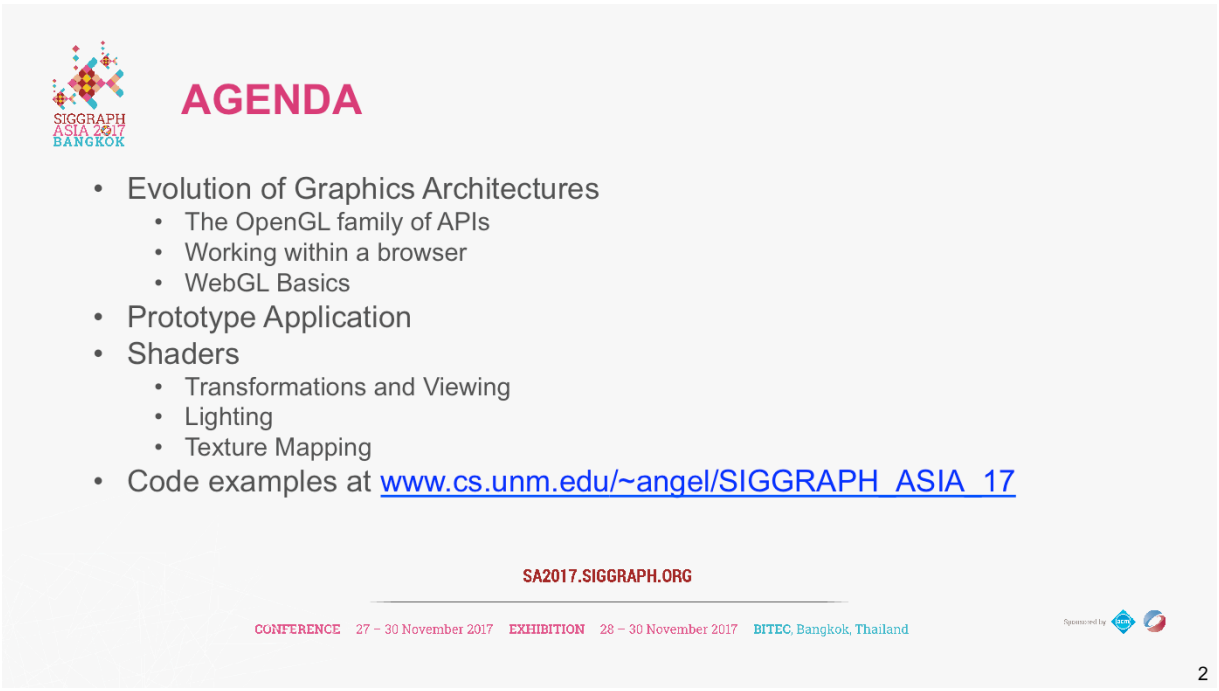
SIGGRAPH ASIA 2017 BANGKOK
CONFERENCE 27 – 30 November 2017
EXHIBITION 28 – 30 November 2017
BITEC, Bangkok, Thailand
SA2017.SIGGRAPH.ORG
Sponsored by

Application Development with WebGL

Ed Angel
University of New Mexico

Dave Shreiner
Unity Technologies



We will use WebGL 1.0. WebGL 2.0 is now being supported by most browsers but requires a better GPU so may not run on older computers or on most cell phones and tablets. See http://webglstats.com/. We will cover the new features supported by WebGL 2.0 in the second half of the course.

We assume that you are familiar with the following:

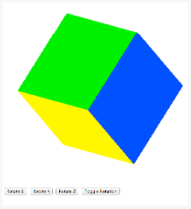
Basic graphics concepts are those in fundamentals course, including the basics of image formation.

For this course, experience with C/C++, Java, Python should suffice.

We assume you are comfortable running remote applications in a browser.

WHAT YOU'LL SEE TODAY

- Evolving an example — rendering a Cube
  - Geometry
  - Interaction
  - Lighting
  - Texture Mapping



SA2017.SIGGRAPH.ORG

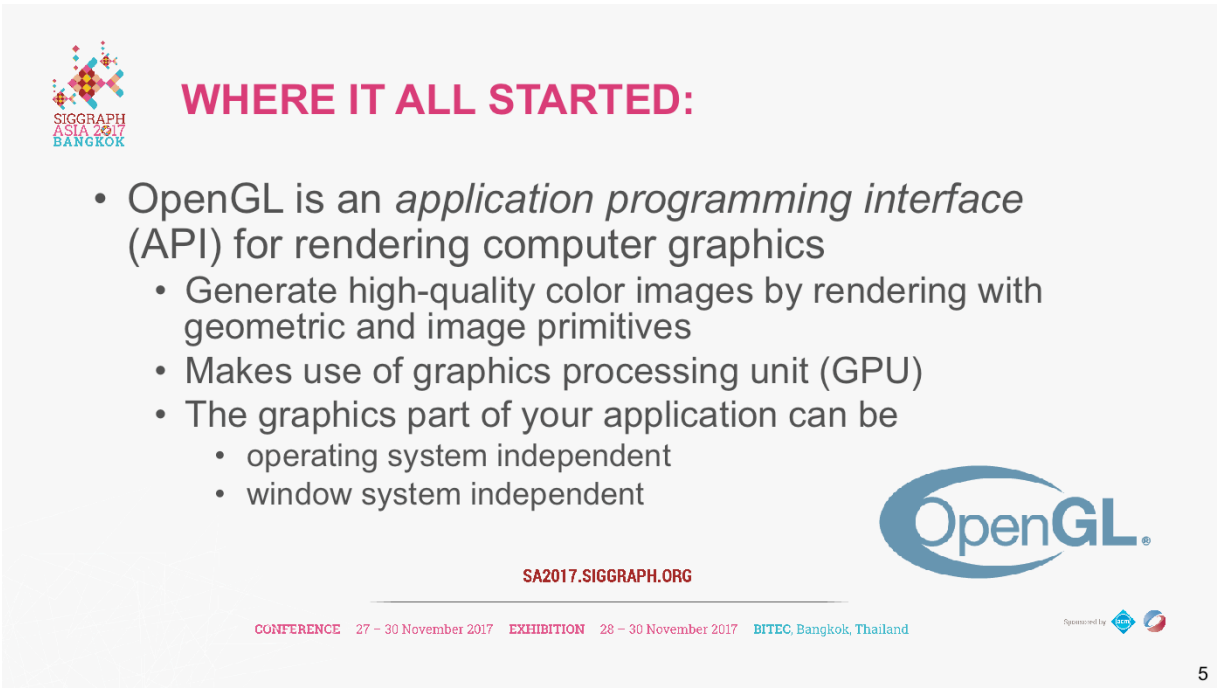
CONFERENCE 27 – 30 November 2017 EXHIBITION 28 – 30 November 2017 BITEC, Bangkok, Thailand

Because WebGL has a limited set of primitives, with WebGL we must model the cube with triangles. With WebGL we can use libraries available on the Web such as jQuery or code devices through HTML. With WebGL we can build our lighting models in a variety of ways including within the shaders. WebGL supports texture mapping using images available in standard formats png, (gif, jpeg) or images defined in the code.

WebGL Lineage and Evolution

SA2017.SIGGRAPH.ORG

CONFERENCE 27 – 30 November 2017 EXHIBITION 28 – 30 November 2017 BITEC, Bangkok, Thailand



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, 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. Likewise, interaction is not part of the API. Although the graphics part of the application is independent of the OS, applications must be recompiled on each architecture.

EVOLUTION OF THE OPENGL PIPELINE

OpenGL Version	Release Date	Notable Features
1.0	1994	Original *fixed-function* interface (function calls for setting all state)
2.0	2004	Added programmable shading: • *vertex shading* for transformations and lighting • *fragment shading* for enhanced fragment coloring
3.0	2008	Added *geometry shading* stage, and introduced mechanism for removing features (*deprecation*)
3.1	2009	Removed fixed-function interfaces
4.0	2010	Added *tessellation shading* stages
4.3	2012	Added *compute shading* stage
4.6	2017	Most recent release

The initial version of OpenGL implemented a *fixed-function pipeline*, in which all 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. 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 perform better than using one of these early versions of OpenGL since they can take advantage of the capabilities of recent Graphics Processing Units (GPUs). 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).

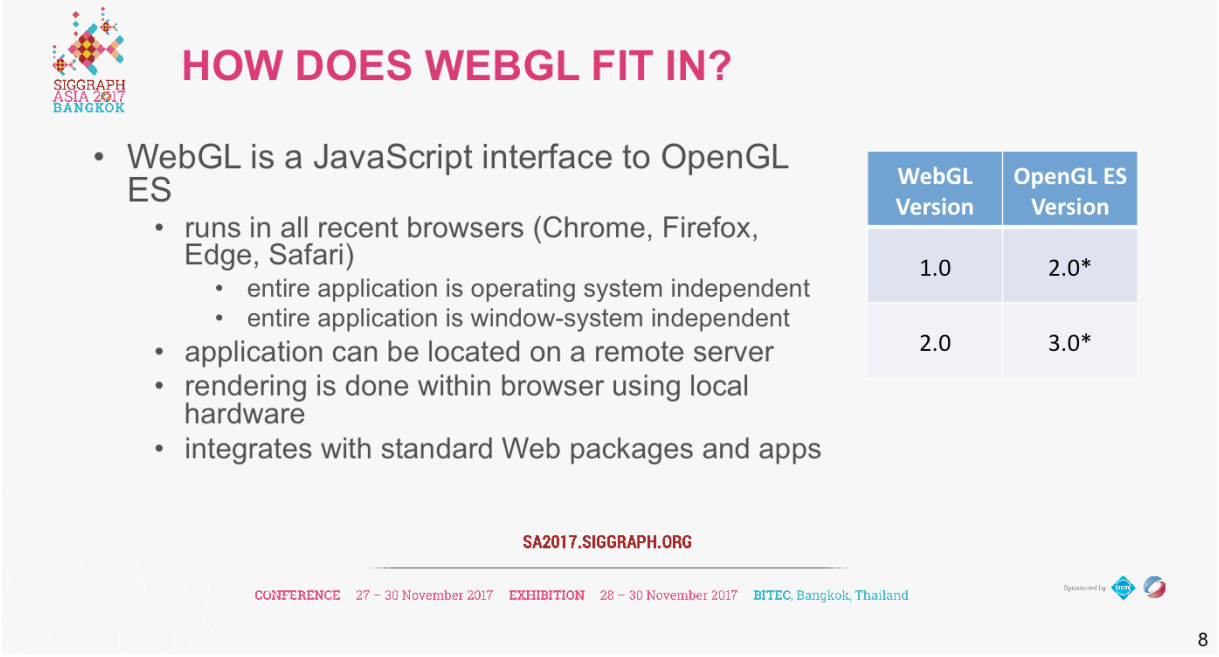
Until OpenGL 3.0, features have only been added (but never removed) from OpenGL, providing a lot of application backwards compatibility (up to the use of extensions). OpenGL version 3.0 introduced the mechanisms for removing features from OpenGL, called the *deprecation model.*

OPENGL VARIANTS: OPENGL ES

- OpenGL ES (Embedded System)
  - Designed for embedded and hand-held devices such as cell phones

OpenGL ES Version	Release Date	OpenGL Version	Notable Features
1.0	2003	1.3	Original *fixed-function* interface (function calls for setting all state)
2.0	2007	2.0	Programmable shading only (vertex and fragment stages)
3.1	2014	—	Added *compute shading* stage
3.2	2015	—	Most recent release

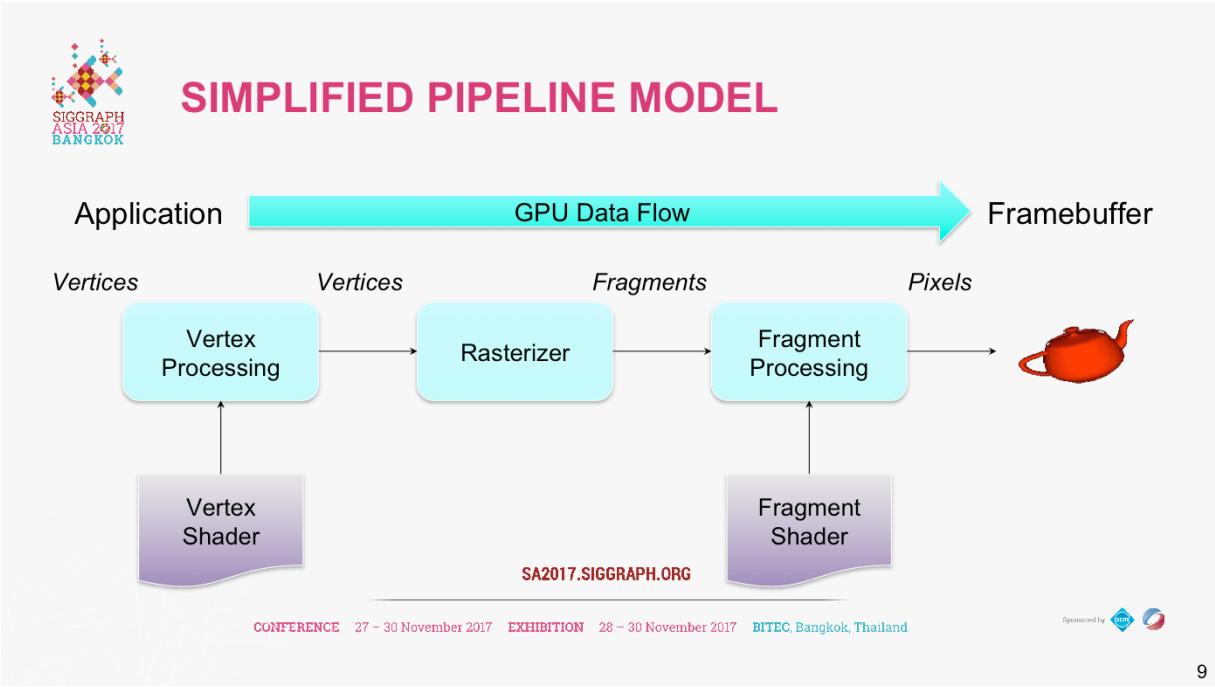
WebGL is becoming increasingly more important because it is supported by all browsers. Besides the advantage of being able to run without recompilation across platforms, it can easily be integrated with other Web applications and make use of a variety of portable packages available over the Web.



With it, you can generate high-quality color images by rendering with geometric and image primitives

Makes use of graphics processing unit (GPU)

By using OpenGL, the graphics part of your



Once our JS and HTML code is interpreted and executes with a basic 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.

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/WebGL applications, you'll usually need to do the following tasks:

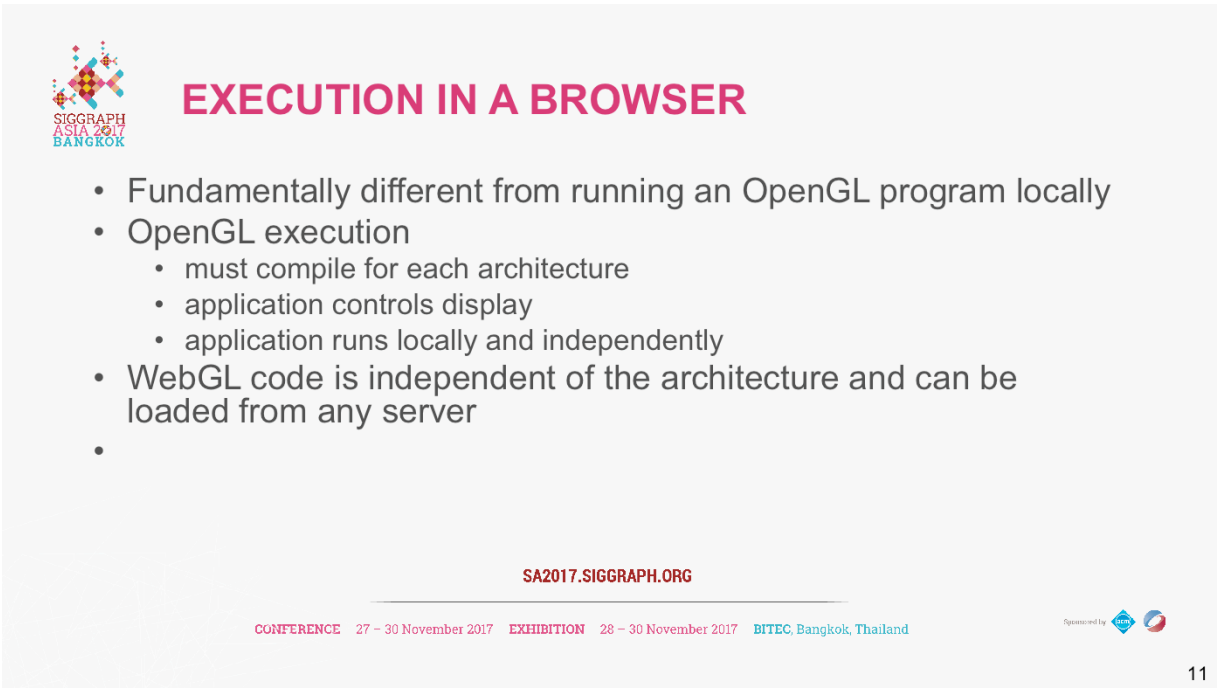
- specify the vertices for your geometry
- load vertex and fragment shaders (and other shaders, if you're using them as well)
- issue your geometry to engage the pipeline for processing

Of course, OpenGL and WebGL are capable of many other operations as well, many of which are outside of the scope of this introductory course. We have included references at the end of the notes for your further research and development.

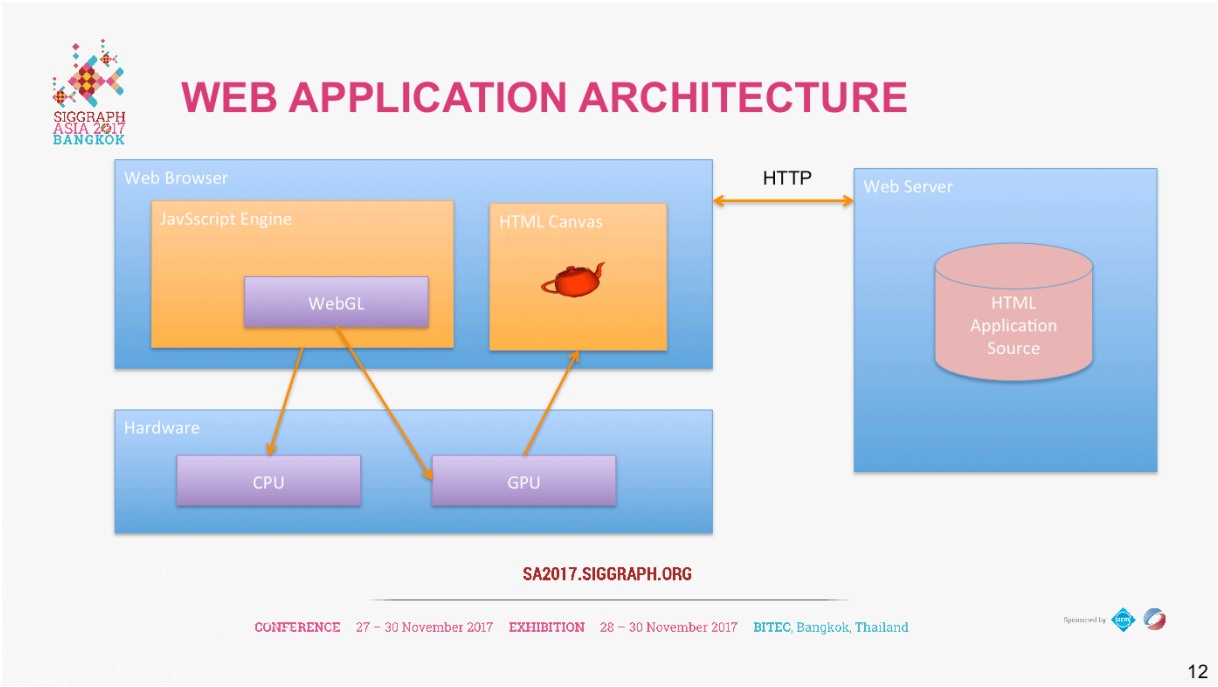
WebGL Application Development

SA2017.SIGGRAPH.ORG

CONFERENCE 27 – 30 November 2017 EXHIBITION 28 – 30 November 2017 BITEC, Bangkok, Thailand

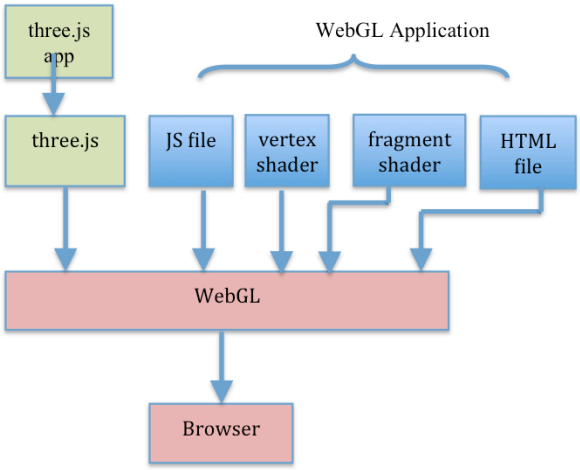


Although OpenGL source code for rendering should be the same across multiple platforms, the code must be recompiled for each architecture, In addition, the non-rendering parts of an application such as opening windows and input are not part of of OpenGL and can be very different on different systems. Almost all OpenGL applications are designed to run locally on the computer on which they live.



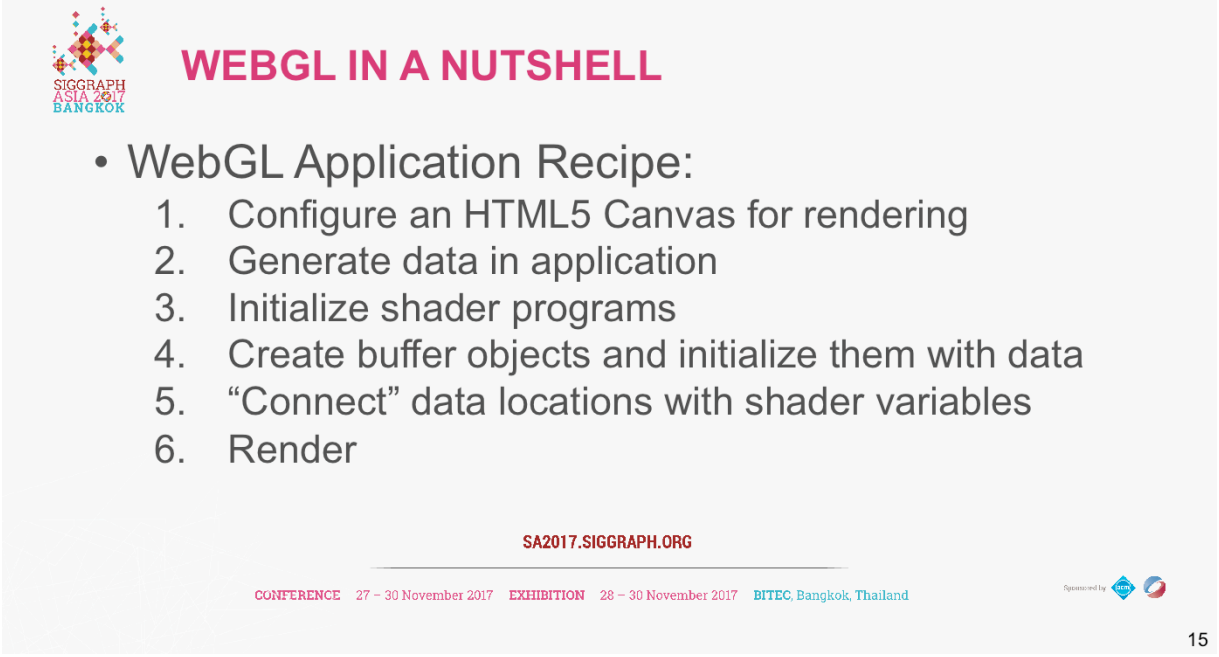
A typical WebGL application consists of a mixture of HTML5, JavaScript and GLSL (shader) code. The application can be located almost anywhere and is accessed through its URL . All browsers can run JavaScript and all modern browsers support HTML. The rendering part of the application is in JavaScript and renders into the HTML5 Canvas element. Thus, the WebGL code is obtained from a server (either locally or remote) and is compiled by the browser's JavaScript engine into code that run on the local CPU and GPU.

WEBGL VS THREE.JS APPLICATIONS



WHY USE WEBGL DIRECTLY?

- Three.js and other libraries (babylon.js, OSG.JS) are handy.
- But:
- They need to be downloaded. Just three.min.js is ~500kB.
- They may not do just what you want, and may have bugs.
- Some ways they have of storing data are inefficient.
- You may already have OpenGL code to port.
- Teaching WebGL crosses over to OpenGL, and DirectX.
- There are many more resources for OpenGL programming.
- Knowing WebGL makes it easier to learn and use other APIs



You’ll find that a few techniques for programming with modern WebGL goes a long way. In fact, most programs – in terms of WebGL 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:

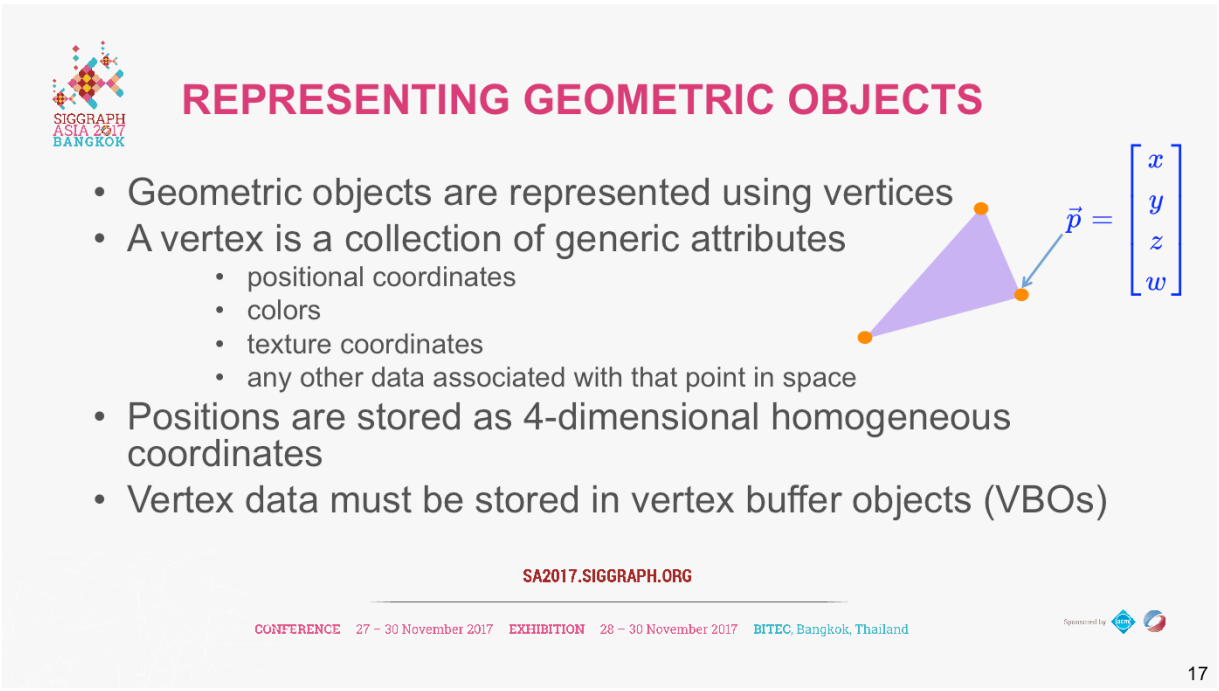
1. First, you’ll load and create WebGL *shader programs* from shader source programs you create
2. Next, you will need to load the data for your objects into WebGL’s memory. You do this by creating *buffer objects* and loading data into them.
3. Continuing, WebGL needs to be told how to interpret the 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

Modeling Geometry in WebGL

SA2017.SIGGRAPH.ORG

CONFERENCE 27 – 30 November 2017 EXHIBITION 28 – 30 November 2017 BITEC, Bangkok, Thailand

Sponsored by



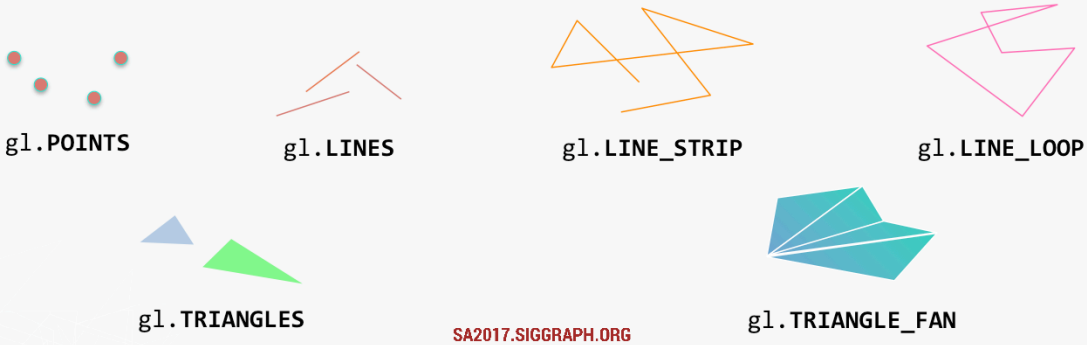
In OpenGL, as in other graphics libraries, objects in the scene are composed of *geometric primitives*, which themselves are described by *vertices*. A vertex in modern OpenGL is a collection of data values associated with a location in space. Those data values might include colors, reflection information 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*.

The homogenous coordinate representation of a point has w = 1 and for a vector w = 0. Perspective cameras can change the value of w. We return to normal 3D coordinates by perspective division which replaces p = [x, y, z, w] by p'= [x/w, y/w, z/w].

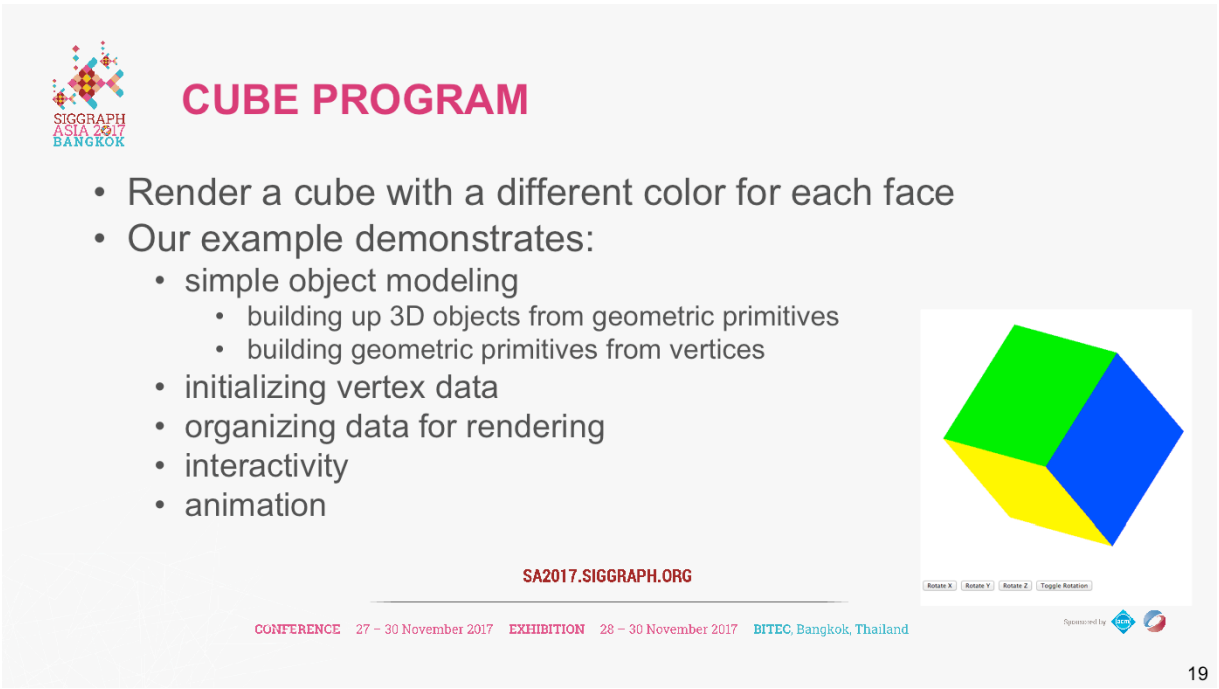
Vertices must be organized in OpenGL server-side objects called *vertex buffer objects* (also known as *VBOs*), which need to contain all of the vertex information for all the primitives that you want to draw at one time.

WEBGL GEOMETRIC PRIMITIVES

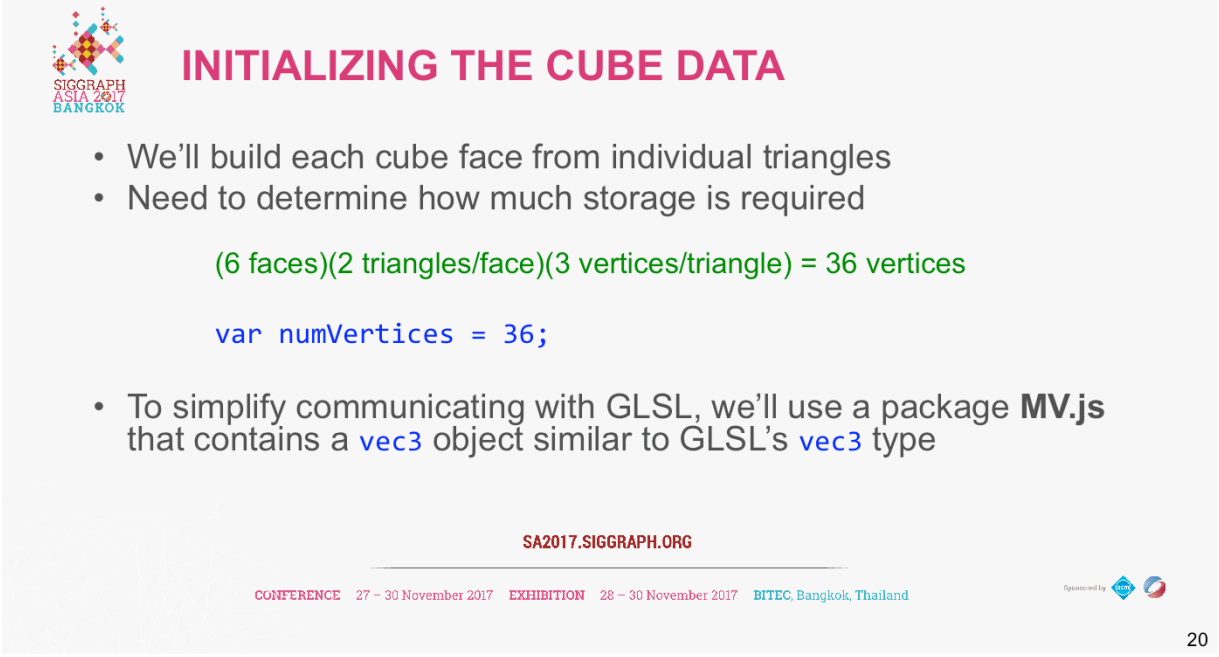
- All primitives are specified by vertices



To form 3D geometric objects, you need to decompose them into geometric primitives that WebGL can draw. WebGL (and modern desktop 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.



The next few slides will introduce our example program, one which simply displays a cube with different colors at each vertex. We aim for simplicity in this example, focusing on the WebGL techniques, and not on optimal performance. This example is animated with rotation about the three coordinate axes and interactive buttons that allow the user to change the axis of rotation and start or stop the rotation.

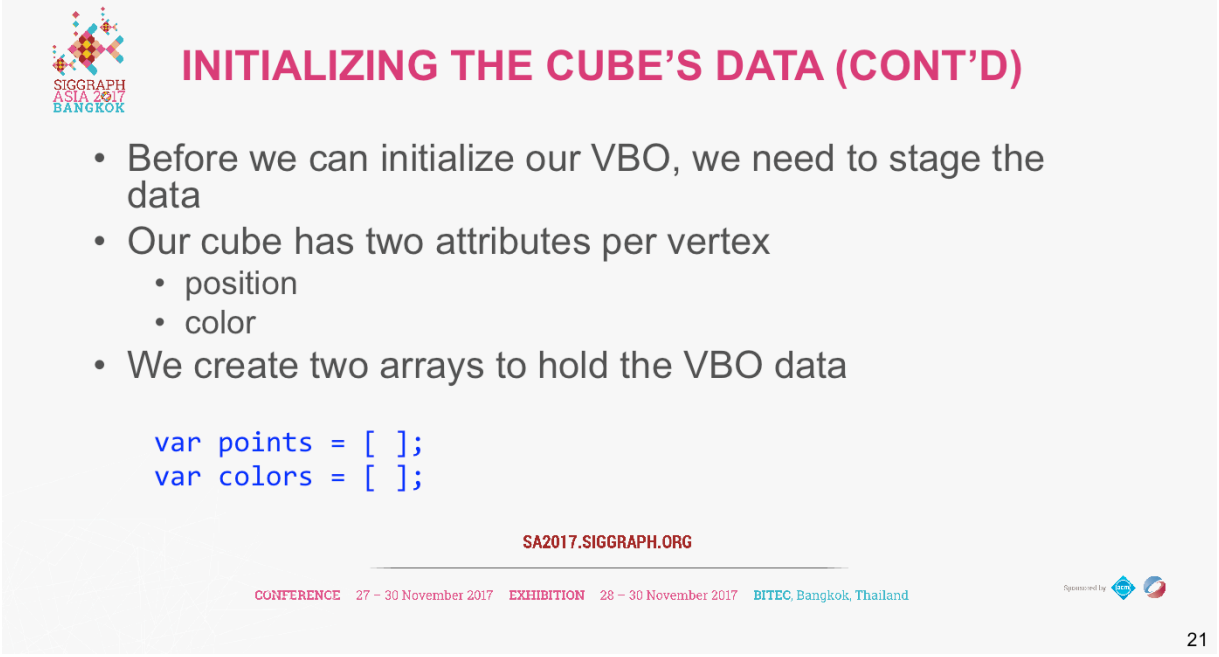


To simplify our application development, we define a few types and constants to make our code more readable and organized.

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

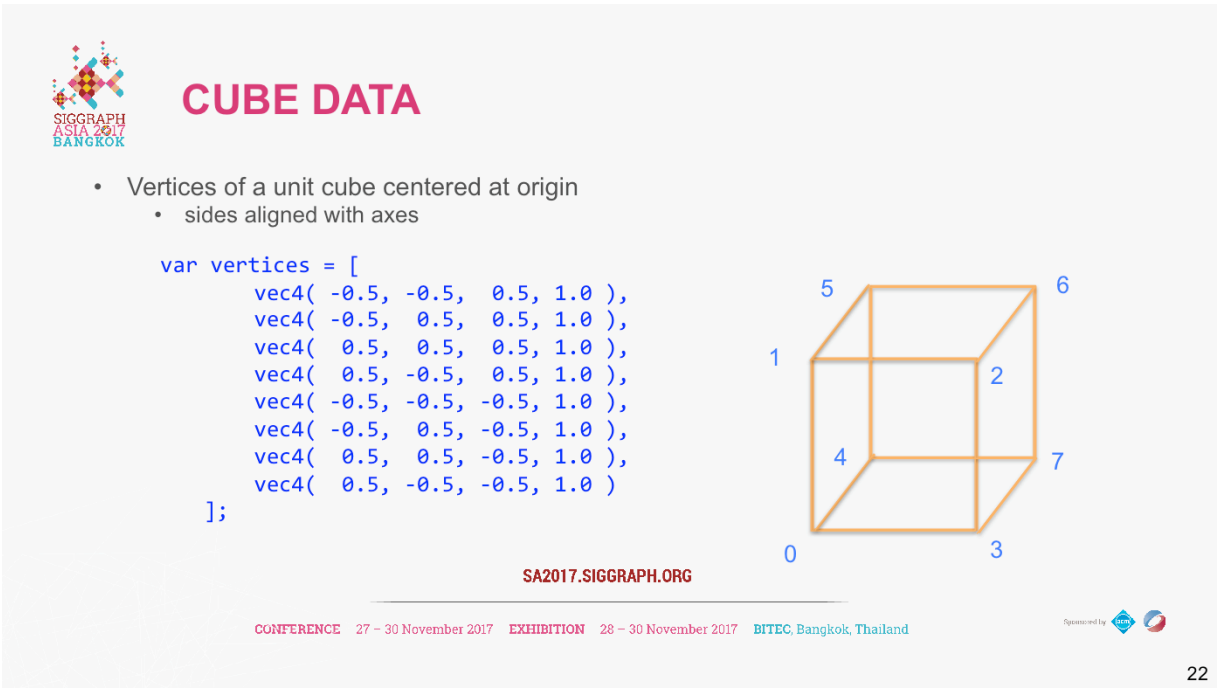
As we shall see, GLSL has vec2, vec3 and vec4 types. All are stored as four element arrays: [x, y, z, w]. The default for vec2's is to set z = 0 and w =1. For vec3's the default is to set w = 1.

MV.js also contains many matrix and viewing functions. The package is available on the course website or at www.cs.unm.edu/~angel/WebGL. MV.js is not necessary for writing WebGL applications but its functions simplify development of 3D applications.



To provide data for WebGL 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.

We note that JavaSript arrays are objects and are not equivalent to simple C/C++/Java arrays. JS arrays are objects with attributes and methods.



22

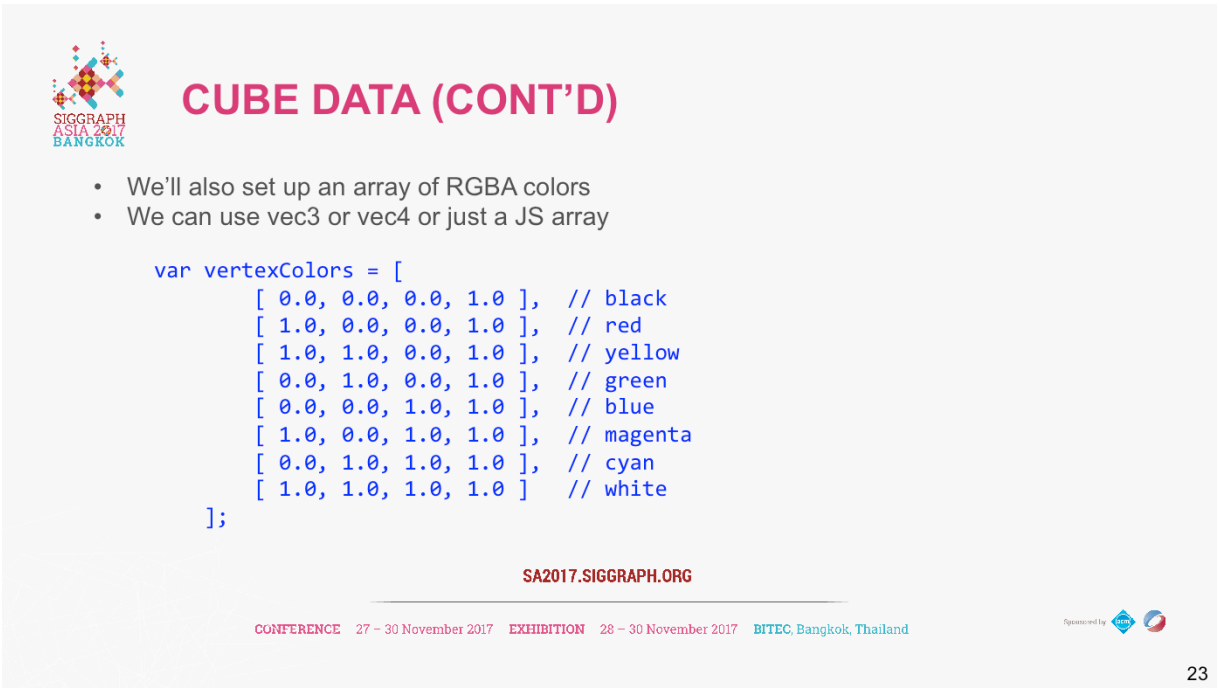
In our example we'll copy the coordinates of our cube model into a VBO for WebGLto use. Here we set up an array of eight coordinates for the corners of a unit cube centered at the origin.

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 vec3 type, i.e.

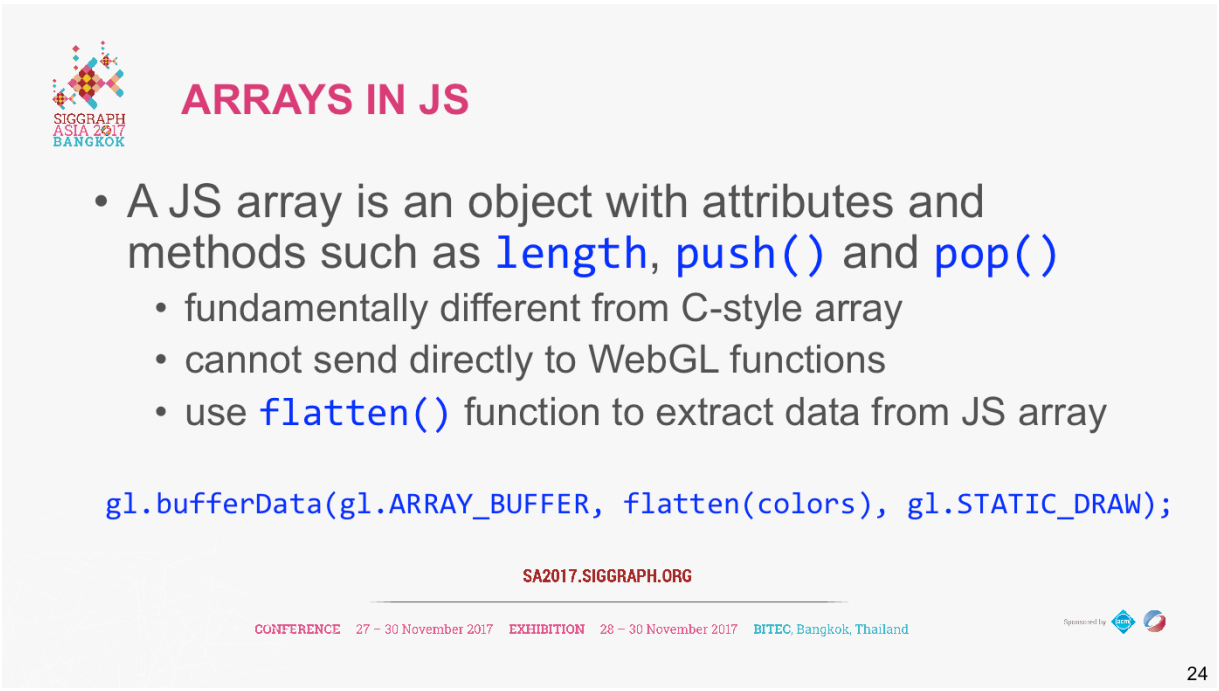
vec3(-0.5, -0.5, 0.5)

which will be stored in 4 dimensions on the GPU.

In this example, we will again use the default camera so our vertices all fit within the default view volume.

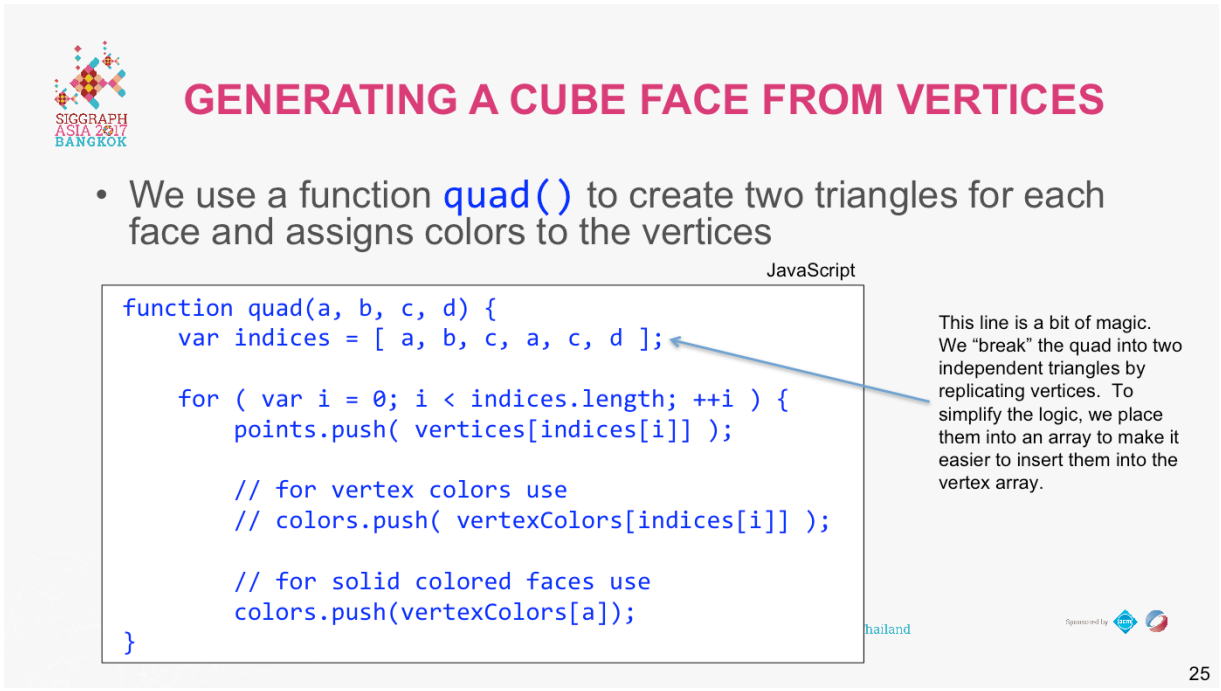


Just like our positional data, we'll set up a matching set of colors for each of the model's vertices, which we'll later copy into our VBO. Here we set up eight RGBA colors. In WebGL, colors are processed in the pipeline as floating-point values in the range [0.0, 1.0]. Your input data can take any for;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



flatten() is in MV.js.

Alternately, we could use typed arrays as we did for the triangle example and avoid the use of flatten for one-dimensional arrays. However. we will still need to convert matrices from two-dimensional to one-dimensional arrays to send them to the shaders. In addition, there are potential efficiency differences between using JS arrays vs typed arrays. It's a very small change to use typed Arrays in MV.js.



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.

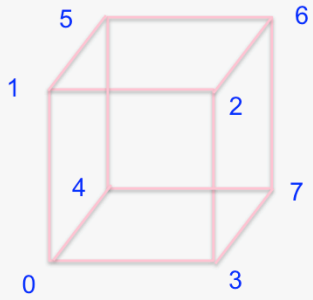
Note the use of the array method push() so we do not have to use indices for the point and color array elements

GENERATING THE CUBE FROM FACES

- Generate 12 triangles for the cube
  - 36 vertices with 36 colors

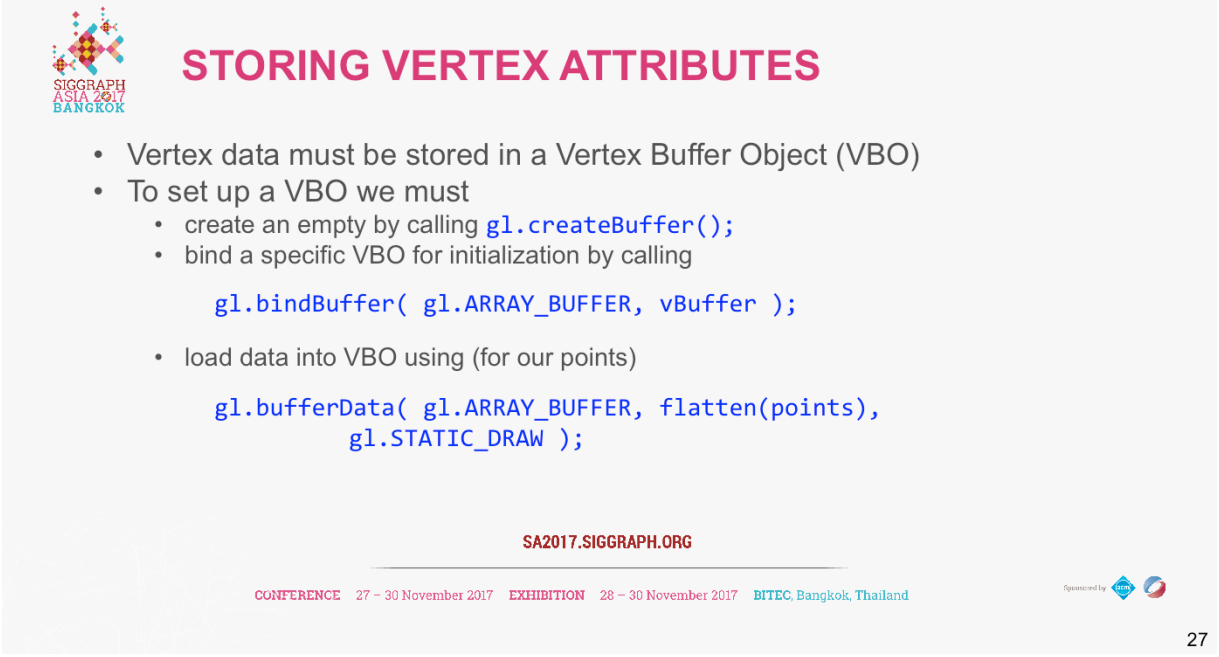
JavaScript

```
function colorCube() {
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```



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.

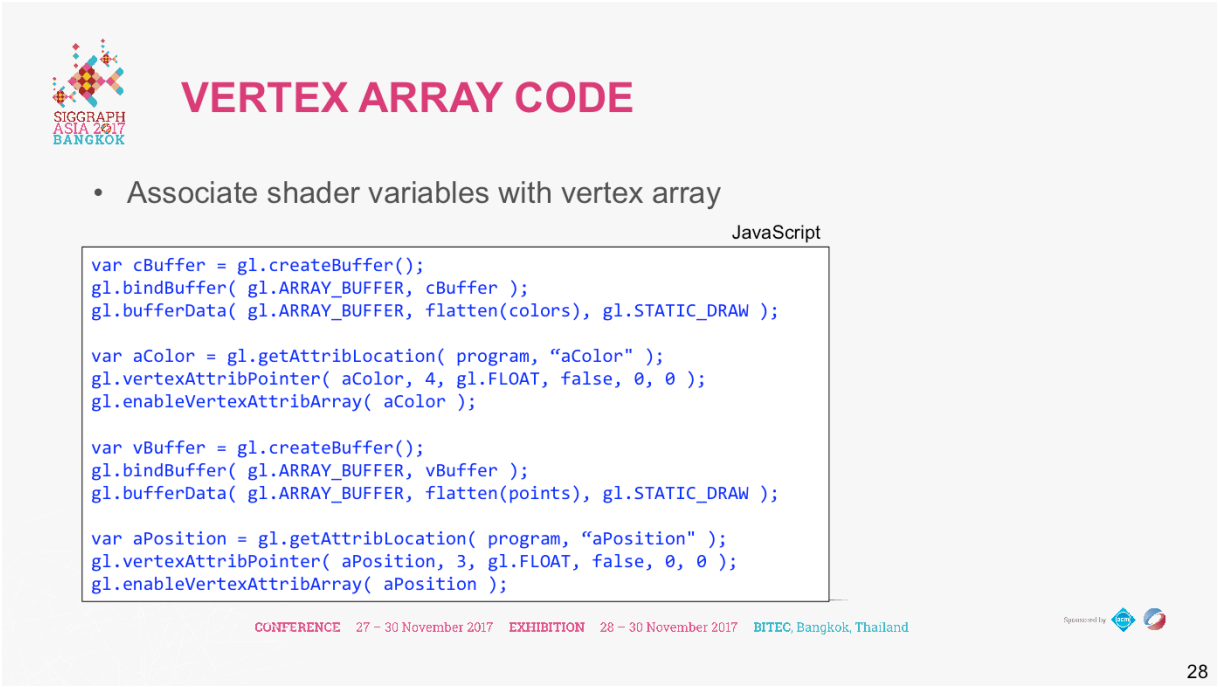


While we've talked a lot about VBOs, we haven't detailed how one goes about creating them. Vertex buffer objects, like all (memory) objects in WebGL (as compared to geometric objects) are created in the same way, using the same set of functions. In fact, you'll see that the pattern of calls we make here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

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

1. Generate a buffer's by calling `gl.createBuffer()`
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 `gl.bindBuffer()`, 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 `gl.bufferData()`, 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 `gl.bindVertexArray()` to make it and its VBOs current again.

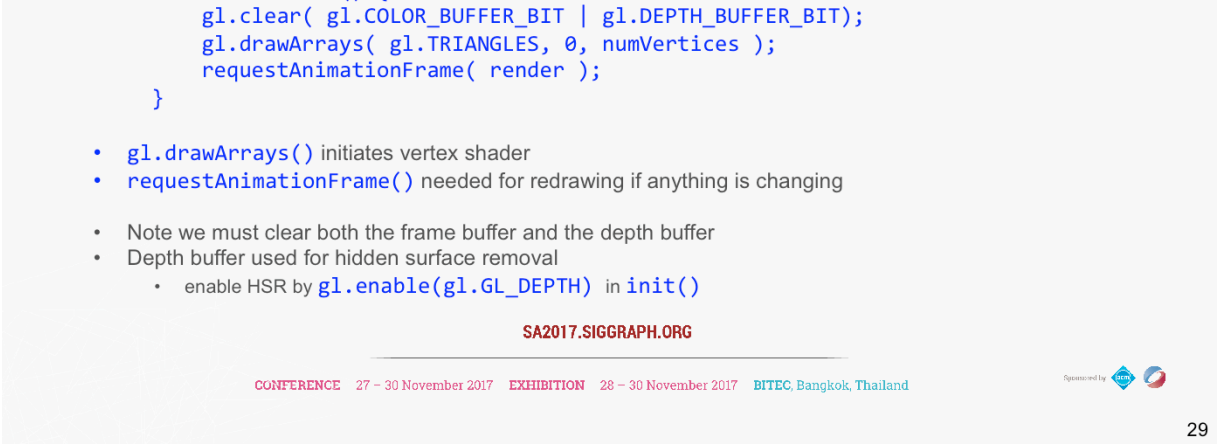
We can replace part of the data in a buffer with gl.bufferSubData()



To complete the “plumbing” of associating our vertex data with variables in our shader programs, you need to tell WebGL 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: `aPosition`, and `aColor`, which we will associate with the data values in our VBOs that we copied form our vertex `positions` and `colors` arrays.

The calls to `gl.getAttribLocation()` 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 `gl.enableVertexAttribArray()` with the selected attribute location.

Here we use the flatten function to extract the data from the JS arrays and put them into the simple form expected by the WebGL functions, basically one dimensional C-style arrays of floats.



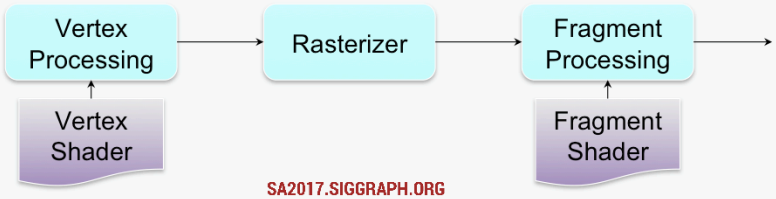
To initiate the rendering of primitives, you need to issue a drawing routine. While there are many routines for this in OpenGL, we'll discuss the most fundamental ones. The simplest routine is `glDrawArrays()`, to which you specify what type of graphics primitive you want to draw (e.g., here we're rending triangles), which vertex in the enabled vertex attribute arrays to start with, and how many vertices to send. If we use triangle strips or triangle fans, we only need to store four vertices for each face of the cube rather than six.

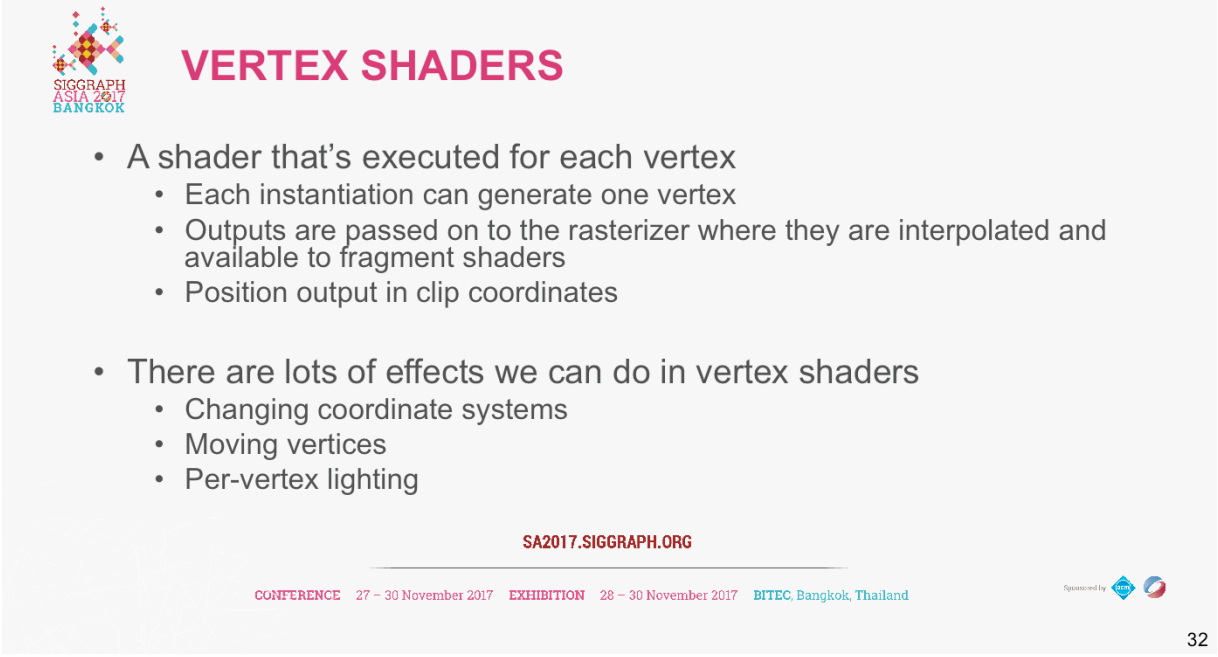
This is the simplest way of rendering geometry in WebGL. You merely need to store you vertex data in sequence, and then `gl.drawArrays()` takes care of the rest. However, in some cases, this won't be the most memory efficient method of doing things. Many geometric objects share vertices between geometric primitives, and with this method, you need to replicate the data once for each vertex.

Shaders

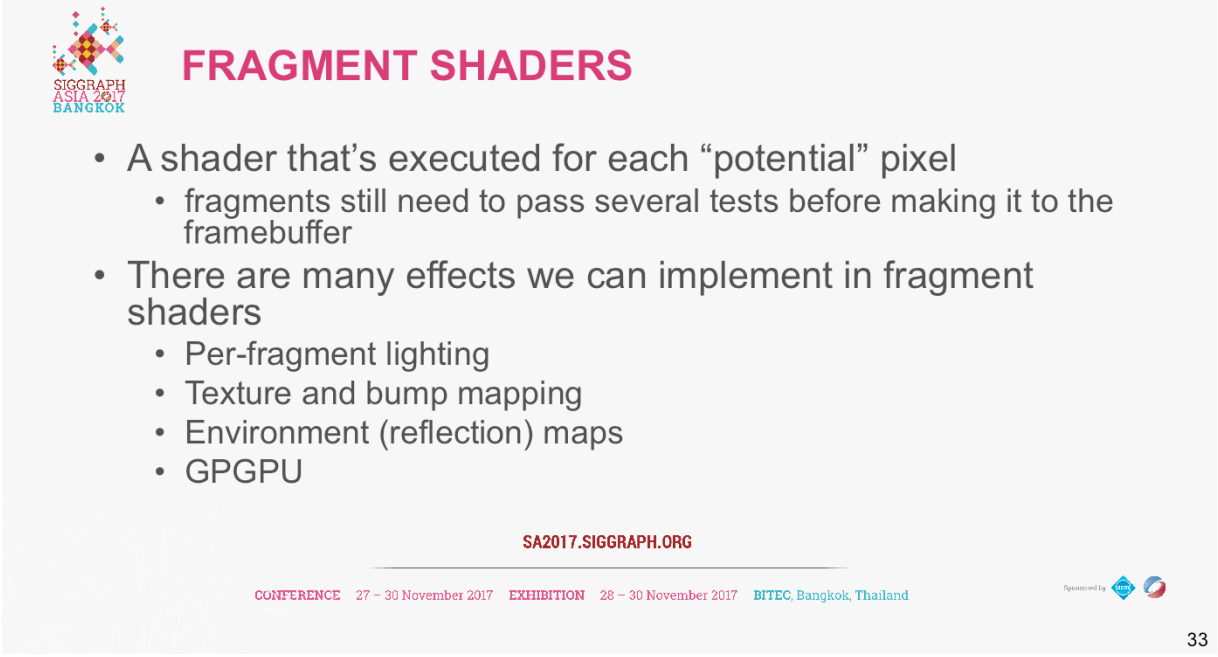
FROM FIXED FUNCTIONS TO SHADERS

- Fixed-function OpenGL had functions for viewing, transformations, texture mapping, attributes
  - Most were executed in CPU and are now deprecated
  - All these functions are now routinely executed within shaders in the GPU
- Vertex shaders deal with geometry
- Fragments shaders deal with pixels





The vertex shader the stage between the application and the raster. It operates in four dimensions and is used primarily for geometric operations such as changes in representations from the object space to the camera space and lighting computations. A vertex shader must output a position in clip coordinates or discard the vertex. It can also output other attributes such as colors and texture coordinates to the rasterizer.



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.

GLSL

- OpenGL Shading Language
  - used for OpenGL, OpenGL ES, and WebGL (with slight variations)
- C-like language with some C++ features
  - principally constructors

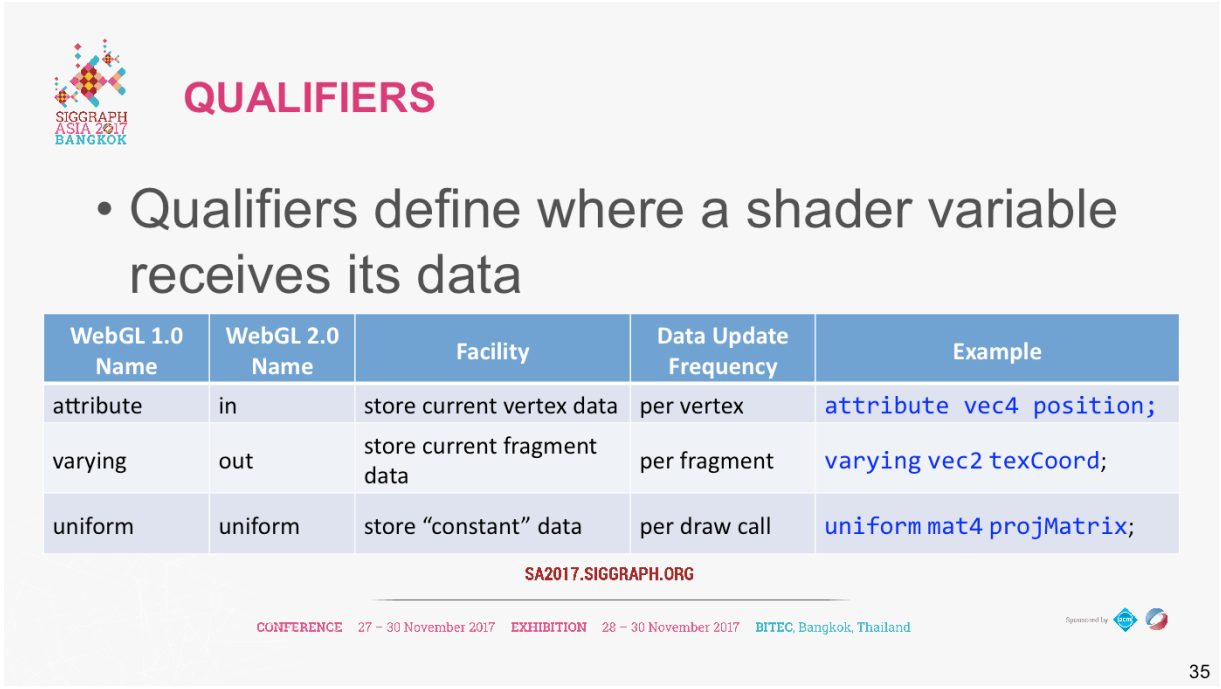
```
vec3 v = vec3(1, 2, 3);
```

- All shaders are written in GLSL
- Each shader has a `main()`
- Large library of standard functions, and support for user-defined functions

Scalar types	`float`, `int`, `bool`
Vector types	`vec2`, `vec3`, …, `ivec4`, …, `bvec4`
Matrix types	`mat2`, `mat2x3`, …, `mat4x3`, `mat4`
Texture Sampler types	`sampler2D`, `sampler3D`, `samplerCube`

Standard sensible mathematical operations defined for types:

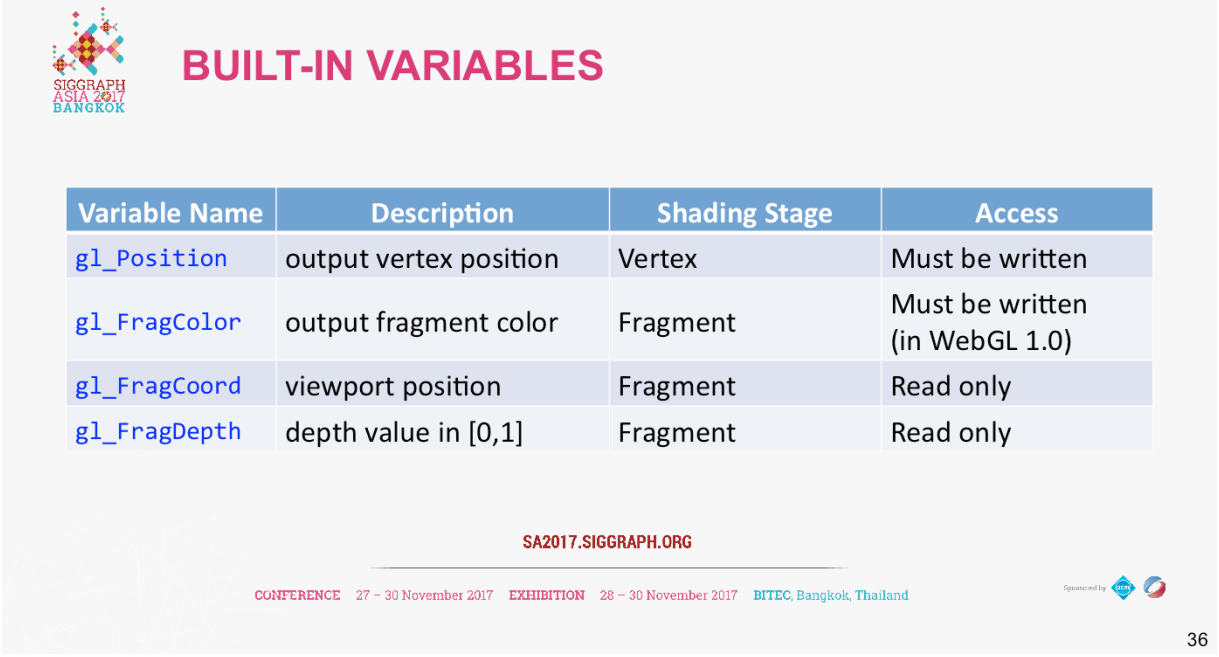
```
mat4 m;
vec4 a, b, c;

b = a * m;
c = m * a;
```



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

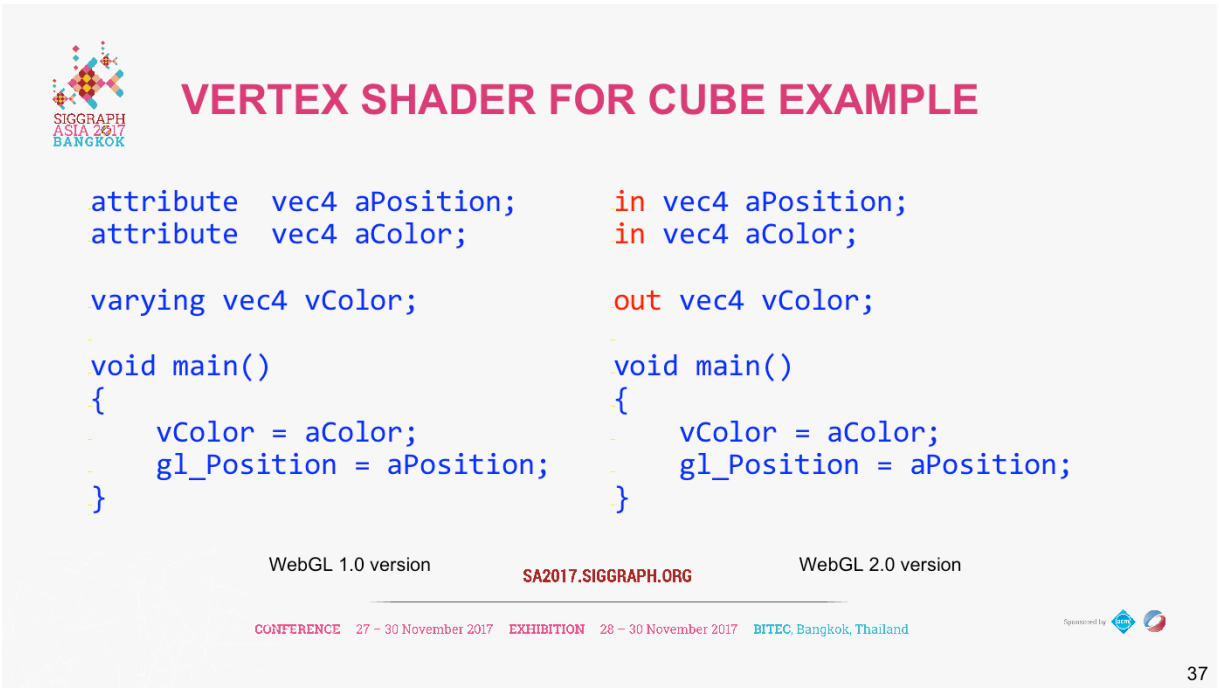
- `attribute` qualifiers indicate the shader variable will receive data flowing into the shader, either from the application,
- `varying` qualifier which tag a variable as data output where data will flow to the next shader stage
- `uniform` qualifiers for accessing data that doesn't change across a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

Recent versions of GLSL replace attribute and varying qualifiers by in and out qualifiers



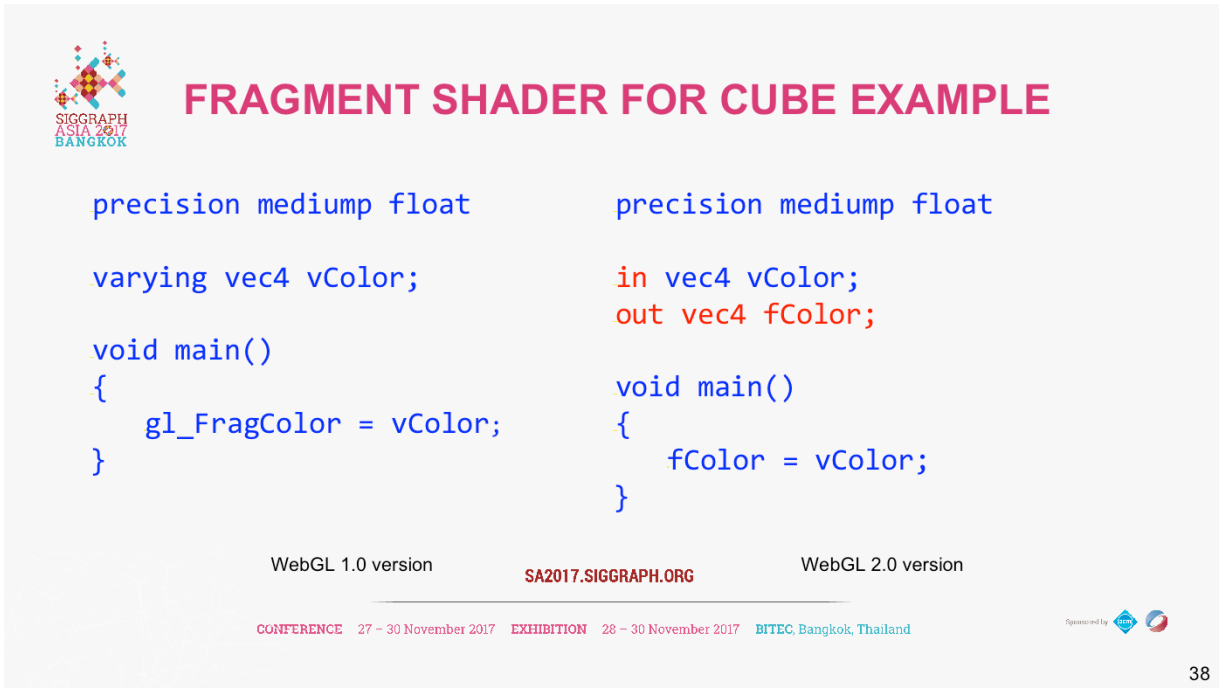
Fundamental to shader processing are a couple of built-in GLSL variable which are the terminus for operations. Vertex data, which can be processed by up to four shader stages in desktop OpenGL, are all ended by setting a positional value into the built-in variable, `gl_Position`.

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



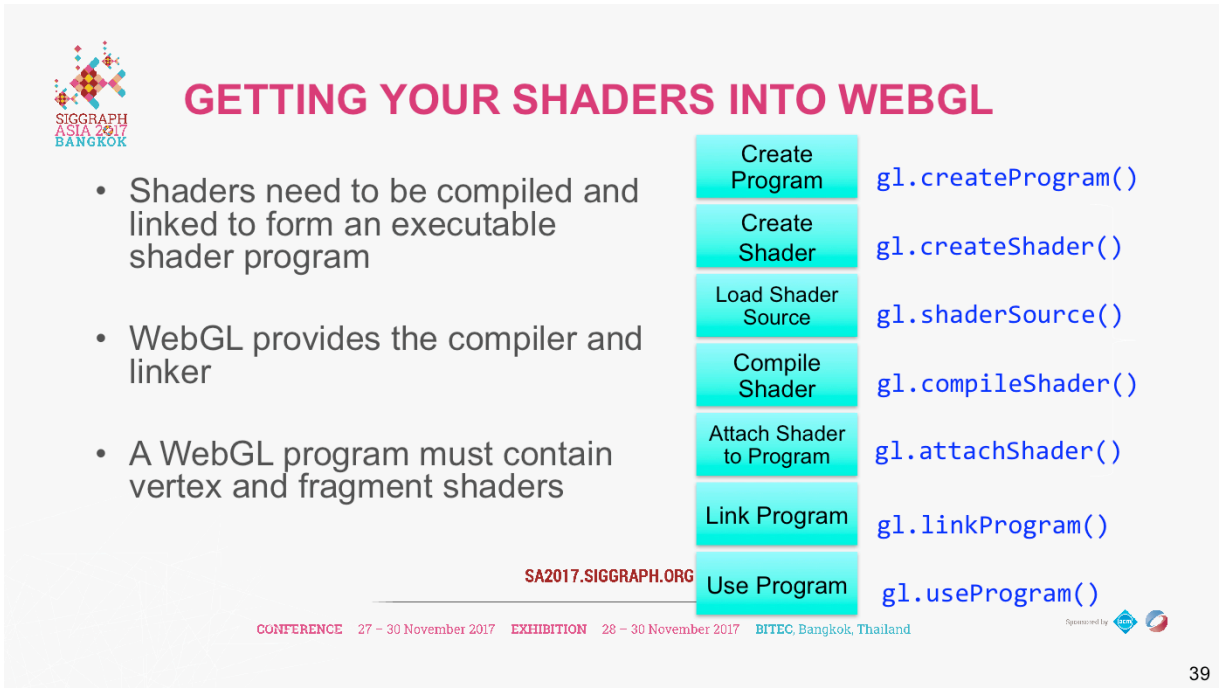
Here's the simple vertex shader we use in our cube rendering example. It accepts two vertex attributes as input: the vertex's position and color, and does very little processing on them; in fact, it merely copies the input into some output variables (with `gl_Position` being implicitly declared). The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

Our naming convention is that attributes defined in the application have their names begin with the letter a as in aPosition. Variables that are defined in the vertex shaders have names that begin with the letter v as in vColor. Variables that are defined in the fragment shader have names that begin with the letter f as in fColor.



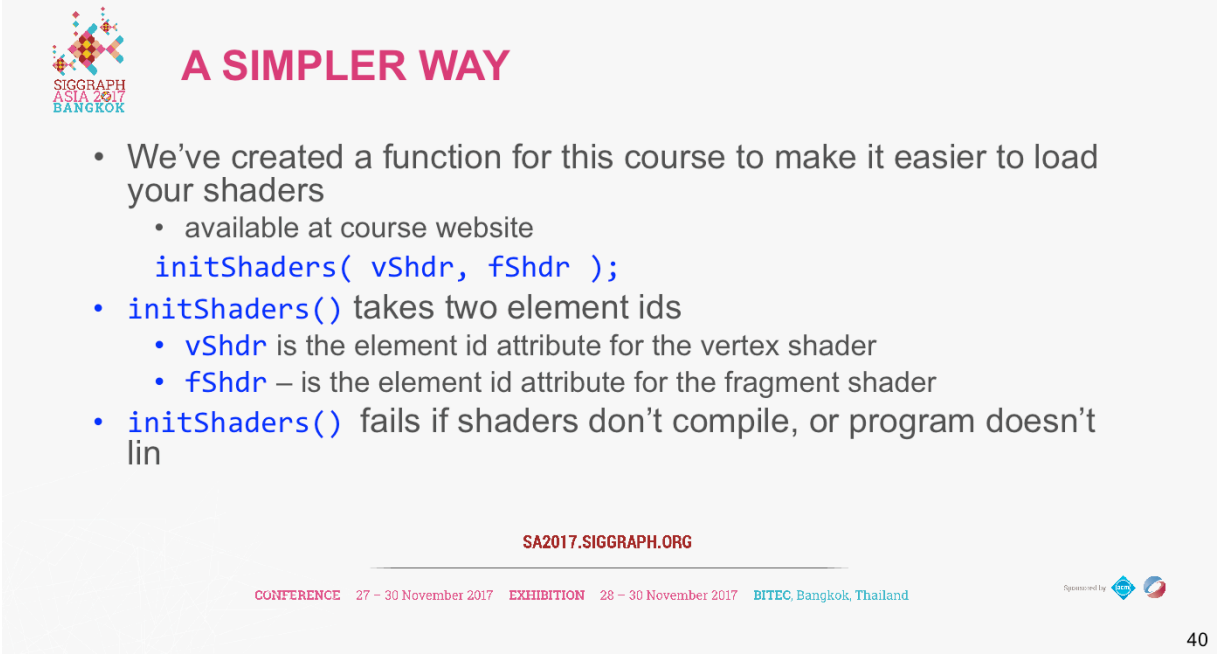
Here's the simple vertex shader we use in our cube rendering example. It accepts two vertex attributes as input: the vertex's position and color, and does very little processing on them; in fact, it merely copies the input into some output variables (with `gl_Position` being implicitly declared). The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

Our naming convention is that attributes defined in the application have their names begin with the letter a as in aPosition. Variables that are defined in the vertex shaders have names that begin with the letter v as in vColor. Variables that are defined in the fragment shader have names that begin with the letter f as in fColor.



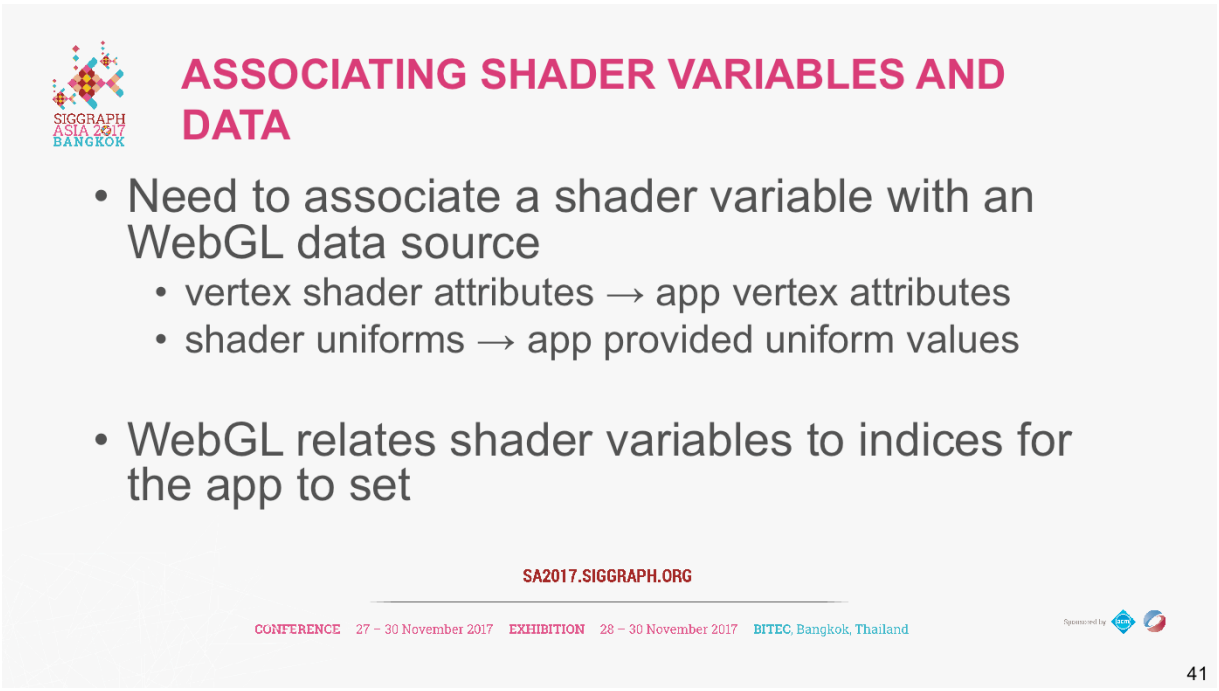
Shaders need to be compiled before they can be used in your program. As compared to C programs, the compiler and linker are implemented within WebGL , and accessible through function calls from within your program. The diagram illustrates the steps required to compile and link each type of shader into your shader program. A program must contain a vertex shader (which replaces the fixed-function vertex processing), a fragment shader (which replaces the fragment coloring stages).

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).

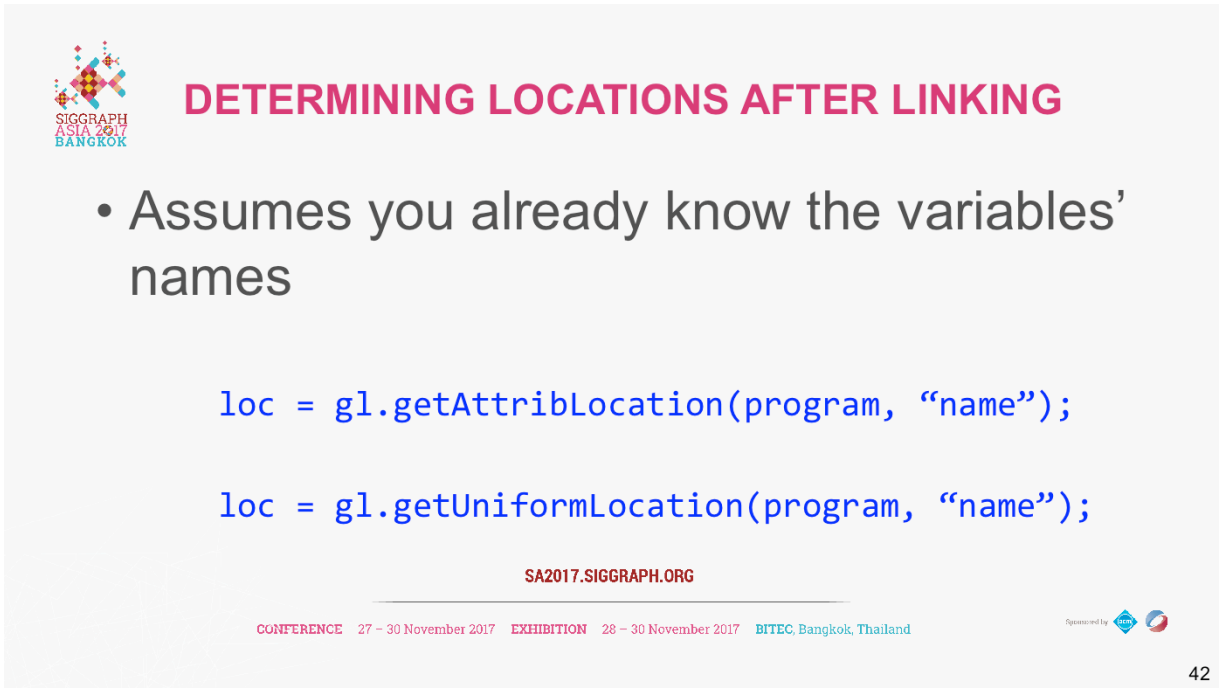
`InitShaders()` accepts two parameters, each a filename to be loaded as source for the vertex and fragment shader stages, respectively. The value returned from `InitShaders()` will be a valid GLSL program id that you can pass into `glUseProgram()`.



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() specifying 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 the shader variables by using the glGetActiveAttrib() function.

INITIALIZING UNIFORM VARIABLE VALUES

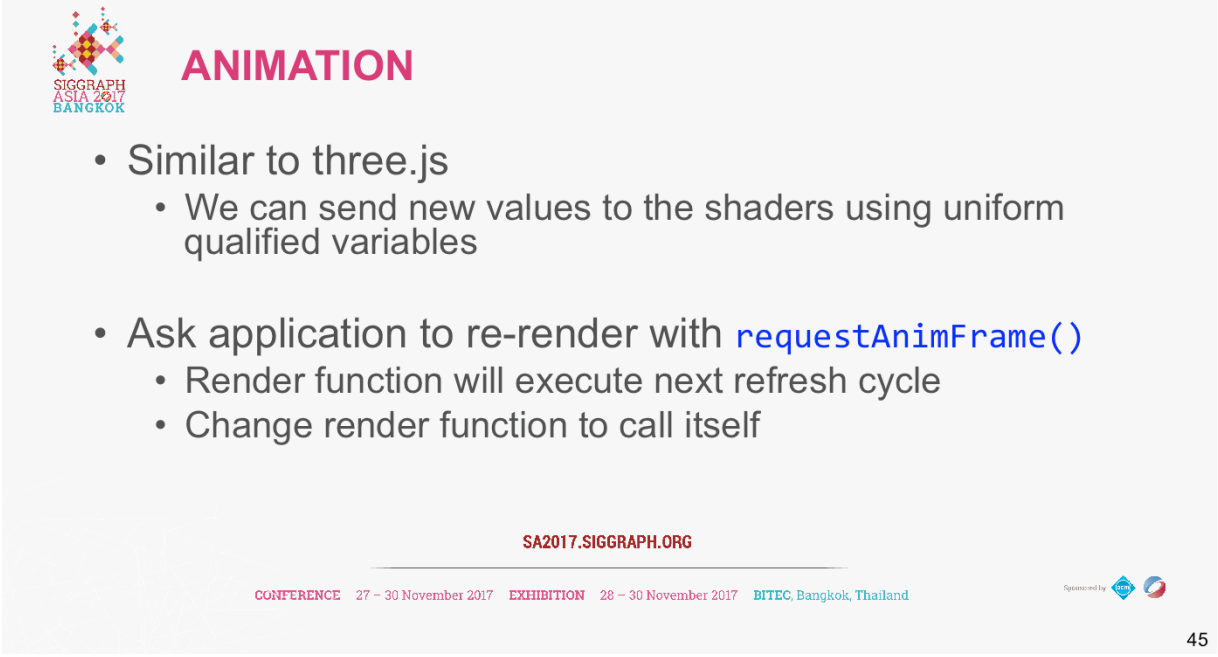
```
gl.uniform4f( index, x, y, z, w );

var mat = mat4( … );
var transpose = gl.GL_FALSE;  //required by WebGL

// Since we were C programmers we have to be
// careful to put data in column major form

gl.uniformMatrix4fv( index, 3, transpose, mat );
```


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.

Animation and Interaction

SA2017.SIGGRAPH.ORG

CONFERENCE 27 – 30 November 2017 EXHIBITION 28 – 30 November 2017 BITEC, Bangkok, Thailand



Uniform qualified variables are constant for an execution of gl.drawArrays(), i.e. are constant for each instantiation of the vertex shader.

ANIMATION EXAMPLE

- Make cube bigger and smaller sinusoidally in time

JavaScript

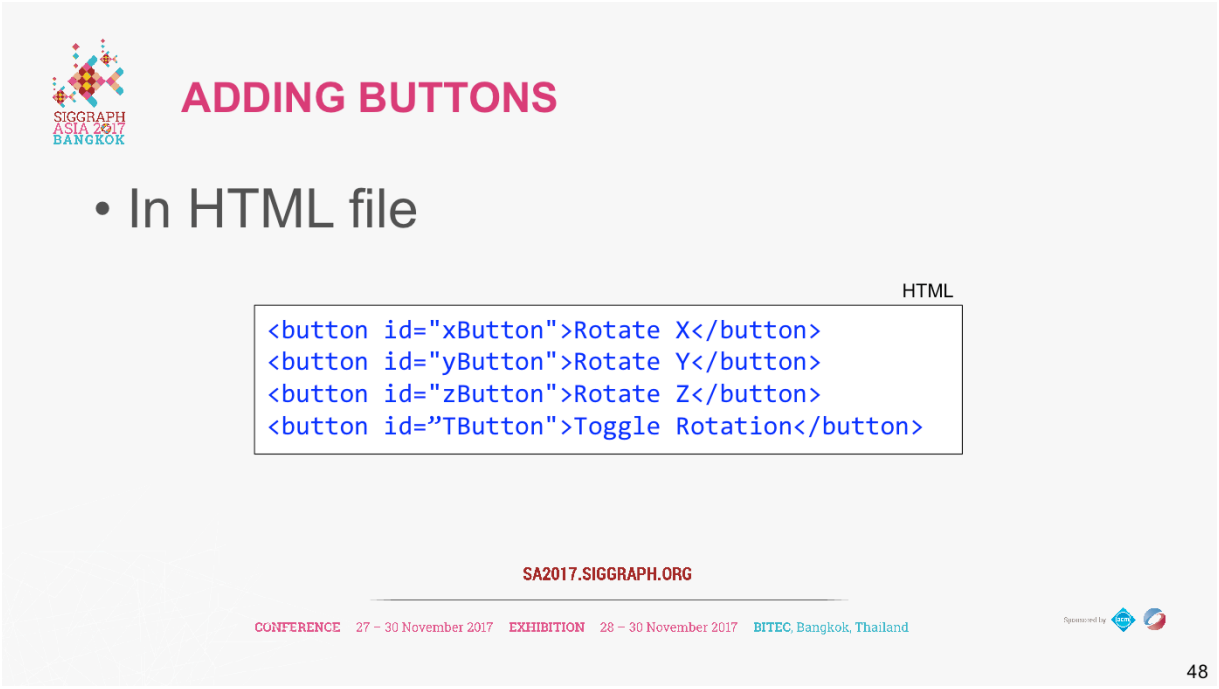
```
timeLoc = gl.getUniformLocation(program, "time");

function render() {
    gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    gl.uniform3fv( thetaLoc, theta );
    time += dt;
    gl.uniform1f( timeLoc, time );
    gl.drawArrays( gl.TRIANGLES, 0, numVertices );
    requestAnimFrame( render );
}
```

ANIMATION EXAMPLE (CONT'D)

- Make cube bigger and smaller sinusoidally in time

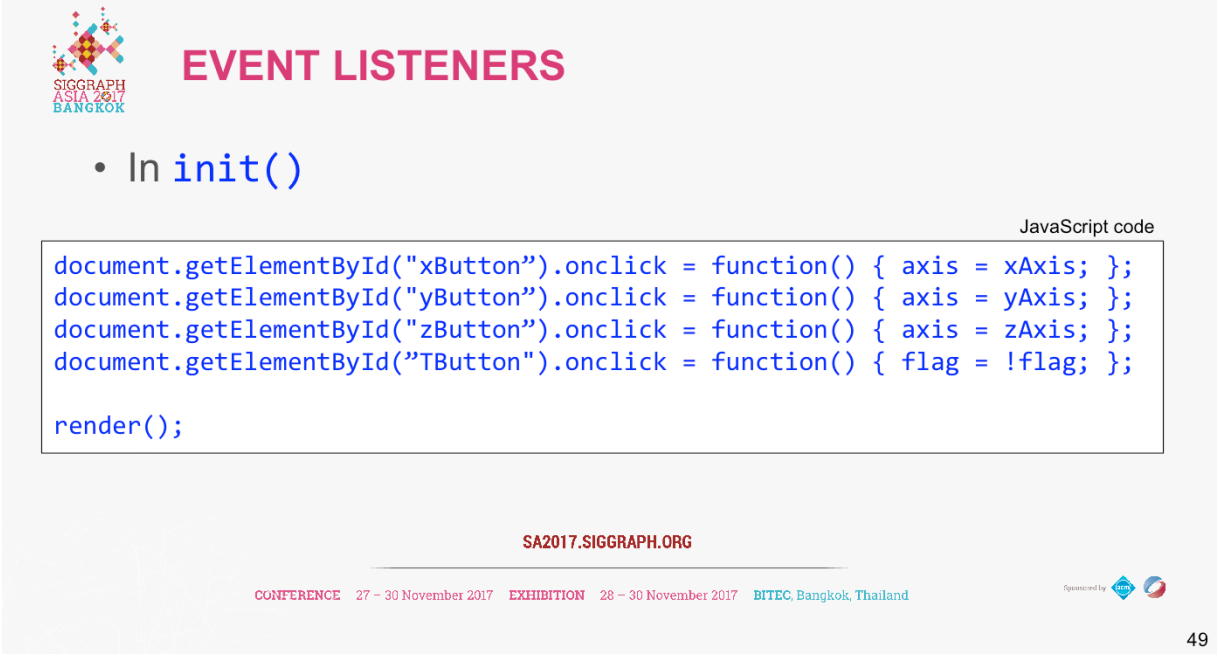
GLSL - Vertex Shader

```
attribute vec4 vPosition;
uniform float time;

void main()
{
    gl_Position = (1.0+0.5*sin(time)) * vPosition;
    gl_Position.w = 1.0;
}
```

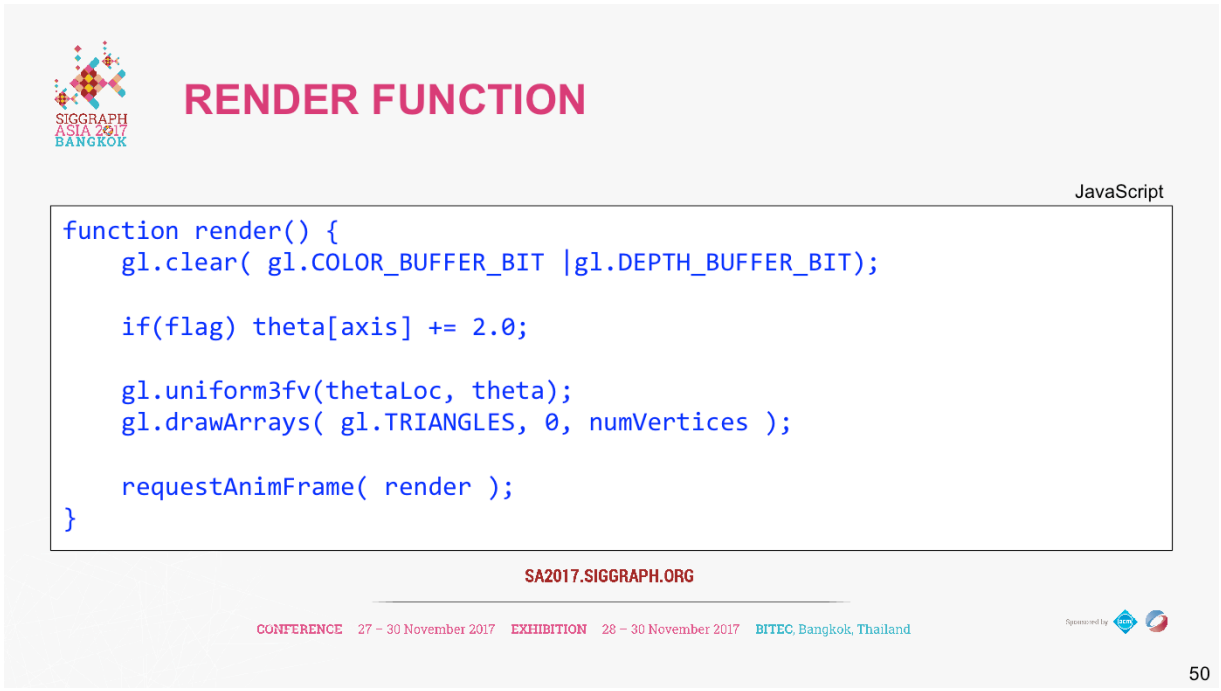


id allows us to refer to button in JS file. Text between <button> and </button> tags is placed on button. Clicking on button generates an event which we will handle with a listener in JS file.

Note buttons are part of HTML not WebGL.



The event of clicking a button (onclick) occurs on the display (document). Thus the name of the event is document.button_id.onclick. For a button, the event returns no information other than it has been clicked. The first three buttons allow us to change the axis about which we increment the angle in the render function. The variable flag is toggled between true and force. When it is true, we increment the angle in the render function.



This completes the rotating cube example.

Other interactive elements such as menus, sliders and text boxes are only slightly more complex to add since they return extra information to the listener. We can obtain position information from a mouse click in a similar manner.

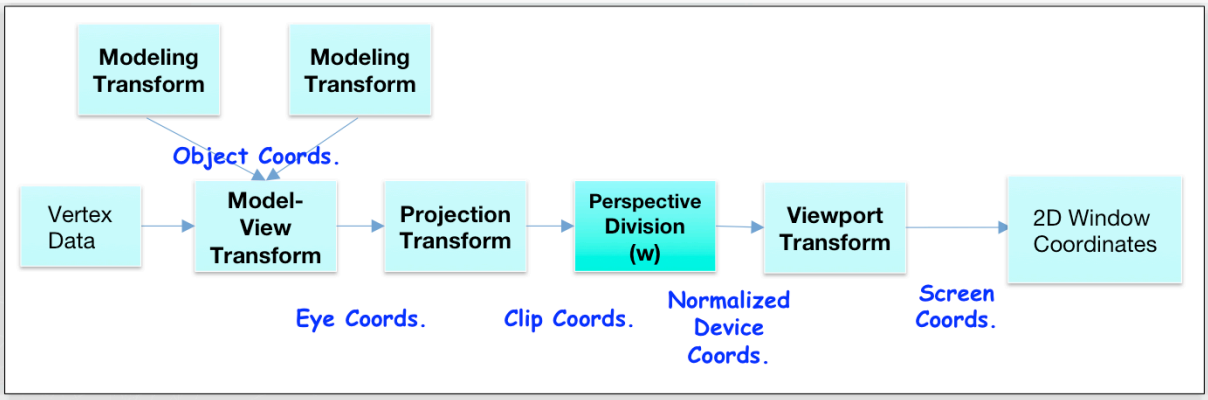
Transformations

TRANSFORMATIONS

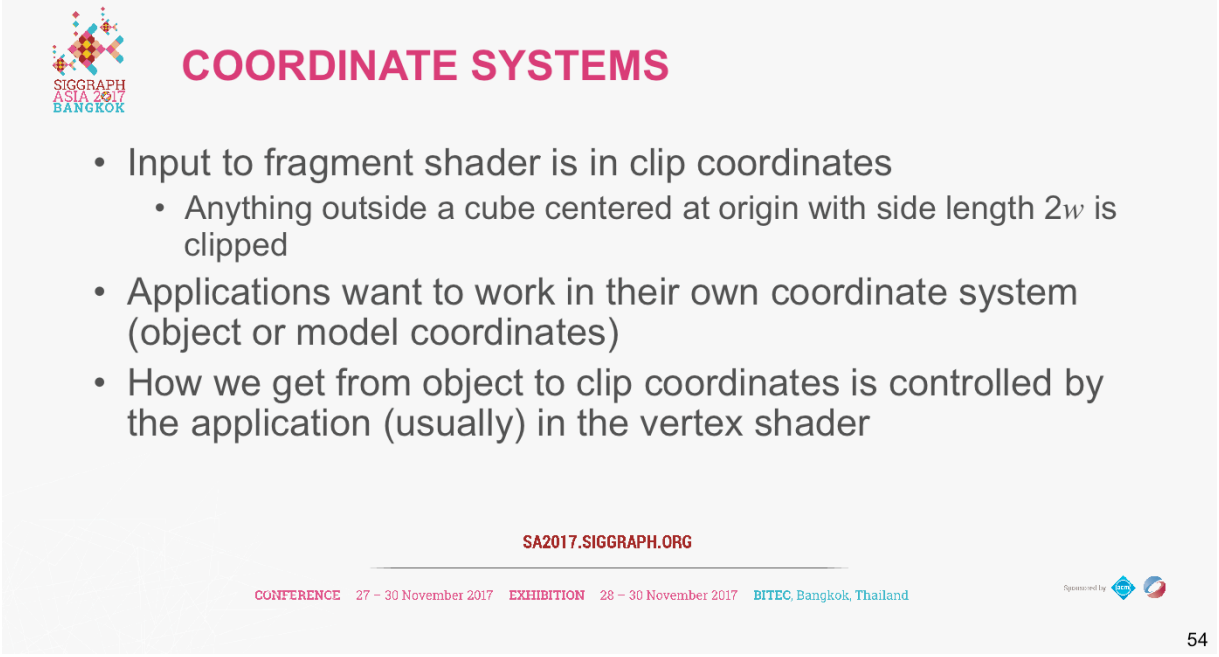
- Transformations are defined by 4 x 4 matrices
  - since positions are stored as *homogeneous coordinates*
  - remember that w term from before?
- Three primary uses:
  - viewing
  - changes in coordinate systems
  - transforming objects (e.g., rotation, translation, scaling)

COORDINATE SYSTEMS

- Transformations take us from one “space” or coordinate system (or frame) to another

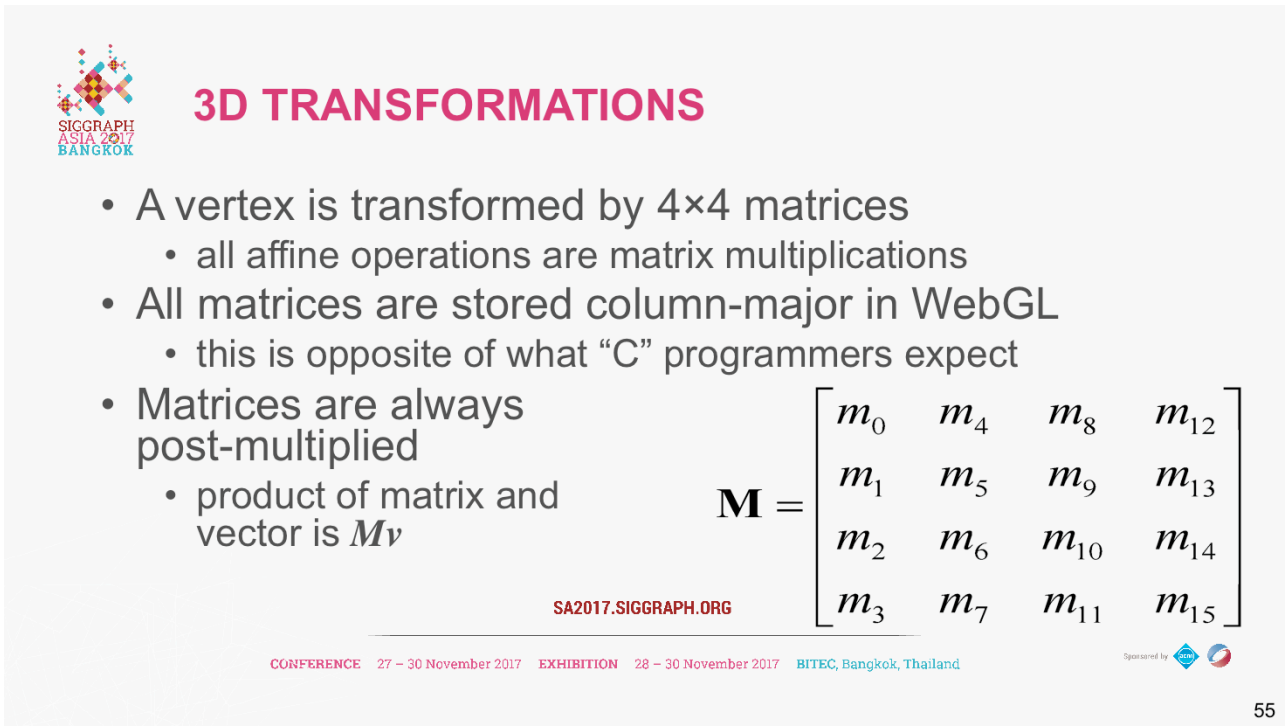


The processing required for converting a vertex from 3D or 4D space into a 2D window coordinate is done by the transform stage of the graphics pipeline. The operations in that stage are illustrated above. Each box represent a matrix multiplication operation. In graphics, all our matrices are 4×4 matrices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



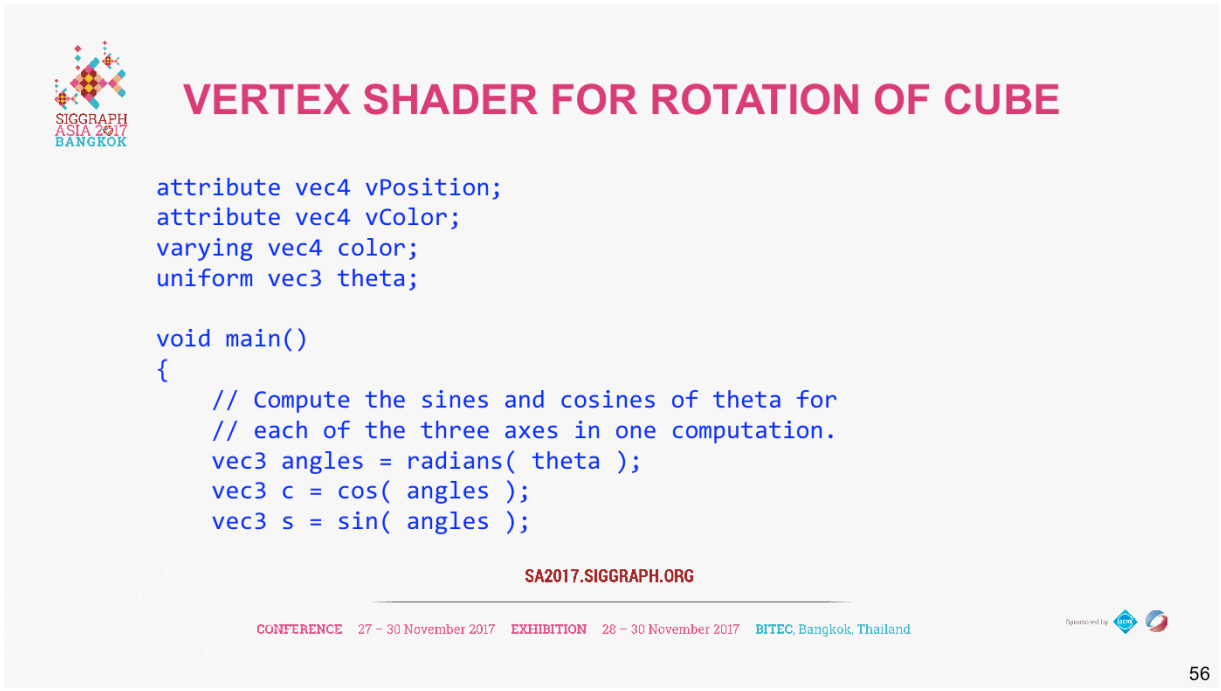
Recall that WebGL uses four dimensional homogeneous coordinates (x, y, z, w). If we use 3D in our application, w defaults to 1.

Clip coordinates and screen coordinates are the only ones required by WebGL. However, applications prefer to use their own coordinates and convert to clip coordinates in the vertex shader.



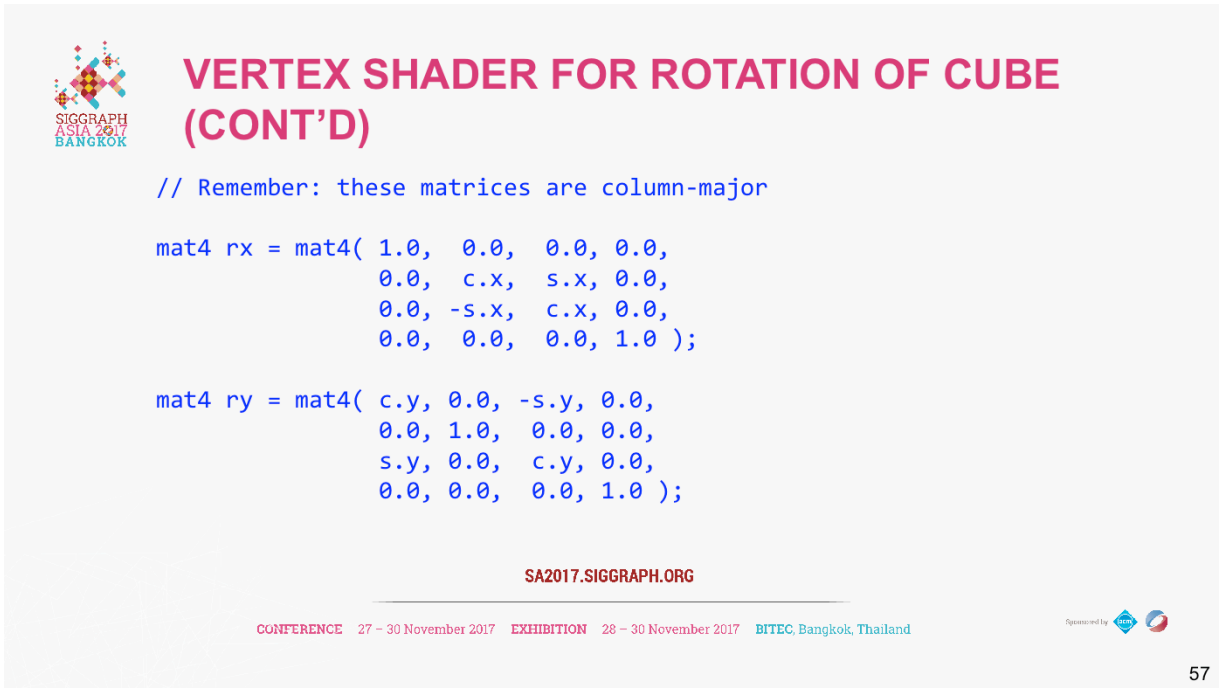
By using 4×4 matrices, OpenGL can represent all geometric transformations using one matrix format. Perspective projections and translations require the 4th row and column. Otherwise, these operations would require an vector-addition operation, in addition to the matrix multiplication.

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.

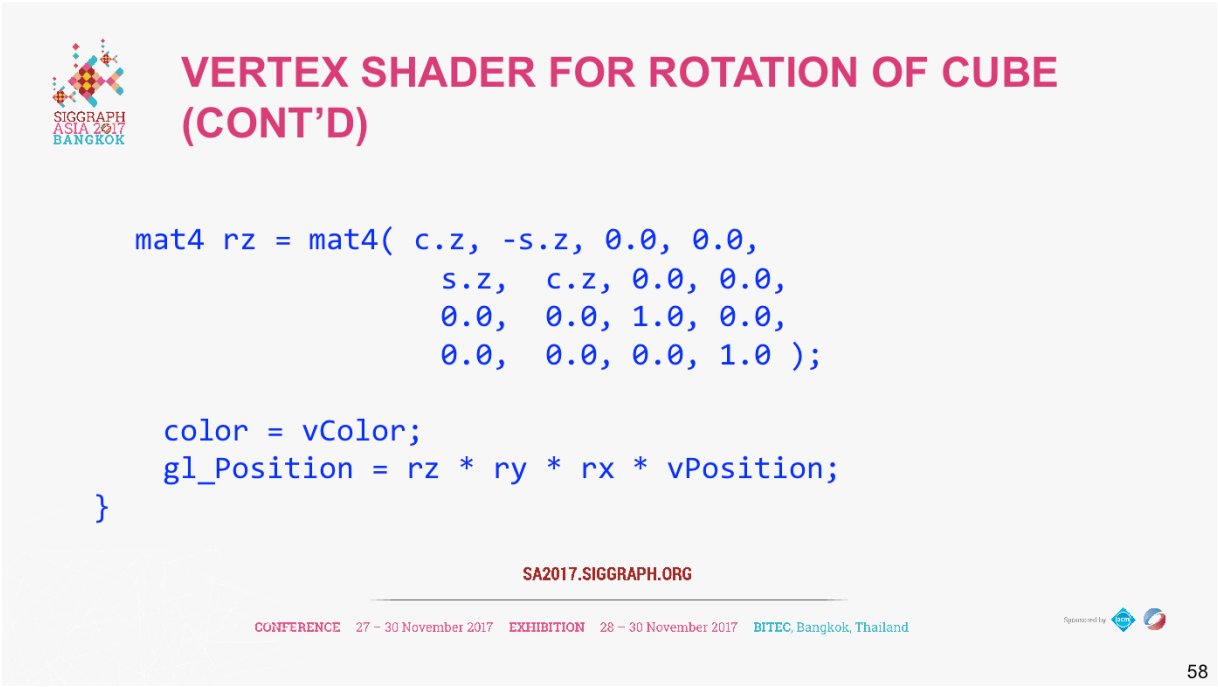


Here's an example vertex shader for rotating our cube. We generate the matrices in the shader (as compared to in the application), based on the input angle `theta`. It's useful to note that we can vectorize numerous computations. For example, we can generate a vector of sines and cosines for the input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

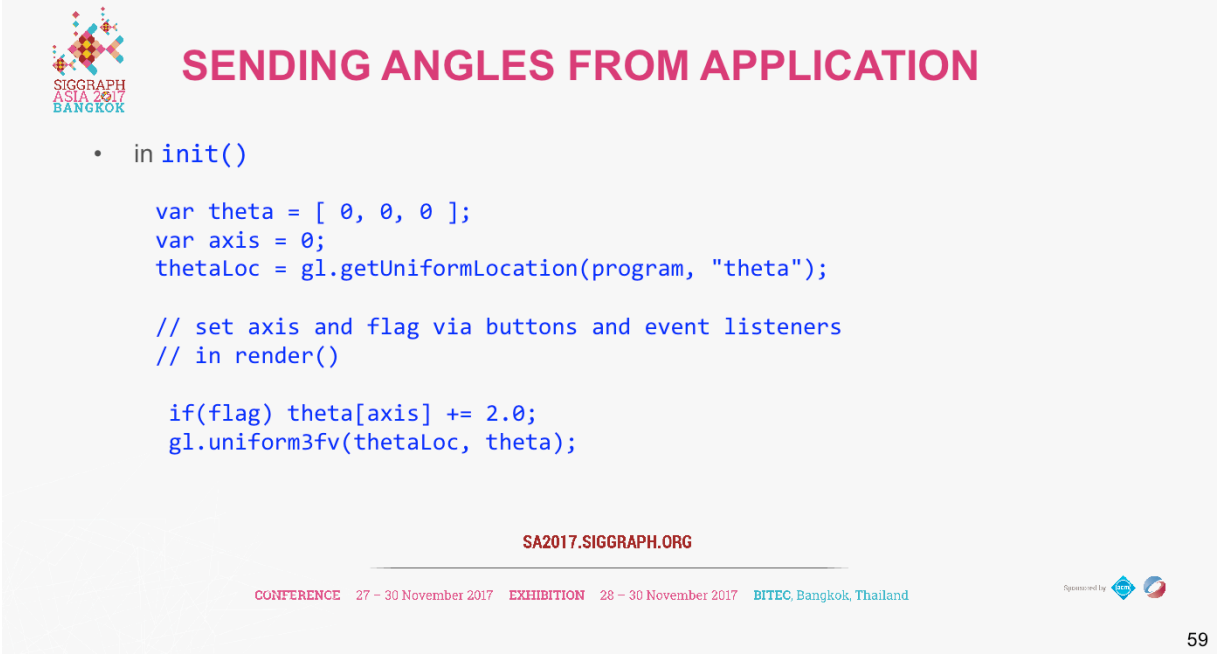
This example is but one way to use the shaders to carry out transformations. We could compute the transformation in the application each iteration and send it to the shader as a uniform. Which is best can depend on the speed of the GPU and how much other work we need to do in the CPU.



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.



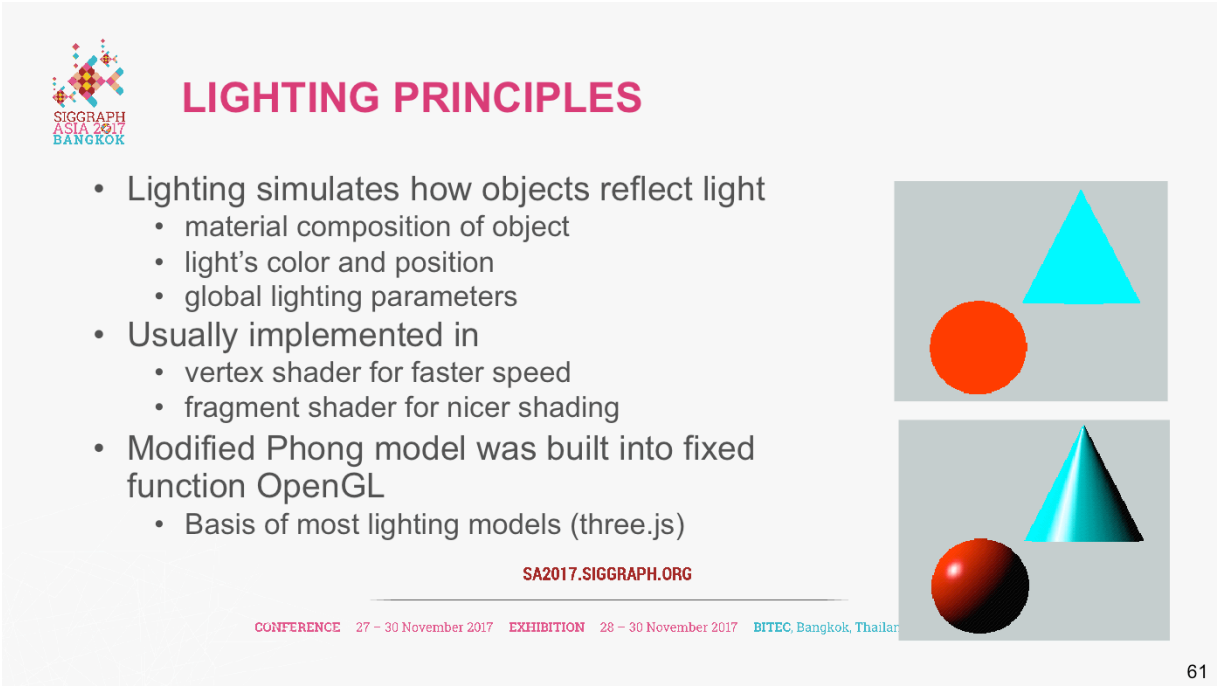
We complete our shader here by generating the last rotation matrix and then using the composition of those matrices to transform the input vertex position. We also *pass-thru* the color values by assigning the input color to an output variable.



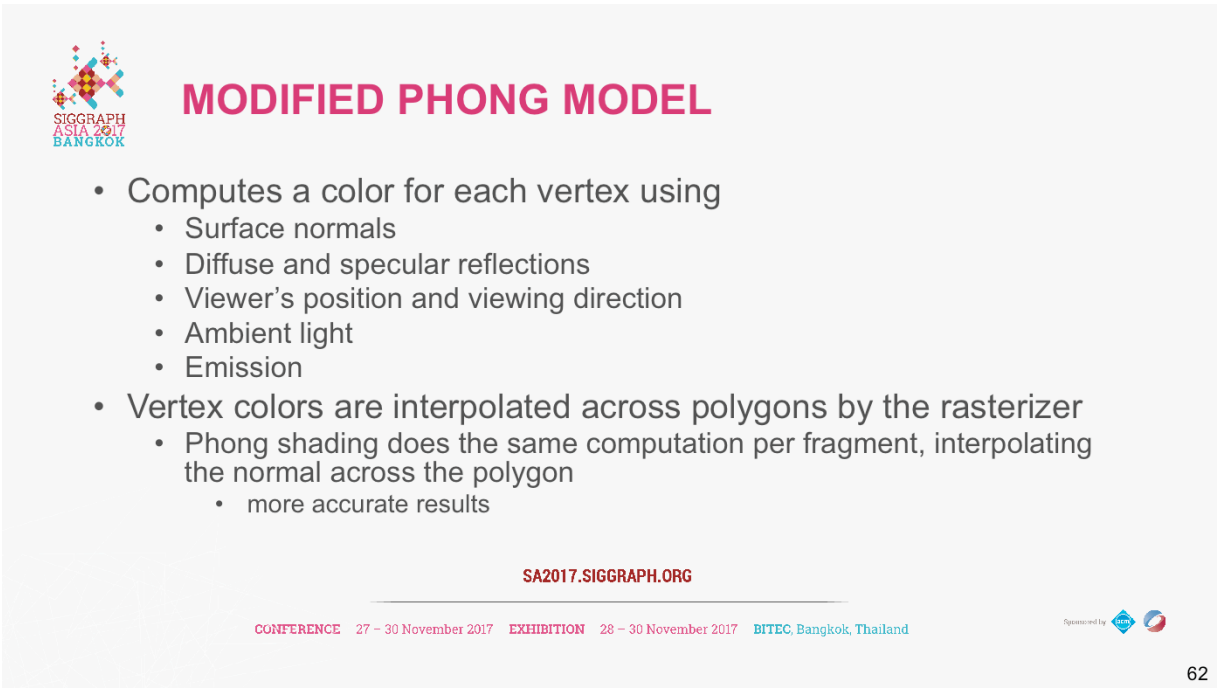
Finally, we merely need to supply the angle values into our shader through our uniform plumbing. In this case, we track each of the axes rotation angle, and store them in a `vec3` that matches the angle declaration in the shader. We also keep track of the uniform's location so we can easily update its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

This is not the only approach. We could also have generated the transformation matrix in the application and send it to the vertex shader as a uniform, which may be more efficient. Either way, the transformation approach should be better than transforming all the vertices in the application and resending them to the shaders.

Lighting

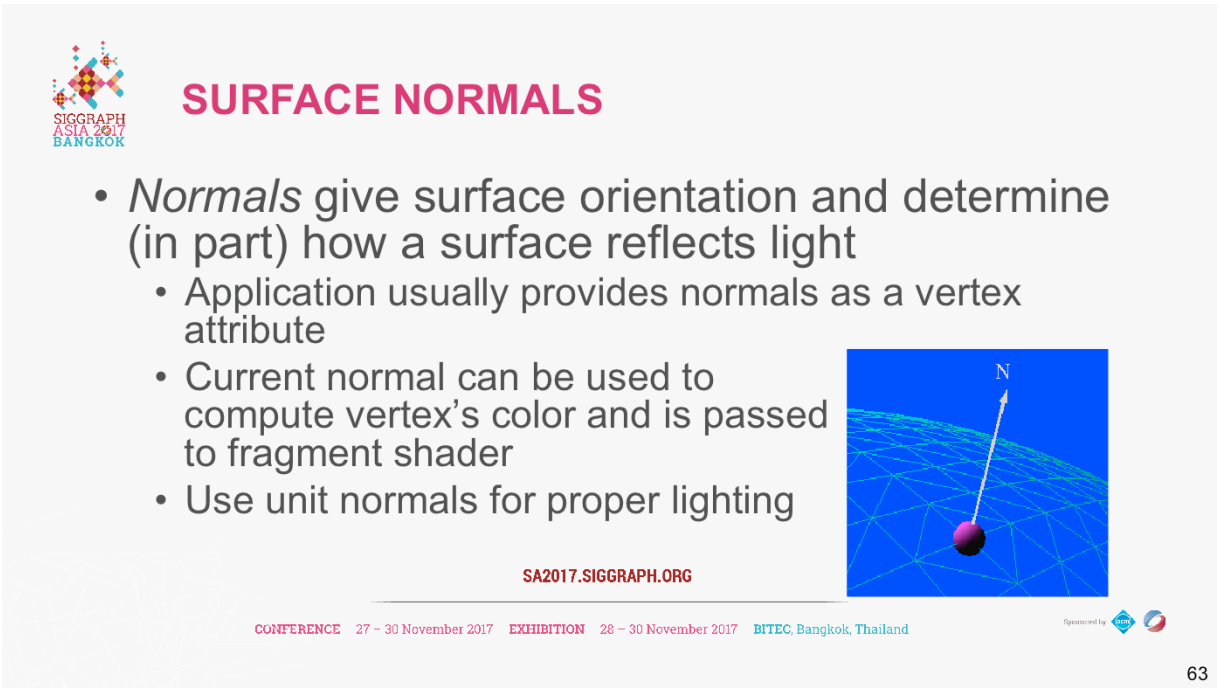


Lighting is an important technique in computer graphics. Without lighting, objects tend to look like they are made from plastic.
The models used in most WebGL applications divide lighting into three parts: material properties, light properties and global lighting parameters.
While we'll discuss the mathematics 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.

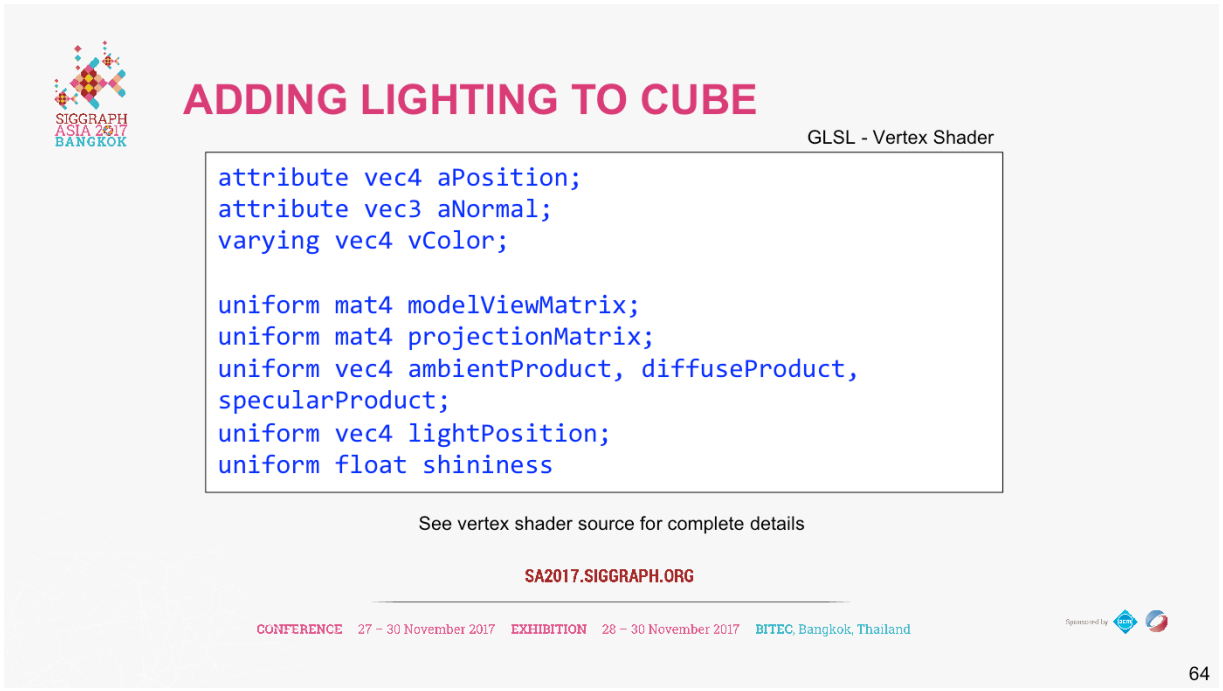


WebGL can use the shade at one vertex to shade an entire polygon (constant shading) or interpolate the shades at the vertices across the polygon (smooth shading), the default.

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

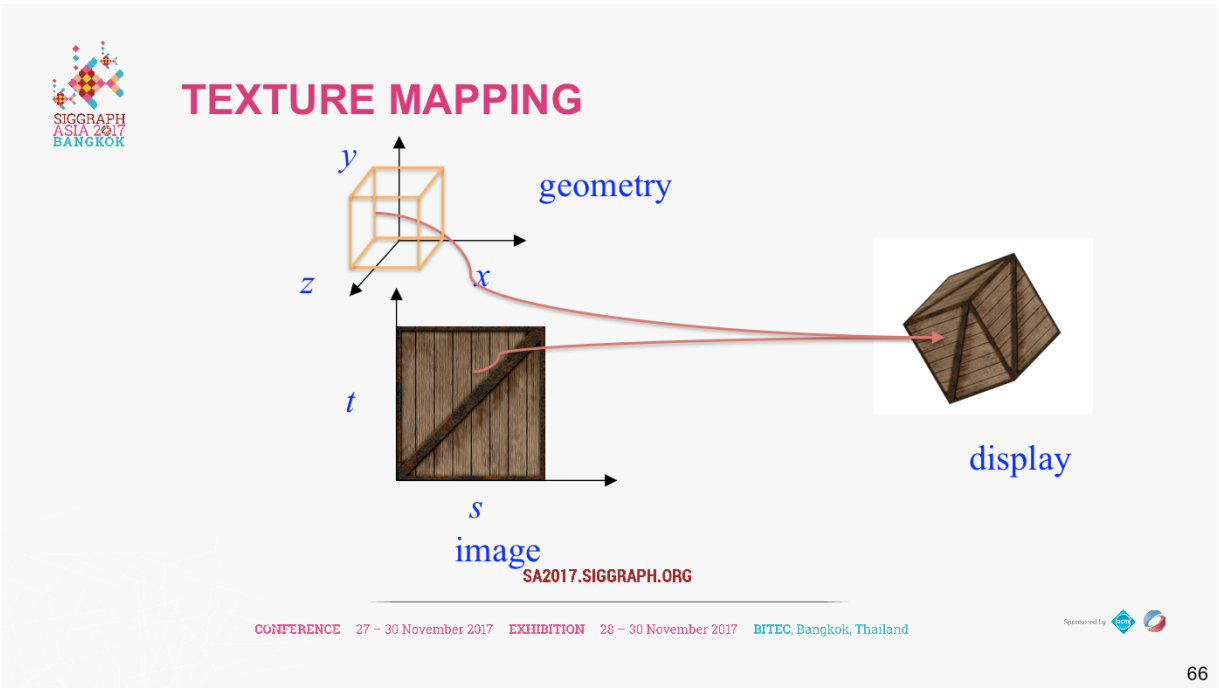


The lighting normal determines how the object reflects light around a vertex. If you imagine that there is a small mirror at the vertex, the lighting normal describes how the mirror is oriented, and consequently how light is reflected.

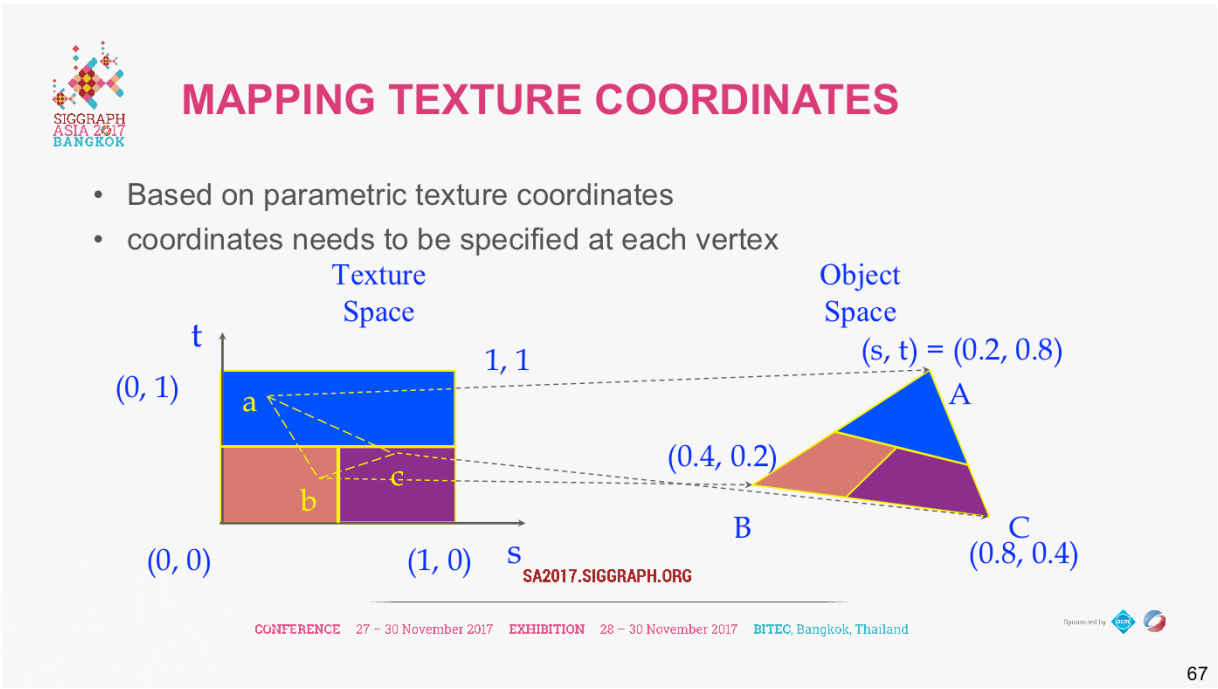


Here we declare numerous variables that we'll use in computing a color using a simple lighting model. All the uniform values are passed in from the application and describe the material and light properties being rendered. We can send these values to either the vertex or fragment shader, depending on how we want to do lighting computation, either on per vertex basis or a per fragment basis.

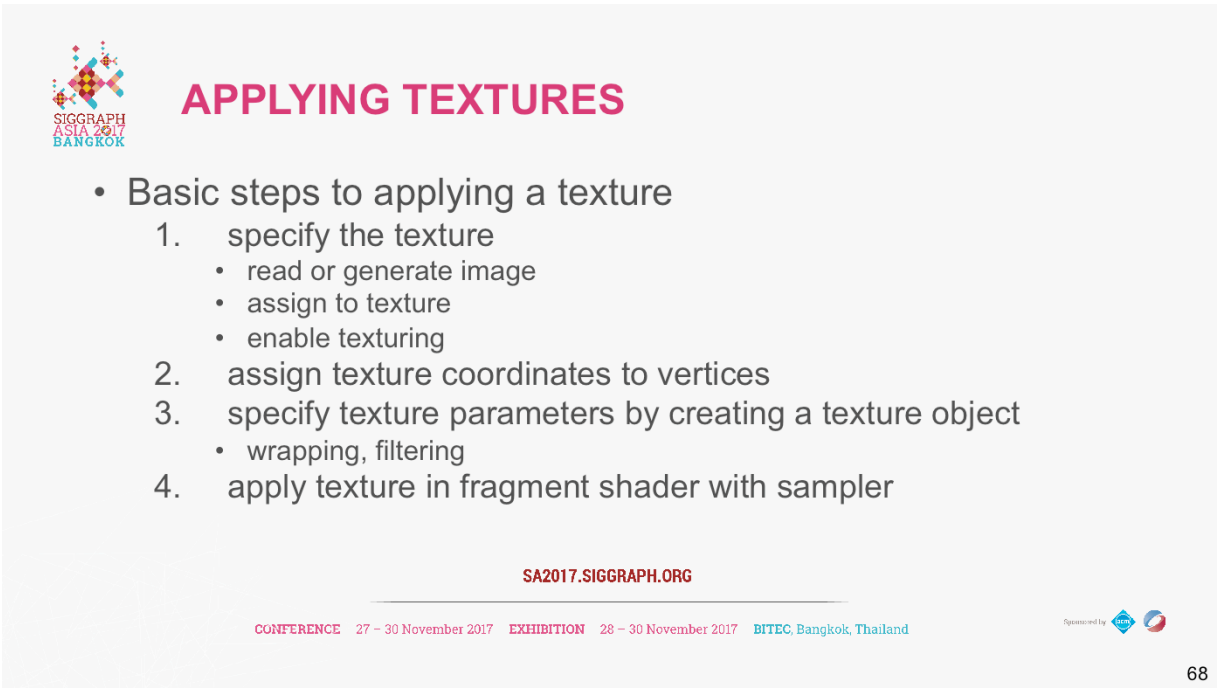
Texture Mapping



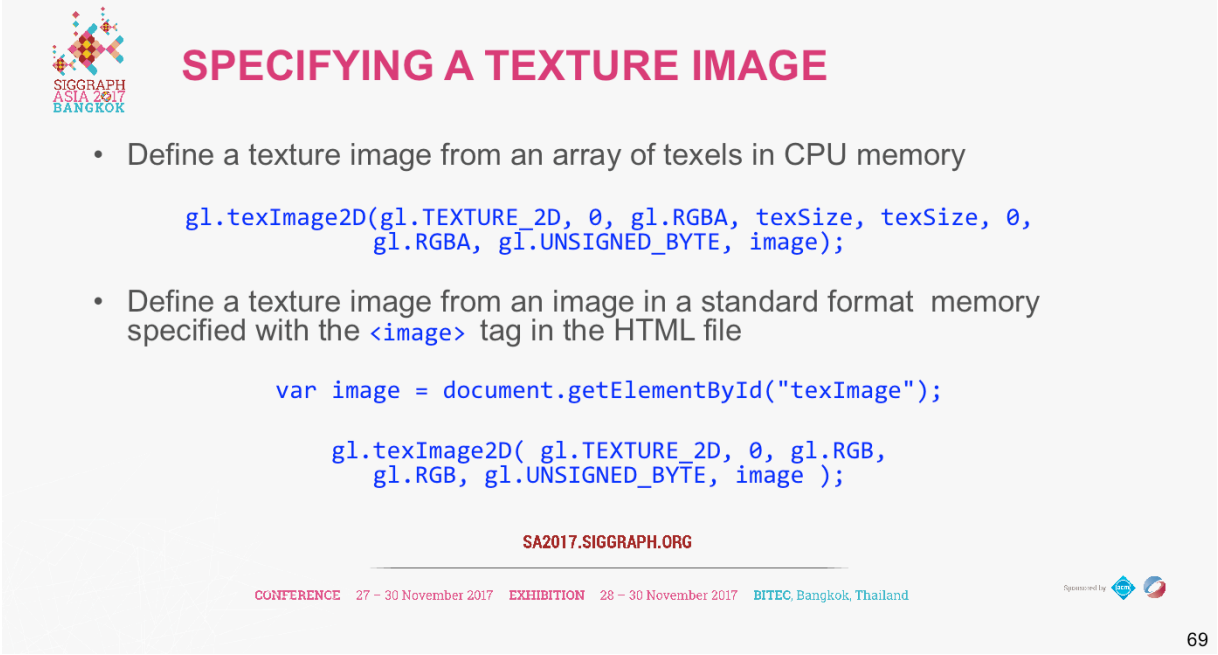
Textures are images that can be thought of as continuous and be one, two, three, or four dimensional. By convention, the coordinates of the image are *s*, *t*, *r* and *q*. Thus for the two dimensional image above, a point in the image is given by its (*s, t*) values with (*0, 0*) in the lower-left corner and (*1, 1*) in the top-right corner.
A texture map for a two-dimensional geometric object in (*x, y, z*) world coordinates maps a point in (*s, t*) space to a corresponding point on the screen.



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.

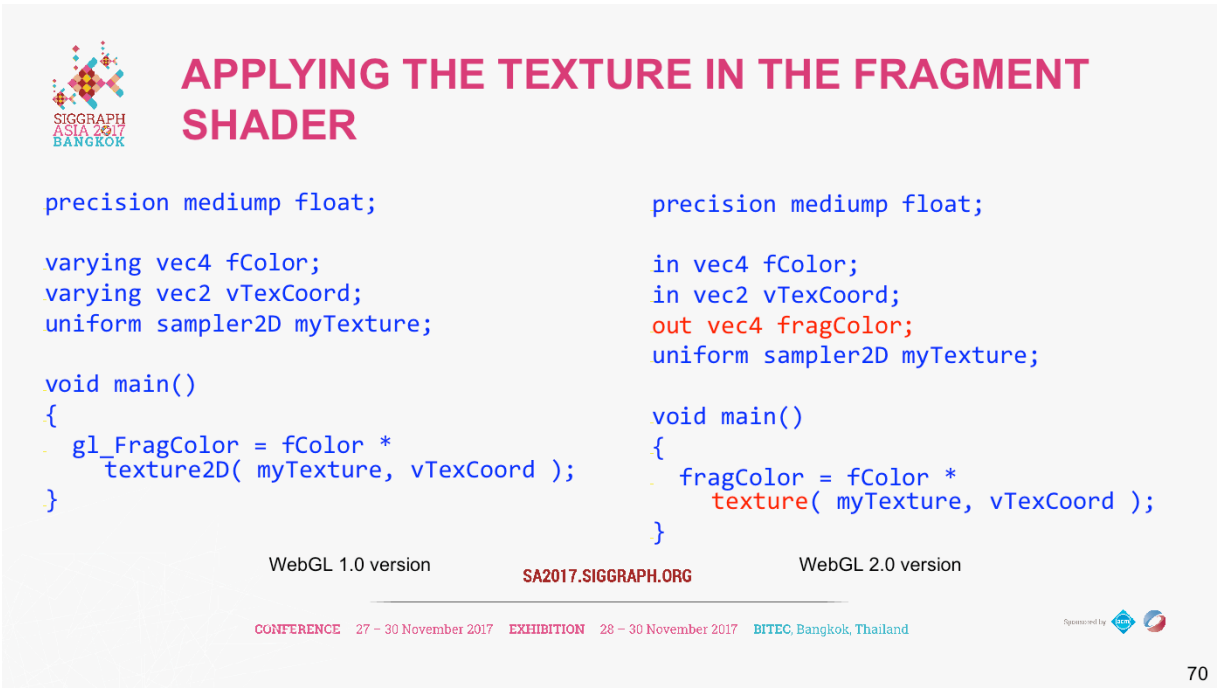


In the simplest approach, we must perform these four steps.
Textures reside in texture memory. When we assign an image to a texture it is copied from processor memory to texture memory where pixels are formatted differently.
Texture coordinates are actually part of the state as are other vertex attributes such as color and normals. As with colors, WebGL interpolates texture inside geometric objects.
Because textures are 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.



Specifying the texels for a texture is done using the `gl.texImage_2D()` 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 WebGL 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.



Just like vertex attributes were associated with data in the application, so too with textures. 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 gl.texImage2D(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).

Putting It All Together

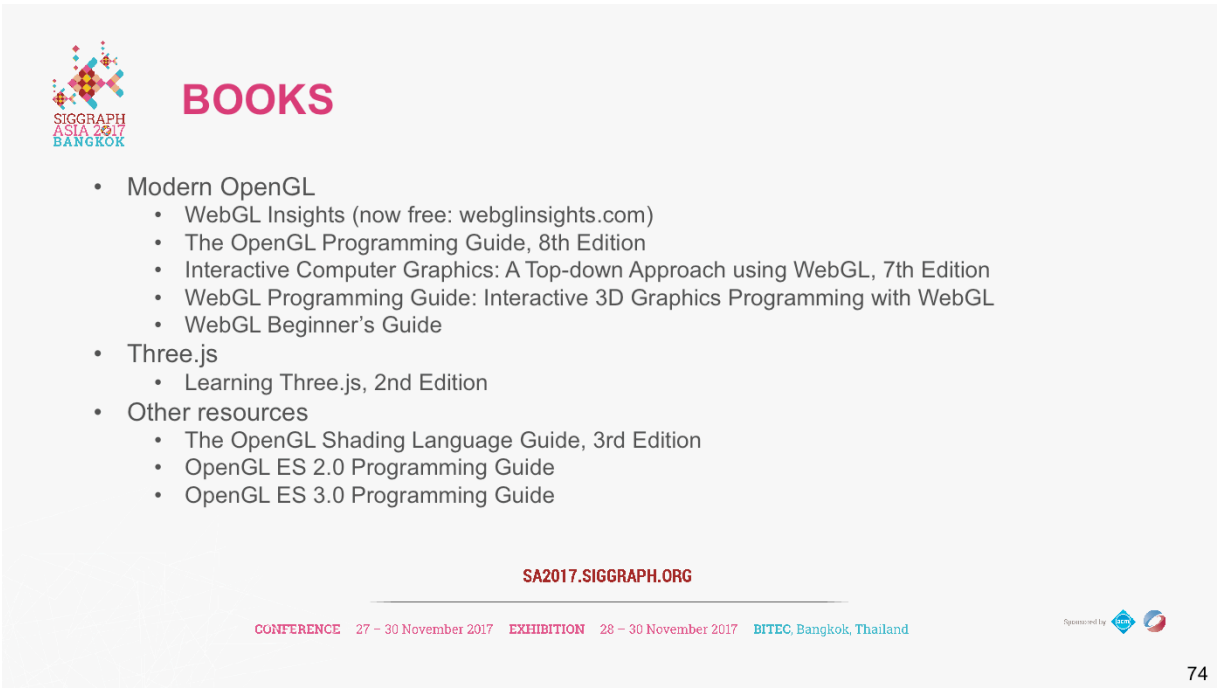
FIVE CHOICES

- desktop OpenGL
- OpenGL ES
- WebGL
- three.js
- Vulkan

- We need all of them

SIGGRAPH ASIA 2017 BANGKOK

Additional Resources



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

ONLINE RESOURCES

- The OpenGL Website: www.opengl.org
- The Khronos Website: www.khronos.org
- Course examples: www.cs.unm.edu/~angel/SIGGRAPH_ASIA_17
- Experiments: www.chromeexperiments.com/webgl
- Links galore: bit.ly/webglhelp
- Three.js's site: threejs.org

THANKS!

You can contact us at:

Ed Angel: angel@cs.unm.edu

Dave Shreiner: shreiner@siggraph.org