Uniforms

by Ian Romanick

1 Introduction

While attributes change per vertex, uniforms are constant across a group of vertexes. Uniforms can be used for things like transformation matrices, lights, and other per-object parameters. Figure 1 uses a uniform called *model_view_projection* to transform the attribute *gl_Vertex*.

Figure 1 Simple vertex shader

```
uniform mat4 model_view_projection;
void main(void)
{
    gl_Position = model_view_projection * gl_Vertex;
}
```

1.1 Uniform Storage Limits

Uniforms are shared by all program units. If there is a uniform *light_position* in the vertex shader and a uniform of the same name in the fragment shader, they will have the same value. By implication, they must also have the same type. A program will fail to link if uniforms with the same name in different compilation units have different types.

Each program unit has a limited amount of storage for uniforms. Classically, uniforms have been implemented as a small, read-only register array on the GPU. GPUs designed in this way have very limited uniform storage, which may be as small as 512 floating-point values. GPUs a generation or two later are capable of storing uniforms in much larger off-chip buffers. GPUs designed in this way can typically support 4,096 floating-point values.

The actual number of uniforms available for each program unit is queryable. Limits are queried by calling `glGetIntegerv` with one of the query enumerants listed in Table 1. Each OpenGL version specifies a minimum value for each of these queries[1]. Note that OpenGL 2.x implementations are only required to support 64 floating-point components for fragment shader uniforms.

Table 1 Uniform Limits

Program Unit	Query Enumerant	OpenGL 2.x Minimum	OpenGL 3.x Minimum
Vertex	`GL_MAX_VERTEX_UNIFORM_COMPONENTS`	512	1024
Fragment	`GL_MAX_FRAGMENT_UNIFORM_COMPONENTS`	64	1024

OpenGL 3.0 introduced a new mechanism, called *uniform buffer objects*, that allows shaders to access even larger amounts of uniform storage. Uniform buffer objects will be covered in a later chapter.

Each GLSL type has a specific number of components that it occupies. These values are listed in Table 2. Many GPUs only allow indexed array access to arrays that are stored internally as `vec4`. As a result, each element in an array can occupy as many as four components. In practice this can depend on the GPU and on the manner in which the array is accessed in all shaders. For example, *data_blob* in Figure 2 would *likely* only use 16 components. If that vertex shader were linked with the fragement shader in Figure 3, *data_blob* would almost certainly require 32 components. This means

[1] Minimum requirements for implementation limits queried by `GL_MAX_` enumerants are often called "minimum maximums".

that a particular uniform used in a shader can use a different number of components in each program where it is linked.

Table 2 Uniform Components

GLSL Type	Components	Components in Array
float, int, or bool	1	4
vec2, ivec2, or bvec2	2	4
vec3, ivec3, or bvec3	4	4
vec4, ivec4, or bvec4	4	4
mat2	4	4
mat3	12	12
mat4	16	16

Figure 2 Vertex shader with a uniform array

```
uniform vec2 data_blob[8];
void main(void)
{
    if (gl_Vertex.x < 6.0) {
        gl_Position = vec4(data_blob[4], data_blob[1]);
    } else {
        gl_Position = vec4(data_blob[2], data_blob[7]);
    }
}
```

Figure 3 Fragment shader with a uniform array

```
uniform vec2 data_blob[8];
void main(void)
{
    gl_Color = vec4(data_blob[int(7.0 * abs(sin(gl_FragPos.x))]],
                    data_blob[int(7.0 * abs(sin(gl_FragPos.y))]]);
}
```

Notice also that a `vec4` can occupy four components, and a `mat3` can occupy 12 components. This is again due to the `vec4`-centric nature of many GPUs. In some cases the compiler may place a non-array scalar and a non-array `vec3` in the single `vec4`.

Implementations are required to implement certain procedures for packing data and eliminating unused uniforms. As with vertex shader inputs (see Chapter 3), the requirements for detecting and eliminating unused uniforms only require that the compiler perform simple static analysis of the shader. This means that a uniform may be eliminated if it does not appear as an operand in the shader text. This means that *`should_eliminate`* in Figure 4 may not be eliminated! Components of uniforms that do not appear in the shader text also will be eliminated. In Figure 5 the *`z`* component of *`offset`* will be eliminated.

Compilers are not required to adhere to the rules of uniform packing for uniform arrays that are accessed with non-constant indexes. Even if the *`z`* of the 5th element of come `vec4` array is never accessed, the compiler is not required to eliminate it. At the same time, the compiler is also not required to not eliminate it.

1.2 Setting Uniform Values

Setting the value of a uniform is a two step process. After a program has been linked, uniforms are assigned numeric locations. The application must query the location associated with a particular name. The location is queried by calling `glGetUniformLocation`. The *`program`* is the name of the program object, and *`name`* is the name of the uniform whose location is to be queried. The value returned is the position of the uniform in the program. The value -1 will be returned in the case of an error.

Figure 4 Shader with unexecuted use of a uniform

```
uniform vec4 should_eliminate;
uniform mat4 mvp;
void main(void)
{
    if (false) {
        gl_Position = mvp * should_eliminate;
    } else {
        gl_Position = mvp * gl_Vertex;
    }
}
```

Figure 5 Shader with unused uniform component

```
uniform vec4 offset;
uniform mat4 mvp;
void main(void)
{
  gl_Position = mvp * gl_Vertex + vec4(offset.xyw, 0.0);
}
```

```
GLint glGetUniformLocation(GLuint program, const GLchar *name);
```

After determining the location, the value for the location can be set. There are a variety of functions for setting uniform values based on the uniform's type. For now only the three most common variants will be discussed. `glUniform4f` and `glUniform4fv` can be used to set the value of a `vec4` varying, and `glUniformMatrix4fv` can be used to set the value of a `mat4` varying.

```
void glUniform4f(GLint location,
                 GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3);

void glUniform4fv(GLint location, GLsizei count, const GLfloat *value);

void glUniformMatrix4fv(GLint location, GLsizei count,
                        GLboolean transpose, const GLfloat *value);
```

For all three functions, *`location`* is the uniform location returned by `glGetUniformLocation`. Notice that there is no explicit program object parameter. Instead the program most recently bound with `glUseProgram` is used. For `glUniform4f`, the remaining parameters are the values to set for the components of the `vec4` uniform.

For the vectored functions (i.e., the ones with `v` at the end of the name), the *`count`* parameter is the number of uniforms to set. This is used when setting values for a uniform array. For non-arrays, 1 should be used. The *`value`* is a pointer to the data to copy to the uniform. It is important that this point to the correct number of elements.

For `glUniformMatrix4fv` the *`transpose`* parameter specifies whether the supplied data is in column-major or row-major ordering. If `GL_FALSE` is passed, the first four elements of *`value`* make up the first column of the matrix. If `GL_TRUE` is passed, the first four elements of *`value`* make up the first row of the matrix.

Figure 6 shows how the *`model_view_projection`* uniform of Figure 1 could be set.

1.3 Pre-initialized Uniforms

In addition to being set through the `glUniform` family of functions, uniforms can be initialized in the shader itself[2]. This is common practice when a uniform has a typical value that only needs to be changed in infrequent circumstances.

[2] GLSL 1.20 or later is required.

Figure 6 Set a matrix uniform

```
float mvp[16];
GLint mvp_location =
    glGetUniformLocation(program, "model_view_projection");
if (mvp_location < 0)
    /* ... error path ... */

calculate_mvp_matrix(mvp);          // Matrix in column-major order.

glUseProgram(program);
glUniformMatrix4fv(mvp_location, 1, GL_FALSE, mvp);
```

The fragment shader in Figure 7 initializes `color` to the vector { 0.0, 1.0, 0.0, 1.0 }. When `prog1` in Figure 8 is used, `color` will have the value initialized in the shader. However, when `prog2` is used, `color` will have the value { 1.0, 0.0, 0.0, 1.0 }.

Figure 7 Initializing a uniform

```
uniform vec4 color = vec4(0.0, 1.0, 0.0, 1.0);
void main(void)
{
    gl_FragColor = color * gl_Color;
}
```

Figure 8 Overriding a pre-initialized uniform

```
GLuint vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSourse(vs, 1, (const GLchar **) &code_from_figure_7, NULL);
glCompileShader(vs);

prog1 = glCreateProgram();
glAttachShader(prog1, vs);
glAttachShader(prog1, fs1);    // fs1 is initialized elsewhere
glLinkProgram(prog1);

prog2 = glCreateProgram();
glAttachShader(prog2, vs);
glAttachShader(prog2, fs1);    // fs1 is initialized elsewhere
glLinkProgram(prog2);

GLint color_location = glGetUniformLocation(prog2, "color");
glUseProgram(prog2);
glUniform4f(color_location, 1.0, 0.0, 0.0, 1.0);
```

1.4 Complex Datatypes

As seen in the preceeding sections, uniforms can be any of the scalar, vector, and matrix built-in types. Uniforms can also be structures, arrays of built-in types, and arrays of structures.