CREATING 3D GAMES

Game Programming in C++

Sanjay **MADHAV**

Game Programming in C++

This is regardless of the width and height of the window (hence *normalized* device coordinates). Internally, the graphics hardware then converts these NDC into the corresponding pixels in the window.

For example, to draw a square with sides of unit length in the center of the window, you need two triangles. The first triangle has the vertices (–0.5, 0.5), (0.5, 0.5), and (0.5, –0.5), and the second triangle has the vertices (0.5, –0.5), (–0.5, –0.5), and (–0.5, 0.5). Figure 5.2 illustrates this square. Keep in mind that if the length and width of the window are not uniform, a square in normalized device coordinates will not look like a square onscreen.

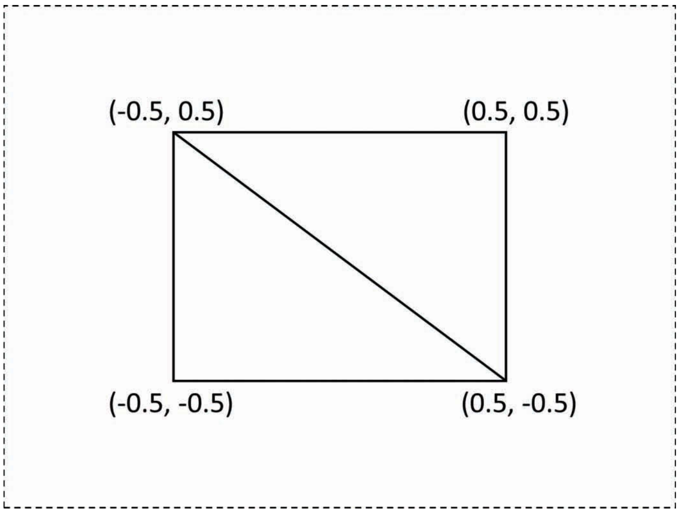

Figure 5.2 A square drawn in 2D normalized device coordinates

In 3D, the z component of normalized device coordinates also ranges from [–1, 1], with a positive z value going into the screen. For now, we stick with a z value of zero. We'll explore 3D in much greater detail in Chapter 6, "3D Graphics."

Vertex and Index Buffers

Suppose you have a 3D model comprised of many triangles. You need some way to store the vertices of these triangles in memory. The simplest approach is to directly store the coordinates of each triangle in a contiguous array or buffer. For example, assuming 3D coordinates, the following array contains the vertices of the two triangles shown in Figure 5.2:

```
float vertices[] = {
   -0.5f,  0.5f, 0.0f,
    0.5f,  0.5f, 0.0f,
    0.5f, -0.5f, 0.0f,
    0.5f, -0.5f, 0.0f,
   -0.5f, -0.5f, 0.0f,
   -0.5f,  0.5f, 0.0f,
};
```

Even in this simple example, the array of vertices has some duplicate data. Specifically, the coordinates (–0.5, 0.5, 0.0) and (0.5, –0.5, 0.0) appear twice. If there were a way to remove these duplicates, you would cut the number of values stored in the buffer by 33%. Rather than having 12 values, you would have only 8. Assuming single-precision floats that use 4 bytes each, you'd save 24 bytes of memory by removing the duplicates. This might seem insignificant, but imagine a much larger model with 20,000 triangles. In this case, the amount of memory wasted due to duplicate coordinates would be high.

The solution to this issue has two parts. First, you create a **vertex buffer** that contains only the *unique* coordinates used by the 3D geometry. Then, to specify the vertices of each triangle, you index into this vertex buffer (much like indexing into an array). The aptly named **index buffer** contains the indices for each individual triangle, in sets of three. For this example's sample square, you'd need the following vertex and index buffers:

```
float vertexBuffer[] = {
   -0.5f,   0.5f, 0.0f, // vertex 0
    0.5f,   0.5f, 0.0f, // vertex 1
    0.5f,  -0.5f, 0.0f, // vertex 2
    -0.5f, -0.5f, 0.0f  // vertex 3
};
unsigned short indexBuffer[] = {
   0, 1, 2,
   2, 3, 0
};
```

For example, the first triangle has the vertices 0, 1, and 2, which corresponds to the coordinates (–0.5, 0.5, 0.0), (0.5, 0.5, 0.0), and (0.5, –0.5, 0.0). Keep in mind that the index is the vertex number, not the floating-point element (for example, vertex 1 instead of "index 2" of the array). Also note that this code uses an unsigned short (typically 16 bits) for the index buffer, which reduces the memory footprint of the index buffer. You can use smaller bit size integers to save memory in the index buffer.

In this example, the vertex/index buffer combination uses $12 \times 4 + 6 \times 2$, or 60 total bytes. On the other hand, if you just used the original vertices, you'd need 72 bytes. While the savings in this example is only 20%, a more complex model would save much more memory by using the vertex/index buffer combination.

To use the vertex and index buffers, you must let OpenGL know about them. OpenGL uses a **vertex array object** to encapsulate a vertex buffer, an index buffer, and the vertex layout. The **vertex layout** specifies what data you store for each vertex in the model. For now, assume the vertex layout is a 3D position (you can just use a z component of `0.0f` if you want something 2D). Later in this chapter you'll add other data to each vertex.

Because any model needs a vertex array object, it makes sense to encapsulate its behavior in a `VertexArray` class. Listing 5.2 shows the declaration of this class.

Listing 5.2 `VertexArray` Declaration

```
class VertexArray
{
public:
   VertexArray(const float* verts, unsigned int numVerts,
      const unsigned int* indices, unsigned int numIndices);
   ~VertexArray();

   // Activate this vertex array (so we can draw it)
   void SetActive();

   unsigned int GetNumIndices() const { return mNumIndices; }
   unsigned int GetNumVerts() const { return mNumVerts; }
private:
   // How many vertices in the vertex buffer?
   unsigned int mNumVerts;
   // How many indices in the index buffer
   unsigned int mNumIndices;
   // OpenGL ID of the vertex buffer
   unsigned int mVertexBuffer;
   // OpenGL ID of the index buffer
   unsigned int mIndexBuffer;
   // OpenGL ID of the vertex array object
   unsigned int mVertexArray;
};
```

The constructor for `VertexArray` takes in pointers to the vertex and index buffer arrays so that it can hand off the data to OpenGL (which will ultimately load the data on the graphics hardware). Note that the member data contains several unsigned integers for the vertex buffer, index buffer, and vertex array object. This is because OpenGL does not return pointers to objects that it creates. Instead, you merely get back an integral ID number. Keep in mind that the ID numbers are not unique across different types of objects. It's therefore very possible to have an ID of 1 for both the vertex and index buffers because OpenGL considers them different types of objects.

The implementation of the `VertexArray` constructor is complex. First, create the vertex array object and store its ID in the `mVertexArray` member variable:

```
glGenVertexArrays(1, &mVertexArray);
glBindVertexArray(mVertexArray);
```

Once you have a vertex array object, you can create a vertex buffer:

```
glGenBuffers(1, &mVertexBuffer);
glBindBuffer(GL_ARRAY_BUFFER, mVertexBuffer);
```

The `GL_ARRAY_BUFFER` parameter to `glBindBuffer` means that you intend to use the buffer as a vertex buffer.

Once you have a vertex buffer, you need to copy the `verts` data passed into the `VertexArray` constructor into this vertex buffer. To copy the data, use `glBufferData`, which takes several parameters:

```
glBufferData(
   GL_ARRAY_BUFFER,               // The active buffer type to write to
   numVerts * 3 * sizeof(float), // Number of bytes to copy
   verts,                         // Source to copy from (pointer)
   GL_STATIC_DRAW                 // How will we use this data?
);
```

Note that you don't pass in the object ID to `glBufferData`; instead, you specify a currently bound buffer type to write to. In this case, `GL_ARRAY_BUFFER` means use the vertex buffer just created.

For the second parameter, you pass in the number of bytes, which is the amount of data for each vertex multiplied by the number of vertices. For now, you can assume that each vertex contains three floats for (x, y, z).

The usage parameter specifies how you want to use the buffer data. A `GL_STATIC_DRAW` usage means you only want to load the data once and use it frequently for drawing.

Next, create an index buffer. This is very similar to creating the vertex buffer, except you instead specify the `GL_ELEMENT_ARRAY_BUFFER` type, which corresponds to an index buffer:

```
glGenBuffers(1, &mIndexBuffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, mIndexBuffer);
```

Then copy the `indices` data into the index buffer:

```
glBufferData(
   GL_ELEMENT_ARRAY_BUFFER,            // Index buffer
   numIndices * sizeof(unsigned int), // Size of data
   indices, GL_STATIC_DRAW);
```

Note that the type here is `GL_ELEMENT_ARRAY_BUFFER`, and the size is the number of indices multiplied by an unsigned int because that's the type used for indices here.

Finally, you must specify a vertex layout, also called the **vertex attributes**. As mentioned earlier, the current layout is a position with three float values.

To enable the first vertex attribute (attribute 0), use `glEnableVertexAttribArray`:

```
glEnableVertexAttribArray(0);
```

You then use `glVertexAttribPointer` to specify the size, type, and format of the attribute:

```
glVertexAttribPointer(
   0,                 // Attribute index (0 for first one)
   3,                 // Number of components (3 in this case)
   GL_FLOAT,          // Type of the components
   GL_FALSE,          // (Only used for integral types)
   sizeof(float) * 3, // Stride (usually size of each vertex)
   0                  // Offset from start of vertex to this attribute
);
```

The first two parameters are 0 and 3 because the position is attribute 0 of the vertex, and there are three components (x, y, z). Because each component is a float, you specify the `GL_FLOAT` type. The fourth parameter is only relevant for integral types, so here you set it to `GL_FALSE`. Finally, the **stride** is the byte offset between consecutive vertices' attributes. But assuming you don't have padding in the vertex buffer (which you usually don't), the stride is just the size of the vertex. Finally, the offset is 0 because this is the only attribute. For additional attributes, you have to pass in a nonzero value for the offset.

The `VertexArray`'s destructor destroys the vertex buffer, index buffer, and vertex array object:

```
VertexArray::~VertexArray()
{
   glDeleteBuffers(1, &mVertexBuffer);
   glDeleteBuffers(1, &mIndexBuffer);
   glDeleteVertexArrays(1, &mVertexArray);
}
```

Finally, the `SetActive` function calls `glBindVertexArray`, which just specifies which vertex array you're currently using:.

```
void VertexArray::SetActive()
{
   glBindVertexArray(mVertexArray);
}
```

The following code in `Game::InitSpriteVerts` allocates an instance of `VertexArray` and saves it in a member variable of `Game` called `mSpriteVerts`:

```
mSpriteVerts = new VertexArray(vertexBuffer, 4, indexBuffer, 6);
```

The vertex and index buffer variables here are the arrays for the sprite quad. In this case, there are 4 vertices in the vertex buffer and 6 indices in the index buffer (corresponding to the 2 triangles in the quad). You will use this member variable later in this chapter to draw sprites, as all sprites will ultimately use the same vertices.

Shaders

In a modern graphics pipeline, you don't simply feed in the vertex/index buffers and have triangles draw. Instead, you specify how you want to draw the vertices. For example, should the triangles be a fixed color, or should they use a color from a texture? Do you want to perform lighting calculations for every pixel you draw?

Because there are many techniques you may want to use to display the scene, there is no truly one-size-fits-all method. To allow for more customization, graphics APIs including OpenGL support **shader programs**—small programs that execute on the graphics hardware to perform specific tasks. Importantly, shaders are *separate programs*, with their own separate `main` functions.

> **note**
>
> Shader programs do not use the C++ programming language. This book uses the GLSL programming language for shader programs. Although GLSL superficially looks like C, there are many semantics specific to GLSL. Rather than present all the details of GLSL at once, this book introduces the concepts as needed.

Because shaders are separate programs, you write them in separate files. Then in your C++ code, you need to tell OpenGL when to compile and load these shader programs and specify what you want OpenGL to use these shader programs for.

Although you can use several different types of shaders in games, this book focuses on the two most important ones: the vertex shader and the fragment (or pixel) shader.

Vertex Shaders

A **vertex shader** program runs once for every vertex of every triangle drawn. The vertex shader receives the vertex attribute data as an input. The vertex shader can then modify these vertex attributes as it sees fit. While it may seem unclear why you'd want to modify vertex attributes, it'll become more apparent as this chapter continues.