CS 418: Interactive Computer Graphics

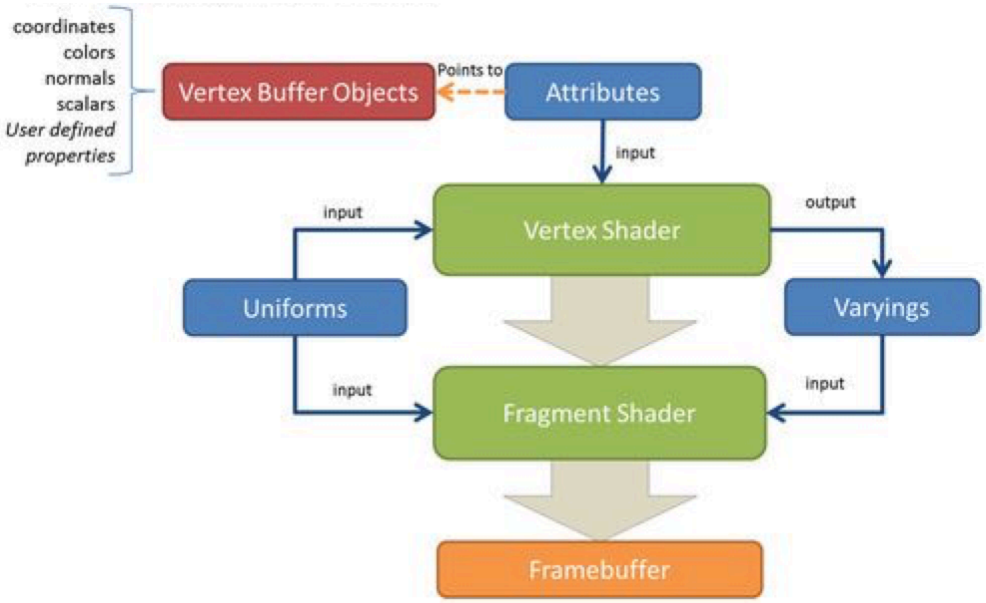
Introduction to WebGL: Geometric Primitives

Eric Shaffer

Things we will learn…

- Loading shaders from the DOM rather than strings
- Geometric primitives supported by WebGL
  - Triangles
  - Lines
  - Point sprites
- you can grab code from https://courses.engr.illinois.edu/cs418/

WebGL Rendering Pipeline



From WebGL Beginner'sGuide by Cantor and Jones

Loading Shaders using the DOM API

```
<script id="shader-vs" type="x-shader/x-vertex">
attribute vec3 aVertexPosition;
void main(){
gl_Position = vec4(aVertexPosition,1.0);
}
</script>

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

We can include the shader code in the HTML using the <script> tag

We can then read the shader code into strings using the Document Object Model API.

Note that we have to give each shader an ID so that we can refer to it later.

Remember that shader code is in GLSL and not JavaScript

Reading the Shaders from the DOM

```
function loadShaderFromDOM(id) {
  var shaderScript = document.getElementById(id);

  // If we don't find an element with the specified id we do an
early exit
  if (!shaderScript) {
    return null;
  }
  // Loop through the children for the found DOM element and
  // build up the shader source code as a string
   var shaderSource = "";
   var currentChild = shaderScript.firstChild;
   while (currentChild) {
       if (currentChild.nodeType == 3) {
       // 3 corresponds to TEXT_NODE
       shaderSource += currentChild.textContent;
       currentChild = currentChild.nextSibling;
    }
   var shader;
   if (shaderScript.type == "x-shader/x-fragment") {
       shader = gl.createShader(gl.FRAGMENT_SHADER);
   } else if (shaderScript.type == "x-shader/x-vertex")  {
       shader = gl.createShader(gl.VERTEX_SHADER);
   } else {
     return null;
   }
   gl.shaderSource(shader, shaderSource);
   gl.compileShader(shader);
   if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
         alert(gl.getShaderInfoLog(shader));
         return null;
   }
   return shader;
}
```

This JavaScript function takes an id as a parameter.

It looks up the specified shader script in the DOM using the id and then builds a JavaScript string containing text read via the DOM.

The appropriate shader is then created and returned.

While it would be better to be able to read the shader script from a file, this method at least lets you easily copy and paste shader code into the HTML file.

WebGL Vertex Buffer Objects

- Vertex Buffer Objects hold data we pass to the vertex shader
- First we need to create a WebGL buffer object buffer

```
var myBuffer = gl.createBuffer();
```

WebGL Vertex Buffer Objects

- WebGL is a state machine
- Binding a buffer makes it the current buffer

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

  - Any subsequent buffer operations apply to it
  - …until a another buffer is bound

WebGL Vertex Buffer Objects

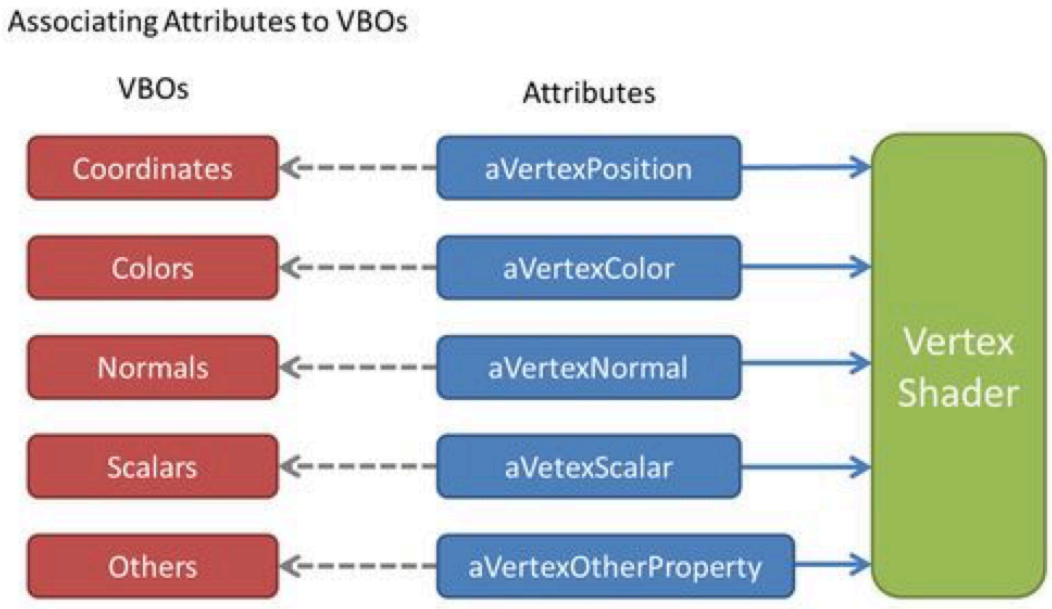
- Next, we need to pass data to the buffer object
- If the vertex data is in a JS array,
  we need to convert it to a typed array

```
gl.bufferData( gl.ARRAY_BUFFER,
                    new Float32Array( vertices), gl.STATIC_DRAW);
```

  - STATIC_DRAW is a hint to WebGL that we won't modify the data

Pointing an attribute to a buffer

- Attribute data in the vertex shader is pulled from VBOs



From WebGL Beginner's Guide by Jones and Kantor

Pointing an attribute to a buffer

- Attribute data in the vertex shader is pulled from VBOs

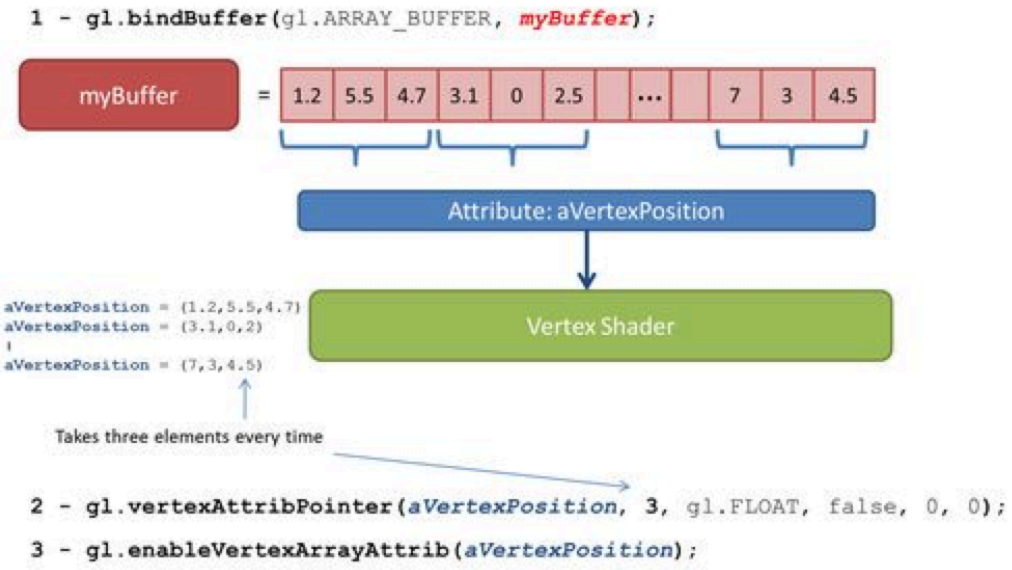
```
gl.vertexAttribPointer( Index, Size, Type, Norm, Stride, Offset);
```

Index → index of the attribute to map the VBO to
Size → number of values per vertex
Type → data type (e.g. FLOAT)
Norm → numeric conversion…we'll set it FALSE mostly
Stride → layout of data in buffer, 0 means sequential
Offset → Where data starts in buffer…usually 0 fo us

Enabling the attribute

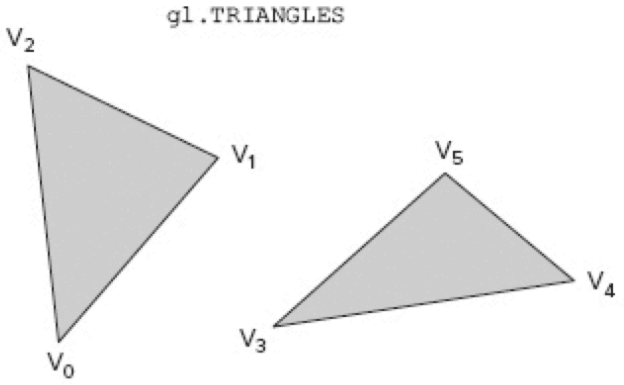
- Finally, we need to actiavte the attribute

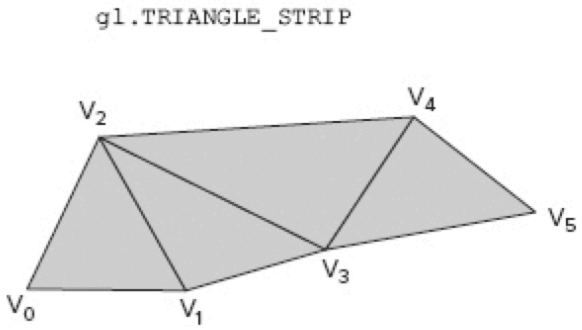
```
gl.enableVertexAttribArray(aVertexPosition)
```

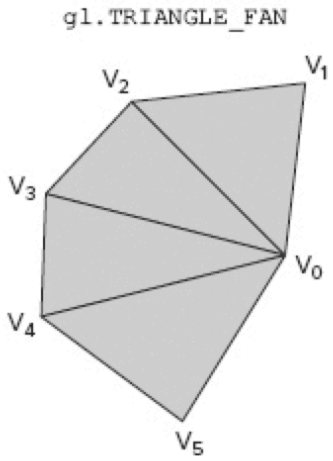


From *WebGL Beginner's Guide* by Jones and Kantor

Geometric Primitives in WebGL







WebGL supports 3 basic geometric primitives:

1. Triangles
2. Lines
3. Point Sprites

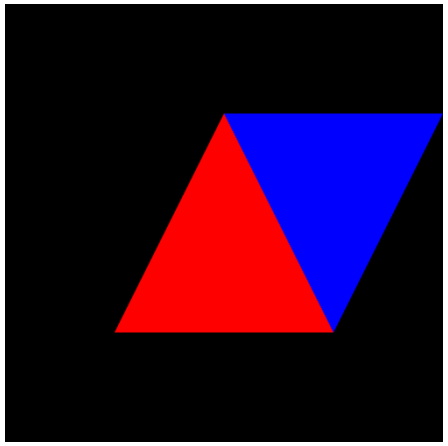
We've already seen one way to send triangles into the pipeline.

There are three different triangle drawing modes depending on how you specify the connectivity:

gl.TRIANGLES
gl.TRIANGLE_STRIP
gl.TRIANGLE_FAN

gl.TRIANGLES

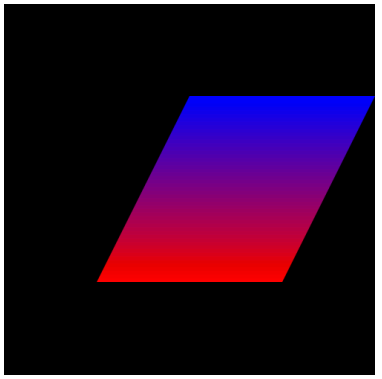
```
vertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexPositionBuffer);
var triangleVertices = [
        0.0,  0.5,  0.0,
       -0.5, -0.5,  0.0,
        0.5, -0.5,  0.0,
        0.0,  0.5,  0.0,
        1.0, 0.5,  0.0,
        0.5, -0.5,  0.0
  ];
gl.bufferData(gl.ARRAY_BUFFER, new
        Float32Array(triangleVertices), gl.STATIC_DRAW);
vertexPositionBuffer.itemSize = 3;
vertexPositionBuffer.numberOfItems = 6;
```



- Assuming you are using gl.drawArrays():
  - Each triangle requires you specify three new vertices
  - i.e. you can't reference vertex data already in the buffer
  - Number of triangles = number vertices/3

gl.TRIANGLE_STRIP

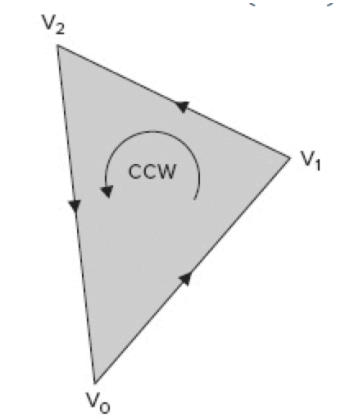
```
vertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER,
vertexPositionBuffer);
var triangleVertices = [
        -0.5, -0.5,  0.0,
         0.5, -0.5,  0.0,
         0.0,  0.5,  0.0,
         1.0, 0.5,  0.0,
];
gl.bufferData(gl.ARRAY_BUFFER, new
        Float32Array(triangleVertices),
gl.STATIC_DRAW);
vertexPositionBuffer.itemSize = 3;
vertexPositionBuffer.numberOfItems = 4;
```



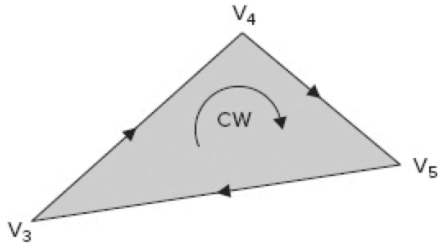
- Allows you to reuse vertices when drawing triangles that share vertices.
- Number of triangles = what?
- Notice that per-triangle color is not easy to achieve
- Order of the vertices is important

Winding Order

- Winding order is determined by the order of the vertices making up a triangle when seen from the viewing direction.
- Equivalently, winding order tells you the direction of the triangle surface normal.
- CCW is traditional and is WebGL default: gl.frontFace(gl.CCW)
- For triangle strips, winding order determines the order in which vertices in the buffer are used to form triangles



Counterclockwise Winding

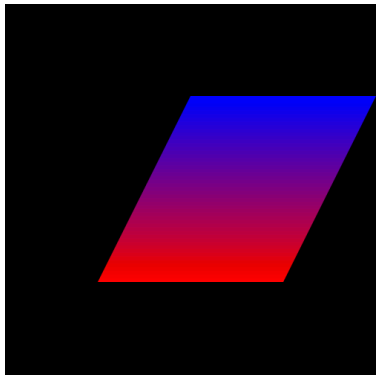


Clockwise Winding

TRIANGLE NUMBER	CORNER 1	CORNER 2	CORNER 3
1	V_0	V_1	V_2
2	V_2	V_1	V_3
3	V_2	V_3	V_4
4	V_4	V_3	V_5

gl.TRIANGLE_FAN

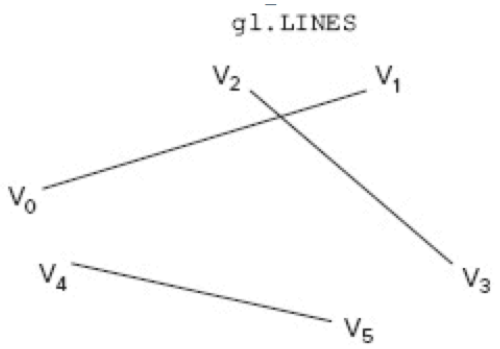
```
vertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER,
vertexPositionBuffer);
var triangleVertices = [
         0.5, -0.5,  0.0,
        1.0, 0.5,  0.0,
         0.0,  0.5,  0.0,
        -0.5, -0.5,  0.0,
];
gl.bufferData(gl.ARRAY_BUFFER, new
        Float32Array(triangleVertices),
gl.STATIC_DRAW);
vertexPositionBuffer.itemSize = 3;
vertexPositionBuffer.numberOfItems = 4;
```

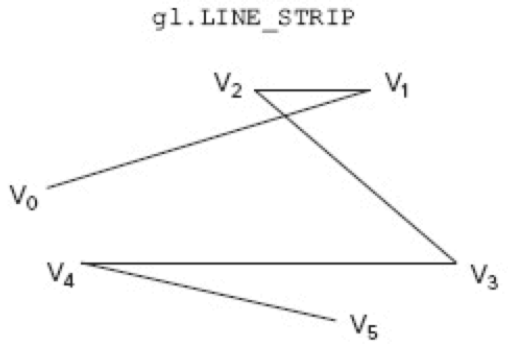


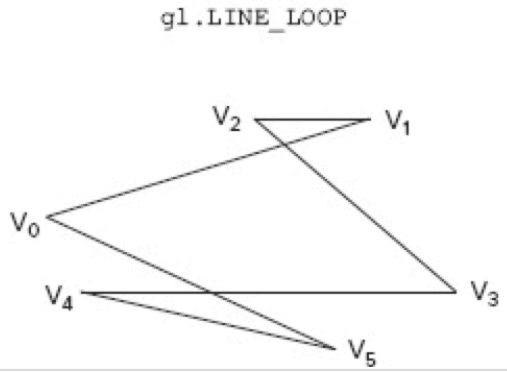
- First vertex is the fan center
- Next two vertices specify the first triangle
- Each succeeding vertex forms a triangle with the center and previous vertex
- How many triangles for a given number of vertices?
- Are fans and strips equivalent?

Lines

- gl.LINES draws independent lines
  (v0,v1), (v2,v3), (v4,v5)
- gl.LINE_STRIP draws a polyline
  (v0,v1),(v1,v2),(v2,v3),(v3,v4),(v4,v5)
- gl.LINE_LOOP draws a line strip with a line connecting the first and final vertex







Point Sprites

- Specified with gl.POINTS mode
- Renders one point per vertex in the buffer
- using N pixels in the point is specified using gl.pointSize(N)

What if I can't draw everything I want in single triangle strip

```
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer1);

gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
                       vertexBuffer1.itemSize, gl.FLOAT, false, 0, 0);

gl.bindBuffer(gl.ARRAY_BUFFER, vertexColorBuffer1);

gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
                       vertexColorBuffer1.itemSize, gl.FLOAT, false, 0, 0);

gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer1);

gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer2);

gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
vertexBuffer2.itemSize, gl.FLOAT, false, 0, 0);

gl.bindBuffer(gl.ARRAY_BUFFER, vertexColorBuffer2);

gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
vertexColorBuffer2.itemSize, gl.FLOAT, false, 0, 0);

gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer2.numItems);
```

- Use more than one vertex buffer
  - Set them up like you did the the first buffer
- You call gl.drawArrays multiple time in your draw function:

Minimizing `draw` calls

- You generally want as few calls to gl.drawArrays as possible
  - Same is true for gl.drawElements…we'll discuss that later
- For triangle strips, you can insert degenerate triangles into the stream
  - These triangles will have two identical vertices and 0 area
- Can connect strips using a sequence of degenerate triangles
  - Better to do this with gl.drawElements
  - Bigger performance hit for gl.drawArrays due to cache effects and processing the same vertex multiple times