Introduction to Computer Graphics with WebGL

• Ed Angel
• Professor Emeritus of Computer Science
• Founding Director, Arts, Research, Technology and Science Laboratory
• University of New Mexico

Primitives

GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN

GL_Points

Alon:
The term ‘location’ refers to location in Memory.
The term ‘position’ refers to geometric position.

<html>
<head></head>
<body>
  <!doctype html>
  <title>CSC 4600 - WebGL Example: Square</title>
  <script id="vertexShader" type="x-shader/x-vertex">...</script>
  <script id="fragmentShader" type="x-shader/x-fragment">...</script>
  <script type="text/javascript">...</script>
  <style>...</style>
  <canvas id="mycanvas" width="834" height="416"></canvas>
</body>
</html>

const fs_source = document.getElementById('fragmentShader').text;
const fs = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fs, fs_source);
gl.compileShader(fs);

prog = gl.createProgram();
gl.attachShader(prog, vs);
gl.attachShader(prog, fs);
gl.linkProgram(prog);
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)

VERTEX SHADER

- **Input** comes from the client program, through “uniform variables” and “attributes”
  - uniform: Constant for all vertices
  - attribute: Some part of a vertex (normal, position, tangent, color, etc.)
- **Output** goes to the rasterizer; through “varying variables”
  - varying: Automatically interpolated values
  - gl_Position: special output, must be written to tell OpenGL the vertex position

USEFUL BUILT-IN OPERATIONS

- normalize(vec)
- length(vec)
- reflect(vec, vec)
- pow, max, sin, cos, etc.
- dot(vec, vec)
- Transforms: mat * vec
- Overloaded operators: +, -, *
- Swizzling vec.xyz = vec.zyx
- transpose(mat)
No Pointers

• There are no pointers in GLSL
• We can use C 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 such as const as C/C++
• Need others due to the nature of the execution model
• Variables can change
  Once per primitive
  Once per vertex
  Once per fragment
  At any time in the application
• Vertex attributes are interpolated by the rasterizer into fragment attributes

Attribute Qualifier

• Attribute-qualified variables can change at most once per vertex
• Input to vertex attribute - at most 16 attributes (vec4)
• There are a few built in variables such as gl_Position but most have been deprecated
• User defined (in application program)
  attribute float temperature
  attribute vec3 velocity
  Recent versions of GLSL use in and out qualifiers to get 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 bounding box of a primitive or transformation matrices

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
Tessellation Shaders, Geometry Shaders, - only in OpenGL but not WebGL

Application (JS) - uses graphics API (on CPU)
is in charge of controlling the shades ,
But shaders run on GPU
Important - Apps have different memory than GPU

Shaders always begin with a version declaration, followed by a list of input and output variables, uniforms and its `main` function.

```
#version version_number

in type in_variable_name;
in type in_variable_name;
out type out_variable_name;
uniform type uniform_name;

void main()
{
// process input(s) and do some weird graphics stuff
...
// output processed stuff to output variable
out_variable_name = weird_stuff_we_processed;
}
```

Buffers

Allocating GPU memory -via buffers

```javascript
var positions =
  [-0.8, 0.4, 0,
   0.8, 0.4, 0,
   0.8, -0.4, 0,
   -0.8, -0.4, 0];

var position_buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, position_buffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);
```

```
<script id="vertexShader" type="x-shader/x-vertex">
attribute vec3 pos;
attribute vec4 clr;
uniform mat4 trans;
varying vec4 vcolor;

void main()
{
  gl_Position = trans * vec4(pos,1);
  vcolor = clr;
}
</script>
```

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

varying vec4 vcolor;

void main()
{
  gl_FragColor = vcolor;
}
</script>
```

Vertices must be in canonical space `[-1,1]` - note the use of `trans`

```javascript
const vs_source = document.getElementById('vertexShader').text;
const vs = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vs, vs_source);
gl.compileShader(vs);

const fs_source = document.getElementById('fragmentShader').text;
const fs = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fs, fs_source);
gl.compileShader(fs);
```

```javascript
const vs = gl.getUniformLocation(prog, 'trans');

var matrix =
  [1,0,0,0,
   0,1,0,0,
   0,0,1,0,
   0,0,0,1];

gl.useProgram(prog);
gl.uniformMatrix4fv(vs, false, matrix);
```
### Attributes for texture

#### Vertex shader

```glsl
attribute vec3 vert_pos;
attribute vec3 vert_tang;
attribute vec3 vert_bitang;
attribute vec2 vert_uv;
uniform mat4 model_mtx;
uniform mat4 norm_mtx;
uniform mat4 proj_mtx;
```

#### Fragment shader

```glsl
varying vec2 frag_uv;
```

### 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.

### Sending Colors from Application

```javascript
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);
```

### Corresponding Fragment Shader

```glsl
precision mediump float;

varying vec3 fColor;

void main()
{
    gl_FragColor = fColor;
}
```
Sending a Uniform Variable

```c
// 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
    - Normalize, reflect, length
- Overloading of vector and matrix types
  ```c
  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** (in computer graphics, swizzles are a class of operations that transform vectors by rearranging components) operator lets us manipulate components
  ```c
  vec4 a, b;
  a.yz = vec2(1.0, 2.0, 3.0, 4.0);
  b = a.yzxw;
  ```