Shader Storage Blocks

In addition to the read-only access to buffer objects that is provided by uniform blocks, buffer objects can be used for general storage from shaders using *shader storage blocks*. These are declared in a similar manner to uniform blocks and backed in the same way by binding a range of a buffer object to one of the indexed `GL_SHADER_STORAGE_BUFFER` targets. However, the biggest difference between a uniform block and a shader storage block is that your shader can *write* into the shader storage block; furthermore, it can even perform *atomic operations* on members of a shader storage block. Shader storage blocks also have a much higher upper size limit.

To declare a shader storage block, simply declare a block in the shader just like you would a uniform block; however, rather than use the **uniform** keyword, use the **buffer** qualifier. Like uniform blocks, shader storage blocks support the **std140** packing layout qualifier, but they also support the **std430**[2] packing layout qualifier,

which allows arrays of integers and floating-point variables (and structures containing them) to be tightly packed (something that is sorely lacking in the **std140** layout). This allows for better efficiency of memory use and tighter cohesion with structure layouts generated by compilers for languages such as C++. An example shader storage block declaration is shown in Listing 5.29.

2. The **std140** and **std430** packing layouts are named for the version of the shading language with which they were introduced—**std140** with GLSL 1.40, which was part of OpenGL 3.1, and **std430** with GLSL 4.30, which was the version released with OpenGL 4.3.

Click here to view code image

```
#version 450 core

struct my_structure
{
    int         pea;
    int         carrot;
    vec4        potato;
};

layout (binding = 0, std430) buffer my_storage_block
{
    vec4            foo;
    vec3            bar;
    int             baz[24];
    my_structure    veggies;
};
```

Listing 5.29: Example shader storage block declaration

The members of a shader storage block can be referred to just as any other variable. To read from them, you could, for example, use them as a parameter to a function; to write into them, you simply assign to them. When the variable is used in an expression, the source of data will be the buffer object; when the variable is assigned to, the data will be written into the buffer object. You can place data into the buffer using functions like **glBufferData()** just as you would with a uniform block. Because the buffer is writable by the shader, if you call **glMapBufferRange()** with `GL_MAP_READ_BIT` (or `GL_MAP_WRITE_BIT`) as the access mode, you will be able to read the data produced by your shader.

Shader storage blocks and their backing buffer objects provide additional advantages over uniform blocks. For example, their size is not really limited. Of course, if you go overboard, OpenGL may fail to allocate memory for you, but there really isn't a hard-wired practical upper limit to the size of a shader storage block. Also, the newer packing rules for **std430** allow an application's data to be more efficiently packed

and directly accessed than would be possible with a uniform block. It is worth noting, though, that due to the stricter alignment requirements of uniform blocks and smaller minimum size, some hardware may handle uniform blocks differently than shader storage blocks and execute more efficiently when reading from them. Listing 5.30 shows how you might use a shader storage block in place of regular inputs in a vertex shader.

Click here to view code image

```
#version 450 core

struct vertex
{
    vec4            position;
    vec3            color;
};

layout (binding = 0, std430) buffer my_vertices
{
    vertex          vertices[];
};

uniform mat4 transform_matrix;

out VS_OUT
{
    vec3        color;
} vs_out;

void main(void)
{
    gl_Position = transform_matrix * vertices[gl_VertexID].position;
    vs_out.color = vertices[gl_VertexID].color;
}
```

Listing 5.30: Using a shader storage block in place of vertex attributes

Although it may seem that shader storage blocks offer so many advantages that they almost make uniform blocks and vertex attributes redundant, you should be aware that their additional flexibility makes it difficult for OpenGL to truly optimize access to storage blocks. For example, some OpenGL implementations may be able to provide faster access to uniform blocks given the knowledge that their content will always be constant. Also, reading the input data for vertex attributes may happen long before your vertex shader runs, letting OpenGL's memory subsystem keep up. Reading vertex data right in the middle of your shader might slow it down quite a bit.

Atomic Memory Operations

In addition to simple reading and writing of memory, shader storage blocks allow you to

perform *atomic operations* on memory. An atomic operation is a sequence of a read from memory potentially followed by a write to memory that must be uninterrupted for the result to be correct. Consider a case where two shader invocations perform the operation `m = m + 1;` using the same memory location represented by `m`. Each invocation will load the current value stored in the memory location represented by `m`, add 1 to it, and then write it back to memory at the same location.

If each invocation operates in lockstep, then we will end up with the wrong value in memory unless the operation can be made atomic. This is because the first invocation will load the value from memory, then the second invocation will read the *same* value from memory. Both invocations will increment their copy of the value, the first invocation will write its incremented value back to memory, and finally the second invocation will overwrite that value with the same, incremented value that it calculated. This problem only gets worse when there are many more than two invocations running at a time.

To get around this problem, atomic operations cause the complete read–modify–write cycle to complete for one invocation before any other invocation gets a chance to even read from memory. In theory, if multiple shader invocations perform atomic operations on different memory locations, then everything should run nice and fast and work just as if you had written the naïve `m = m + 1;` code in your shader. If two invocations access the same memory locations (a condition known as *contention*), then they will be *serialized* and only one will get to go at one time. To execute an atomic operation on a member of a shader storage block, you call one of the atomic memory functions listed in Table 5.5.

Atomic Function	Behavior
`atomicAdd(mem, data)`	Reads from `mem`, adds it to `data`, writes the result back to `mem`, and then returns the value originally stored in `mem`.
`atomicAnd(mem, data)`	Reads from `mem`, logically ANDs it with `data`, writes the result back to `mem`, and then returns the value originally stored in `mem`.
`atomicOr(mem, data)`	Reads from `mem`, logically ORs it with `data`, writes the result back to `mem`, and then returns the value originally stored in `mem`.
`atomicXor(mem, data)`	Reads from `mem`, logically exclusive ORs it with `data`, writes the result back to `mem`, and then returns the value originally stored in `mem`.
`atomicMin(mem, data)`	Reads from `mem`, determines the minimum of the retrieved value and `data`, writes the result back to `mem`, and then returns the value originally stored in `mem`.
`atomicMax(mem, data)`	Reads from `mem`, determines the maximum of the retrieved value and `data`, writes the result back to `mem`, and then returns the value originally stored in `mem`.
`atomicExchange(mem, data)`	Reads from `mem`, writes the value of `data` into `mem`, and then returns the value originally stored in `mem`.
`atomicCompSwap(mem, comp, data)`	Reads from `mem`, compares the retrieved value with `comp`, and, if they are equal, writes `data` into `mem`, but always returns the value originally stored in `mem`.

Table 5.5: Atomic Operations on Shader Storage Blocks

In Table 5.5, all of the functions have an integer (**int**) and unsigned integer (**uint**)

version. For the integer versions, `mem` is declared as **inout int** `mem`; `data` and `comp` (for `atomicCompSwap`) are declared as **int** `data`; and **int** `comp` and the return values of all functions are **int**. Likewise, for the unsigned integer versions, all parameters are declared using **uint** and the return type of the function is **uint**. Notice that there are no atomic operations on floating-point variables, vectors or matrices, or integer values that are not 32 bits wide. All of the atomic memory access functions shown in Table 5.5 return the value that was in memory *prior* to the atomic operation taking place. When an atomic operation is attempted by multiple invocations of your shader to the same location at the same time, they are *serialized*, which means that they take turns. This means that you're not guaranteed to receive any particular return value from an atomic memory operation.

Synchronizing Access to Memory

When you are only reading from a buffer, data is *almost* always going to be available when you think it should be and you don't need to worry about the order in which your shaders read from it. However, when your shader starts writing data into buffer objects, either through writes to variables in shader storage blocks or through explicit calls to the atomic operation functions that might write to memory, there are cases where you need to avoid *hazards*.

Memory hazards fall roughly into three categories:

- A read-after-write (RAW) hazard can occur when your program attempts to read from a memory location right after it has written to it. Depending on the system architecture, the read and write may be reordered such that the read actually ends up being executed *before* the write is complete, resulting in the old data being returned to the application.
- A write-after-write (WAW) hazard can occur when a program performs a write to the same memory location twice in a row. You might expect that whatever data was written last would overwrite the data written first and be the value that ends up staying in memory. Again, on some architectures this is not guaranteed; in some circumstances the *first* data written by the program might actually be the data that ends up in memory.
- A write-after-read (WAR) hazard normally occurs only in parallel processing systems (such as graphics processors) and may happen when one thread of execution (such as a shader invocation) performs a write to memory after another thread believes that it has read from memory. If these operations are reordered, the thread that performed the read may end up getting the data that was written by the second thread without expecting it.

Because of the deeply pipelined and highly parallel nature of the systems that OpenGL

is expected to be running on, it includes a number of mechanisms to alleviate and control memory hazards. Without these features, OpenGL implementations would need to be far more conservative about reordering your shaders and running them in parallel. The main apparatus for dealing with memory hazards is the *memory barrier*.

A memory barrier essentially acts as a marker that tells OpenGL, "Hey, if you're going to start reordering things, that's fine—just don't let anything I say after this point actually happen before anything I say before it." You can insert barriers both in your application code—with calls to OpenGL—and in your shaders.

Using Barriers in Your Application

The function to insert a barrier is **glMemoryBarrier()** and its prototype is

Click here to view code image

```
void glMemoryBarrier(GLbitfield barriers);
```

The **glMemoryBarrier()** function takes a `GLbitfield` parameter, `barriers`, which allows you to specify which of OpenGL's memory subsystems should obey the barrier and which are free to ignore it and continue as they would have. The barrier affects ordering of memory operations in the categories specified in `barriers`. If you want to bash OpenGL with a big hammer and just synchronize everything, you can set `barriers` to `GL_ALL_BARRIER_BITS`. However, there are quite a number of bits defined that you can add together to be more precise about what you want to synchronize. A few examples are listed here:

- Including `GL_SHADER_STORAGE_BARRIER_BIT` tells OpenGL that you want it to let any accesses (writes in particular) performed by shaders that are run before the barrier complete before letting any shaders access the data after the barrier. Thus, if you write into a shader storage buffer from a shader and then call **glMemoryBarrier()** with `GL_SHADER_STORAGE_BARRIER_BIT` included in `barriers`, shaders you run after the barrier will "see" that data. Without such a barrier, this is not guaranteed.
- Including `GL_UNIFORM_BARRIER_BIT` in `barriers` tells OpenGL that you might have written something into memory that might be used as a uniform buffer after the barrier, and it should wait to make sure that shaders that write into the buffer have completed before letting shaders that use it as a uniform buffer run. You would set this, for example, if you wrote into a buffer using a shader storage block in a shader and then wanted to use that buffer as a uniform buffer later.
- Including `GL_VERTEX_ATTRIB_ARRAY_BARRIER_BIT` ensures that OpenGL will wait for shaders that write to buffers to complete before using any of those buffers as the source of vertex data through a vertex attribute. For example, you would set this if you write into a buffer through a shader storage block and

then want to use that buffer as part of a vertex array to feed data into the vertex shader of a subsequent drawing command.

There are plenty more of these bits that control the ordering of shaders with respect to OpenGL's other subsystems; we will introduce them as we talk more in depth about those subsystems. The key points about **glMemoryBarrier()** are that the items included in `barriers` are the *destination* subsystems and that the mechanism by which you updated the data is assumed to be writing it to memory with a shader.

Using Barriers in Your Shaders

Just as you can insert memory barriers in your application's code to control the ordering of memory accesses performed by your shaders relative to your application, so you can also insert barriers into your shaders to stop OpenGL from reading or writing memory in some order other than what your shader code says. The basic memory barrier function in GLSL is

```
void memoryBarrier();
```

If you call `memoryBarrier()` from your shader code, any memory reads or writes that you might have performed will complete before the function returns. This means that it's safe to read data back that you might have just written. Without a barrier, it's even possible that when you read from a memory location that you just wrote to, OpenGL will return the *old* data instead of the new!

To provide finer control over which types of memory accesses are ordered, there are some more specialized versions of the `memoryBarrier()` function. For example, `memoryBarrierBuffer()` orders transactions on reads and writes to buffers, but to nothing else. We'll introduce the other barrier functions as we talk about the types of data that they protect.

Atomic Counters

Atomic counters are a special type of variable that represents storage that is shared across multiple shader invocations. This storage is backed by a buffer object, and functions are provided in GLSL to increment and decrement the values stored in the buffer. What is special about these operations is that they are *atomic*: Just like with the equivalent functions for members of shader storage blocks (shown in Table 5.5), they return the original value of the counter before it was modified. Just like the other atomic operations, if two shader invocations increment the same counter at the same time, OpenGL will make them take turns. One shader invocation will receive the original value of the counter, the other will receive the original value plus 1, and the final value of the counter will be that of the original value plus 2. Also, just as with shader storage block atomics, there is no guarantee of the order in which these operations will occur,

so you can't rely on receiving any specific value.

To declare an atomic counter in a shader, do this:

Click here to view code image

```
layout (binding = 0) uniform atomic_uint my_variable;
```

OpenGL provides a number of binding points to which you can bind the buffers where it will store the values of atomic counters. Additionally, each atomic counter is stored at a specific offset within the buffer object. The buffer binding index and the offset within the buffer bound to that binding can be specified using the **binding** and **offset** layout qualifiers, which can be applied to an atomic counter uniform declaration. For example, if we wish to place `my_variable` at offset 8 within the buffer bound to atomic counter binding point 3, then we could write:

Click here to view code image

```
layout (binding = 3, offset = 8) uniform atomic_uint my_variable;
```

To provide storage for the atomic counter, we can now bind a buffer object to the `GL_ATOMIC_COUNTER_BUFFER` indexed binding point. Listing 5.31 shows how to do this.

Click here to view code image

```
// Generate a buffer name
GLuint buf;
glGenBuffers(1, &buf);
// Bind it to the generic GL_ATOMIC_COUNTER_BUFFER target and
// initialize its storage
glBindBuffer(GL_ATOMIC_COUNTER_BUFFER, buf);
glBufferData(GL_ATOMIC_COUNTER_BUFFER, 16 * sizeof(GLuint),
             NULL, GL_DYNAMIC_COPY);
// Now bind it to the fourth indexed atomic counter buffer target
glBindBufferBase(GL_ATOMIC_COUNTER_BUFFER, 3, buf);
```

Listing 5.31: Setting up an atomic counter buffer

Before using the atomic counter in your shader, it's a good idea to reset it first. To do this, you can either call **glBufferSubData()** and pass the address of a variable holding the value you want to reset the counter(s) to; map the buffer using **glMapBufferRange()** and write the values directly into it; or use **glClearBufferSubData()**. Listing 5.32 shows an example of all three methods.

Click here to view code image

```
// Bind our buffer to the generic atomic counter buffer
// binding point
glBindBuffer(GL_ATOMIC_COUNTER_BUFFER, buf);
```

```
// Method 1 - use glBufferSubData to reset an atomic counter
const GLuint zero = 0;
glBufferSubData(GL_ATOMIC_COUNTER_BUFFER, 2 * sizeof(GLuint),
                sizeof(GLuint), &zero);

// Method 2 - Map the buffer and write the value directly into it
GLuint * data =
    (GLuint *)glMapBufferRange(GL_ATOMIC_COUNTER_BUFFER,
                               0, 16 * sizeof(GLuint),
                               GL_MAP_WRITE_BIT |
                               GL_MAP_INVALIDATE_RANGE_BIT);
data[2] = 0;
glUnmapBuffer(GL_ATOMIC_COUNTER_BUFFER);

// Method 3 - use glClearBufferSubData
glClearBufferSubData(GL_ATOMIC_COUNTER_BUFFER,
                     GL_R32UI,
                     2 * sizeof(GLuint),
                     sizeof(GLuint),
                     GL_RED_INTEGER, GL_UNSIGNED_INT,
                     &zero);
```

Listing 5.32: Setting up an atomic counter buffer

Now that you have created a buffer and bound it to an atomic counter buffer target, and you have declared an atomic counter uniform in your shader, you are ready to start counting things. First, to increment an atomic counter, call

Click here to view code image

```
uint atomicCounterIncrement(atomic_uint c);
```

This function reads the current value of the atomic counter, adds 1 to it, writes the new value back to the atomic counter, and returns the original value it read—and it does it all atomically. Because the order of execution between different invocations of your shader is not defined, calling `atomicCounterIncrement` twice in a row won't necessarily give you two consecutive values.

Next, to decrement an atomic counter, call

Click here to view code image

```
uint atomicCounterDecrement(atomic_uint c);
```

This function reads the current value of the atomic counter, subtracts 1 from it, writes the value back into the atomic counter, and returns the *new* value of the counter to you. Notice that this is the opposite of `atomicCounterIncrement`. If only one invocation of a shader is executing, and it calls `atomicCounterIncrement` followed by `atomicCounterDecrement`, it should receive the same value from

both functions. However, in most cases, many invocations of the shader will be executing in parallel; in practice, it is unlikely that you will receive the same value from a pair of calls to these functions.

If you simply want to know the value of an atomic counter, you can call

Click here to view code image

```
uint atomicCounter(atomic_uint c);
```

This function simply returns the current value stored in the atomic counter `c`.

As an example of using atomic counters, Listing 5.33 shows a simple fragment shader that increments an atomic counter each time it executes. This has the effect of producing the screen space area of the objects rendered with this shader in the atomic counter.

Click here to view code image

```
#version 450 core

layout (binding = 0, offset = 0) uniform atomic_uint area;

void main(void)
{
    atomicCounterIncrement(area);
}
```

Listing 5.33: Counting area using an atomic counter

One thing you might notice about the shader in Listing 5.33 is that it doesn't have any regular outputs (variables declared with the **out** storage qualifier) and won't write any data into the framebuffer. In fact, we'll disable writing to the framebuffer while we run this shader. To turn off writing to the framebuffer, we can call

Click here to view code image

```
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
```

To turn framebuffer writes back on again, we can call

Click here to view code image

```
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
```

Because atomic counters are stored in buffers, it's possible now to bind our atomic counter to another buffer target, such as one of the `GL_UNIFORM_BUFFER` targets, and retrieve its value in a shader. This allows us to use the value of an atomic counter to control the execution of shaders that your program runs later. Listing 5.34 shows an example shader that reads the result of our atomic counter through a uniform block and uses it as part of the calculation of its output color.

Click here to view code image

```
#version 450 core

layout (binding = 0) uniform area_block
{
    uint    counter_value;
};

out vec4 color;

uniform float max_area;

void main(void)
{
    float brightness = clamp(float(counter_value) / max_area,
                             0.0, 1.0);

    color = vec4(brightness, brightness, brightness, 1.0);
}
```

Listing 5.34: Using the result of an atomic counter in a uniform block

When we execute the shader in Listing 5.33, it simply counts the area of the geometry that's being rendered. That area then shows up in Listing 5.34 as the first and only member of the `area_block` uniform buffer block. We divide it by the maximum expected area and then use that result as the brightness of further geometry. Consider what happens when we render with these two shaders. If an object is close to the viewer, it will appear larger and cover more screen area—the ultimate value of the atomic counter will be greater. When the object is far from the viewer, it will be smaller and the atomic counter won't reach such a high value. The value of the atomic counter will be reflected in the uniform block in the second shader, affecting the brightness of the geometry it renders.

Synchronizing Access to Atomic Counters

Atomic counters represent locations in buffer objects. While shaders are executing, their values may well reside in special memory inside the graphics processor (which is what makes them faster than simple atomic memory operations on members of shader storage blocks, for example). However, when your shader is done executing, the values of the atomic counters will be written back into memory. As such, incrementing and decrementing atomic counters is considered a form of memory operation and so can be susceptible to the hazards described earlier in this chapter. In fact, the **glMemoryBarrier()** function supports a bit specifically for synchronizing access to atomic counters with other parts of the OpenGL pipeline. Calling

```
glMemoryBarrier(GL_ATOMIC_COUNTER_BARRIER_BIT);
```

will ensure that any access to an atomic counter in a buffer object will reflect updates to that buffer by a shader. You should call **glMemoryBarrier()** with the `GL_ATOMIC_COUNTER_BARRIER_BIT` set when something has *written* to a buffer that you want to see reflected in the values of your atomic counters. If you update the values in a buffer by using an atomic counter and then use that buffer for something else, the bit you include in the `barriers` parameter to **glMemoryBarrier()** should correspond to what you want that buffer to be used for, which will not necessarily include `GL_ATOMIC_COUNTER_BARRIER_BIT`.

Similarly, there is a version of the GLSL `memoryBarrier()` function, called `memoryBarrierAtomicCounter()`, that ensures operations on atomic counters are completed before it returns.