CS 418: Interactive Computer Graphics

The GLSL Shading Language

Eric Shaffer

Some Slides Adapted from
Angel and Shreiner: Interactive
Computer Graphics 7E © Addison-Wesley 2015
GLSL Data Types

- C/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)
Memory Layout and Matrices

- The OpenGL/WebGL/GLSL convention is to layout matrices in what they call **column-major order**

\[
\begin{pmatrix}
a & b & c & t_x \\
d & e & f & t_y \\
g & h & i & t_z \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

is laid out as 16 contiguous floating point numbers \([a, d, g, 0, b, e, h, 0, c, f, i, 0, t_x, t_y, t_z, 1]\)

- This is the layout the glmatrix library uses
No Pointers

- There are no pointers in GLSL
  - Bonus question: Is dynamic memory allocation allowed
- Can use C structs which can be copied back from functions
- Matrices and vectors are basic types
  - they can be passed into and returned from from GLSL functions
  - e.g. mat3 func(mat3 a)
- Arguments passed by copy
  - Bonus bonus question: Is recursion allowed?
Things GLSL Does Have…

- A ridiculous number of built-in functions
- Check them out before writing code on your own to do cross-product, etc.
Function Arguments

- Function parameters can be qualified as:
  - `in` argument is copied into function, any changes to value are NOT read back.
  - `out` argument NOT copied into Function, value copied back.
  - `inout` argument copied into Function, value copied back.
  - Default is `in`.

- Example

```c
void MyFunction(in float inputValue,
                 out int outputValue,
                 inout float inAndOutValue);
```
Qualifiers

- GLSL has many of the same qualifiers as C/C++
  - e.g. const
- Need others due to the nature of the execution model
- Certain types of variables can be set
  - Once per shader execution (i.e. once per draw call)
  - Once per vertex
  - Once per fragment
Attribute Qualifier

- Attribute-qualified variables
  - change at most once per vertex
  - A few built in variables such as `gl_Position`

- User defined (in application program)
  - `attribute float temperature`
  - `attribute vec3 velocity`
Uniform Qualified

- Variables that are constant for a shader invocation
- Can be changed in application and sent to shaders
- Cannot be changed in shader
- Passes information to shader like 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

```glsl
varying vec4 color;
```
Example: Vertex Shader

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

```plaintext
precision mediump float;

varying vec4 fColor;
void main()
{
    gl_FragColor = fColor;
}
```
Operators and Functions

- Standard C functions
  - Trigonometric
  - Arithmetic
- Also have vector-specific functions such as: normalize, reflect, length
- Overloading of vector and matrix types
  ```c
  mat4 a;
  vec4 b, c, d;
  c = b*a;
  d = a*b;
  ```
- NOTE: multiplying a vector from the left to a matrix corresponds to multiplying it from the right to the transposed matrix
  - Useful when you want to use a transposed matrix...
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;
  b = a.yxzw;
  ```
Linking Shaders with Application

- Read shaders
- Compile shaders
- Create a program object
- Link everything together
- Link variables in application with variables in shaders
  - Vertex attributes
  - Uniform variables
Program Object

- Container for shaders
  - Can contain multiple shaders
  - Other GLSL functions

```javascript
var program = gl.createProgram();

gl.attachShader( program, vertShdr );
gl.attachShader( program, fragShdr );
gl.linkProgram( program );
```
Reading a Shader

- Shaders are added to the program object and compiled
- Can pass a shader as a null-terminated string using the function
  ```javascript
  gl.shaderSource( fragShdr, fragElem.text );
  ```
- If shader source is in HTML file, can get it by `getElementById` method
- If shader is in a file, we can write a reader to convert the file to a string
Adding a Vertex Shader

```javascript
var vertShdr;
var vertElem =
    document.getElementById( vertexShaderId );

vertShdr = gl.createShader( gl.VERTEX_SHADER );

gl.shaderSource( vertShdr, vertElem.text );
gl.compileShader( vertShdr );

// after program object created
gl.attachShader( program, vertShdr );
```
Following code may be a security issue with some browsers

- if you try to run it locally
- Cross Origin Request

```javascript
function getShader(gl, shaderName, type) {
  var shader = gl.createShader(type);
  shaderScript = loadFileAJAX(shaderName);
  if (!shaderScript) {
    alert("Could not find shader source: "+shaderName);
  }
}
```
In GLSL for WebGL we must specify desired precision in fragment shaders
- artifact inherited from OpenGL ES
- ES must run on very simple embedded devices that may not support 32-bit floating point
- All implementations must support mediump
- No default for float in fragment shader

Can use preprocessor directives (#ifdef) to check if highp supported and, if not, default to mediump
#ifdef GL_FRAGMENT_SHADER_PRECISION_HIGH
  precision highp float;
#else
  precision mediump float;
#endif

varying vec4 fcolor;
void main(void)
{
  gl_FragColor = fcolor;
}