Vertex Shader Inputs

by Ian Romanick

1 Introduction

This chapter delves deeper into vertex shader attributes. Both the association of an attribute index with the name used in a shader and the association of data in a buffer object with a specific attribute index will be covered.

1.1 Attribute Placement

The careful observer will notice a mismatch between the OpenGL API and the shading language. In GLSL vertex shader inputs are accessed by names. However, in the API vertex shader inputs are accessed by numerical index. Each vertex input accessed by the vertex shader is assigned an index during program linking. Inputs names that are declared but not used are not assigned indexes. Several interfaces are provided to query the index assigned to a name and to request that a name be assigned a specific index.

After a program has been linked, the location of a particular vertex shader input can be queried with `glGetAttribLocation`. This function works much like `glGetUniformLocation` introduced in chapter 2.

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

The *program* is the name of the program object, and *name* is the name of the vertex shader input whose location is to be queried. The value returned is the position of the input in the program. The value -1 will be returned in the case of an error. Some error conditions include querying the location of an non-active vertex shader input or querying the location of a reserved vertex shader input name (i.e., one that begins with *gl_*).

What is a non-active vertex shader input? Fetching vertex shader inputs from memory uses memory bandwidth and requires that the data be in memory accessible by the GPU. GLSL compilers are encouraged to "optimize out" inputs that are not actually used by the program to save these resources.

The algorithms used by GLSL compilers to determine that an input (or any other value) is not used are complex and vary from compiler to compiler. At the very least, compilers will eliminate inputs that are not used as the source of any operation. In Figure 1 the input *normal* would be eliminated. Querying its location with `glGetAttribLocation` would therefore return -1.

Figure 1 Vertex shader with an unused input

```
uniform mat4 mvp;
attribute vec4 position;
attribute vec3 normal;

void main(void)
{
    gl_Position =  mvp * position;
}
```

A GLSL compiler may or may not be able to determine that *color* is unused in Figure 2. In general, if the compiler can determine that an input is unused without any knowledge of the semantics of program statements, the input will probably be eliminated. In case of Figure 2 the compiler must understand the semantics of `cos` and know that cosine of 120 degrees is -0.5.

Figure 2 Vertex shader with an unused input

```
uniform mat4 mvp;
varying vec3 result_color;
attribute vec4 position;
attribute vec3 color;

void main(void)
{
    if (cos(120.0) > 0.0) {
        result_color = color;
    } else {
        result_color = vec3(0.0, 1.0, 0.0);
    }

    gl_Position =  mvp * position;
}
```

The number of active vertex shader inputs can be queried by calling `glGetProgramiv` with the parameter `GL_ACTIVE_ATTRIBUTES`.

The `glGetActiveAttrib` can be used query information about each active attribute. The *`progr-am`* and *`index`* select the linked program and the input index to be queried. This index value bears no relation

The name of the specified active attribute is stored in the buffer pointed to by *`name`*. At most *`bufSize`* bytes, including the `NUL`-terminator will be written. The actual number of bytes written, not including the `NUL`-terminator, will be stored in *`length`*. If *`length`* is `NULL`, this value will not be stored.

```
void glGetActiveAttrib(GLuint program, GLuint index, GLsizei bufSize,
                       GLsizei *length, GLint *size, GLenum *type,
                       GLchar *name);
```

The *`size`* and *`type`* parameters are not at all related to the *`size`* and *`type`* parameters of `glVerte-xAttribPointer`. The value stored in the storage pointed to by the *`type`* parameter is an encoding of the GLSL data type. Table 1 shows a complete type of the possible values.

Table 1 Enumerants for GLSL types

GLSL Type	Enumerant	GLSL version
float	`GL_FLOAT`	
vec2	`GL_FLOAT_VEC2`	
vec3	`GL_FLOAT_VEC3`	
vec4	`GL_FLOAT_VEC4`	
mat2	`GL_FLOAT_MAT2`	
mat3	`GL_FLOAT_MAT3`	
mat4	`GL_FLOAT_MAT4`	
mat2x3	`GL_FLOAT_MAT2x3`	1.20 or later
mat2x4	`GL_FLOAT_MAT2x4`	1.20 or later
mat3x2	`GL_FLOAT_MAT3x2`	1.20 or later
mat3x4	`GL_FLOAT_MAT3x4`	1.20 or later
mat4x2	`GL_FLOAT_MAT4x2`	1.20 or later
mat4x3	`GL_FLOAT_MAT4x3`	1.20 or later

The value stored in the storage pointed to by the *`size`* parameter is the number of elements of the type returned in *`type`* are used by the input. Since vertex shader inputs cannot be arrays (as of GLSL version 1.40 at least), this value will always be 1.

The length of the longest name of any active input can be queried by calling `glGetProgramiv` with the parameter `GL_ACTIVE_ATTRIBUTE_MAX_LENGTH`. Figure 3 combines all of these elements to display information about the set of active vertex shader inputs. The function `get_name_of_GLSL_-type` is supplied by the application to convert on of the enumerant values in Table 1 to a text string.

Figure 3 Display information about active vertex shader inputs

```
char *name;
GLint active_attribs, max_length;

glGetProgramiv(prog, GL_ACTIVE_ATTRIBUTES, &active_attribs);
glGetProgramiv(prog, GL_ACTIVE_ATTRIBUTE_MAX_LENGTH, &max_length);

name = malloc(max_length + 1);

for (unsigned i = 0; i < active_attribs; i++) {
    GLint size;
    GLenum type;

    glGetActiveAttrib(prog, i, max_length + 1, NULL,
                      &size, &type, name);
    printf("%s %s is at location %d\n", get_name_of_GLSL_type(type),
           name, glGetAttribLocation(prog, name));
}
free(name);
```

In reality, these interfaces are rarely used by applications. By allowing the compiler to place inputs and eliminate unused inputs, applications would have to track whether or not each input was active along with its location. This is further complicated in cases where multiple vertex shaders are linked in non-trivial ways.

Instead, OpenGL provides an API for applications to instruct the compiler where to place specific attributes. Since the application no longer needs to query the locations of inputs, the application code can be greatly simplified. This allows the application to put attributes with a specific semantic at a specific index, for example.

Names are bound to a specific index by calling `glBindAttribLocation`. The *`program`* and *`index`* select the *unlinked* program and the input index to be set. The input name to be associated with *`index`* is specified with *`name`*.

```
void glBindAttribLocation(GLuint program, GLuint index, const GLchar *name);
```

If the input specified by *`name`* is a matrix, the columns of the matrix will be associated with attributes *`index`* through *`index`* + *`n`* - 1, where *`n`* is the number of columns of the matrix. A `mat4` will use 4 attributes, and a `mat3x4` will use 3 attributes.

Calling `glBindAttribLocation` will generate an error if *`index`* (or *`index`* + *`n`* - 1 in the case of a matrix) is greater than or equal to the maximum number of vertex shader inputs supported by the implementation. This value can be queried by calling `glGetIntegerv` with the parameter *`GL_MAX_VERTEX_ATTRIBS`*.

It is also invalid to use `glBindAttribLocation` to bind the location of a built-in input name (i.e., any name beginning with *`gl_`*). [1]

It is very important to note that the effects of calling `glBindAttribLocation` do not take effect until the next time the program is linked. The program snippet in Figure 4 will print 2, not 3!

Figure 4 Input location settings only take effect after linking

```
glBindAttribLocation(prog, 2, "color");
glBindAttribLocation(prog, 3, "secondary_color");
glLinkProgram(prog);
glBindAttribLocation(prog, 3, "color");

GLint loc = glGetAttribLocation(prog, "color");
printf("%d\n", loc);
```

[1] With the exception of *`gl_Vertex`*, these names are associated with data through a different interface. These interfaces were removed in OpenGL 3.1 (and OpenGL ES 2.0) and should be avoided in new programs.

If the program in Figure 4 were relinked after calling glBindAttribLocation in line 4, the *color* input would be bound to index 3, and *secondary_color* would not be bound to index 3. In this case *secondary_color* would automatically be assigned a location by the linker. It is not possible to bind multiple names to the same index, nor is it possible to bind multiple indexes to the same name.

1.2 Attribute Data

As was introduced in chapter 2, data is associated with a particular attribute index by calling glVertexAttribPointer.

```
void glVertexAttribPointer(GLuint index, GLint size, GLenum type,
                           GLboolean normalized, GLsizei stride,
                           const GLvoid *pointer);
```

The *size* and *type* specified to glVertexAttribPointer do not need to match the type of the input declared in the shader. It is very common for per-vertex color data to be stored as triples of normalized unsigned bytes but used in the shader as a vec4.

What happens when the numbers of data elements do not match? In the previous example the data supplied has three elements, but the declaration has four. If the declared input has less elements than the source data, the extra data is simply dropped. If the declared input has more elements than the source data, the missing *x*, *y* and *z* fields of the shader variable are initialized to 0.0, and the *w* field is initialized to 1.0.

Listing the *x* field as a possible missing parameter in the previous paragraph is not a mistake. If the attribute has been disabled by calling glDisableVertexAttribArray, the vertex shader input will take the default value { 0.0, 0.0, 0.0, 1.0 }. Commands from the immediate mode API, such as glVertexAttrib4f, can change the default value used when the attribute array is disabled. [2]

1.3 Changes in OpenGL 3.0 and GLSL 1.30

Several subtle changes were made to vertex shader inputs in GLSL 1.30. In GLSL 1.30 the attribute keyword was deprecated, and it was removed in GLSL 1.40. Shaders using GLSL 1.30 can use in in place of attribute. In GLSL 1.40 and later the use of in is required, and using attribute will generate an error.

GLSL 1.30 also adds true integer support. This allows vertex shader inputs to be signed or unsigned integers and signed integer vectors. Table 2 list the values that are added to Table 1 as possible values returned by glGetActiveAttrib in the *type* parameter.

Table 2 Enumerants for GLSL 1.30 types

GLSL Type	Enumerant	GLSL version
int	GL_INT	1.30 or later
ivec2	GL_INT_VEC2	1.30 or later
ivec3	GL_INT_VEC3	1.30 or later
ivec4	GL_INT_VEC4	1.30 or later
uint	GL_UNSIGNED_INT	1.30 or later
uivec2	GL_UNSIGNED_INT_VEC2	1.30 or later
uivec3	GL_UNSIGNED_INT_VEC3	1.30 or later
uivec4	GL_UNSIGNED_INT_VEC4	1.30 or later

[2] These interfaces were removed in OpenGL 3.1 (and OpenGL ES 2.0) and should be avoided in new programs.