Introduction to Computer Graphics with WebGL

Ed Angel

Shaders

GLSL

- OpenGL ES Shading Language (ESSL)
- Some key differences between ESSL and recent versions of the OpenGL Shading Language
  - ESSL requires fragment shader to set precision
  - varying qualifiers replaced by in and out
  - `gl_FragColor` deprecated

Data Types

- C types: int, float, bool
- Vectors:
  - float vec2, vec3, vec4
  - Also int (ivec) and boolean (bvec)
- Matrices: mat2, mat3, mat4
  - Stored by columns
  - Standard referencing m[row][column]
- C++ style constructors
  - vec3 a =vec3(1.0, 2.0, 3.0)
  - vec2 b = vec2(a)

No Pointers

- There are no pointers in GLSL
- We can use C type structs which can be copied back from functions
- Because matrices and vectors are basic types they can be passed into and output from GLSL functions, e.g.

```
mat3 func(mat3 a)
```

- variables passed by copying

Qualifiers

- GLSL has many of the same qualifiers as C/C++ such as `const`
- Need others due to the nature of the execution model
- Variables can change
  - Once per primitive (uniform qualified)
  - Once per vertex (attribute qualified)
  - Once per fragment (varying qualified)
  - At any time in the application
- Vertex attributes are output by the vertex shader are interpolated by the rasterizer into fragment attributes

Attribute Qualifier

- Attribute-qualified variables can change at most once per vertex
- There are a few built in variables such as `gl_Position` but most have been deprecated
- User defined (in application program)

```
-attribute vec4 color
-attribute float temperature
-attribute vec3 velocity
```

- recent versions of GLSL use **`in`** and **`out`** qualifiers to get data to and from shaders

Uniform Qualified

- Variables that are constant for an entire primitive
- Can be changed in application and sent to shaders
- Cannot be changed in shader
- Used to pass information to shader such as the time or a rotation angle for transformations

Varying Qualified

- Variables that are passed from vertex shader to fragment shader
- Automatically interpolated by the rasterizer
- With WebGL, GLSL uses the varying qualifier in both shaders

```
varying vec4 color;
```

- More recent versions of WebGL use `out` in vertex shader and `in` in the fragment shader

```
out vec4 color; //vertex shader
in vec4 color;  // fragment shader
```


Our Naming Convention

- attributes passed to vertex shader have names beginning with v (v Position, vColor) in both the application and the shader
  - Note that these are different entities with the same name
- Variable variables begin with f (fColor) in both shaders
  - must have same name
- Uniform variables are unadorned and can have the same name in application and shaders

Example: Vertex Shader

```
attribute vec4 vColor;
varying vec4 fColor;
void main()
{
  gl_Position = vPosition;
  fColor = vColor;
}
```


Corresponding Fragment Shader

```
precision mediump float;

varying vec3 fColor;
void main()
{
  gl_FragColor = fColor;
}
```


Sending Colors from Application

```
var cBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
gl.bufferData( gl.ARRAY_BUFFER,
        flatten(colors), gl.STATIC_DRAW );

var vColor = gl.getAttribLocation( program,
             "vColor" );
gl.vertexAttribPointer( vColor, 3, gl.FLOAT,
              false, 0, 0 );
gl.enableVertexAttribArray( vColor );
```

Sending a Uniform Variable

```
// in application

vec4 color = vec4(1.0, 0.0, 0.0, 1.0);
colorLoc = gl.getUniformLocation( program, "color" );
gl.uniform4f( colorLoc, color);

// in fragment shader (similar in vertex shader)

uniform vec4 color;

void main()
{
    gl_FragColor = color;
}
```


Operators and Functions

- Standard C functions
  - Trigonometric
  - Arithmetic
- Geometry helper functions
  - Normalize, reflect, length
- Overloading of vector and matrix types

```
mat4 a;
vec4 b, c, d;
c = b*a; // a column vector stored as a 1d array
d = a*b; // a row vector stored as a 1d array
```


Swizzling and Selection

- Can refer to array elements by element using [] or selection (.) operator with
  - `x, y, z, w`
  - `r, g, b, a`
  - `s, t, p, q`
  - `a[2], a.b, a.z, a.p` are the same
- **Swizzling** operator lets us manipulate components

```
vec4 a, b;
a.yz = vec2(1.0, 2.0, 3.0, 4.0);
b = a.yxzw;
```