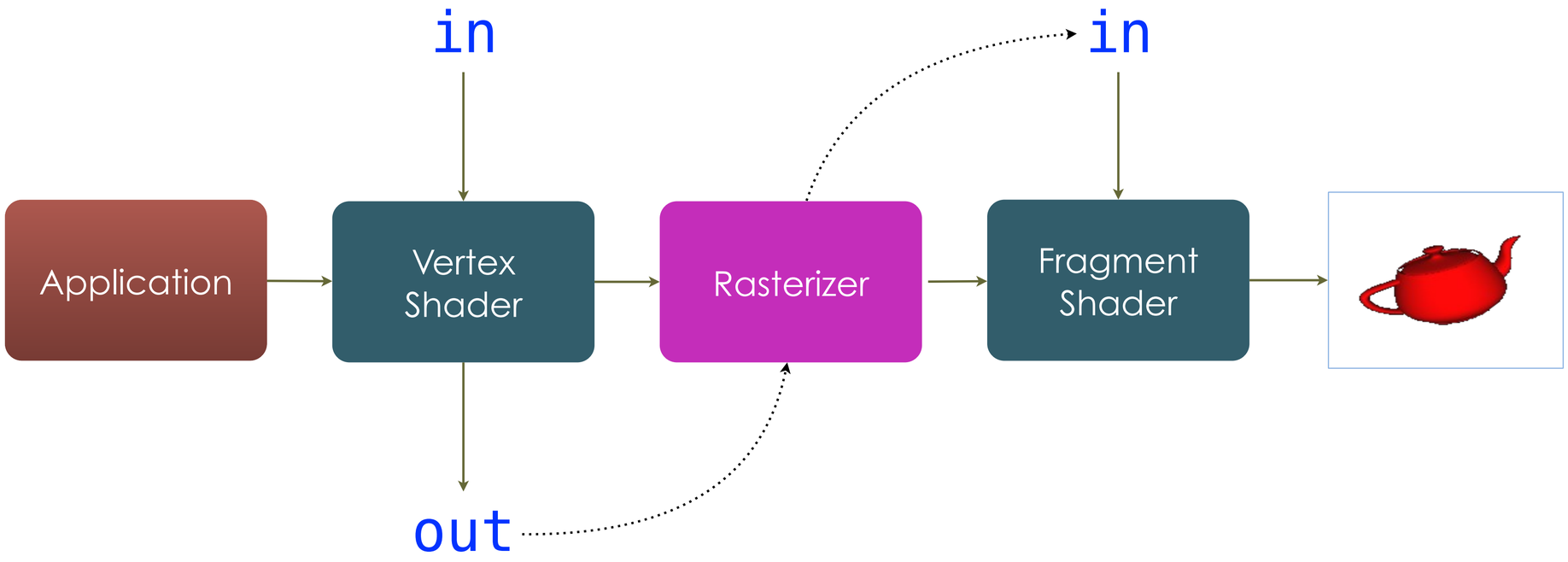
Shader Programs and Geometry

CS 385 - Class 3
1 February 2022

Shader Programs

Shader Keywords



In WebGL 1.0, `in`s were labeled `attribute`, `out`s were `varyings`

Vertex Shaders and HTML

- For HTML, a vertex shader is an additional type of script
- Just "wrap" your shader code in a pair of `<script>` tags
- Name the shader with its *id* attribute
  - we'll use this name later to load the shader
- Specify its type using the *type* attribute
  - use `x-shader/x-vertex` for vertex shaders

```
<script id="vertex-shader"
    type="x-shader/x-vertex">
in  vec4 aPosition;
out vec4 vColor;

void main()
{
  vColor = vec4(0.0, 0.0, 1.0, 1.0);
  gl_Position = aPosition;
}
</script>
```

Fragment Shaders and HTML

- Virtually the same idea as declaring vertex shaders
- Specify its type using the *type* attribute as `x-shader/x-fragment`

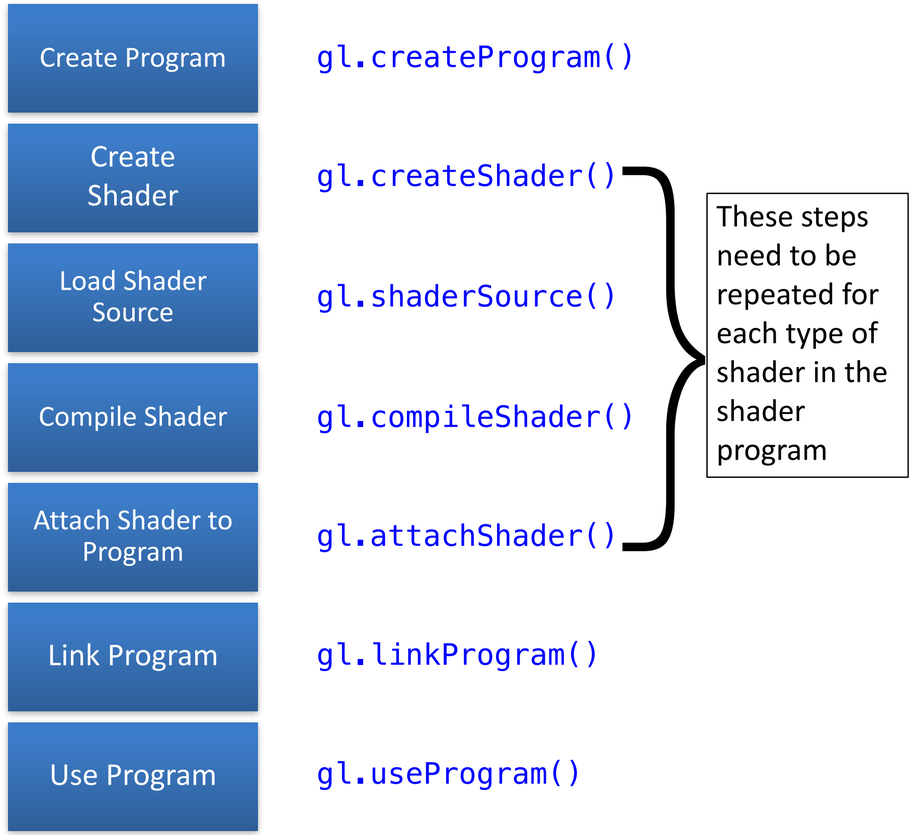
```
<script id="fragment-shader"
    type="x-shader/x-fragment">
precision highp float;

in   vec4 vColor;
out  vec4 fColor;

void main()
{
  fColor = vColor;
}
</script>
```

Shader Programs

- In WebGL, a *shader program* is a compiled collection of shaders
- A *shader* is a (potentially complete) WebGL SL function
- A *program* is a collection of shaders linked together
- We'll write shaders, but we'll use programs

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 program must contain avertex and fragment shader



That’s a lot of work

- We have a helper JavaScript function `initShaders()` to help
  - provided in the `initShaders.js`
- It does all the nastiness shown in the previous slide
- It takes the *id* names of the vertex and fragment shaders
- After compiling the shaders into a program, we'll use it to control rendering
  - we need to use the program
  - call `gl.useProgram()`
- We'll often encapsulate our program inside of a JavaScript object

```
var program = initShaders(
    gl,                   // our WebGL context
    "vertex-shader",      // vertex shader id
    "fragment-shader"); // fragment shader id


gl.useProgram(program);
```

Coordinate Systems and Basic Modeling

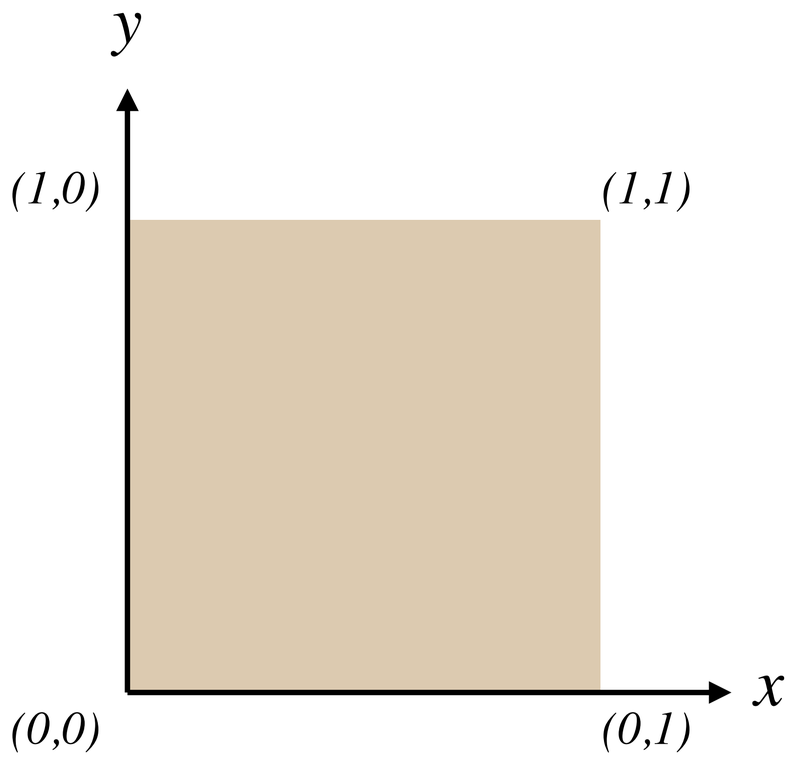
Vertices

- Recall from last week that a *vertex* is a collection of attributes for a point in space
- Every vertex must have a position attribute
- WebGL internally represents positions as *homogenous coordinates*
- *We* can use 2D or 3D coordinates when specifying positions
  - all values for an attribute must be the same type for all vertices in the model
  - for example, all vertex positions for a model must either be 2D or 3D; you can't mix them in the same model

Coordinate	Default Value
x	0.0
y	0.0
z	0.0
w	1.0

Modeling a Unit Square

- Unit square has:
  - one vertex at the origin
  - side lengths of 1.0 units
- Squares are planar, so we can use 2D coordinates



Model Objects

- We're going to store our geometric models's data in JavaScript objects
- A JavaScript object is like a structure or dictionary in other languages
  - it has *properties* which are referenced by name, and which have a value
    - values can be of any type — scalars, arrays, other objects, etc. — as we'll see
  - each property-value pair is followed by a comma
    - except for the last one
- *Everything* in JavaScript is an object

```
var Object = {
    property1 : value1,
    property2 : value2,
    ...
};
```

Creating Objects

- In addition to declaring an object, an function can be used to create an object
  - think of it as a constructor
- The function implicitly creates an object which you can add properties to through the `this` construct
- This include defining methods attached to the object
  - we'll use this to create a render method when we construct our class

```
function Object( params ) {
    this.property1 = value1;
    this.property2 = value2;
    this.methodName = function () {
        ...
    }
};
```

Our Square Object

- Some of our object properties will be for vertex attributes
- We'll store each vertex attribute's data in an object
- Suppose we have both *positions* and *colors* for each vertex in our square
  - we'll define a JavaScript object for each attribute
  - populate specific fields for each attribute

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

Vertex Count Property

- It will also be useful to know how many vertices the object has
- For consistency's sake, name this *count*
- Every attribute we add to the vertex needs to have *count* entries

```
function Square() {
    this.count = 4; // because a square has four corners

    this.positions = { ... };
    this.colors = { ... };
};
```

Our Square's Positions

- Initialize our square's positions
  - we can use 2D positions (i.e., $z = 0$)
- Store the vertex values in a property named *values*
- Vertex attributes need to be stored in a special type of buffer
  - `Float32Array`
  - initialized using a standard JavaScript array
- No grouping of values is required
  - just a simple list
- The order vertices are specified is important

```
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
        ])
     };
     this.colors = { ... };
};
```

Our Square's Positions

- It will also be useful to record how many components are in each position
  - e.g., how many dimensions did we use for our vertex positions
- Recommend adding a new property to store that value
  - We'll use this value later
- We should do this for each vertex attribute

```
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 // 2 components for each
                          // position (2D coords)
     };
     this.colors = { ... };
};
```

Vertex Buffers

Vertex Buffers

- In order to draw with WebGL, vertex data must be sent to WebGL
- We do this using *vertex buffers*
  - they're internal data structures stored in the WebGL context
- We'll use the information in our model object to configure and initialize our vertex buffers
- We'll see other types of buffers as well
  - don't confuse a vertex buffer with frame buffers
  - there are just a lot of buffers in WebGL

Sequence to Initialize a Buffer

Step	Action	WebGL Function
1	Create a buffer	`gl.createBuffer`
2	Bind the buffer	`gl.bindBuffer`
3	Load the buffer with data	`gl.bufferData`
4	Find the vertex shader variable associated with the buffer's data	`gl.getAttribLocation`
5	Enable the vertex array	`gl.enableVertexAttribArray`
6	Associate the buffer with the attribute, and tell WebGL how to decipher the data in the buffer	`gl.vertexAttribPointer`

1. Create the buffer

- Create a buffer for each vertex attribute
  - assign it as a new property into that attribute's object

```
function Square () {
    ...

    this.positions.buffer = gl.createBuffer();
}
```

2. Bind the Buffer

- *Binding* makes all subsequent operations affect the bound object
  - bind the buffer you just created so you can update it
- There are multiple types of buffers, use `gl.ARRAY_BUFFER`

```
function Square () {
    ...

    this.positions.buffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, this.positions.buffer);
}
```

3. Load Data into the Buffer

- Copies data from your buffer into WebGL
- `gl.STATIC_DRAW` indicates that the data values won't change
  - use `gl.DYNAMIC_DRAW` if we anticipate the values will change.

```
function Square () {
    ...

    this.positions.buffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, this.positions.buffer);
    gl.bufferData(gl.ARRAY_BUFFER, this.positions.values, gl.STATIC_DRAW);
}
```

4. Finding the Shader's Vertex Attribute Variable

- Need to find the attribute handle for the vertex shader's variable
- Using the compiled shader program, just use for the variable's name

```
function Square () {
    ...

    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”);
 }
```

5. Enable the Vertex Array

- Finally, turn on the vertex array

```
function Square () {
    ...

    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, “aPosition”);
    gl.enableVertexAttribArray(this.positions.attributeLoc);
 }
```

Notes on Vertex Arrays

- Repeat that process for each vertex attribute
- Update your object's collection of attributes
- Then set up using the initialization sequence
- While our examples used static arrays of data, you can also dynamically generate data values

```
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 // 2 components for each
                          // position (2D coords)
     };
     this.colors = {
         values : new Float32Array([ ... ]),
         numComponets : 3
     };
};
```

Drawing

- A method is a property in a JavaScript object
  - Just assign it a function
- We'll add a `render` method that contains our drawing commands

```
function Square () {
    ...

    this.render = function () {
        ...
     }
}
```

Rendering (Setup)

Sequence to Draw using a Buffer

Step	Action	WebGL Function
1	Bind the buffer	`gl.bindBuffer`
2	Associate the buffer with the attribute, and tell WebGL how to decipher the data in the buffer	`gl.vertexAttribPointer`

Binding a Buffer

- Since each object has its own collections of buffers, we need to bind to the right buffer before rendering.
  - this call is identical to the one that you made in creating the buffer

```
function Square () {
    ...

    this.render = function () {
        gl.bindBuffer(gl.ARRAY_BUFFER, this.positions.buffer);
    };
}
```

Telling WebGL the Format of the Buffer's Data

- Whoa!
- These parameters will work in most situations
  - we'll explain what happens next time

```
function Square () {
    ...

    this.render = function () {
        gl.bindBuffer(gl.ARRAY_BUFFER, this.positions.buffer);
        gl.vertexAttribPointer(this.positions.attributeLoc,
            this.positions.numComponents, gl.FLOAT, 0, 0);
    };
}
```

Rendering (Drawing)

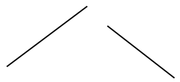
Rendering

- 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 one next time

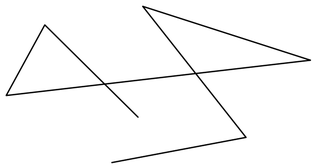
```
function Square() {
    this.count = 4;

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

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

WebGL Geometric Primitives

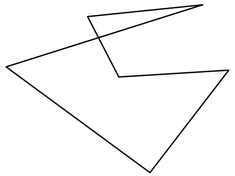
gl.POINTS



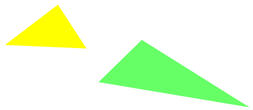
gl.LINES



gl.LINE_STRIP



gl.LINE_LOOP



gl.TRIANGLES

gl.TRIANGLE_STRIP

gl.TRIANGLE_FAN