have a texture that is symmetrical around some point and you wish to mirror it once and once only. Figure 5.11 shows an example of this mode in action.

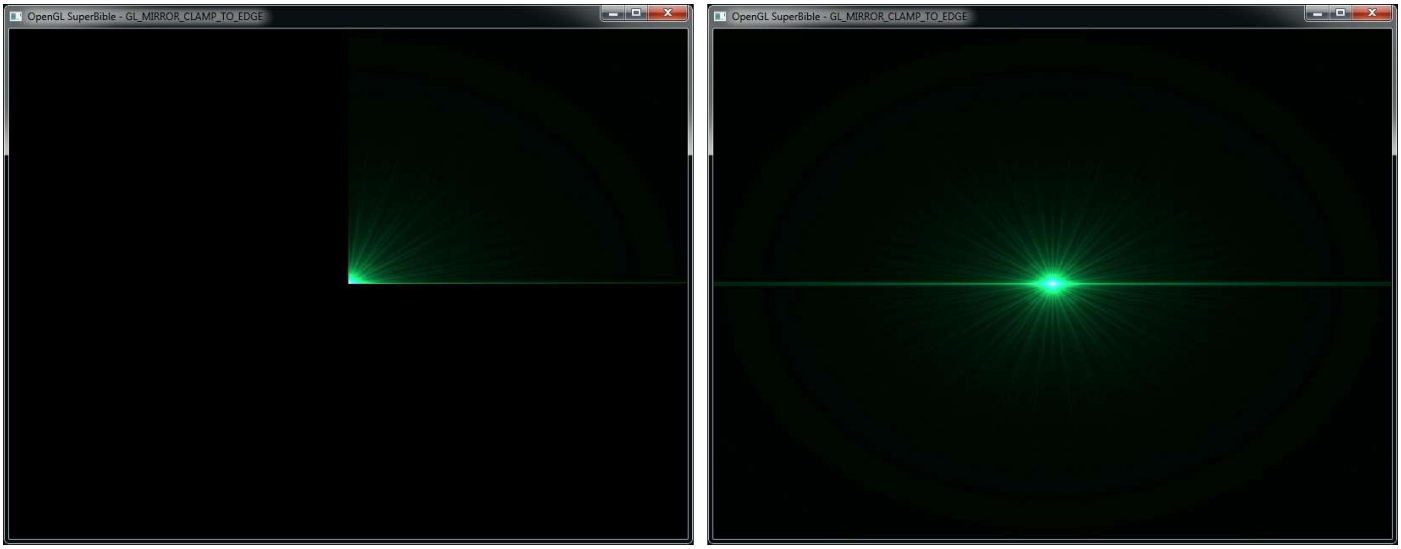

Figure 5.11: `GL_MIRROR_CLAMP_TO_EDGE` in action

In Figure 5.11, the lens flare texture is symmetrical around its center, so only a single quarter of the image need be stored in the texture. This texture is mirrored in both the *x* and *y* dimensions to create the complete image. The picture on the left of Figure 5.11 uses the `GL_CLAMP_TO_BORDER` mode to make the area outside the texture appear black. The image on the right of Figure 5.11 uses the `GL_MIRROR_CLAMP_TO_EDGE` mode to mirror the image horizontally and vertically to show the complete flare.

Array Textures

Previously we discussed the idea that multiple textures could be accessed at once via different texture units. This is extremely powerful and useful as your shader can gain access to several texture objects at the same time by declaring multiple sampler uniforms. We can actually take this a bit further by using a feature called *array textures*. With an array texture, you can load up several 1D, 2D, or cube map images into a single texture object. The concept of having more than one image in a single texture is not new. It happens with mipmapping, as each mip level is a distinct image, and with cube mapping, where each face of the cube map has its own image and even its own set of mip levels. With texture arrays, however, you can have a whole array of texture images bound to a single texture object and then index through them in the shader, thus greatly increasing the amount of texture data available to your application at any one time.

Most texture types have an array equivalent. You can create 1D and 2D array textures, and even cube map array textures. However, you can't create a 3D array texture, as this is not supported by OpenGL. As with cube maps, array textures can have mipmaps. Another interesting thing to note is that if you were to create an array of sampler uniforms in your shader, the value you use to index into that array must be uniform.

However, with a texture array, each lookup into the texture map can come from a different element of the array. In part to distinguish between elements of an array of textures and a single element of an array texture, the elements are usually referred to as *layers*.

You may be wondering what the difference between a 2D array texture and a 3D texture is (or a 1D array texture and a 2D texture, for that matter). The biggest difference is probably that no filtering is applied between the layers of an array texture. Also, the maximum number of array texture layers supported by an implementation may be greater than the maximum 3D texture size, for example.

Loading a 2D Array Texture

To create a 2D array, simply create a new texture object bound to the `GL_TEXTURE_2D_ARRAY` target, allocate storage for it using **glTexStorage3D()**, and then load the images into it using one or more calls to **glTexSubImage3D()**. Notice the use of the 3D versions of the texture storage and data functions. These are required because the `depth` and *z* coordinates passed to them are interpreted as the array element, or *layer*. Simple code to load a 2D array texture is shown in Listing 5.42.

Click here to view code image

```
GLuint tex;

glCreateTextures(GL_TEXTURE_2D_ARRAY1, &tex);

glTextureStorage3D(tex,
                   8,
                   GL_RGBA8,
                   256,
                   256,
                   100);

for (int i = 0; i < 100; i++)
{
    glTextureSubImage3D(tex,
                        0,
                        0, 0,
                        i,
                        256, 256,
                        1,
                        GL_RGBA,
                        GL_UNSIGNED_BYTE,
                        image_data[i]);
}
```

Listing 5.42: Initializing an array texture

Conveniently, the `.KTX` file format supports array textures, so the book's loader code

can load them directly from disk. Simply use `sb7::ktx::file::load` to load an array texture from a file.

To demonstrate texture arrays, we create a program that renders a large number of cartoon aliens raining on the screen. The sample uses an array texture where each slice of the texture holds one of 64 separate images of an alien. The array texture is packed into a single `.KTX` file called `alienarray.ktx`, which we load into a single texture object. To render the alien rain, we draw hundreds of instances of a four-vertex triangle strip, each of which forms a quad. Using the instance number as the index into the texture array gives each quad a different texture, even though all of the quads are drawn with the same command. Additionally, we use a uniform buffer to store a per-instance orientation, *x* offset, and *y* offset that are set up by the application.

In this case, our vertex shader uses no vertex attributes and is shown in its entirety in Listing 5.43.

Click here to view code image

```
#version 450 core

layout (location = 0) in int alien_index;

out VS_OUT
{
    flat int alien;
    vec2 tc;
} vs_out;

struct droplet_t
{
    float x_offset;
    float y_offset;
    float orientation;
    float unused;
};
layout (std140) uniform droplets
{
    droplet_t droplet[256];
};
void main(void)
{
     const vec2[4] position = vec2[4](vec2(-0.5, -0.5),
                                      vec2(0.5, -0.5),
                                      vec2(-0.5, 0.5),
                                      vec2(0.5, 0.5));
     vs_out.tc = position[gl_VertexID].xy + vec2(0.5);
     float co = cos(droplet[alien_index].orientation);
     float so = sin(droplet[alien_index].orientation);
     mat2 rot = mat2(vec2(co, so),
```

```
                    vec2(-so, co));
    vec2 pos = 0.25 * rot * position[gl_VertexID];
    gl_Position = vec4(pos.x + droplet[alien_index].x_offset,
                       pos.y + droplet[alien_index].y_offset,
                       0.5, 1.0);
}
```

Listing 5.43: Vertex shader for the alien rain sample

In our vertex shader, the position of the vertex and its texture coordinate are taken from a hard-coded array. We calculate a per-instance rotation matrix, `rot`, allowing our aliens to spin. Along with the texture coordinate, `vs_out.tc`, we pass the value of `gl_InstanceID` (modulo 64) to the fragment shader via `vs_out.alien`. In the fragment shader, we simply use the incoming values to sample from the texture and write to our output. The fragment shader is shown in Listing 5.44.

Click here to view code image

```
#version 450 core

layout (location = 0) out vec4 color;

in VS_OUT
{
    flat int alien;
    vec2 tc;
} fs_in;

layout (binding = 0) uniform sampler2DArray tex_aliens;

void main(void)
{
    color = texture(tex_aliens, vec3(fs_in.tc, float(fs_in.alien)));
}
```

Listing 5.44: Fragment shader for the alien rain sample

Notice how in Listing 5.44 we sample from our array texture using a **vec3** texture coordinate. This coordinate is constructed from the 2D texture coordinate interpolated from our vertex shader's output as well as from the integer[5] alien index.

5. Yes, it's odd that GLSL expects a floating-point value for the index into an array texture, but that's just the way it is.

Accessing Texture Arrays

In the fragment shader (shown in Listing 5.44), we declare our sampler for the 2D array texture, **sampler2DArray**. To sample this texture we use the `texture` function as

normal, but pass in a three-component texture coordinate. The first two components of this texture coordinate, the *s* and *t* components, are used as typical two-dimensional texture coordinates. The third component, the *p* element, is actually an integer index into the texture array. Recall that we set this index in the vertex shader, and it is going to vary from 0 to 63, with a different value for each alien.

The complete rendering loop for the alien rain sample is shown in Listing 5.45.

Click here to view code image

```
void render(double currentTime)
{
    static const GLfloat black[] = { 0.0f, 0.0f, 0.0f, 0.0f };
    float t = (float)currentTime;

    glViewport(0, 0, info.windowWidth, info.windowHeight);
    glClearBufferfv(GL_COLOR, 0, black);

    glUseProgram(render_prog);

    glBindBufferBase(GL_UNIFORM_BUFFER, 0, rain_buffer);
    vmath::vec4 * droplet =
        (vmath::vec4 *)glMapBufferRange(
                            GL_UNIFORM_BUFFER,
                            0,
                            256 * sizeof(vmath::vec4),
                            GL_MAP_WRITE_BIT |
                            GL_MAP_INVALIDATE_BUFFER_BIT);

    for (int i = 0; i < 256; i++)
    {
        droplet[i][0] = droplet_x_offset[i];
        droplet[i][1] = 2.0f - fmodf((t + float(i)) *
                                     droplet_fall_speed[i], 4.31f);
        droplet[i][2] = t * droplet_rot_speed[i];
        droplet[i][3] = 0.0f;
    }
    glUnmapBuffer(GL_UNIFORM_BUFFER);

    int alien_index;
    for (alien_index = 0; alien_index < 256; alien_index++)
    {
        glVertexAttribI1i(0, alien_index);
        glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    }
}
```

Listing 5.45: Rendering loop for the alien rain sample

As you can see, there is only a simple loop around one drawing command in our rendering function. On each frame, we update the values of the data in the

`rain_buffer` buffer object that we use to store our per-droplet values. Then, we execute a loop of 256 calls to **glDrawArrays()**, which will draw 256 individual aliens. On each iteration of the loop, we update the `alien_index` input to the vertex shader. Note that we use the **glVertexAttribI*()** variant of **glVertexAttrib*()**, as we are using an integer input to our vertex shader. The final output of the alien rain sample program is shown in Figure 5.12.

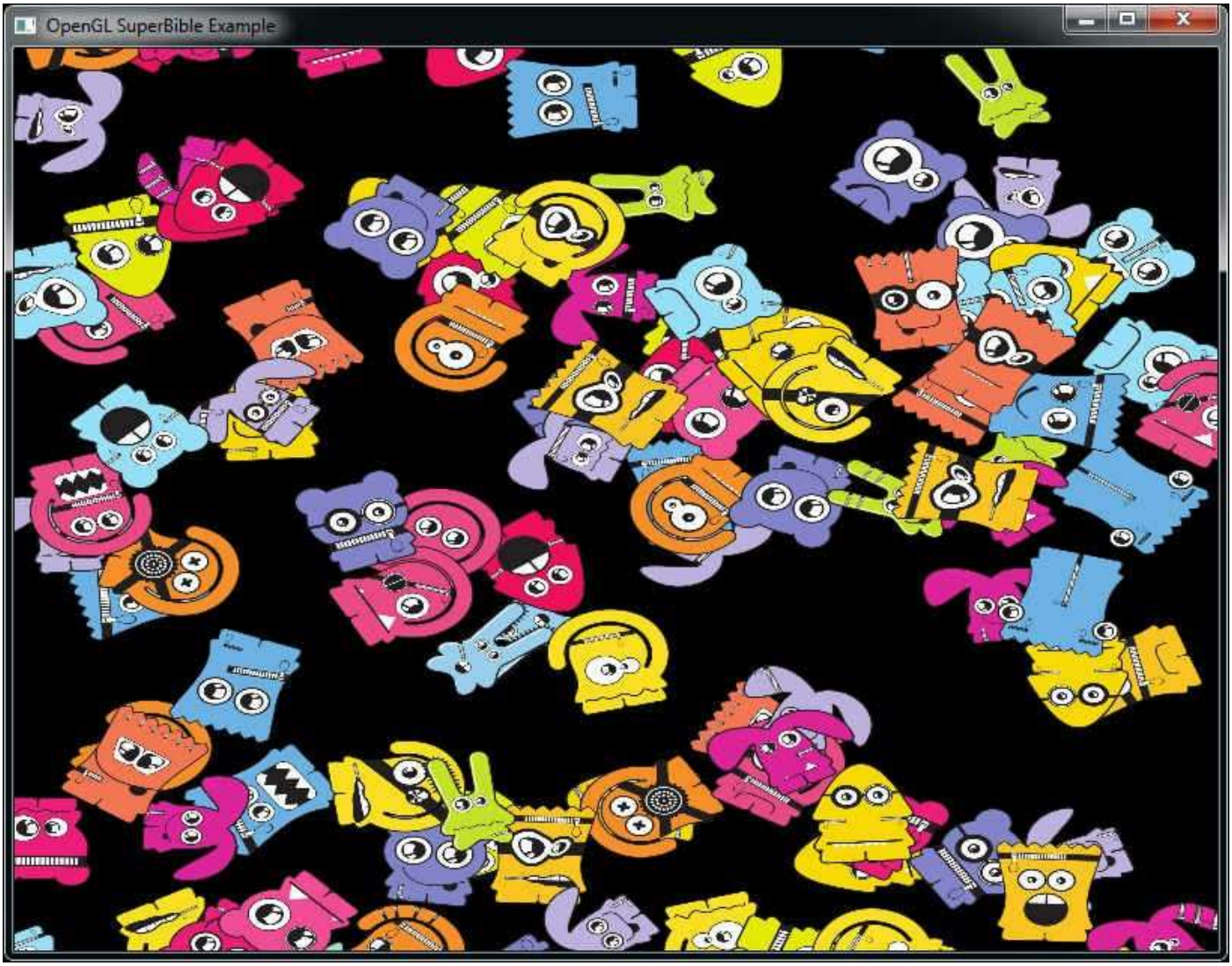

Figure 5.12: Output of the alien rain sample

Writing to Textures in Shaders

A texture object is a collection of images that, when the mipmap chain is included, supports filtering, texture coordinate wrapping, and so on. Not only does OpenGL allow you to read from textures with all of those features, it also allows you to *write to* textures directly in your shaders. Just as you use a sampler variable in shaders to represent an entire texture and the associated sampler parameters (whether from a sampler object or from the texture object itself), so you can use an *image* variable to

represent a single image from a texture.

Image variables are declared just like sampler uniforms. There are several types of image variables that represent different data types and image dimensionalities. Table 5.9 shows the image types available to OpenGL.

Image Type	Description
`image1D`	1D image
`image2D`	2D image
`image3D`	3D image
`imageCube`	Cube map image
`imageCubeArray`	Cube map array image
`imageRect`	Rectangle image
`image1DArray`	1D array image
`image2DArray`	2D array image
`imageBuffer`	Buffer image
`image2DMS`	2D multisample image
`image2DMSArray`	2D multisample array image

Table 5.9: Image Types

First, you need to declare an image variable as a uniform so that you can associate it with an *image unit*. Such a declaration generally looks like this:

```
uniform image2D my_image;
```

Once you have an image variable, you can read from it using the `imageLoad` function and write into it using the `imageStore` function. Both of these functions are *overloaded*, which means that there are multiple versions of each for various parameter types. The versions for the **image2D** type are

Click here to view code image

```
vec4 imageLoad(readonly image2D image, ivec2 P);
void imageStore(image2D image, ivec2 P, vec4 data);
```

The `imageLoad()` function will read the data from `image` at the coordinates specified in `P` and return it to your shader. Similarly, the `imageStore()` function will take the values you provide in `data` and store them into `image` at `P`. Notice that the type of `P` is an *integer* type (an integer vector in the case of 2D images). This is just like the `texelFetch()` function—no filtering is performed for loads and filtering really doesn't make sense for stores. The dimension of `P` and the return type of the function depend on the type of the `image` parameter.

Just as with sampler types, image variables can represent floating-point data stored in images. However, it's also possible to store signed and unsigned integer data in images, in which case the image type is prefixed with `i` or `u`, respectively (as in **iimage2D** and **uimage2D**). When an integer image variable is used, the return type of the `imageLoad` function and the data type of the `data` parameter to `imageStore` change appropriately. For example, we have

Click here to view code image

```
ivec4 imageLoad(readonly iimage2D image, ivec2 P);
void  imageStore(iimage2D image, ivec2 P, ivec4 data);
uvec4 imageLoad(readonly uimage2D image, ivec2 P);
void  imageStore(uimage2D image, ivec2 P, uvec4 data);
```

To bind a texture for load and store operations, you need to bind it to an *image unit* using the **glBindImageTexture()** function, whose prototype is

Click here to view code image

```
void glBindImageTexture(GLuint unit,
                        GLuint texture,
                        GLint level,
                        GLboolean layered,
                        GLint layer,
                        GLenum access,
                        GLenum format);
```

The function looks like it has a lot of parameters, but they're all fairly self-explanatory. First, the `unit` parameter is a zero-based index of the image unit to which you want to bind the image. Next, the `texture` parameter is the name of a texture object that you've created using **glCreateTextures()** and allocated storage for with **glTextureStorage2D()** (or the appropriate function for the type of texture you're using). `level` specifies which mipmap level you want to access in your shader, starting with zero for the base level and progressing to the number of mipmap levels in the image.

The `layered` parameter should be set to `GL_FALSE` if you want to bind a single layer of an array texture as a regular 1D or 2D image, in which case the `layer` parameter specifies the index of that layer. Otherwise, `layered` should be set to `GL_TRUE` and a whole level of an arrray texture will be bound to the image unit (with `layer` being ignored).

Finally, the `access` and `format` parameters describe how you will use the data in the image. `access` should be one of `GL_READ_ONLY`, `GL_WRITE_ONLY`, or `GL_READ_WRITE` to say that you plan to only read, only write, or do both to the image, respectively. The `format` parameter specifies the format in which the data in the image should be interpreted. There is a lot of flexibility here, with the only real

requirement being that the image's internal format (the one you specified in **glTextureStorage2D()**) is in the same *class* as the one specified in the `format` parameter. Table 5.10 lists the acceptable image formats and their classes.

Format	Class
GL_RGBA32F	4×32
GL_RGBA32I	4×32
GL_RGBA32UI	4×32
GL_RGBA16F	4×16
GL_RGBA16UI	4×16
GL_RGBA16I	4×16
GL_RGBA16_SNORM	4×16
GL_RGBA16	4×16
GL_RGBA8UI	4×8
GL_RGBA8I	4×8
GL_RGBA8_SNORM	4×8
GL_RGBA8	4×8
GL_R11F_G11F_B10F	(a)
GL_RGB10_A2UI	(b)
GL_RGB10_A2	(b)
GL_RG32F	2×32
GL_RG32UI	2×32
GL_RG32I	2×32
GL_RG16F	2×16
GL_RG16UI	2×16
GL_RG16I	2×16

GL_RG16_SNORM	2×16
GL_RG16	2×16
GL_RG8UI	2×8
GL_RG8I	2×8
GL_RG8	2×8
GL_RG8_SNORM	2×8
GL_R32F	1×32
GL_R32UI	1×32
GL_R32I	1×32
GL_R16F	1×16
GL_R16UI	1×16
GL_R16I	1×16
GL_R16_SNORM	1×16
GL_R16	1×16
GL_R8UI	1×8
GL_R8I	1×8
GL_R8	1×8
GL_R8_SNORM	1×8

Table 5.10: Image Data Format Classes

Referring to Table 5.10, you can see that the `GL_RGBA32F`, `GL_RGBA32I`, and `GL_RGBA32UI` formats are in the same format class (4×32), which means that you can take a texture that has a `GL_RGBA32F` internal format and bind one of its levels to an image unit using the `GL_RGBA32I` or `GL_RGBA32UI` image format. When you store into an image, the appropriate number of bits from your source data are chopped off and written to the image as is. However, if you want to read from an image, you must also supply a matching image format using a *format layout qualifier* in your shader code.

The `GL_R11F_G11F_B10F` format, which has the marker (a) for its format class, and `GL_RGB10_A2UI` and `GL_RGB10_A2`, which have the marker (b) for their format class, have their own special classes. `GL_R11F_G11F_B10F` is not compatible with anything else, and `GL_RGB10_A2UI` and `GL_RGB10_A2` are compatible only with each other.

The appropriate format layout qualifiers for each of the various image formats are shown in Table 5.11.

Format	Format Qualifier
GL_RGBA32F	rgba32f
GL_RGBA32I	rgba32i
GL_RGBA32UI	rgba32ui
GL_RGBA16F	rgba16f
GL_RGBA16UI	rgba16ui
GL_RGBA16I	rgba16i
GL_RGBA16_SNORM	rgba16_snorm
GL_RGBA16	rgba16
GL_RGB10_A2UI	rgb10_a2ui
GL_RGB10_A2	rgb10_a2
GL_RGBA8UI	rgba8ui
GL_RGBA8I	rgba8i
GL_RGBA8_SNORM	rgba8_snorm
GL_RGBA8	rgba8
GL_R11F_G11F_B10F	r11f_g11f_b10f
GL_RG32F	rg32f

GL_RG32UI	rg32ui
GL_RG32I	rg32i
GL_RG16F	rg16f
GL_RG16UI	rg16ui
GL_RG16I	rg16i
GL_RG16_SNORM	rg16_snorm
GL_RG16	rg16
GL_RG8UI	rg8ui
GL_RG8I	rg8i
GL_RG8_SNORM	rg8_snorm
GL_RG8	rg8
GL_R32F	r32f
GL_R32UI	r32ui
GL_R32I	r32i
GL_R16F	r16f
GL_R16UI	r16ui
GL_R16I	r16i
GL_R16_SNORM	r16_snorm
GL_R16	r16
GL_R8UI	r8ui
GL_R8I	r8i
GL_R8_SNORM	r8_snorm
GL_R8	r8

Table 5.11: Image Data Format Classes

Listing 5.46 shows an example fragment shader that copies data from one image to another using image loads and stores, logically inverting that data along the way.

Click here to view code image

```
#version 450 core

// Uniform image variables:
// Input image - note use of format qualifier because of loads
layout (binding = 0, rgba32ui) readonly uniform uimage2D image_in;
// Output image
layout (binding = 1) uniform writeonly uimage2D image_out;

void main(void)
{
    // Use fragment coordinate as image coordinate
```

```
    ivec2 P = ivec2(gl_FragCoord.xy);

    // Read from input image
    uvec4 data = imageLoad(image_in, P);

    // Write inverted data to output image
    imageStore(image_out, P, ~data);
}
```

Listing 5.46: Fragment shader performing image loads and stores

Obviously, the shader shown in Listing 5.46 is quite trivial. However, the power of image loads and stores is that you can include any number of them in a single shader and their coordinates can be anything. This means that a fragment shader is not limited to writing out to a fixed location in the framebuffer, but can write anywhere in an image, and can write to multiple images by using multiple image uniforms. Furthermore, any shader stage can write data into images, not just fragment shaders. Be aware, though, that with this power comes a lot of responsibility. It's perfectly easy for your shader to trash its own data—if multiple shader invocations write to the same location in an image, it's not well defined what will happen unless you use *atomics*, which are described in the context of images in the next section.

Atomic Operations on Images

Just as with the shader storage blocks described earlier in this chapter, you can perform *atomic operations* on data stored in images. Again, an atomic operation is a sequence of a read, a modification, and a write that must be indivisible to achieve the desired result. Also, like atomic operations on members of a shader storage block, atomic operations on images are performed using a number of built-in functions in GLSL. These functions are listed in Table 5.12.

Atomic Function	Behavior
`imageAtomicAdd`	Reads from `image` at `P`, adds it to `data`, writes the result back to `image` at `P`, and then returns the value originally stored in `image` at `P`.
`imageAtomicAnd`	Reads from `image` at `P`, logically ANDs it with `data`, writes the result back to `image` at `P`, and then returns the value originally stored in `image` at `P`.
`imageAtomicOr`	Reads from `image` at `P`, logically ORs it with `data`, writes the result back to `image` at `P`, and then returns the value originally stored in `image` at `P`.
`imageAtomicXor`	Reads from `image` at `P`, logically exclusive ORs it with `data`, writes the result back to `image` at `P`, and then returns the value originally stored in `image` at `P`.
`imageAtomicMin`	Reads from `image` at `P`, determines the minimum of the retrieved value and `data`, writes the result back to `image` at `P`, and returns the value originally stored in `image` at `P`.
`imageAtomicMax`	Reads from `image` at `P`, determines the maximum of the retrieved value and `data`, writes the result back to `image` at `P`, and returns the value originally stored in `image` at `P`.
`imageAtomicExchange`	Reads from `image` at `P`, writes the value of `data` into `mem`, and then returns the value originally stored in `image` at `P`.
`imageAtomicCompSwap`	Reads from `image` at `P`, compares the retrieved value with `comp` and, if they are equal, writes `data` into `image` at `P`, and returns the value originally stored in `image` at `P`.

Table 5.12: Atomic Operations on Images

For all of the functions listed in Table 5.12 except for `imageAtomicCompSwap`, the parameters are an image variable, a coordinate, and a piece of data. The dimension of the coordinate depends on the type of image variable. 1D images use a single integer

coordinate, 2D images and 1D array images take a 2D integer vector (i.e., **ivec2**), and 3D images and 2D array images take a 3D integer vector (i.e., **ivec3**). For example, we have

Click here to view code image

```
uint imageAtomicAdd(uimage1D image, int P, uint data);
uint imageAtomicAdd(uimage2D image, ivec2 P, uint data);
uint imageAtomicAdd(uimage3D image, ivec3 P, uint data);
```

and so on. The `imageAtomicCompSwap` is unique in that it takes an additional parameter, `comp`, which it compares with the existing content in memory. If the value of `comp` is equal to the value already in memory, then it is replaced with the value of `data`. The prototypes of `imageAtomicCompSwap` include

Click here to view code image

```
uint imageAtomicCompSwap(uimage1D image, int P, uint comp, uint data);
uint imageAtomicCompSwap(uimage2D image, ivec2 P, uint comp, uint data);
uint imageAtomicCompSwap(uimage3D image, ivec3 P, uint comp, uint data);
```

All of the atomic functions return the data that was originally in memory before the operation was performed. This is useful if you wish to append data to a list, for example. To do this, you would simply determine how many items you want to append to the list, call `imageAtomicAdd` with the number of elements, and then start writing your new data into memory at the location that it returns. Although you can't add an arbitrary number to an atomic counter (and the number of atomic counters supported in a single shader is usually not great), you can do similar things with shader storage buffers.

The memory you write to could be a shader storage buffer or another image variable. If the image containing the "filled count" variables is pre-initialized to 0, then the first shader invocation to attempt to append to the list will receive 0 as the location and write there, the next invocation will receive whatever the first added, the next will receive whatever the third added, and so on.

Another application for atomics is constructing data structures such as linked lists in memory. To build a linked list from a shader, you need three pieces of storage—the first is somewhere to store the list items, the second is somewhere to store the item count, and the third is the "head pointer," which is the index of the last item in the list. Again, you can use a shader storage buffer to store items for the linked list, an atomic counter to store the current item count, and an image to store the head pointer for the list(s). To append an item to the list, you would follow three steps:

1. Increment the atomic counter and retrieve its previous value, which is returned by `atomicCounterIncrement`.
2. Use `imageAtomicExchange` to exchange the updated counter value with the

current head pointer.

3. Store your data into the data store. The structure for each element includes a *next* index, which you fill with the previous value of the head pointer retrieved in step 2.

If the "head pointer" image is a 2D image the size of the framebuffer, then you can use this method to create a per-pixel list of fragments. You can later walk this list and perform whatever operations you like. The shader shown in Listing 5.47 demonstrates how to append fragments to a linked list stored in a shader storage buffer by using a 2D image to store the head pointers and an atomic counter to keep the fill count.

Click here to view code image

```
#version 450 core

// Atomic counter for filled size
layout (binding = 0, offset = 0) uniform atomic_uint fill_counter;

// 2D image to store head pointers
layout (binding = 0) uniform uimage2D head_pointer;

// Shader storage buffer containing appended fragments
struct list_item
{
    vec4        color;
    float       depth;
    int         facing;
    uint        next;
};

layout (binding = 0, std430) buffer list_item_block
{
    list_item item[];
};

// Input from vertex shader
in VS_OUT
{
    vec4 in;
} fs_in;

void main(void)
{
    ivec2 P = ivec2(gl_FragCoord.xy);

    uint index = atomicCounterIncrement(fill_counter);

    uint old_head = imageAtomicExchange(head_pointer, P, index);

    item[index].color = fs_in.color;
```

```
    item[index].depth = gl_FragCoord.z;
    item[index].facing = gl_FrontFacing ? 1 : 0;
    item[index].next = old_head;
}
```

Listing 5.47: Filling a linked list in a fragment shader

You might notice the use of the `gl_FrontFacing` built-in variable. This is a Boolean input to the fragment shader whose value is generated by the back-face culling stage described in the "Primitive Assembly, Clipping, and Rasterization" section in Chapter 3. Even if back-face culling is disabled, this variable will still contain **true** if the polygon is considered front-facing and **false** otherwise.

Before executing this shader, the head pointer image is cleared to a known value that can't possibly be the index of an item in the list (such as the maximum value of an unsigned integer) and the atomic counter is reset to 0. The first item appended will be item 0, that value will be written to the head pointer, and its "next" index will contain the reset value of the head pointer image. The next value appended to the list will be at index 1, which is written to the head pointer, the old value of which (0) is written to the "next" index and so on. The result is that the head pointer image contains the index of the last item appended to the list and each item contains the index of the previous one appended. Eventually, the "next" index of an item will be the value originally used to clear the head image, which indicates that the end of the list has been reached.

To traverse the list, we load the index of the first item in it from the head pointer image and read it from the shader storage buffer. For each item, we simply follow the "next" index until we reach the end of the list, or until the maximum number of fragments has been traversed (which protects us from accidentally running off the end of the list). The shader shown in Listing 5.48 shows an example. The shader walks the linked list, keeping a running total of the depth of the fragments stored for each pixel. The depth value of front-facing primitives is added to the running total and the depth value of back-facing primitives is subtracted from the total. The result is the total filled depth of the interior of convex objects, which can be used to render volumes and other filled spaces.

Click here to view code image

```
#version 450 core

// 2D image to store head pointers
layout (binding = 0, r32ui) coherent uniform uimage2D head_pointer;

// Shader storage buffer containing appended fragments
struct list_item
{
    vec4        color;
```

```
    float        depth;
    int          facing;
    uint         next;
};

layout (binding = 0, std430) buffer list_item_block
{
   list_item    item[];
};

layout (location = 0) out vec4 color;

const uint max_fragments = 10;

void main(void)
{
    uint frag_count = 0;
    float depth_accum = 0.0;
    ivec2 P = ivec2(gl_FragCoord.xy);

    uint index = imageLoad(head_pointer, P).x;

    while (index != 0xFFFFFFFF && frag_count < max_fragments)
    {
        list_item this_item = item[index];

        if (this_item.facing != 0)
        {
            depth_accum -= this_item.depth;
        }
        else
        {
            depth_accum += this_item.depth;
        }

        index = this_item.next;
        frag_count++;
    }

    depth_accum *= 3000.0;

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

Listing 5.48: Traversing a linked list in a fragment shader

The result of rendering with the shaders of Listings 5.47 and 5.48 is shown in Figure 5.13.

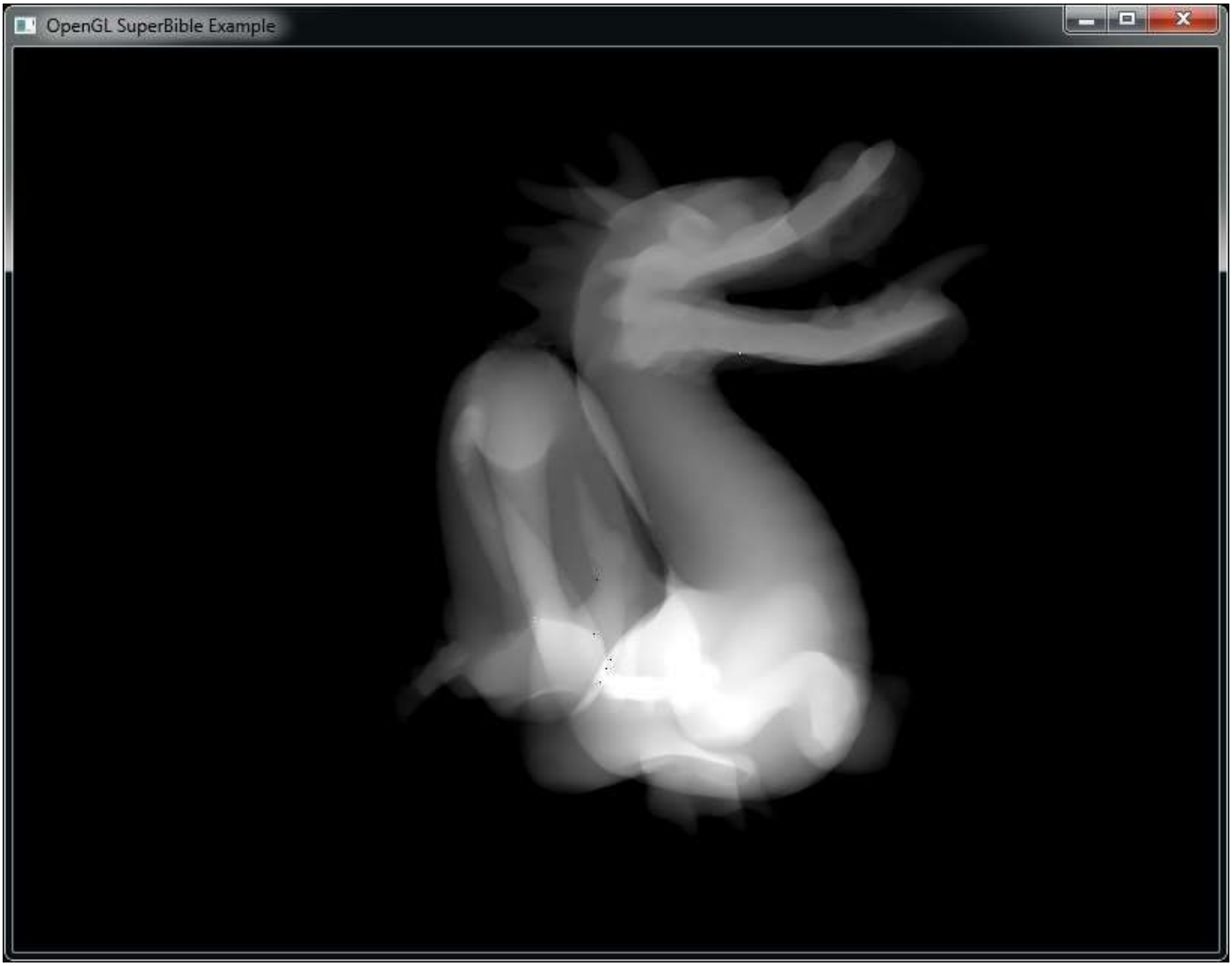

Figure 5.13: Resolved per-fragment linked lists

Synchronizing Access to Images

As images represent large regions of memory and we have just explained how to write directly into images from your shaders, you may have guessed that we'll now explain the memory barrier types that you can use to synchronize access to that memory. Just as with buffers and atomic counters, you can call

Click here to view code image

```
glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BIT);
```

You should call **glMemoryBarrier()** with the `GL_SHADER_IMAGE_ACCESS_BIT` set when something has *written* to an image that you want read from images later—including other shaders.

Similarly, a version of the GLSL `memoryBarrier()` function, `memoryBarrierImage()`, ensures that operations on images from inside your shader are completed before it returns.

Texture Compression

Textures can take up an incredible amount of space. Modern games can easily use a gigabyte or more of texture data in a single level. That's a lot of data! Where do you put it all? Textures are an important part of making rich, realistic, and impressive scenes, but if you can't load all of the data onto the GPU, your rendering will be slow, if not impossible. One way to deal with storing and using a large amount of texture data is to compress the data. Compressed textures have two major benefits. First, they reduce the amount of storage space required for image data. Although the texture formats supported by OpenGL are generally not compressed as aggressively as in formats such as JPEG, they do provide substantial space benefits. The second (and possibly more important) benefit is that, because the graphics processor needs to read less data when fetching from a compressed texture, less *memory bandwidth* is required when compressed textures are used.

OpenGL supports a number of compressed texture formats. All OpenGL implementations support at least the compression schemes listed in Table 5.13.

Formats (GL_COMPRESSED_*)	Type
RED	Generic
RG	Generic
RGB	Generic
RGBA	Generic
SRGB	Generic
SRGB_ALPHA	Generic
RED_RGTC1	RGTC
SIGNED_RED_RGTC1	RGTC
RG_RGTC2	RGTC
SIGNED_RG_RGTC2	RGTC
RGBA_BPTC_UNORM	BPTC
SRGB_ALPHA_BPTC_UNORM	BPTC
RGB_BPTC_SIGNED_FLOAT	BPTC
RGB_BPTC_UNSIGNED_FLOAT	BPTC
RGB8_ETC2	ETC2
SRGB8_ETC2	ETC2
RGB8_PUNCHTHROUGH_ALPHA1_ETC2	ETC2
SRGB8_PUNCHTHROUGH_ALPHA1_ETC2	ETC2
RGBA8_ETC2_EAC	ETC2
SRGB8_ALPHA8_ETC2_EAC	ETC2
R11_EAC	EAC
SIGNED_R11_EAC	EAC
RG11_EAC	EAC
SIGNED_RG11_EAC	EAC

Table 5.13: Native OpenGL Texture Compression Formats

The first six formats listed in Table 5.13 are generic and allow the OpenGL driver to decide which compression mechanism to use. As a result, your driver can use the format that best suits the current conditions. The catch is that the choice is implementation specific; although your code will work on many platforms, the result of rendering with them might not be the same.

The RGTC (Red–Green Texture Compression) format breaks a texture image into 4 × 4 texel blocks, compressing the individual channels within each block using a series of codes. This compression mode works only for one- and two-channel signed and unsigned textures, and only for certain texel formats. You don’t need to worry about the

exact compression scheme unless you are planning on writing a compressor. Just note that the space savings from using RGTC is 50%.

The BPTC (Block Partitioned Texture Compression) format also breaks textures up into blocks of 4 × 4 texels, each represented as 128 bits (16 bytes) of data in memory. The blocks are encoded using a rather complex scheme that essentially comprises a pair of endpoints and a representation of the position on a line between those two endpoints. It allows the endpoints to be manipulated to generate a variety of values as output for each texel. The BPTC formats are capable of compressing 8-bit per-channel normalized data and 32-bit per-channel floating-point data. The compression ratio for BPTC formats ranges from 25% for RGBA floating-point data to 33% for RGB 8-bit data.

Ericsson Texture Compression (ETC2) and Ericsson Alpha Compression (EAC)[6] are low-bandwidth formats that are also[7] available in OpenGL ES 3.0. They are designed for extremely low bit-per-pixel applications such as those found in mobile devices, which have substantially less memory bandwidth than the high-performance GPUs found in desktop and workstation computers.

6. Although this is the official acronym, it's a bit of a misnomer: EAC can be used for more than just alpha compression.

7. The EAC and ETC2 formats were added to OpenGL 4.3 in an effort to drive convergence between desktop and mobile versions of the API. At the time of writing, few if any desktop GPUs actually supported them natively, with most OpenGL implementations decompressing the data you give them. Use them with caution.

Your implementation may also support other compressed formats, such as S3TC[8] and ETC1. You should check for the availability of formats not required by OpenGL before attempting to use them. The best way to do so is to check for support of the related extension. For example, if your implementation of OpenGL supports the S3TC format, it will advertise the `GL_EXT_texture_compression_s3tc` extension string.

8. S3TC is also known as the earlier version of the DXT format.

Using Compression

You can ask OpenGL to compress a texture in some formats when you load it, although it's strongly recommended to compress textures yourself and store the compressed texture in a file. If OpenGL does support compression for your selected format, all you have to do is request that the internal format be one of the compressed formats; OpenGL will then take your uncompressed data and compress it as the texture image is loaded. There is no real difference in how you use compressed textures and uncompressed textures. The GPU handles the conversion when it samples from the texture. Many imaging tools used for creating textures and other images allow you to save your data directly in a compressed format.

The `.KTX` file format allows compressed data to be stored in it and the book's texture loader will load compressed images transparently to your application. You can check

whether a texture is compressed by calling **glGetTexLevelParameteriv()** with one of two parameters. First, you can check the `GL_TEXTURE_INTERNAL_FORMAT` parameter of the texture and explicitly test whether it's one of the compressed formats. To do this, either keep a lookup table of recognized formats in your application or call **glGetInternalFormativ()** with the parameter `GL_TEXTURE_COMPRESSED`. Alternatively, simply pass the `GL_TEXTURE_COMPRESSED` parameter directly to **glGetTexLevelParameteriv()**, which will return `GL_TRUE` if the texture has compressed data in it and `GL_FALSE` otherwise.

Once you have loaded a texture using a nongeneric compressed internal format, you can get the compressed image back by calling **glGetCompressedTexImage()**. Just pick the texture target and mipmap level you are interested in. Because you may not know how the image is compressed or which format is used, you should check the image size to make sure you have enough room for the whole surface. You can do this by calling **glGetTexParameteriv()** and passing the `GL_TEXTURE_COMPRESSED_IMAGE_SIZE` token.

Click here to view code image

```
Glint imageSize = 0;
glGetTextureParameteriv(GL_TEXTURE_2D,
                        GL_TEXTURE_COMPRESSED_IMAGE_SIZE,
                        &imageSize);
void *data = malloc(imageSize);
glGetCompressedTextureImage(GL_TEXTURE_2D, 0, data);
```

If you want to load compressed texture images yourself rather than using the book's `.KTX` loader, you can call **glTextureStorage2D()** or **glTextureStorage3D()** with the desired compressed internal format to allocate storage for the texture, and then call **glCompressedTextureSubImage()** 2D or **glCompressedTextureSubImage()** 3D to upload data into it. When you do this, you need to ensure that `xoffset`, `yoffset`, and other parameters obey the texture format's specific rules. In particular, most texture compression formats compress blocks of texels. These blocks are usually sizes such as 4 × 4 texels. The regions that you update with **glCompressedTexSubImage2D()** need to line up on block boundaries for these formats to work.

Shared Exponents

Although shared exponent textures are not technically a compressed format in the truest sense, they do allow you to use floating-point texture data while saving storage space. Instead of storing an exponent for each of the R, G, and B values, shared exponent formats use the same exponent value for the whole texel. The fractional and exponential parts of each value are stored as integers and then assembled when the texture is

sampled. For the format `GL_RGB9_E5`, 9 bits are used to store each color and 5 bits are the common exponent for all channels. This format packs three floating-point values into 32 bits; that's a savings of 67%! To make use of shared exponents, you can get the texture data directly in this format from a content creation tool or write a converter that compresses your float RGB values into a shared exponent format.

Texture Views

In most cases when you're using textures, you'll know ahead of time which format your textures are in and what you're going to use them for, and your shaders will match the data they're fetching. For instance, a shader that expects to read from a 2D array texture might declare a sampler uniform as a **sampler2DArray**. Likewise, a shader that expects to read from an integer format texture might declare a corrseponding sampler as **isampler2D**. However, there may be times when the textures you create and load might not match what your shaders expect. In this case, you can use *texture views* to reuse the texture data in one texture object with another. This has two main use cases (although there are certainly many more):

- A texture view can be used to "pretend" that a texture of one type is actually a texture of a different type. For example, you can take a 2D texture and create a view of it that treats it as a 2D array texture with only one layer.
- A texture view can be used to pretend that the data in the texture object is actually a different format than what is really stored in memory. For example, you might take a texture with an internal format of `GL_RGBA32F` (i.e., four 32-bit floating-point components per texel) and create a view of it that sees them as `GL_RGBA32UI` (four 32-bit unsigned integers per texel) so that you can get at the individual bits of the texels.

Of course, you can do both of these things at the same time—that is, take a texture and create a view of it with both a different format and a different type.

Creating Texture Views

To create a view of a texture, we use the **glTextureView()** function, whose prototype is

Click here to view code image

```
void glTextureView(GLuint texture,
                   GLenum target,
                   GLuint origtexture,
                   GLenum internalformat,
                   GLuint minlevel,
                   GLuint numlevels,
                   GLuint minlayer,
                   GLuint numlayers);
```

The first parameter, `texture`, is the name of the texture object you'd like to make into a view. You should get this name from a call to **glGenTextures()**. Next, `target` specifies which *type* of texture you'd like to create. This can be pretty much any of the texture targets (`GL_TEXTURE_1D`, `GL_TEXTURE_CUBE_MAP`, or `GL_TEXTURE_2D_ARRAY`, for example), but it must be *compatible* with the type of the original texture, whose name is given in `origtexture`. The compatibility between various targets is indicated in Table 5.14.

If `origtexture` is... (GL_TEXTURE_*)	You can create a view of it as... (GL_TEXTURE_*)
1D	1D or 1D_ARRAY
2D	2D or 2D_ARRAY
3D	3D
CUBE_MAP	CUBE_MAP, 2D, 2D_ARRAY, or CUBE_MAP_ARRAY
RECTANGLE	RECTANGLE
BUFFER	*none*
1D_ARRAY	1D or 1D_ARRAY
2D_ARRAY	2D or 2D_ARRAY
CUBE_MAP_ARRAY	CUBE_MAP, 2D, 2D_ARRAY, or CUBE_MAP_ARRAY
2D_MULTISAMPLE	2D_MULTISAMPLE or 2D_MULTISAMPLE_ARRAY
2D_MULTISAMPLE_ARRAY	2D_MULTISAMPLE or 2D_MULTISAMPLE_ARRAY

Table 5.14: Texture View Target Compatibility

As you can see, for most texture targets you can at least create a view of the texture with the same target. The exception is buffer textures, which are essentially already views of a buffer object—you can simply attach the same buffer object to another buffer texture to get another view of its data.

The `internalformat` parameter specifies the internal format for the new texture view. This must be compatible with the internal format of the original texture. This point can be tricky to understand, so we'll explain it in a moment.

The last four parameters allow you to make a view of a *subset* of the original texture's data. The `minlevel` and `numlevels` parameters specify the first mipmap level and the number of mipmap levels, respectively, to include in the view. This allows you to create a texture view that represents part of an entire mipmap pyramid of another

texture. For example, to create a texture that represents just the base level (level 0) of another texture, you can set `minlevel` to 0 and `numlevels` to 1. To create a view that represents the four lowest-resolution mipmaps of a ten-level texture, you would set `minlevel` to 6 and `numlevels` to 4.

Similarly, `minlayer` and `numlayers` can be used to create a view of a subset of the layers of an array texture. For instance, if you want to create an array texture view that represents the middle four layers of a 20-layer array texture, you can set `minlayer` to 8 and `numlayers` to 4. Whatever you choose for the `minlevel`, `numlevels`, `minlayer`, and `numlayers` parameters, they must be consistent with the source and destination textures. For example, if you want to create a non-array texture view representing a single layer of an array texture, you must set `minlayer` to a layer that actually exists in the source texture and `numlayers` to 1 because the destination doesn't have any layers (rather, it effectively has one layer).

We mentioned that the internal format of the source texture and the new texture view (specified in the `internalformat` parameter) must be compatible with each other. To be compatible, two formats must be in the same *class*. Several format classes are available; they are listed, along with the internal formats that are members of that class, in Table 5.15.

Format Class	Members of the Class
128-bit	GL_RGBA32F, GL_RGBA32UI, GL_RGBA32I
96-bit	GL_RGB32F, GL_RGB32UI, GL_RGB32I
64-bit	GL_RGBA16F, GL_RG32F, GL_RGBA16UI, GL_RG32UI, GL_RGBA16I, GL_RG32I, GL_RGBA16, GL_RGBA16_SNORM
48-bit	GL_RGB16, GL_RGB16_SNORM, GL_RGB16F, GL_RGB16UI, GL_RGB16I
32-bit	GL_RG16F, GL_R11F_G11F_B10F, GL_R32F, GL_RGB10_A2UI, GL_RGBA8UI, GL_RG16UI, GL_R32UI, GL_RGBA8I, GL_RG16I, GL_R32I, GL_RGB10_A2, GL_RGBA8, GL_RG16, GL_RGBA8_SNORM, GL_RG16_SNORM, GL_SRGB8_ALPHA8, GL_RGB9_E5
24-bit	GL_RGB8, GL_RGB8_SNORM, GL_SRGB8, GL_RGB8UI, GL_RGB8I
16-bit	GL_R16F, GL_RG8UI, GL_R16UI, GL_RG8I, GL_R16I, GL_RG8, GL_R16, GL_RG8_SNORM, GL_R16_SNORM
8-bit	GL_R8UI, GL_R8I, GL_R8, GL_R8_SNORM
RGTC1_RED	GL_COMPRESSED_RED_RGTC1, GL_COMPRESSED_SIGNED_RED_RGTC1
RGTC2_RG	GL_COMPRESSED_RG_RGTC2, GL_COMPRESSED_SIGNED_RG_RGTC2
BPTC_UNORM	GL_COMPRESSED_RGBA_BPTC_UNORM, GL_COMPRESSED_SRGB_ALPHA_BPTC_UNORM
BPTC_FLOAT	GL_COMPRESSED_RGB_BPTC_SIGNED_FLOAT, GL_COMPRESSED_RGB_BPTC_UNSIGNED_FLOAT

Table 5.15: Texture View Format Compatibility

In addition to formats that match each other's classes, you can always create a view of a texture with the same format as the original—even for formats that are not listed in Table 5.15.

Once you have created a view of a texture, you can use it like any other texture of the new type. For instance, if you have a 2D array texture, and you create a 2D non-array texture view of one of its layers, you can call **glTexSubImage2D()** to put data into the view, and the same data will end up in the corresponding layer of the array texture. As another example, you can create a 2D non-array texture view of a single layer of a

2D array texture and access it from a **sampler2D** uniform in a shader. Likewise, you can create a single-layer 2D array texture view of a 2D non-array texture and access that from a **sampler2DArray** uniform in a shader.

Summary

In this chapter you have learned about how OpenGL deals with the vast amounts of data required for graphics rendering. At the start of the pipeline, you saw how to automatically feed your vertex shaders with data using buffer objects. We also discussed methods of getting constant values, known as uniforms, into your shaders—first using buffers and then using the *default uniform block*. This block is also where the uniforms that represent textures, images, and storage buffers live; we used them to show you how to directly read and write images from and to textures and buffers using your shader code. You saw how to take a texture and pretend that part of it is actually a different type of texture, possibly with a different data format. You also learned about atomic operations, which touched on the massively parallel nature of modern graphics processors.