Computer Graphics

Minho Kim

Dept. of Computer Science

University of Seoul

Chapter 6: The OpenGL ES Shading Language (GLSL ES)

What to Learn

- Data, variables, and variable types
- Vector, matrix, structure, array, and sampler types
- Operators, control flow, and functions
- Attributes, uniform, and varying variables
- Precision qualifiers
- Preprocessor and directives

- WebGL 1.0 shading language is based on GLSL ES (OpenGL Shading Language for Embedded Systems) 1.0, but does not support all but subset of the features.
- GLSL ES 1.0 Spec can be found here. (pdf)

Overview of GLSL ES

- Streamlined version of GLSL for (1) reduced power consumption and (2) reduced manufacturing costs.
- Based on the C language syntax.

GLSL ES

- Any shader program has its main function “`void main()`”.
- Two data types: numerial and boolean (`true` and `false`)
- Reserved keywords should be avoided as variable names
- Variable names cannot start with `gl_`, `webgl_`, or `_webgl_`.
- Basic types: `float`, `int`, `bool`
- Type sensitive → “`float x = 8;`” generates an error.
  → requires an explicit type conversion such as “`float(8)`”
- An expression like “`8.0f`” is not allowed. → Use “`8.0`” instead.

Vectors and Matrices

- Vector types: vec2/vec3/vec4, ivec2/ivec3/ivec4, bvec2/bvec3/bvec4
- Matrix types: mat2/mat3/mat4
- Legitimate constructions

```
vec3 v3 = vec3(1.0, 0.0, 0.5);
vec2 v2 = vec2(v3); // vec2 v2 = vec2(v3[0], v3[1]);
vec4 v4 = vec4(1.0); // vec4 v4 = vec4(1.0, 1.0, 1.0, 1.0);
vec4 v5 = vec4(v3, 0.0); // vec v5 = vec4(v3[0], v3[1], v3[2], 0.0);
```

- Matrices are constructed in column-major order.
- A matrix can be constructed from vectors as its columns.

```
mat2 m = mat2(v1,v2); // v1 and v2 are vec2 types
```

Vector and Matrices

- Accessing components by names
  - `x,y,z,w` – useful for coordinates
  - `r,g,b,a` – useful for colors
  - `s,t,p,q` – useful for texcoords
- Swizzling – Multiple components can be accessed by combining component names
  - Ex) `v.xy`, `v.xx`, `v.grb`
  - Any component can be repeated
  - Also can used for left-side expression: `v.xy = vec2(1.0, 2.0)`
  - Cannot mix names from different sets: `v.rx` (not allowed)

Vectors and Matrices

- Accessing components using the array indexing operator `[]` is possible.
- For a matrix, `[]` is used to select a column. → The 3rd component of the 2nd column of `m` can be access as "`m[1][2]`" or "`m[1].y`"
- The index should be a constant index. (loop index allowed.)

Structures

- Declaration / construction

```
struct light {
    vec4 color;
    vec3 position;
}
light l1, l2;
```

- Assignment

```
l1 = light(vec4(0.0, 1.0, 0.0, 1.0), vec3(8.0, 3.0, 0.0));
```

- Operations: = (assignment), ==, ! = (comparison)

Arrays

- One-dimensional only
- Declaration
  ```
  float floatArray[2];
  ```
  - Cannot be initialized at declaration time.
- The array size should be determined when declared.
- Arrays cannot be qualified as `const`.
- Can be accessed using the array indexing operator `[]`.
- Only an integral constant expression or uniform variable can be used as an index of an array.
- Initialization – component-wise
  ```
  floatArray[0] = 1.0;
  floatArray[1] = 2.0;
  ```
  - cf) WebGL 2.0 (https://xregy.github.io/webgl/src/WebGL2_array_in_shader.html)
    ```
    floatArray = float[](1.0, 2.0);
    ```

Samplers

- Used to access textures
- Two types: `sampler2D` and `samplerCube`
- Can be used as a uniform variable only
- Only texture unit number can be assigned using `gl.uniform1i()`.
- Only three operations are allowed: =, ==, !=
- Minimum # of variables of the sampler type
  - vertex shader: 0
    `const mediump int gl_MaxVertexTextureImageUnits`
  - fragment shader: 8
    `const mediump int gl_MaxTextureImageUnits`
- Host-side samplers are supported in WebGL 2.0

Precedence of Operators

- Almost the same as in JavaScript and C.
- Bitwise operators are reserved for future support.

Conditional Control Flow and Iteration

- Almost the same as in JavaScript and C.
- `switch-case` statement is not supported.
- The for statement
  - The loop index of can be declared only in the *for-init-statement*.
    - e.g. `for(int i=0 ; i<3 ; i++) {…}`
  - Empty condition becomes true.
  - **Several restrictions due to inline expansion** (loop unrolling)
    - Only a single loop index is allowed. The loop index must have the type `int` or `float`.
    - *loop-index-expression* must have one of the following forms: `i++`, `i--`, `i+=`*constant-expression*, `i-=`*constant-expression*
    - *conditional-expression* is a comparison between a loop index and an integral constant expression.
    - Within the body of the loop, the loop index cannot be assigned.
- `continue`, `break`, `discard`

Functions

- Vector and matrix types can be used as parameters and be returned.
- All function calls are in-line. → A recursive call isn't allowed.
- Parameter qualifiers
  - `in`: Default. Passed by value
  - `const in`: Pass by constant value. Cannot be modified.
  - `out`: Passed by reference. No initial value.
  - `inout`: Passed by reference.

Built-In Functions

- Math functions
- Texture lookup functions
  - `texture2D()`, `textureCube()`, `texture2DProj()`, etc.
- https://www.khronos.org/opengles/sdk/docs/reference_cards/OpenGL-ES-2_0-Reference-card.pdf

Global and Local Variables

- The same as in JavaScript and C.

Storage Qualifiers

- `const`, `attribute`, `uniform`, `varying`
- Implementation-dependent limit on the # of variables available
  - `attribute`: `gl_MaxVertexAttribs`
  - `uniform`: `gl_MaxVertexUniformVectors` (vertex shader), `gl_MaxFragmentUniformVectors` (fragment shader)
  - `varying`: `gl_MaxVaryingVectors`
- `const`
  - Should be initialized at their declaration time.
- `attribute`
  - Available only in vertex shaders
  - Only `float`, `vec*`, `mat*` types allowed. (Other types support in WebGL2)
  - (WebGL2) switched to "`in`" in vertex shaders.

Storage Qualifiers (cont’d)

- `uniform`
  - Read-only
  - Can be declared as any data type other than array and structure.
- `varying`
  - Linearly interpolated by the rasterizer → `float`, `vec*`, `mat*` types only
  - (WebGL2) switched to “`out`” in vertex shaders and “`in`” in fragment shaders

Precision Qualifiers

- Newly introduced in GLSL ES
- Two purposes
  - To execute shader programs more efficiently
  - To reduce memory size
  - → To reduce power consumption
- Specifies how much precision (# of bits) each data type should have
- Lower precision may lead to incorrect results → Balancing required
- `highp`, `mediump`, `lowp`
- https://www.khronos.org/opengles/sdk/docs/reference_cards/OpenGL-ES-2_0-Reference-card.pdf

Precision Qualifiers: Notes

- Fragment shaders may not support highp in some WebGL implementations
  - Supported if `GL_FRAGMENT_PRECISION_HIGH` is defined.
- The actual range and precision are implementation dependent
  - Can be checked by `gl.getShaderPrecisionFormat()`
- Examples
  - `mediump float size;`
- A default for each data type can be set using the keyword precision
  - e.g., `precision mediump float;`
- Default precisions are set except `float`

Preprocessor Directives

- `#if`, `#ifdef`, `#idndef`
- `#define`, `#undef`
- Predefined macros
  - `GL_ES`, `GL_FRAGMENT_PRECISION_HIGH`
- `#version` *number*
  - `100` for GLSL ES 1.00 and `101` for GLSL ES 1.01
  - Must be specified at the top of the shader program and only be proceded by comments and white space.
    → Requires care when the shader sources are embedded in the HTML file.