STEVENS INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

Week 02:
Intro to WebGL and JavaScript

CS-537: Interactive Computer Graphics
Dr. Chloe LeGendre
Department of Computer Science

For academic use only.

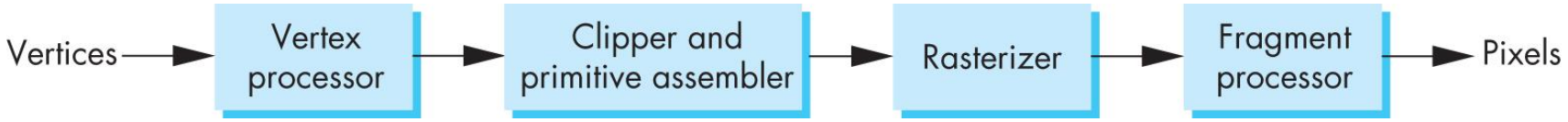
Some materials from the companion slides of Angel and Shreiner, “Interactive Computer Graphics, A Top-Down Approach with WebGL.”

Objectives

- Introduce and justify WebGL
- Introduce fundamentals of JavaScript required for this course
- Introduce a few “hello world” graphics programs using WebGL

A [Very] Brief History of Graphics APIs

- Our textbook spends a bunch of time talking about the history of the development of graphics APIs. You don't need to remember the details, but you should know that WebGL is the web-friendly version of OpenGL. Web-GL = Graphics Library for the web.
- OpenGL is a platform-independent graphics API that allows programmers to read/write data to/from GPU and run GPU programs. Lots of versions exist.
- GPU programs are called "shaders" – you can think of these as a set of instructions to perform on input data, producing some output data. Usually (but not always) we want to display the output data.
- Application's job is to send data to GPU, and then the GPU does all the rendering steps.



Why WebGL?

- Before WebGL: No clear way to use the GPU of a local machine to render an image in a web browser.
- WebGL: JavaScript interface to OpenGL ES 2.0
  - ES = “Embedded systems” – indicates that this “flavor” of OpenGL is suitable for consoles, phones, appliances, and vehicles, or other “low power” devices. These kinds of devices are often accessing websites.
- With WebGL: user visits URL of a WebGL application on a remote system, and the program will run on the local system making use of the local graphics hardware (your PC or phone’s GPU).
- WebGL does **not** have all features of the latest version of OpenGL, but rather a **subset**.
- We’re using WebGL for this course because we can learn the foundations of graphics programming without worrying about compiling code for different operating systems and hardware configurations.

Coding in WebGL

- You can run WebGL programs on any recent browser (Chrome, Firefox, Safari)
- All application code is written in JavaScript (JS)
- JS runs within a browser
- To make a WebGL example, at a minimum you will need an HTML file and a JS file, loaded by the browser. (No prior knowledge of JS or HTML is assumed for this course).
- We will take a minimalistic approach and use only core JS functions and HTML for our assignments, with no extras or variants. We won't use CSS or JQuery.
- We'll focus on graphics, so the websites you'll make for the assignments will be very basic. You're welcome to use CSS for assignments, but it won't be required.

JavaScript Notes

- JavaScript (JS): language of the Web
  - All browsers will execute JS code
  - JS is an interpreted object-oriented language
  - Lots of decent web tutorials for JS, and our assignments assume no prior knowledge.
- Is JS slow?
  - JS engines in browsers are getting fast.
  - It's not a key issue for us for graphics programming, because once we transfer data to the GPU, it doesn't really matter how we got it there.
- JS has a lot of components, but we don't need them all
  - Choose which parts we need to use
  - Don't try to make your code look like C or Java

JavaScript Notes (II)

- JS has very few native types:
  - numbers
  - strings
  - booleans
- Only one numerical type: 32-bit float. Consider:
  - var x = 1;
  - var x = 1.0; // these two lines produce the same result
  - Two operators for equality: == and ===
- JS is dynamically typed
  - JS interpreter assigns variables a type at runtime based on the variable's value at the time (not at compile time)

JavaScript Variable Scoping

- JavaScript has two scopes – global and local
- Any variable declared outside of a function belongs to the global scope and is therefore accessible from anywhere in your code.
- Each function has its own scope, and any variable declared within that function is only accessible from that function and any nested functions.
- JS, unlike many other languages, does not support block level scoping. This means that declaring a variable inside of a block structure like a for-loop, does not restrict that variable to the loop. Instead, the variable will be accessible from the entire function.
- Our code samples and starter assignments will use a lot of global variables, which is not the recommended best programming practice for JavaScript. Our focus is WebGL, not JavaScript, so we will content ourselves to use global variables as needed for simplicity.

JavaScript Variable Scoping (II)

- Variables are “hoisted” within a function
  - “Hoisting” = all variable declarations (*not* definition/assignments) are moved to the top of their scope automatically by JS interpreter.
  - This means that, weirdly, you can use a variable before it is declared
- Note that functions are first class objects in JS, which means they can be:
  - Stored in a variable, object, or array.
  - Passed as an argument to a function.
  - Returned from a function.

JavaScript Arrays

- JS arrays are objects, which inherit a few methods

```
var a = [1, 2, 3];
a.length     // 3
a.push(4); // length now 4
a.pop();   // returns 4
```

- This avoids use of many loops and indexing
- But it's a problem for WebGL which expects C-style arrays

JavaScript Typed Arrays

- JS has typed arrays that are like C arrays
  - var a = new Float32Array(3)
  - var b = new Uint8Array(3)
- Generally we'll prefer to work with standard JS arrays and convert them to typed arrays only when we need to send data to the GPU.
- There's a function in a library that we'll use for this course that converts between the types:
  - flatten() function in the file MV.js which is a basic matrix and vector library we'll use for our assignments.

A First WebGL example

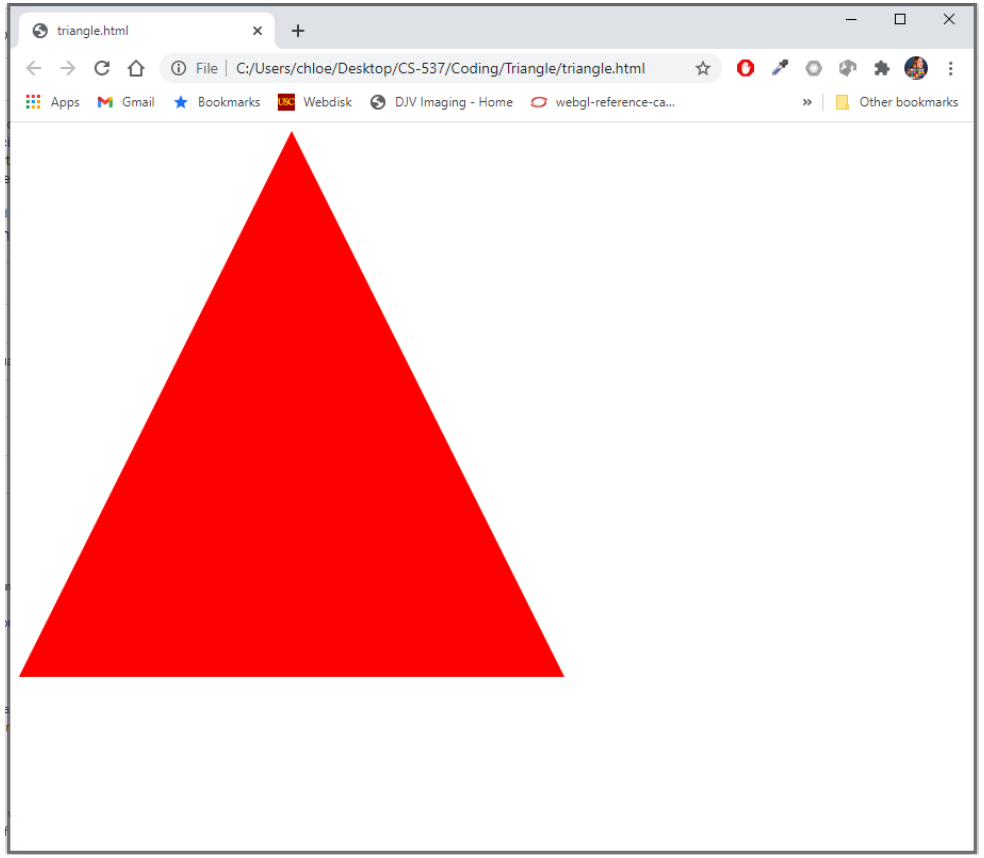
- Five Steps
  1. Describe the web page (in HTML file)
     - Request WebGL canvas
     - Read in necessary files
  2. Define shaders (GPU program) (in HTML file)
     - Can be done with a separate file, depending on the browser
  3. Compute or specify data (in JS file)
  4. Send data to GPU (in JS file)
  5. Render data (in JS file)

Program Execution

- WebGL runs within the browser
  - Complex interaction among the operating system, the window system, the browser and your code (HTML and JS)
- Simple model
  - Start with HTML file
  - Files are read in asynchronously
  - Start with onload function
    - Event driven input

A First WebGL Example: triangle.html

- You should run this example for yourself (see instructions in course module) and inspect the code.



Example code: triangle.html (I)

```
<!DOCTYPE html>
<html>
<head>
<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec4 vPosition;
void main(){
  gl_Position = vPosition;
}
</script>

<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;
void main(){
    gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
</script>
```

“vertex shader”: part of GPU program which, in this example, shows how to compute the position for each input vertex

“fragment shader”: part of GPU program which, in this example, shows how to compute the color for each fragment (potential pixel)

Example code: triangle.html (II)

```
<script type="text/javascript" src="../Common/webgl-utils.js"></script>
<script type="text/javascript" src="../Common/initShaders.js"></script>
<script type="text/javascript" src="../Common/MV.js"></script>
<script type="text/javascript" src="triangle.js"></script>
</head>
<body>
<canvas id="gl-canvas" width="512" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>
</body>
</html>
```

Common libraries we'll use in our examples

JS file for our WebGL application

Defining the rendering "canvas" or working space

Example code: triangle.js (I)

```
var gl;
var points;

window.onload = function init(){
    var canvas = document.getElementById( "gl-canvas" );
     gl = WebGLUtils.setupWebGL( canvas );
     if ( !gl ) { alert( "WebGL isn't available" );}

// Define three 2D vertices

var vertices = [
        vec2( -1, -1 ),
        vec2(  0,  1 ),
        vec2(  1, -1 )
];
```

Declare some global variables we'll use later

WebGL initialization boilerplate using functions from webgl-utils.js

Note: vec2 data type defined in MV.js library

Example code: triangle.js (II)

```
//  Configure WebGL
gl.viewport( 0, 0, canvas.width, canvas.height );
gl.clearColor( 1.0, 1.0, 1.0, 1.0 );

//  Load shaders
var program = initShaders( gl, "vertex-shader", "fragment-shader" );
gl.useProgram( program );

// Initialize attribute buffers and Load the data into the GPU
var bufferId = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
gl.bufferData( gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW );
```

Shader initialization boilerplate using functions from initShaders.js
GL object will use the GPU program defined by these two shaders.

Creates a GPU buffer (location in memory) with an ID number, tells the GL object that future steps involving buffers will involve this buffer ID,
loads vertices data to GPU (notice “flatten” to convert to C-type array)

gl.STATIC_DRAW enum tells the GPU that you’re unlikely to be passing new data to this buffer during program execution

Example code: triangle.js (III)

```
// Associate shader variables with data buffer
var vPosition = gl.getAttribLocation( program, "vPosition" );
gl.vertexAttribPointer( vPosition, 2, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vPosition );
render();
}; // init() function definition ends here

function render() {
   gl.clear( gl.COLOR_BUFFER_BIT );
   gl.drawArrays( gl.TRIANGLES, 0, 3 );
}
```

Get the location of the shader attribute "position" – you want to upload data from the application to the GPU here. Note "vPosition" matches the name of the shader variable in the vertex shader.

Then describe the kind of vertex attribute at this location:
2 = size of each data point to be uploaded (our vertices are 2D);
gl.FLOAT = data type

Then make sure the shader knows to use the data ("enabled")

Clear the frame buffer and render triangles;
0 = starting index
3 = number of vertices

Event Loop

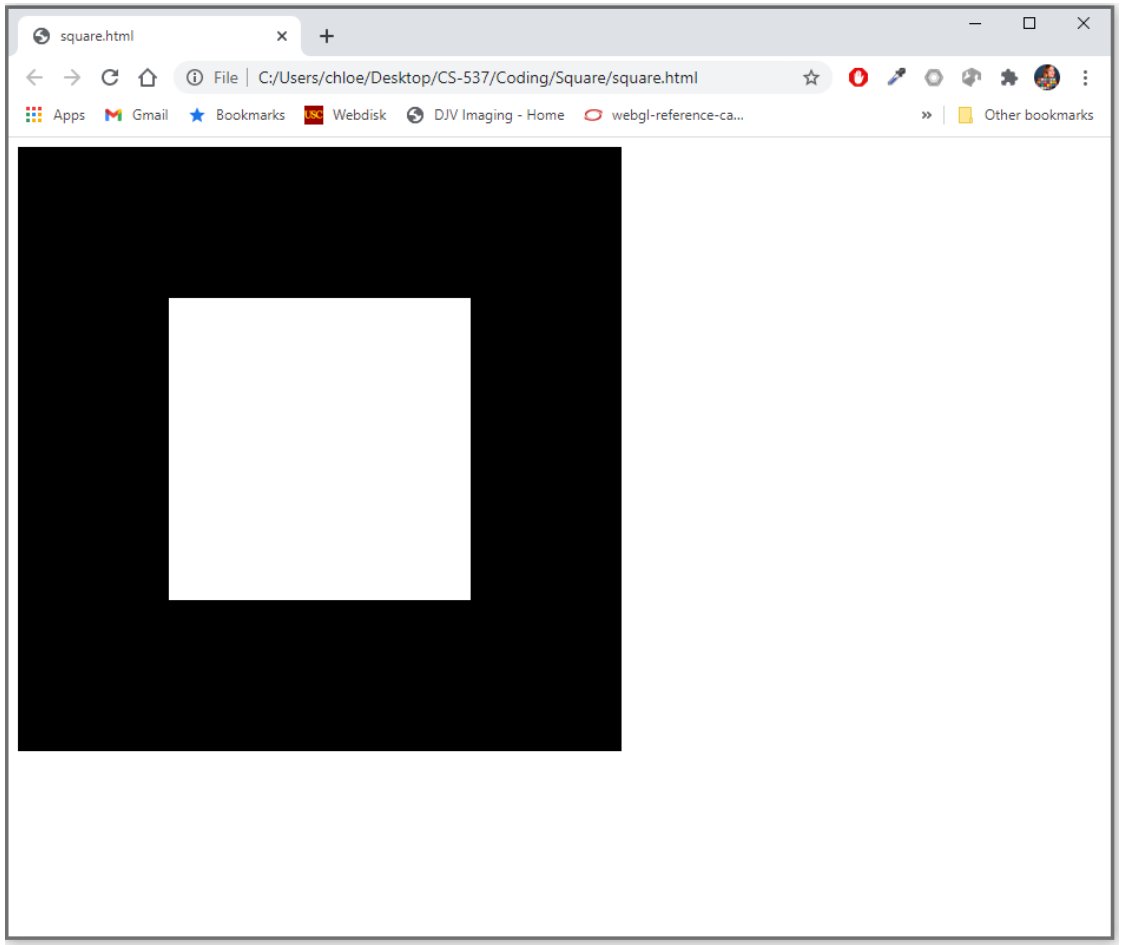
- The sample program specifies a render function with is an *event listener* or *callback* function
- Every program needs a render callback function
- For a static application (as the example just shown), we only need to execute the render function once
- In a dynamic application, the render function can call itself recursively, be each redrawing of the display must be triggered by an event (more on this later when we cover animation).

WebGL and GLSL

- WebGL requires shaders (GPU programs)
- Most rendering “action” happens in shaders
- The main job of the WebGL application is to get data to the GPU
- The vertex and fragment shader form the GPU program (we saw a very simple pair of vertex/fragment shaders in the example above)
- GLSL (OpenGL Shading Language) is the programming language used to write shader programs (within our HTML file)
- WebGL has functions to compile, link, and get information to shaders.

A Second WebGL Example: square.html

- You should run this example for yourself (see instructions in course module) and inspect the code.



Example code: square.html (I)

```
<!DOCTYPE html>
<html>
<head>
<script id="vertex-shader" type="x-shader/x-vertex">

attribute vec4 vPosition;
void main()
{
    gl_Position = vPosition;
}
</script>

<script id="fragment-shader" type="x-shader/x-fragment">

precision mediump float;

void main()
{
    gl_FragColor = vec4( 1.0, 1.0, 1.0, 1.0 );
}
</script>
```

Shaders

- We assign names to the shaders that we can use in the JS file
- These two are trivial pass-through (do nothing) shaders that which set the two required built-in variables
  - gl_Position
  - gl_FragColor
- Note both shaders are full programs
- Must set precision in fragment shader

Example code: square.html (II)

```
<script type="text/javascript" src="../Common/webgl-utils.js"></script>
<script type="text/javascript" src="../Common/initShaders.js"></script>
<script type="text/javascript" src="../Common/MV.js"></script>
<script type="text/javascript" src="square.js"></script>
</head>

<body>
<canvas id="gl-canvas" width="512" height="512">
Oops ... your browser doesn't support the HTML5 canvas element
</canvas>
</body>
</html>
```

Files

- **`../Common/webgl-utils.js`**: Standard utilities for setting up WebGL context in Common directory
- **`../Common/initShaders.js`**: contains JS and WebGL code for reading, compiling and linking the shaders
- **`../Common/MV.js`**: our matrix-vector package (for this course)
- **`square.js`**: the application file

Example code: square.js (I)

```
var gl;
var points;

window.onload = function init(){
var canvas = document.getElementById( "gl-canvas" );

    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" );
 }

// Four Vertices

var vertices = [
        vec2( -0.5, -0.5 ),
        vec2(  -0.5,  0.5 ),
        vec2(  0.5, 0.5 ),
        vec2( 0.5, -0.5)
];
```

Notes on JS file (I)

- **`onload`**: determines where to start execution when all code is loaded
- canvas gets WebGL context from HTML file
- vertices use vec2 type in MV.js
- JS array is not the same as a C or Java array
  - object with methods: vertices.length // 4
- Values in clip coordinates (later we'll define all the required coordinate frames)

Example code: square.js (II)

```
 //  Configure WebGL
 gl.viewport( 0, 0, canvas.width, canvas.height );
 gl.clearColor( 0.0, 0.0, 0.0, 1.0 );

  //  Load shaders
var program = initShaders( gl, "vertex-shader", "fragment-shader" );
 gl.useProgram( program );

  // Initialize attribute buffers and load the data into the GPU

 var bufferId = gl.createBuffer();
 gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
 gl.bufferData( gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW );

  // Associate out shader variables with our data buffer
 var vPosition = gl.getAttribLocation( program, "vPosition" );
 gl.vertexAttribPointer( vPosition, 2, gl.FLOAT, false, 0, 0 );
 gl.enableVertexAttribArray( vPosition );
```

Notes on JS file (II)

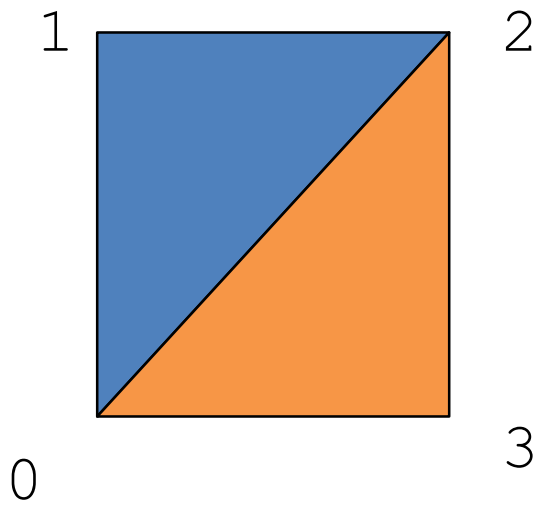
- **`initShaders`** used to load, compile and link shaders to form a program object
- Load data onto GPU by creating a **vertex buffer object** on the GPU
  - Note use of flatten() to convert JS array to an array of float32's
- Finally we must connect variable in program with variable in shader
  - need name, type, location in buffer

Example code: square.js (III)

```
 render();
};

function render() {
    gl.clear( gl.COLOR_BUFFER_BIT );
    gl.drawArrays( gl.TRIANGLE_FAN, 0, 4 );
}
```

Clear the frame buffer and render two triangles using the “triangle fan”;
0 = starting index
4 = number of vertices (note this is not a triangle, it’s a different type - we’ll return here next week!)



A first note on Coordinate Systems

- The units in **`vertices`** are determined by the application and are called *object, world, model* or *problem coordinates*
- Eventually pixels will be produced in *screen space* (in the WebGL canvas)
- WebGL also uses some internal representations that usually are not visible to the application but are important in the shaders
- Most important is *clip coordinates* (see next slides)

A first note on Coordinate Systems and Shaders

- Vertex shader must output in clip coordinates
  - To be visible, vertices must be in the range -1.0 and 1.0 in clip coordinates.
- Application can provide vertex data in any coordinate system, but the shader must eventually produce gl_Position in clip coordinates
- Simple examples above use clip coordinates directly

Default WebGL camera

- WebGL places a camera at the origin in object space pointing in the negative z direction
- The default viewing volume is a box centered at the origin with sides of length 2