In the examples you've seen so far, either we have used hard-coded data directly in our shaders, or we have passed values to shaders one at a time. While sufficient to demonstrate the configuration of the OpenGL pipeline, this is hardly representative of modern graphics programming. Recent graphics processors are designed as streaming processors that consume and produce huge amounts of data. Passing a few values to OpenGL at a time is extremely inefficient. To allow data to be stored and accessed by OpenGL, we include two main forms of data storage—buffers and textures. In this chapter, we first introduce buffers, which are linear blocks of untyped data and can be seen as generic memory allocations. Next, we introduce textures, which are normally used to store multidimensional data, such as images or other data types.

Buffers

In OpenGL, buffers are linear allocations of memory that can be used for a number of purposes. They are represented by *names*, which are essentially opaque handles that OpenGL uses to identify them. Before you can start using buffers, you have to ask OpenGL to reserve some names for you and then use them to allocate memory and put data into that memory. The memory allocated for a buffer object is called its *data store*. The data store of the buffer is where OpenGL stores its data. You can put data into the buffer using OpenGL commands, or you can *map* the buffer object, which means that you can get a pointer that your application can use to write directly into (or read directly out of) the buffer.

Once you have the name of a buffer, you can attach it to the OpenGL context by *binding* it to a buffer binding point. Binding points are sometimes referred to as *targets*; these terms may be used interchangeably.[1] There are a large number of buffer binding points in OpenGL and each has a different use, although the buffers you bind to them are the same. For example, you can use the contents of a buffer to automatically supply the inputs of a vertex shader, to store the values of variables that will be used by your shaders, or as a place for shaders to store the data they produce. You can even use the same buffer for multiple purposes at the same time.

1. It's not technically correct to conflate *target* and *binding point*, because a single target may have multiple binding points. However, for most use cases, it is well understood what is meant.

Creating Buffers and Allocating Memory

Before you can ask OpenGL to allocate memory, you need to create a buffer object to represent that allocation. Like most objects in OpenGL, buffer objects are represented by a GLuint variable, which is generally called its *name*. One or more buffer objects can be created using the **glCreateBuffers()** function, whose prototype is

Click here to view code image

```
void glCreateBuffers(GLsizei n, GLuint* buffers);
```

The first parameter to **glCreateBuffers()**, `n`, is the number of buffer objects to create. The second parameter, `buffers`, is the address of the variable or variables that will be used to store the names of the buffer objects. If you need to create only one buffer object, set `n` to 1 and set `buffers` to the address of a single GLuint variable. If you need to create more than one buffer at a time, simply set `n` to that number and point `buffers` to the beginning of an array of at least `n` GLuint variables. OpenGL will just trust that the array is big enough and will write that many buffer names to the pointer that you specify.

Each of the names you get back from **glCreateBuffers()** represents a single buffer object. You can bind the buffer objects to the current OpenGL context by calling **glBindBuffer()**, the prototype of which is

Click here to view code image

```
void glBindBuffer(GLenum target, GLuint buffer);
```

Before you can actually use the buffer objects, you need to allocate their *data stores*, which is another term for the memory represented by the buffer object. The functions that are used to allocate memory using a buffer object are **glBufferStorage()** and **glNamedBufferStorage()**. Their prototypes are

Click here to view code image

```
void glBufferStorage(GLenum target,
                     GLsizeiptr size,
                     const void* data,
                     GLbitfield flags);
void glNamedBufferStorage(GLuint buffer,
                          GLsizeiptr size,
                          const void* data,
                          GLbtifield flags);
```

The first function affects the buffer object bound to the binding point specified by `target`; the second function directly affects the buffer specified by `buffer`. The remainder of the parameters serve the same purpose in both functions. The `size` parameter specifies how big the storage region is to be, in bytes. The `data` parameter

is used to pass a pointer to any data that you want to initialize the buffer with. If this is `NULL`, then the storage associated with the buffer object will at first be uninitialized. The final parameter, `flags`, is used to tell OpenGL how you're planning to use the buffer object.

Once storage has been allocated for a buffer object using either **glBufferStorage()** or **glNamedBufferStorage()**, it cannot be reallocated or respecified, but is considered *immutable*. To be clear, the contents of the buffer object's data store can be changed, but its size or usage flags may not. If you need to resize a buffer, you need to delete it, create a new one, and set up new storage for that.

The most interesting parameter to these two functions is the `flags` parameter. This should give OpenGL enough information to allocate memory suitable for your intended purpose and allow it to make an informed decision about the storage requirements of the buffer. `flags` is a GLbitfield type, which means that it's a combination of one or more bits. The flags that you can set are shown in Table 5.1.

Flag	Description
`GL_DYNAMIC_STORAGE_BIT`	Buffer contents can be updated directly.
`GL_MAP_READ_BIT`	Buffer data store will be mapped for reading.
`GL_MAP_WRITE_BIT`	Buffer data store will be mapped for writing.
`GL_MAP_PERSISTENT_BIT`	Buffer data store can be mapped persistently.
`GL_MAP_COHERENT_BIT`	Buffer maps are to be coherent.
`GL_CLIENT_STORAGE_BIT`	If all other conditions can be met, prefer storage local to the client (CPU) rather than to the server (GPU).

Table 5.1: Buffer Storage Flags

The flags listed in Table 5.1 may seem a little terse and probably deserve more explanation. In particular, the absence of certain flags can mean something to OpenGL, some flags may be used only in combination with others, and the specification of these flags can have an effect on what you're allowed to do with the buffer later. We'll provide a brief explanation of each of these flags here and then dive deeper into some of their meanings as we cover further functionality.

First, the `GL_DYNAMIC_STORAGE_BIT` flag is used to tell OpenGL that you mean to update the contents of the buffer directly—perhaps once for every time that you use the

data. If this flag is not set, OpenGL will assume that you're not likely to need to change the contents of the buffer and might put the data somewhere that is less accessible. If you don't set this bit, you won't be able to use commands like **glBufferSubData()** to update the buffer content, although you will be able to write into it directly from the GPU using other OpenGL commands.

The mapping flags `GL_MAP_READ_BIT`, `GL_MAP_WRITE_BIT`, `GL_MAP_PERSISTENT_BIT`, and `GL_MAP_COHERENT_BIT` tell OpenGL if and how you're planning to map the buffer's data store. Mapping is the process of getting a pointer that you can use from your application that represents the underlying data store of the buffer. For example, you may map the buffer for read or write access only if you specify the `GL_MAP_READ_BIT` or `GL_MAP_WRITE_BIT` flags, respectively. Of course, you can specify both if you wish to map the buffer for both reading and writing.

If you specify `GL_MAP_PERSISTENT_BIT`, then this flag tells OpenGL that you wish to map the buffer and then *leave it mapped* while you call other drawing comands. If you don't set this bit, then OpenGL requires that you don't have the buffers mapped while you're using it from drawing commands. Supporting *peristent maps* might come at the expense of some performance, so it's best not to set this bit unless you really need to. The final bit, `GL_MAP_COHERENT_BIT`, goes further and tells OpenGL that you want to be able to share data quite tightly with the GPU. If you don't set this bit, you need to tell OpenGL when you've written data into the buffer, even if you don't unmap it.

Click here to view code image

```
// The type used for names in OpenGL is GLuint
GLuint buffer;

// Create a buffer
glCreateBuffers(1, &buffer);

// Specify the data store parameters for the buffer
glNamedBufferStorage(
                buffer,             // Name of the buffer
                1024 * 1024,        // 1 MiB of space
                NULL,               // No initial data
                GL_MAP_WRITE_BIT);  // Allow map for writing

// Now bind it to the context using the GL_ARRAY_BUFFER binding point
glBindBuffer(GL_ARRAY_BUFFER, buffer);
```

Listing 5.1: Creating and initializing a buffer

After the code in Listing 5.1 has executed, `buffer` contains the name of a buffer object that has been initialized to represent one megabyte of storage for whatever data we choose. Using the `GL_ARRAY_BUFFER` target to refer to the buffer object suggests to

OpenGL that we're planning to use this buffer to store vertex data, but we'll still be able to take that buffer and bind it to some other target later. There are a handful of ways to get data into the buffer object. You may have noticed the `NULL` pointer that we pass as the third argument to **glNamedBufferStorage()** in Listing 5.1. Had we instead supplied a pointer to some data, that data would have been used to initialize the buffer object. Using this pointer, however, allows us to set only the initial data to be stored in the buffer.

Another way get data into a buffer is to give the buffer to OpenGL and tell it to copy data there. This allows you to dynamically update the content of a buffer after it has already been initialized. To do this, we call either **glBufferSubData()** or **glNamedBufferSubData()**, passing the size of the data we want to put into the buffer, the offset in the buffer where we want it to go, and a pointer to the data in memory that should be put into the buffer. **glBufferSubData()** and **glNamedBufferSubData()** are declared as follows:

Click here to view code image

```
void glBufferSubData(GLenum target,
                     GLintptr offset,
                     GLsizeiptr size,
                     const GLvoid * data);
void glNamedBufferSubData(GLuint buffer,
                          GLintptr offset,
                          GLsizeiptr size,
                          const void * data);
```

To update a buffer object using **glBufferSubData()**, you must have told OpenGL that you want to put data into it that way. To do this, include `GL_DYNAMIC_STORAGE_BIT` in the `flags` parameter to **glBufferStorage()** or **glNamedBufferStorage()**. Like **glBufferStorage()** and **glNamedBufferStorage()**, **glBufferSubData()** affects the buffer bound to the binding point specified by `target`, and **glNamedBufferSubData()** affects the buffer object specified by `buffer`. Listing 5.2 shows how we can put the data originally used in Listing 3.1 into a buffer object, which is the first step in automatically feeding a vertex shader with data.

Click here to view code image

```
// This is the data that we will place into the buffer object
static const float data[] =
{
     0.25, -0.25, 0.5, 1.0,
    -0.25, -0.25, 0.5, 1.0,
     0.25, 0.25, 0.5, 1.0
};
```

```
// Put the data into the buffer at offset zero
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(data), data);
```

Listing 5.2: Updating the content of a buffer with **glBufferSubData()**

Another method for getting data into a buffer object is to ask OpenGL for a pointer to the memory that the buffer object represents and then copy the data there yourself. This is known as *mapping* the buffer. Listing 5.3 shows how to do this using the **glMapNamedBuffer()** function.

Click here to view code image

```
// This is the data that we will place into the buffer object
static const float data[] =
{
     0.25, -0.25, 0.5, 1.0,
    -0.25, -0.25, 0.5, 1.0,
     0.25, 0.25, 0.5, 1.0
};

// Get a pointer to the buffer's data store
void * ptr = glMapNamedBuffer(buffer, GL_WRITE_ONLY);

// Copy our data into it...
memcpy(ptr, data, sizeof(data));

// Tell OpenGL that we're done with the pointer
glUnmapNamedBuffer(GL_ARRAY_BUFFER);
```

Listing 5.3: Mapping a buffer's data store with **glMapNamedBuffer()**

As with many other buffer functions in OpenGL, there are two versions—one that affects the buffer bound to one of the targets of the current context, and one that operates directly on a buffer whose name you specify. Their prototypes are

Click here to view code image

```
void *glMapBuffer(GLenum target,
                  GLenum usage);
void *glMapNamedBuffer(GLuint buffer,
                       GLenum usage);
```

To unmap the buffer, we call either **glUnmapBuffer()** or **glUnmapNamedBuffer()**, as shown in Listing 5.3. Their prototypes are

Click here to view code image

```
void glUnmapBuffer(GLenum target);
void glUnmapNamedBuffer(GLuint buffer);
```

Mapping a buffer is useful if you don't have all the data handy when you call the function. For example, you might be about to generate the data, or to read it from a file. If you wanted to use **glBufferSubData()** (or the initial pointer passed to **glBufferData()**), you'd have to generate or read the data into temporary memory and then get OpenGL to make another copy of the data into the buffer object. If you map a buffer, you can simply read the contents of the file directly into the mapped buffer. When you unmap it, if OpenGL can avoid making a copy of the data, it will. Regardless of whether we used **glBufferSubData()** or **glMapBuffer()** and an explicit copy to get data into our buffer object, it now contains a copy of `data[]` and we can use it as a source of data to feed our vertex shader.

The **glMapBuffer()** and **glMapNamedBuffer()** functions can sometimes be a little heavy handed. They map the entire buffer, and do not provide any information about the type of mapping operation to be performed besides the `usage` parameter. Even that serves only as a hint. A more surgical approach can be taken by calling either **glMapBufferRange()** or **glMapNamedBufferRange()**, whose prototypes are

Click here to view code image

```
void *glMapBufferRange(GLenum target,
                       GLintptr offset,
                       GLsizeiptr length,
                       GLbitfield access);

void *glMapNamedBufferRange(GLuint buffer,
                            GLintptr offset,
                            GLsizeiptr length,
                            GLbitfield access);
```

As with the **glMapBuffer()** and **glMapNamedBuffer()** functions, there are two versions of these functions—one that affects a currently bound buffer and one that affects a directly specified buffer object. These functions, rather than mapping the entire buffer object, map only a specific range of the buffer object. This range is given using the `offset` and `length` parameters. The `parameter` contains flags that tell OpenGL how the mapping should be performed. These flags can be a combination of any of the bits listed in Table 5.2.

Flag	Description
GL_MAP_READ_BIT	Buffer data store will be mapped for reading.
GL_MAP_WRITE_BIT	Buffer data store will be mapped for writing.
GL_MAP_PERSISTENT_BIT	Buffer data store can be mapped persistently.
GL_MAP_COHERENT_BIT	Buffer maps are to be coherent.
GL_MAP_INVALIDATE_RANGE_BIT	Tells OpenGL that you no longer care about the data in the specified range.
GL_MAP_INVALIDATE_BUFFER_BIT	Tells OpenGL that you no longer care about any of the data in the whole buffer.
GL_MAP_FLUSH_EXPLICIT_BIT	You promise to tell OpenGL about any data you modify inside the mapped range.
GL_MAP_UNSYNCHRONIZED_BIT	Tells OpenGL that you will perform any synchronization yourself.

Table 5.2: Buffer-Mapping Flags

As with the bits that you can pass to **glBufferStorage()**, these bits can control some advanced functionality of OpenGL and, in some cases, their correct usage depends on other OpenGL functionality. However, these bits are *not* hints and OpenGL will enforce their correct usage. You should set `GL_MAP_READ_BIT` if you plan to read from the buffer and `GL_MAP_WRITE_BIT` if you plan to write to it. Reading or writing into the mapped range without setting the appropriate bits is an error. The `GL_MAP_PERSISTENT_BIT` and `GL_MAP_COHERENT_BIT` flags have similar meanings to their identically named counterparts in **glBufferStorage()**. All four of these bits are required to match between when you specify storage and when you request a mapping. That is, if you want to map a buffer for reading using the `GL_MAP_READ_BIT` flag, then you must also specify the `GL_MAP_READ_BIT` flag when you call **glBufferStorage()**.

We'll dig deeper into the remaining flags when we cover synchronization primitives a little later in the book. However, because of the additional control and stronger contract provided by **glMapBufferRange()** and **glMapNamedBufferRange()**, it is

generally preferred to call these functions rather than **glMapNamedBuffer()** (or **glMapBuffer()**). You should get into the habbit of using these functions even if you're not using any of their more advanced features.

Filling and Copying Data in Buffers

After allocating storage space for your buffer object using **glBufferStorage()**, one possible next step is to fill the buffer with known data. Whether you use the initial `data` parameter of **glBufferStorage()**, use **glBufferSubData()** to put the initial data in the buffer, or use **glMapBufferRange()** to obtain a pointer to the buffer's data store and fill it with your application, you will need to get the buffer into a known state before you can use it productively. If the data you want to put into a buffer is a constant value, it is probably much more efficient to call **glClearBufferSubData()** or **glClearNamedBufferSubData()**, whose prototypes are

Click here to view code image

```
void glClearBufferSubData(GLenum target,
                          GLenum internalformat,
                          GLintptr offset,
                          GLsizeiptr size,
                          GLenum format,
                          GLenum type,
                          const void * data);

void glClearNamedBuffeSubData(GLuint buffer,
                              GLenum internalformat,
                              GLintptr offset,
                              GLsizeiptr size,
                              GLenum format,
                              GLenum type,
                              const void * data);
```

These functions take a pointer to a variable containing the values that you want to clear the buffer object to and, after converting it to the format specified in `internalformat`, replicate the data across the range of the buffer's data store specified by `offset` and `size`, both of which are measured in bytes. `format` and `type` tell OpenGL about the data pointed to by `data`. The `format` can be one of `GL_RED`, `GL_RG`, `GL_RGB`, or `GL_RGBA` to specify one-, two-, three-, or four-channel data, for example. Meanwhile, `type` should represent the data type of the components. For instance, it could be `GL_UNSIGNED_BYTE` or `GL_FLOAT` to specify unsigned bytes or floating-point data, respectively. The most common types supported by OpenGL and their corresponding C data types are listed in Table 5.3.

Type Token	C Type
GL_BYTE	GLchar
GL_UNSIGNED_BYTE	GLuchar
GL_SHORT	GLshort
GL_UNSIGNED_SHORT	GLushort
GL_INT	GLint
GL_UNSIGNED_INT	GLuint
GL_FLOAT	GLfloat
GL_DOUBLE	GLdouble

Table 5.3: Basic OpenGL Type Tokens and Their Corresponding C Types

Once your data has been sent to the GPU, it's entirely possible you may want to share that data between buffers or copy the results from one buffer into another. OpenGL provides an easy-to-use way of doing that. **glCopyBufferSubData()** and **glCopyNamedBufferSubData()** let you specify which buffers are involved as well as the size and offsets to use.

Click here to view code image

```
void glCopyBufferSubData(GLenum readtarget,
                         GLenum writetarget,
                         GLintptr readoffset,
                         GLintptr writeoffset,
                         GLsizeiptr size);
```

Click here to view code image

```
void glCopyNamedBufferSubData(GLuint readBuffer,
                              GLuint writeBuffer,
                              GLintptr readOffset,
                              GLintptr writeOffset,
                              GLsizeiptr size);
```

For **glCopyBufferSubData()**, the `readtarget` and `writetarget` are the targets where the two buffers you want to copy data between are bound. They can be buffers bound to any of the available buffer binding points. However, since buffer binding points can have only one buffer bound at a time, you couldn't copy between two buffers that are both bound to the `GL_ARRAY_BUFFER` target, for example. Thus, when you perform the copy, you need to pick two targets to bind the buffers to, which will disturb the OpenGL state.

To resolve this, OpenGL provides the `GL_COPY_READ_BUFFER` and `GL_COPY_WRITE_BUFFER` targets. These targets were added specifically to allow you to copy data from one buffer to another without any unintended side effects. Because

they are not used for anything else in OpenGL, you can bind your read and write buffers to these binding points without affecting any other buffer target.

Alternatively, you can use the **glCopyNamedBufferSubData()** form, which takes the names of the two buffers directly. Of course, you can specify the same buffer for both `readBuffer` and `writeBuffer` to copy a region of data between two offsets in the same buffer object. Be careful that the regions to be copied don't overlap, though, as in this case the results of the copy are undefined. You can consider **glCopyNamedBufferSubData()** as a form of the C function `memcpy` for buffer objects.

The `readoffset` and `writeoffset` parameters tell OpenGL where in the source and destination buffers to read or write the data, and the `size` parameter tells it how big the copy should be. Be sure that the ranges you are reading from and writing to remain within the bounds of the buffers; otherwise, your copy will fail.

You may notice the types of `readoffset`, `writeoffset`, and `size`, which are `GLintptr` and `GLsizeiptr`. These types are special definitions of integer types that are at least wide enough to hold a pointer variable.

Feeding Vertex Shaders from Buffers

In Chapter 2, "Our First OpenGL Program," you were briefly introduced to the vertex array object (VAO). During that discussion, we explained how the VAO represented the inputs to the vertex shader—though at the time, we didn't use any real inputs to our vertex shaders and opted instead for hard-coded arrays of data. Then, in Chapter 3 we introduced the concept of *vertex attributes*, but we discussed only how to change their static values. Although the vertex array object stores these static attribute values for you, it can do a lot more. Before we can proceed, we need to create a vertex array object to store our vertex array state and bind it to our context so that we can use it:

Click here to view code image

```
GLuint vao;
glCreateVertexArrays(1, &vao);
glBindVertexArray(vao);
```

Now that we have our VAO created and bound, we can start filling in its state. Rather than using hard-coded data in the vertex shader, we can instead rely entirely on the value of a vertex attribute and ask OpenGL to fill it automatically using the data stored in a buffer object that we supply. Each vertex attribute gets to fetch data from a buffer bound to one of several *vertex buffer bindings*. To set the binding that a vertex attribute uses to reference a buffer, call the **glVertexArrayAttribBinding()** function:

Click here to view code image

```
void glVertexArrayAttribBinding(GLuint vaobj,
```

```
                                GLuint attribindex,
                                GLuint bindingindex);
```

The **glVertexArrayAttribBinding()** function tells OpenGL that when the vertex array object named `vaobj` is bound, the vertex attribute at the index specified in `attribindex` should source its data from the buffer bound at `bindingindex`. To tell OpenGL which buffer object our data is in and where in that buffer object the data resides, we use the **glVertexArrayVertexBuffer()** function to bind a buffer to one of the vertex buffer bindings. We use the **glVertexArrayAttribFormat()** function to describe the layout and format of the data, and finally we enable automatic filling of the attribute by calling **glEnableVertexAttribArray()**. The prototype of **glVertexArrayVertexBuffer()** is

Click here to view code image

```
void glVertexArrayVertexBuffer(GLuint vaobj,
                               GLuint bindingindex,
                               GLuint buffer,
                               GLintptr offset,
                               GLsizei stride);
```

Here, the first parameter is the vertex array object whose state you're modifying. The second parameter, `bindingindex`, is the index of the vertex buffer, which matches the parameter sent to **glVertexArrayAttribBinding()**. The `buffer` parameter specifies the name of the buffer object that we're binding. The last two parameters, `offset` and `stride`, tell OpenGL where in the buffer object the attribute data lies. `offset` says where the first vertex's data starts and `stride` says how far apart each vertex is. Both are measured in bytes.

Next, we have **glVertexArrayAttribFormat()**, whose prototype is

Click here to view code image

```
void glVertexArrayAttribFormat(GLuint vaobj,
                               GLuint attribindex,
                               GLint size,
                               GLenum type,
                               GLboolean normalized,
                               GLuint relativeoffset);
```

For **glVertexArrayAttribFormat()**, the first parameter is again the vertex array whose state we're modifying. `attribindex` is the index of the vertex attribute. You can define a large number of attributes as input to a vertex shader and then refer to them by their index, as explained in the "Vertex Attributes" section in Chapter 3. `size` is the number of components that are stored in the buffer for each vertex and `type` is the type of the data, which would normally be one of the types in Table 5.3.

The `normalized` parameter tells OpenGL whether the data in the buffer should be normalized (scaled between 0.0 and 1.0) before being passed to the vertex shader or if it should be left alone and passed as is. This parameter is ignored for floating-point data, but for integer data types, such as `GL_UNSIGNED_BYTE` or `GL_INT`, it is important. For example, if `GL_UNSIGNED_BYTE` data is normalized, it is divided by 255 (the maximum value representable by an unsigned byte) before being passed to a floating-point input to the vertex shader. The shader will therefore see values of the input attribute between 0.0 and 1.0. However, if the data is not normalized, it is simply cast to floating-point values and the shader will receive numbers between 0.0 and 255.0, even though the input to the vertex shader consists of floating-point data.

The `stride` parameter tells OpenGL how many bytes are between the start of one vertex's data and the start of the next, but you can set this parameter to 0 to let OpenGL calculate it for you based on the values of `size` and `type`. Finally, `relativeoffset` is the offset from the vertex's data where the specific attribute's data starts. This all seems pretty complex, but the pseudocode to compute the location in a buffer object is fairly simple:

Click here to view code image

```
location = binding[attrib.binding].memory + // Start of data store in
memory
           binding[attrib.binding].offset + // Offset of vertex attribute
in buffer
           binding[attrib.binding].stride * vertex.index + // Start of
*this* vertex
           vertex.relative_offset;          // Start of attribute relative
to vertex
```

Finally, **glEnableVertexAttribArray()** and the converse **glDisableVertexAttribArray()** have the prototypes:

Click here to view code image

```
void glEnableVertexAttribArray(GLuint index);
```

When a vertex attribute is enabled, OpenGL will feed data to the vertex shader based on the format and location information you've provided with **glVertexArrayVertexBuffer()** and **glVertexArrayAttribFormat()**. When the attribute is disabled, the vertex shader will be provided with the static information you provide with a call to **glVertexAttrib*()**.

Listing 5.4 shows how to use **glVertexArrayVertexBuffer()** and **glVertexArrayAttribFormat()** to configure a vertex attribute. Notice that we also call **glEnableVertexArrayAttrib()** after setting up the offset, stride, and format information. This tells OpenGL to use the data in the buffer to fill the vertex attribute

rather than using data we provide through one of the **glVertexAttrib*()** functions.

Click here to view code image

```
// First, bind a vertex buffer to the VAO
glVertexArrayVertexBuffer(vao,                  // Vertex array object
                          0,                    // First vertex buffer
binding
                          buffer,               // Buffer object
                          0,                    // Start from the
beginning
                          sizeof(vmath::vec4)); // Each vertex is one vec4

// Now, describe the data to OpenGL, tell it where it is, and turn on
automatic
// vertex fetching for the specified attribute
glVertexArrayAttribFormat(vao,                  // Vertex array object
                          0,                    // First attribute
                          4,                    // Four components
                          GL_FLOAT,             // Floating-point data
                          GL_FALSE,             // Normalized - ignored
for floats
                          0);                   // First element of the
vertex

glEnableVertexArrayAttrib(vao, 0);
```

Listing 5.4: Setting up a vertex attribute

After Listing 5.4 has been executed, OpenGL will automatically fill the first attribute in the vertex shader with data it has read from the buffer that was bound to the VAO by **glVertexArrayVertexBuffer()**.

We can modify our vertex shader to use only its input vertex attribute rather than a hard-coded array. This updated shader is shown in Listing 5.5.

Click here to view code image

```
#version 450 core

layout (location = 0) in vec4 position;

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

Listing 5.5: Using an attribute in a vertex shader

As you can see, the shader of Listing 5.5 is greatly simplified over the original shader shown in Chapter 2. Gone is the hard-coded array of data. As an added bonus, this

shader can be used with an arbitrary number of vertices. You can literally put millions of vertices' worth of data into your buffer object and draw them all with a single command such as a call to **glDrawArrays()**.

If you are done using data from a buffer object to fill a vertex attribute, you can disable that attribute again with a call to **glDisableVertexAttribArray()**, whose prototype is

Click here to view code image

```
void glDisableAttribArray(GLuint index);
```

Once you have disabled the vertex attribute, it goes back to being static and passing the value you specify with **glVertexAttrib*()** to the shader.

Using Multiple Vertex Shader Inputs

As you have learned, you can get OpenGL to feed data into your vertex shaders and use data you've placed in buffer objects. You can also declare multiple inputs to your vertex shaders, and assign each one a unique location that can be used to refer to it. Combining these things together means that you can get OpenGL to provide data to multiple vertex shader inputs simultaneously. Consider the input declarations to a vertex shader shown in Listing 5.6.

Click here to view code image

```
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;
```

Listing 5.6: Declaring two inputs to a vertex shader

If you have a linked program object whose vertex shader has multiple inputs, you can determine the locations of those inputs by calling

Click here to view code image

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

Here, `program` is the name of the program object containing the vertex shader and `name` is the name of the vertex attribute. In our example declarations of Listing 5.6, passing `"position"` to **glGetAttribLocation()** will cause it to return 0, and passing `"color"` will cause it to return 1. Passing something that is not the name of a vertex shader input will cause **glGetAttribLocation()** to return −1. Of course, if you always specify locations for your vertex attributes in your shader code, then **glGetAttribLocation()** should return whatever you specified. If you don't specify locations in shader code, OpenGL will assign locations for you, and those locations

will be returned by **glGetAttribLocation()**.

There are two ways to connect vertex shader inputs to your application's data, referred to as *separate attributes* and *interleaved attributes*. When attributes are separate, they are located either in different buffers or at least at different locations in the same buffer. For example, if you want to feed data into two vertex attributes, you could create two buffer objects, bind each to a different vertex buffer binding with a call to **glVertexArrayVertexBuffer()**, and then specify the two indices of the two vertex buffer binding points that you used when you call **glVertexArrayAttribBinding()** for each. Alternatively, you could place the data at different offsets within the same buffer, bind it to a single vertex buffer binding with one call to **glVertexArrayVertexBuffer()**, and then call **glVertexArrayAttribBinding()** for both attributes, passing the same binding index to each. Listing 5.7 shows this approach.

Click here to view code image

```
GLuint buffer[2];
GLuint vao;

static const GLfloat positions[] = { ... };
static const GLfloat colors[] = { ... };

// Create the vertex array object
glCreateVertexArrays(1, &vao)

// Get create two buffers
glCreateBuffers(2, &buffer[0]);

// Initialize the first buffer
glNamedBufferStorage(buffer[0], sizeof(positions), positions, 0);

// Bind it to the vertex array - offset zero, stride = sizeof(vec3)
glVertexArrayVertexBuffer(vao, 0, buffer[0], 0, sizeof(vmath::vec3));

// Tell OpenGL what the format of the attribute is
glVertexArrayAttribFormat(vao, 0, 3, GL_FLOAT, GL_FALSE, 0);

// Tell OpenGL which vertex buffer binding to use for this attribute
glVertexArrayAttribBinding(vao, 0, 0);

// Enable the attribute
glEnableVertexArrayAttrib(vao, 0);

// Perform similar initialization for the second buffer
glNamedBufferStorage(buffer[1], sizeof(colors), colors, 0);
glVertexArrayVertexBuffer(vao, 1, buffer[1], 0, sizeof(vmath::vec3));
glVertexArrayAttribFormat(vao, 1, 3, GL_FLOAT, GL_FALSE, 0);
glVertexArrayAttribBinding(vao, 1, 1);
```

```
glEnableVertexAttribArray(1);
```

Listing 5.7: Multiple separate vertex attributes

In both cases of separate attributes, we have used *tightly packed* arrays of data to feed both attributes. This is effectively structure-of-arrays (SoA) data. We have a set of tightly packed, independent arrays of data. However, it's also possible to use an array-of-structures (AoS) form of data. Consider how the following structure might represent a single vertex:

```
struct vertex
{
    // Position
    float x;
    float y;
    float z;

    // Color
    float r;
    float g;
    float b;
};
```

Now we have two inputs to our vertex shader (position and color) interleaved together in a single structure. Clearly, if we make an array of these structures, we have an AoS layout for our data. To represent this with calls to **glVertexArrayVertexBuffer()**, we have to use its `stride` parameter. The `stride` parameter tells OpenGL how far apart *in bytes* the beginning of each vertex's data is. If we leave it as 0, OpenGL will use the same data for every vertex. However, to use the `vertex` structure declared above, we can simply use **sizeof** `(vertex)` for the `stride` parameter and everything will work out. Listing 5.8 shows the code to do this.

Click here to view code image

```
GLuint vao;
GLuint buffer;

static const vertex vertices[] = { ... };

// Create the vertex array object
glCreateVertexArrays(1, &vao);

// Allocate and initialize a buffer object
glCreateBuffers(1, &buffer);
glNamedBufferStorage(buffer, sizeof(vertices), vertices, 0);

// Set up two vertex attributes - first positions
glVertexArrayAttribBinding(vao, 0, 0);
glVertexArrayAttribFormat(vao, 0, 3, GL_FLOAT, GL_FALSE, offsetof(vertex,
```

```
x));
glEnableVertexArrayAttrib(0);

// Now colors
glVertexArrayAttribBinding(vao, 1, 0);
glVertexArrayAttribFormat(vao, 1, 3, GL_FLOAT, GL_FALSE, offsetof(vertex,
r));
glEnableVertexArrayAttrib(1);

// Finally, bind our one and only buffer to the vertex array object
glVertexArrayVertexBuffer(vao, 0, buffer);
```

Listing 5.8: Multiple interleaved vertex attributes

After executing the code in Listing 5.8, you can bind the vertex array object and start pulling data from the buffers bound to it.

After the vertex format information has been set up with calls to **glVertexArrayAttribFormat()**, you can change the vertex buffers that are bound with further calls to **glVertexArrayAttribBinding()**. If you want to render a lot of geometry stored in different buffers but with similar vertex formats, simply call **glVertexArrayAttribBinding()** to switch buffers and start drawing from them.

Loading Objects from Files

As you can see, you could potentially use a large number of vertex attributes in a single vertex shader. As we progress through various techniques, you will see that we'll regularly use four or five vertex attributes, and possibly more. Filling buffers with data to feed all of these attributes and then setting up the vertex array object and all of the vertex attribute pointers can be a chore. Further, encoding all of your geometry data directly in your application isn't practical for anything but the simplest models. Therefore, it makes sense to store model data in files and load it into your application. There are plenty of model file formats out there, and most modeling programs support several of the more common formats.

For the purpose of this book, we have devised a simple object file definition called an .SBM file, which stores the information we need without being either too simple or too overly engineered. Complete documentation for the format is found in Appendix B, "The SBM File Format." The `sb7` framework also includes a loader for this model format, called `sb7::object`. To load an object file, create an instance of `sb7::object` and call its `load` function as follows:

Click here to view code image

```
sb7::object my_object;

my_object.load("filename.sbm");
```

If this operation is successful, the model will be loaded into the instance of `sb7::object` and you will be able to render it. During loading, the class will create and set up the object's vertex array object and then configure all of the vertex attributes contained in the model file. The class also includes a `render` function that binds the object's vertex array object and calls the appropriate drawing command. For example, calling

```
my_object.render();
```

will render a single copy of the object with the current shaders. In many of the examples in the remainder of this book, we'll simply use our object loader to load object files (several of which are included with the book's source code) and render them.

Uniforms

Although not really a form of storage, uniforms are an important way to get data into shaders and to hook them up to your application. You have already seen how to pass data to a vertex shader using vertex attributes, and you have seen how to pass data from stage to stage using interface blocks. Uniforms allow you to pass data directly from your application into any shader stage. There are two flavors of uniforms, which differ based on how they are declared. The first are uniforms declared in the default block and the second are uniform blocks, whose values are stored in buffer objects. We will discuss both now.

Default Block Uniforms

While attributes are needed for per-vertex positions, surface normals, texture coordinates, and so on, a uniform is how we pass data into a shader that stays the same —is uniform—for an entire primitive batch or longer. Probably the single most common uniform for a vertex shader is the transformation matrix. We use transformation matrices in our vertex shaders to manipulate vertex positions and other vectors. Any shader variable can be specified as a uniform, and uniforms can be in any of the shader stages (even though we discuss only vertex and fragment shaders in this chapter). Making a uniform is as simple as placing the keyword **uniform** at the beginning of the variable declaration:

```
uniform float fTime;
uniform int iIndex;
uniform vec4 vColorValue;
uniform mat4 mvpMatrix;
```

Uniforms are always considered to be; and they cannot be assigned values by your shader code. However, you can initialize their default values at declaration time in a manner such as this:

```
uniform int answer = 42;
```

If you declare the same uniform in multiple shader stages, each of those stages will "see" the same value of that uniform.

Arranging Your Uniforms

After a shader has been compiled and linked into a program object, you can use one of the many functions defined by OpenGL to set the shader's values (assuming you don't want the defaults defined by the shader). Just as with vertex attributes, these functions refer to uniforms by their *location* within their program object. It is possible to specify the locations of uniforms in your shader code by using a location *layout qualifier*. When you do this, OpenGL will try to assign the locations that you specify to the uniforms in your shaders. The location layout qualifier looks like this:

Click here to view code image

```
layout (location = 17) uniform vec4 myUniform;
```

Notice the similarity between the location layout qualifer for uniforms and the one we've used for vertex shader inputs. In this case, `myUniform` will be allocated to location 17. If you don't specify a location for your uniforms in your shader code, OpenGL will automatically assign locations to them for you. You can figure out which locations were assigned by calling the **glGetUniformLocation()** function, whose prototype is

Click here to view code image

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

This function returns a signed integer that represents the location of the variable named by `name` in the program specified by `program`. For example, to get the location of a uniform variable named `vColorValue`, we would do something like this:

Click here to view code image

```
GLint iLocation = glGetUniformLocation(myProgram, "vColorValue");
```

In the previous example, passing `"myUniform"` to **glGetUniformLocation()** would result in the value 17 being returned. If you know a priori where your uniforms are because you assigned locations to them in your shaders, then you don't need to find them and you can avoid the calls to **glGetUniformLocation()**. This is the recommended way of doing things.

If the return value of **glGetUniformLocation()** is −1, it means the uniform name could not be located in the program. You should bear in mind that even if a shader compiles correctly, a uniform name may still "disappear" from the program if it is not

used directly in at least one of the attached shaders—even if you assign it a location explicitly in your shader source code. You do not need to worry about uniform variables being optimized away, but if you declare a uniform and then do not use it, the compiler will toss it out. Also, know that shader variable names are case sensitive, so you must get the case right when you query their locations.

Setting Uniforms

OpenGL supports a large number of data types both in the shading language and in the API. To to allow you to pass all this data around, it includes a huge number of functions just for setting the value of uniforms. A single scalar or vector data type can be set with any of the following variations on the **glUniform*()** function.

For example, consider the following four variables declared in a shader:

Click here to view code image

```
layout (location = 0) uniform float fTime;
layout (location = 1) uniform int iIndex;
layout (location = 2) uniform vec4 vColorValue;
layout (location = 3) uniform bool bSomeFlag;
```

To find and set these values in the shader, your C/C++ code might look something like this:

Click here to view code image

```
glUseProgram(myShader);
glUniform1f(0, 45.2f);
glUniform1i(1, 42);
glUniform4f(2, 1.0f, 0.0f, 0.0f, 1.0f);
glUniform1i(3, GL_FALSE);
```

Note that we used an integer version of **glUniform*()** to pass in a **bool** value. Booleans can also be passed in as floats, with 0.0 representing **false** and any non-zero value representing **true**.

The **glUniform*()** function also comes in flavors that take a pointer, potentially to an array of values. These forms end in the letter `v`, indicating that they consume a vector, and take a `count` value that represents how many elements are in each array of *x* number of components, where *x* is the number at the end of the function name. For example, suppose you had this uniform with four components:

```
uniform vec4 vColor;
```

In C/C++, you could represent this as an array of floats:

Click here to view code image

```
GLfloat vColor[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
```

But this is a single array of four values, so passing it into the shader would look like this:

Click here to view code image

```
glUniform4fv(iColorLocation, 1, vColor);
```

Now suppose you had an array of color values in your shader:

```
uniform vec4 vColors[2];
```

Then in C++, you could represent the data and pass it in like this:

Click here to view code image

```
GLfloat vColors[4][2] = { { 1.0f, 1.0f, 1.0f, 1.0f } ,
                          { 1.0f, 0.0f, 0.0f, 1.0f } };
...
glUniform4fv(iColorLocation, 2, vColors);
```

At its simplest, you can set a single floating-point uniform like this:

Click here to view code image

```
GLfloat fValue = 45.2f;
glUniform1fv(iLocation, 1, &fValue);
```

Finally, we see how to set a matrix uniform. Shader matrix data types only come in the single- and double-precision floating-point variety, so we have far less variation. To set the values in uniform matrices, we call the **glUniformMatrix*()** * commands.

In all of these functions, the variable `count` represents the number of matrices stored at the pointer parameter `m` (yes, you can have arrays of matrices!). The Boolean flag `transpose` is set to `GL_FALSE` if the matrix is already stored in column-major ordering (the way OpenGL prefers). Setting this value to `GL_TRUE` causes the matrix to be transposed when it is copied into the shader. This might be useful if you are using a matrix library that uses a row-major matrix layout instead (for example, some other graphics APIs use row-major ordering and you might want to use a library designed for one of them).

Uniform Blocks

Eventually, the shaders you'll be writing will become very complex. Some of them will require a lot of constant data, and passing all this to the shader using uniforms can become quite inefficient. If you have a lot of shaders in an application, you'll need to set up the uniforms for every one of those shaders, which means a lot of calls to the various **glUniform*()** functions. You'll also need to keep track of which uniforms change. Some change for every object and some change once per frame, while others may require initializing only once for the whole application. This means that you either need

to update different sets of uniforms in different places in your application (making it more complex to maintain) or update all the uniforms all the time (costing performance).

To alleviate the cost of all the **glUniform*()** calls, to make updating a large set of uniforms simpler, and to be able to easily share a set of uniforms between different programs, OpenGL allows you to combine a group of uniforms into a *uniform block* and store the whole block in a buffer object. The buffer object is just like any other that has been described earlier. You can quickly set the whole group of uniforms by either changing your buffer binding or overwriting the content of a bound buffer. You can also leave the buffer bound while you change programs, and the new program will see the current set of uniform values. This functionality is called the uniform buffer object (UBO). In fact, the uniforms you've used up until now live in the default block. Any uniform declared at the global scope in a shader ends up in the default uniform block. You can't keep the default block in a uniform buffer object; you need to create one or more named uniform blocks.

To declare a set of uniforms to be stored in a buffer object, you need to use a named uniform block in your shader. This looks a lot like the interface blocks described in the "Interface Blocks" section in Chapter 3, but it uses the **uniform** keyword instead of **in** or **out**. Listing 5.9 shows what the code looks like in a shader.

Click here to view code image

```
uniform TransformBlock
{
    float scale;              // Global scale to apply to everything
    vec3 translation;          // Translation in X, Y, and Z
    float rotation[3];        // Rotation around X, Y, and Z axes
    mat4 projection_matrix; // A generalized projection matrix to apply
                              // after scale and rotate
} transform;
```

Listing 5.9: Example uniform block declaration

This code declares a uniform block whose name is `TransformBlock`. It also declares a single instance of the block called `transform`. Inside the shader, you can refer to the members of the block using its instance name, `transform` (for example, `transform.scale` or `transform.projection_matrix`). However, to set up the data in the buffer object that you'll use to back the block, you need to know the location of a member of the block; for that, you need the block name, `TransformBlock`. If you wanted to have multiple instances of the block, each with its own buffer, you could make `transform` an array. The members of the block will have the same locations within each block, but there will now be several instances of the block that you can refer to in the shader. Querying the locations of members within a block is important when you want to fill the block with data, which is explained in the

following section.

Building Uniform Blocks

Data accessed in the shader via named uniform blocks can be stored in buffer objects. In general, it is the application's job to fill the buffer objects with data using functions like **glBufferData()** and **glMapBuffer()**. The question is, then, what is the data in the buffer supposed to look like? There are actually two possibilities here, and whichever one you choose is a trade-off.

The first method is to use a standard, agreed-upon layout for the data. This means that your application can just copy data into the buffers and assume specific locations for members within the block—you can even store the data on disk ahead of time and simply read it straight into a buffer that's been mapped using **glMapBuffer()**. The standard layout may leave some empty space between the various members of the block, making the buffer larger than it needs to be, and you might trade some performance for this convenience. Even so, using the standard layout is probably safe in almost all situations.

Another alternative is to let OpenGL decide where it would like the data. This can produce the most efficient shaders, but it means that your application needs to figure out where to put the data so that OpenGL can read it. Under this scheme, the data stored in uniform buffers is arranged in a *shared* layout. This is the default layout and is what you get if you don't explicitly ask OpenGL for something else. With the shared layout, the data in the buffer is laid out however OpenGL decides is best for runtime performance and access from the shader. This can sometimes allow for greater performance to be achieved by the shaders, but it requires more work from the application. The reason this is called the shared layout is that while OpenGL has arranged the data within the buffer, that arrangement will be the same between multiple programs and shaders sharing the same declaration of the uniform block. This allows you to use the same buffer object with any program. To use the shared layout, the application must determine the locations within the buffer object of the members of the uniform block.

First, we'll describe the *standard* layout, which is what we recommend that you use for your shaders (even though it's not the default). To tell OpenGL that you want to use the standard layout, you need to declare the uniform block with a layout qualifier. A declaration of a `TransformBlock` uniform block, with the standard layout qualifier, **std140**, is shown in Listing 5.10.

Click here to view code image

```
layout(std140) uniform TransformBlock
{
    float scale;             // Global scale to apply to everything
    vec3 translation;         // Translation in X, Y, and Z
```

```
    float rotation[3];       // Rotation around X, Y, and Z axes
    mat4 projection_matrix; // A generalized projection matrix to
                            // apply after scale and rotate
} transform;
```

Listing 5.10: Declaring a uniform block with the **std140** layout

Once a uniform block has been declared to use the standard, or **std140**, layout, each member of the block consumes a predefined amount of space in the buffer and begins at an offset that is predictable by following a set of rules. A summary of the rules follows. Any type consuming *N* bytes in a buffer begins on an *N*-byte boundary within that buffer. That means that standard GLSL types such as **int**, **float**, and **bool** (which are all defined to be 32-bit or 4-byte quantities) begin on multiples of 4 bytes. A vector of these types of length 2 always begins on a 2*N*-byte boundary. For example, a **vec2**, which is 8 bytes long in memory, always starts on an 8-byte boundary. Three- and four-element vectors always start on a 4*N*-byte boundary; **vec3** and **vec4** types start on 16-byte boundaries, for instance. Each member of an array of scalar or vector types (arrays of **int** or **vec3**, for example) always starts on a boundary defined by these same rules, but rounded up to the alignment of a **vec4**. In particular, this means that arrays of anything but **vec4** (and $N \times 4$ matrices) won't be tightly packed, but instead will have a gap between each of the elements. Matrices are essentially treated like short arrays of vectors, and arrays of matrices are treated like very long arrays of vectors. Finally, structures and arrays of structures have additional packing requirements; the whole structure starts on the boundary required by its largest member, rounded up to the size of a **vec4**.

Particular attention must be paid to the difference between the **std140** layout and the packing rules that are often followed by your C++ (or other application language) compiler of choice. In particular, an array in a uniform block is not necessarily tightly packed. This means that you can't create, for example, an array of **float** in a uniform block and simply copy data from a C array into it, because the data from the C array will be packed and the data in the uniform block won't be.

This all sounds complex, but it is logical and well defined, and allows a large range of graphics hardware to implement uniform buffer objects efficiently. Returning to our `TransformBlock` example, we can figure out the offsets of the members of the block within the buffer using these rules. Listing 5.11 shows an example of a uniform block declaration along with the offsets of its members.

Click here to view code image

```
layout(std140) uniform TransformBlock
{
```

```
// Member                          base alignment offset  aligned offset
   float scale;                 // 4               0       0
   vec3 translation;             // 16              4       16
   float rotation[3];           // 16              28      32 (rotation[0])
                                 //                          48 (rotation[1])
                                 //                          64 (rotation[2])
   mat4 projection_matrix;       // 16              80      80 (column 0)
                                 //                          96 (column 1)
                                 //                          112 (column 2)
                                 //                          128 (column 3)
} transform;
```

Listing 5.11: Example uniform block with offsets

There is a complete example of the alignments of various types in the original `ARB_uniform_buffer_object` extension specification.

As an alternative to using the **std140** layout, it's possible to directly specify the offsets of members of uniform blocks in your shader code. You still have to follow the alignment rules as required by **std140**, but you can do things like leave gaps between members and declare members out of order. To specify the offsets of members in a uniform block, use the **offset** layout qualifier. For an example, see Listing 5.12.

Click here to view code image

```
layout(std140) uniform ManuallyLaidOutBlock
{
    layout (offset = 32) vec4       foo;    // At offset 32 bytes
    layout (offset = 8) vec2        bar;    // At offset 8 bytes
    layout (offset = 48) vec3       baz;    // At offset 48 bytes
}  myBlock;
```

Listing 5.12: Uniform block with user-specified offsets

In Listing 5.12 you will notice that the first member in the block, `foo`, is declared as starting at offset 32 in the block. This is fine because 32 is a multiple of 16 (the size of a **vec4**) and meets the alignment requirements for the type of `foo`. `bar` starts at offset 8—again, this satisfies the alignment requirements for a variable of type **vec2**. However, it's *before* `foo` in memory—we have decared the members out of order. Next, we declare `baz` at offset 48. Although `baz` is a **vec3** variable, it must be aligned on a 16-byte boundary.

It's also possible to explicitly align types on boundaries that are multiples of their native alignment. To do this, we use the `align` layout qualifier. This is used similarly to the **offset** layout qualifier, but simply pushes the member to the next multiple of the specified alignment, so long as it meets the alignment requirements of the member. The

`align` qualifier can be used on a whole block to force all of its members to be aligned to the specified size. You can use `align` and **offset** together to push members to the next offset that is greater than or equal to its specified value *and* is a multiple of of the block alignment.

In Listing 5.13, we have redeclared our `ManuallyLaidOutBlock` uniform block with an alignment of 16. This satisfies the requirements of **vec4** and **vec3** types, so the offsets of `foo` and `baz` are not affected. However, the natural alignment of `bar` is only 8 bytes, which is not a multiple of 16. Therefore, `baz` will be aligned on the next 16-byte boundary after the specified alignment (which is 16).

Click here to view code image

```
layout (std140, align = 16) uniform ManuallyLaidOutBlock
{
    layout (offset = 32) vec4       foo;    // At offset 32 bytes
    layout (offset = 8) vec2        bar;    // At offset 16 bytes
    layout (offset = 48) vec3       baz;    // At offset 48 bytes
} myBlock;
```

Listing 5.13: Uniform block with user-specified alignments

You can, of course, choose to leave everything in the hands of OpenGL by using the **shared** layout, and it *might* produce a slightly more efficient layout than **std140**, but it's probably not worth the additional effort. If you really want to use the shared layout, you can determine the offsets that OpenGL assigned to your block members. Each member of a uniform block has an index that is used to refer to it to find its size and location within the block. To get the index of a member of a uniform block, call

Click here to view code image

```
void glGetUniformIndices(GLuint program,
                         GLsizei uniformCount,
                         const GLchar ** uniformNames,
                         GLuint * uniformIndices);
```

This function allows you to get the indices of a large set of uniforms—perhaps even all of the uniforms in a program—with a single call to OpenGL, even if they're members of different blocks. It takes a count of the number of uniforms you'd like the indices for (`uniformCount`) and an array of uniform names (`uniformNames`) and puts their indices in an array for you (`uniformIndices`). Listing 5.14 contains an example of how you would retrieve the indices of the members of `TransformBlock`, which we declared earlier.

Click here to view code image

```
static const GLchar * uniformNames[4] =
```

```
{
    "TransformBlock.scale",
    "TransformBlock.translation",
    "TransformBlock.rotation",
    "TransformBlock.projection_matrix"
};
GLuint uniformIndices[4];

glGetUniformIndices(program, 4, uniformNames, uniformIndices);
```

Listing 5.14: Retrieving the indices of uniform block members

After this code has run, you have the indices of the four members of the uniform block in the `uniformIndices` array. Now that you have the indices, you can use them to find the locations of the block members within the buffer. To do this, call

Click here to view code image

```
void glGetActiveUniformsiv(GLuint program,
                           GLsizei uniformCount,
                           const GLuint * uniformIndices,
                           GLenum pname,
                           GLint * params);
```

This function can give you a lot of information about specific uniform block members. The information that we're interested in is the offset of the member within the buffer, the array stride (for `TransformBlock.rotation`), and the matrix stride (for `TransformBlock.projection_matrix`). These values tell us where to put data within the buffer so that it can be seen in the shader. We can retrieve these from OpenGL by setting `pname` to `GL_UNIFORM_OFFSET`, `GL_UNIFORM_ARRAY_STRIDE`, and `GL_UNIFORM_MATRIX_STRIDE`, respectively. Listing 5.15 shows what the code looks like.

Click here to view code image

```
GLint uniformOffsets[4];
GLint arrayStrides[4];
GLint matrixStrides[4];
glGetActiveUniformsiv(program, 4, uniformIndices,
                      GL_UNIFORM_OFFSET, uniformOffsets);
glGetActiveUniformsiv(program, 4, uniformIndices,
                      GL_UNIFORM_ARRAY_STRIDE, arrayStrides);
glGetActiveUniformsiv(program, 4, uniformIndices,
                      GL_UNIFORM_MATRIX_STRIDE, matrixStrides);
```

Listing 5.15: Retrieving the information about uniform block members

Once the code in Listing 5.15 has run, `uniformOffsets` contains the offsets of the members of the `TransformBlock` block, `arrayStrides` contains the strides of

the array members (only rotation, for now), and `matrixStrides` contains the strides of the matrix members (only `projection_matrix`).

The other information that you can find out about uniform block members includes the data type of the uniform, the size in bytes that it consumes in memory, and layout information related to arrays and matrices within the block. You need some of that information to initialize a buffer object with more complex types, although the size and types of the members should be known to you already if you wrote the shaders. The other accepted values for `pname` and what you get back are listed in Table 5.4.

Value of pname	What You Get Back
GL_UNIFORM_TYPE	The data type of the uniform as a GLenum.
GL_UNIFORM_SIZE	The size of arrays, in units of whatever GL_UNIFORM_TYPE gives you. If the uniform is not an array, this will always be 1.
GL_UNIFORM_NAME_LENGTH	The length, in characters, of the names of the uniforms.
GL_UNIFORM_BLOCK_INDEX	The index of the block that the uniform is a member of.
GL_UNIFORM_OFFSET	The offset of the uniform within the block.
GL_UNIFORM_ARRAY_STRIDE	The number of bytes between consecutive elements of an array. If the uniform is not an array, this will be 0.
GL_UNIFORM_MATRIX_STRIDE	The number of bytes between the first element of each column of a column-major matrix or each row of a row-major matrix. If the uniform is not a matrix, this will be 0.
GL_UNIFORM_IS_ROW_MAJOR	Each element of the output array will either be 1 if the uniform is a row-major matrix, or 0 if it is a column-major matrix or not a matrix at all.

Table 5.4: Uniform Parameter Queries via **glGetActiveUniformsiv()**

If the type of the uniform you're interested in is a simple type such as **int**, **float**, **bool**, or even vectors of these types (**vec4** and so on), all you need is its offset. Once you know the location of the uniform within the buffer, you can either pass the offset to **glBufferSubData()** to load the data at the appropriate location, or use the offset directly in your code to assemble the buffer in memory. We demonstrate the latter option here because it reinforces the idea that the uniforms are stored in memory, just like vertex information can be stored in buffers. It also means fewer calls are made to OpenGL, which can sometimes lead to higher performance. For these examples, we assemble the data in the application's memory and then load it into a buffer using **glBufferSubData()**. Alternatively, you could use **glMapBufferRange()** to get a pointer to the buffer's memory and assemble the data directly into that.

Let's start by setting the simplest uniform in the `TransformBlock` block, `scale`. This uniform is a single float whose location is stored in the first element of our `uniformIndices` array. Listing 5.16 shows how to set the value of the single float.

Click here to view code image

```
// Allocate some memory for our buffer (don't forget to free it later)
unsigned char * buffer = (unsigned char *)malloc(4096);

// We know that TransformBlock.scale is at uniformOffsets[0] bytes
// into the block, so we can offset our buffer pointer by that and
// store the scale there.
*((float *)(buffer + uniformOffsets[0])) = 3.0f;
```

Listing 5.16: Setting a single float in a uniform block

Next, we can initialize data for `TransformBlock.translation`. This is a **vec3**, which means it consists of three floating-point values packed tightly together in memory. To update this, all we need to do is find the location of the first element of the vector and store three consecutive floats in memory starting there. This is shown in Listing 5.17.

Click here to view code image

```
// Put three consecutive GLfloat values in memory to update a vec3
((float *)(buffer + uniformOffsets[1]))[0] = 1.0f;
((float *)(buffer + uniformOffsets[1]))[1] = 2.0f;
((float *)(buffer + uniformOffsets[1]))[2] = 3.0f;
```

Listing 5.17: Retrieving the indices of uniform block members

Now, we tackle the array `rotation`. We could have also used a **vec3** here, but for the purposes of this example, we use a three-element array to demonstrate the use of the

GL_UNIFORM_ARRAY_STRIDE parameter. When the **shared** layout is used, arrays are defined as a sequence of elements separated by an implementation-defined stride in bytes. This means that we have to place the data at locations in the buffer defined both by GL_UNIFORM_OFFSET and GL_UNIFORM_ARRAY_STRIDE, as in the code snippet of Listing 5.18.

Click here to view code image

```
// TransformBlock.rotations[0] is at uniformOffsets[2] bytes into
// the buffer. Each element of the array is at a multiple of
// arrayStrides[2] bytes past that.
const GLfloat rotations[] = {  30.0f, 40.0f, 60.0f };
unsigned int offset = uniformOffsets[2];

for (int n = 0; n < 3; n++)
{
    *((float *)(buffer + offset)) = rotations[n];
    offset += arrayStrides[2];
}
```

Listing 5.18: Specifying the data for an array in a uniform block

Finally, we set up the data for TransformBlock.projection_matrix. Matrices in uniform blocks behave much like arrays of vectors. For column-major matrices (which is the default), each column of the matrix is treated like a vector, the length of which is the height of the matrix. Likewise, a row-major matrix is treated like an array of vectors where each row is an element in that array. Just like normal arrays, the starting offset for each column (or row) in the matrix is determined by an implementation-defined quantity. This can be queried by passing the GL_UNIFORM_MATRIX_STRIDE parameter to **glGetActiveUniformsiv()**. Each column of the matrix can be initialized using code similar to that used to initialize the **vec3** TransformBlock.translation. This setup code is given in Listing 5.19.

Click here to view code image

```
// The first column of TransformBlock.projection_matrix is at
// uniformOffsets[3] bytes into the buffer. The columns are
// spaced matrixStride[3] bytes apart and are essentially vec4s.
// This is the source matrix - remember, it's column major.
const GLfloat matrix[] =
{
    1.0f, 2.0f, 3.0f, 4.0f,
    9.0f, 8.0f, 7.0f, 6.0f,
    2.0f, 4.0f, 6.0f, 8.0f,
    1.0f, 3.0f, 5.0f, 7.0f
};

for (int i = 0; i < 4; i++)
```

```
    {
        GLuint offset = uniformOffsets[3] + matrixStride[3] * i;
        for (j = 0; j < 4; j++)
        {
            *((float *)(buffer + offset)) = matrix[i * 4 + j];
            offset += sizeof(GLfloat);
        }
    }
```

Listing 5.19: Setting up a matrix in a uniform block

This method of querying offsets and strides works for any of the layouts. With the shared layout, it is the only option. However, it's somewhat inconvenient, and as you can see, you need quite a lot of code to lay out your data in the buffer in the correct way. This is why we recommend that you use the *standard* layout. This allows you to determine where in the buffer data should be placed based on a set of rules that specify the sizes and alignments for the various data types supported by OpenGL. These rules are common across all OpenGL implementations, so you don't need to query anything to use them (although, should you query offsets and strides, the results will be correct). There is some chance that you'll trade a small amount of shader performance for its use, but the savings in code complexity and application performance are well worth it.

Regardless of which packing mode you choose, you can bind your buffer full of data to a uniform block in your program. Before you can do this, you need to retrieve the index of the uniform block. Each uniform block in a program has a compiler-assigned index. There is a fixed maximum number of uniform blocks that can be used by a single program, as well as a maximum number that can be used in any given shader stage. You can find these limits by calling **glGetIntegerv()** with the `GL_MAX_UNIFORM_BUFFERS` parameter (for the total per program) and either `GL_MAX_VERTEX_UNIFORM_BUFFERS`, `GL_MAX_GEOMETRY_UNIFORM_BUFFERS`, `GL_MAX_TESS_CONTROL_UNIFORM_BUFFERS`, `GL_MAX_TESS_EVALUATION_UNIFORM_BUFFERS`, or `GL_MAX_FRAGMENT_UNIFORM_BUFFERS` for the vertex, geometry, tessellation control, tessellation evaluation, and fragment shader limits, respectively. To find the index of a uniform block in a program, call

Click here to view code image

```
GLuint glGetUniformBlockIndex(GLuint program,
                              const GLchar * uniformBlockName);
```

This returns the index of the named uniform block. In our example uniform block declaration here, `uniformBlockName` would be `"TransformBlock"`. There is a set of buffer binding points to which you can bind a buffer to provide data for the

uniform blocks. It is essentially a two-step process to bind a buffer to a uniform block. Uniform blocks are assigned binding points, and then buffers can be bound to those binding points, matching buffers with uniform blocks. This way, different programs can be switched in and out without changing buffer bindings, and the fixed set of uniforms will automatically be seen by the new program. Contrast this to the values of the uniforms in the default block, which are per program state. Even if two programs contain uniforms with the same names, their values must be set for each program and will change when the active program is changed.

To assign a binding point to a uniform block, call

Click here to view code image

```
void glUniformBlockBinding(GLuint program,
                           GLuint uniformBlockIndex,
                           GLuint uniformBlockBinding);
```

Here, `program` is the program where the uniform block you're changing lives. `uniformBlockIndex` is the index of the uniform block to which you're assigning a binding point; you just retrieved that by calling **glGetUniformBlockIndex()**. `uniformBlockBinding` is the index of the uniform block binding point. An implementation of OpenGL supports a fixed maximum number of binding points, and you can determine that limit by calling **glGetIntegerv()** with the `GL_MAX_UNIFORM_BUFFER_BINDINGS` parameter.

Alternatively, you can specify the binding index of your uniform blocks directly in your shader code. To do this, we again use the layout qualifier, this time with the **binding** keyword. For example, to assign our `TransformBlock` block to binding 2, we could declare it as

Click here to view code image

```
layout(std140, binding = 2) uniform TransformBlock
{
    ...
}  transform;
```

Notice that the **binding** layout qualifier can be specified at the same time as the **std140** (or any other) qualifier. Assigning bindings in your shader source code avoids the need to call **glUniformBlockBinding()**, or even to determine the block's index from your application; consequently, it is usually the best method of assigning block location.

Once you've assigned binding points to the uniform blocks in your program, whether through the **glUniformBlockBinding()** function or through a layout qualifier, you can bind buffers to those same binding points to make the data in the buffers appear in the uniform blocks. To do this, call

Click here to view code image

```
glBindBufferBase(GL_UNIFORM_BUFFER, index, buffer);
```

Here, `GL_UNIFORM_BUFFER` tells OpenGL that we're binding a buffer to one of the uniform buffer binding points; `index` is the index of the binding point and should match what you specified either in your shader or in `uniformBlockBinding` in your call to **glUniformBlockBinding()**; and `buffer` is the name of the buffer object that you want to attach. It's important to note that `index` is not the index of the uniform block (`uniformBlockIndex` in **glUniformBlockBinding()**), but rather the index of the uniform buffer binding point. This is a common mistake to make and is easy to miss.

This mixing and matching of binding points with uniform block indices is illustrated in Figure 5.1.

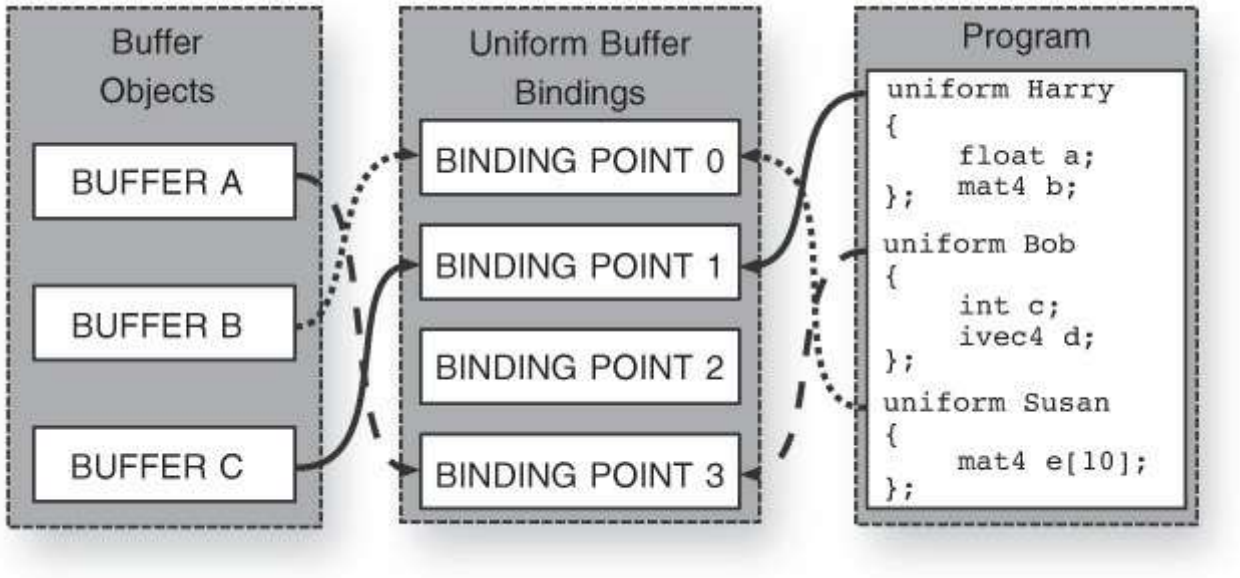

Figure 5.1: Binding buffers and uniform blocks to binding points

In Figure 5.1, there is a program with three uniform blocks (`Harry`, `Bob`, and `Susan`) and three buffer objects (`A`, `B`, and `C`). `Harry` is assigned to binding point 1, and buffer `C` is bound to binding point 1, so `Harry`'s data comes from buffer `C`. Likewise, `Bob` is assigned to binding point 3, to which buffer `A` is bound, so `Bob`'s data comes from buffer `A`. Finally, `Susan` is assigned to binding point 0 and buffer `B` is bound to binding point 0, so `Susan`'s data comes from buffer `B`. Notice that binding point 2 is not used. That doesn't matter. There could be a buffer bound there, but the program doesn't use it.

The code to set this up is simple and is given in Listing 5.20.

Click here to view code image

```
// Get the indices of the uniform blocks using glGetUniformBlockIndex
GLuint harry_index = glGetUniformBlockIndex(program, "Harry");
GLuint bob_index   = glGetUniformBlockIndex(program, "Bob");
```

```
GLuint susan_index = glGetUniformBlockIndex(program, "Susan");

// Assign buffer bindings to uniform blocks, using their indices
glUniformBlockBinding(program, harry_index, 1);
glUniformBlockBinding(program, bob_index, 3);
glUniformBlockBinding(program, susan_index, 0);

// Bind buffers to the binding points
// Binding 0, buffer B, Susan's data
glBindBufferBase(GL_UNIFORM_BUFFER, 0, buffer_b);
// Binding 1, buffer C, Harry's data
glBindBufferBase(GL_UNIFORM_BUFFER, 1, buffer_c);
// Note that we skipped binding 2
// Binding 3, buffer A, Bob's data
glBindBufferBase(GL_UNIFORM_BUFFER, 3, buffer_a);
```

Listing 5.20: Specifying bindings for uniform blocks

If we had set the bindings for our uniform blocks in our shader code by using the **binding** layout qualifier, we could have avoided the calls to **glUniformBlockBinding()** in Listing 5.20. This approach is shown in Listing 5.21.

Click here to view code image

```
layout (binding = 1) uniform Harry
{
    // ...
};

layout (binding = 3) uniform Bob
{
   // ...
};

layout (binding = 0) uniform Susan
{
   // ...
};
```

Listing 5.21: Uniform blocks binding layout qualifiers

After a shader containing the declarations shown in Listing 5.21 is compiled and linked into a program object, the bindings for the `Harry`, `Bob`, and `Susan` uniform blocks will be set to the same things as they would be after executing Listing 5.20. Setting the uniform block binding in the shader can be useful for a number of reasons. First, it reduces the number of calls to OpenGL that your application must make. Second, it allows the shader to associate a uniform block with a particular binding point without the application needing to know its name. This can be helpful if you have some data in a buffer with a standard layout, but want to refer to it with different names in different

shaders.

A common use for uniform blocks is to separate steady state from transient state. By setting up the bindings for all your programs using a standard convention, you can leave buffers bound when you change the program. For example, if you have some relatively fixed state—say the projection matrix, the size of the viewport, and a few other things that change once per frame or less often—you can leave that information in a buffer bound to binding point 0. Then, if you set the binding for the fixed state to 0 for all programs, whenever you switch program objects using **glUseProgram()**, the uniforms will be sitting there in the buffer, ready to use.

Suppose you have a fragment shader that simulates some material (cloth or metal, for example); you could put the parameters for the material into another buffer. In your program that shades each material, bind the uniform block containing the material parameters to binding point 1. Each object would maintain a buffer object containing the parameters of its surface. As you render each object, it uses the common material shader and simply binds its parameter buffer to buffer binding point 1.

A final significant advantage of uniform blocks is that they can be quite large. The maximum size of a uniform block can be determined by calling **glGetIntegerv()** and passing the `GL_MAX_UNIFORM_BLOCK_SIZE` parameter. Also, the number of uniform blocks that you can access from a single program can be retrieved by calling **glGetIntegerv()** and passing the `GL_MAX_UNIFORM_BLOCK_BINDINGS` parameter. OpenGL guarantees that uniform blocks may be at least 64K in size, and that you can have at least 14 of them referenced by a single program. Taking the example of the previous paragraph a little further, you could pack all of the properties for all of the materials used by your application into a single, large uniform block containing a big array of structures. As you render the objects in your scene, you need simply communicate the index within the array of the material you wish to use. You can achieve that with a static vertex attribute or traditional uniform, for example. This could be substantially faster than replacing the contents of a buffer object or changing uniform buffer bindings between each object. If you're really clever, you could even render objects made up of multiple surfaces with different materials using a single drawing command.