Textures

Textures are a structured form of storage that can be made accessible to shaders both for reading and for writing. They are most often used to store image data and come in many forms and arrangements. Perhaps the most common texture layout is two-dimensional, but textures can also be created in one-dimensional or three-dimensional layouts, array forms (with multiple textures stacked together to form one logical object), cubes, and so on. Textures are represented as objects that can be generated, bound to *texture units*, and manipulated. To use textures, we first need to ask OpenGL to create some for us by calling **glCreateTextures()**. At this point, the name we get back represents a just-created texture object ready to be filled with data and used by binding it to a context.

Creating and Initialzing Textures

The full creation of a texture involves specifying the type of texture you want to create, and then telling OpenGL the size of the image you want to store in it. Listing 5.35 shows how to create a new texture object with **glCreateTextures()**, use **glBindTexture()** to bind it to the `GL_TEXTURE_2D` target (which is one of several available texture targets), and then use the **glTexStorage2D()** function to allocate storage for the texture.

Click here to view code image

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

// Create a new 2D texture object
glCreateTextures(GL_TEXTURE_2D, 1, &texture);

// Specify the amount of storage we want to use for the texture
glTextureStorage2D(texture,         // Texture object
```

```
                      1,                // 1 mipmap level
                      GL_RGBA32F,       // 32-bit floating-point RGBA data
                      256, 256);        // 256 x 256 texels

    // Now bind it to the context using the GL_TEXTURE_2D binding point
    glBindTexture(GL_TEXTURE_2D, texture);
```

Listing 5.35: Generating, initializing, and binding a texture

Compare Listing 5.1 and Listing 5.35, noting how similar they are. In both cases, you create a new object, define the storage for the data it contains, and then bind it to a target ready for use by your OpenGL program. For textures, the function we've used to do this is **glTextureStorage2D()**. It takes as parameters the name of the texture we want to allocate storage for; the number of *levels* that are used in *mipmapping*, which we are not using here (but will explain shortly); the *internal format* of the texture (we chose `GL_RGBA32F` here, which is a four-channel floating-point format); and the width and height of the texture. When we call this function, OpenGL will allocate enough memory to store a texture with those dimensions for us. After defining storage for the texture, we can bind it to the context to use it. Next, we need to specify some data for the texture. To do this, we use **glTexSubImage2D()** as shown in Listing 5.36.

Click here to view code image

```
    // Define some data to upload into the texture
    float * data = new float[256 * 256 * 4];

    // generate_texture() is a function that fills memory with image data
    generate_texture(data, 256, 256);

    // Assume that "texture" is a 2D texture that we created earlier
    glTextureSubImage2D(texture,        // Texture object
                        0,              // Level 0
                        0, 0,           // Offset 0, 0
                        256, 256,       // 256 x 256 texels, replace entire
    image
                        GL_RGBA,        // Four-channel data
                        GL_FLOAT,       // Floating-point data
                        data);          // Pointer to data

    // Free the memory we allocated before - OpenGL now has our data
    delete [] data;
```

Listing 5.36: Updating texture data with **glTexSubImage2D()**

After the code in Listing 5.36 has run, OpenGL keeps a copy of the original texture data that we gave it and we're free to release the application's memory to the operating system.

If all you want to do is initialize a texture to a fixed value, then you can use **glClearTexSubImage()**, whose prototype is

Click here to view code image

```
void glClearTexSubImage(GLuint texture,
                        GLint level,
                        GLint xoffset,
                        GLint yoffset,
                        GLint zoffset,
                        GLsizei width,
                        GLsizei height,
                        GLsizei depth,
                        GLenum format,
                        GLenum type,
                        const void * data);
```

For **glClearTexSubImage()**, any texture type can be passed in `texture`—the dimensionality of the texture is deduced from the object you pass. `level` specifies the mipmap level you want to clear; `xoffset`, `yoffset`, and `zoffset` provide the starting offset of the region to be cleared; and `width`, `height`, and `depth` specify the dimensions of the region. The `format` and `type` parameters are interpreted exactly as they are for **glTexSubImage2D()**, but `data` is assumed to be a single texel's worth of data, which is then replicated across the whole texture. This command is not particularly useful if you're about to fill the texture with known data or bind the texture to a framebuffer to draw into it, but it can be very useful when you are going to write directly to the texture as an image variable in a shader, which will be discussed shortly.

Texture Targets and Types

The example in Listing 5.36 demonstrates how to create a 2D texture by binding a new name to the 2D texture target specified with `GL_TEXTURE_2D`. This is just one of several targets that are available to bind textures to, and a new texture object takes on the type determined by the target to which it is first bound. Thus, texture targets and types are often used interchangably. Table 5.6 lists the available targets and describes the type of texture that will be created when a new name is bound to that target.

Texture Target (GL_TEXTURE_*)	Description
1D	One-dimensional texture
2D	Two-dimensional texture
3D	Three-dimensional texture
RECTANGLE	Rectangle texture
1D_ARRAY	One-dimensional array texture
2D_ARRAY	Two-dimensional array texture
CUBE_MAP	Cube map texture
CUBE_MAP_ARRAY	Cube map array texture
BUFFER	Buffer texture
2D_MULTISAMPLE	Two-dimensional multisample texture
2D_MULTISAMPLE_ARRAY	Two-dimensional array multisample texture

Table 5.6: Texture Targets and Description

The `GL_TEXTURE_2D` texture target is probably the one you will deal with the most. This is our standard, two-dimensional image that you imagine would represent a picture. The `GL_TEXTURE_1D` and `GL_TEXTURE_3D` types allow you to create one-dimensional and three-dimensional textures, respectively. A 1D texture behaves just like a 2D texture with a height of 1, for the most part. A 3D texture, however, can be used to represent a *volume* and actually has a three-dimensional texture coordinate. Rectangle textures[3] are a special case of 2D textures that have subtle differences in how they are read in shaders and which parameters they support.

3. Rectangle textures were introduced into OpenGL when not all hardware could support textures whose dimensions were not integer powers of 2. Modern graphics hardware supports this almost universally, so rectangle textures have essentially become a subset of the 2D texture, and there isn't much need to use one in preference to a 2D texture.

The `GL_TEXTURE_1D_ARRAY` and `GL_TEXTURE_2D_ARRAY` types represent arrays of texture images aggregated into single objects. They are covered in more detail later in this chapter. Likewise, cube map textures (created by binding a texture name to the `GL_TEXTURE_CUBE_MAP` target) represent a collection of six square images that form a cube, which can be used to simulate lighting environments, for example. Just as the `GL_TEXTURE_1D_ARRAY` and `GL_TEXTURE_2D_ARRAY` targets represent 1D and 2D textures that are arrays of 1D or 2D images, so the `GL_TEXTURE_CUBE_MAP_ARRAY` target represents a texture that is an array of cube maps.

Buffer textures, represented by the `GL_TEXTURE_BUFFER` target, are a special type of texture that are much like a 1D texture, except that their storage is actually

represented by a buffer object. In addition, they differ from a 1D texture in that their maximum size can be much larger than a 1D texture.

The minimum requirement from the OpenGL specification is 65,536 texels, but in practice most implementations will allow you to create much larger buffers—usually in the range of several hundred megabytes. Buffer textures also lack a few of the features supported by the 1D texture type, such as filtering and mipmaps.

Finally, the multisample texture types `GL_TEXTURE_2D_MULTISAMPLE` and `GL_TEXTURE_2D_MULTISAMPLE_ARRAY` are used for *multisample antialiasing*, which is a technique for improving image quality, especially at the edges of lines and polygons.

Reading from Textures in Shaders

Once you've created a texture object and placed some data in it, you can read that data in your shaders and use it to color fragments, for example. Textures are represented in shaders as *sampler variables* and are hooked up to the outside world by declaring uniforms with sampler types. Just as there can be textures with various dimensionalities that can be created and used through the various texture targets, so there are corresponding sampler variable types that can be used in GLSL to represent them. The sampler type that represents two-dimensional textures is **sampler2D**. To access our texture in a shader, we can create a uniform variable with the **sampler2D** type, and then use the `texelFetch` built-in function with that uniform and a set of texture coordinates at which to read from the texture. Listing 5.37 shows an example of how to read from a texture in GLSL.

Click here to view code image

```
#version 450 core

uniform sampler2D s;

out vec4 color;

void main(void)
{
    color = texelFetch(s, ivec2(gl_FragCoord.xy), 0);
}
```

Listing 5.37: Reading from a texture in GLSL

The shader of Listing 5.37 simply reads from the uniform sampler `s` using a texture coordinate derived from the built-in variable `gl_FragCoord`. This variable is an input to the fragment shader that holds the floating-point coordinate of the fragment being processed in window coordinates. However, the `texelFetch` function accepts

integer-point coordinates that range from (0, 0) to the width and height of the texture. Therefore, we construct a two-component integer vector (**ivec2**) from the *x* and *y* components of `gl_FragCoord`. The third parameter to `texelFetch` is the mipmap level of the texture. Because the texture in this example has only one level, we set it to zero. The result of using this shader with our single-triangle example is shown in Figure 5.4.

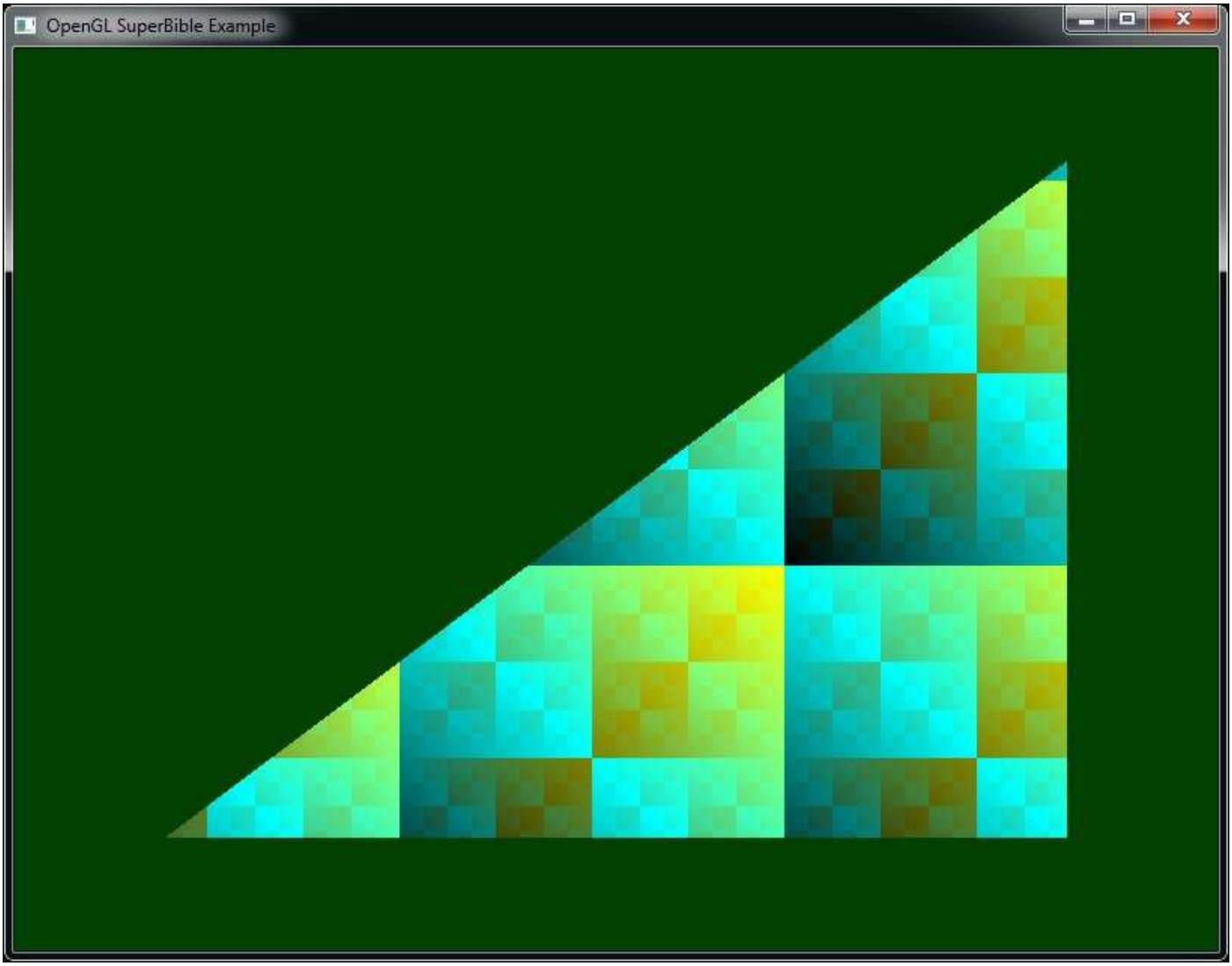

Figure 5.4: A simple textured triangle

Sampler Types

Each dimensionality of texture has a target to which texture objects are bound, as introduced in the previous section, and each target has a corresponding sampler type that is used in the shader to access them. Table 5.7 lists the basic texture types and the sampler that should be used in shaders to access them.

Texture Target	Sampler Type
GL_TEXTURE_1D	sampler1D
GL_TEXTURE_2D	sampler2D
GL_TEXTURE_3D	sampler3D
GL_TEXTURE_RECTANGLE	sampler2DRect
GL_TEXTURE_1D_ARRAY	sampler1DArray
GL_TEXTURE_2D_ARRAY	sampler2DArray
GL_TEXTURE_CUBE_MAP	samplerCube
GL_TEXTURE_CUBE_MAP_ARRAY	samplerCubeArray
GL_TEXTURE_BUFFER	samplerBuffer
GL_TEXTURE_2D_MULTISAMPLE	sampler2DMS
GL_TEXTURE_2D_MULTISAMPLE_ARRAY	sampler2DMSArray

Table 5.7: Basic Texture Targets and Sampler Types

As shown in Table 5.7, to create a 1D texture and then use it in your shader, you would bind a new texture name to the `GL_TEXTURE_1D` target and then use a **sampler1D** variable in your shader to read from it. Likewise, for 2D textures, you'd use `GL_TEXTURE_2D` and **sampler2D**; for 3D textures, you'd use `GL_TEXTURE_3D` and **sampler3D**; and so on.

The GLSL sampler types **sampler1D**, **sampler2D**, and so on represent floating-point data. It is also possible to store signed and unsigned integer data in textures and retrieve it in your shader. To represent a texture containing signed integer data, we prefix the equivalent floating-point sampler type with the letter `i`. Similarly, to represent a texture containing unsigned integer data, we prefix the equivalent floating-point sampler type with the letter `u`. For example, a 2D texture containing signed integer data would be represented by a variable of type **isampler2D**, and a 2D texture containing unsigned integer data would be represented by a variable of type **usampler2D**.

As shown in our introductory example of Listing 5.37, we read from textures in shaders using the `texelFetch` built-in function. There are actually many variations of this function, because it is *overloaded*. This means that there are several versions of the function, each of which has a different set of function parameters. Each function takes a sampler variable as the first parameter, with the main differentiator between the functions being the type of that sampler. The remaining parameters to the function depend on the type of sampler being used. In particular, the number of components in the texture coordinate depends on the dimensionality of the sampler, and the return type of the function depends on the type of the sampler (floating point, signed, or unsigned

integer). For example, the following are all declarations of the `texelFetch` function:

Click here to view code image

```
vec4 texelFetch(sampler1D s, int P, int lod);
vec4 texelFetch(sampler2D s, ivec2 P, int lod);
ivec4 texelFetch(isampler2D s, ivec2 P, int lod);
uvec4 texelFetch(usampler3D s, ivec3 P, int lod);
```

Notice how the version of `texelFetch` that takes a **sampler1D** sampler type expects a one-dimensional texture coordinate, **int** `P`, but the version that takes a **sampler2D** expects a two-dimensional coordinate, **ivec2** `P`. You can also see that the return type of the `texelFetch` function is influenced by the type of sampler that it takes. The version of `texelFetch` that takes a **sampler2D** produces a floating-point vector, whereas the version that takes an **isampler2D** sampler returns an integer vector. This type of overloading is similar to that supported by languages such as C++. That is, functions can be overloaded by parameter types, but not by return type, unless that return type is determined by one of the parameters.

All of the `texture` functions return a four-component vector, regardless of whether that vector is floating-point or integer, and independently from the format of the texture object bound to the texture unit referenced by the sampler variable. If you read from a texture that contains fewer than four channels, the default value of 0 will be filled in for the green and blue channels and 1 will be filled in for the alpha channel. If one or more channels of the returned data never get used by your shader, that's fine: It's likely that the shader compiler will optimize away any code that becomes redundant as a result.

You can get more information about the samplers in your shader by using a few other functions provided by GLSL. First, the `textureSize` function returns the size of the texture. It, too, is overloaded and returns a different-dimension result that depends on the argument you specify. A few examples follow:

Click here to view code image

```
int textureSize(sampler1D sampler, int lod);
ivec2 textureSize(sampler2D sampler, int lod);
ivec3 textureSize(gsampler3D sampler, int lod);
```

Like the `texelFetch` function, `textureSize` takes an `lod` parameter, which specifies which of the texture's mipmap levels you'd like to know the size of. If you're working with multisample textures (a special kind of texture that has multiple colors, or *samples* for each texel), you can find out how many samples it contains by calling

Click here to view code image

```
int textureSamples(sampler2DMS sampler);
```

The `textureSamples` function returns the number of samples in the texture as a

single integer. Again, several overloaded versions of this function exist for the various sampler types, but they all return a single integer.

Loading Textures from Files

In our simple example, we generated the texture data directly in our application. However, this clearly isn't practical in a real-world application where you most likely have images stored on disk or on the other end of a network connection. Your options are either to convert your textures into hard-coded arrays (yes, there are utilities that will do this for you) or to load them from files within your application.

There are lots of image file formats that store pictures with or without compression, some of which are more suited to photographs and some of which are more suited to line drawings or text. However, very few image formats exist that can properly store all of the formats supported by OpenGL or represent advanced features such as mipmaps, cube maps, and so on. One such format is `.KTX`, or the *Khronos TeXture format*, which was specifically designed for the storage of pretty much anything that can be represented as an OpenGL texture. In fact, the `.KTX` file format includes most of the parameters you need to pass to texturing functions such as **glTextureStorage2D()** and **glTextureSubImage2D()** to load the texture directly in the file.

The structure of a `.KTX` file header is shown in Listing 5.38.

Click here to view code image

```
struct header
{
    unsigned char       identifier[12];
    unsigned int        endianness;
    unsigned int        gltype;
    unsigned int        gltypesize;
    unsigned int        glformat;
    unsigned int        glinternalformat;
    unsigned int        glbaseinternalformat;
    unsigned int        pixelwidth;
    unsigned int        pixelheight;
    unsigned int        pixeldepth;
    unsigned int        arrayelements;
    unsigned int        faces;
    unsigned int        miplevels;
    unsigned int        keypairbytes;
};
```

Listing 5.38: The header of a `.KTX` file

In this header, `identifier` contains a series of bytes that allow the application to

verify that this is a legal `.KTX` file and `endianness` contains a known value that will be different depending on whether a little-endian or big-endian machine created the file. The `gltype`, `glformat`, `glinternalformat`, and `glbaseinternalformat` fields are actually the raw values of the `GLenum` types that will be used to load the texture. The `gltypesize` field stores the size, in bytes, of one element of data in the `gltype` type, and is used in case the endianness of the file does not match the native endianness of the machine loading the file, in which case each element of the texture must be byte swapped as it is loaded. The remaining fields —`pixelwidth`, `pixelheight`, `pixeldepth`, `arrayelements`, `faces`, and `miplevels`—store information about the dimensions of the texture. Finally, the `keypairbytes` field is used to allow applications to store additional information after the header and before the texture data. After this information, the raw texture data begins.

Because the `.KTX` file format was designed specifically for use in OpenGL-based applications, writing the code to load a `.KTX` file is actually pretty straightforward. Even so, a basic loader for `.KTX` files is included in this book's source code. To use the loader, you can simply reserve a new name for a texture using **glGenTextures()**, and then pass it, along with the name of the `.KTX` file, to the loader. If you wish, you can even omit the OpenGL name for the texture (or pass 0) and the loader will call **glCreateTextures()** for you. If the `.KTX` file is recognized, the loader will bind the texture to the appropriate target and load it with the data from the `.KTX` file. An example is shown in Listing 5.39.

Click here to view code image

```
// Generate a name for the texture
glGenTextures(1, &texture);

// Load texture from file
sb7::ktx::file::load("media/textures/icemoon.ktx", texture);
```

Listing 5.39: Loading a `.KTX` file

If you think that Listing 5.39 looks simple, you're right! The `.KTX` loader takes care of almost all the details for you. If the loader is successful in loading and allocating the texture, it will return the name of the texture you passed in (or the one it generated for you); if it fails for some reason, it will return 0. If you want to use the texture, you'll need to bind it to one of the texture units. Don't forget to delete the texture when you're done with it by calling **glDeleteTextures()** on the name returned by the `.KTX` loader. Applying the texture loaded in Listing 5.39 to the whole viewport produces the image shown in Figure 5.5.

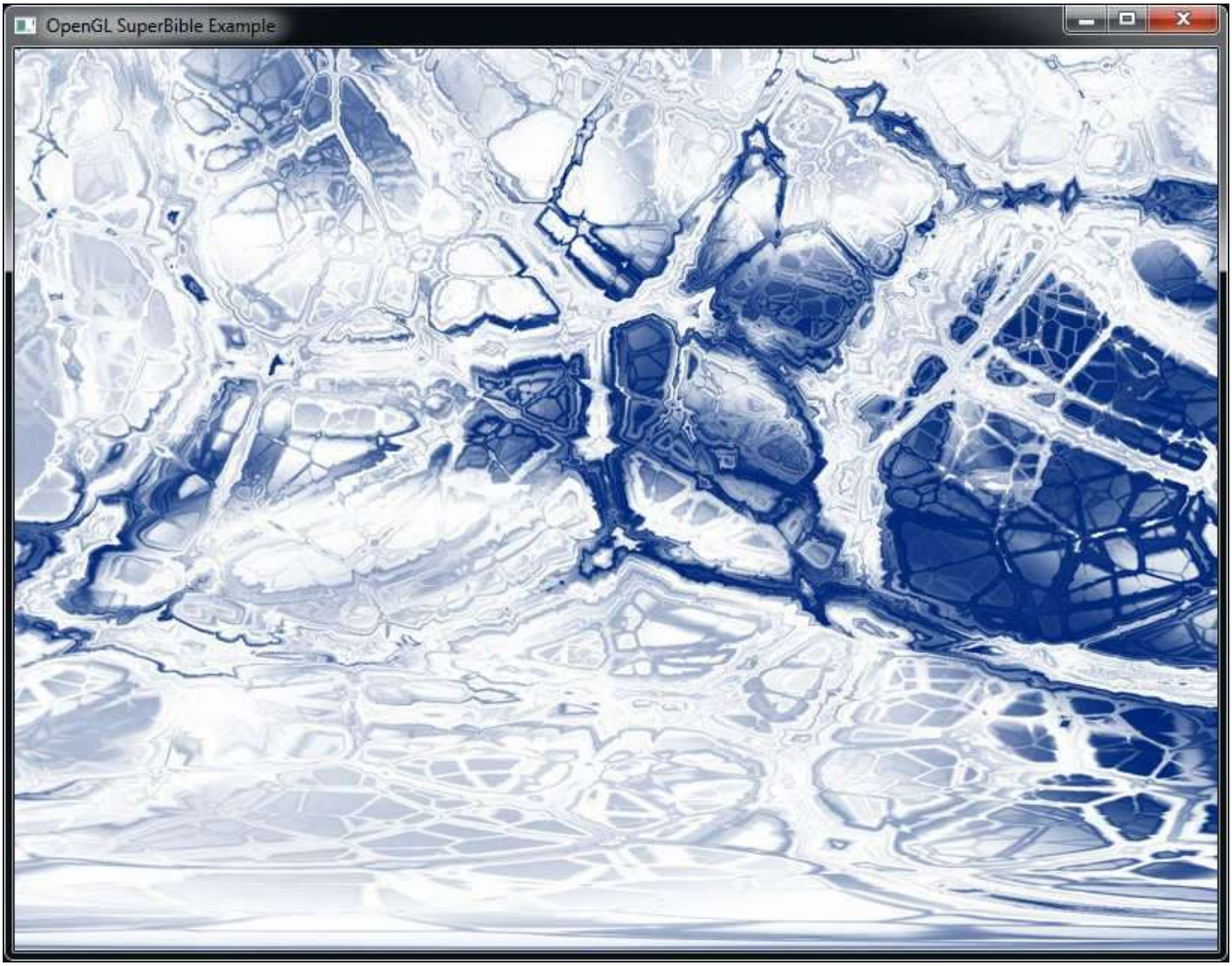

Figure 5.5: A full-screen texture loaded from a `.KTX` file

Texture Coordinates

In the simple example shown earlier in this chapter, we used the current fragment's window-space coordinate as the position at which to read from the texture. You can actually use any values you want, though in a fragment shader, they will usually be derived from one of the inputs that are smoothly interpolated from across each primitive by OpenGL. It is then the vertex (or geometry or tessellation evaluation) shader's responsibility to produce the values of these coordinates. The vertex shader will generally pull the texture coordinates from a per-vertex input and pass them through unmodified. When you use multiple textures in your fragment shader, there is nothing to stop you from using a unique set of texture coordinates for each texture, but for many applications a single set of texture coordinates will be used for every texture.

A simple vertex shader that accepts a single texture coordinate and passes it through to the fragment shader is shown in Listing 5.40; the corresponding fragment shader is shown in Listing 5.41.

Click here to view code image

```
#version 450 core

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

layout (location = 0) in vec4 position;
layout (location = 4) in vec2 tc;

out VS_OUT
{
    vec2 tc;
} vs_out;

void main(void)
{
    // Calculate the position of each vertex
    vec4 pos_vs = mv_matrix * position;

    // Pass the texture coordinate through unmodified
    vs_out.tc = tc;

    gl_Position = proj_matrix * pos_vs;
}
```

Listing 5.40: Vertex shader with a single texture coordinate

The shader shown in Listing 5.41 not only takes as input the texture coordinate produced by the vertex shader, but also scales it non-uniformly. The texture's wrapping mode is set to `GL_REPEAT`, which means that the texture will be repeated several times across the object.

Click here to view code image

```
#version 450 core

layout (binding = 0) uniform sampler2D tex_object;

// Input from vertex shader
in VS_OUT
{

   vec2 tc;
} fs_in;

// Output to framebuffer
out vec4 color;

void main(void)
{
```

```
    // Simply read from the texture at the (scaled) coordinates and
    // assign the result to the shader's output.
    color = texture(tex_object, fs_in.tc * vec2(3.0, 1.0));
}
```

Listing 5.41: Fragment shader with a single texture coordinate

By passing a texture coordinate with each vertex, we can *wrap* a texture around an object. Texture coordinates can then be generated offline procedurally or assigned by hand by an artist using a modeling program and stored in an object file. If we load a simple checkerboard pattern into a texture and apply it to an object, we can see how the texture is wrapped around it. Such an example is shown in Figure 5.6. On the left is the object with a checkerboard pattern wrapped around it. On the right is the same object using a texture loaded from a file.

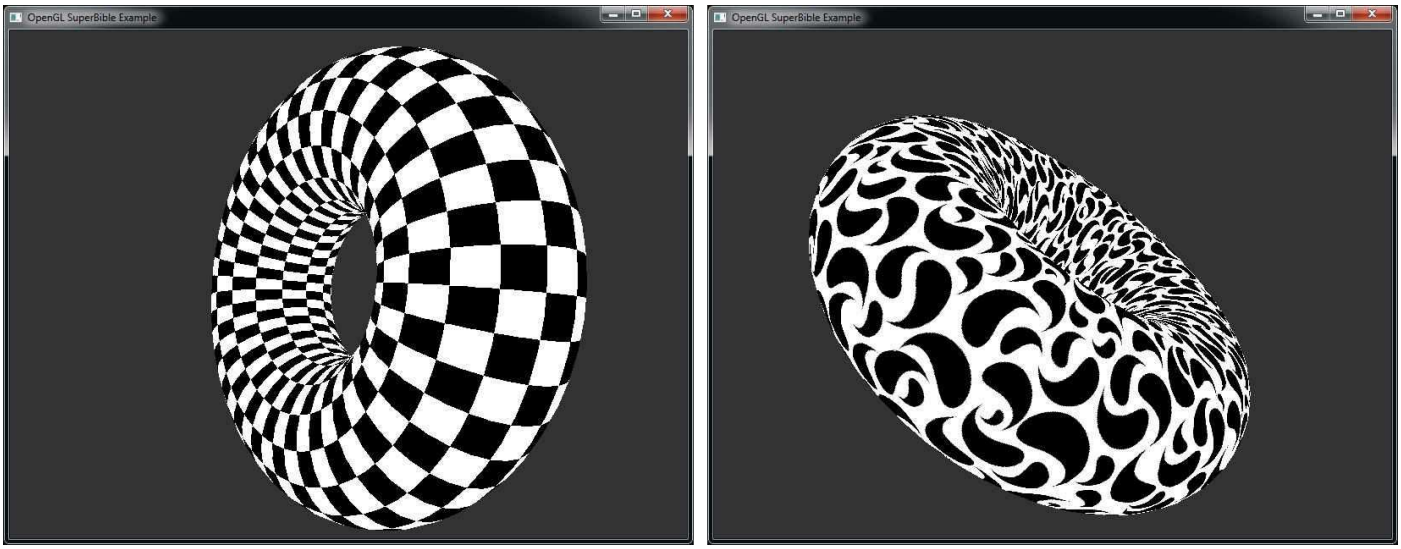

Figure 5.6: An object wrapped in simple textures

Controlling How Texture Data Is Read

OpenGL provides a lot of flexibility in how it reads data from textures and returns it to your shader. Usually, texture coordinates are normalized—that is, they range between 0.0 and 1.0. OpenGL lets you control what happens when the texture coordinates you supply fall outside this range. This is called the *wrapping mode* of the sampler. Also, you get to decide how values *between* the real samples are calculated. This is called the *filtering mode* of a sampler. The parameters controlling the wrapping and filtering modes of a sampler are stored in a *sampler object*.

To create one or more sampler objects, call

Click here to view code image

```
void glCreateSamplers(GLsizei n, GLuint * samplers);
```

Here, `n` is the number of sampler objects you want to create and `samplers` is the

address of at least `n` unsigned integer variables that will be used to store the names of the newly created sampler objects.

Sampler objects are manipulated in the same way as other objects in OpenGL. The two main functions you will use to set the parameters of a sampler object are

Click here to view code image

```
void glSamplerParameteri(GLuint sampler,
                         GLenum pname,
                         GLint param);
```

and

Click here to view code image

```
void glSamplerParameterf(GLuint sampler,
                         GLenum pname,
                         GLfloat param);
```

Notice that **glSamplerParameteri()** and **glSamplerParameterf()** both take the sampler object name as the first parameter. You will need to bind a sampler object to use it, but in this case you bind it to a texture unit just as you would a texture. The function used to bind a sampler object to one of the texture units is **glBindSampler()**, whose prototype is

Click here to view code image

```
void glBindSampler(GLuint unit, GLuint sampler);
```

Rather than taking a texture target, **glBindSampler()** takes the index of the texture unit to which it should bind the sampler object. Together, the sampler object and the texture object bound to a given texture unit form a complete set of data and parameters required for constructing texels as demanded by your shaders. By separating the parameters of the texture sampler from the texture data, three important behaviors become possible:

- You can use the same set of sampling parameters for a large number of textures without specifying those parameters for each of the textures.
- You can change the texture bound to a texture unit without updating the sampler parameters.
- You can read from the same texture with multiple sets of sampler parameters at the same time.

Although nontrivial applications will likely opt to use their own sampler objects, each texture effectively contains an embedded sampler object that includes the sampling parameters to be used for that texture when no sampler object is bound to the corresponding texture unit. You can think of this as the default sampling parameters for a

texture. To access the sampler object stored inside the texture object, call

Click here to view code image

```
void glTextureParameterf(GLuint texture,
                         GLenum pname,
                         GLfloat param);
```

or

Click here to view code image

```
void glTextureParameteri(GLuint texture,
                         GLenum pname,
                         GLint param);
```

In these cases, the `texture` parameter specifies the texture whose embedded sampler object you want to access and `pname` and `parameter` have the same meanings as for **glSamplerParameteri()** and **glSamplerParameterf()**.

Using Multiple Textures

If you want to use multiple textures in a single shader, you'll need to create multiple sampler uniforms and set them up to reference different texture units. You'll also need to bind multiple textures to your context at the same time. To allow this, OpenGL supports multiple texture units. The number of units supported can be queried by calling **glGetIntegerv()** with the `GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS` parameter, as in

Click here to view code image

```
GLint units;
glGetIntegerv(GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS, &units);
```

This will tell you the maximum number of texture units that might be accessible to all shader stages at any one time. To bind a texture to a specific texture unit, rather than calling **glBindTexture()** as you have been doing so far, you need to instead call **glBindTextureUnit()**, whose prototype is

Click here to view code image

```
void glBindTextureUnit(GLuint unit,
                       GLuint texture);
```

Here, `unit` is the zero-based index of the unit to which you want to bind the texture and `texture` is the name of the texture object you're going to bind. As an example, we can bind multiple textures by performing the following:

Click here to view code image

```
GLuint textures[3];
```

```
// Create three 2D textures
glCreateTextures(3, GL_TEXTURE_2D, &textures);

// Bind the three textures to the first three texture units
glBindTextureUnit(0, textures[0]);
glBindTextureUnit(1, textures[1]);
glBindTextureUnit(2, textures[2]);
```

Once you have bound multiple textures to your context, you need to make the sampler uniforms in your shaders refer to the different units. Samplers (which represent a texture and a set of sampling parameters) are represented by uniform variables in your shaders. If you don't initialize them, they will, by default, refer to unit 0. This might be fine for a simple application that uses a single texture (you'll notice that we've settled for that default in our examples so far), but the uniforms will need to be initialized to refer to the correct texture units in more complex applications. To do this, you can initialize its value at shader compilation time by using the **binding** layout qualifier in your shader code. To create three sampler uniforms referring to texture units 0, 1, and 2, we can write

Click here to view code image

```
layout (binding = 0) uniform sampler2D foo;
layout (binding = 1) uniform sampler2D bar;
layout (binding = 2) uniform sampler2D baz;
```

After compiling this code and linking it into a program object, the sampler `foo` will reference texture unit 0, `bar` will reference unit 1, and `baz` will reference unit 2. Setting the unit to which a sampler refers directly in the shader code is convenient and explicit and does not require changes to the application's source code. This is the method we will use in the majority of the samples in the remainder of the book.

Texture Filtering

There is almost never a one-to-one correspondence between texels in the texture map and pixels on the screen. A careful programmer could achieve this result, but only by texturing geometry[4] that was carefully planned to appear on screen such that the texels and pixels lined up. (This is actually often done when OpenGL is used for image processing applications.) Consequently, texture images are always either stretched or shrunk as they are applied to geometric surfaces. Due to the orientation of the geometry, a given texture could even be stretched in one dimension and shrunk in another dimension at the same time across the surface of some object.

4. Such geometry is common in 2D graphics rendering such as user-interface elements, text, and so on.

In the samples presented so far, we have been using the `texelFetch()` function, which fetches a single texel from the selected texture at specific integer texture

coordinates. Clearly, if we want to achieve a fragment-to-texel ratio that is not an integer, this function isn't going to cut it. Here, we need a more flexible function, and that function is simply called `texture()`. Like `texelFetch()`, it has several overloaded prototypes:

Click here to view code image

```
vec4 texture(sampler1D s, float P);
vec4 texture(sampler2D s, vec2 P);
ivec4 texture(isampler2D s, vec2 P);
uvec4 texture(usampler3D s, vec3 P);
```

As you might have noticed, unlike the `texelFetch()` function, the `texture()` function accepts *floating-point* texture coordinates. The range 0.0 to 1.0 in each dimension maps exactly once onto the texture. Obviously, not every possible value between 0.0 and 1.0 is going to map directly to a single texture—sometimes the coordinate will fall *between* texels. Further, the texture coordinates can even stray far outside the range 0.0 to 1.0. The next few sections describe how OpenGL takes these floating-point numbers and uses them to produce texel values for your shaders.

The process of calculating color fragments from a stretched or shrunken texture map is called *texture filtering*. Stretching a texture is also known as *magnification* and shrinking a texture is also known as *minification*. Using the sampler parameter functions, OpenGL allows you to set the method of constructing the values of texels under both magnification and minification. These conditions are known as *filters*. The parameter names for these two filters are `GL_TEXTURE_MAG_FILTER` and `GL_TEXTURE_MIN_FILTER` for the magnification and minification filters, respectively. For now, you can select from two basic texture filters for them, `GL_NEAREST` and `GL_LINEAR`, which correspond to nearest neighbor and linear filtering, respectively. Make sure you always choose one of these two filters for the `GL_TEXTURE_MIN_FILTER`—the default filter setting does not work without mipmaps (see the "Mipmaps" section, up next).

Nearest neighbor filtering is the simplest and fastest filtering method you can choose. With this approach, texture coordinates are evaluated and plotted against a texture's texels, and whichever texel the coordinate falls in, that color is used for the fragment texture color. In signal processing terms, this is known as *point sampling*. Nearest neighbor filtering is characterized by large blocky pixels when the texture is stretched over an especially large area. An example is shown on the left of Figure 5.7. You can set the texture filter for both the minification and the magnification filters by using these two function calls:

Click here to view code image

```
glSamplerParameteri(sampler, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glSamplerParameteri(sampler, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

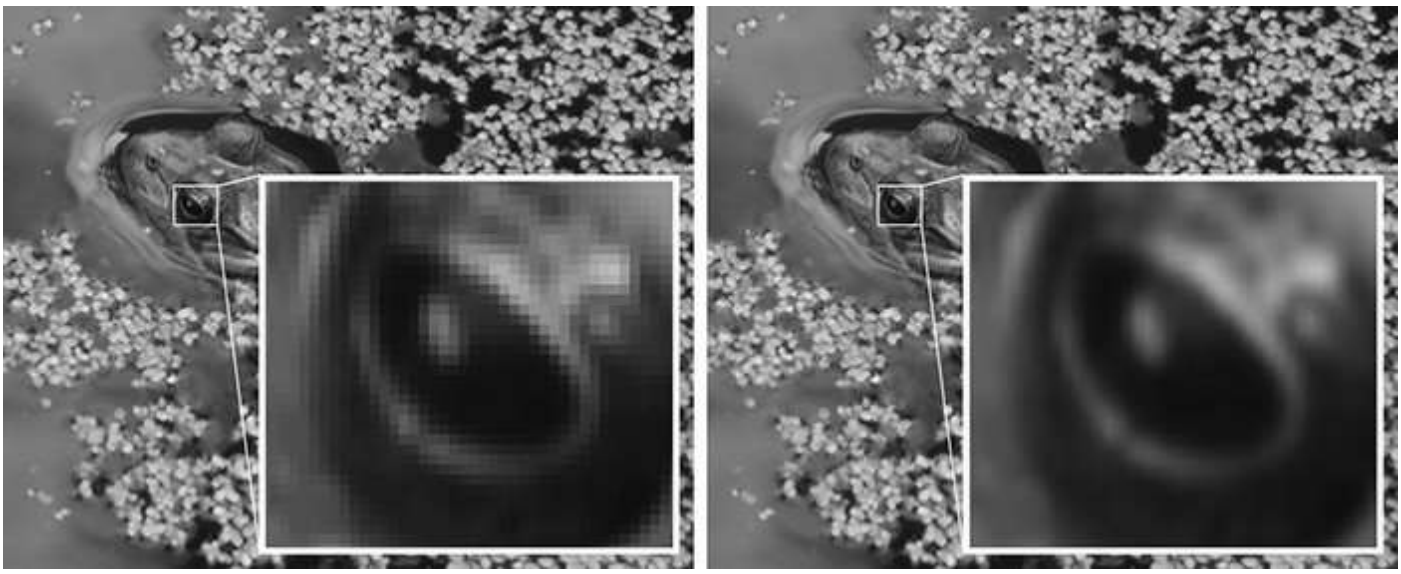

Figure 5.7: Texture filtering—nearest neighbor (left) and linear (right)

Linear filtering requires more work than nearest neighbor filtering but often is worth the extra overhead. On today's commodity hardware, the extra cost of linear filtering is approximately zero. Linear filtering works not by taking the nearest texel to the texture coordinate, but rather by applying the weighted average of the texels surrounding the texture coordinate (a linear interpolation). For this interpolated fragment to match the texel color exactly, the texture coordinate needs to fall directly in the center of the texel. Linear filtering is characterized by "fuzzy" graphics when a texture is stretched. This fuzziness, however, often lends a more realistic and less artificial look than the jagged blocks of the nearest neighbor filtering mode. A contrasting example is shown on the right of Figure 5.7. You can set linear filtering simply enough by using the following lines:

Click here to view code image

```
glSamplerParameteri(sampler, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glSamplerParameteri(sampler, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

As you can see in Figure 5.7, the image on the left (which has nearest neighbor filtering applied) is blocky and jagged, especially around areas of high contrast. The image on the right of Figure 5.7, however, is smooth (if a little blurry).

Mipmaps

Mipmapping is a powerful texturing technique that can improve both the rendering performance and the visual quality of a scene. It does this by addressing two common problems with standard texture mapping.

The first is an effect called scintillation (aliasing artifacts) that appears on the surface of objects rendered very small on screen compared to the relative size of the texture applied. Scintillation can be seen as a sort of sparkling that occurs as the sampling area

on a texture map moves disproportionately to its size on the screen. The negative effects of scintillation are most noticeable when the camera or the objects are in motion.

The second issue is more performance related but is due to the same scenario that leads to scintillation. That is, a large amount of texture memory is used to store the texture, but it is accessed very sparsely as adjacent fragments on the screen access texels that are disconnected in texture space. This causes texturing performance to suffer greatly as the size of the texture increases and the sparsity of access becomes greater.

The solution to both of these problems is to simply use a smaller texture map. However, this solution then creates a new problem: When near the same object, it must be rendered larger, and a small texture map will then be stretched to the point of creating a hopelessly blurry or blocky textured object. The solution to both of these issues is mipmapping. Mipmapping gets its name from the Latin phrase *multum in parvo,* which means "many things in a small place." In essence, you load not just a single image into the texture object, but a whole series of images from largest to smallest into a single "mipmapped" texture. OpenGL then uses a new set of filter modes to choose the best-fitting texture or textures for the given geometry. At the cost of some extra memory (and possibly considerably more processing work), you can eliminate scintillation and the texture memory processing overhead for distant objects simultaneously, while maintaining higher-resolution versions of the texture available when needed.

A mipmapped texture consists of a series of texture images, each one-half the size on each axis or one-fourth the total number of pixels of the previous image. This scenario is shown in Figure 5.8. Mipmap levels do not have to be square, but the halving of the dimensions continues until the last image is 1 × 1 texel. When one of the dimensions reaches 1, further divisions occur on the other dimension only. For 2D textures, using a square set of mipmaps requires about one-third more memory than not using mipmaps at all.

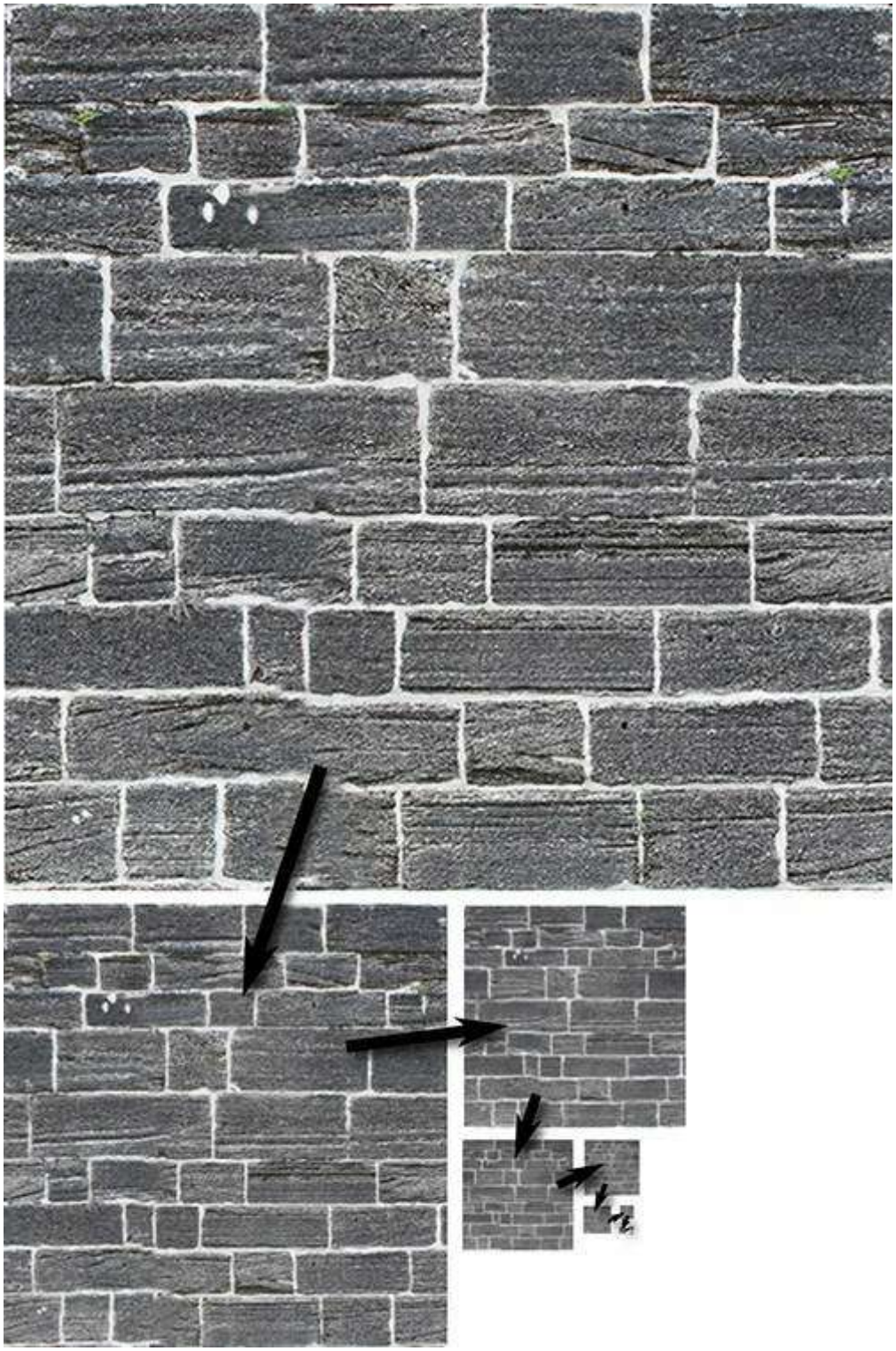

Figure 5.8: A series of mipmapped images

Mipmap levels are loaded with **glTextureSubImage2D()** (for 2D textures). Now the `level` parameter comes into play, because it specifies which mip level the image data is for. The first level is 0, then 1, 2, and so on. If mipmapping is not being used, you would typically use only level 0. When you allocate your texture with **glTextureStorage2D()** (or the appropriate function for the type of texture you're

allocating), you can set the number of levels to include in the texture in the `levels` parameter. Then, you can use mipmapping with those levels present in the texture. You can further constrain the number of mipmap levels that will be used during rendering by setting the base and maximum levels to be used with the `GL_TEXTURE_BASE_LEVEL` and `GL_TEXTURE_MAX_LEVEL` texture parameters. For example, if you want to specify that only mip levels 0 through 4 should be accessed, you call glTextureParameteri twice as shown here:

Click here to view code image

```
glTextureParameteri(texture, GL_TEXTURE_BASE_LEVEL, 0);
glTextureParameteri(texture, GL_TEXTURE_MAX_LEVEL, 4);
```

Mipmap Filtering

Mipmapping adds a new twist to the two basic texture filtering modes `GL_NEAREST` and `GL_LINEAR` by giving four permutations for mipmapped filtering modes. They are listed in Table 5.8.

Constant	Description
GL_NEAREST	Perform nearest neighbor filtering using only the base mip level.
GL_LINEAR	Perform linear filtering using only the base mip level.
GL_NEAREST_MIPMAP_NEAREST	Select the nearest mip level and perform nearest neighbor filtering.
GL_NEAREST_MIPMAP_LINEAR	Perform a linear interpolation between mip levels and perform nearest neighbor filtering within each.
GL_LINEAR_MIPMAP_NEAREST	Select the nearest mip level and perform linear filtering within it.
GL_LINEAR_MIPMAP_LINEAR	Perform a linear interpolation between mip levels and perform linear filtering; also called *trilinear* filtering.

Table 5.8: Texture Filters, Including Mipmapped Filters

Just loading the mip levels with **glTextureSubImage2D()** does not by itself enable mipmapping. If the minification filter mode in the texture or associated sampler object is set to `GL_LINEAR` or `GL_NEAREST`, only the base texture level is used, and any mip levels loaded are ignored. You must specify one of the mipmapped filters listed

for the loaded mip levels to be used. The constants have the form GL_<FILTER>_MIPMAP_<SELECTOR>, where <FILTER> specifies the texture filter to be used on the mip level selected, and <SELECTOR> specifies how the mip level is selected. For example, NEAREST selects the nearest matching mip level. Using LINEAR for the selector creates a linear interpolation between the two nearest mip levels, which is again filtered by the chosen texture filter.

Which filter you select varies depending on the application and the performance requirements at hand. GL_NEAREST_MIPMAP_NEAREST, for example, gives very good performance and low aliasing (scintillation) artifacts, but nearest neighbor filtering is often not visually pleasing. GL_LINEAR_MIPMAP_NEAREST is often used to speed up applications because a higher-quality linear filter is used, but a fast selection (nearest) is made between the different-sized mip levels available. Note that you can use the GL_<*>_MIPMAP_<*> filter modes only for the GL_TEXTURE_MIN_FILTER setting—the GL_TEXTURE_MAG_FILTER setting must always be either GL_NEAREST or GL_NEAREST.

Using NEAREST as the mipmap selector (as in both examples in the preceding paragraph), however, can also leave an undesirable visual artifact. For oblique views, you can often see the transition from one mip level to another across a surface. It can be seen as a distortion line or a sharp transition from one level of detail to another. The GL_LINEAR_ MIPMAP_LINEAR and GL_NEAREST_MIPMAP_LINEAR filters perform an additional interpolation between mip levels to eliminate this transition zone, but at the extra cost of substantially more processing overhead. The GL_LINEAR_MIPMAP_LINEAR filter is often referred to as trilinear mipmapping and, although there are more advanced techniques for image filtering, produces very good results. In practice, there is very little performance difference between GL_LINEAR_MIPMAP_NEAREST and GL_LINEAR_MIPMAP_LINEAR; thus, given that the latter provides superior visual quality, it's a good idea to just use that mode unless you have a particular reason not to.

Generating Mip Levels

As mentioned previously, mipmapping for 2D textures requires approximately one-third more texture memory than just loading the base texture image. It also requires that all the smaller versions of the base texture image be available for loading. Sometimes this can be inconvenient because the lower-resolution images may not necessarily be available to either the programmer or the end user of your software. While having precomputed mip levels for your textures yields the very best results, it is convenient and somewhat common to have OpenGL generate the textures for you. You can generate all the mip levels for a texture once you've loaded level 0 with the function **glGenerateTextureMipmap()**.

Click here to view code image

```
void glGenerateTextureMipmap(GLenum target);
```

The `texture` parameter can refer to a texture of type `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, `GL_TEXTURE_CUBE_MAP`, `GL_TEXTURE_1D_ARRAY`, or `GL_TEXTURE_2D_ARRAY` (more on these last three later). The quality of the filter used to create the smaller textures may vary widely from implementation to implementation. In addition, generating mipmaps on the fly may not be any faster than actually loading prebuilt mipmaps. This is something to think about in performance-critical applications. For the very best visual quality (as well as for consistency), you should load your own pregenerated mipmaps. This book's `.KTX` file loader supports loading mipmaps from `.KTX` files on disk.

Mipmaps in Action

The example program `tunnel` shows off mipmapping as described in this chapter and demonstrates visually the different filtering and mipmap modes. This sample program loads three textures at start-up and then switches between them to render a tunnel. The prefiltered images that make up the textures are stored in the `.KTX` files containing the texture data. The tunnel has a brick wall pattern with different materials on the floor and ceiling. The output from `tunnel` is shown in Figure 5.9 with the texture minification mode set to `GL_LINEAR_MIPMAP_LINEAR`. As you can see, the texture becomes blurrier as you get farther down the tunnel.

Figure 5.9: A tunnel rendered with three textures and mipmapping

Texture Wrap

Normally, you specify texture coordinates between 0.0 and 1.0 to map out the texels in a texture map. If texture coordinates fall outside this range, OpenGL handles them according to the current texture wrapping mode specified in the sampler object. You can set the wrap mode for each component of the texture coordinate individually by calling **glSamplerParameteri()** with `GL_TEXTURE_WRAP_S`, `GL_TEXTURE_WRAP_T`, or `GL_TEXTURE_WRAP_R` as the parameter name. The wrap mode can then be set to one of the following values: `GL_REPEAT`, `GL_MIRRORED_REPEAT`, `GL_CLAMP_TO_EDGE`, or `GL_CLAMP_TO_BORDER`. The value of `GL_TEXTURE_WRAP_S` affects 1D, 2D, and 3D textures; `GL_TEXTURE_WRAP_T` affects only 2D and 3D textures; and `GL_TEXTURE_WRAP_R` affects only 3D textures.

The `GL_REPEAT` wrap mode simply causes the texture to repeat in the direction in which the texture coordinate has exceeded 1.0. The texture repeats again for every

integer texture coordinate. This mode is useful for applying a small tiled texture to large geometric surfaces. Well-done seamless textures can lend the appearance of a seemingly much larger texture, but at the cost of a much smaller texture image. The GL_MIRRORED_REPEAT mode is similar, but as each component of the texture passes 1.0, it starts moving back toward the origin of the texture until it reaches 2.0, at which point the pattern repeats. it is The other modes do not repeat, but are "clamped"—thus their name.

If the only implication of the wrap mode is whether the texture repeats, you would need just two wrap modes: repeat and clamp. However, the texture wrap mode also has a great deal of influence on how texture filtering is done at the edges of the texture maps. For GL_NEAREST filtering, there are no consequences to the wrap mode because the texture coordinates are always snapped to some particular texel within the texture map. However, the GL_LINEAR filter takes an average of the pixels surrounding the evaluated texture coordinate, which creates a problem for texels that lie along the edges of the texture map. This problem is resolved quite neatly when the wrap mode is GL_REPEAT. The texel samples are simply taken from the next row or column, which in repeat mode wraps back around to the other side of the texture. This mode works perfectly for textures that wrap around an object and meet on the other side (such as spheres).

The clamped texture wrap mode offers a couple of options for the way texture edges are handled. For GL_CLAMP_TO_BORDER, the needed texels are taken from the texture border color (which can be set by passing GL_TEXTURE_BORDER_COLOR to **glSamplerParameterfv()**). The GL_CLAMP_TO_EDGE wrap mode forces texture coordinates out of range to be sampled along the last row or column of valid texels.

Figure 5.10 shows a simple example of the various texture wrapping modes. The same mode is used for both the S and T components of the texture coordinates. The four squares in the image have the same texture applied to them, but with different texture wrapping modes used. The texture is a simple square with nine arrows pointing up and to the right, with a bright band around the top and right edges. For the top left square, the GL_CLAMP_TO_BORDER mode is used. The border color has been set to a dark color; it is clear that when OpenGL ran out of texture data, it used the dark color instead. However, in the lower-left square, the GL_CLAMP_TO_EDGE mode is used. In this case, the bright band is continued to the top and right of the texture data.

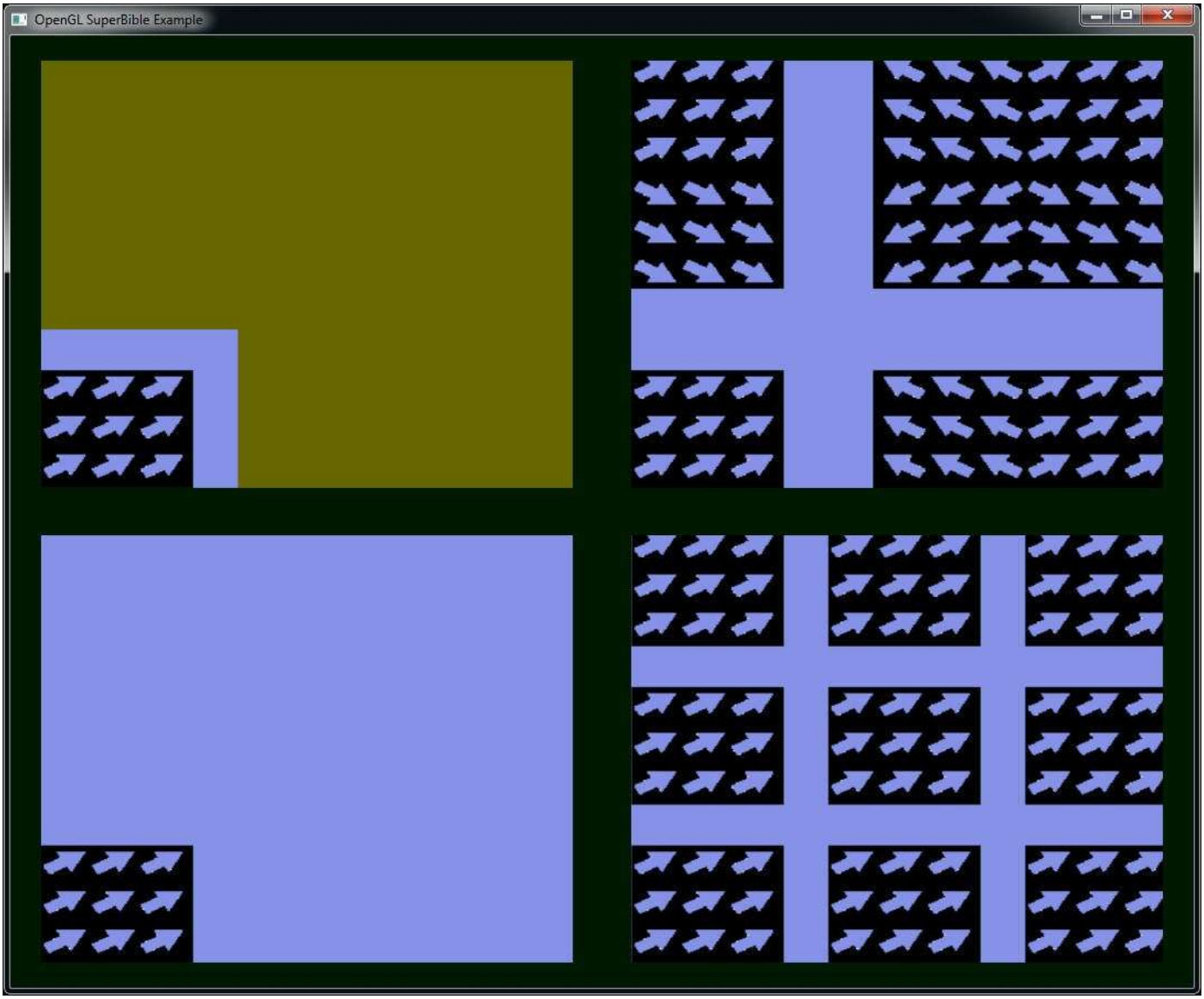

Figure 5.10: Example of texture coordinate wrapping modes

The lower-right square is drawn using the `GL_REPEAT` mode, which wraps the texture over and over. As you can see, there are several copies of our arrow texture, and all the arrows are pointing in the same direction. Compare this to the square on the upper-right of Figure 5.10. It is using the `GL_MIRRORED_REPEAT` mode and, as you can see, the texture has been repeated across the square. However, the first copy of the image is the right way around, then the next copy is flipped, the next copy is the right way around again, and so on.

One final texture repeat mode is `GL_MIRROR_CLAMP_TO_EDGE`, which is a hybrid of the `GL_MIRRORED_REPEAT` and `GL_CLAMP_TO_EDGE` modes. In `GL_MIRROR_CLAMP_TO_EDGE`, the texture coordinate is handled normally between 0.0 and 1.0; between 1.0 and 2.0 (and between 0.0 and −1.0), it is handled as if the mode were `GL_MIRRORED_REPEAT` (the texture is repeated mirrored); and then outside that range, it is clamped to the edge of the texture. This mode is useful when you

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.

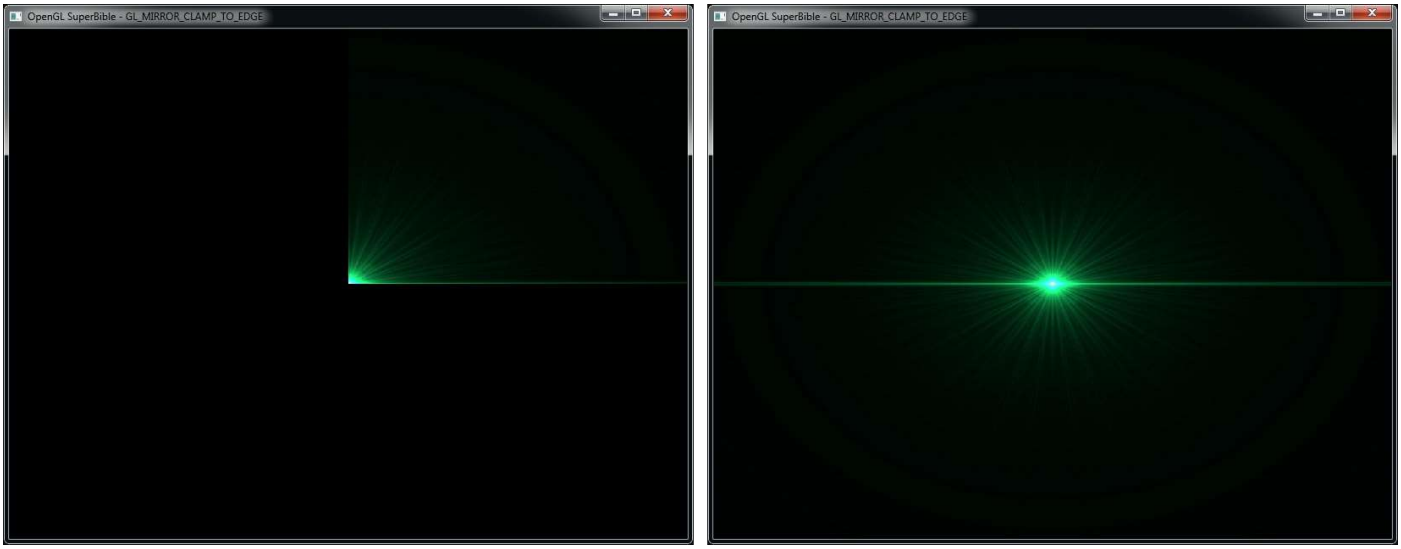

Figure 5.11: `GL_MIRROR_CLAMP_TO_EDGE` in action

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.