Advanced Techniques

CS 385 - Class 29
10 May 2022

Bufferless Rendering

Recall Creating Buffers?

- We used buffers to store our vertices
  - positions
  - colors
  - lighting normals
  - texture coordinates
- We'd set up a buffer for each set of data, or combine them together
-

```
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([ ... ]),
         numComponents : 3
     };
};
```

And then we had to decipher the data

- We had all these parameters to figure out
  - number of components
  - data type
  - starting position
  - data packing

```
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);
    };
}
```

It's not always necessary

- We can *procedurally* generate data
  - as long as there's a formula of some nature
- We just make one draw call
- No data setup required
- The bound vertex shader will be executed four times (in this case)

JavaScript

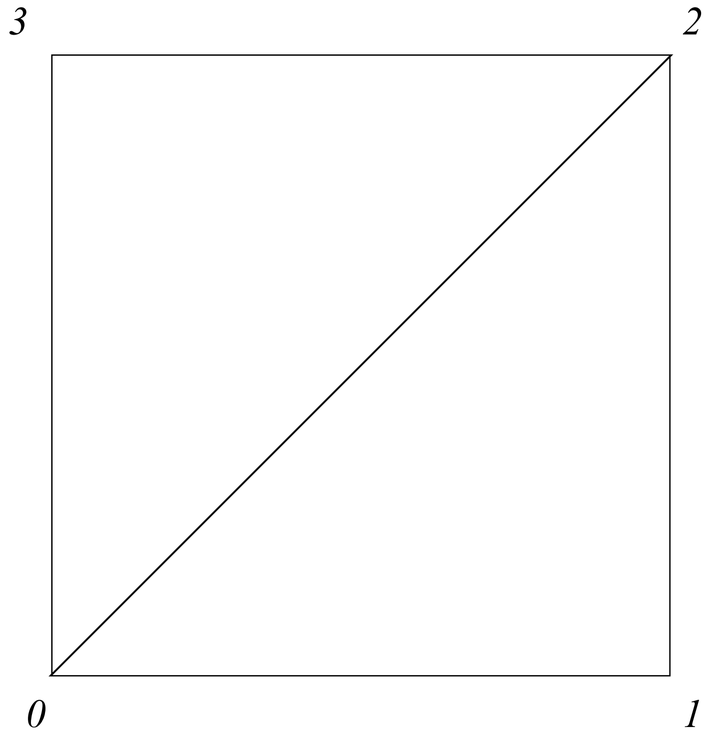
```
function Quad() {
  this.render = function() {
    gl.useProgram(program);
    // Load shader uniforms like P, MV

    gl.drawArrays(gl.TRIANGLE_FAN, 0, 4);
  };
};
```

Introducing `gl_VertexID`

- Every vertex is assigned a vertex id

Draw Call	`gl_VertexID` Value
gl.drawArrays	order
gl.drawElements	index value (what was passed in the `gl.ELEMENT_ARRAY_BUFFER`)

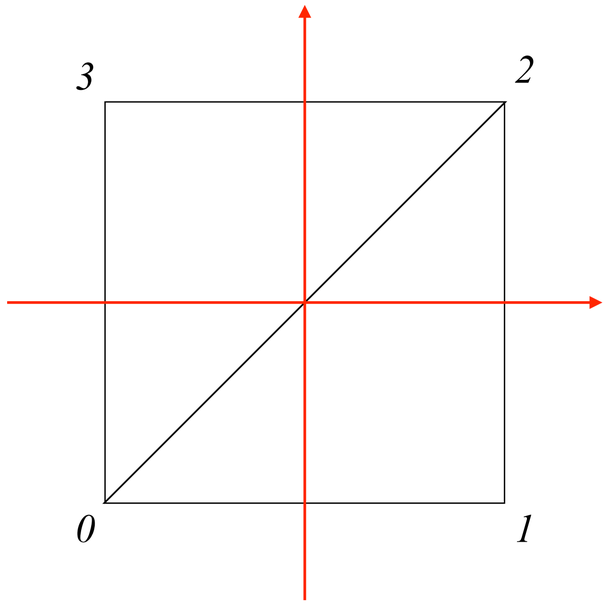
One way we could render a `gl.TRIANGLE_FAN`



Note: the positions aren't set here - we'll compute them based on the vertex ID

Generating Coordinates

- Position a quad centered at the origin
  - side length of one

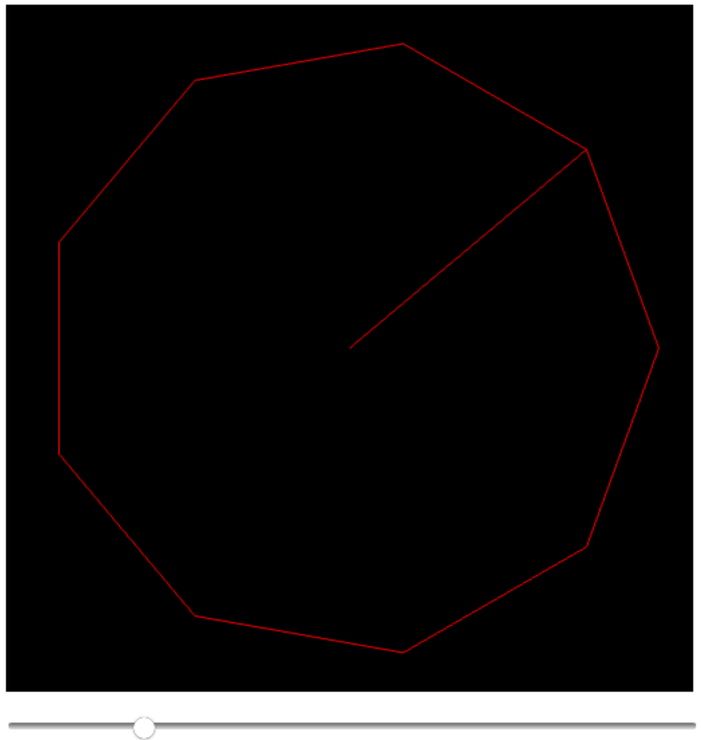


Vertex Shader

```
uniform mat4  P;
uniform mat4  MV;

void main()
{
  vec2  v;
  v.x = float(gl_VertexID == 1 || gl_VertexID == 2);
  v.y = float(gl_VertexID / 2);
  v -= 0.5;

  gl_Position = P * MV * vec4(v, 0.0, 1.0);
}
```

Generating Coordinates

- How about a disk?



Vertex Shader

```
uniform mat4  P;
uniform mat4  MV;
uniform float  slices;

void main() {
  const float PI = 3.141592653589793;

  float angle = float(gl_VertexID) * 2.0 * PI / (slices - 1.0);
  vec2 v = float(gl_VertexID > 0) * vec2(cos(angle), sin(angle));
  v *= 0.90;

  gl_Position = P * MV * vec4(v, 0.0, 1.0);
}
```

Instanced Rendering

Instances

- A new draw call to repeat rendering

```
gl.drawArraysInstanced(primitive, start, count, instances)
```

- Just like calling `gl.drawArrays()` `instances` times
- Provides a new shader variable to know which instance you're processing
  - `gl_InstanceID`
- There's also an indexed-draw version
  - `gl.drawElementsInstanced()`

Rendering with Instances

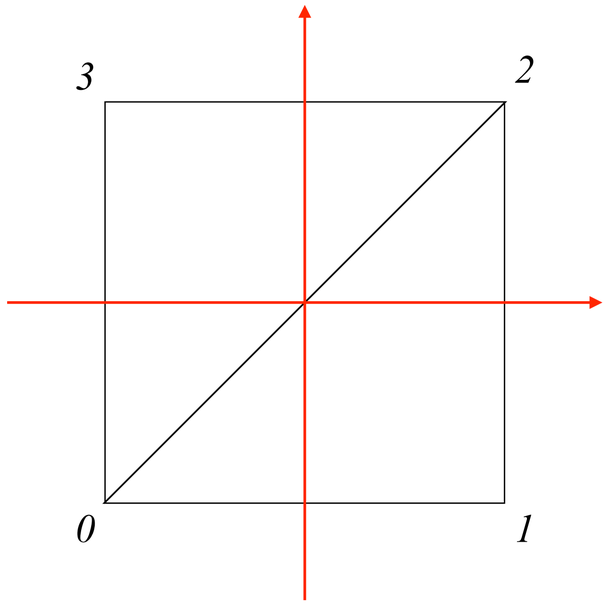
- Change draw call to an instanced form

JavaScript

```
function Quads() {
  this.render = function() {
    gl.useProgram(program);
    // Load shader uniforms like P, MV

    // draw 16 separate quads
    gl.drawArraysInstanced(gl.TRIANGLE_FAN,
        0, 4, 16);
  };
};
```

Generating Coordinates

- Position a quad centered at the origin
  - side length of one



Vertex Shader

```
uniform mat4  P;
uniform mat4  MV;

void main()
{
  vec2  v;
  v.x = float(gl_VertexID == 1 || gl_VertexID == 2);
  v.y = float(gl_VertexID / 2);
  v -= 0.5;

  gl_Position = P * MV * vec4(v, 0.0, 1.0);
}
```