );
    Index++;

    vColors[Index] = colors[b];
    vPositions[Index] = positions[b];
    vTexCoords[Index] = vec2( 1.0, 0.0 );
    Index++;
    … // rest of vertices
}
```

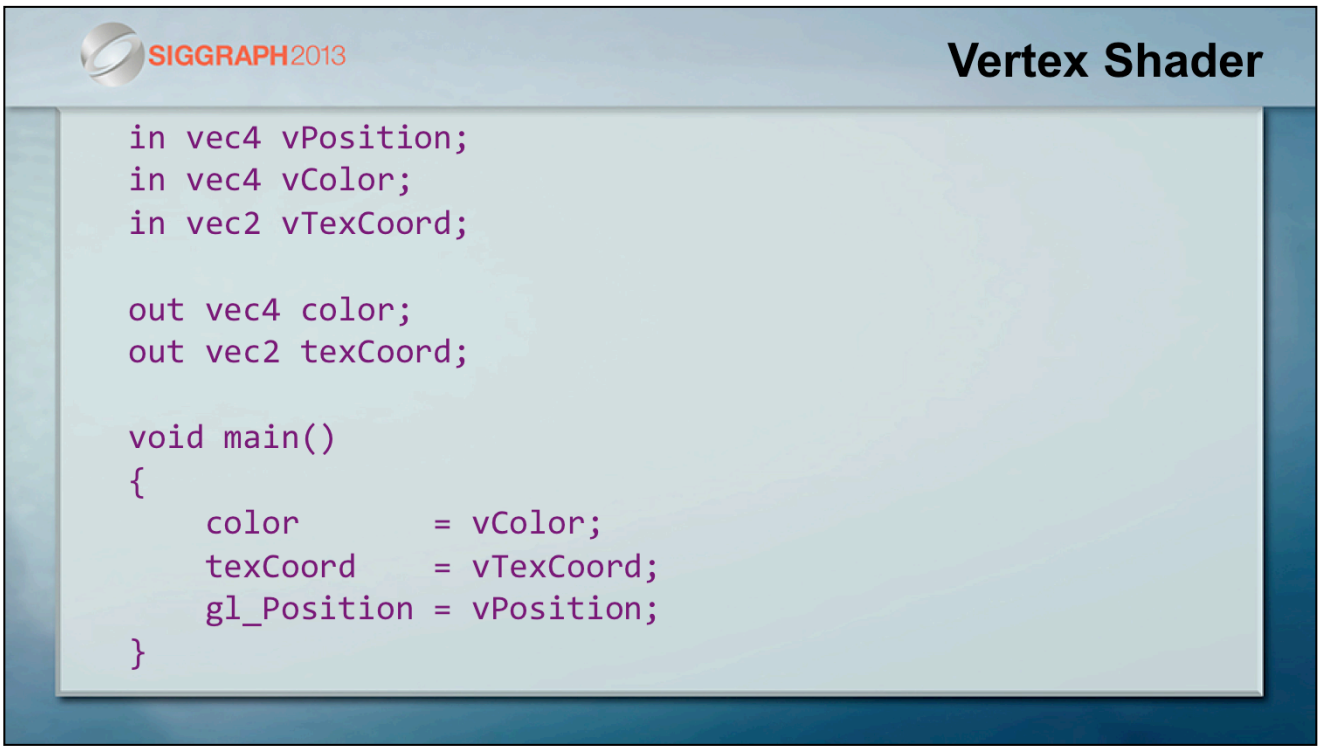
Similar to our first cube example, if we want to texture our cube, we need to provide texture coordinates for use in our shaders. Following our previous example, we merely add an additional vertex attribute that contains our texture coordinates. We do this for each of our vertices. We will also need to update VBOs and shaders to take this new attribute into account.



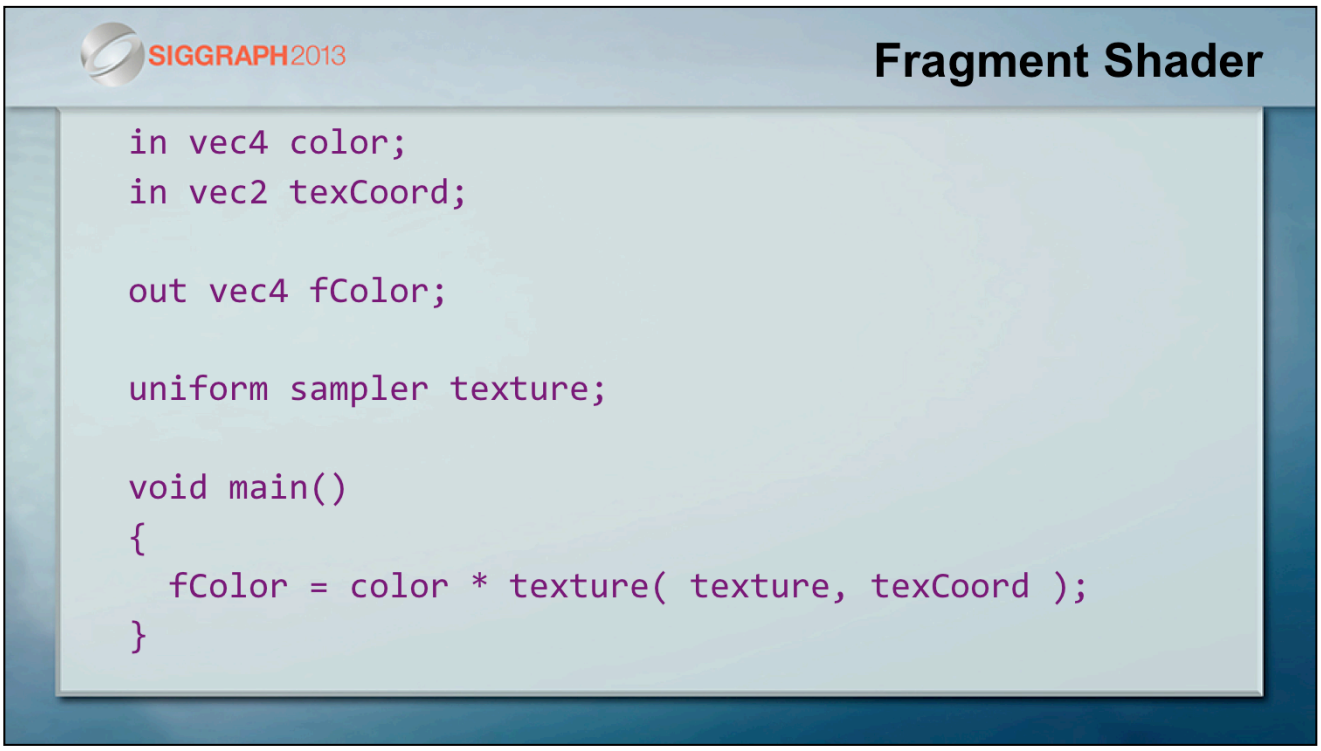
The code snippet above demonstrates procedurally generating a two 64 × 64 texture maps.



The above OpenGL commands completely specify a texture object. The code creates a texture id by calling glGenTextures(). It then binds the texture using glBindTexture() to open the object for use, and loading in the texture by calling glTexImage2D(). After that, numerous sampler characteristics are set, including the texture wrap modes, and texel filtering.

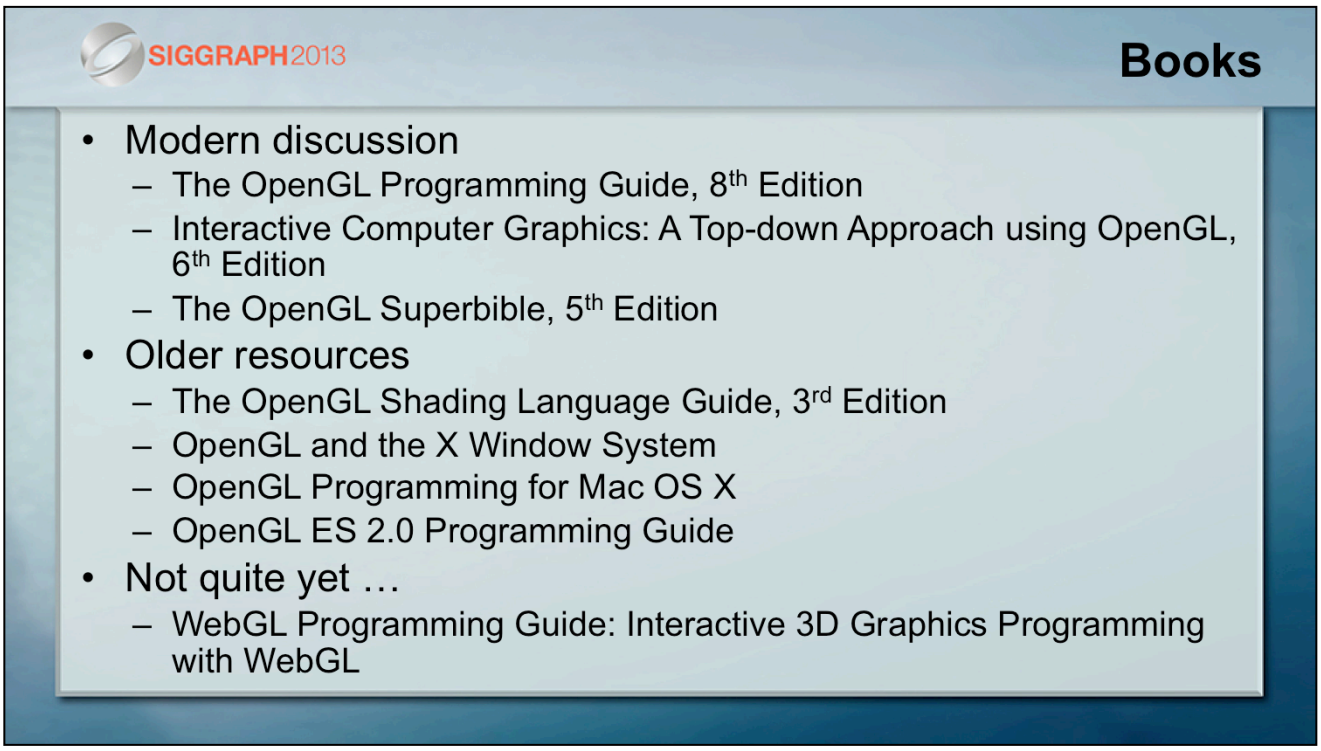


In order to apply textures to our geometry, we need to modify both the vertex shader and the pixel shader. Above, we add some simple logic to pass-thru the texture coordinates from an attribute into data for the rasterizer.

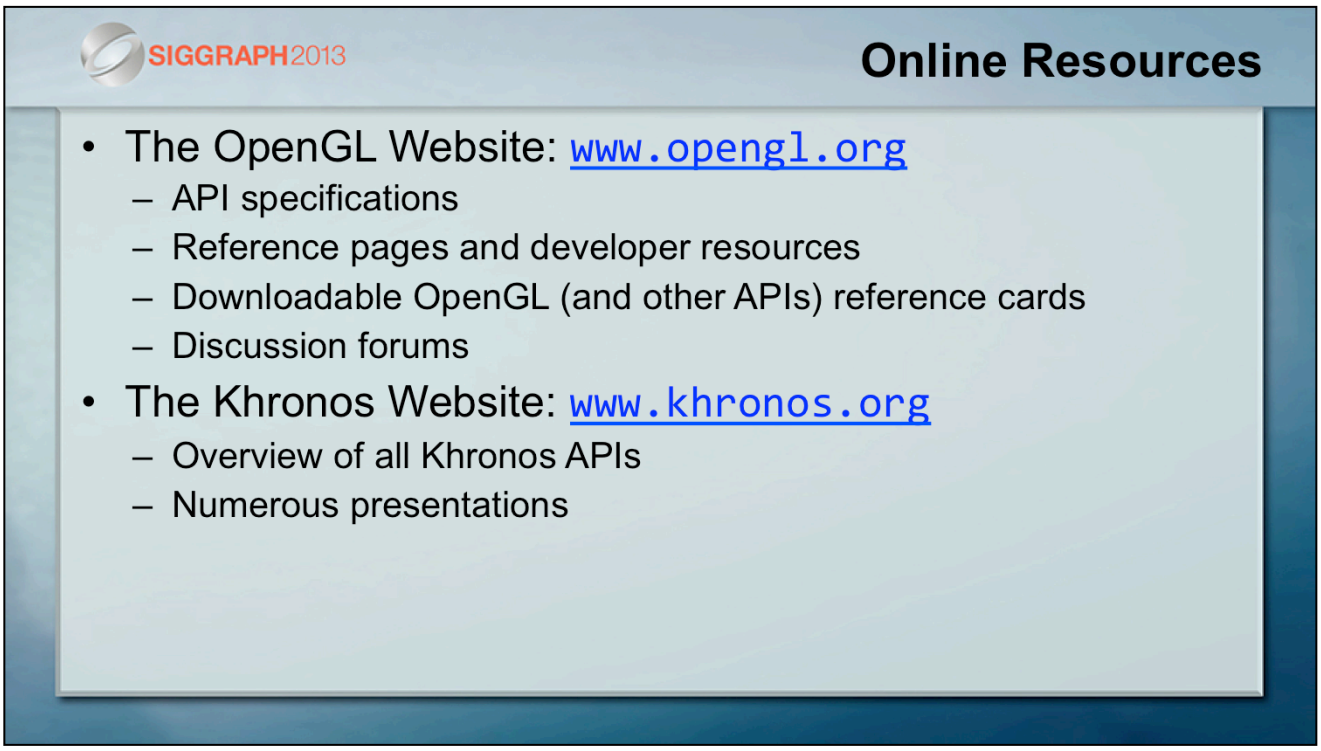


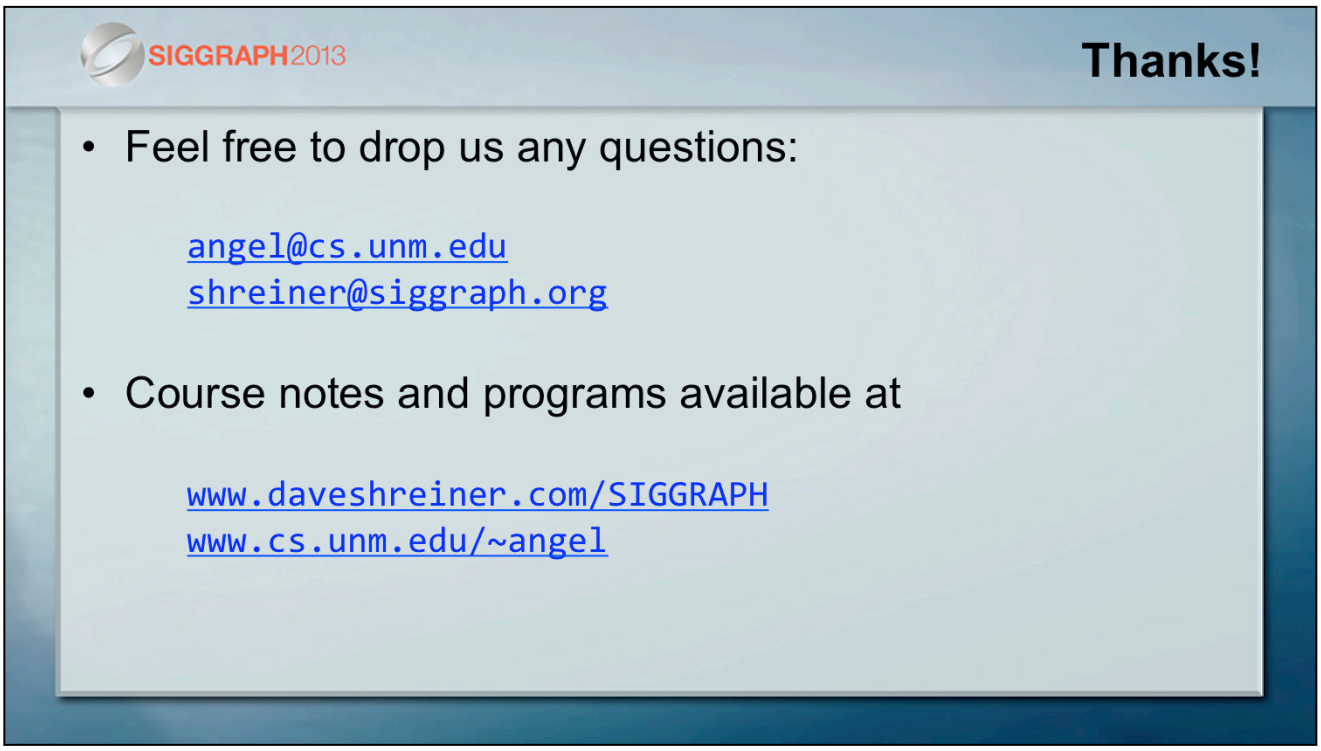
Continuing to update our shaders, we add some simple code to modify our fragment shader to include sampling a texture. How the texture is sampled (e.g., coordinate wrap modes, texel filtering, etc.) is configured in the application using the glTexParameter*() call.


Resources
SIGGRAPH2013
The 40th International Conference and Exhibition
on Computer Graphics and Interactive Techniques



All the above books except Angel and Shreiner, Interactive Computer Graphics (Addison-Wesley), are in the Addison-Wesley Professional series of OpenGL books.


SIGGRAPH2013
Online Resources
• The OpenGL Website: www.opengl.org
– API specifications
– Reference pages and developer resources
– Downloadable OpenGL (and other APIs) reference cards
– Discussion forums
• The Khronos Website: www.khronos.org
– Overview of all Khronos APIs
– Numerous presentations



Many example programs, a C++ matrix-vector package and the InitShader function are under the Book Support tab at www.cs.unm.edu/~angel