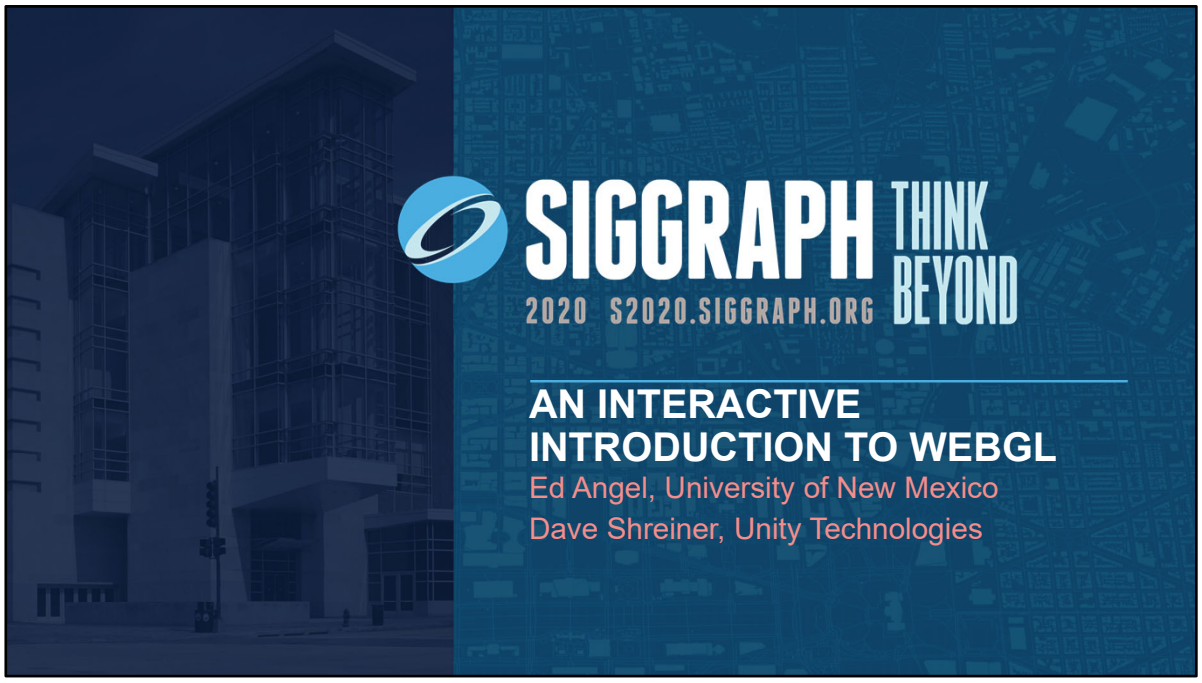

SIGGRAPH
THINK
BEYOND
2020 S2020.SIGGRAPH.ORG
AN INTERACTIVE
INTRODUCTION TO WEBGL
Ed Angel, University of New Mexico
Dave Shreiner, Unity Technologies

COURSE MATERIALS

- This presentation is part of the virtual SIGGRAPH 2020
  - Video is three hours long
  - There will be an interactive Q&A session during the conference
- Please visit the course website at
  - InteractiveComputerGraphics.com/SIGGRAPH/2020
    - Includes latest examples, notes, and source code
- You can run our examples in any recent Web browser
  - including many mobile browsers (on iOS and Android)

Join us live!

Tuesday, August 25th 2020
12:00 – 12:30 PM PST

WHY THIS COURSE?

- Explosion of interest in 3D graphics through a browser
  - Write once – deploy anywhere (mostly ☺)
  - Application runs locally
  - Performance is comparable to native applications
- During our time, we'll try to these questions
  - Which graphics API should I use?
  - How do I get started with WebGL?
  - What can I do in a WebGL application, and what else might be possible?

The emphasis of this course focuses on the capabilities of the WebGL application programming interface (i.e., programming library, often called an *API*).

A WORD TO EDUCATORS

- WebGL is a huge asset for teaching Computer Graphics
- Libraries and Application code are the same on every system
  - Checking student projects is easy
  - Students love to be able to show their work to friends and family on hand-held devices
- OpenGL programs must be recompiled for each architecture
  - Libraries must be obtained for each system (Windows, linux, Mac OSX)
  - Libraries often change with versions of the OS
  - Beginning of semester can be a nightmare for the instructor
- To students, JavaScript is just another language

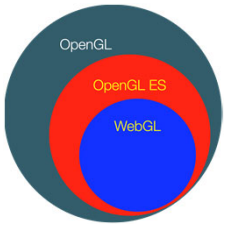
WHAT WE ASSUME YOU KNOW

- Basic graphics concepts
  - Equivalent to what is presented in SIGGRAPH's graphics fundamentals course
- Programming in a high-level language
  - WebGL is effectively a JavaScript library
  - Knowing some Java, C, or C++ is sufficient for this course
- Internet familiarity
  - Basic HTML
  - Webpage components
    - head, body, scripts

We assume that you're familiar with computer-graphics concepts: vertices, geometry, rendering, simple illumination and lighting, and texture mapping. And since we're developing applications for the Web, we assume you know the fundamentals of web browsers and servers, and familiarity with structured programming languages like C or C++, Java, or Python.

WHAT ARE GRAPHICS APIS, WEBGL, AND WHY LEARN THEM?

- A *(Computer) Graphics API* is a programming library for drawing graphics and other operations
  - API is short to *Application Programming Interface*
- WebGL is part of the OpenGL family of APIs
- WebGL implements OpenGL ES in JavaScript
  - runs in all recent browsers (Chrome, Firefox, Edge, 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



API Name (Latest Version)	Device Environment
OpenGL (4.5)	Personal Computers
OpenGL ES (3.2)	Embedded devices
WebGL (2.0)	Web Browsers

WebGL is a JavaScript (*JS*) implementation of ES 2.0, and runs within the browser, so it is independent of the operating and window systems. Additionally, the signatures of the functions (i.e., the list of parameters) are identical in all but a few cases, so learning WebGL gives the added benefit of knowing a lot about programming OpenGL and OpenGL ES as well. Further, because WebGL uses the HTML canvas element, it does not require system-dependent libraries for opening windows and interaction.

WebGL 2.0 is a JavaScript implementation of ES 3.0.

AGENDA

- WebGL Architecture
  - Evolution of Graphics Architectures
  - The OpenGL family of APIs
  - Working within a browser
- Introduction to WebGL
- Analysis of a Complete Example
  - Modeling geometry
  - Shader overview
  - Transformations
  - Lighting
  - Texture Mapping
- Real-world Examples and Advanced Techniques
- Things to explore
  - Advances in graphics APIs and Web technologies
  - Topics for self study
- Resources & Q&A

The development of APIs for 3D computer graphics is exemplified by the development of the OpenGL family of APIs. The original version contained many functions (commonly called the *fixed-function pipeline*) for manipulating and rendering three-dimensional geometry, which while simple to use, limited flexibility. As GPUs became programmable, APIs supported *programmable shaders*. Consequently, the fixed-function pipeline was replaced with a shader-based pipeline where applications are expected to provide the shaders for rendering. Under both these paradigms, the graphics libraries were accessed via code compiled for and executing on the CPU (in programming languages like C/C++, Java, and Python, to name a few). As such, applications needed to be recompiled for each CPU architecture

With the advent of the World Wide Web, the focus of interactive-graphics applications switched to HTML, primarily through the HTML5 Canvas element. Such applications could be distributed from a remote server to a web browser running on a machine and make use of local hardware, especially the GPU. WebGL is a JavaScript implementation of OpenGL ES that can be used with HTML5 and thus any recent browser. Because WebGL uses the local hardware, its performance is close to that of desktop OpenGL.

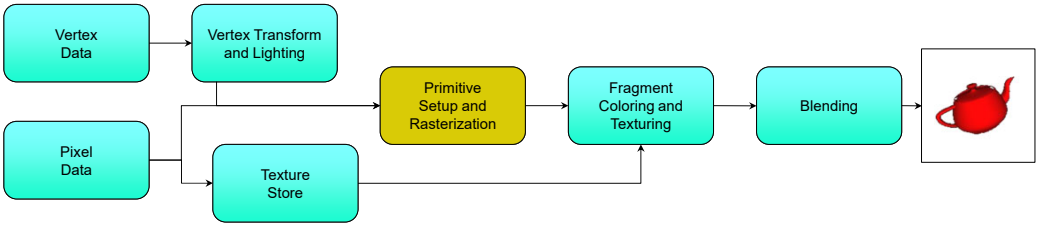
Both desktop OpenGL and WebGL require the application to provide shaders, and to do so requires knowledge of their shading language — GLSL — and how to create and manipulate various buffer and many other tasks which may not be of interest to application programmer. Scene graphs avoid many of these issue by providing a higher-level API which calls into a system's OpenGL/WebGL implementation. For web applications, three.js is the dominant API for three-dimensional, interactive graphics applications. A basic application needs only to describe a scene using objects, cameras, and attributes (e.g., colors, textures, materials) that are part of the API.

In this section, we'll describe the architecture of WebGL, describing its pipeline, and highlighting the important parts for WebGL applications.

s1

EVOLUTION OF THE OPENGL PIPELINE

- OpenGL 1.0 (1992) had fixed, limited functionality
- OpenGL 2.0 (2004) added programmable shaders
  - vertex shading augmented the fixed-function transform and lighting stage
  - fragment shading augmented the fragment coloring stage
- OpenGL 3.1 (2008) deprecated the fixed-function interface and required shaders



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) through function calls. 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.

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:

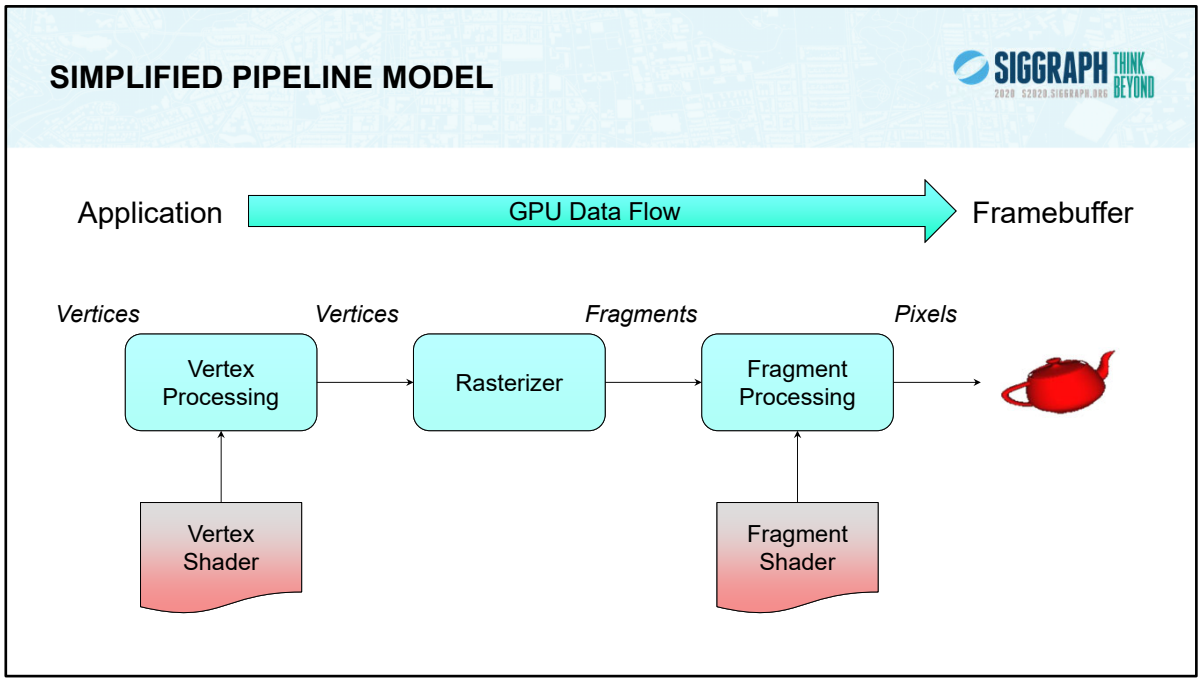
vertex shading enabled the application full control over manipulation of the 3D geometry provided by the application

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*.

Slide 9

s1 shreiner, 7/1/2020



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.

OPENGL ES AND WEBGL

- OpenGL ES 2.0 and ES 3.0
  - Designed for embedded and hand-held devices such as cell phones
  - OpenGL ES 2.0 is based on OpenGL 2.0 but *requires shaders*
    - no fixed-function features are available
- WebGL
  - WebGL 1.0: JavaScript implementation of ES 2.0
    - Supported 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.

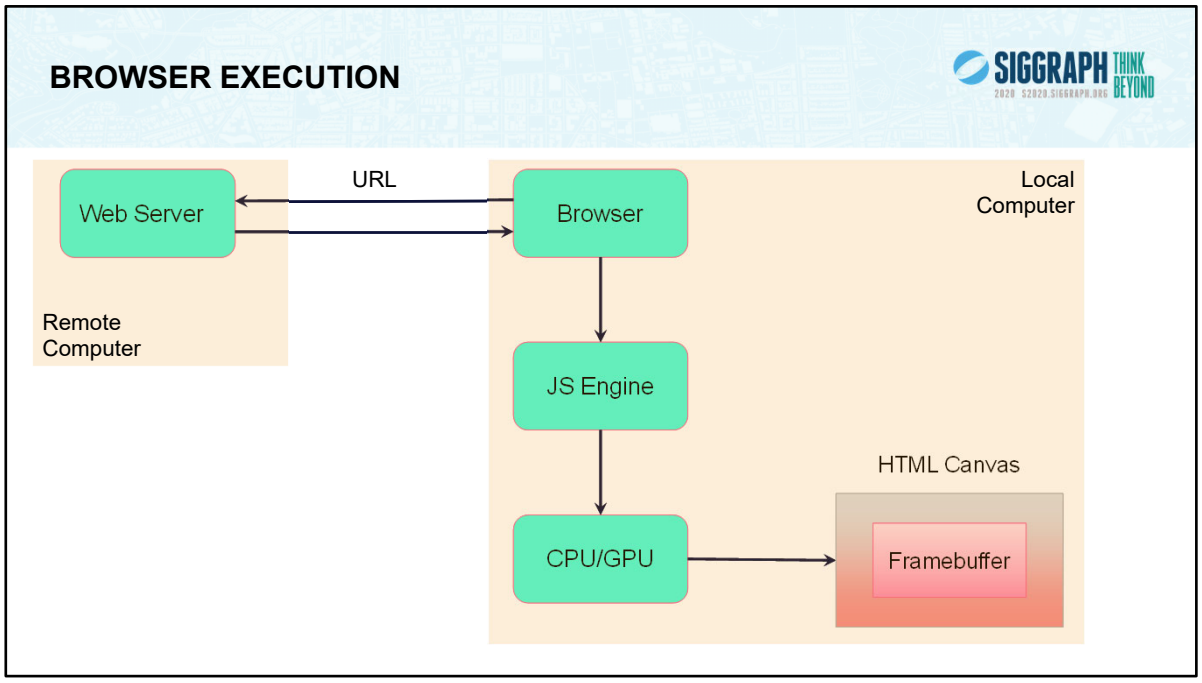
On Windows systems, Chrome and Firefox use an intermediate layer called ANGLE, which takes OpenGL calls and turns them into DirectX calls. This is done because the DirectX drivers are generally more efficient for Windows, since they've undergone more development. Command-line options can disable the use of ANGLE.

EXECUTION IN A BROWSER

- Fundamentally different from running an OpenGL program locally
- OpenGL execution
  - Compiled executable for each architecture
  - application controls display and manages window creation
  - application runs on local machine using its resources
    - CPU
    - Memory
    - GPU
- WebGL code
  - Independent of machine architecture
  - Runs in a web browser on local computer
  - Can be served from a web browser or use files on local machine

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 processing are not part of OpenGL and can be vary significantly 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.


SIGGRAPH
THINK
BEYOND
2020 S2020.SIGGRAPH.ORG
INTRODUCTION TO WebGL

WHY USE WEBGL DIRECTLY?

- Three.js and other libraries (e.g., babylon.js, OSG.JS) are handy
- However:
  - 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

Clearly, there are advantages to using a toolkit like three.js, so why work directly in WebGL? Most of those libraries increase the download size of the web application, which can impact both application load times, as well as their ability to run on mobile devices. Further, toolkits prescribe the order of operations and facilitates. WebGL allows the application programmer complete control over the operation of the graphics pipeline from the application. Of course, that level of control comes at the cost of needing to know considerably more about the operation of computer graphics, and how to implement those algorithms.

That said, knowing how WebGL operates can make you more efficient and informed when using a higher-level toolkit.

WEBGL IN A NUTSHELL

- All WebGL programs must do the following:
  1. Set up canvas to render onto
  2. Generate data in application
  3. Create shader programs
  4. Create buffer objects and load data into them
  5. “Connect” data locations with shader variables
  6. 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:

First, you’ll load and create WebGL shader programs from shader source programs you create

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.

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.

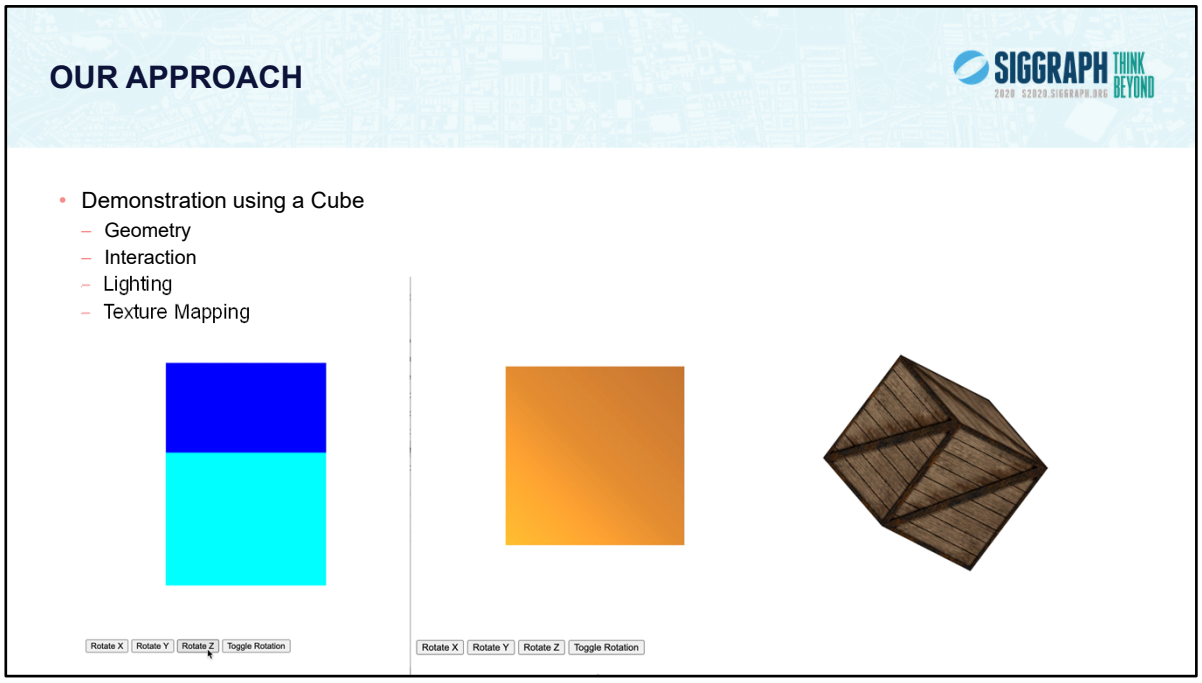
Finally, with your data initialized and shaders set up, you’ll render your objects

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

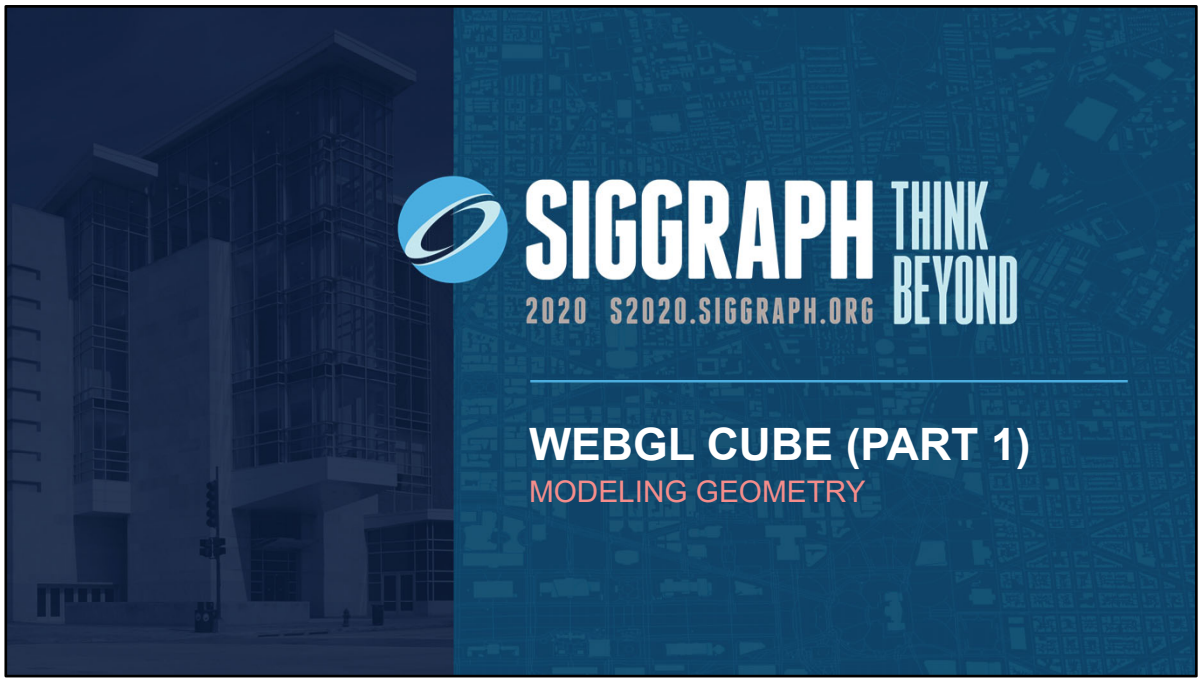
HTML (hypertext markup language) is the standard for describing Web pages. A page consists of a several elements which are described by tags, HTML5 introduced the canvas element which provides a window that WebGL can render into. Note other applications can also render into the canvas or on the same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



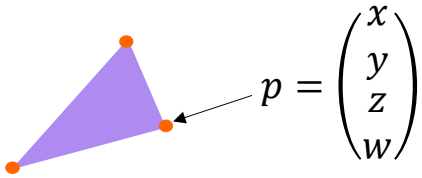
The cube is one of computer graphics' fundamental primitives. It's a built-in object to three.js, but for WebGL, we'd need to specify the cube using the primitives available for WebGL, most notably triangles. To shade our cube in WebGL, we'll need to understand concepts like lighting, textures and texture mapping, and perhaps blending. These concepts are also available in three.js, and require much less work to apply them to a geometric object. Similarly, interacting with the geometric objects in three.js is quite simple. By contrast, WebGL doesn't have any facilities for interaction; the application programmer needs to receive and interpret the user's interaction with the application, and convert those into operations affecting how WebGL manipulates its geometric objects.

At this point, you may be asking yourself "Why would anyone want to code directly in WebGL?". Low-level interfaces like WebGL provide the ultimate flexibility to an application, and used appropriately, may provide superior performance. Additionally, three.js prescribes how and the order that operations are done. If your application needs to do something outside of those capabilities, you may need to modify three.js's operation, which can be done using WebGL.


SIGGRAPH
THINK
BEYOND
2020 S2020.SIGGRAPH.ORG
WEBGL CUBE (PART 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)



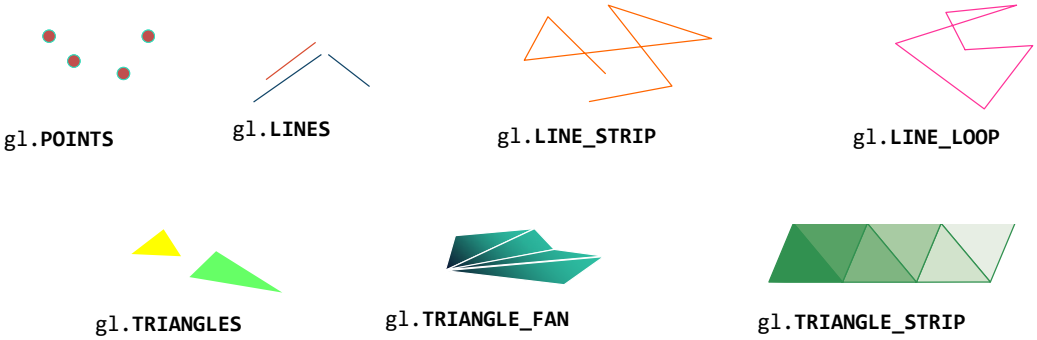
In OpenGL, as in other graphics libraries, objects in the scene are composed of geometric primitives, which themselves are described by vertices. A vertex in modern OpenGL is a collection of data values associated with a location in space. Those data values might include colors, reflection information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

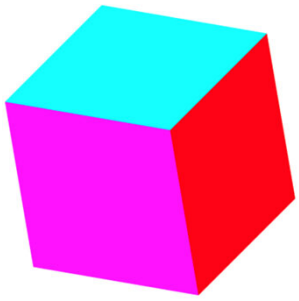
- All primitives are specified by vertices



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
    - build 3D objects from geometric primitives
    - specify geometric primitives with vertices
  - initializing vertex data
  - organizing data for rendering
  - interactivity
  - Animation
- Code online at the course website
  - www.interactivecomputergraphics.com/SIGGRAPH/2020



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 like 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 (empty) arrays to hold the VBO data

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

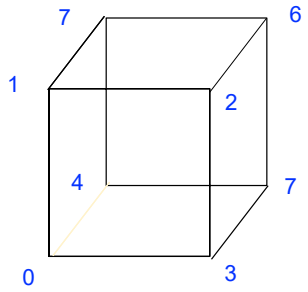
To provide data for WebGL to use, we need to stage it so that we can load it into the VBOs that our application will use. In your applications, you might load these data from a file, or generate them on the fly. For each vertex, we want to use two bits of data – vertex attributes in OpenGL speak – to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

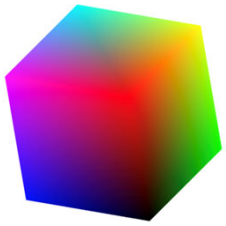
```
function quad(a, b, c, d) {
  var indices = [ a, b, c, a, c, d ];
  for ( var i = 0; i < indices.length; ++i ) {
    points.push( vertices[indices[i]] );

    // 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

INTERPOLATING COLORS

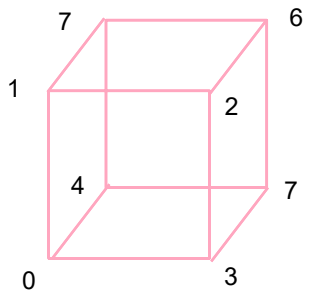
- Vertices are sent through the rasterizer which generates fragments
  - For a points the default for the rasterizer is to produce a single fragment
  - For a line, the rasterizer, produces fragments whose positions interpolate between pairs of vertex positions
  - For a triangle, the rasterizer interpolates the three vertex positions to generate interior fragments
- For vertex attributes, such as colors, the rasterizer interpolates from their vertex values



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:

Generate a buffer's by calling gl.createBuffer().

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.

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.

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 );
    requestAnimationFrame( render );
}
```

- `gl.drawArrays()` initiates vertex shader
- `requestAnimationFrame()` needed for redrawing if anything is changing

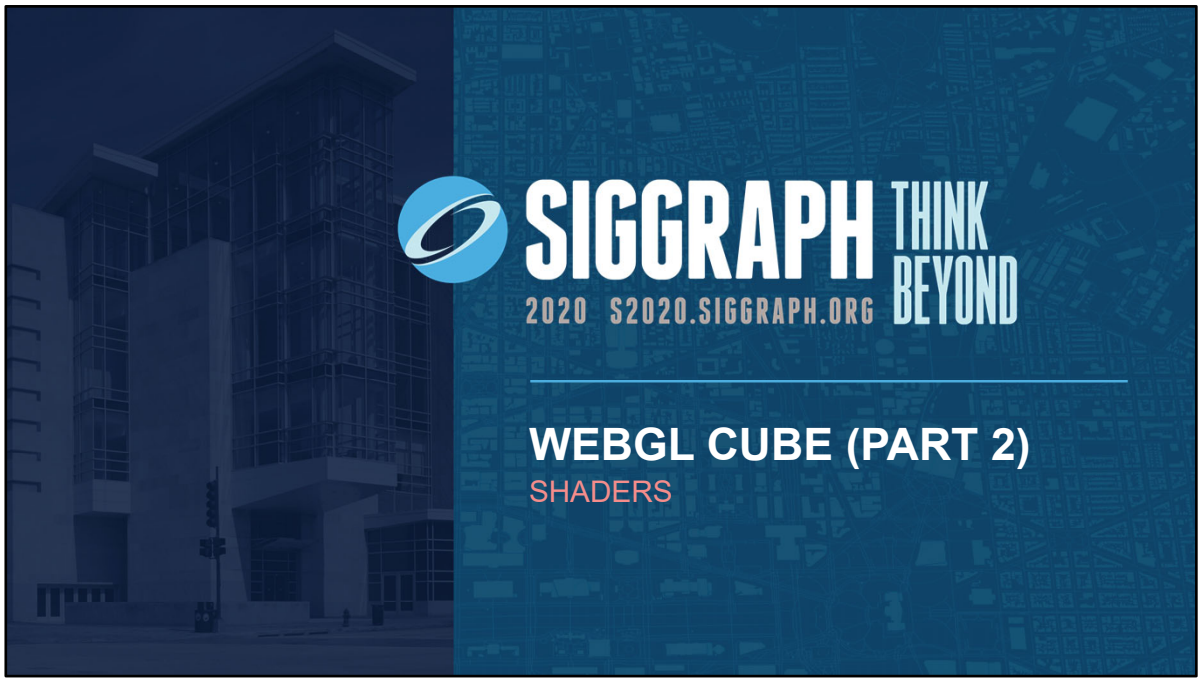
To initiate rendering in your application, you need to issue a drawing routine. The render() function shown above contains the essence of what needs to be done each frame to render with WebGL.

First, we clear where we want to render by calling gl.clear(). In the case shown above, we clear two buffers: the color buffer, where our generated image will appear; and the depth buffer, used for hidden surface removal. In order to remove hidden surfaces, you need to ask WebGL to enable depth testing, using the call gl.enable(gl.DEPTH_TEST), which we would have specified in our init() routine (assuming we wanted it enabled for the entirety of the application).

While there are many routines for rendering in WebGL, we'll discuss the most fundamental ones. The simplest routine is gl.drawArrays(), specifies the type of graphics primitive you want to draw (e.g., here we're rending triangles); the 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 your 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.


SIGGRAPH
THINK
BEYOND
2020 S2020.SIGGRAPH.ORG
WEBGL CUBE (PART 2)
SHADERS

VERTEX SHADERS

- A shader that's executed for each vertex
  - Each execution can generate one vertex
  - Outputs are passed on to the rasterizer where they are interpolated and used to spawn fragment shader executions
  - Vertex's output position is 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- to 4-dimensional matrix and vector types
- Both vertex and fragment shaders are written in GLSL
- Each shader has a `main()` entry point

Generally, GLSL code is compiled by WebGL as opposed to the HML and JS code which is interpreted. After successful compilation the shaders are put into a program object which is linked with the application code. WebGL allows for multiple program objects and thus multiple shaders within an application.

GLSL DATA TYPES

Data Type	WebGL GLSL Type
Scalars	`float`, `int`, `bool`

Vectors	`vec2`, `vec3`, `vec4`
	`ivec2`, `ivec3`, `ivec4`
	`bvec2`, `bvec3`, `bvec4`

Matrices	`mat2`, `mat3`, `mat4`

Texture Samplers	`sampler2D`
	`sampler3D` (WebGL 2.0)
	`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

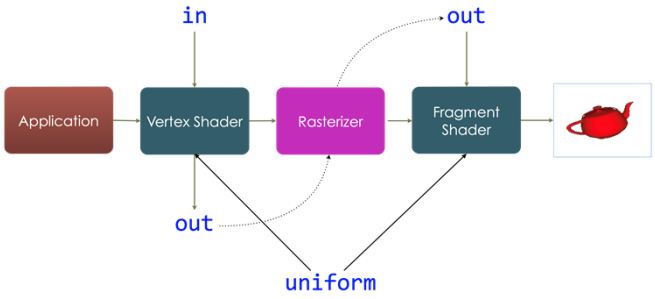
```
mat4 m;
vec4 a, b, c;

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

- `in` (`attribute`)
  - vertex attributes from application
- `in/out` (`varying`)
  - values are sent from vertex shader to rasterizer
    - `out vec2 texCoord;`
    - `out vec4 color;`
  - fragment shader receives interpolated values
    - `in vec2 texCoord;`
    - `in 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`
- Support for user-defined functions

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 in WebGL 1.0
  - *not used* in WebGL 2.0
    - Replaced by a user-defined `out` variable
- `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 several 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 (WEBGL 1.0 AND 2.0)


```
attribute vec4 aPosition;
attribute vec4 aColor;

varying vec4 vColor;

void main()
{
    vColor = aColor;
    gl_Position = aPosition;
}
```

WebGL 1.0 version

```
in vec4 aPosition;
in vec4 aColor;

out vec4 vColor;

void main()
{
    vColor = aColor;
    gl_Position = aPosition;
}
```

WebGL 2.0 version

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 (WEBGL 1.0 AND 2.0)

```
precision mediump float;

varying vec4 vColor;


void main()
{
    gl_FragColor = vColor;
}
```

WebGL 1.0 version

```
precision mediump float;

in vec4 vColor;
out vec4 fColor;

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

WebGL 2.0 version

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.

Create Program	`gl.createProgram()`
Create Shader	`gl.createShader()`
Load Shader Source	`gl.shaderSource()`
Compile Shader	`gl.compileShader()`
Attach Shader to Program	`gl.attachShader()`
Link Program	`gl.linkProgram()`
Use Program	`gl.useProgram()`

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 variables with application data sources
  - we specify shader variables as strings in the shader source
  - WebGL internally maps those strings to index handles (called *locations*)
- Need to retrieve a variable's location, then associate that location with a data source

Variable Qualifier	Location Function	Data Association Function	Application Data Storage
`in` `attribute`	`gl.getAttribLocation()`	`gl.vertexAttribPointer()`	WebGL buffer (`gl.createBuffer()`)
`uniform`	`gl.getUniformLocation()`	`gi.uniform*fv()` `gl.uniformMatrix*fv()`	JavaScript Array Typed Array (i.e., use `flatten()`)

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.

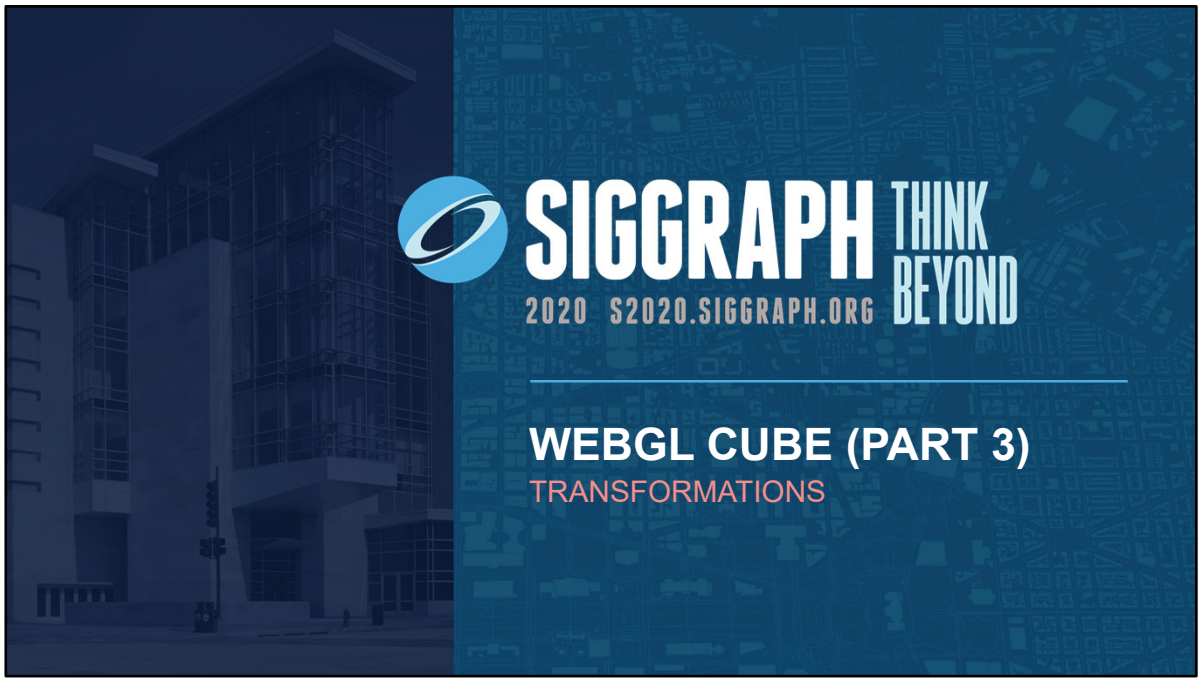
Each of the shader's input data variables (ins and uniforms) needs to be connected to a data source in the application. We've already seen glGetAttribLocation() for retrieving information for connecting vertex data in a VBO to shader variable. You will also use the same process for uniform variables,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

RENDER FUNCTION

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

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

    gl.uniform3fv( thetaLoc, theta );

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

    requestAnimationFrame( 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.


SIGGRAPH
THINK
BEYOND
2020 S2020.SIGGRAPH.ORG
WEBGL CUBE (PART 3)
TRANSFORMATIONS

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

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

By using 4×4 matrices, OpenGL can represent all affine and perspective 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.

TRANSFORMATIONS

- In WebGL and three.js transformations are defined by 4×4 matrices that operate in homogeneous coordinates

```
mat4 * vec4 = vec4
mat4 * mat4 = mat4
```

  - other matrix dimensions (e.g., 3×3, or 4×2) are also supported with types of the form `mat3` or `mat4x2`

- Transformations have three main uses:
  - viewing and projection
  - changes in coordinate systems
  - transforming objects (rotation, translation, scaling)

Matrix operations are supported directly in GLSL where matrices and vectors are atomic types. In the application code, we either carry out the operations in our code or use a library such as MV.js or glMatrix.

CAMERA ANALOGY AND TRANSFORMATIONS

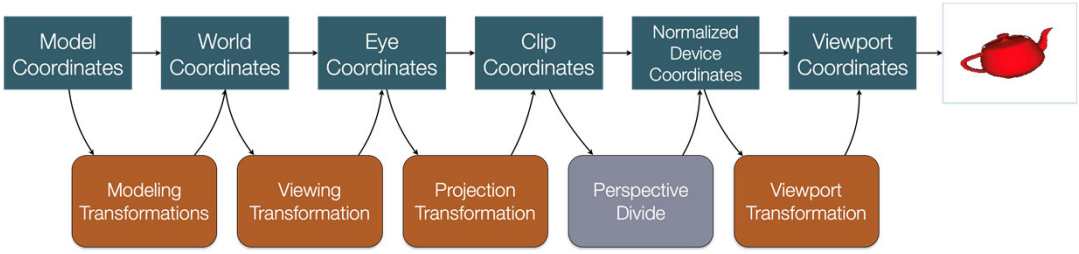
- Projection transformations
  - adjust the lens of the camera
- Viewing transformations
  - tripod–define position and orientation of the viewing volume in the world
- Modeling transformations
  - moving the model
- Viewport transformations
  - enlarge or reduce the physical photograph

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.

TRANSFORMATION PIPELINE

- 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).

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

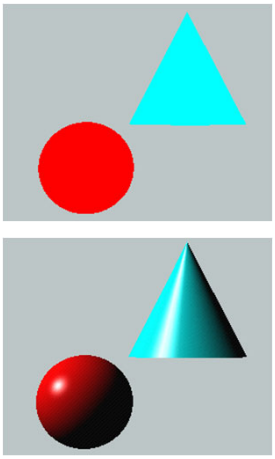
Modeling transformations, in combination with viewing transforms, which dictate where the viewing frustum is in world coordinates, are the first transformation that a vertex goes through. Next, the projection transform is applied which maps the vertex into another space called clip coordinates, which is where clipping occurs. After clipping, we divide by the w value of the vertex, which is modified by projection. This division operation is what allows the farther-objects-being-smaller activity. The transformed, clipped coordinates are then mapped into the

window.


SIGGRAPH
THINK
BEYOND
2020 S2020.SIGGRAPH.ORG
WEBGL CUBE (PART 4)
LIGHTING

LIGHTING IN INTERACTIVE APPLICATIONS

- Lighting (illumination) simulates how objects reflect light
  - Adds realism to the scene
- Lighting is a very complicated topic
  - Global illumination in computer graphics is a very active problem
- We are limited in what we can do rasterization-based applications
  - Simple lighting effects are possible, but lack many elements
    - shadows
    - reflections
    - inter-object interactions (e.g., refraction, blending of translucent objects)
  - More accurate lighting requires advanced WebGL techniques
    - framebuffer objects and multiple passes

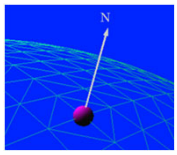


Lighting is an important technique in computer graphics. Without lighting, objects tend to look like they are made from plastic.
The models used in most WebGL applications divide lighting into three parts: material properties, light properties and global lighting parameters.
While we'll discuss the mathematics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

LIGHTING COMPONENTS

- The *Phong* lighting model fits our needs

$$I_{total} = I_a + I_d + I_s + I_e$$



Term	Description	Parameters		Equation
Ambient	Object's color in low-light	• Material	• Light color	$I_a = M_a * L_a$
Diffuse	Object's base color when lit	• Material • Vertex position • Vertex normal	• Light color • Light position	$I_d = M_d * L_d(\hat{n} \cdot \hat{\ell})$
Specular	Highlight on objects	• Material • Vertex position • Vertex normal	• Light color • Light position • Viewer position	$I_s = M_s * L_s(\hat{n} \cdot \hat{h})^s$
Emissive	Glow color	• Material		$I_e = M_e$

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.

LIGHTING PARAMETERS

```
in vec4  aPosition; // vertex position
in vec3  aNormal;   // vertex normal
out vec4 vColor;
```

```
struct Material {
    vec4 a;      // ambient
    vec4 d;      // diffuse
    vec4 s;      // specular
    float shine; // shininess
};
uniform Material M;
```

```
struct Light {
    vec4 position;
    vec4 a; // ambient
    vec4 d; // diffuse
    vec4 s; // specular
};
uniform Light L;
```

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.

LIGHTING MATHEMATICS (VERTEX VERSION)

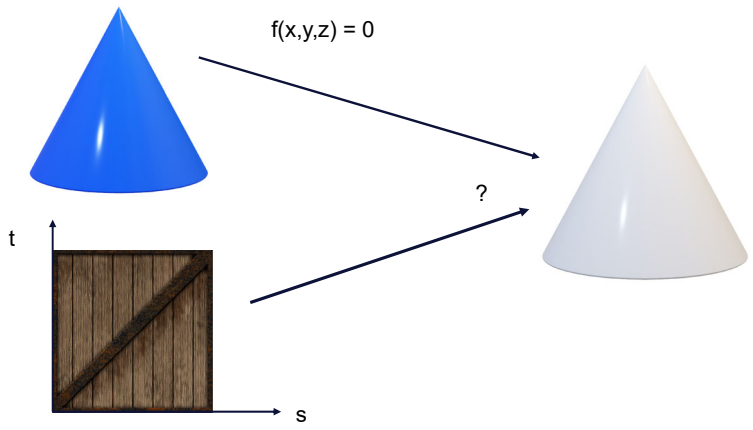

```
vec3 nHat = nomralize(aNormal);                          // Normalized vertex normal
vec3 lHat = normalize(L.position.xyz – aPosition.xyz);   // Light vector
vec3 hHat = normalize(vec3(0,0,1) + lHat));              // Half-angle vector

vec4 I = M.a * L.a                                                  // ambient
       + M.d * L.d * max(dot(nHat, lHat), 0.0)                      // diffuse
       + M.s * L.s * pow(max(dot(nHat, hHat), 0.0), M.shine)        // specular
       + M.e;                                                       // emissive

vColor = I;
```


SIGGRAPH
THINK
BEYOND
2020 S2020.SIGGRAPH.ORG
WEBGL CUBE (PART 5)
TEXTURE

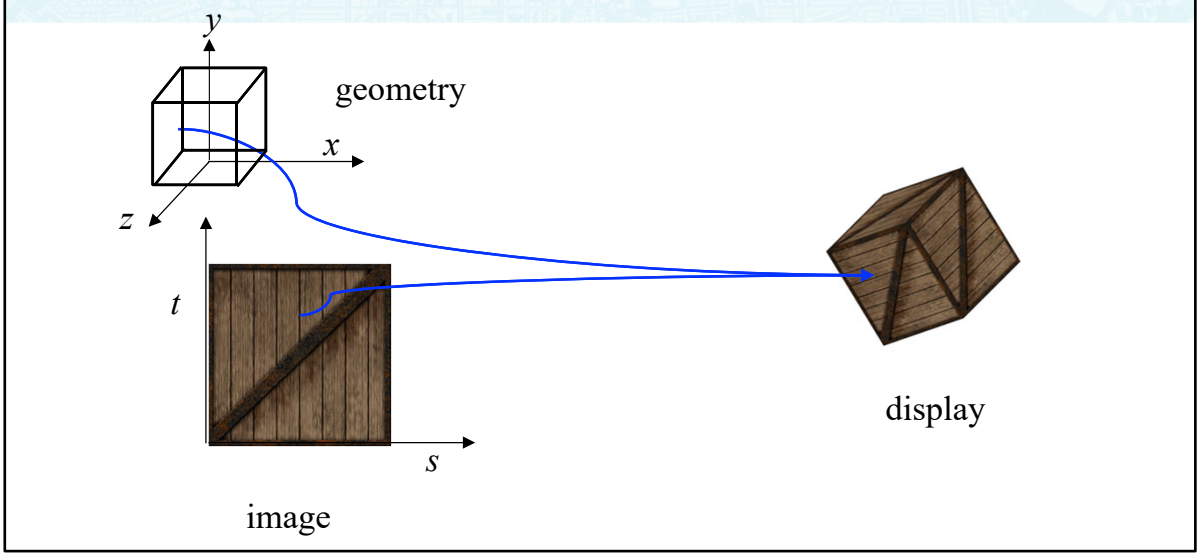
WHY TEXTURE MAPPING

- Although WebGL can process millions of triangles per second on any recent GPU
  - Many applications can produce far more triangles
  - Faster and more accurate to process geometry on a fragment by fragment basis
- Basic idea: Map a 2D image to a surface
  - Gives appearance of great complexity with simple geometry
  - Difficult mathematical problem
- Later we'll see many other uses of texture mapping

WHY GENERAL TEXTURE MAPPING IS HARD



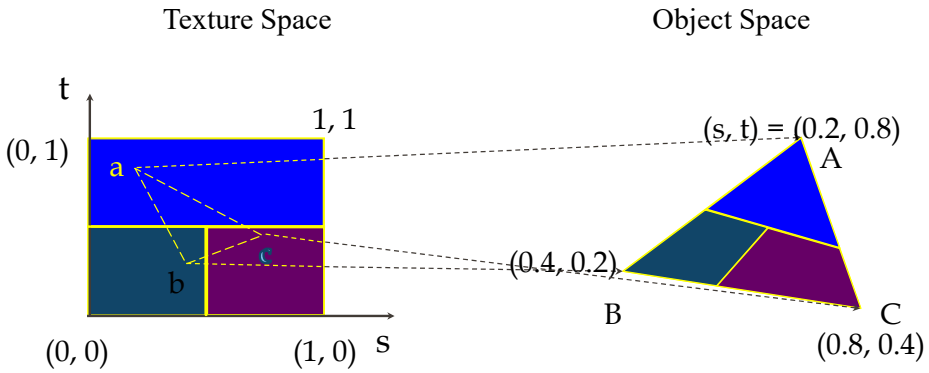
TEXTURE MAPPING



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.

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
  - specify the texture
    1. read or generate image
    2. assign to texture
    3. enable texturing
  - assign texture coordinates to vertices
  - 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.

TEXTURE COORDINATES

- Texture coordinates are a vertex attribute
- Push onto an array for each vertex

```
function quad(a, b, c, d) {
    pointsArray.push(vertices[a]);
    colorsArray.push(vertexColors[a]);
    texCoordsArray.push(texCoord[0]);
```

- Set up VBO

```
var tBuffer = gl.createBuffer();
  gl.bindBuffer( gl.ARRAY_BUFFER, tBuffer );
  gl.bufferData( gl.ARRAY_BUFFER, flatten(texCoordsArray), gl.STATIC_DRAW );

  var texCoordLoc = gl.getAttribLocation( program, "aTexCoord" );
  gl.vertexAttribPointer( texCoordLoc, 2, gl.FLOAT, false, 0, 0 );
  gl.enableVertexAttribArray( texCoordLoc );
```

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.

CONFIGURING A TEXTURE

```
function configureTexture( image ) {

        texture = gl.createTexture();
        gl.bindTexture( gl.TEXTURE_2D, texture );

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

        gl.generateMipmap( gl.TEXTURE_2D );
        gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST_MIPMAP_LINEAR );
        gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST );

        gl.uniform1i(gl.getUniformLocation(program, "texture"), 0);
}
```

APPLYING TEXTURES IN THE FRAGMENT SHADER WEBGL 1.0


```
precision mediump float;

varying vec4 vColor;
varying  vec2 vTexCoord;
uniform sampler2D texture;

void main()
{
    gl_FragColor = vColor*texture2D( texture, vTexCoord );
}

// 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).

APPLYING TEXTURES IN THE FRAGMENT SHADER WEBGL 2.0


```
precision mediump float;

in vec4 vColor;
in  vec2 vTexCoord;
uniform sampler2D texture;
out vec4 fColor;

void main()
{
    fColor = vColor*texture( texture, vTexCoord );
}

// Full example on website
```

In WebGL 2.0, we need to specify a variable for output color.
Note that there is a single texture function.


SIGGRAPH
THINK
BEYOND
2020 S2020.SIGGRAPH.ORG
PRACTICAL WEBGL

INSTANCED RENDERING

- Suppose we want to display multiple instances of an object in a spatial array
- Each with a different color
- But don't want to send each instance

from the GPU to the CPU

- Note each teapot has 1700+ vertices
- WebGL 2.0: Instanced Rendering
- Send one teapot to GPU
- Execute shaders multiple times

INSTANCED RENDERING OF 27 TEAPOTS

- In JS file

```
gl.drawArraysInstanced(gl.TRIANGLES, 0, numVertices, numInstances)
```

- In vertex shader use built-in variable to `gl_InstanceID` to adjust color and position as the shader is executed numInstances times
- Use `gl_InstanceID` to compute an x, y, and z translations

```
int x = gl_InstanceID/(9);
int y = (gl_InstanceID-9*x)/3;
int z = gl_InstanceID- 3*(gl_InstanceID/3);
vec4 translation = someFunction(x, y, z);
```

INSTANCED TEAPOTS (CONT)

- Shader applies translation to each position

```
gl_Position = projectionMatrix * modelViewMatrix * (aPosition + translation);
```

- In color computation, we use x, y and to determine the diffuse color

```
vec4 diffuse =  Kd*vec4(x, y, z, 1.0);
```

```
  vec4 translation = vec4(10.0*(float(x)-1.0) - 5.0, 10.0*(float(y)-1.0) - 5.0,
10.0*(float(z)-1.0) - 5.0, 0.0);
  vec3 pos = (modelViewMatrix * aPosition).xyz;
    vec3 light = lightPosition.xyz;
    vec3 L = normalize( light - pos );


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

    // Transform vertex normal into eye coordinates

    vec3 N = normalize( normalMatrix*aNormal.xyz);

    // Compute terms in the illumination equation
    vec4 ambient = ambientProduct;

    float Kd = max( dot(L, N), 0.0 );
    //vec4  diffuse = Kd*diffuseProduct;
    vec4 diffuse =  Kd*vec4(x, y,z, 1.0);

    float Ks = pow( max(dot(N, H), 0.0), shininess );
    vec4  specular = Ks * specularProduct;

    if( dot(L, N) < 0.0 ) {
              specular = vec4(0.0, 0.0, 0.0, 1.0);
    }

    gl_Position = projectionMatrix * modelViewMatrix * (aPosition + translation);
    gl_Position.xyz = 0.5*gl_Position.xyz;
    vColor = ambient + diffuse +specular;

    vColor.a = 1.0;

}
```

SCENE GRAPHS

- Scene = Geometric Objects + Camera + Light Sources
- Each can be specified as an object
- Higher level API
  - Three.js
  - Built on top of WebGL
  - Gives access to high level WebGL functionality
  - Does not require application to write shaders

CUBE WITH THREE.JS

```
window.onload = function init() {

        var scene = new THREE.Scene();
        var camera = new THREE.PerspectiveCamera( 45, 1.0, 0.3, 4.0 );
        var renderer = new THREE.WebGLRenderer();

        renderer.setClearColor(0xEEEEEE);
        renderer.setSize(512, 512);
        document.body.appendChild(renderer.domElement);

        var cubeGeometry = new THREE.BoxGeometry( 1, 1, 1 );
        var cubeMaterial = new THREE.MeshBasicMaterial( { color: 0xff0000, wireframe: true} );
        var cube = new THREE.Mesh(cubeGeometry, cubeMaterial);

        scene.add(cube);

        camera.position.x = 2.0; camera.position.y = 2.0; camera.position.z = -2.0;
        camera.lookAt(scene.position);

        renderer.render(scene, camera);
}
```

ADVANCED APPLICATIONS OF WEBGL

- 3D Textures (WebGL 2.0)
- Reflection and Bump Maps
- Point Sprites
- Multi- and Off-Screen rendering
  - Particle Systems
  - Shadow Maps
  - Projective Textures
  - GPGPU

3D TEXTURE MAPPING

- WebGL 2.0 supports 3D textures
- A 3D texture is a volume of texels or voxels
- Set up just as with 2D textures

```
gl.texImage3D(gl.TEXTURE_3D, 0, gl.RGBA, texSize, texSize, texSize, 0,
      gl.RGBA, gl.UNSIGNED_BYTE, image3);
```

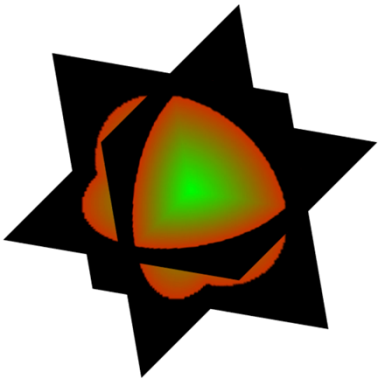
- In fragment shader

```
in vec3 vTexCoord;
uniform sampler3D uTextureMap3D;
fColor = texture(uTextureMap3D, vTexCoord);
```

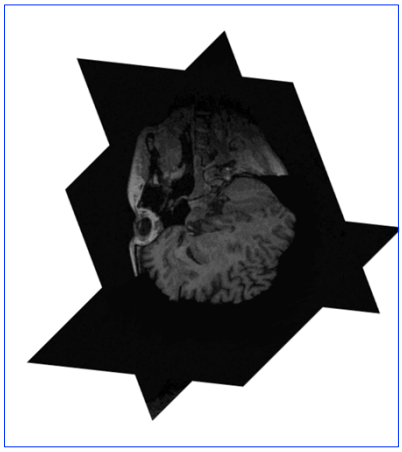
APPLYING 3D TEXTURES

- Using textures to determine colors
  - Apply to surface by assigning texture coordinates by (x, y, z) of fragment
  - Texel value becomes color of surface
- For volume visualization
  - Display three planes aligned with axes to “cut through” texels
  - Used in medical imaging (CT, MRI)
- Can use transparency and render multiple parallel planes

3D TEXTURE MAPPING FOR VOLUME VIZ



texture3D3.html



CUBE MAPS

- Cube maps use six 2D textures corresponding to sides of a cube
- For example

```
gl.texImage2D(gl.TEXTURE_CUBE_MAP_POSITIVE_X, …)
```

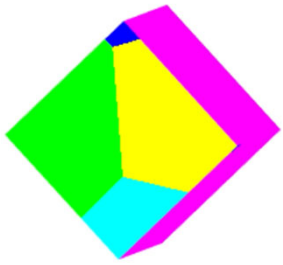
- In fragment shader

```
vec4 texColor = textureCube(texMap, direction);
```

- We can let the direction be the direction of a reflection from the surface of a surface giving a reflection map
- We can also determine six texture maps from rendering a scene giving an environment map

```
precision mediump float;

in vec3 R;
out vec4 fColor;

uniform samplerCube uTexMap;

void main()
{
    vec4 texColor = texture(uTexMap, R);

    fColor = texColor;
}
```

REFLECTION MAP



In JS file, for each face, define a texture map. For example:

```
gl.texImage2D(gl.TEXTURE_CUBE_MAP_POSITIVE_X
,0,gl.RGBA, 1,1,0,gl.RGBA,gl.UNSIGNED_BYTE, red);
```

In vertex shader, compute refection direction and output it

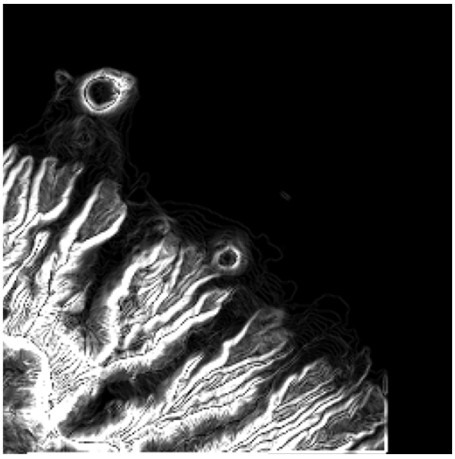
```
out vec3 R;
R = reflect(eyePos, N);
```

In fragment shader, apply texture map as usual

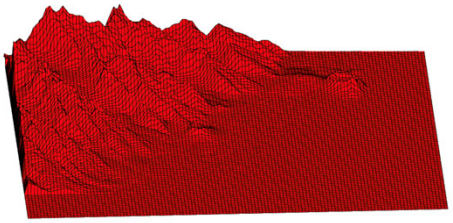
```
in vec3 R;
fColor = texture(uTexMap, R);
```

ReflectionMap1.html

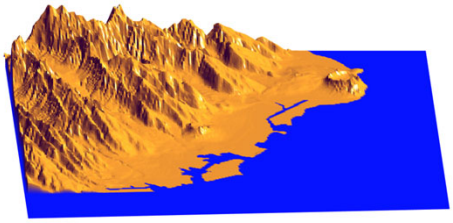
DATA TO MESH



hawaiiImage.html

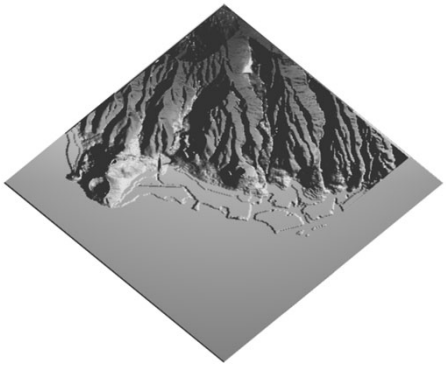


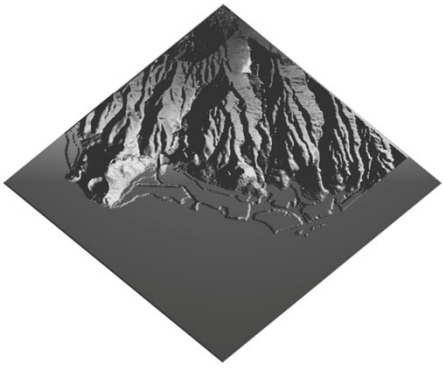
honoluluMesh.html



honoluluMesh3.html

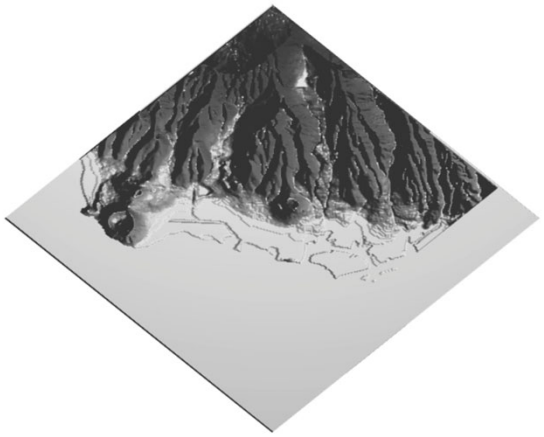
BUMP MAP





BUMP MAP

bumpMap2,html



GPGPU

- The fragment shader can be used to perform many two-dimensional calculations on a grid
  - General Purpose Graphics Processing Unit computing
- Basic idea
  - Render a single rectangle (two triangles)
  - Treat each fragment as a pixel whose color is determined by a calculation in the fragment shader
  - `gl_FragCoord` gives location of pixel

GPGPU EXAMPLES

- Mandelbrot Set:
  - Each point in image is determined by an iterative calculation using complex numbers
  - Fragment color determined by convergence of result
- Image processing
- Each fragment has its own 2D coordinates
  - no loops are needed in shader
  - Muliple fragments are processed in parallel on most GPUs

MANDELBROT SET

Based on iterating the complex equation

$z_{k+1} = z_k^2 + c$

for each c being the coordinates of each fragment

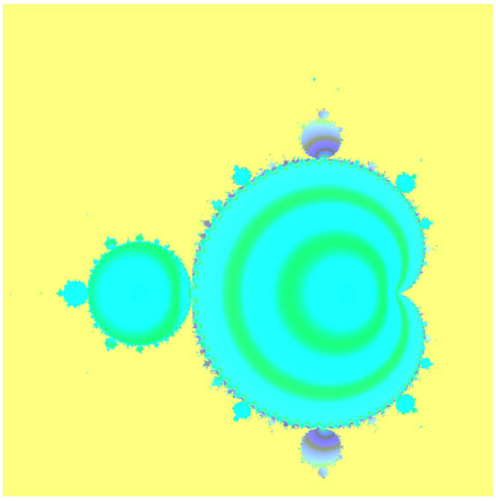
z can
converge hence c is in set
diverege hence c is not in set
or we can't tell after many iterations

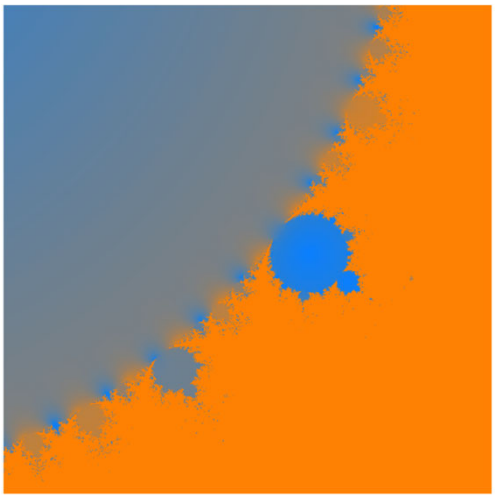
Color fragment based on what happens

Concentrate on regions where we can't tell

Example: mandelbrot2.html

GPGPU MANDELBROT SET





The Mandelbrot set is based on iterating on the equation $z_{k=1} = z_k^2 + c$ where z and c are complex numbers. We do the iteration for c equal to each pixel's location. We color each pixel depending on whether is converges or not. The interesting parts are when the equation neither converges or diverges after a large number if iterations. We can adjust the area to look at regions at different resolution, each time we zoom in showing more complexity. The image on the right covers a small area of the image on the left.

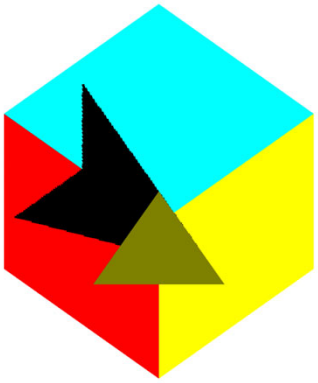
Shaders for complex arithmetic are very simple using the GLSL code, allowing the iteration to be carried out totally in the fragment shader.

OFF-SCREEN RENDERING

- We can render to a buffer other than the frame buffer
  - Buffer contents can then be used as a texture for a normal rendering to the frame buffer
  - Known as Render-to-Texture
- Examples:
  - Shadow Maps (shadowMap.html)
  - Projective Textures (projectiveTexture.html)

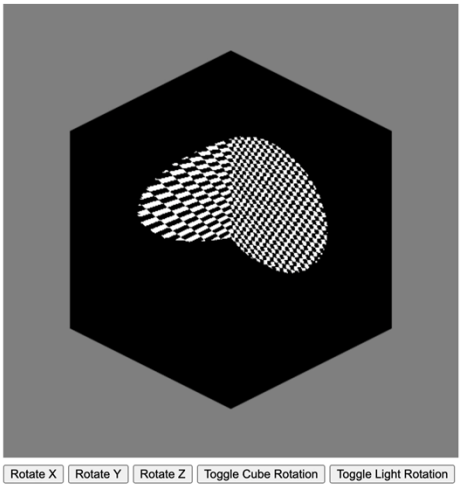
SHADOW MAP

- Shadow is result of projecting object viewed from light source
- Compute and store as texture
- View shadow from viewers perspective during normal rendering



PROJECTIVE TEXTURE

- Like projecting a color slide onto an object.
- Need modelview matrix and position for a viewer at the light source to render texture map correctly
- Use `textureProj()` in fragment shader

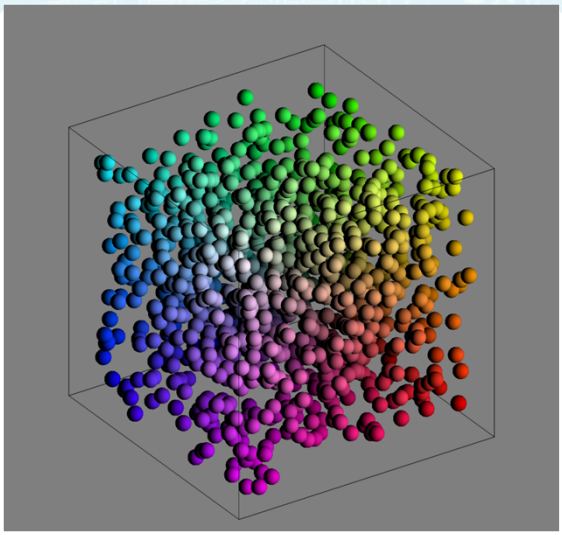


POINT CLOUDS AND POINT SPRITES

- Suppose we have a set of data points in 3D
- Rather than building a geometric object such as a mesh we can render each as a point
- With WebGL each point can be rendered as any other primitive
  - Render each point with a pixel rectangle (`gl_PointSize`)
  - Address each fragment in rectangle (`gl_PointCoord`)
  - Shade each fragment

POINT SPRITE PARTICLE SYSTEM

- CPU/GPU updates location of each point
- Each point is rendered as a square of fragments
- Each fragment in square is shaded as if it were on a sphere using normal and lighting




SIGGRAPH
THINK
BEYOND
2020 S2020.SIGGRAPH.ORG
FINAL THOUGHTS

OTHER TOPICS IN GRAPHICS

- Additional *deferred* techniques
  - draw the frame, then use that rendering as a texture to make a new frame
    - deferred lighting
    - post-processing
- GPU programming and compute
  - GPUs are massively powerful computers (even in mobile phones)
  - Solve generic problems on the GPU
    - particularly useful for *data-parallel* computation
- Augmented and Virtual Reality
  - WebXR brings XR support to Web Browsers

EVOLUTION OF GRAPHICS APIS

- APIs like OpenGL, WebGL, and Direct3D11 are considered *immediate-mode* graphics APIs
  - Information is delivered to the API through function calls
  - Graphics processing occurs when you make a function call (e.g., `gl.drawArrays`)
  - Lots of error checking before any drawing can occur
- Move to more modern approach
  - entire data for a scene is submitted at once
  - no runtime error checking
    - configure a special *validation* mode
- Caused creation of new APIs
- Influencing Web APIs as well - *WebGPU*

API	Creator	Environment
Direct3D12	Microsoft	Windows
Metal	Apple	iOS, macOS, tvOS
Vulkan	The Khronos Group	Windows, Linux, Android

THE FUTURE OF WEBGL

- Recall that WebGL is based on OpenGL ES
  - most new development of features going into Vulkan
- Compute shader facility
  - currently being prototyped as a WebGL extension
  - other extensions to WebGL expose additional features
    - more pixel and texture formats
    - HDR
- Addition shader stages
  - unlikely to see tessellation and geometry shaders in WebGL anytime soon

WebGL 2.0 is a very fully-featured API, capable of addressing many application domains. There are some features from desktop APIs like OpenGL that currently exposed in WebGL, even on platforms that could support them. Many of those features manifest as additional shading stages. If you're curious, *geometry shading*, where procedural generation of additional geometry based on a single geometric primitive (e.g., a line or triangle) isn't well supported on mobile graphics architectures, which is a primary market of WebGL. Further, *tessellation shading*, which is the generation of lines or a mesh from an algebraically-defined surface (e.g., like a Bezier curve, Catmull-Clark surface) also isn't available in the WebGL pipeline. Again, this is likely a result of the GPU architectures that OpenGL ES (and thus WebGL) can support.

Finally, compute shaders aren't universally exposed in WebGL either currently, not so much for lack of support in GPUs, but more for security of web browsers, which is the environment where WebGL executes. However, compute shaders are definitely under discussion and could be coming to a browser near you in the very near future.


SIGGRAPH
THINK
BEYOND
2020 S2020.SIGGRAPH.ORG
RESOURCES

BOOKS

- Modern OpenGL
  - WebGL Insights
  - The OpenGL Programming Guide, 9th Edition
  - Interactive Computer Graphics: A Top-down Approach using WebGL, 8th Edition
  - WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL
  - WebGL Beginner's Guide
- Three.js
  - Learning Three.js, 2nd Edition
- Other resources
  - 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	InteractiveComputerGraphics.com
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 Haines's Udacity course	bit.ly/intro3D

THANKS!

- Feel free to drop us any questions:
  - angel@cs.unm.edu
  - shreiner@siggraph.org
- Course notes and programs available at
  - InteractiveComputerGraphics.com
  - www.cs.unm.edu/~angel

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