

Vertex Processing

The first programmable stage in the OpenGL pipeline (i.e., one that you can write a shader for) is the vertex shader. Before the shader runs, OpenGL will fetch the inputs to the vertex shader in the *vertex fetch* stage, which we will describe first. Your vertex shader's responsibility is to set the position¹ of the vertex that will be fed to the next stage in the pipeline. It can also set a number of other user-defined and built-in outputs that further describe the vertex to OpenGL.

¹. Under certain circumstances, you may even omit this.

Vertex Shader Inputs

The first step in any OpenGL graphics pipeline is actually the vertex fetch stage, unless the configuration does not require any vertex attributes, as was the case in some of our earliest examples. This stage runs before your vertex shader and is responsible for forming its inputs. You have already been introduced to the

glVertexAttribPointer() function and we have explained how it hooks data in buffers up to vertex shader inputs. Now we'll take a closer look at vertex attributes.

In the example programs presented thus far, we've used only a single vertex attribute and have filled it with four-component floating-point data, which matches the data types we have used for our uniforms, uniform blocks, and hard-coded constants. However,

OpenGL supports a large number of vertex attributes, and each can have its own format, data type, number of components, and so on. Also, OpenGL can read the data for each attribute from a different buffer object. **glVertexAttribPointer()** is a handy way to set up virtually everything about a vertex attribute. However, it can actually be considered more of a helper function that sits on top of a few lower-level functions: **glVertexAttribFormat()**, **glVertexAttribBinding()**, and **glBindVertexBuffer()**. Their prototypes are

[Click here to view code image](#)

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

void glVertexAttribBinding(GLuint attribindex,
                           GLuint bindingindex);

void glBindVertexBuffer(GLuint bindingindex,
                      GLuint buffer,
                      GLintptr offset,
                      GLintptr stride);
```

To understand how these functions work, first let's consider a simple vertex shader fragment that declares a number of inputs. In [Listing 7.1](#), notice the use of the location layout qualifier to set the locations of the inputs explicitly in the shader code.

[Click here to view code image](#)

```
#version 450 core

// Declare a number of vertex attributes
layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 tex_coord;
// Note that we intentionally skip location 3 here
layout (location = 4) in vec4 color;
layout (location = 5) in int material_id;
```

Listing 7.1: Declaration of multiple vertex attributes

The shader fragment in [Listing 7.1](#) declares five inputs: `position`, `normal`, `tex_coord`, `color`, and `material_id`. Now, consider that we are using a data structure to represent our vertices, which is defined in C as follows:

[Click here to view code image](#)

```
typedef struct VERTEX_t
{
```

```

    vmath::vec4      position;
    vmath::vec3      normal;
    vmath::vec2      tex_coord;
    GLubyte         color[3];
    int             material_id;
} VERTEX;

```

Notice that our vertex structure in C mixes use of vmath types and plain-old data (for `color`).

The first attribute is pretty standard and should be familiar to you—it's the position of the vertex, specified as a four-component floating-point vector. To describe this input using the **`glVertexAttribFormat()`** function, we would set `size` to 4 and `type` to `GL_FLOAT`. The second attribute, the normal of the geometry at the vertex, is in `normal` and would be passed to **`glVertexAttribFormat()`** with `size` set to 3 and `type` set to `GL_FLOAT`. Likewise, `tex_coord` can be used as a two-dimensional texture coordinate and might be specified by setting `size` to 2 and `type` to `GL_FLOAT`.

The `color` input to the vertex shader is declared as a `vec4`, but the `color` member of our `VERTEX` structure is actually an array of 3 bytes. Both the size (number of elements) and the data type are different. OpenGL can convert the data for you as it reads it into the vertex shader. To hook our 3-byte `color` member up to our four-component vertex shader input, we call **`glVertexAttribFormat()`** with `size` set to 3 and `type` set to `GL_UNSIGNED_BYTE`. This is where the `normalized` parameter comes in. As you probably know, the range of values representable by an unsigned byte is 0 to 255. However, that's not what we want in our vertex shader. There, we want to represent colors as values between 0.0 and 1.0. If you set `normalized` to `GL_TRUE`, then OpenGL will automatically divide each component of the input by the maximum possible representable positive value, *normalizing* it.

Because two's-complement numbers are able to represent a greater-magnitude negative number than positive number, this can place one value below -1.0 (-128 for `GLbyte`, $-32,768$ for `GLshort`, and $-2,147,483,648$ for `GLint`). Those most negative numbers are treated specially and are clamped to the floating-point value -1.0 during normalization. If `normalized` is `GL_FALSE`, then the value will be converted directly to floating point and presented to the vertex shader. In the case of unsigned byte data (like `color`), this means that the values will be between 0.0 and 255.0.

[Table 7.1](#) shows the tokens that can be used for the `type` parameter, their corresponding OpenGL types, and the range of values that they can represent.

Type	OpenGL Type	Range
GL_BYTE	GLbyte	-128 to 127
GL_SHORT	Glshort	-32,768 to 32,767
GL_INT	GLint	-2,147,483,648 to 2,147,483,647
GL_FIXED	GLfixed	-32,768 to 32,767
GL_UNSIGNED_BYTE	GLubyte	0 to 255
GL_UNSIGNED_SHORT	GLushort	0 to 65,535
GL_UNSIGNED_INT	GLuint	4,294,967,295
GL_HALF_FLOAT	GLhalf	—
GL_FLOAT	GLfloat	—
GL_DOUBLE	GLdouble	—

Table 7.1: Vertex Attribute Types

In [Table 7.1](#), the floating-point types (GLhalf, GLfloat, and GLdouble) don't have ranges because they can't be normalized. The GLfixed type is a special case. It represents *fixed-point* data that is made up of 32 bits with the binary point at position 16 (halfway through the number); as such, it is treated as one of the floating-point types and cannot be normalized.

In addition to the scalar types shown in [Table 7.1](#), **glVertexAttribFormat()** supports several *packed* data formats that use a single integer to store multiple components. The two packed data formats supported by OpenGL are

GL_UNSIGNED_INT_2_10_10_10_REV and GL_INT_2_10_10_10_REV, which both represent four components packed into a single 32-bit word.

The GL_UNSIGNED_INT_2_10_10_10_REV format provides 10 bits for each of the x , y , and z components of the vector and only 2 bits for the w component, which are all treated as unsigned quantities. This gives a range of 0 to 1023 for each of x , y , and z , and 0 to 3 for w . Likewise, the GL_INT_2_10_10_10_REV format provides 10 bits for x , y , and z , and 2 bits for w , but in this case each component is treated as a signed quantity. That means that while x , y , and z have a range of -512 to 511, w may range from -2 to 1. While this may not seem terribly useful, there are a number of use cases for three component vectors with more than 8 bits of precision (24 bits in total) but that do not require 16 bits of precision (48 bits in total). Even though those last 2 bits might be wasted, 10 bits of precision per component provides what is needed.

When one of the packed data types (GL_UNSIGNED_INT_2_10_10_10_REV or GL_INT_2_10_10_10_REV) is specified, then size must be set either to 4 or the special value GL_BGRA. The latter applies an automatic swizzle to the incoming data to reverse the order of the r , g , and b components (which are equivalent to the x , y , and z

components) of the incoming vectors. This provides compatibility with data stored in that order² without needing to modify your shaders.

2. The BGRA ordering is quite common in some image formats and is the default ordering used by some graphics APIs.

Finally, returning to our example vertex declaration, we have the `material_id` field, which is an integer. In this case, because we want to pass an integer value as is to the vertex shader, we'll use a variation on the **glVertexAttribFormat()**, **glVertexAttribIFormat()**, whose prototype is

[Click here to view code image](#)

```
void glVertexAttribIFormat(GLuint attribindex,
                           GLint size,
                           GLenum type,
                           GLuint relativeoffset);
```

Again, the `attribindex`, `size`, `type`, and `relativeoffset` parameters specify the attribute index, number of components, type of those components, and offset from the start of the vertex of the attribute that's being set up, respectively. However, you'll notice that the `normalized` parameter is missing. That's because this version of **glVertexAttribFormat()** is *only* for integer types—`type` must be one of the integer types (`GL_BYTE`, `GL_SHORT`, or `GL_INT`; one of their unsigned counterparts; or one of the packed data formats) and integer inputs to a vertex shader are never normalized. Thus, the complete code to describe our vertex format is

[Click here to view code image](#)

```
// position
glVertexAttribFormat(0, 4, GL_FLOAT, GL_FALSE, offsetof(VERTEX,
position));

// normal
glVertexAttribFormat(1, 3, GL_FLOAT, GL_FALSE, offsetof(VERTEX, normal));

// tex_coord
glVertexAttribFormat(2, 2, GL_FLOAT, GL_FALSE, offsetof(VERTEX,
texcoord));

// color[3]
glVertexAttribFormat(4, 3, GL_UNSIGNED_BYTE, GL_TRUE, offsetof(VERTEX,
color));

// material_id
glVertexAttribIFormat(5, 1, GL_INT, offsetof(VERTEX, material_id));
```

Now that you've set up the vertex attribute format, you need to tell OpenGL which buffers to read the data from. If you recall our discussion of uniform blocks and how they map to buffers, you can apply similar logic to vertex attributes. Each vertex shader

can have any number of input attributes (up to an implementation-defined limit), and OpenGL can provide data for them by reading from any number of buffers (again, up to a limit). Some vertex attributes can share space in a buffer; others may reside in different buffer objects. Rather than individually specifying which buffer objects are used for each vertex shader input, we can instead group inputs together and associate groups of them with a set of buffer binding points. Then, when you change the buffer bound to one of these binding points, it will change the buffer used to supply data for all of the attributes that are mapped to that binding point.

To establish the mapping between vertex shader inputs and buffer binding points, you can call **glVertexAttribBinding()**. The first parameter

glVertexAttribBinding(), `attribIndex`, is the index of the vertex attribute; the second parameter, `bindingIndex`, is the buffer binding point index. In our example, we're going to store all of the vertex attributes in a single buffer. To set this up, we simply call **glVertexAttribBinding()** once for each attribute and specify zero for the `bindingIndex` parameter each time:

[Click here to view code image](#)

```
glVertexAttribBinding(0, 0); // position  
glVertexAttribBinding(1, 0); // normal  
glVertexAttribBinding(2, 0); // tex_coord  
glVertexAttribBinding(4, 0); // color  
glVertexAttribBinding(5, 0); // material_id
```

However, we could establish a more complex binding scheme. Suppose, for example, that we wanted to store `position`, `normal`, and `tex_coord` in one buffer, `color` in a second buffer, and `material_id` in a third buffer. We could set this up as follows:

[Click here to view code image](#)

```
glVertexAttribBinding(0, 0); // position  
glVertexAttribBinding(1, 0); // normal  
glVertexAttribBinding(2, 0); // tex_coord  
glVertexAttribBinding(4, 1); // color  
glVertexAttribBinding(5, 2); // material_id
```

Finally, we need to bind a buffer object to each of the binding points that is used by our mapping. To do this, we call **glBindVertexBuffer()**. This function takes four parameters: `bindingIndex`, `buffer`, `offset`, and `stride`. The first is the index of the buffer binding point where you want to bind the buffer, and the second is the name of the buffer object that you're going to bind. `offset` is an offset into the buffer object where the vertex data starts, and `stride` is the distance, in bytes, between the start of each vertex's data in the buffer. If your data is tightly packed (that is, there are no gaps between the vertices), you can just set this to the total size of your vertex data (which

would be `sizeof(VERTEX)` in our example); otherwise, you'll need to add the size of the gaps to the size of the vertex data.

Vertex Shader Outputs

After your vertex shader has decided what to do with the vertex data, it must send the data to its outputs. We have already discussed the `gl_Position` built-in output variable, and have shown how you can create your own outputs from shaders that can be used to pass data into the following stages. Along with `gl_Position`, OpenGL defines a few more output variables—`gl_PointSize`, `gl_ClipDistance[]`, and `gl_CullDistance[]`—and wraps them up into an interface block called `gl_PerVertex`. Its declaration is

```
out gl_PerVertex
{
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
    float gl_CullDistance[];
};
```

Again, you should be familiar with `gl_Position`. `gl_ClipDistance[]` and `gl_CullDistance[]` are used for clipping and culling, which will be described in some detail later in this chapter. The other output, `gl_PointSize`, is used for controlling the size of points that might be rendered.

Variable Point Sizes

By default, OpenGL will draw points with a size of a single fragment. However, as you saw in [Chapter 2, “Our First OpenGL Program”](#), you can change the size of points that OpenGL draws by calling `glPointSize()`. The maximum size that OpenGL will draw your points is implementation defined, but it will be least 64 pixels. You determine the actual upper limit by calling `glGetInteger()` to find the value of `GL_POINT_SIZE_RANGE`. This function writes *two* integers to the output variable, so make sure you point it at an array of two integers. The first element of the array will be filled with the minimum point size (which will be at most 1), and the second element will be filled with the maximum point size.

Of course, setting all of your points to be big blobs isn't going to produce particularly appealing images. To deal with this issue, you can set the point size programmatically in the vertex shader (or whatever stage is last in the front end). To do this, write the desired value of the point diameter to the built-in variable `gl_PointSize`. Once you have a shader that does this, you need to tell OpenGL that you wish to use the size written to the point size variable. To do this, call

[Click here to view code image](#)

```
glEnable(GL_PROGRAM_POINT_SIZE);
```

A common use for this function is to determine the size of a point based on its distance from the viewer. When you use the **glPointSize()** function to set the size of points, every point will have the same size no matter what their positions are. By choosing a value for `gl_PointSize`, you can implement any function you wish and each point produced by a single draw command can have a different size. This includes points generated in the geometry shader or by the tessellation engine when the tessellation evaluation shader specifies `point_mode`.

The following formula is often used to implement distance-based point size attenuation, where d is the distance of the point from the eye and a , b , and c are configurable parameters of a quadratic equation. You can store those in uniforms and update them with your application, or if you have a particular set of parameters in mind, you might want to make them constants in your vertex shader. For example, if you want a constant size, set a to a non-zero value and b and c to zero. If a and c are zero and b is non-zero, then point size will fall off linearly with distance. Likewise, if a and b are zero but c is non-zero, then point size will fall off quadratically with distance.

$$\text{size} = \text{clamp} \left(\sqrt{\frac{1.0}{a + b \times d + c \times d^2}} \right)$$

Drawing Commands

Until now, we have written every example using only a single drawing command —**glDrawArrays()**. OpenGL includes many drawing commands, however. While some could be considered supersets of others, they can be generally categorized as either indexed or non-indexed and as direct or indirect. All of these possibilities are covered in the next few sections.

Indexed Drawing Commands

The **glDrawArrays()** command is a non-indexed drawing command. That is, the vertices are issued in order, and any vertex data stored in buffers and associated with vertex attributes is simply fed to the vertex shader in the order that it appears in the buffer. An indexed draw, in contrast, includes an indirection step that treats the data in each of those buffers as an array; rather than index into that array sequentially, the command reads from another array of indices. After reading the index, OpenGL uses its value to index into the array. To make an indexed drawing command work, you need to bind a buffer to the `GL_ELEMENT_ARRAY_BUFFER` target. This buffer will contain the indices of the vertices that you want to draw. Next, you call one of the indexed

drawing commands, all of which contain the word `Elements` in their names. For example, **`glDrawElements()`** is the simplest of these functions; its prototype is

[Click here to view code image](#)

```
void glDrawElements(GLenum mode,
                    GLsizei count,
                    GLenum type,
                    const GLvoid * indices);
```

When you call **`glDrawElements()`**, `mode` and `type` have the same meaning as they do for **`glDrawArrays()`**. `type` specifies the type of data used to store each index and may be `GL_UNSIGNED_BYTE` to indicate 1 byte per index, `GL_UNSIGNED_SHORT` to indicate 16 bits per index, or `GL_UNSIGNED_INT` to indicate 32 bits per index. Although `indices` is defined as a pointer, it is actually interpreted as the offset into the buffer currently bound to the `GL_ELEMENT_ARRAY_BUFFER` binding where the first index is stored. [Figure 7.1](#) shows how the indices specified by a call to **`glDrawElements()`** are used by OpenGL.

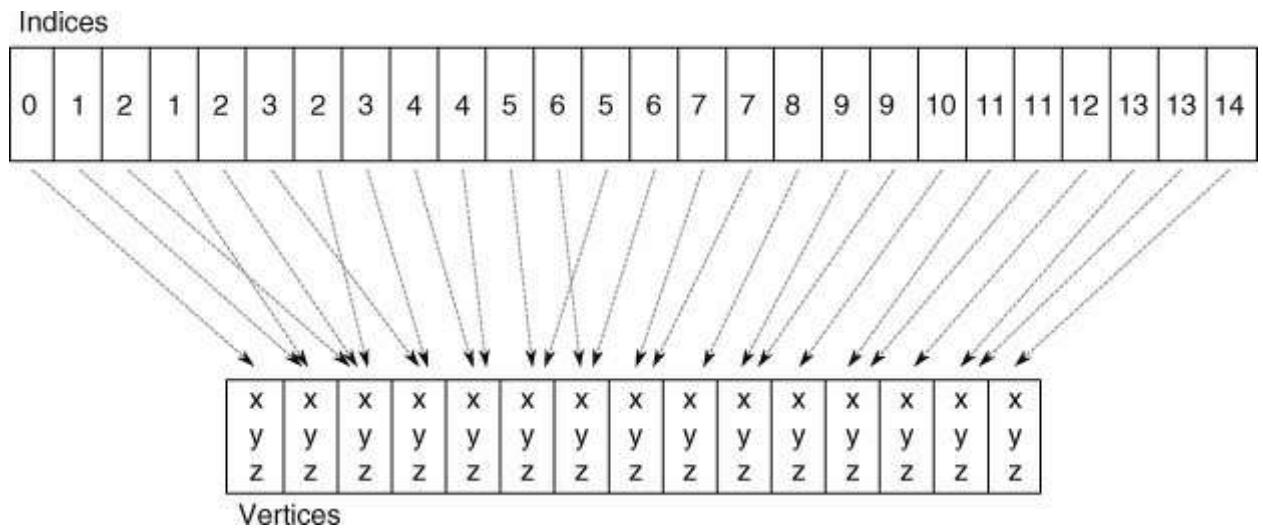


Figure 7.1: Indices used in an indexed draw

The **`glDrawArrays()`** and **`glDrawElements()`** commands are actually subsets of the complete functionality supported by the direct drawing commands of OpenGL. The set of the most generalized OpenGL drawing commands is given in [Table 7.2](#); all other OpenGL drawing commands can be expressed in terms of these functions.

Draw Type	Command
Direct, Non-Indexed	<code>glDrawArraysInstancedBaseInstance()</code>
Direct, Indexed	<code>glDrawElementsInstancedBaseVertexBaseInstance()</code>
Indirect, Non-Indexed	<code>glMultiDrawArraysIndirect()</code>
Indirect, Indexed	<code>glMultiDrawElementsIndirect()</code>

Table 7.2: Draw Type Matrix

Recall the spinning cube example in [Chapter 5](#), “[Data](#),” and in particular the geometry setup performed in [Listing 5.22](#). To draw a cube, we drew 12 triangles (two for each face of the cube), and each one consumed 36 vertices. However, a cube really has only 8 corners, so it should need only 8 vertices of information, right? We can use an indexed draw to greatly reduce the amount of vertex data, especially for geometry that has a lot of vertices. We can rewrite the setup code of [Listing 5.22](#) to define just the 8 corners of the cube, but also define a set of 36 indices that tell OpenGL which corner to use for each vertex of each triangle. The new setup code looks like [Listing 7.2](#).

[Click here to view code image](#)

```

static const GLfloat vertex_positions[] =
{
    -0.25f, -0.25f, -0.25f,
    -0.25f, 0.25f, -0.25f,
    0.25f, -0.25f, -0.25f,
    0.25f, 0.25f, -0.25f,
    0.25f, -0.25f, 0.25f,
    0.25f, 0.25f, 0.25f,
    -0.25f, -0.25f, 0.25f,
    -0.25f, 0.25f, 0.25f,
};

static const GLushort vertex_indices[] =
{
    0, 1, 2,
    2, 1, 3,
    2, 3, 4,
    4, 3, 5,
    4, 5, 6,
    6, 5, 7,
    6, 7, 0,
    0, 7, 1,
    6, 0, 2,
    2, 4, 6,
    7, 5, 3,
    7, 3, 1
};

```

```

glGenBuffers(1, &position_buffer);
glBindBuffer(GL_ARRAY_BUFFER, position_buffer);
glBufferData(GL_ARRAY_BUFFER,
             sizeof(vertex_positions),
             vertex_positions,
             GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
 glEnableVertexAttribArray(0);

glGenBuffers(1, &index_buffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, index_buffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
             sizeof(vertex_indices),
             vertex_indices,
             GL_STATIC_DRAW);

```

Listing 7.2: Setting up indexed cube geometry

As you can see from [Listing 7.2](#), the total amount of data required to represent our cube is greatly reduced—it went from 108 floating-point values (36 triangles times 3 components each, which is 432 bytes) down to 24 floating-point values (just the 8 corners at 3 components each, which is 72 bytes) and 36 16-bit integers (another 72 bytes), for a total of 144 bytes, representing a reduction of two thirds. To use the index data in `vertex_indices`, we need to bind a buffer to the `GL_ELEMENT_ARRAY_BUFFER` and put the indices in it just as we did with the vertex data. In [Listing 7.2](#), we do that immediately after we set up the buffer containing the vertex positions.

Once you have a set of vertices and their indices in memory, you’ll need to change your rendering code to use **`glDrawElements()`** (or one of the more advanced versions of it) instead of **`glDrawArrays()`**. Our new rendering loop for the spinning cube example is shown in [Listing 7.3](#).

[Click here to view code image](#)

```

// Clear the framebuffer with dark green
static const GLfloat green[] = { 0.0f, 0.25f, 0.0f, 1.0f };
glClearBufferfv(GL_COLOR, 0, green);

// Activate our program
glUseProgram(program);

// Set the model-view and projection matrices
glUniformMatrix4fv(mv_location, 1, GL_FALSE, mv_matrix);
glUniformMatrix4fv(proj_location, 1, GL_FALSE, proj_matrix);

// Draw 6 faces of 2 triangles of 3 vertices each = 36 vertices
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_SHORT, 0);

```

Listing 7.3: Drawing indexed cube geometry

Notice that we're still drawing 36 vertices, but now 36 *indices* will be used to index into an array of only 8 unique vertices. The result of rendering with the vertex index and position data in our two buffers and a call to **glDrawElements()** is identical to that shown in [Figure 5.2](#).

The Base Vertex

The first advanced version of **glDrawElements()** that takes an extra parameter is **glDrawElementsBaseVertex()**, whose prototype is

[Click here to view code image](#)

```
void glDrawElementsBaseVertex(GLenum mode,
                           GLsizei count,
                           GLenum type,
                           GLvoid * indices,
                           GLint basevertex);
```

When you call **glDrawElementsBaseVertex()**, OpenGL will fetch the vertex index from the buffer bound to the `GL_ELEMENT_ARRAY_BUFFER` and then add `basevertex` to it before it is used to index into the array of vertices. This allows you to store a number of different pieces of geometry in the same buffer and then offset into it using `basevertex`. [Figure 7.2](#) shows how `base vertex` is added to vertices in an indexed drawing command.

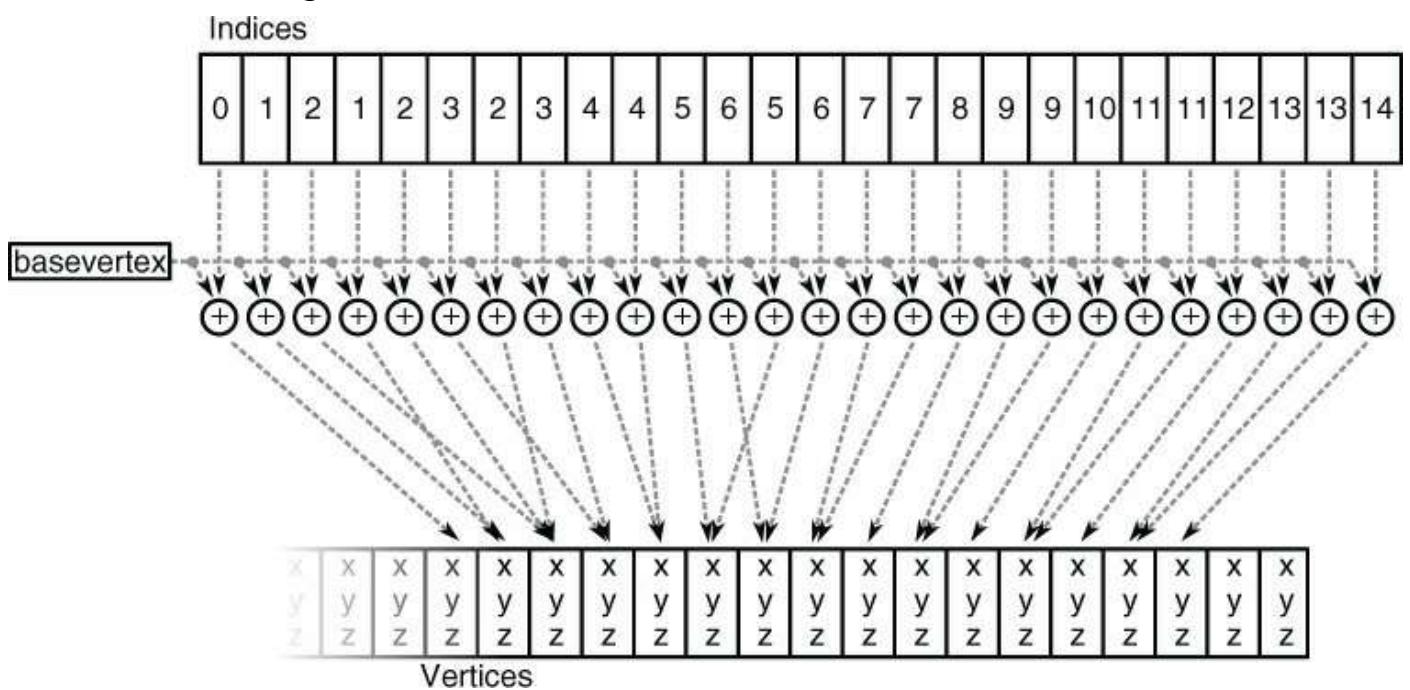


Figure 7.2: Base vertex used in an indexed draw

As you can see from [Figure 7.2](#), each vertex index is essentially fed into an addition

operation, which adds the base vertex to it before OpenGL fetches the underlying vertex data. Clearly, if `basevertex` is zero, then **`glDrawElementsBaseVertex()`** is equivalent to **`glDrawElements()`**. In fact, we consider calling **`glDrawElements()`** to be equivalent to calling **`glDrawElementsBaseVertex()`** with `basevertex` set to zero.

Combining Geometry Using Primitive Restart

Many tools are available that “stripify” geometry. The idea underlying these tools is that by taking “triangle soup,” meaning a large collection of unconnected triangles, and attempting to merge it into a set of triangle strips, performance can be improved. This works because each individual triangle is represented by three vertices, but a triangle strip reduces this to a single vertex per triangle (not counting the first triangle in the strip). By converting the geometry from triangle soup to triangle strips, there is less geometry data to process, and the system should run faster. If the tool does a good job and produces a small number of long strips containing many triangles each, this generally works well. There has been a lot of research into this type of algorithm, and a new method’s success is measured by passing some well-known models through the new “stripifier” and comparing the number and average length of the strips generated by the new tool to those produced by current state-of-the-art stripifiers.

Despite all of this research, the reality is that a soup can be rendered with a single call to **`glDrawArrays()`** or **`glDrawElements()`**, but unless we use the functionality that is about to be introduced, a set of strips needs to be rendered with separate calls to OpenGL. As a consequence, there are likely to be a lot more function calls in a program that uses stripified geometry. Moreover, if the stripping application hasn’t done a decent job or if the model just doesn’t lend itself well to stripification, this can eat up any performance gains achieved by using strips in the first place.

A feature that can help here is *primitive restart*. Primitive restart applies to the `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_LINE_STRIP`, and `GL_LINE_LOOP` geometry types. It is a method of informing OpenGL when one strip (or fan or loop) has ended and another should be started. To indicate the position in the geometry where one strip ends and the next begins, a special marker is placed as a reserved value in the element array. As OpenGL fetches vertex indices from the element array, it checks for this special index value; whenever it comes across it, OpenGL ends the current strip and starts a new one with the next vertex. This mode is disabled by default but can be enabled by calling

[Click here to view code image](#)

```
glEnable(GL_PRIMITIVE_RESTART);
```

and disabled again by calling

[Click here to view code image](#)

```
glDisable(GL_PRIMITIVE_RESTART);
```

When primitive restart mode is enabled, OpenGL watches for the special index value as it fetches values from the element array buffer. When it comes across this value, it stops the current strip and starts a new one. To set the index that OpenGL should watch for, call

[Click here to view code image](#)

```
glPrimitiveRestartIndex(index);
```

OpenGL watches for the value specified by `index` and uses that as the primitive restart marker. Because the marker is a vertex index, primitive restart is best used with indexed drawing functions such as **glDrawElements()**. If you draw with a non-indexed drawing command such as **glDrawArrays()**, the primitive restart index is simply ignored.

The default value of the primitive restart index is zero. Because that's almost certainly the index of a real vertex that will be contained in the model, it's a good idea to set the restart index to a new value whenever you're using primitive restart mode. A good value to use is the maximum value representable by the index type you're using (0xFFFFFFFF for `GL_UNSIGNED_INT`, 0xFFFF for `GL_UNSIGNED_SHORT`, and 0xFF for `GL_UNSIGNED_BYTE`) because you can be almost certain that it will not be used as a valid index of a vertex. Many stripping tools have an option to create either separate strips or a single strip with the restart index in it. The stripping tool may use a predefined index or output the index it used when creating the stripped version of the model (for example, one greater than the number of vertices in the model). You need to know which option is used and use the **glPrimitiveRestartIndex()** function to set it to use the output of the tool in your application. The primitive restart feature is illustrated in [Figure 7.3](#).

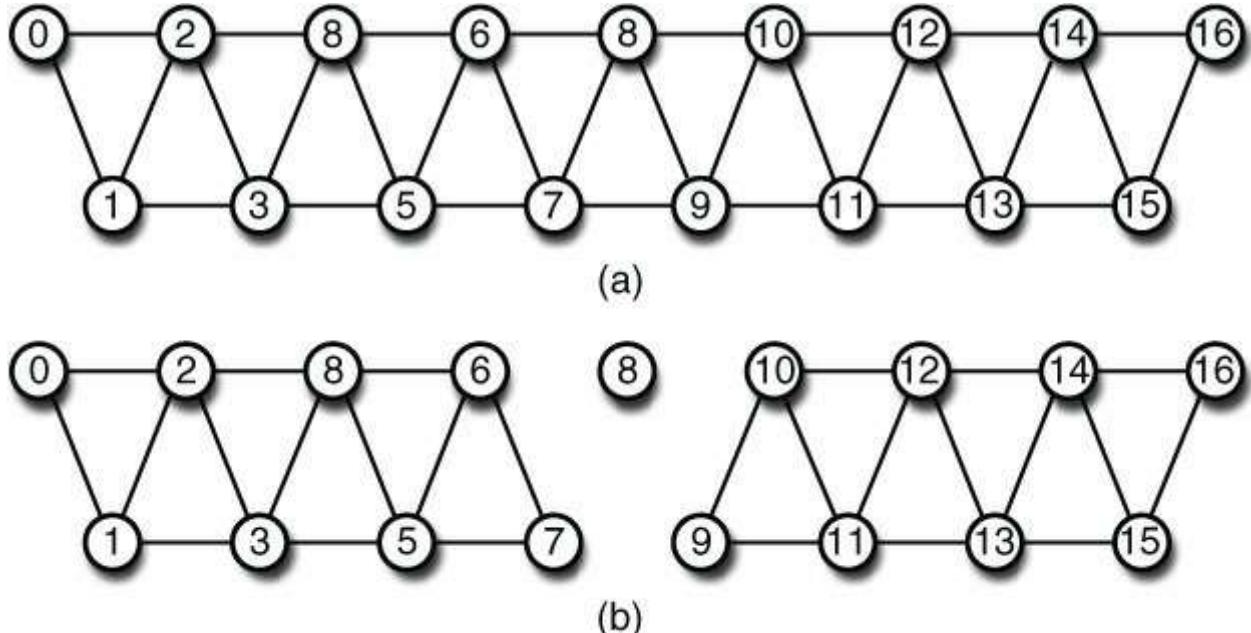


Figure 7.3: Triangle strips (a) without primitive restart and (b) with primitive restart

In [Figure 7.3](#), a triangle strip is pictured with the vertices marked with their indices. In [Figure 7.3\(a\)](#), the strip is made up of 17 vertices, which produces a total of 15 triangles in a single, connected strip. By enabling primitive restart mode and setting the primitive restart index to 8, as in [Figure 7.3\(b\)](#), the eighth index (whose value is also 8) is recognized by OpenGL as the special restart marker, and the triangle strip is terminated at vertex 7. The actual position of vertex 8 is ignored because it is not seen by OpenGL as the index of a real vertex. The next vertex processed (vertex 9) becomes the start of a new triangle strip. Thus, while 17 vertices are still sent to OpenGL, the result is that two separate triangle strips of 8 vertices and 6 triangles each are drawn.

Instancing

There will probably be times when you want to draw the same object many times. Imagine a fleet of starships or a field of grass. There could be thousands of copies of what are essentially identical sets of geometry, modified only slightly from instance to instance. A simple application might just loop over all of the individual blades of grass in a field and render them separately, calling **glDrawArrays()** once for each blade and perhaps updating a set of shader uniforms on each iteration. If each blade of grass was made up of a strip of four triangles, the code might look something like [Listing 7.4](#).

[Click here to view code image](#)

```
glBindVertexArray(grass_vao);
for (int n = 0; n < number_of_blades_of_grass; n++)
{
    SetupGrassBladeParameters();
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 6);
}
```

Listing 7.4: Drawing the same geometry many times

How many blades of grass are there in a field? What is the value of `number_of_blades_of_grass`? It could be thousands, maybe millions. Each blade of grass is likely to take up a very small area on the screen, and the number of vertices representing the blade is also very small. Your graphics card doesn't really have a lot of work to do to render a single blade of grass, and the system is likely to spend most of its time sending commands to OpenGL rather than actually drawing anything. OpenGL addresses this through instanced rendering, which is a way to ask it to draw many copies of the same geometry.

Instanced rendering is a method provided by OpenGL to specify that you want to draw many copies of the same geometry with a single function call. This functionality is accessed through instanced rendering functions, such as

[Click here to view code image](#)

```
void glDrawArraysInstanced(GLenum mode,
                           GLint first,
                           GLsizei count,
                           GLsizei instancecount);
```

and

[Click here to view code image](#)

```
void glDrawElementsInstanced(GLenum mode,
                             GLsizei count,
                             GLenum type,
                             const void * indices,
                             GLsizei instancecount);
```

These two functions behave much like **glDrawArrays()** and **glDrawElements()**, except that they tell OpenGL to render `instancecount` copies of the geometry. The first parameters of each (`mode`, `first`, and `count` for

glDrawArraysInstanced(), and `mode`, `count`, `type`, and `indices` for **glDrawElementsInstanced()**) have the same meaning as in the regular, non-instanced versions of the functions. When you call one of these functions, OpenGL makes any preparations it needs to draw your geometry (such as copying vertex data to the graphics card's memory) only once and then renders the same vertices many times.

If you set `instancecount` to 1, then **glDrawArraysInstanced()** and **glDrawElementsInstanced()** will draw a single instance of your geometry.

Obviously, this is equivalent to calling **glDrawArrays()** or **glDrawElements()**, but we normally state this equivalency the other way around—that is, we say that calling **glDrawArrays()** is equivalent to calling **glDrawArraysInstanced()** with

instancecount set to 1 and that calling **glDrawElements()** is equivalent to calling **glDrawElementsInstanced()** with instancecount set to 1. As we discussed earlier, though, calling **glDrawElements()** is also equivalent to calling **glDrawElementsBaseVertex()** with basevertex set to 0. In fact, there is another drawing command that combines both basevertex and instancecount: **glDrawElementsInstancedBaseVertex()**. Its prototype is

[Click here to view code image](#)

```
void glDrawElementsInstancedBaseVertex(GLenum mode,  
                                      GLsizei count,  
                                      GLenum type,  
                                      GLvoid * indices,  
                                      GLsizei instancecount,  
                                      GLint basevertex);
```

So, in fact, calling **glDrawElements()** is equivalent to calling **glDrawElementsInstancedBaseVertex()** with instancecount set to 1 and basevertex set to 0. Likewise, calling **glDrawElementsInstanced()** is equivalent to calling **glDrawElementsInstancedBaseVertex()** with basevertex set to 0.

Finally, just as we can pass basevertex to **glDrawElementsBaseVertex()** and **glDrawElementsInstancedBaseVertex()**, so we can pass a baseinstance parameter to versions of the instanced drawing commands. These functions are **glDrawArraysInstancedBaseInstance()**, **glDrawElementsInstancedBaseInstance()**, and the exceedingly long **glDrawElementsInstancedBaseVertexBaseInstance()**, which takes both a basevertex and a baseinstance parameter. Now that we have introduced all of the direct drawing commands, it should be clear that they are all subsets of **glDrawArraysInstancedBaseInstance()** and **glDrawElementsInstancedBaseVertexBaseInstance()**, and that where they are missing, basevertex and baseinstance are assumed to be 0 and instancecount is assumed to be 1.

If all that these functions did was send many copies of the same vertices to OpenGL as if **glDrawArrays()** or **glDrawElements()** had been called in a tight loop, they wouldn't be very useful. In fact, one thing that makes instanced rendering usable and very powerful is a special, built-in variable in GLSL named `gl_InstanceID`. The `gl_InstanceID` variable appears in the vertex as if it were a static integer vertex attribute. When the first copy of the vertices is sent to OpenGL, `gl_InstanceID` will be 0. It will then be incremented once for each copy of the geometry and will eventually reach instancecount -1.

The **glDrawArraysInstanced()** function essentially operates as if the pseudocode in [Listing 7.5](#) were executed. To reiterate, this is pseudocode—`gl_InstanceID` is not a real vertex attribute and you can't set it as shown in the listing.

[Click here to view code image](#)

```
// Loop over all of the instances (i.e. instancecount)
for (int n = 0; n < instancecount; n++)
{
    // Set the gl_InstanceID attribute - here gl_InstanceID is a C
    // variable
    // holding the location of the "virtual" gl_InstanceID input.
    glVertexAttribI(gl_InstanceID, n);

    // Now, when we call glDrawArrays, the gl_InstanceID variable in the
    // shader will contain the index of the instance that's being
    // rendered.
    glDrawArrays(mode, first, count);
}
```

Listing 7.5: Pseudocode for **glDrawArraysInstanced()**

Likewise, the **glDrawElementsInstanced()** function operates similarly to the pseudocode in [Listing 7.6](#). Again, this is pseudocode and won't compile as written.

[Click here to view code image](#)

```
for (int n = 0; n < instancecount; n++)
{
    // Set the value of gl_InstanceID
    glVertexAttribI(gl_InstanceID, n);

    // Make a normal call to glDrawElements
    glDrawElements(mode, count, type, indices);
}
```

Listing 7.6: Pseudocode for **glDrawElementsInstanced()**

Of course, `gl_InstanceID` is not a real vertex attribute, and you can't get a location for it by calling **glGetAttribLocation()**. The value of `gl_InstanceID` is managed by OpenGL and is very likely generated in hardware, meaning that it's essentially free to use in terms of performance. The power of instanced rendering comes from imaginative use of this variable, along with instanced arrays, which are explained in a moment.

The value of `gl_InstanceID` can be used directly as a parameter to a shader function or to index into data such as textures or uniform arrays. To return to our example of the field of grass, let's figure out what we're going to do with `gl_InstanceID` to make our field not just be thousands of identical blades of grass

growing out of a single point. Each of our grass blades is made out of a little triangle strip with four triangles in it, for a total of just six vertices. It could be tricky to get them to all look different. However, with some shader magic, we can make each blade of grass look sufficiently different so as to produce an interesting output. We won't go over the shader code here, but we will walk through a few ideas of how you can use `gl_InstanceID` to add variation to your scenes.

First, we need each blade of grass to have a different position; otherwise, they'll all be drawn on top of each other. Let's arrange the blades of grass more or less evenly. If the number of blades of grass we're going to render is a power of 2, we can use half of the bits of `gl_InstanceID` to represent the *x* coordinate of the blade, and the other half to represent the *z* coordinate (our ground lies in the *x*-*z* plane, with *y* being altitude).

For this example, we render 2^{20} , or a little over 1 million blades of grass (actually 1,048,576 blades, but who's counting?). By using the ten least significant bits (bits 9 through 0) as the *x* coordinate and the ten most significant bits (19 through 10) as the *z* coordinate, we have a uniform grid of grass blades. Let's take a look at [Figure 7.4](#) to see what we have so far.

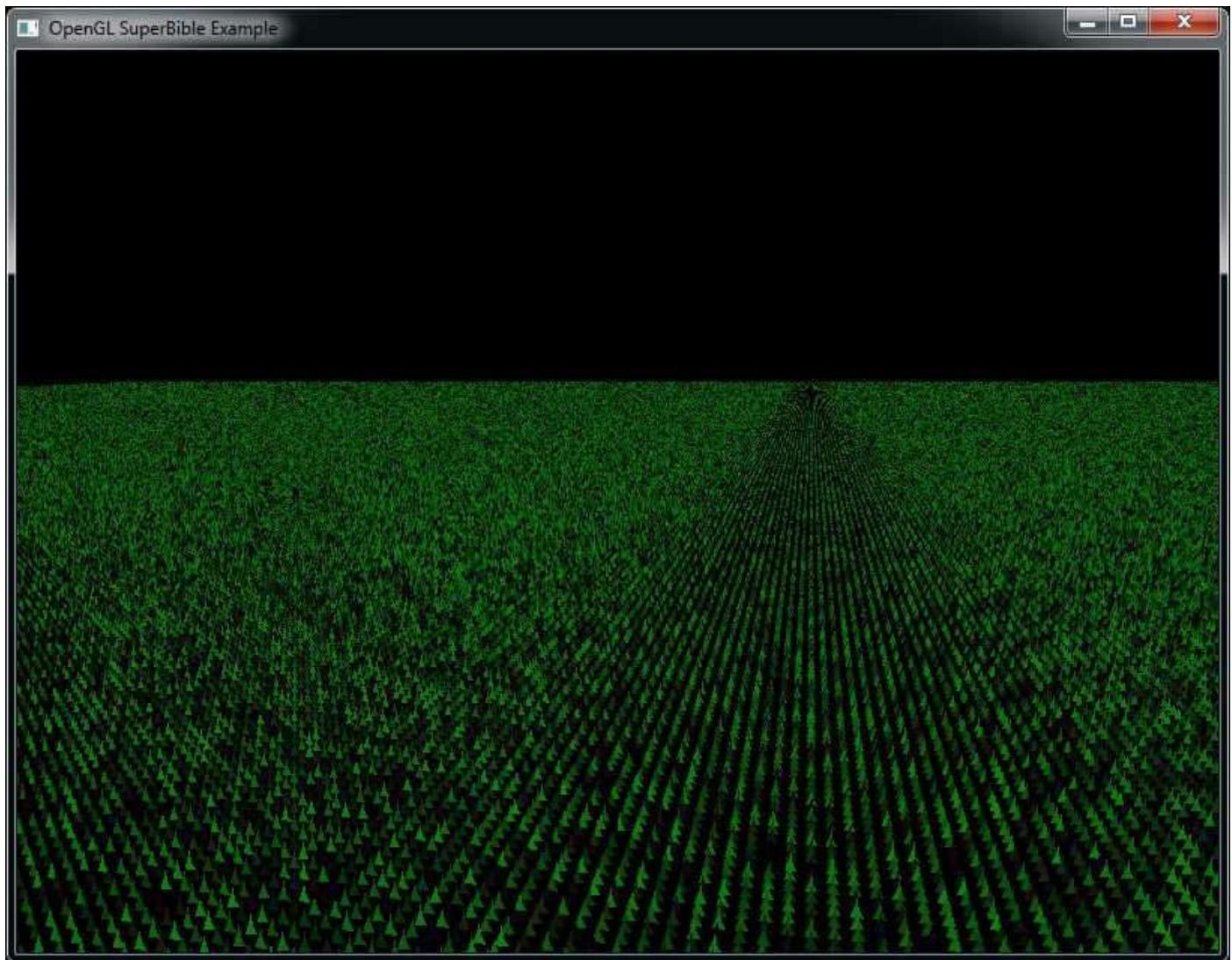


Figure 7.4: First attempt at an instanced field of grass

Our uniform grid of grass probably looks a little plain, as if a particularly attentive groundskeeper hand-planted each blade. What we really need to do is displace each blade of grass by some random amount within its grid square. That'll make the field look a little less uniform. A simple way of generating random numbers is to multiply a seed value by a large number and take a subset of the bits of the resulting product and use it as the input to a function. We're not aiming for a perfect distribution here, so this simple generator should do. Usually, with this type of algorithm, you'd reuse the seed value as input to the next iteration of the random number generator. In this case, though, we can just use `gl_InstanceID` directly because we're really generating the next few numbers after `gl_InstanceID` in a pseudo-random sequence. By iterating over our pseudo-random function only a couple of times, we can get a reasonably random distribution. Because we need to displace in both the x and z directions, we generate two successive random numbers from `gl_InstanceID` and use them to displace the blade of grass within the plane. Look at [Figure 7.5](#) to see what we get now.

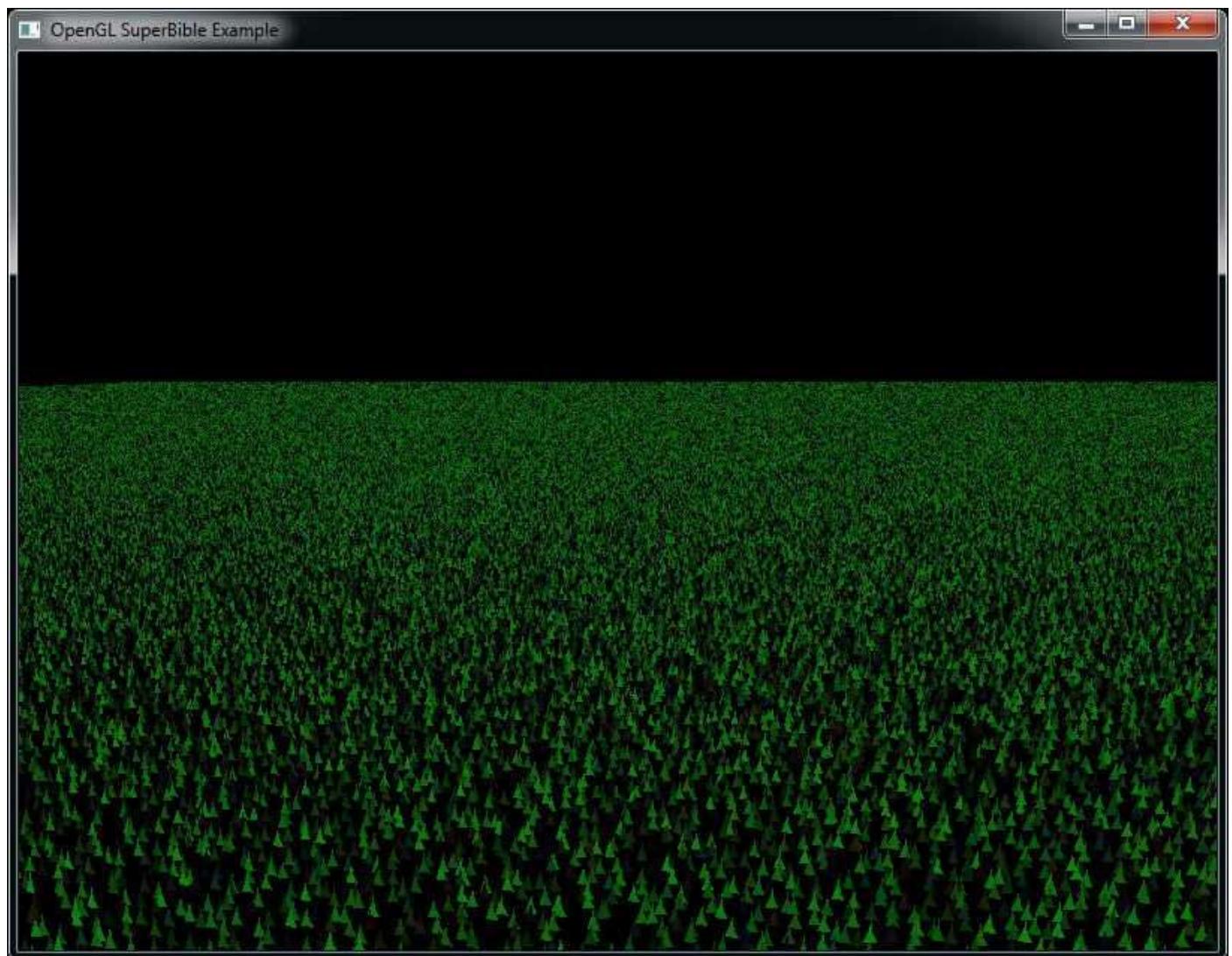


Figure 7.5: Slightly perturbed blades of grass

At this point, our field of grass is distributed evenly, with random perturbations in position for each blade of grass. All the grass blades look the same, though. (Actually, we used the same random number generator to assign a slightly different color to each blade of grass just so that they'd show up in the figures.) We can apply some variation over the field to make each blade look slightly different. This is something that we'd probably want to have control over, so we use a texture to hold information about blades of grass.

You have x and z coordinates for each blade of grass that were calculated by generating a grid coordinate directly from `gl_InstanceID` and then generating a random number and displacing the blade within the x - z plane. That coordinate pair can be used to look up a texel within a 2D texture, and you can put whatever you want in it. Let's control the length of the grass using the texture. We can put a length parameter in the texture (let's use the red channel) and multiply the y coordinate of each vertex of the grass geometry by that to make longer or shorter grass. A value of 0 in the texture would produce very short (or nonexistent) grass, and a value of 1 would produce grass of some maximum length. Now you can design a texture where each texel represents the length of the grass in a region of your field. Why not draw a few crop circles?

At this point, the grass is evenly distributed over the field, and you have control of the length of the grass in different areas. However, the grass blades are still just scaled copies of each other. Perhaps we can introduce some more variation. We decide to rotate each blade of grass around its axis according to another parameter from the texture. We use the green channel of the texture to store the angle through which the grass blade should be rotated around the y axis, with 0 representing no rotation and 1 representing a full 360 degrees. We've still performed only one texture fetch in our vertex shader, and the only input to the shader is still `gl_InstanceID`. Things are starting to come together. Take a look at [Figure 7.6](#).

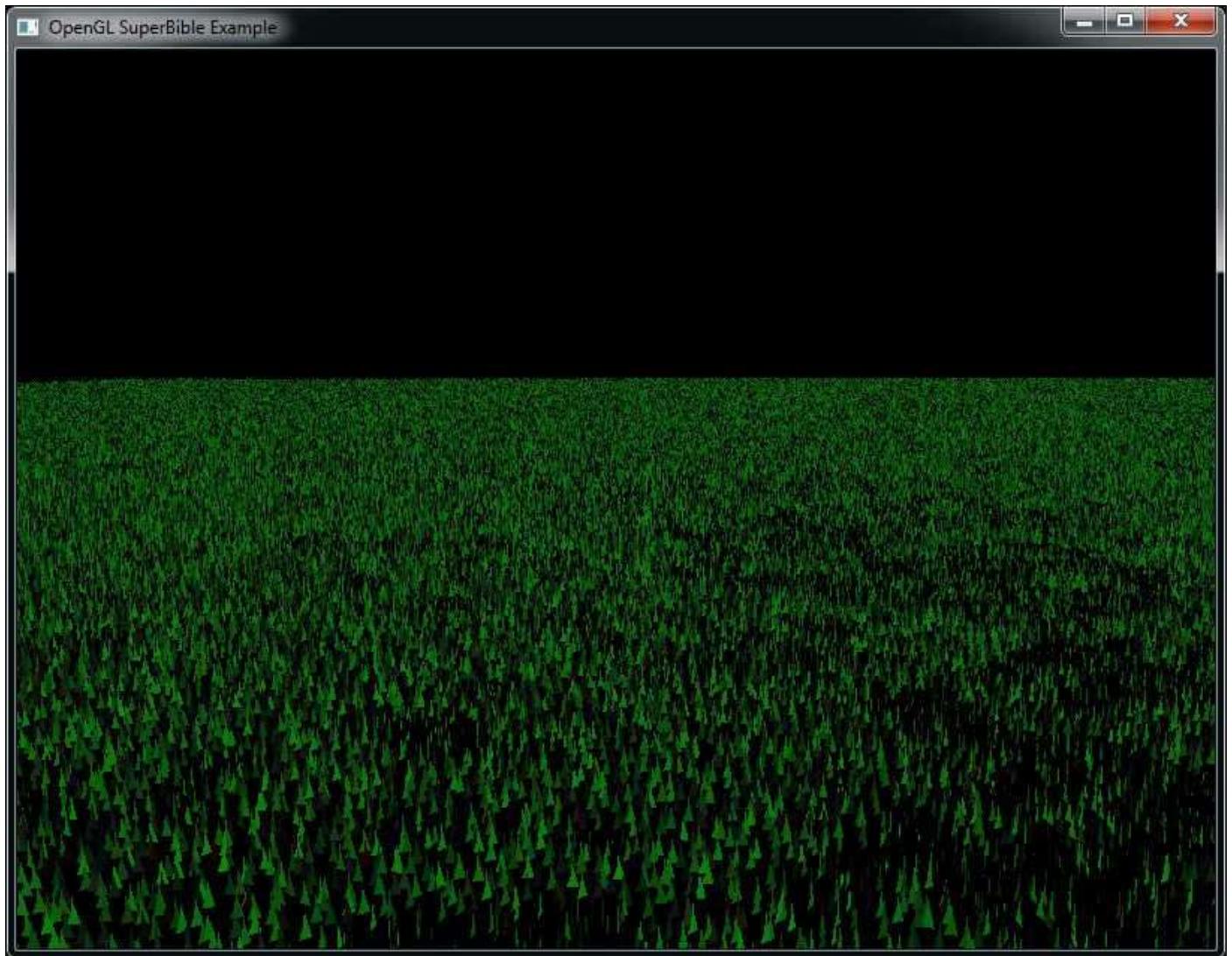


Figure 7.6: Control over the length and orientation of our grass

Our field is still looking a little bland. The grass just sticks straight up and doesn't move. Real grass sways in the wind and gets flattened when things roll over it. We need the grass to bend, and we'd like to have control over that behavior. Why not use another channel from the parameter texture (the blue channel) to control a bend factor? We can use that as another angle and rotate the grass around the x axis before we apply the rotation in the green channel. This allows us to make the grass bend over based on the parameter in the texture. Use 0 to represent no bending (the grass stands straight up) and 1 to represent fully flattened grass. Normally, the grass will sway gently, so the parameter will have a low value. When the grass gets flattened, the value can be much higher.

Finally, we can control the color of the grass. It seems logical to just store the color of the grass in a large texture. This might be a good idea if you want to draw a sports field with lines, markings, or advertising on it, for example, but it's fairly wasteful if the grass is all varying shades of green. Instead, let's make a palette for our grass in a 1D texture and use the final channel within our parameter texture (the alpha channel) to

store the index into that palette. The palette can start with an anemic-looking dead-grass yellow at one end and a lush, deep green at the other end. Now we read the alpha channel from the parameter texture along with all the other parameters and use it to index into the 1D texture—a dependent texture fetch. Our final field is shown in [Figure 7.7](#).

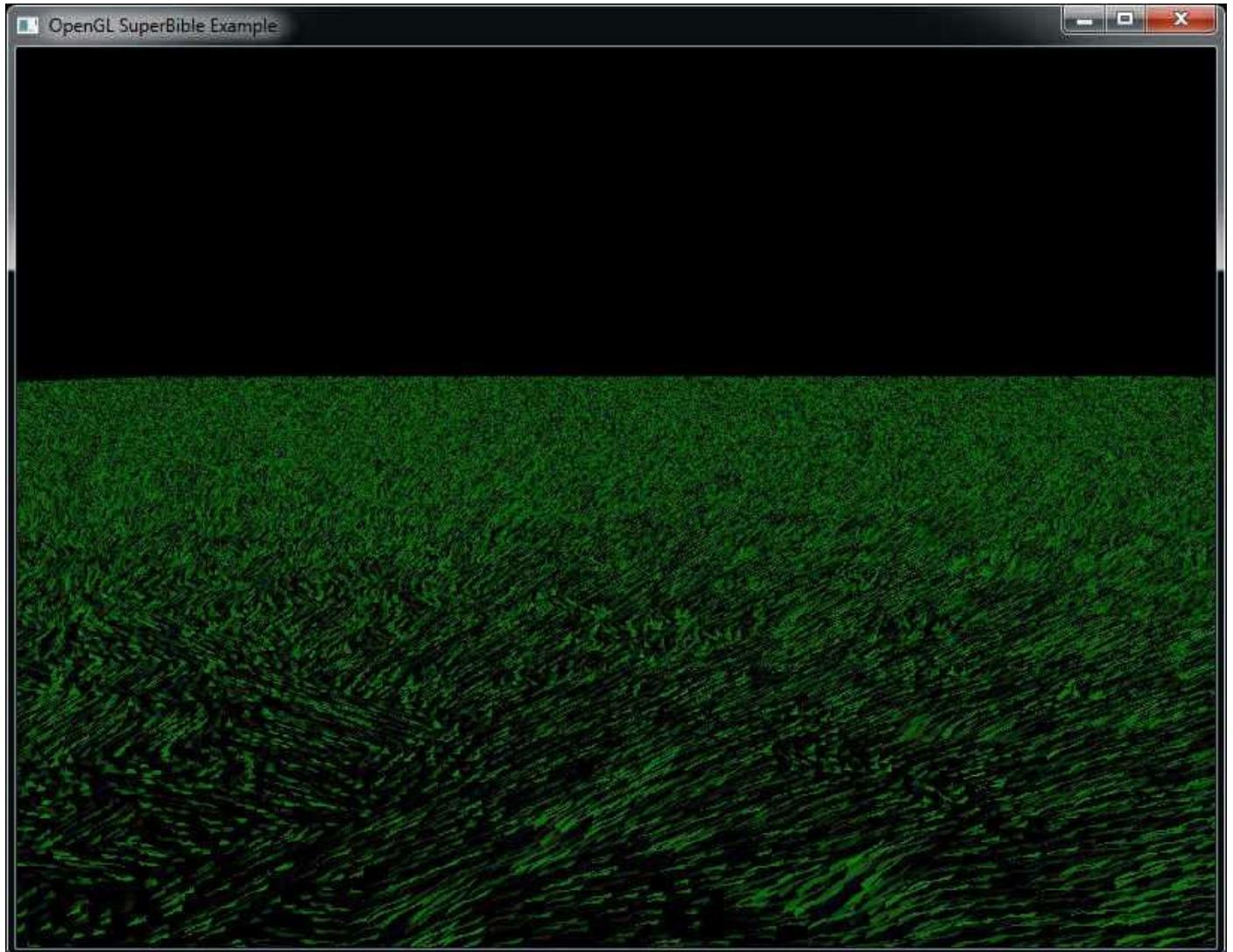


Figure 7.7: The final field of grass

Our final field has approximately 1 million blades of grass, evenly distributed, with application control over length, “flatness,” direction of bend or sway, and color. Remember, the only input to the shader that differentiates one blade of grass from another is `gl_InstanceID`, the total amount of geometry sent to OpenGL is six vertices, and the total amount of code required to draw all the grass in the field is a single call to **`glDrawArraysInstanced()`**.

The parameter texture can be read using linear texturing to provide smooth transitions between regions of grass and can be a fairly low resolution. If you want to make your grass wave in the wind or get trampled as hoards of armies march across it, you can

animate the texture by updating it every frame or two and uploading a new version of it before you render the grass. Also, because `gl_InstanceID` is used to generate random numbers, adding an offset to this variable before passing it to the random number generator allows a different but predetermined chunk of “random” grass to be generated with the same shader.

Getting Your Data Automatically

When you call one of the instanced drawing commands such as

`glDrawArraysInstanced()` or **`glDrawElementsInstanced()`**, the built-in variable `gl_InstanceID` will be available in your shaders to tell you which instance you’re working on, and it will be incremented by 1 for each new instance of the geometry that you’re rendering. It’s actually available even when you’re not using one of the instanced drawing functions—it’ll just be zero in those cases. This means that you can use the same shaders for instanced and non-instanced rendering.

You can use `gl_InstanceID` to index into arrays that are the same length as the number of instances that you’re rendering. For example, you can use it to look up texels in a texture or to index into a uniform array. Really, what you’re actually doing is treating the array as if it were an “instanced attribute.” That is, a new value of the attribute is read for each instance you’re rendering. OpenGL can feed this data to your shader automatically using a feature called instanced arrays. To use instanced arrays, declare an input to your shader in the usual way. The input attribute will have an index that you would use in calls to functions like **`glVertexAttribPointer()`**. Normally, the vertex attributes would be read per vertex and a new value would be fed to the shader. However, to make OpenGL read attributes from the arrays once per instance, you can call

[Click here to view code image](#)

```
void glVertexAttribDivisor(GLuint index,  
                           GLuint divisor);
```

Pass the index of the attribute to the function in `index` and set `divisor` to the number of instances you’d like to pass between each new value being read from the array. If `divisor` is zero, then the array becomes a regular vertex attribute array with a new value read per vertex. If `divisor` is non-zero, however, new data is read from the array once every `divisor` instance. For example, if you set `divisor` to 1, you’ll get a new value from the array for each instance. If you set `divisor` to 2, you’ll get a new value for every second instance, and so on. You can mix and match the divisors, setting different values for each attribute. An example of when you might want to use this functionality would be when you want to draw a set of objects with different colors. Consider the simple vertex shader in [Listing 7.7](#).

[Click here to view code image](#)

```
#version 450 core

in vec4 position;
in vec4 color;

out Fragment
{
    vec4 color;
} fragment;

uniform mat4 mvp;

void main(void)
{
    gl_Position = mvp * position;
    fragment.color = color;
}
```

Listing 7.7: Simple vertex shader with per-vertex color

Normally, the attribute `color` would be read once per vertex, so every vertex would end up having a different color. The application would have to supply an array of colors with as many elements as there were vertices in the model. Also, it wouldn't be possible for every instance of the object to have a different color because the shader doesn't know anything about instancing. We can make `color` an instanced array if we call

[Click here to view code image](#)

```
glVertexAttribDivisor(index_of_color, 1);
```

where `index_of_color` is the index of the slot to which the `color` attribute has been bound. Now, a new value of `color` will be fetched from the vertex array once per instance. Every vertex within any particular instance will receive the same value for `color`, and the result will be that each instance of the object will be rendered in a different color. The size of the vertex array holding the data for `color` needs to be only as long as the number of indices we want to render. If we increase the value of the divisor, new data will be read from the array with less and less frequency. If the divisor is 2, a new value of `color` will be presented every second instance; if the divisor is 3, `color` will be updated every third instance; and so on.

If we render geometry using this simple shader, each instance will be drawn on top of the others. We need to modify the position of each instance so that we can see each one. We can use another instanced array for this. [Listing 7.8](#) shows a simple modification to the vertex shader in [Listing 7.7](#).

[Click here to view code image](#)

```
#version 450 core

in vec4 position;
in vec4 instance_color;
in vec4 instance_position;

out Fragment
{
    vec4 color;
} fragment;

uniform mat4 mvp;

void main(void)
{
    gl_Position = mvp * (position + instance_position);
    fragment.color = instance_color;
}
```

Listing 7.8: Simple instanced vertex shader

Now, we have a per-instance position as well as a per-vertex position. We can add these together in the vertex shader before multiplying with the model–view–projection matrix. We can set the `instance_position` input attribute to an instanced array by calling

[Click here to view code image](#)

```
glVertexAttribDivisor(index_of_instance_position, 1);
```

Again, `index_of_instance_position` is the index of the location to which the `instance_position` attribute has been bound. Any type of input attribute can be made instanced using **glVertexAttribDivisor()**. This example is simple and uses only a translation (the value held in `instance_position`). A more advanced application could use matrix vertex attributes or pack some transformation matrices into uniforms and pass matrix weights in instanced arrays. For example, the application can use this approach to render an army of soldiers, each with a different pose, or a fleet of spaceships, all flying in different directions.

Now let's hook this simple shader up to a real program. First, we load our shaders in the usual way before linking the program. The vertex shader is shown in [Listing 7.8](#), the fragment shader simply passes the `color` input to its output, and the application code to hook all this up is shown in [Listing 7.9](#). In the code, we declare some data and load it into a buffer and attach it to a vertex array object. Some of the data is used as per-vertex positions, but the rest is used as per-instance colors and positions.

[Click here to view code image](#)

```
static const GLfloat square_vertices[] =
{
    -1.0f, -1.0f, 0.0f, 1.0f,
    1.0f, -1.0f, 0.0f, 1.0f,
    1.0f, 1.0f, 0.0f, 1.0f,
    -1.0f, 1.0f, 0.0f, 1.0f
};

static const GLfloat instance_colors[] =
{
    1.0f, 0.0f, 0.0f, 1.0f,
    0.0f, 1.0f, 0.0f, 1.0f,
    0.0f, 0.0f, 1.0f, 1.0f,
    1.0f, 1.0f, 0.0f, 1.0f
};

static const GLfloat instance_positions[] =
{
    -2.0f, -2.0f, 0.0f, 0.0f,
    2.0f, -2.0f, 0.0f, 0.0f,
    2.0f, 2.0f, 0.0f, 0.0f,
    -2.0f, 2.0f, 0.0f, 0.0f
};

GLuint offset = 0;
glGenVertexArrays(1, &square_vao);
glGenBuffers(1, &square_vbo);
 glBindVertexArray(square_vao);
 glBindBuffer(GL_ARRAY_BUFFER, square_vbo);
 glBufferData(GL_ARRAY_BUFFER,
             sizeof(square_vertices) +
             sizeof(instance_colors) +
             sizeof(instance_positions), NULL, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, offset,
                 sizeof(square_vertices),
                 square_vertices);
offset += sizeof(square_vertices);
glBufferSubData(GL_ARRAY_BUFFER, offset,
                 sizeof(instance_colors), instance_colors);
offset += sizeof(instance_colors);
glBufferSubData(GL_ARRAY_BUFFER, offset,
                 sizeof(instance_positions), instance_positions);
offset += sizeof(instance_positions);

glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0,
                     (GLvoid *)sizeof(square_vertices));
glVertexAttribPointer(2, 4, GL_FLOAT, GL_FALSE, 0,
                     (GLvoid *) (sizeof(square_vertices) +
                     sizeof(instance_colors)));

```

```
glEnableVertexAttribArray(0);  
glEnableVertexAttribArray(1);  
glEnableVertexAttribArray(2);
```

Listing 7.9: Getting ready for instanced rendering

Now all that remains is to set the vertex attribute divisors for the `instance_color` and `instance_position` attribute arrays

```
glVertexAttribDivisor(1, 1);  
glVertexAttribDivisor(2, 1);
```

We draw four instances of the geometry we put into our vertex buffer. Each instance consists of four vertices, each with its own position, which means that the same vertex in each instance has the same position. However, all of the vertices in a single instance see the same value of `instance_color` and `instance_position`, and a new value of each is presented to each instance. Our rendering loop looks like this:

[Click here to view code image](#)

```
static const GLfloat black[] = { 0.0f, 0.0f, 0.0f, 0.0f };  
glClearBufferfv(GL_COLOR, 0, black);  
  
glUseProgram(instancingProg);  
glBindVertexArray(square_vao);  
glDrawArraysInstanced(GL_TRIANGLE_FAN, 0, 4, 4);
```

What we get is shown in [Figure 7.8](#). In the figure, you can see that four rectangles have been rendered. Each is at a different position, and each has a different color. This technique can be extended to thousands or even millions of instances, and modern graphics hardware should be able to handle this without any issue.

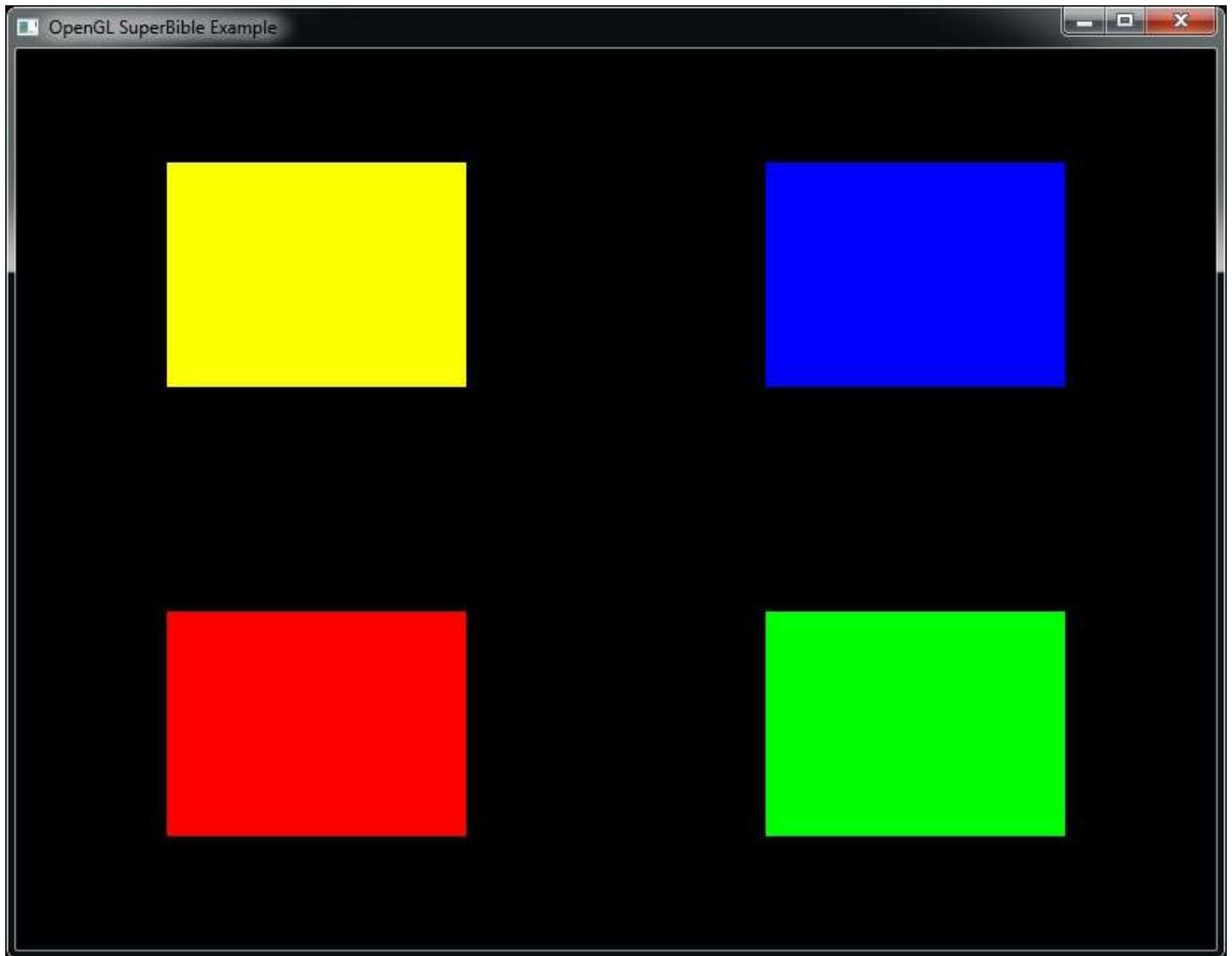


Figure 7.8: Result of instanced rendering

When you have instanced vertex attributes, you can use the `baseInstance` parameter to drawing commands such as **`glDrawArraysInstancedBaseInstance()`** to *offset* where in their respective buffers the data is read from. If you set this to zero (or call one of the functions that lacks this parameter), the data for the first instance comes from the start of the array. However, if you set it to a non-zero value, the index within the instanced array from which the data comes is offset by that value. This is very similar to the `baseVertex` parameter described earlier.

The actual formula for calculating the index from which attributes are fetched is

$$\left\lfloor \frac{\text{instance}}{\text{divisor}} \right\rfloor + \text{baseInstance}$$

We will use the `baseInstance` parameter in some of the following examples to provide offsets into instanced vertex arrays.

Indirect Draws

So far, we have covered only direct drawing commands. In these commands, we pass the parameters of the drawing command, such as the number of vertices or instances, directly to the function. However, there is a family of drawing commands that allow the parameters of each drawing command to be stored in a buffer object. As a consequence, at the time that your application calls the drawing command, it doesn't actually need to know those parameters—only the location in the buffer where the parameters are stored. This opens a few interesting possibilities:

- Your application can generate the parameters for a drawing command ahead of time, possibly even offline, and then load them into a buffer and send them to OpenGL when it's ready to draw.
- You can get OpenGL to generate the parameters at runtime by executing a shader that stores those parameters in a buffer object, effectively causing the GPU to generate its own work.
- You can use many threads on the CPU to generate the parameters for drawing commands. Because the commands are not sent directly to OpenGL but rather are stored in a buffer for later processing, there is no issue with using many threads for this operation.

There are four indirect drawing commands in OpenGL. The first two have direct equivalents: **glDrawArraysInstancedBaseInstance()** is similar to **glDrawArraysIndirect()** and **glDrawElementsInstancedBaseVertexBaseInstance()** is similar to **glDrawElementsIndirect()**. The prototypes of these indirect functions are

[Click here to view code image](#)

```
void glDrawArraysIndirect(GLenum mode,  
                           const void * indirect);
```

and

[Click here to view code image](#)

```
void glDrawElementsIndirect(GLenum mode,  
                           GLenum type,  
                           const void * indirect);
```

For both functions, `mode` is one of the primitive modes, such as `GL_TRIANGLES` or `GL_PATCHES`. For **glDrawElementsIndirect()**, `type` is the type of the indices to be used (just like the `type` parameter to **glDrawElements()**) and should be set to `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`. For both functions, `indirect` is interpreted as an offset into the buffer object bound to the `GL_DRAW_INDIRECT_BUFFER` target, but the contents of the buffer at this address differ depending on which function is being used. When expressed as a C-style structure

definition, for **glDrawArraysIndirect()**, the form of the data in the buffer is

```
typedef struct {
    GLuint vertexCount;
    GLuint instanceCount;
    GLuint firstVertex;
    GLuint baseInstance;
} DrawArraysIndirectCommand;
```

For **glDrawElementsIndirect()**, the form of the data in the buffer is

[Click here to view code image](#)

```
typedef struct {
    GLuint vertexCount;
    GLuint instanceCount;
    GLuint firstIndex;
    GLint baseVertex;
    GLuint baseInstance;
} DrawElementsIndirectCommand;
```

Calling **glDrawArraysIndirect()** will cause OpenGL to behave as if you had called **glDrawArraysInstancedBaseInstance()** with the mode you passed to **glDrawArraysIndirect()** but with the count, first, instancecount, and baseinstance parameters taken from the vertexCount, firstVertex, instanceCount, and baseInstance fields of the DrawArraysIndirectCommand structure stored in the buffer object at the offset given in the indirect parameter.

Likewise, calling **glDrawElementsIndirect()** will cause OpenGL to behave as if you had called **glDrawElementsInstancedBaseVertexBaseInstance()** with the mode and type parameters passed directly through, and with the count, instancecount, basevertex, and baseinstance parameters taken from the vertexCount, instanceCount, baseVertex, and baseInstance fields of the DrawElementsIndirectCommand structure in the buffer. However, one difference here is that the firstIndex parameter is in units of indices rather than bytes, so it is multiplied by the size of the index type to form the offset that would have been passed in the indices parameter to **glDrawElements()**.

As handy as it may seem to be able to do this, what makes this feature particularly powerful is the *multi* versions of these two functions:

[Click here to view code image](#)

```
void glMultiDrawArraysIndirect(GLenum mode,
                               const void * indirect,
                               GLsizei drawcount,
                               GLsizei stride);
```

and

[Click here to view code image](#)

```
void glMultiDrawElementsIndirect(GLenum mode,
                                  GLenum type,
                                  const void * indirect,
                                  GLsizei drawcount,
                                  GLsizei stride);
```

These two functions behave very similarly to **glDrawArraysIndirect()** and **glDrawElementsIndirect()**. However, you have probably noticed two additional parameters to each of the functions. Both functions essentially perform the same operation as their non-multi variants in a loop on an array of `DrawArraysIndirectCommand` or `DrawElementsIndirectCommand` structures. `drawcount` specifies the number of structures in the array, and `stride` specifies the number of bytes between the start of each of the structures in the buffer object. If `stride` is zero, then the arrays are considered to be tightly packed. Otherwise, you can have structures with additional data in between them, and OpenGL will skip over that data as it traverses the array.

The practical upper limit on the number of drawing commands you can batch together using these functions depends only on the amount of memory available to store them. The `drawcount` parameter can literally range to the billions, but with each command taking 16 or 20 bytes, 1 billion draw commands would consume 20 gigabytes of memory and probably take several seconds or even minutes to execute. However, it's perfectly reasonable to batch together tens of thousands of draw commands into a single buffer. Given this, you can either preload a buffer object with the parameters for many draw commands or generate a very large number of commands on the GPU. When you generate the parameters for your drawing commands using the GPU directly into the buffer object, you don't need to wait for those parameters to be ready before calling the indirect draw command that will consume them, and the parameters never make a round trip from the GPU to your application and back.

[Listing 7.10](#) shows a simple example of how **glMultiDrawArraysIndirect()** might be used.

[Click here to view code image](#)

```
typedef struct {
    GLuint vertexCount;
    GLuint instanceCount;
    GLuint firstVertex;
    GLuint baseInstance;
} DrawArraysIndirectCommand;

DrawArraysIndirectCommand draws[] = {
```

```

        {
            42,      // Vertex count
            1,       // Instance count
            0,       // First vertex
            0        // Base instance
        },
        {
            192,
            1,
            327,
            0,
        },
        {
            99,
            1,
            901,
            0
        }
    };
}

// Put 'draws[]' into a buffer object
GLuint buffer;

glGenBuffers(1, &buffer);
 glBindBuffer(GL_DRAW_INDIRECT_BUFFER, buffer);
 glBindBuffer(GL_DRAW_INDIRECT_BUFFER, sizeof(draws),
             draws, GL_STATIC_DRAW);

// This will produce 3 draws (the number of elements in draws[]), each
// drawing disjoint pieces of the bound vertex arrays
 glMultiDrawArraysIndirect(GL_TRIANGLES,
                           NULL,
                           sizeof(draws) / sizeof(draws[0]),
                           0);

```

Listing 7.10: Example use of an indirect draw command

Simply batching together three drawing commands isn't really that interesting, though. To show the real power of the indirect draw command, we'll draw an asteroid field. This field will consist of 30,000 individual asteroids. First, we will take advantage of the `sb7::object` class's ability to store multiple meshes within a single file. When such a file is loaded from disk, all of the vertex data is loaded into a single buffer object and associated with a single vertex array object. Each of the sub-objects has a starting vertex and a count of the number of vertices used to describe it. We can retrieve these from the object loader by calling `sb7::object::get_sub_object_info()`. The total number of sub-objects in the `.sbm` file is made available through the `sb7::object::get_sub_object_count()` function. Therefore, we can construct an indirect draw buffer for our asteroid field using the code shown in [Listing 7.11](#).

[Click here to view code image](#)

```
object.load("media/objects/asteroids.sbm");

 glGenBuffers(1, &indirect_draw_buffer);
 glBindBuffer(GL_DRAW_INDIRECT_BUFFER, indirect_draw_buffer);
 glBufferData(GL_DRAW_INDIRECT_BUFFER,
              NUM_DRAWS * sizeof(DrawArraysIndirectCommand),
              NULL,
              GL_STATIC_DRAW);

DrawArraysIndirectCommand * cmd = (DrawArraysIndirectCommand *)
    glMapBufferRange(GL_DRAW_INDIRECT_BUFFER,
                     0,
                     NUM_DRAWS * sizeof(DrawArraysIndirectCommand),
                     GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT);

for (i = 0; i < NUM_DRAWS; i++)
{
    object.get_sub_object_info(i % object.get_sub_object_count(),
                               cmd[i].first,
                               cmd[i].count);
    cmd[i].primCount = 1;
    cmd[i].baseInstance = i;
}

glUnmapBuffer(GL_DRAW_INDIRECT_BUFFER);
```

Listing 7.11: Setting up the indirect draw buffer for asteroids

Next, we need a way to communicate which asteroid we’re drawing to the vertex shader. There is no direct way to get this information from the indirect draw command into the shader. However, we can take advantage of the fact that all drawing commands are actually instanced drawing commands—commands that draw just a single copy of the object can be considered to draw a single instance. Therefore, we can set up an instanced vertex attribute, set the `baseInstance` field of the indirect drawing command structure to the index within that attribute’s array of the data that we wish to pass to the vertex shader, and then use that data for whatever we wish. In [Listing 7.11](#), notice that we set the `baseInstance` field of each structure to the loop counter.

Next, we need to set up a corresponding input to our vertex shader. The input declaration for our asteroid field renderer is shown in [Listing 7.12](#).

[Click here to view code image](#)

```
#version 450 core

layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;
```

```
layout (location = 10) in uint draw_id;
```

Listing 7.12: Vertex shader inputs for asteroids

As usual, we have a position and normal input. However, we've also used an attribute at location 10, `draw_id`, to store our draw index. This attribute will be instanced and associated with a buffer that simply contains an identity mapping. We'll use the `sb7::object` loader's functions to access and modify its vertex array object to inject our extra vertex attribute. The code to do this is shown in [Listing 7.13](#).

[Click here to view code image](#)

```
glBindVertexArray(object.get_vao());

glGenBuffers(1, &draw_index_buffer);
glBindBuffer(GL_ARRAY_BUFFER, draw_index_buffer);
glBufferData(GL_ARRAY_BUFFER,
             NUM_DRAWNS * sizeof(GLuint),
             NULL,
             GL_STATIC_DRAW);

GLuint * draw_index =
    (GLuint *)glMapBufferRange(GL_ARRAY_BUFFER,
                               0,
                               NUM_DRAWNS * sizeof(GLuint),
                               GL_MAP_WRITE_BIT |
                               GL_MAP_INVALIDATE_BUFFER_BIT);
for (i = 0; i < NUM_DRAWNS; i++)
{
    draw_index[i] = i;
}

glUnmapBuffer(GL_ARRAY_BUFFER);

glVertexAttribIPointer(10, 1, GL_UNSIGNED_INT, 0, NULL);
glVertexAttribDivisor(10, 1);
 glEnableVertexAttribArray(10);
```

Listing 7.13: Per-indirect draw attribute setup

Once we've set up our `draw_id` vertex shader input, we can use it to make each mesh unique. Without this step, each asteroid would be a simple rock placed at the origin. In this example, we will directly create an orientation and translation matrix in the vertex shader from `draw_id`. The complete vertex shader is shown in [Listing 7.14](#).

[Click here to view code image](#)

```
#version 450 core

layout (location = 0) in vec4 position;
```

```

layout (location = 1) in vec3 normal;
layout (location = 10) in uint draw_id;

out VS_OUT
{
    vec3 normal;
    vec4 color;
} vs_out;

uniform float time = 0.0;

uniform mat4 view_matrix;
uniform mat4 proj_matrix;
uniform mat4 viewproj_matrix;

const vec4 color0 = vec4(0.29, 0.21, 0.18, 1.0);
const vec4 color1 = vec4(0.58, 0.55, 0.51, 1.0);

void main(void)
{
    mat4 m1;
    mat4 m2;
    mat4 m;
    float t = time * 0.1;
    float f = float(draw_id) / 30.0;

    float st = sin(t * 0.5 + f * 5.0);
    float ct = cos(t * 0.5 + f * 5.0);

    float j = fract(f);
    float d = cos(j * 3.14159);

    // Rotate around Y
    m[0] = vec4(ct, 0.0, st, 0.0);
    m[1] = vec4(0.0, 1.0, 0.0, 0.0);
    m[2] = vec4(-st, 0.0, ct, 0.0);
    m[3] = vec4(0.0, 0.0, 0.0, 1.0);

    // Translate in the XZ plane
    m1[0] = vec4(1.0, 0.0, 0.0, 0.0);
    m1[1] = vec4(0.0, 1.0, 0.0, 0.0);
    m1[2] = vec4(0.0, 0.0, 1.0, 0.0);
    m1[3] = vec4(260.0 + 30.0 * d, 5.0 * sin(f * 123.123), 0.0, 1.0);

    m = m * m1;

    // Rotate around X
    st = sin(t * 2.1 * (600.0 + f) * 0.01);
    ct = cos(t * 2.1 * (600.0 + f) * 0.01);
}

```

```

m1[0] = vec4(ct, st, 0.0, 0.0);
m1[1] = vec4(-st, ct, 0.0, 0.0);
m1[2] = vec4(0.0, 0.0, 1.0, 0.0);
m1[3] = vec4(0.0, 0.0, 0.0, 1.0);

m = m * m1;

// Rotate around Z
st = sin(t * 1.7 * (700.0 + f) * 0.01);
ct = cos(t * 1.7 * (700.0 + f) * 0.01);

m1[0] = vec4(1.0, 0.0, 0.0, 0.0);
m1[1] = vec4(0.0, ct, st, 0.0);
m1[2] = vec4(0.0, -st, ct, 0.0);
m1[3] = vec4(0.0, 0.0, 0.0, 1.0);

m = m * m1;

// Non-uniform scale
float f1 = 0.65 + cos(f * 1.1) * 0.2;
float f2 = 0.65 + cos(f * 1.1) * 0.2;
float f3 = 0.65 + cos(f * 1.3) * 0.2;

m1[0] = vec4(f1, 0.0, 0.0, 0.0);
m1[1] = vec4(0.0, f2, 0.0, 0.0);
m1[2] = vec4(0.0, 0.0, f3, 0.0);
m1[3] = vec4(0.0, 0.0, 0.0, 1.0);

m = m * m1;

gl_Position = viewproj_matrix * m * position;
vs_out.normal = mat3(view_matrix * m) * normal;
vs_out.color = mix(color0, color1, fract(j * 313.431));
}

```

Listing 7.14: Asteroid field vertex shader

In the vertex shader shown in [Listing 7.14](#), we calculate the orientation, position, and color of the asteroid directly from `draw_id`. First, we convert `draw_id` to floating point and scale it. Next, we calculate a number of translation, scaling, and rotation matrices based on its value and the value of the `time` uniform. These matrices are concatenated to form a model matrix, `m`. The position is transformed first by the model matrix and then by the view-projection matrix. The vertex's normal is also transformed by the model and view matrices. Finally, an output color is computed for the vertex by interpolating between two colors (one is a chocolate brown, the other a sandy gray) to give the asteroid its final color. A simple lighting scheme is used in the fragment shader to give the asteroids a sense of depth.

The rendering loop for this application is extremely simple. First, we set up our view

and projection matrices, and then we render all of the models with a single call to **glMultiDrawArraysIndirect()**. The drawing code is shown in [Listing 7.15](#).

[Click here to view code image](#)

```
glBindVertexArray(object.get_vao());  
  
if (mode == MODE_MULTIDRAW)  
{  
    glMultiDrawArraysIndirect(GL_TRIANGLES, NULL, NUM_DRAWNS, 0);  
}  
else if (mode == MODE_SEPARATE_DRAWNS)  
{  
    for (j = 0; j < NUM_DRAWNS; j++)  
    {  
        GLuint first, count;  
        object.get_sub_object_info(j % object.get_sub_object_count(),  
                                    first, count);  
        glDrawArraysInstancedBaseInstance(GL_TRIANGLES,  
                                         first,  
                                         count,  
                                         1, j);  
    }  
}
```

Listing 7.15: Drawing asteroids

As you can see from [Listing 7.15](#), we first bind the object's vertex array object by calling `object.get_vao()` and passing the result to **glBindVertexArray()**. When mode is `MODE_MULTIDRAW`, the entire scene is drawn with a single call to **glMultiDrawArraysIndirect()**. However, if mode is `MODE_SEPARATE_DRAWNS`, we loop over all of the loaded sub-objects and draw each separately by passing the same parameters that are loaded into the indirect draw buffer directly to a call to **glDrawArraysInstancedBaseInstance()**. Depending on your OpenGL implementation, the separate draw mode could be substantially slower. The resulting output is shown in [Figure 7.9](#).

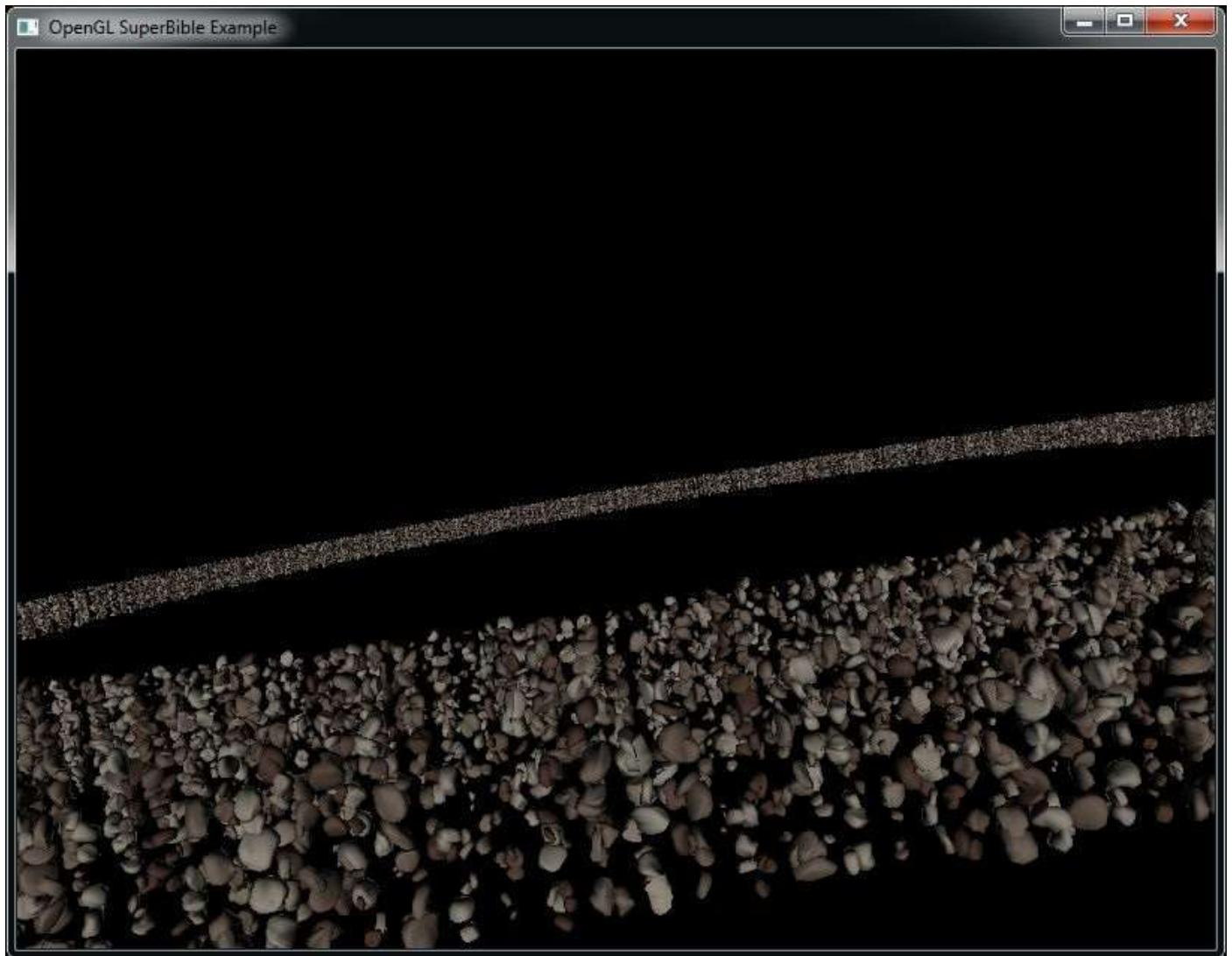


Figure 7.9: Result of asteroid rendering program

In our example, using a typical consumer graphics card, we can achieve 60 frames per second with 30,000 unique³ models, which is equivalent to 1.8 million drawing commands every second. Each mesh has approximately 500 vertices, which means that we're rendering almost 1 billion vertices per second, and our bottleneck is almost certainly not the rate at which we are submitting drawing commands.

³. The asteroids in this example are not truly unique—they are selected from a large batch of unique rock models, and then a different scale and color are applied to each one. The chance of finding two rocks of the same shape, at the same scale, and with the same color is vanishingly small.

With clever use of the `draw_id` input (or other instanced vertex attributes), more interesting geometry with more complex variation could be rendered. For example, we could use texture mapping to apply surface detail, storing a number of different surfaces in an array texture and selecting a layer using `draw_id`. Although in this example we compute the transformation matrices directly in our vertex shader using `draw_id`, it's also possible to put large arrays of matrices in a uniform block or shader storage buffer and use `draw_id` to index into it, giving better control over the placement of objects

by the application. There's also no reason why the content of the indirect draw buffer needs to be static. In fact, we can generate its content directly in the graphics process by using various techniques to achieve truly dynamic rendering without application intervention.