More Rendering

CS 385 - Class 4
3 February 2022

Piazza Discussion Board

- Please join the board at

piazza.com/sonoma/spring2022/cs385

Rendering (Drawing)

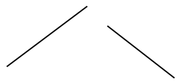
Rendering

- WebGL supports two drawing commands
- `gl.drawArrays()`
  - sends sequential vertices to the vertex shader
    - `gl.TRIANGLE_STRIP` indicates how collections of vertices should be formed into geometric primitives
    - `start` indicates which vertex in the buffer to send first
    - `count` is the number of vertices to send
- `gl.drawElements()`
  - we'll talk about this momentarily

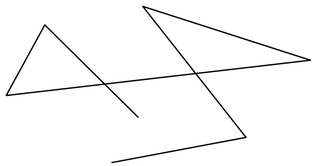
```
function Square() {
    this.count = 4;

    this.render = function () {
        ... // bind buffers

        var start = 0;
        var count = this.count;
        gl.drawArrays(gl.TRIANGLE_STRIP,
            start, count);
     };
};
```

WebGL Geometric Primitives

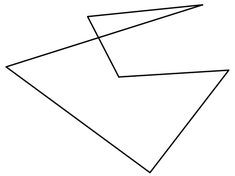
gl.POINTS



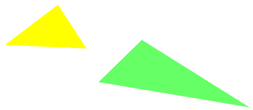
gl.LINES



gl.LINE_STRIP



gl.LINE_LOOP



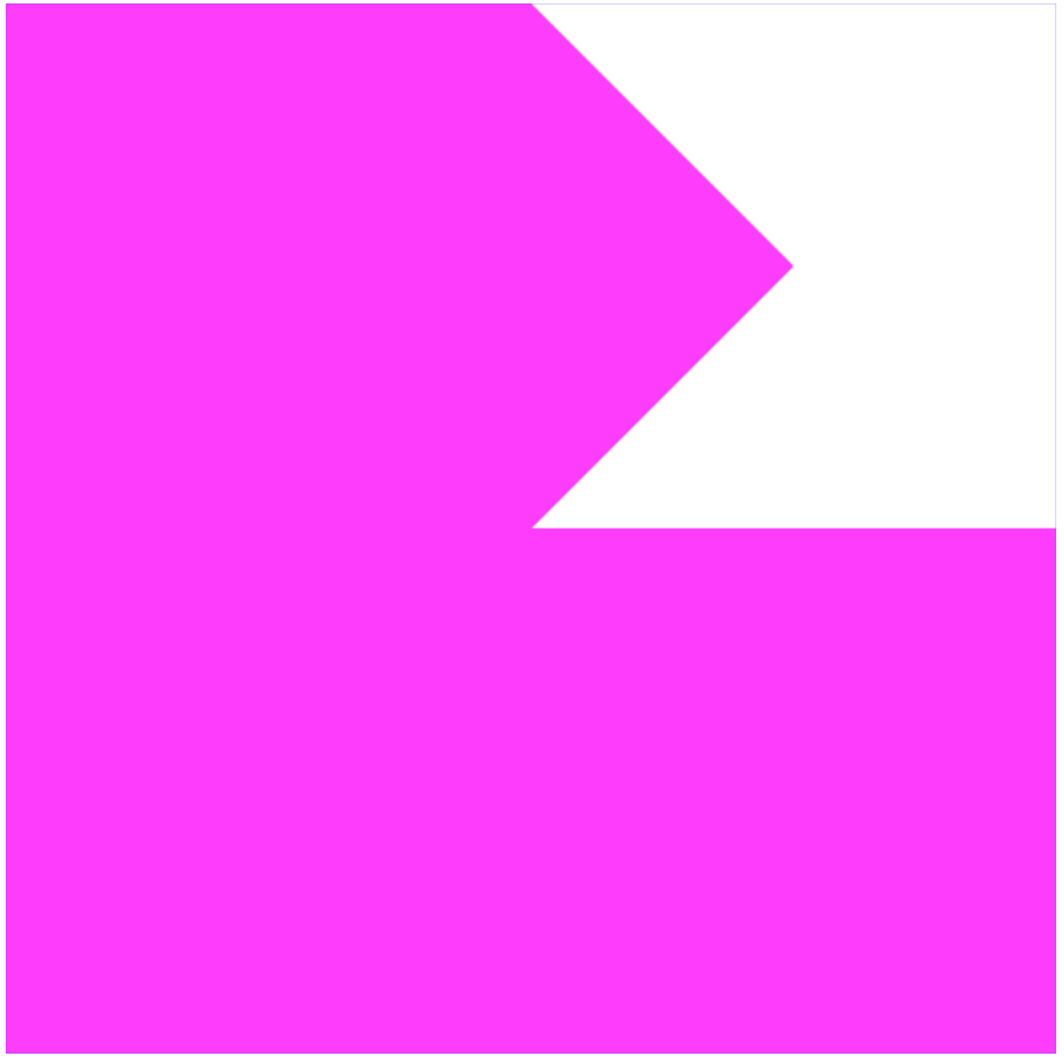
gl.TRIANGLES

gl.TRIANGLE_STRIP

gl.TRIANGLE_FAN

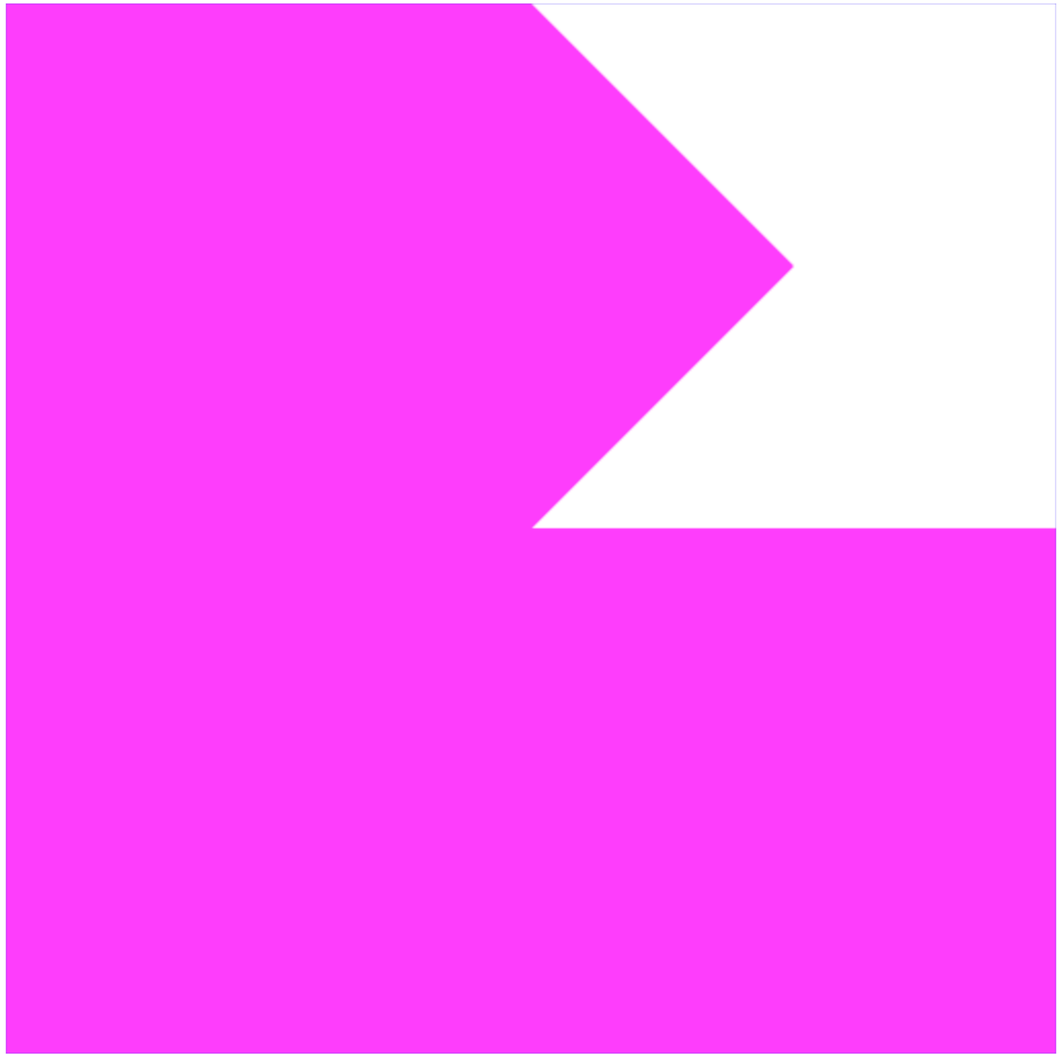
Rendering Results

- Using
  - a vertex shader that doesn't modify the incoming vertex positions
  - a fragment shader that colors pixels white
- That doesn’t look like a square!
- What went wrong?



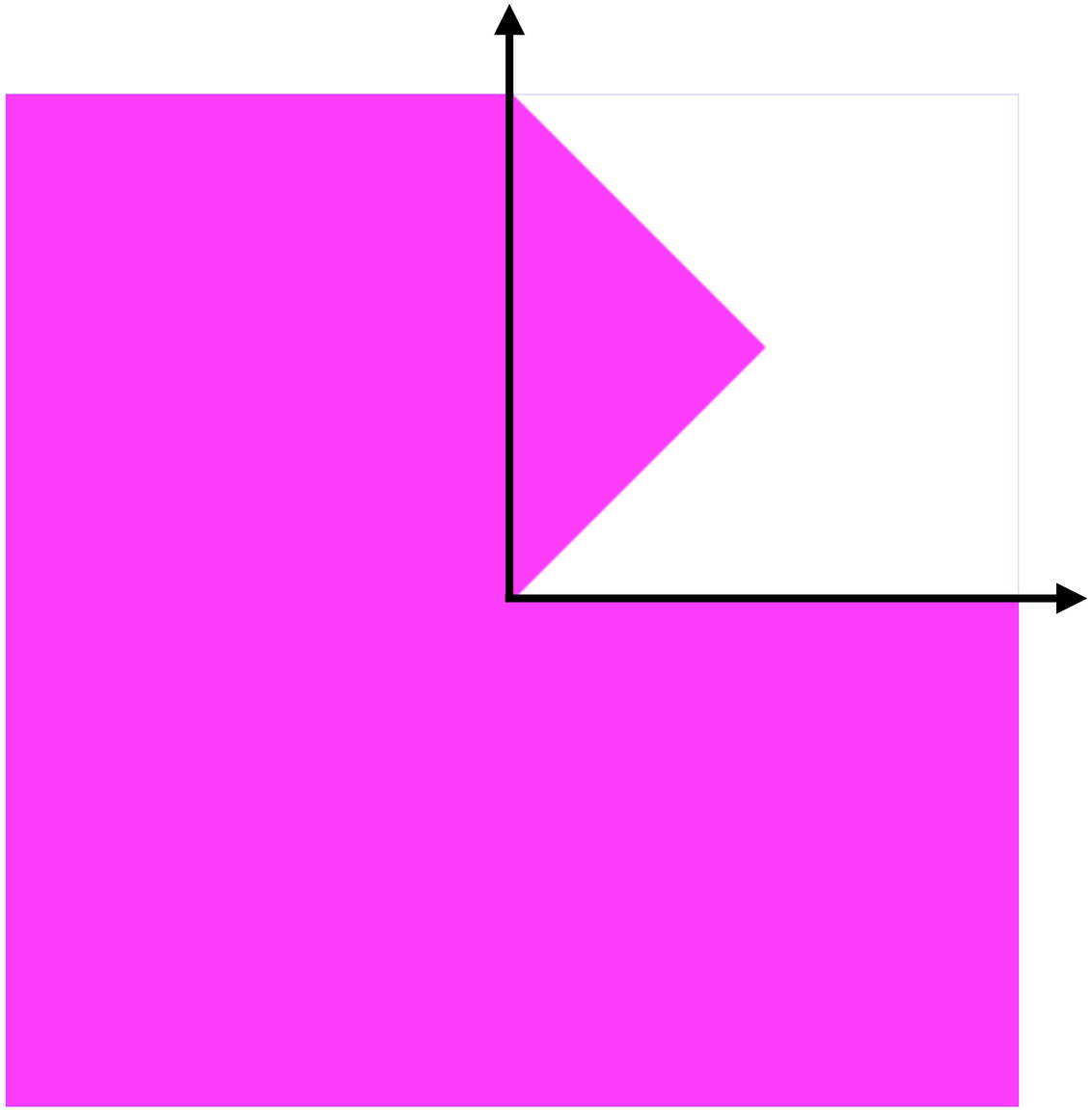
Quick Aside: Where Are We Drawing?

- Computer graphics has numerous coordinate spaces
  - 3D ones for working with 3D objects
  - 2D ones for working with pixels
  - 3D ones to help "normalize" math
    - we'll see examples where we solve a problem by separating it into two pieces:
      - a single specific solution (usually involving getting things into a specific orientation)
      - find a way to take an arbitrary orientation, and get it into the specific one



Normalized Device Coordinates (NDC)

- With no other processing in a vertex shader, vertices are expected to be in *normalized device coordinates*
- It's a simple 3D space to make follow-on computations simpler
  - $x, y, z \in$ [-1.0, 1.0]



Connected Primitives

Rendering Triangle Strips

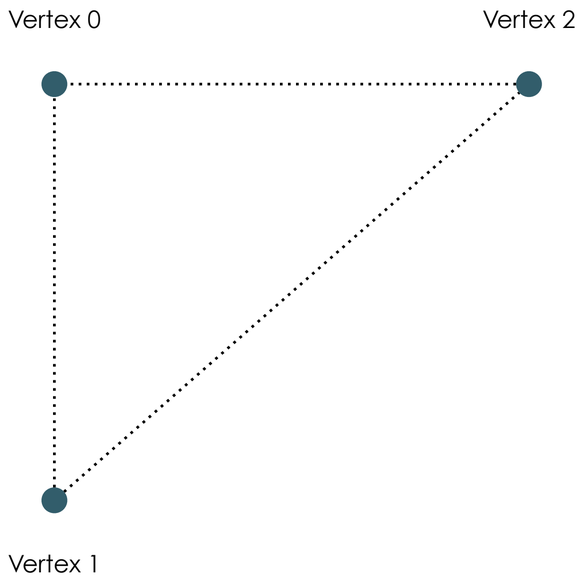
- Triangle strips process vertices in a special way

Rendering Triangle Strips

- Triangle strips process vertices in a special way

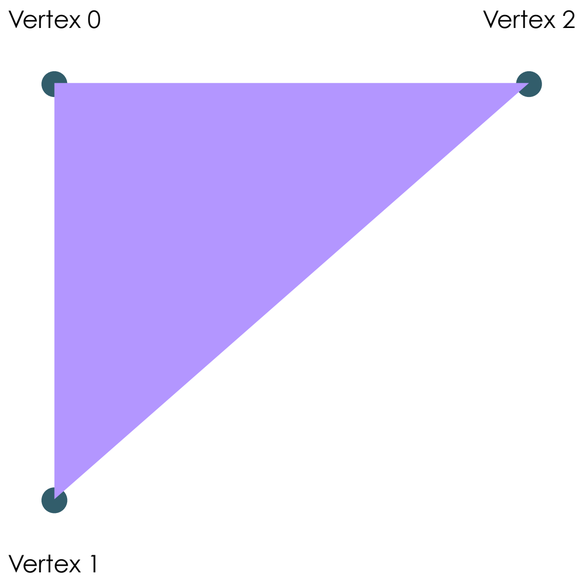
Rendering Triangle Strips

- Triangle strips process vertices in a special way



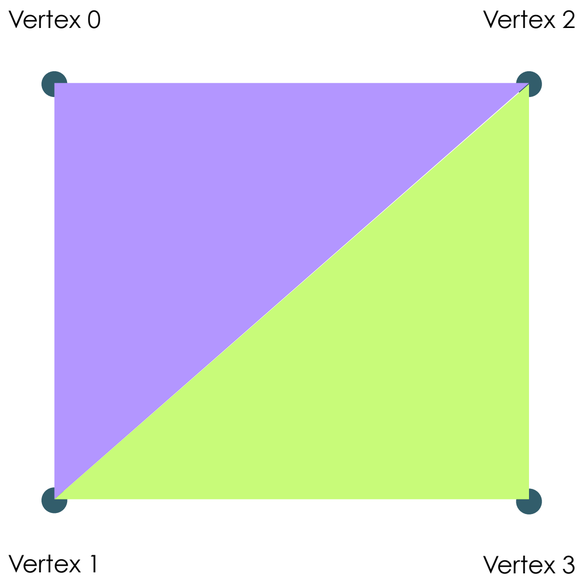
Rendering Triangle Strips

- Triangle strips process vertices in a special way



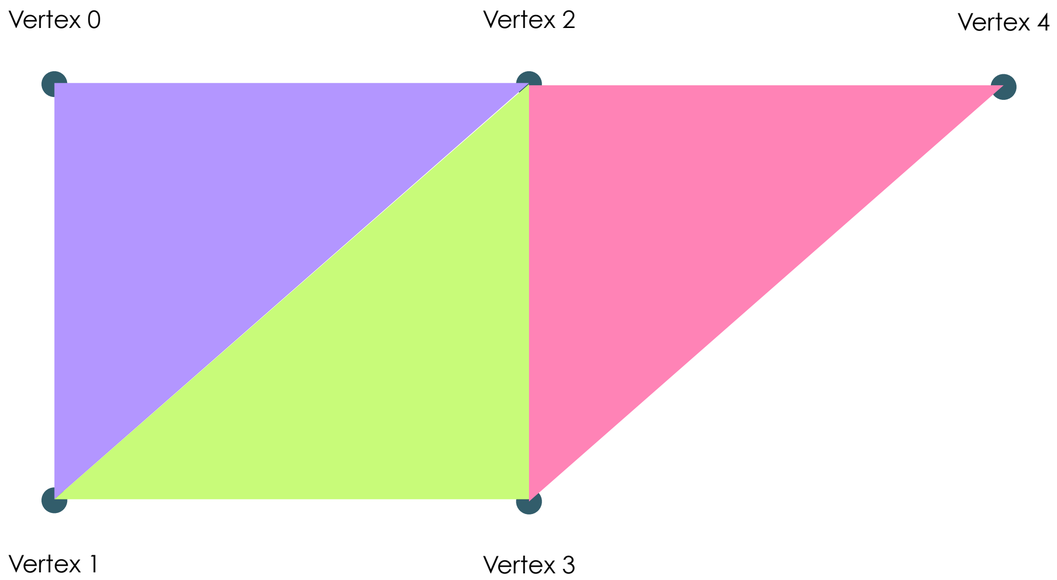
Rendering Triangle Strips

- Triangle strips process vertices in a special way



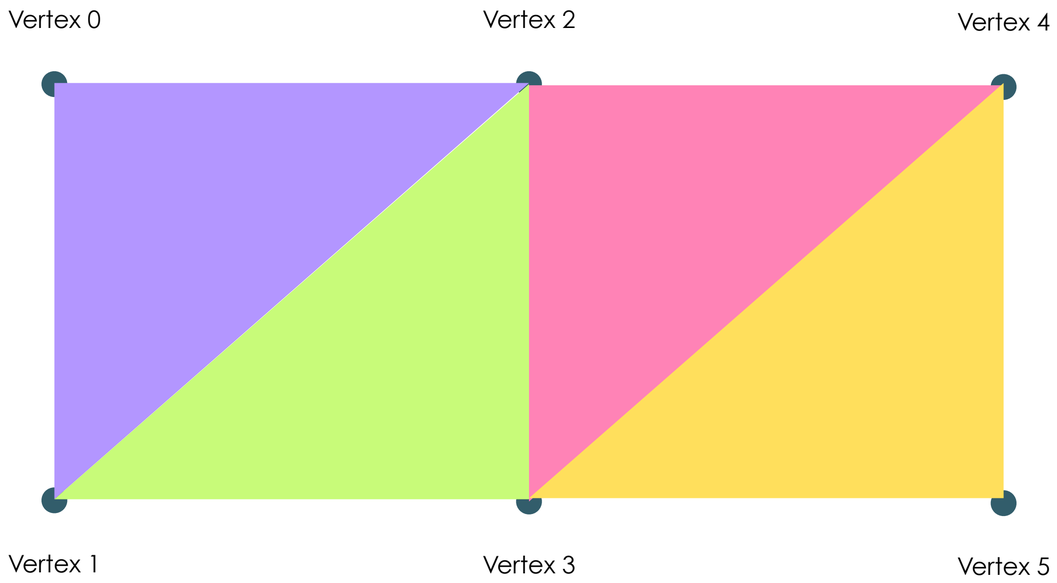
Rendering Triangle Strips

- Triangle strips process vertices in a special way



Rendering Triangle Strips

- Triangle strips process vertices in a special way



Vertex Ordering

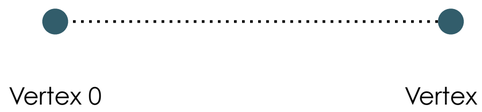
- The problem with our square is that the vertices are issued in the wrong order
  - we can’t go around the perimeter

Vertex 0

```
var Square = {
 count : 4,
 positions : {
  values : new Float32Array([
    0.0, 0.0,  // Vertex 0
    1.0, 0.0,  // Vertex 1
    1.0, 1.0,  // Vertex 2
    0.0, 1.0   // Vertex 3
   ]),
   numComponents : 2
  },
  colors : {
   values : new Float32Array([ ... ]),
   numComponets : 3
  }
};
```

Vertex Ordering

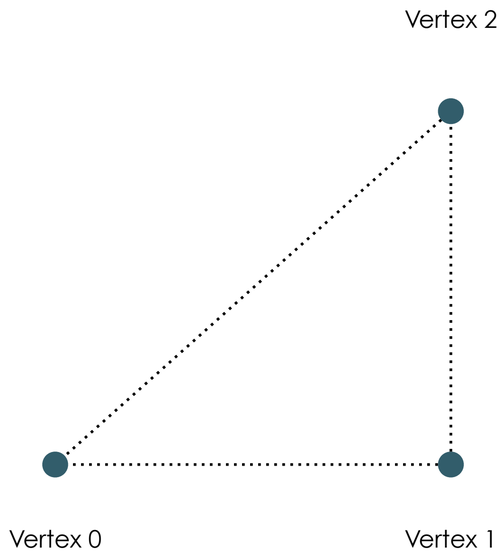
- The problem with our square is that the vertices are issued in the wrong order
  - we can’t go around the perimeter




```
var Square = {
 count : 4,
 positions : {
  values : new Float32Array([
    0.0, 0.0,  // Vertex 0
    1.0, 0.0,  // Vertex 1
    1.0, 1.0,  // Vertex 2
    0.0, 1.0   // Vertex 3
   ]),
   numComponents : 2
  },
  colors : {
   values : new Float32Array([ ... ]),
   numComponets : 3
  }
};
```

Vertex Ordering

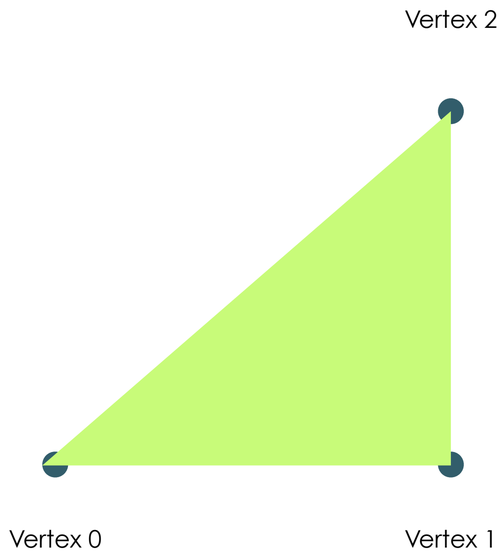
- The problem with our square is that the vertices are issued in the wrong order
  - we can’t go around the perimeter




```
var Square = {
 count : 4,
 positions : {
  values : new Float32Array([
    0.0, 0.0,  // Vertex 0
    1.0, 0.0,  // Vertex 1
    1.0, 1.0,  // Vertex 2
    0.0, 1.0   // Vertex 3
   ]),
   numComponents : 2
  },
  colors : {
   values : new Float32Array([ ... ]),
   numComponets : 3
  }
};
```

Vertex Ordering

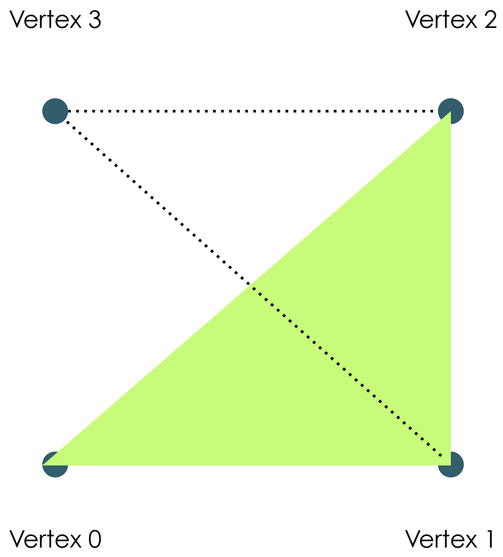
- The problem with our square is that the vertices are issued in the wrong order
  - we can’t go around the perimeter




```
var Square = {
 count : 4,
 positions : {
  values : new Float32Array([
    0.0, 0.0,  // Vertex 0
    1.0, 0.0,  // Vertex 1
    1.0, 1.0,  // Vertex 2
    0.0, 1.0   // Vertex 3
   ]),
   numComponents : 2
  },
  colors : {
   values : new Float32Array([ ... ]),
   numComponets : 3
  }
};
```

Vertex Ordering

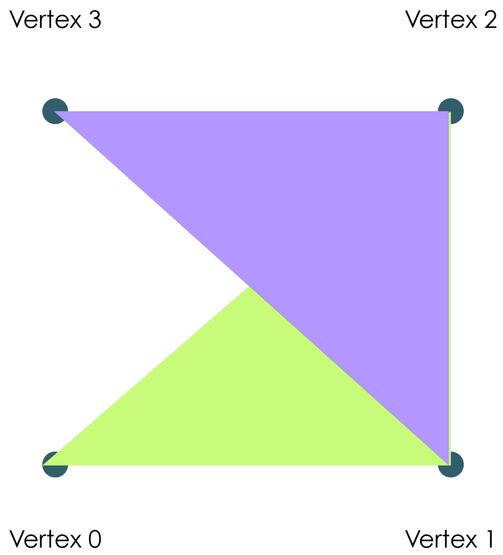
- The problem with our square is that the vertices are issued in the wrong order
  - we can’t go around the perimeter




```
var Square = {
 count : 4,
 positions : {
  values : new Float32Array([
    0.0, 0.0,  // Vertex 0
    1.0, 0.0,  // Vertex 1
    1.0, 1.0,  // Vertex 2
    0.0, 1.0   // Vertex 3
   ]),
   numComponents : 2
  },
  colors : {
   values : new Float32Array([ ... ]),
   numComponets : 3
  }
};
```

Vertex Ordering

- The problem with our square is that the vertices are issued in the wrong order
  - we can’t go around the perimeter




```
var Square = {
 count : 4,
 positions : {
  values : new Float32Array([
    0.0, 0.0,  // Vertex 0
    1.0, 0.0,  // Vertex 1
    1.0, 1.0,  // Vertex 2
    0.0, 1.0   // Vertex 3
   ]),
   numComponents : 2
  },
  colors : {
   values : new Float32Array([ ... ]),
   numComponets : 3
  }
};
```

Vertex Ordering

- The order vertices are sent into WebGL affects construction of primitives
  - here, we need to swap vertices 2 & 3

Vertex 0

```
var Square = {
 count : 4,
 positions : {
  values : new Float32Array([
    0.0, 0.0,  // Vertex 0
    1.0, 0.0,  // Vertex 1
    0.0, 1.0,  // Vertex 3
    1.0, 1.0   // Vertex 2
   ]),
   numComponents : 2
  },
  colors : {
   values : new Float32Array([ ... ]),
   numComponets : 3
  }
};
```

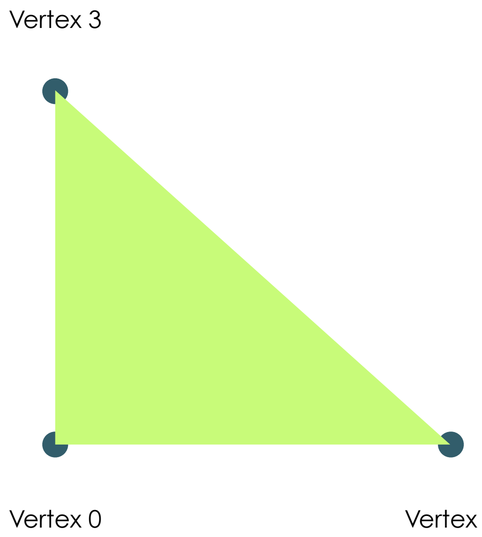
Vertex Ordering

- The order vertices are sent into WebGL affects construction of primitives
  - here, we need to swap vertices 2 & 3


```
var Square = {
 count : 4,
 positions : {
  values : new Float32Array([
    0.0, 0.0,  // Vertex 0
    1.0, 0.0,  // Vertex 1
    0.0, 1.0,  // Vertex 3
    1.0, 1.0   // Vertex 2
   ]),
   numComponents : 2
  },
  colors : {
   values : new Float32Array([ ... ]),
   numComponets : 3
  }
};
```

Vertex Ordering

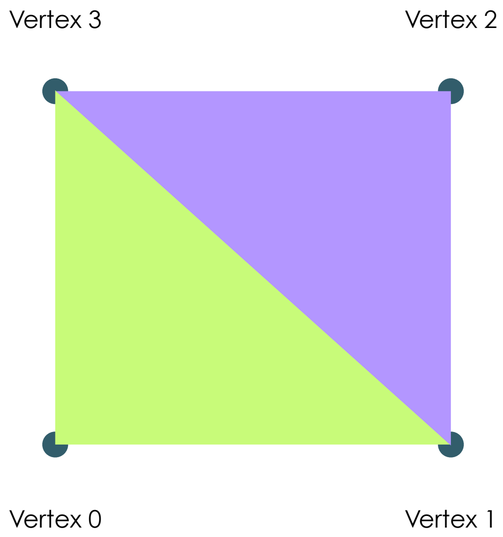
- The order vertices are sent into WebGL affects construction of primitives
  - here, we need to swap vertices 2 & 3




```
var Square = {
 count : 4,
 positions : {
  values : new Float32Array([
    0.0, 0.0,  // Vertex 0
    1.0, 0.0,  // Vertex 1
    0.0, 1.0,  // Vertex 3
    1.0, 1.0   // Vertex 2
   ]),
   numComponents : 2
  },
  colors : {
   values : new Float32Array([ ... ]),
   numComponets : 3
  }
};
```

Vertex Ordering

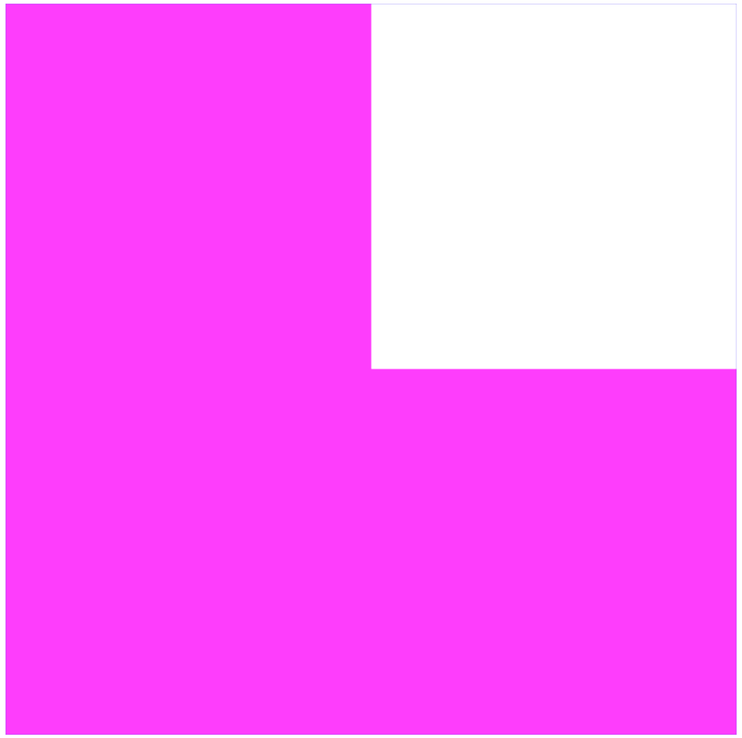
- The order vertices are sent into WebGL affects construction of primitives
  - here, we need to swap vertices 2 & 3




```
var Square = {
 count : 4,
 positions : {
  values : new Float32Array([
    0.0, 0.0,  // Vertex 0
    1.0, 0.0,  // Vertex 1
    0.0, 1.0,  // Vertex 3
    1.0, 1.0   // Vertex 2
   ]),
   numComponents : 2
  },
  colors : {
   values : new Float32Array([ ... ]),
   numComponets : 3
  }
};
```

Vertex Ordering

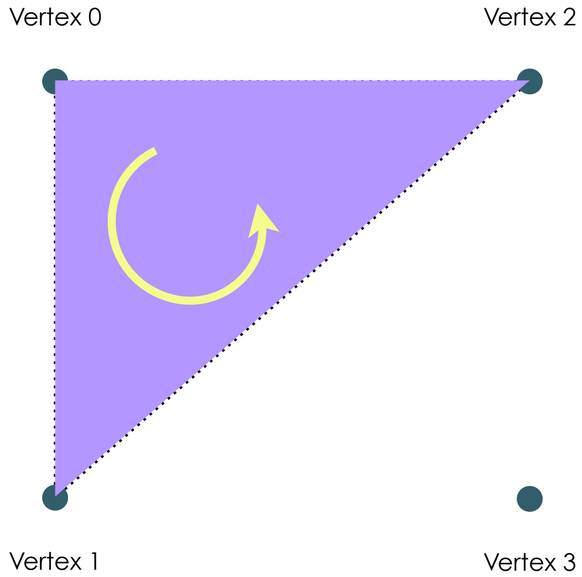
- Proof!



```
var Square = {
 count : 4,
 positions : {
  values : new Float32Array([
    0.0, 0.0,  // Vertex 0
    1.0, 0.0,  // Vertex 1
    0.0, 1.0,  // Vertex 3
    1.0, 1.0   // Vertex 2
   ]),
   numComponents : 2
  },
  colors : {
   values : new Float32Array([ ... ]),
   numComponets : 3
  }
};
```

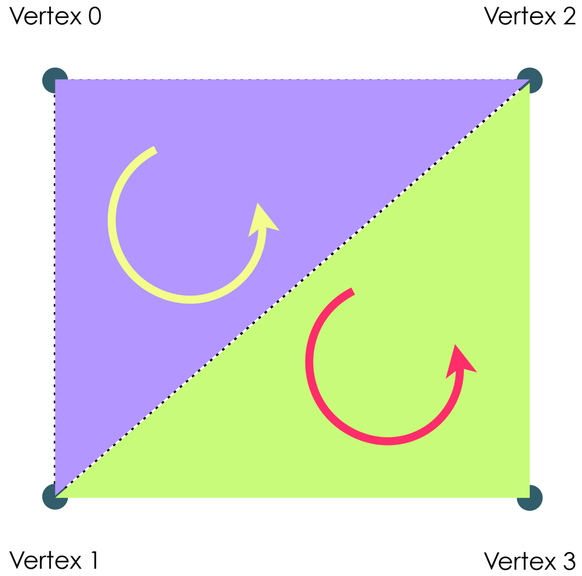
More on Vertex Ordering

- The issue order of vertices also affects the *facedness* of a triangle
  - triangles can have *front* and *back* faces
  - which face you're looking at is controlled by the *screen-space projection* of the primitive
- Model a primitive such that the vertices are issued *counter-clockwise* when you're looking at the front face



More on Vertex Ordering

- Vertex strips (and fans) set all triangles to have the same facedness as the first one



Indexed Rendering

Sending Vertices to WebGL

- Last class we saw `gl.drawArrays()` which sends a sequential set of vertices for rendering

```
gl.drawArrays(primType, start, count);
```

```
gl.drawArrays(gl.TRIANGLES, 2, 3);
```

Sending Vertices to WebGL

- Last class we saw `gl.drawArrays()` which sends a sequential set of vertices for rendering

```
gl.drawArrays(primType, start, count);
```

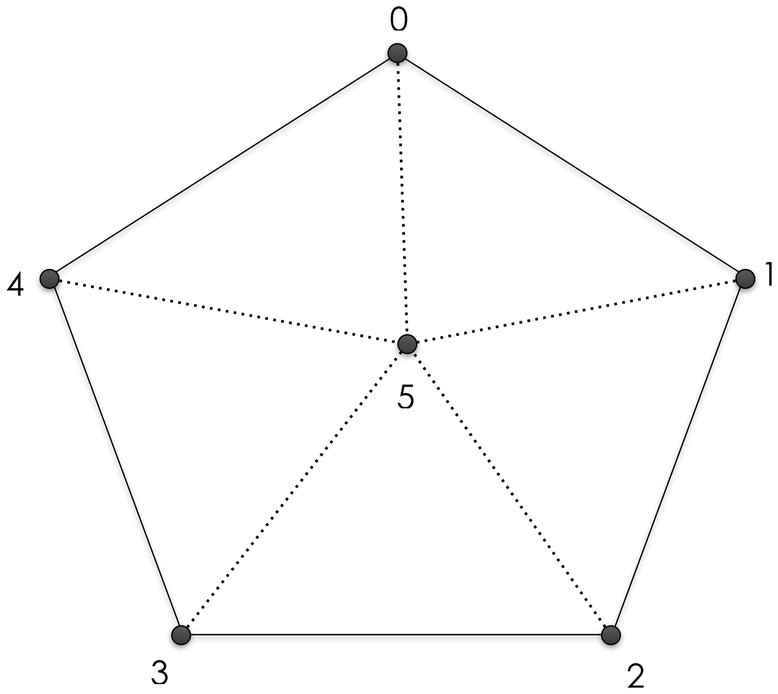
```
gl.drawArrays(gl.TRIANGLES, 2, 3);
```

Sending Vertices to WebGL

- Often multiple primitives in a model will share a vertex
- For this example, you'd need to construct a vertex buffer with vertices

```
{ 0, 5, 1, 1, 5, 2, 2, 5, 3, 3, 5, 4, 4, 5, 0 }
```

- There are really only six *unique* vertices in the model
  - we can draw in another mode that uses a list of vertices, and a list of indices into that list
  - `gl.drawElements()` does just that



Adding Indices to our Shape

- Recall our square rendering required updating the order of our vertices
- We could have used indexed rendering to fix that problem

```
function Square () {
 this.count = 4;
 this.positions = {
  values = new Float32Array([
    0.0, 0.0,  // Vertex 0
    1.0, 0.0,  // Vertex 1
    1.0, 1.0,  // Vertex 2
    0.0, 1.0   // Vertex 3
   ]),
   numComponents = 2
  };
  this.colors = { ... };
  this.indices = {
   values = new Uint16Array([ 0, 1, 3, 2 ])
  };
  ...
}
```

Indexed Rendering

- In order to use `gl.drawElements()`, we need to set up another buffer that contains the indices
- You create a buffer just like you did previously, except you use a different type — `gl.ELEMENT_ARRAY_BUFFER`

```
this.indices.buffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, this.indices.buffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, this.indices.values, gl.STATIC_DRAW);
```

Binding a Buffer

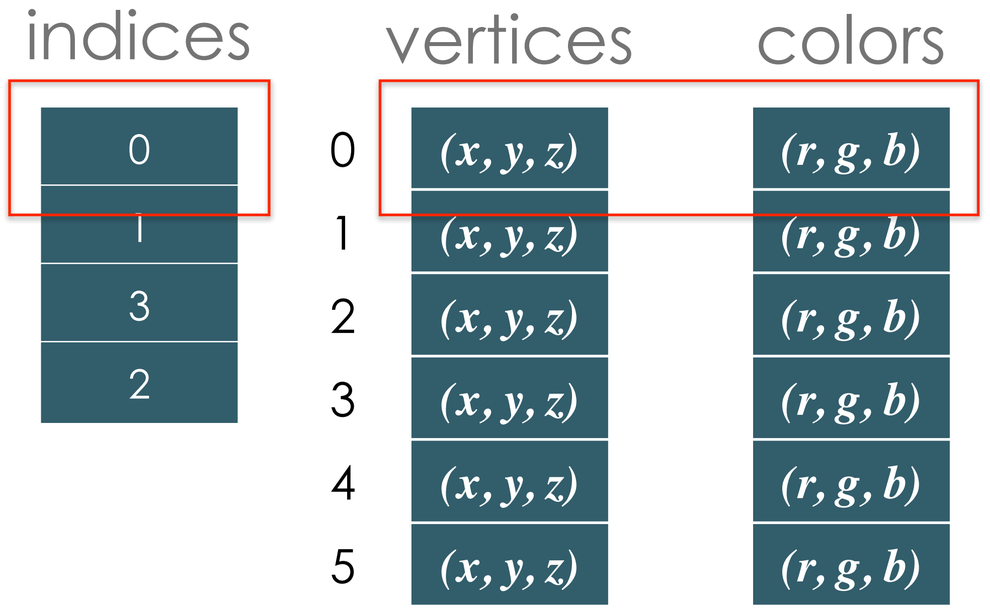
- Before you can use your element buffer, you need to bind it
  - just like you did for your vertex data

```
function Square () {
    ...

    this.render = function () {
        ...
        gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, this.indices.buffer);
    };
}
```

Indexed Rendering

- When you issue a `gl.drawElements()`, the index list is traversed sequentially
  - the vertices are dispatched using the index order
  - there's a type parameter that needs to always be `gl.UNSIGNED_SHORT`

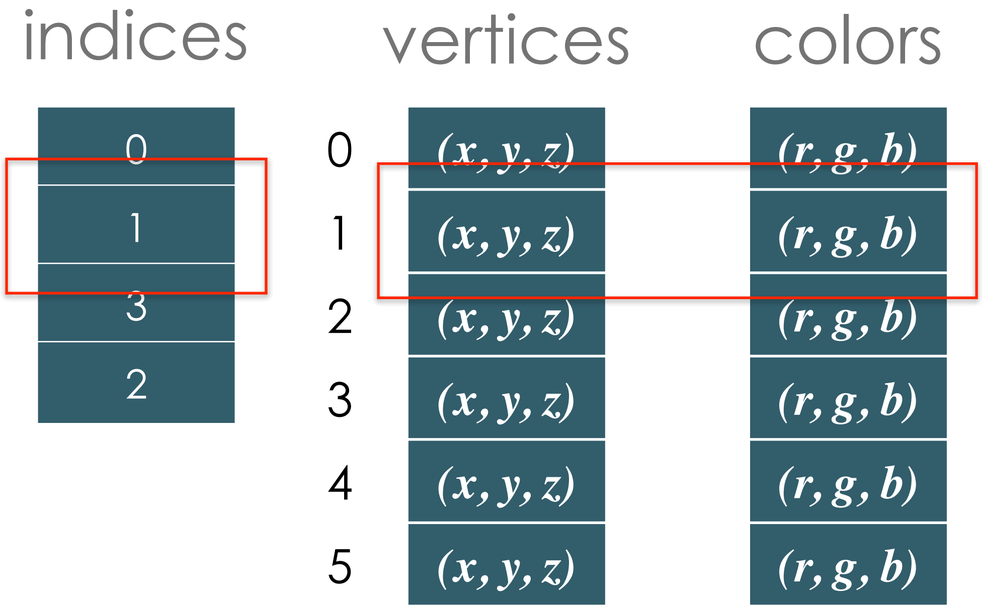
```
gl.drawElements(gl.TRIANGLE_STRIP,
  this.indices.values.length,
  gl.UNSIGNED_SHORT, 0);
```



Indexed Rendering

- When you issue a `gl.drawElements()`, the index list is traversed sequentially
  - the vertices are dispatched using the index order
  - there's a type parameter that needs to always be `gl.UNSIGNED_SHORT`

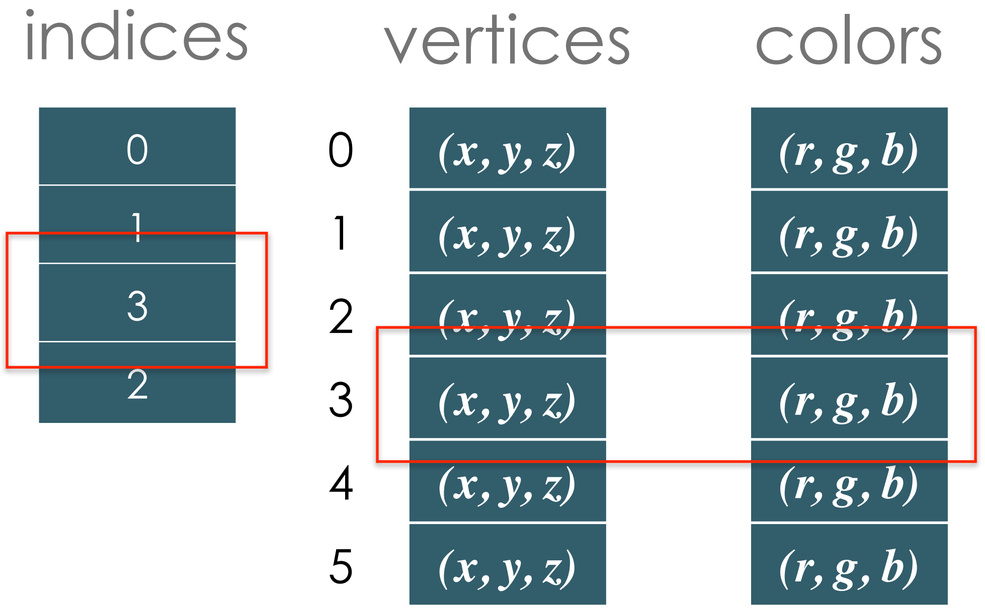
```
gl.drawElements(gl.TRIANGLE_STRIP,
  this.indices.values.length,
  gl.UNSIGNED_SHORT, 0);
```



Indexed Rendering

- When you issue a `gl.drawElements()`, the index list is traversed sequentially
  - the vertices are dispatched using the index order
  - there's a type parameter that needs to always be `gl.UNSIGNED_SHORT`

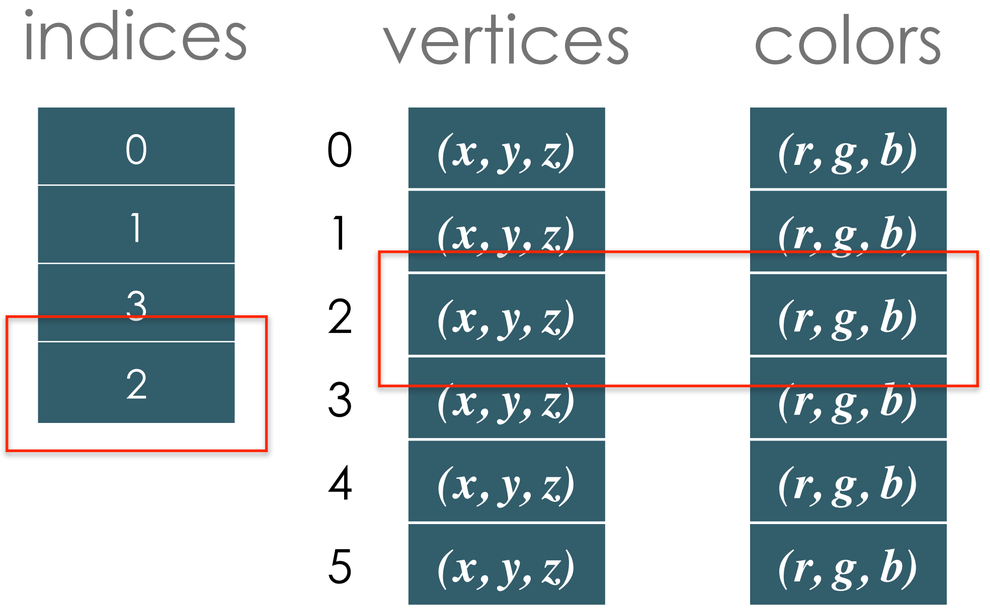
```
gl.drawElements(gl.TRIANGLE_STRIP,
  this.indices.values.length,
  gl.UNSIGNED_SHORT, 0);
```



Indexed Rendering

- When you issue a `gl.drawElements()`, the index list is traversed sequentially
  - the vertices are dispatched using the index order
  - there's a type parameter that needs to always be `gl.UNSIGNED_SHORT`

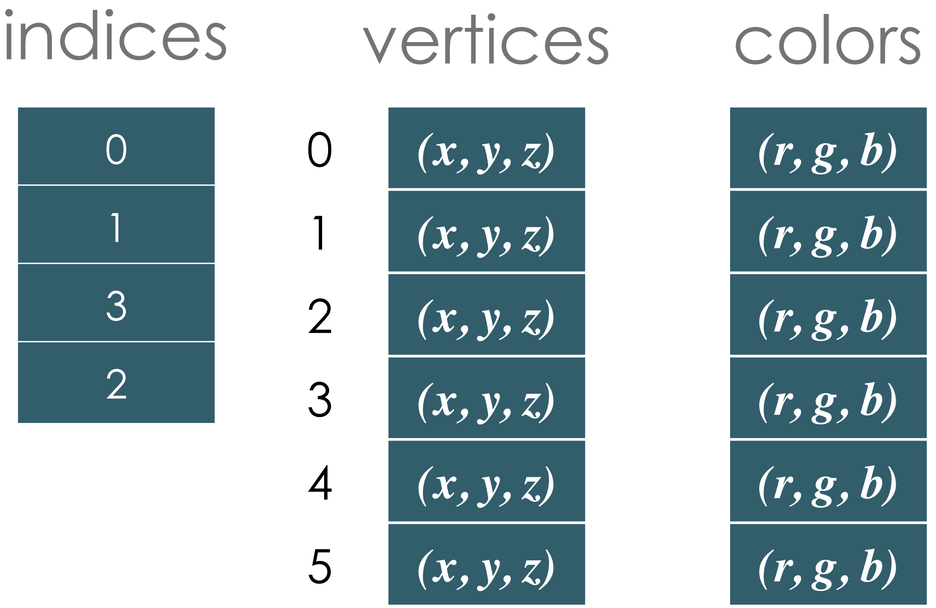
```
gl.drawElements(gl.TRIANGLE_STRIP,
  this.indices.values.length,
  gl.UNSIGNED_SHORT, 0);
```



Indexed Rendering

- When you issue a `gl.drawElements()`, the index list is traversed sequentially
  - the vertices are dispatched using the index order
  - there's a type parameter that needs to always be `gl.UNSIGNED_SHORT`

```
gl.drawElements(gl.TRIANGLE_STRIP,
  this.indices.values.length,
  gl.UNSIGNED_SHORT, 0);
```



Rendering Tidbits

Rendering Different Primitives

- The first parameter of `gl.drawArrays` defines the primitive type
- A good debugging step is to render `gl.POINTS`
- Particularly useful if you're using indexed rendering
  - changing from `gl.drawElements` to `gl.drawArrays` can verify that vertex data is correct

```
function Square() {
    this.count = 4;

    this.render = function () {
        ... // bind buffers

        var start = 0;
        var count = this.count;
        gl.drawArrays(gl.POINTS,
            start, count);
     };
};
```

Pixels can be small

- Control the size of `gl.POINTS`
- Set the point size in a vertex shader
  - assign a value to `gl_PointSize`

```
in vec4 aPosition;

void main()
{
    gl_PointSize = 5.0;
    gl_Position = aPosition;
 }
```

Buffers, Binding, and Encapsulation

Buffer Setup

- When you have multiple objects, you need to specify the plumbing immediately before rendering
- Recall these calls

```
this.positions.buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, this.positions.buffer);
gl.bufferData(gl.ARRAY_BUFFER, this.positions.values, gl.STATIC_DRAW);
this.positions.attributeLoc = gl.getAttribLocation(this.program, “vPosition”);
gl.vertexAttribPointer(this.positions.attributeLoc, this.positions.numComponents,
     gl.FLOAT, gl.FALSE, 0, 0);
gl.enableVertexAttribArray(this.positions.attributeLoc);
```

Buffer Initialization

- Do these calls when you initialize the vertex buffer when creating the JavaScript object

```
this.positions.buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, this.positions.buffer);
gl.bufferData(gl.ARRAY_BUFFER, this.positions.values, gl.STATIC_DRAW);
this.positions.attributeLoc = gl.getAttribLocation(this.program, “vPosition”);
gl.vertexAttribPointer(this.positions.attributeLoc, this.positions.numComponents,
     gl.FLOAT, gl.FALSE, 0, 0);
gl.enableVertexAttribArray(this.positions.attributeLoc);
```

Rendering Initialization

- Do these calls right before drawing

```
this.positions.buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, this.positions.buffer);
gl.bufferData(gl.ARRAY_BUFFER, this.positions.values, gl.STATIC_DRAW);
this.positions.attributeLoc = gl.getAttribLocation(this.program, “vPosition”);
gl.vertexAttribPointer(this.positions.attributeLoc, this.positions.numComponents,
     gl.FLOAT, gl.FALSE, 0, 0);
gl.enableVertexAttribArray(this.positions.attributeLoc);
```

A Note about Binding Buffers

- Why do we need these `gl.bindBuffer()` calls?
- Draw calls use data from the currently bound buffers
  - `gl.drawArrays()` only uses the currently bound `gl.ARRAY_BUFFER`
  - `gl.drawElements()` uses both the current `gl.ARRAY_BUFFER` and the `gl.ELEMENT_ARRAY_BUFFER`
- For every object, we'll create the appropriate buffers, and bind them when it's rendering time

Initializing an Object

- Recall we'll create a JavaScript object
- This is done by specifying a *function* that initializes the JavaScript object
- First; declare and initialize your vertex attributes and (potentially) index buffers
- Then build the *shader program* from the shaders
  - use our `initShaders()` function from **initShaders.js** file

```
function Square () {
  this.count = 4;
  this.positions = { ... };
  this.colors = { ... };
  this.indices = { ... };

  this.program = initShaders(gl, “vertex-shader”,
      “fragment-shader”);
 };
```

Initializing an Object

- Next, create, bind, and initialize buffers

```
function Square () {
  this.count = 4;
  this.positions = { ... };
  this.colors = { ... };
  this.indices = { ... };

  this.program = initShaders(gl, “vertex-shader”,
      “fragment-shader”);
  this.positions.buffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, this.positions.buffer);
  gl.bufferData(gl.ARRAY_BUFFER, this.positions.values,
    gl.STATIC_DRAW);

 };
```

Initializing an Object

- Find the *vertex attribute* (from the vertex shader) associated with this data buffer
- and enable the associated array

```
function Square () {
  this.count = 4;
  this.positions = { ... };
  this.colors = { ... };
  this.indices = { ... };

  this.program = initShaders(gl, “vertex-shader”,
      “fragment-shader”);
  this.positions.buffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, this.positions.buffer);
  gl.bufferData(gl.ARRAY_BUFFER, this.positions.values,
    gl.STATIC_DRAW);
  this.positions.attributeLoc = gl.getAttribLocation(
     this.program, “vPosition”);
  gl.enableVertexAttribArray(this.positions.attributeLoc);
 };
```

Initializing an Object

- Do this same sequence for each vertex attribute you're using

```
function Square () {
  this.count = 4;
  this.positions = { ... };
  this.colors = { ... };
  this.indices = { ... };

  this.program = initShaders(gl, “vertex-shader”,
      “fragment-shader”);
  ... // positions vertex buffer setup
  this.colors.buffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, this.colors.buffer);
  gl.bufferData(gl.ARRAY_BUFFER, this.colors.values,
    gl.STATIC_DRAW);
 this.colors.attributeLoc = gl.getAttribLocation(
     this.program, “vColor”);
  gl.enableVertexAttribArray(this.colors.attributeLoc);

 };
```

Initializing an Object

- Initialize your index array if your using *indexed rendering*

```
function Square () {
  this.count = 4;
  this.positions = { ... };
  this.colors = { ... };
  this.indices = { ... };

  this.program = initShaders(gl, “vertex-shader”,
      “fragment-shader”);
  ...
  this.colors.buffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, this.colors.buffer);
  gl.bufferData(gl.ARRAY_BUFFER, this.colors.values,
    gl.STATIC_DRAW);
  this.colors.attributeLoc = gl.getAttribLocation(
     this.program, “vColor”);
  gl.enableVertexAttribArray(this.colors.attributeLoc);
  this.indices.buffer = gl.createBuffer();
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER,
     this.indices.buffer);
  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
    this.indices.values, gl.STATIC_DRAW);
 };
```

Drawing an Object

- Just like initializing an object, let's create a method for drawing
  - let's call it ... `render()` 😀

```
function Square () {
  this.count = 4;
  this.positions = { ... };
  this.colors = { ... };
  this.indices = { ... };

  this.program = initShaders(gl, “vertex-shader”,
      “fragment-shader”);
  ...
  this.render = function () {
     ...
  }
 };
```

Specifying the Shader Program

- First, specify which shader program you'll use to render this object
  - for our simple examples, we'll usually only have a single program
  - however, more complex objects may have more than one shader program that references the vertex data

```
function Square () {
  this.count = 4;
  this.positions = { ... };
  this.colors = { ... };
  this.indices = { ... };

  this.program = initShaders(gl, “vertex-shader”,
      “fragment-shader”);
  ...
  this.render = function () {
     gl.useProgram(this.program);
  }
 };
```

Drawing an Object

- In `render()`, for each buffers we created:
  1. bind the buffer
  2. associate the attribute variable with that buffer

```
function Square () {
  this.count = 4;
  this.positions = { ... };
  this.colors = { ... };
  this.indices = { ... };

  this.program = initShaders(gl, “vertex-shader”,
      “fragment-shader”);
  ...
  this.render = function () {
    gl.useProgram(this.program);
    gl.bindBuffer(gl.ARRAY_BUFFER, this.positions.buffer);
    gl.vertexAttribPointer(this.positions.attributeLoc,
     this.positions.numComponents, gl.FLOAT, gl.FALSE, 0, 0);

    gl.bindBuffer(gl.ARRAY_BUFFER, this.colors.buffer);
    gl.vertexAttribPointer(this.colors.attributeLoc,
     this.colors.numComponents, gl.FLOAT, gl.FALSE, 0, 0);
  };
 };
```

Drawing an Object

- In `render()`, then bind our element array if we're doing indexed rendering

```
function Square () {
  this.count = 4;
  this.positions = { ... };
  this.colors = { ... };
  this.indices = { ... };

  this.program = initShaders(gl, “vertex-shader”,
      “fragment-shader”);
  ...
  this.render = function () {
    gl.useProgram(this.program);
    gl.bindBuffer(gl.ARRAY_BUFFER, this.positions.buffer);
    gl.vertexAttribPointer(this.positions.attributeLoc,
     this.positions.numComponents, gl.FLOAT, gl.FALSE, 0, 0);

    gl.bindBuffer(gl.ARRAY_BUFFER, this.colors.buffer);
    gl.vertexAttribPointer(this.colors.attributeLoc,
     this.colors.numComponents, gl.FLOAT, gl.FALSE, 0, 0);

   gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, this.indices.buffer);
  };
 };
```

Drawing an Object

- And finally, draw
  - actually, you'll probably have more than one draw call for more complex objects

```
function Square () {
  this.count = 4;
  this.positions = { ... };
  this.colors = { ... };
  this.indices = { ... };

  this.program = initShaders(gl, “vertex-shader”,
      “fragment-shader”);
  ...
  this.render = function () {
    ...

   gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, this.indices.buffer);
   gl.drawElements(gl.TRIANGLES, this.indices.values.length,
    gl.UNSIGNED_SHORT, 0);
    };
 };
```