AN INTERACTIVE INTRODUCTION TO WEBGL AND THREE.JS

ED ANGEL, UNIVERSITY OF NEW MEXICO
ERIC HAINES, AUTODESK, INC.

AGENDA

- Evolution of Graphics Architectures
- The OpenGL family of APIs
- Working within a browser
- three.js
- WebGL
- Recent Developments

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 note some of the new features supported by WebGL 2.0 at the end of the course. three.js is starting to support WebGL 2.0.

WHAT IS OPENGL?

- OpenGL is a computer graphics rendering *application programming interface (*API)
  - 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 application can be
    - operating system independent
    - window system independent

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.

WHAT IS WEBGL?

- WebGL 1.0: JavaScript implementation of OpenGL ES 2.0
  - runs in all recent browsers (Chrome, Firefox, IE, Safari)
    - entire application is operating system independent
    - entire application is window-system independent
  - application can be located on a remote server
  - rendering is done within browser using local hardware
  - integrates with standard Web packages and apps

OpenGL ES is a smaller version of OpenGL that was designed for embedded systems which did have the hardware capability to run desktop OpenGL. OpenGL ES 2.0 is based on desktop OpenGL2.0 and thus is shader based. Each application must provide both a vertex shader and a fragment shader. Functions from the fixed function pipeline that were in ES 1.0 have been deprecated. OpenGL ES has become the standard API for developing 3D cell phone applications.

WebGL runs within the browser so is independent of the operating and window systems. It is an implementation of ES 2.0 and with a few exceptions the functions are identical. Because WebGL uses the HTML canvas element, it does not require system dependent packages for opening windows and interaction.

WebGL 2.0 is a JS implementation of ES 3.0

WHAT IS THREE.JS

- JavaScript 3D engine
- Object oriented scene graph
  - scene = camera + objects + lights + materials
- Renders with WebGL
- Makes use of GPU but hides details of rendering and modeling

WHAT WE ASSUME YOU KNOW

- Basic graphics concepts
  - Equivalent to fundamentals course
- Programming in a high level language
  - JavaScript is the language of WebGL and three.js
  - Close enough to Java, C, C++ for this course
- Internet familiarity

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

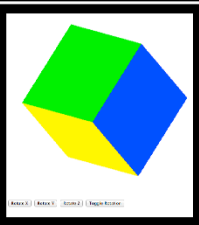
For this course, experience with C/C++, Java, Python should suffice.

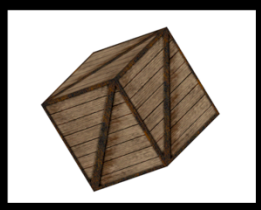
We assume you are comfortable running remote applications in a browser.

OUR APPROACH

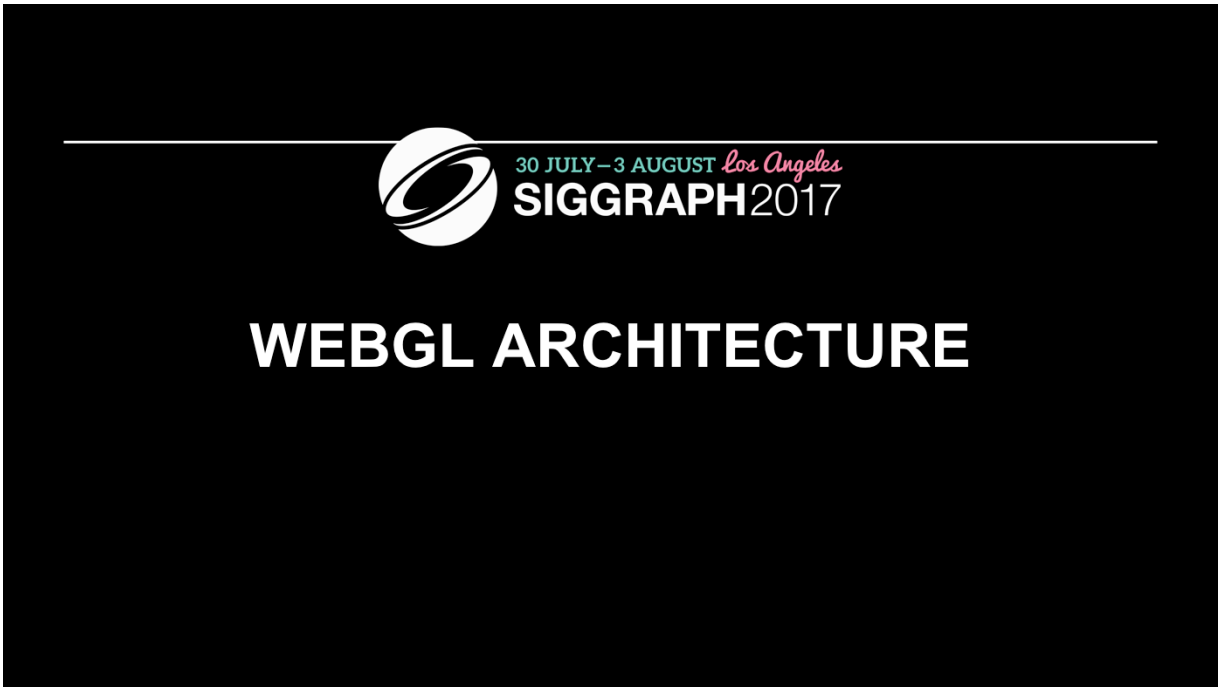
30 JULY – 3 AUGUST Los Angeles SIGGRAPH2017

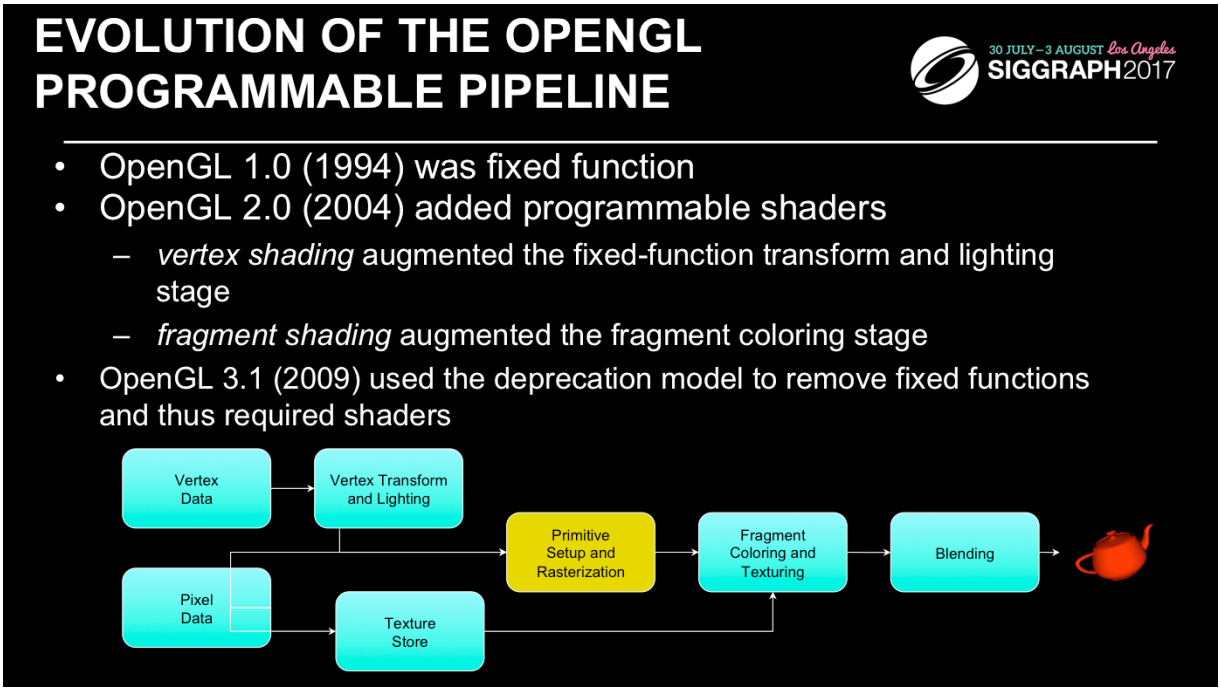
- Interactive intro to three.js
- Follow with similar example using WebGL
- Core example: Cube
  - Geometry
  - Interaction
  - Lighting
  - Texture Mapping





Cube is one of the geometries built into three.js. Because WebGL has a limited set of primitives, with WebGL we must model the cube with triangles. three.js includes libraries with interactive controls. With WebGL we can use libraries available on the Web such as jQuery or code devices through HTML. The standard lighting models are included with three.js. With WebGL we can build our lighting models in a variety of ways including within the shaders. Both three.js and WebGL support texture mapping using either images available in standard formats (gif, jpeg) or images defined in the code.


30 JULY–3 AUGUST Los Angeles
SIGGRAPH2017
WEBGL ARCHITECTURE



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:

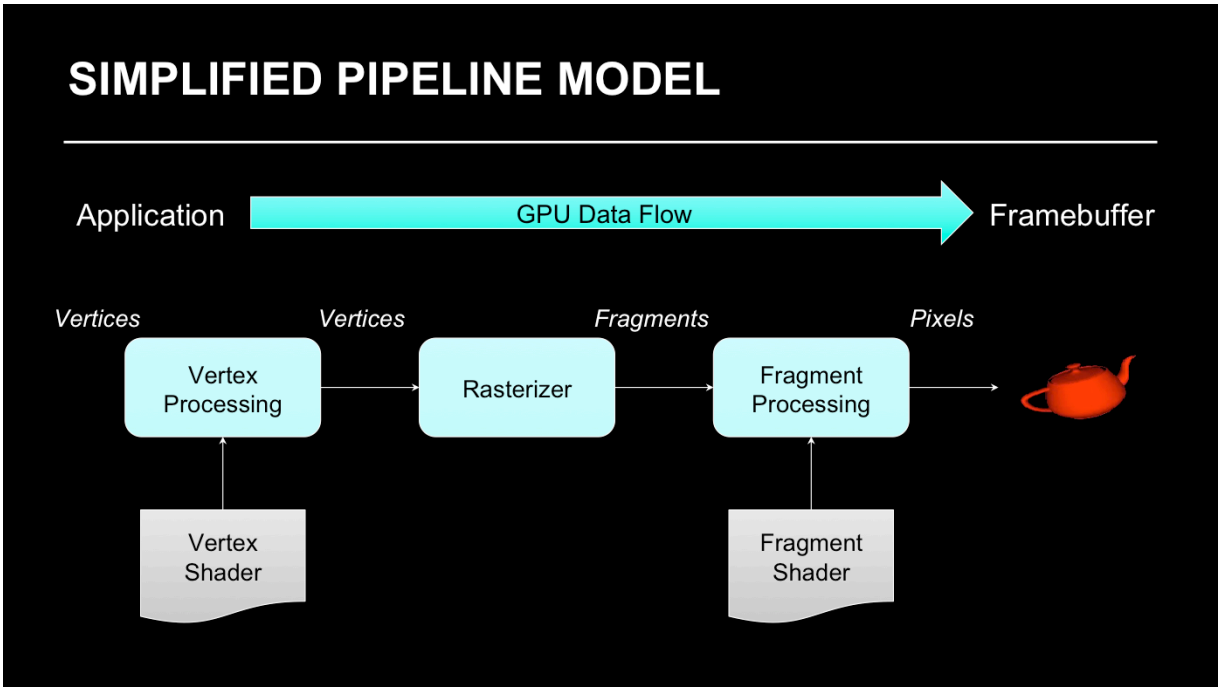
- *vertex shading* enabled the application full control over manipulation of the 3D geometry provided by the application
- *fragment shading* provided the application capabilities for *shading* pixels (the terms classically used for determining a pixel's color).

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 ES AND WEBGL

- OpenGL ES 2.0 and ES 3.0
  - Designed for embedded and hand-held devices such as cell phones
  - ES 2.0 based on OpenGL 2.0 but does not support fixed function pipeline
  - Requires application to provide its own shaders
- WebGL
  - WebGL 1.0: JavaScript implementation of ES 2.0
  - Runs in all recent browsers
  - WebGL 2.0: JavaScript implementation of ES 3.0
  - Starting to be supported in recent releases of browsers

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.



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:

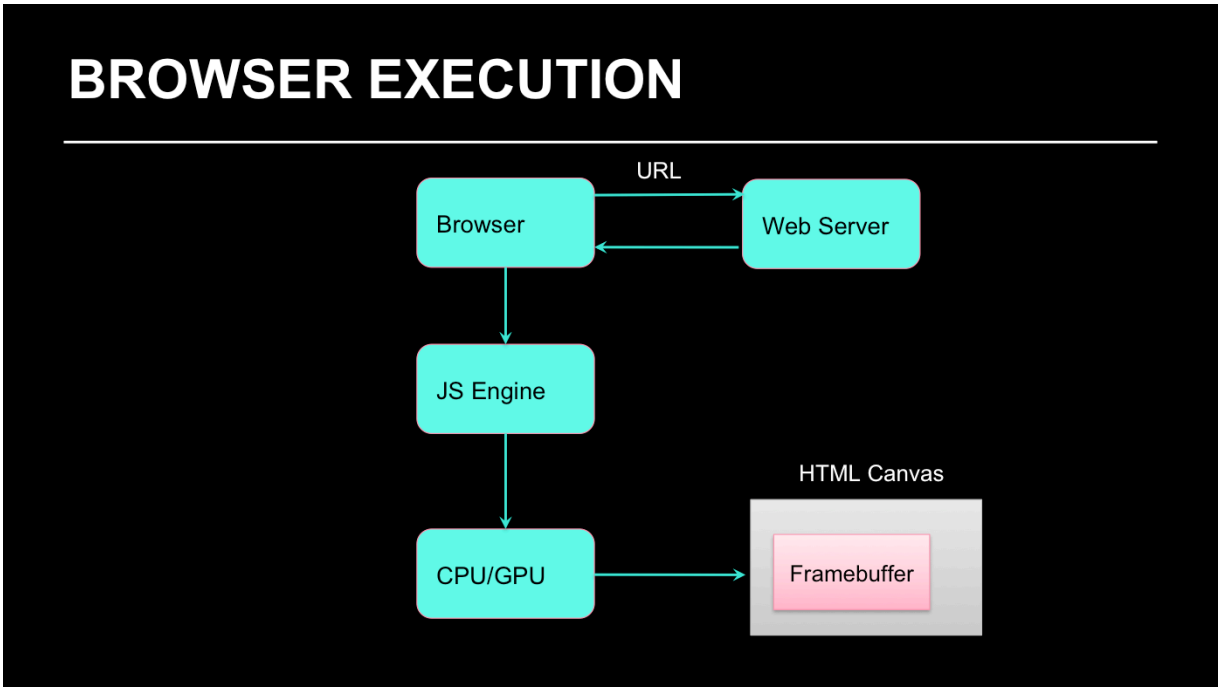
- specify the vertices for your geometry
- load vertex and fragment shaders (and other shaders, if you're using them as well)
- issue your geometry to engage the pipeline for processing

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.

EXECUTION IN A BROWSER

- Fundamentally different from running an OpenGL program locally
- OpenGL execution
  - must compile for each architecture
  - application controls display
  - application runs locally and independently
- WebGL code is independent of the architecture and can be loaded from any server

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.



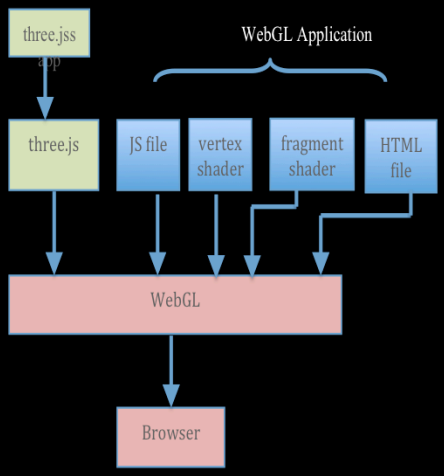
A typical WebGL application consists of a mixture of HTML5, JavaScript and GLSL (shader) code. The application can be located almost anywhere and is accessed through its URL . All browsers can run JavaScript and all modern browsers support HTML. The rendering part of the application is in JavaScript 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.

ENTER THREE.JS

- WebGL requires every application to have at least a JS/HTML file and two shaders written in GLSL
- Modern OpenGL does not provide functions for modeling, shading, transformations or interaction
  - all are responsibility of the application
- three.js is a scene graph API that provides all these features
  - Using WebGL rendering, we get advantages of WebGL without the complexity

OpenGL is not object oriented. three.js is a scene graph which supports a variety of objects including geometric objects (boxes, spheres, planes), cameras, materials, lights and controls and methods for creating and manipulating them. three.js also supports high level functionality such as shadows and kinematics. three.js supports rendering by WebGL, SVG and the HTML Canvas although most applications use WebGL.

WEBGL VS THREE.JS APPLICATIONS



INTERACTIVE INTRODUCTION TO THREE.JS

Go to this site: http://bit.ly/basics3js


30 JULY–3 AUGUST Los Angeles
SIGGRAPH2017
WEBGL APPLICATION DEVELOPMENT

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 three.js

WEBGL IN A NUTSHELL

- All WebGL programs must do the following:
  - Set up canvas to render onto
  - Generate data in application
  - Create shader programs
  - Create buffer objects and load data into them
  - "Connect" data locations with shader variables
  - Render

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

APPLICATION FRAMEWORK

- WebGL applications need a place to render into
  - HTML5 Canvas element
- We can put all code into a single HTML file
- We prefer to put setup in an HTML file and the application in a separate JavaScript file
  - HTML file includes shaders
  - HTML file reads in utilities and application

HTML (hypertext markup language) is the standard for describing Web pages. A page consists of a several elements which are described by tags, HTML5 introduced the canvas element which provides a window that WebGL can render into. Note other applications can also render into the canvas or on the same page.

Generally, we use HTML to set up the canvas, bring in the necessary files and set up other page elements such as buttons and sliders. We can embed our JavaScript WebGL code in the same file or have the HTML file load the JavaScript from a file or URL. Likewise with the shaders.

WEBGL CUBE 1

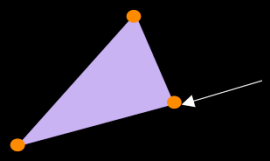
MODELING GEOMETRY

REPRESENTING GEOMETRIC OBJECTS

Geometric objects are represented using vertices

A vertex is a collection of generic attributes

- positional coordinates
- colors
- texture coordinates
- any other data associated with that point in space



Position stored in 4 dimensional homogeneous coordinates

Vertex data must be stored in vertex buffer objects (VBOs)

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.

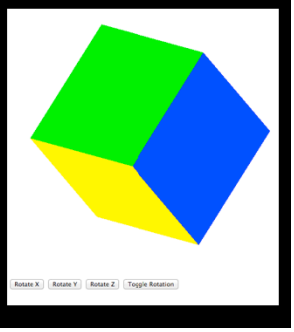
WebGL Geometric Primitives

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.

CUBE PROGRAM

- Render a cube with a different color for each face
- Our example demonstrates:
  - simple object modeling
    - building up 3D objects from geometric primitives
    - building geometric primitives from vertices
  - initializing vertex data
  - organizing data for rendering
  - interactivity
  - animation



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.

INITIALIZING THE CUBE'S DATA

- We'll build each cube face from individual triangles
- Need to determine how much storage is required
  - (6 faces)(2 triangles/face)(3 vertices/triangle)

```
var numVertices = 36;
```

- To simplify communicating with GLSL, we'll use a package `MV.js` that contains a `vec3` object similar to GLSL's `vec3` type

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.

INITIALIZING THE CUBE'S DATA (CONT'D)

- Before we can initialize our VBO, we need to stage the data
- Our cube has two attributes per vertex
  - position
  - color
- We create two arrays to hold the VBO data

```
var points = [ ];
var colors = [ ];
```

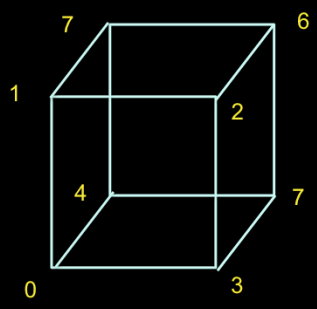
To provide data for WebGL to use, we need to stage it so that we can load it into the VBOs that our application will use. In your applications, you might load these data from a file, or generate them on the fly. For each vertex, we want to use two bits of data – *vertex attributes* in OpenGL speak – 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.

CUBE DATA

- Vertices of a unit cube centered at origin
  - sides aligned with axes

```
var vertices = [
    vec4( -0.5, -0.5,  0.5, 1.0 ),
    vec4( -0.5,  0.5,  0.5, 1.0 ),
    vec4(  0.5,  0.5,  0.5, 1.0 ),
    vec4(  0.5, -0.5,  0.5, 1.0 ),
    vec4( -0.5, -0.5, -0.5, 1.0 ),
    vec4( -0.5,  0.5, -0.5, 1.0 ),
    vec4(  0.5,  0.5, -0.5, 1.0 ),
    vec4(  0.5, -0.5, -0.5, 1.0 )
];
```



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.

CUBE DATA (CONT'D)

- We'll also set up an array of RGBA colors
- We can use vec3 or vec4 or just a JS array

```
var vertexColors = [
        [ 0.0, 0.0, 0.0, 1.0 ],  // black
        [ 1.0, 0.0, 0.0, 1.0 ],  // red
        [ 1.0, 1.0, 0.0, 1.0 ],  // yellow
        [ 0.0, 1.0, 0.0, 1.0 ],  // green
        [ 0.0, 0.0, 1.0, 1.0 ],  // blue
        [ 1.0, 0.0, 1.0, 1.0 ],  // magenta
        [ 0.0, 1.0, 1.0, 1.0 ],  // cyan
        [ 1.0, 1.0, 1.0, 1.0 ]   // white
    ];
```

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.

ARRAYS IN JS

- A JS array is an object with attributes and methods such as length, push() and pop()
  - fundamentally different from C-style array
  - cannot send directly to WebGL functions
  - use flatten() function to extract data from JS array

```
gl.bufferData( gl.ARRAY_BUFFER, flatten(colors),
    gl.STATIC_DRAW );
```

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. See the website.

GENERATING A CUBE FACE FROM VERTICES

To simplify generating the geometry, we use a convenience function `quad()`
create two triangles for each face and assigns colors to the vertices

```
function quad(a, b, c, d) {
var indices = [ a, b, c, a, c, d ];
    for ( var i = 0; i < indices.length; ++i ) {
        points.push( vertices[indices[i]] );

        // for vertex colors use
        //colors.push( vertexColors[indices[i]] );
        // for solid colored faces use
        colors.push(vertexColors[a]);
}
```

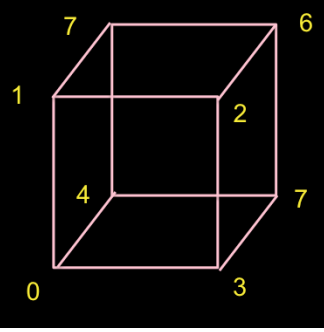
As our cube is constructed from square cube faces, we create a small function, `quad()`, which takes the indices into the original vertex color and position arrays, and copies the data into the VBO staging arrays. If you were to use this method (and we'll see better ways in a moment), you would need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

```
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.

STORING VERTEX ATTRIBUTES

- Vertex data must be stored in a Vertex Buffer Object (VBO)
- To set up a VBO we must
  - create an empty by calling gl.createBuffer();
  - bind a specific VBO for initialization by calling

```
gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
```

  - load data into VBO using (for our points)

```
gl.bufferData( gl.ARRAY_BUFFER, flatten(points),
gl.STATIC_DRAW );
```

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()

VERTEX ARRAY CODE

Associate shader variables with vertex arrays

```
var cBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(colors), gl.STATIC_DRAW );

var vColor = gl.getAttribLocation( program, "vColor" );
gl.vertexAttribPointer( vColor, 4, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vColor );

var vBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW );

var vPosition = gl.getAttribLocation( program, "vPosition" );
gl.vertexAttribPointer( vPosition, 3, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vPosition );
```

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: `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 `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.

DRAWING GEOMETRIC PRIMITIVES

For contiguous groups of vertices, we can use the simple render function

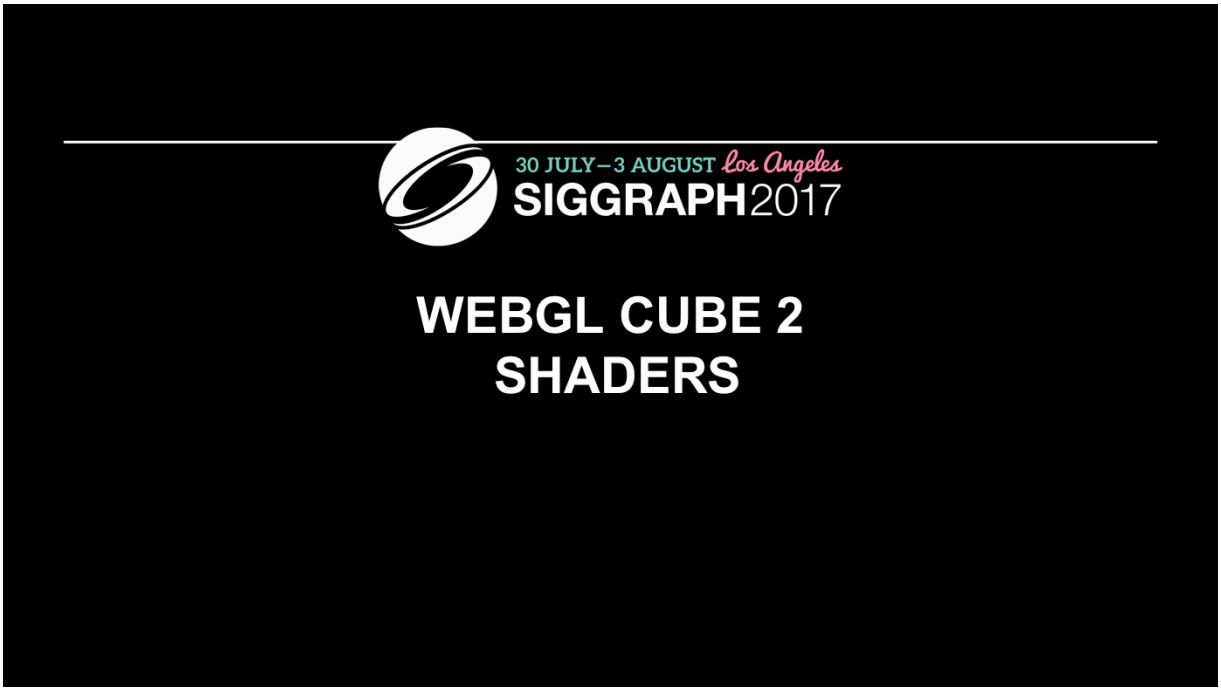
```
function render()
{
    gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    gl.drawArrays( gl.TRIANGLES, 0, numVertices );
    requestAnimFrame( render );
}
```

gl.drawArrays initiates vertex shader
requestAnimationFrame needed for redrawing if anything is changing

Note we must clear both the frame buffer and the depth buffer
Depth buffer used for hidden surface removal
enable HSR by gl.enable(gl.GL_DEPTH) in init()

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.


30 JULY–3 AUGUST Los Angeles
SIGGRAPH2017
WEBGL CUBE 2
SHADERS

VERTEX SHADERS

- A shader that's executed for each vertex
  - Each instantiation can generate one vertex
  - Outputs are passed on to the rasterizer where they are interpolated and available to fragment shaders
  - Position output in clip coordinates
- There are lots of effects we can do in vertex shaders
  - Changing coordinate systems
  - Moving vertices
  - Per vertex lighting
  - Height fields

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.

FRAGMENT SHADERS

- A shader that's executed for each "potential" pixel
  - fragments still need to pass several tests before making it to the framebuffer
- There are many effects we can implement in fragment shaders
  - Per-fragment lighting
  - Texture and bump mapping
  - Environment (reflection) maps

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
- C-like language with some C++ features
- 2-4 dimensional matrix and vector types
- Both vertex and fragment shaders are written in GLSL
- Each shader has a main()

GLSL DATA TYPES

- Scalar types: `float, int, bool`
- Vector types: `vec2, vec3, vec4`
  `ivec2, ivec3, ivec4`
  `bvec2, bvec3, bvec4`
- Matrix types: `mat2, mat3, mat4`
- Texture sampling: `sampler1D, sampler2D, sampler3D, samplerCube`
- C++ Style Constructors

```
vec3 a = vec3(1.0, 2.0, 3.0);
```

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 texture operations. We'll discuss texture samplers in the texture mapping section.

GLSL OPERATORS

- Standard C/C++ arithmetic and logic operators
- Overloaded operators for matrix and vector operations

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

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

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.

QUALIFIERS

- `attribute`
  - `vertex attributes from application`
- `varying (in/out)`
  - copy vertex attributes and other variables from vertex shaders to fragment shaders
  - values are interpolated by rasterizer

```
varying  vec2 texCoord;
varying vec4 color;
```

- `uniform`
  - shader-constant variable from application

```
uniform float time;
uniform vec4 rotation;
```

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

FUNCTIONS

- Built in
  - Arithmetic: `sqrt`, `power`, `abs`
  - Trigonometric: `sin`, `asin`
  - Graphical: `length`, `reflect`
- User defined

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 data-flow engines with data coming in, being processed, and sent on for further processing.

BUILT-IN VARIABLES

`gl_Position`
(required) output position from vertex shader

gl_FragColor
(required) output color from fragment shader

`gl_FragCoord`
input fragment position

`gl_FragDepth`
input depth value in fragment shader

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.

SIMPLE VERTEX SHADER FOR CUBE EXAMPLE


```
attribute  vec4 vPosition;
attribute  vec4 vColor;

varying vec4 fColor;

void main()
{
    fColor = vColor;
    gl_Position = vPosition;
}
```

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.

SIMPLE FRAGMENT SHADER FOR CUBE EXAMPLE


```
precision mediump float;

varying vec4 fColor;
void main()
{
   gl_FragColor = fColor;
}
```

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. 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.

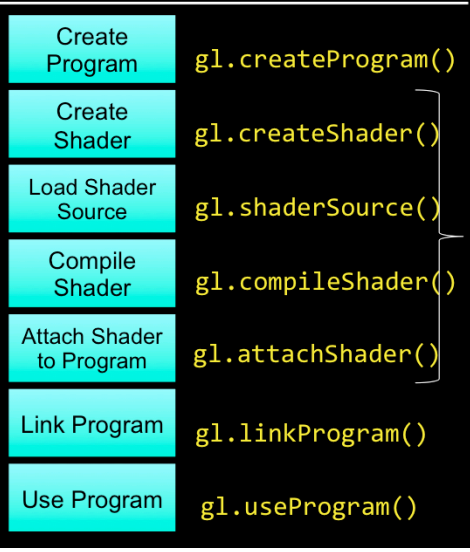
The precision for floats must be specified. All WebGL implementations must support medium precision.

GETTING YOUR SHADERS INTO WEBGL

Shaders need to be compiled and linked to form an executable shader program.

WebGL provides the compiler and linker.

A WebGL program must contain vertex and fragment shaders.



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.

A SIMPLER WAY

- We've created a function for this course to make it easier to load your shaders
  - available at course website

```
initShaders( vShdr, fShdr );
```

- `initShaders` takes two element ids
  - `vShdr` is the element id attribute for the vertex shader
  - `fShdr` – is the element id attribute for the fragment shader
- `initShaders()` fails if shaders don't compile, or program doesn't link

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()`.

ASSOCIATING SHADER VARIABLES AND DATA

- Need to associate a shader variable with an WebGL data source
  - vertex shader attributes → app vertex attributes
  - shader uniforms → app provided uniform values
- WebGL relates shader variables to indices for the app to set

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.

DETERMINING LOCATIONS AFTER LINKING

Assumes you already know the variables' names

```
loc = gl.getAttribLocation(program, "name");

loc = gl.getUniformLocation(program, "name");
```

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

Uniform Variables

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

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.

WebGL Cube 3
Animation and Interaction

ANIMATION

- In a manner similar to three.js
  - We can send new values to the shaders using uniform qualified variables
- Ask application to re-render with requestAnimFrame()
  - Render function will execute next refresh cycle
  - Change render function to call itself

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

```
timeLoc = gl.getUniformLocation(program, "time"); // in init()
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 );
}
```

```
// in vertex shader
uniform float time;
gl_Position = (1.0+0.5*sin(time))*vPosition;
gl_Position.w = 1.0;
```

ADDING BUTTONS

- In HTML file

```
<button id= "xButton">Rotate X</button>
<button id= "yButton">Rotate Y</button>
<button id= "zButton">Rotate Z</button>
<button id = "ButtonT">Toggle Rotation</button>
```

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.

EVENT LISTENERS

In init()

```
document.getElementById( "xButton" ).onclick =
   function () { axis = xAxis; };
document.getElementById( "yButton" ).onclick =
   function () { axis = yAxis; };
document.getElementById( "zButton" ).onclick =
   function () { axis = zAxis;};
document.getElementById("ButtonT").onclick =
   function(){ flag = !flag; };

render();
```

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.

RENDER FUNCTION


```
function render()
{
    gl.clear( gl.COLOR_BUFFER_BIT |gl.DEPTH_BUFFER_BIT);

    if(flag) theta[axis] += 2.0;

    gl.uniform3fv(thetaLoc, theta);

    gl.drawArrays( gl.TRIANGLES, 0, numVertices );

    requestAnimFrame( render );
}
```

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.


30 JULY–3 AUGUST Los Angeles
SIGGRAPH2017
WEBGL CUBE 4
TRANSFORMATIONS

TRANSFORMATIONS

- In WebGL and three.js transformations are defined by 4 x 4 matrices that operate in homogeneous coordinates
  - mat4*vec4 = vec4
  - mat4*mat4 = mat4
  - 3 x3 and 2 x 2 are special cases
- Three main uses
  - viewing
  - changes in coordinate sysems
  - transforming objects (rotation, translation, scaling)

COORDINATE SYSTEMS

- Input to fragment shader is in clip coordinates
  - Everything outside a cube centered at origin with side length 2*w is clipped out
- Applications want to work in their own coordinate system (object or model coordinates)
- How we get from object to clip coordinates is within the application and shader
  - Usual way follows coordinate systems used in deprecated fixed-function OpenGL
- final output in screen 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.

CAMERA ANALOGY AND TRANSFORMATIONS

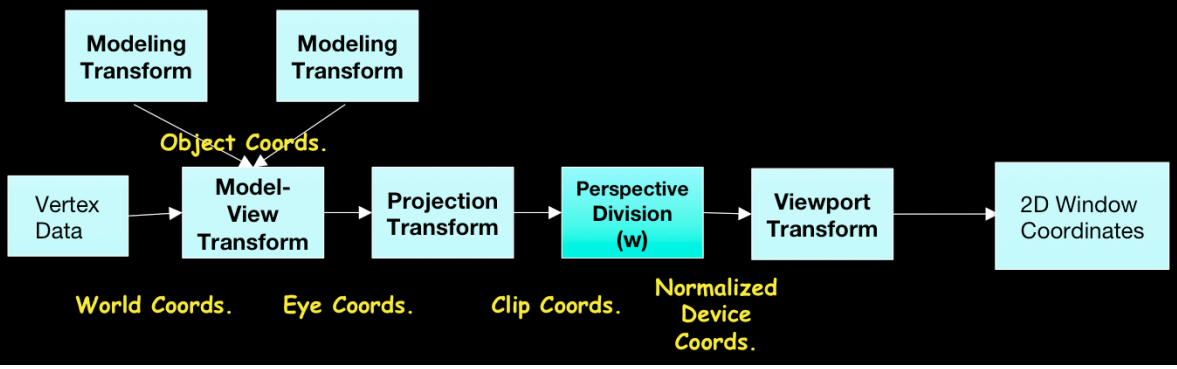
- Projection transformations
  - adjust the lens of the camera
- Viewing transformations
  - tripod–define position and orientation of the viewing volume in the world
- Modeling transformations
  - moving the model
- Viewport transformations
  - enlarge or reduce the physical photograph

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

These transformations were built into the original fixed-function OpenGL, Although the functions that used these coordinate systems have been deprecated (other than the viewport transformation), most applications prefer to build in all these transformations.

TRANSFORMATIONS

- Transformations take us from one "space" or coordinate system (or frame) to another
  - All our transforms are 4×4 matrices



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).

3D TRANSFORMATIONS

- A vertex is transformed by 4×4 matrices
  - all affine operations are matrix multiplications
- All matrices are stored column-major in WebGL
  - this is opposite of what "C" programmers expect
- Matrices are always post-multiplied
  - product of matrix and vector is Mv

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

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

While OpenGL specifies matrices in column-major order, this is often confusing for "C" programmers who are used to row-major ordering for two-dimensional arrays. OpenGL provides routines for loading both column- and row-major matrices. However, for standard OpenGL transformations, there are functions that automatically generate the matrices for you, so you don't generally need to be concerned about this until you start doing more advanced operations.
For operations other than perspective projection, the fourth row is always (0, 0, 0, 1) which leaves the w-coordinate unchanged.

SPECIFYING WHAT YOU CAN SEE

- Set up a viewing frustum to specify how much of the world we can see
- Done in two steps
  - specify the size of the frustum (projection transform)
  - specify its location in space (model-view transform)
- Anything outside of the viewing frustum is clipped
  - primitive is either modified or discarded (if entirely outside frustum)

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 primitives 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.

SPECIFYING WHAT YOU CAN SEE (CONT'D)

- OpenGL projection model uses eye coordinates
  - the "eye" is located at the origin
  - looking down the -z axis
- Projection matrices use a six-plane model:
  - near (image) plane and far (infinite) plane
    - both are distances from the eye (positive values)
  - enclosing planes
    - top & bottom, left & right

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.

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
- Build transformation from simple transformations
  - translation, rotation, scale
- Or use `lookAt( eye, at, up )`
  - up vector determines unique orientation
  - lookAt() is in MV.js and is functionally equivalent to deprecated OpenGL function

`lookAt()` generates a viewing matrix based on several points.
`LookAt()` provides natural 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.

lookAt() is in MV.js as are translation, scaling and rotation functions that we discuss in the next few slides.

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.

VERTEX SHADER FOR ROTATION OF CUBE


```
attribute vec4 vPosition;
attribute vec4 vColor;
varying 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 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.

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 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.

SENDING ANGLES FROM APPLICATION


```
// in init()

var theta = [ 0, 0, 0 ];
var axis = 0;
thetaLoc = gl.getUniformLocation(program, "theta");

// set axis and flag via buttons and event listeners

// in render()

 if(flag) theta[axis] += 2.0;
 gl.uniform3fv(thetaLoc, theta);
```

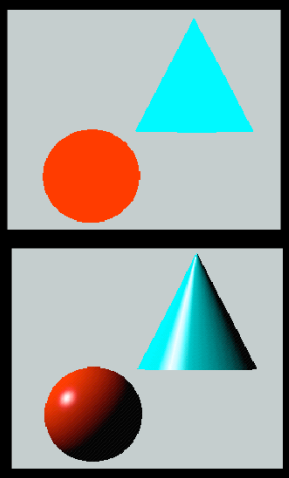
Finally, we merely need to supply the angle values into our shader through our uniform plumbing. In this case, we track each of the axes rotation angle, and store them in a `vec3` that matches the angle declaration in the shader. We also keep track of the uniform's location so we can easily update its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


30 JULY–3 AUGUST Los Angeles
SIGGRAPH2017
WEBGL CUBE 5
LIGHTING

LIGHTING PRINCIPLES

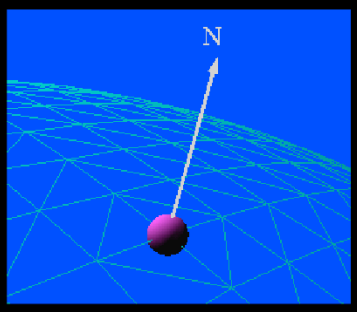
- Lighting simulates how objects reflect light
  - material composition of object
  - light's color and position
  - global lighting parameters
- Usually implemented in
  - vertex shader for faster speed
  - fragment shader for nicer shading
- Modified Phong model was built into fixed function OpenGL
  - Basis of most lighting models (three.js)



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.

SURFACE NORMALS

- Normals give surface orientation and determine (in part) how a surface reflects light
  - Application usually provides normals as a vertex attribute
  - Current normal can be used to compute vertex's color is passed to fragment shader
  - Use unit normals for proper lighting
    - scaling affects a normal's length



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.

MODIFIED PHONG MODEL

- Computes a color for each vertex using
  - Surface normals
  - Diffuse and specular reflections
  - Viewer's position and viewing direction
  - Ambient light
  - Emission
- Vertex colors are interpolated across polygons by the rasterizer
  - *Phong shading* does the same computation per fragment, interpolating the normal across the polygon
    - more accurate 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.

ADDING LIGHTING TO CUBE

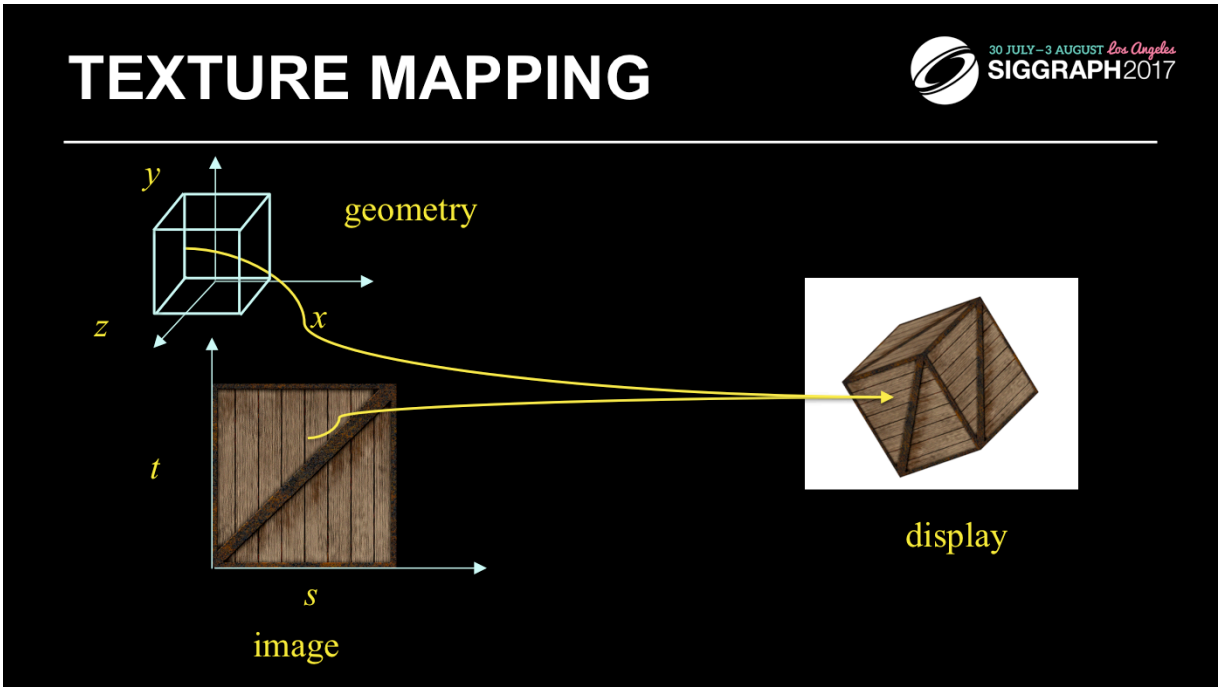
```
// vertex shader

in vec4 vPosition; // or varying in older versions
in vec3 vNormal;
out vec4 color;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform vec4 ambientProduct, diffuseProduct, specularProduct;
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 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.

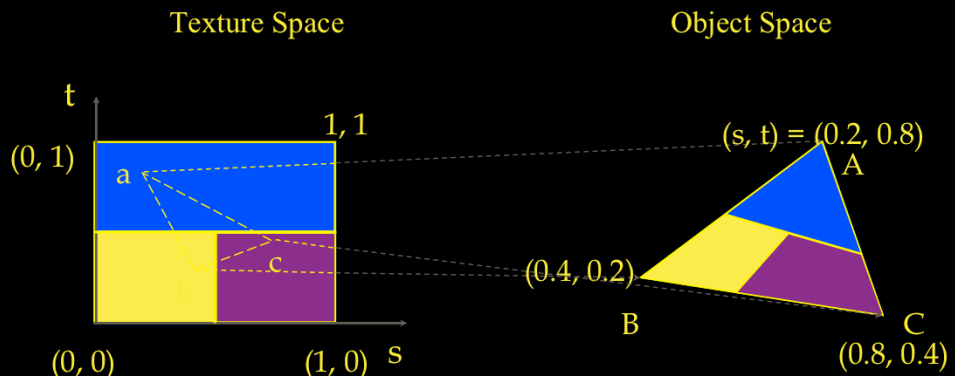

30 JULY–3 AUGUST Los Angeles
SIGGRAPH2017
WEBGL CUBE 6
TEXTURE



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.

MAPPING TEXTURE COORDINATES

- Based on parametric texture coordinates
- coordinates needs to be specified at each vertex



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.

APPLYING TEXTURES

- Basic steps to applying a texture
  1. specify the texture
     - read or generate image
     - assign to texture
     - enable texturing
  2. assign texture coordinates to vertices
  3. specify texture parameters by creating a texture object
     - wrapping, filtering
  4. apply texture in fragment shader with sampler

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 A TEXTURE IMAGE

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

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, texSize,
        texSize, 0,  gl.RGBA, gl.UNSIGNED_BYTE, image);
```

• Define a texture image from an image in a standard format memory specified with the <image> tag in the HTML file

```
var image = document.getElementById("texImage");

gl.texImage2D( gl.TEXTURE_2D, 0, gl.RGB,
       gl.RGB, gl.UNSIGNED_BYTE, image );
```

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.

APPLYING THE TEXTURE IN THE FRAGMENT SHADER


```
precision mediump float;

varying vec4 fColor;
varying  vec2 fTexCoord;
uniform sampler2D texture;

void main()
{
   gl_FragColor = fColor*texture2D( texture, fTexCoord );
}

// Full example on website
```

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).

30 JULY – 3 AUGUST Los Angeles
SIGGRAPH2017

PUTTING IT ALL TOGETHER

FIVE CHOICES

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

- We need all of them

WHAT WE HAVEN'T TALKED ABOUT

- Off-screen rendering
- Compositing
- Cube maps
- Deferred Rendering

- Lots more

POTENTIAL JS “GOTCHAS”

- Almost everything is an object
  - contains methods and attributes
- Scoping is different from most APIs
  - watch out for globals
- Object model may be unfamiliar
  - based on prototypes

Only atomic primitives in JS are numbers (64 bit floats), strings and booleans. Everything else is an object. Objects inherit from a prototype object and thus even the simplest objects have some members and functions that are defined in the prototype. JS uses function scope rather than block scope as in most other languages. There are global variables defined outside of your program, e.g. window.

Typing is dynamic so we can change the type of a variable anyplace in the program.

JavaScript is a large language that has matured over the past few years. However, there are multiple ways to accomplish a task with JS, some good and some bad. See for example Crockford, *JavaScript, the Good Parts*.

WHAT'S MISSING IN WEBGL (FOR NOW)

- Other shader stages
  - geometry shaders
  - tessellation shaders
  - compute shaders
    - WebCL exists
- Vertex Array Objects (added in WebGL 2.0)

ES 2015 AND WEBGL 2.0

- OpenGL ES 3.0 released August 2012
- Adds new features
  - Buffer array objects
  - 3D textures
  - updates ES GLSL
  - Depth textures
  - Transform feedback
- WebGL 2.0 now supported in most browsers

Buffer array objects let us put multiple vertex attributes together. When combined with transform feedback, simulation applications can be executed entirely in the GPU.

ES 2015

- Present specification for JS
- Commonly known as ES 6
- Almost fully supported in recent browsers
- Adds features that allow applications to be written that are closer to Python
- Many variants and transpilers

Many of these variants allow programmers more familiar with Java and Python to write JS code that is more familiar to them and avoids some of the “gotchas” in JS. Some variants also allow the programmer to write more concise code.

Some of the ES6 additions allow for more familiar object types and scoping.

OPENGL AND VULKAN

- Vulkan: successor to OpenGL
- released Spring 2016
- Why Vulkan
  - OpenGL based on 25 year old model
  - Not easy to adapt to recent hardware
  - Driven by high-end game applications
  - Application knows its needs better than the driver

OPENGL AND VULKAN (CONT)

- Vulkan puts most of the control in the application rather than the driver
  - Opposite of OpenGL
- Allows for optimization for architecture
- Not simple to write applications compared with OpenGL

Generally, OpenGL programs are fairly small and the driver large. Consequently, it is straightforward to write an OpenGL application since the complexity is in the driver. But that limits the ability of the application programmer to take advantage of many options that have been set in the driver. Vulkan takes the opposite view and puts a tremendous amount of control in the application and requires a relatively small driver. Thus with Vulkan an application can adjust to the hardware, e.g. an integrated processor vs separate CPU and GPU. For the most applications, we can get the performance we need with WebGL and OpenGL.

VULKAN AND WEBGL (CONT)

- Very different design criteria
- Vulkan is not concerned with working within a browser
- WebGL should continue to following its own development path
- Expect more high-end OpenGL features to be added to WebGL

OpenGL may also continue to develop in parallel with Vulkan since OpenGL has a large user community that does not need to deal with the complexity of Vulkan.

30 JULY – 3 AUGUST *Los Angeles*
SIGGRAPH2017

RESOURCES

BOOKS

- Modern OpenGL
  - WebGL Insights (now free: webglinsights.com)
  - The OpenGL Programming Guide, 8th Edition
  - Interactive Computer Graphics: A Top-down Approach using WebGL, 7th Edition
  - WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL
  - WebGL Beginner's Guide
- Three.js
  - Learning Three.js, 2nd Edition
- Other resources
  - The OpenGL Shading Language Guide, 3rd Edition
  - OpenGL ES 2.0 Programming Guide
  - OpenGL ES 3.0 Programming Guide

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

- This course's notes: http://bit.ly/webgls17
- The OpenGL Website: www.opengl.org
- The Khronos Website: www.khronos.org
- Ed's course examples: www.cs.unm.edu/~angel/WebGL/7E
- Experiments: www.chromeexperiments.com/webgl
- Links galore: bit.ly/webglhelp
- Three.js's site: threejs.org
- Eric's Udacity course: bit.ly/intro3D


30 JULY–3 AUGUST Los Angeles
SIGGRAPH2017
Q & A

THANKS!

- Feel free to drop us any questions:

  angel@cs.unm.edu
  erich@acm.org

- Course notes and programs available at

  www.cs.unm.edu/~angel
  http://bit.ly/basics3js

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