Computer Graphics

spring, 2013

Chapter 5
OpenGL ES Shading Language

What to Discuss…

- Variables and variable types
- Vector and matrix construction and selection
- Constants
- Structures and arrays
- Operators, control flow, and functions
- Attributes, uniforms, and varyings
- Preprocessor and directives
- Uniform and varying packing
- Precision qualifiers and invariance

Variable Types

- **Scalars:** `float`, `int`, `bool`
- **Floating-point vectors:** `float`, `vec2`, `vec3`, `vec4`
- **Integer vectors:** `int`, `ivec2`, `ivec3`, `ivec4`
- **Boolean vectors:** `bool`, `bvec2`, `bvec3`, `bvec4`
- **Matrices (floating-point):** `mat2`, `mat3`, `mat4`

Variable Constructors

- Very strict rules for type conversion -- Variables can only be assigned to or operated on other variables of the same type
- Constructors are used for initialization & type conversion

```
float myFloat = 1.0;
bool myBool = true;
int myInt = 0;
myFloat = float(myBool); // Convert from bool -> float
myFloat = float(myInt); // Convert from int -> float
myBool = bool(myInt); // Convert from int -> bool
```

Vector Constructors

- Two ways to pass arguments to vector constructors
  - One scalar argument -- assigned to all components
  - Multiple scalar or vector arguments -- values are set from left to right

```
vec4 myVec4 = vec4(1.0);              // myVec4 = {1.0, 1.0, 1.0, 1.0}
vec3 myVec3 = vec3(1.0, 0.0, 0.5); // myVec3 = {1.0, 0.0, 0.5}
vec3 temp = vec3(myVec3);             // temp = myVec3
vec2 myVec2 = vec2(myVec3);           // myVec2 = {myVec3.x, myVec3.y}
myVec4 = vec4(myVec2, temp, 0.0);  // myVec4 = {myVec2.x, myVec2.y,
                                      //temp, 0.0}
```

Matrix Constructors

- Basic rules
  - Only one scalar -- assigned to diagonal components (others set to zeros)
  - Can be constructed out of multiple vectors
  - Can be constructed out of multiple scalars
- Can be constructed from any combination of scalars and vectors
- Matrices are stored in column major order

Vectors and Matrix Components

- Two ways to access components -- “.” operator or array subscripting
  - {x,y,z,w}, {r,g,b,a}, {s,t,r,q}
  - ex) A.x, A[0]
- Swizzling supported
  - ex) A.xy
  - Cannot mix naming convention -- ex) A.xgr
  - Components can be reordered

```
vec3 myVec3 = vec3(0.0, 1.0, 2.0); // myVec3 = {0.0, 1.0, 2.0}
vec3 temp;
temp = myVec3.xyz;                 // temp = {0.0, 1.0, 2.0}
temp = myVec3.xxx;                 // temp = {0.0, 0.0, 0.0}
temp = myVec3.zyx;                 // temp = {2.0, 1.0, 0.0}
```

Vector and Matrix Components (cont'd)

- Accessing vector/matrix element using variable index ("dynamic indexing") might not supported in GLES2! -- except using uniform variables
- Matrices are treated as being composed of multiple vectors

```
mat4 myMat4 = mat4(1.0);   // Initialize diagonal to 1.0 (identity)
vec4 col0 = myMat4[0];     // Get col0 vector out of the matrix
float m1_1 = myMat4[1][1]; // Get element at [1][1] in matrix
float m2_2 = myMat4[2].z;  // Get element at [2][2] in matrix
```

Constants

- Read-only
- Cannot be modified
- Should be initialized when declared

```
const float zero = 0.0;
const float pi = 3.14159;
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
const mat4 identity = mat4(1.0);
```

Structures

- A new type & constructor are defined by a structure

```
struct fogStruct {
   vec4  color;
   float start;
   float end;
} fogVar;
fogVar = fogStruct(vec4(0.0, 1.0, 0.0, 0.0), // color
                   0.5,                      // start
                   2.0);                     // end
```

Arrays

- As vectors/matrices, dynamic indexing is not supported on many implementations (except uniform variables)
- No syntax to initialize an array at creation time --> Arrays cannot be const qualified

Operators

- Almost the same as C
- Very strict type rules between operators
- Comparison operators only for scalars (built-in functions for vectors/matrices in Appendix B)

```
float myFloat;
vec4  myVec4;
mat4  myMat4;
myVec4 = myVec4 * myFloat; // Multiplies each component of myVec4
                           // by a scalar myFloat
myVec4 = myVec4 * myVec4;  // Multiplies each component of myVec4
                           // together (e.g., myVec4 ^ 2 )
myVec4 = myMat4 * myVec4;  // Does a matrix * vector multiply of
                           // myMat4 * myVec4
myMat4 = myMat4 * myMat4;  // Does a matrix * matrix multiply of
                           // myMat4 * myMat4
myMat4 = myMat4 * myFloat; // Multiplies each matrix component by
                           // the scalar myFloat
```

Functions

- Almost the same with C
- Special qualifiers to define whether a variable can be modified by the function
  - `in` (default) -- passed by value, not modified
  - `inout` -- passed by reference
  - `out` -- value is not passed, but modified on return
- Recursive functions not supported -- some implementations make functions inline due to the lack of stack/control flow

Built-in Functions

- See Appendix B
- Use built-in functions as much as possible for best performance!

if-then-else

- It’s slow!
- The conditional statement must be boolean type

`for` Loops

- must have an iteration count that is known at compile time
- there must be only one loop iteration variable and it must be incremented or decremented using a simple statement (i++, i--, i+=constant, i-=constant)
- the stop condition must be a comparison between the loop index and a constant expression
- you must not change the value of the iterator in the loop.
- Essentially, the OpenGL ES Shading Language does not require hardware to provide looping support (loop unrolling)

`for` Loops (cont'd)

```
float myArr[4];
for(int i = 0; i < 3; i++)
{
    sum += myArr[i]; // NOT ALLOWED IN OPENGL ES, CANNOT DO
                     // INDEXING WITH NONCONSTANT EXPRESSION
}
...
uniform int loopIter;
// NOT ALLOWED IN OPENGL ES, loopIter ITERATION COUNT IS NONCONSTANT
for(int i = 0; i < loopIter; i++)
{
    sum += i;
}
```

Uniforms

- Read-only values passed by the application through the GLES2
- All parameters to a shader that is constant across either all vertices or fragments, but that is not known at compile time
- ex) transformation matrices, light parameter, colors, etc.
- Stored in “constant store” --> # of uniforms are limited
- # of uniforms >= 128 (vert), 16 (frag)
  - gl_MaxVertex(Fragment)UniformVectors (in shaders)
  - GL_MAX_VERTEX(FRAGMENT)_UNIFORM_VECTORS (in host)

Attributes

- Per-vertex input parameters
- Available only in the vert shader
- ex) positions, normals, tex coords, colors, etc.
- Usually stored as a vertex array in the host
- # of attribs >= 8
  - gl_MaxVertexAttribs (in shaders)
  - GL_MAX_VERTEX_ATTRIBS (in host)

Varyings

- Output of a vert shader, input of a frag shader
- Linearly interpolated across the primitive during rasterization
- Declared in both vert and frag shaders identically
- Stored in “interpolators”
- # of available varyings >= 8
  - gl_MaxVaryingVectors (in shaders)
  - GL_MAX_VARYING_VECTORS (in host)

Example

```
// Vertex shader
uniform mat4 u_matViewProjection;
attribute vec4 a_position;
attribute vec2 a_texCoord0;
varying vec2 v_texCoord; // Varying in vertex shader
void main(void)
{
    gl_Position = u_matViewProjection * a_position;
    v_texCoord = a_texCoord0;
}

// Fragment shader
precision mediump float;
varying vec2 v_texCoord; // Varying in fragment shader
uniform sampler2D s_baseMap;
uniform sampler2D s_lightMap;
void main()
{
    vec4 baseColor;
    vec4 lightColor;
    baseColor = texture2D(s_baseMap, v_texCoord);
    lightColor = texture2D(s_lightMap, v_texCoord);
    gl_FragColor = baseColor * (lightColor + 0.25);
}
```

Preprocessor and Directives

- Follows many conventions of a standard C++ preprocessor
- #define, #undef, #if, #ifdef, #ifndef, #else, #elif, #endif
- Macros cannot be defined with parameters
- Predefined macros -- __LINE__, __FILE__, __VERSION__, GL_ES
- #error -- error message during compilation
- #pragma -- implementation-specific directives
- #version
  - sets the shader version (100 for GLES2)
  - Must occur at the beginning

Preprocessor and Directives (cont'd)

- #extension -- enables & sets the behavior of extensions
- Syntax

```
// Set behavior for an extension
#extension extension_name : behavior
// Set behavior for ALL extensions
#extension all : behavior
```

- behaviors
  - `require`: error if not supported
  - `enable`: warn if not supported
  - `warn`: warn on any use
  - `disable`: error on any use

Uniform and Varying Packings

- Uniforms stored in “constant store”
- Varyings stored in “interpolators”
- How are various variable declarations mapped to the physical storage space? --> packing rules
- Physical storage is organized as a nx4 grid
- Packing rules to make the complexity of the generated codes remain constant
- Arrays packed across boundaries
- # of uniforms & varyings should not exceed the minimum allowed storage size after packing

Packing Rules

```
uniform mat3 m;
uniform float f[6];
uniform vec3 v;
```

Location	X	Y	Z	W
0	M[0].x	m[0].y	m[0].z	-
1	M[1].x	m[1].y	m[1].z	-
2	M[2].x	m[2].y	m[2].z	-
3	f[0]	-	-	-
4	f[1]	-	-	-
5	f[2]	-	-	-
6	f[3]	-	-	-
7	f[4]	-	-	-
8	f[5]	-	-	-
9	v.x	v.y	v.z	-6

Without packing rules

Location	X	Y	Z	W
0	M[0].x	m[0].y	m[0].z	f[0]
1	M[1].x	m[1].y	m[1].z	f[1]
2	M[2].x	m[2].y	m[2].z	f[2]
3	v.x	v.y	v.z	f[3]
4	-	-	-	f[4]
5	-	-	-	f[5]

With packing rules

Precision Qualifiers

- To specify the precision with which computations for a shader variables are performed
- low (`lowp`), medium (`mediump`), high (`highp`)
- Hints to the compiler
- Lower precision --> faster shaders, better power efficiency
- Supporting multiple / high precisions is not mandatory
- The precision specified by a precision qualifier has an implementation-dependent range and precision
- Default precision qualifiers
  - Syntax: `precision highp float;`

Precision Qualifiers (cont'd)

- Default precision
  - vert shaders: highp for both float & int
  - frag shaders: no default precision
- GL_FRAGMENT_PRECISION_HIGH, OES_fragment_precision_high

```
#ifdef GL_FRAGMENT_PRECISION_HIGH
    precision highp float;
#else
    precision mediump float;
#endif
```

Invariance

- Applied to varyings
- Instructions might be reordered due to compiler optimization --> exactly identical results may not be guaranteed
- May be an issue for multipass shader effects
- Specify that if the same computations are used to compute an output, its value must be exactly the same (or invariant)
- Using invariance might degrade the performance
- Doesn’t guarantee invariance across GLES implementations

Invariance (cont'd)

```
uniform mat4 u_viewProjMatrix;
attribute vec4 a_vertex;
invariant gl_Position;
void main
{
    // ...
    gl_Position = u_viewProjMatrix * a_vertex; // Will be the same
                                               // value in all
                                               // shaders with the
                                               // same viewProjMatrix
                                               // and vertex
}
```

- To make all variables globally invariant
  - `#pragma STDGL invariant(all)`