OpenGL ES 3.0 Cookbook

"Write once, use anywhere" is truly the power behind OpenGL ES and has made it an embedded industry standard. The library provides cutting-edge, easy-to-use features to build a wide range of applications in the gaming, simulation, augmented-reality, image-processing, and geospatial domains.

The book starts by providing you with all the necessary OpenGL ES 3.0 setup guidelines on iOS and Android platforms. You’ll go on to master the fundamentals of modern 3D graphics, such as drawing APIs, transformations, buffer objects, the model-view-project analogy, and much more. The book goes on to deal with advanced topics and offers a wide range of recipes on the light shading, real-time rendering techniques with static and procedure textures to create stunning visualizations and runtime effects.

What this book will do for you...

- Learn the essentials and exciting new features of OpenGL ES 3.0
- Discover the physics behind fundamentals of light and material concepts with easy-to-follow examples
- Scratch the fragments with procedural shaders and learn how they work
- Master the basics of texturing, use compressed textures, and learn numerous mapping techniques
- Manage complex scenes with scene graphs and learn post-processing and image-processing techniques
- Build your font engine with multilingual support
- Master the working of recognized anti-aliasing techniques and implement FXAA and adaptive anti-aliasing

Inside the Cookbook...

- Straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions to solve problems efficiently
- Clear explanations for what you did
- Apply solutions to other situations

Over 90 ready-to-serve, real-time rendering recipes on Android and iOS platforms using OpenGL ES 3.0 and GL shading language 3.0 to solve day-to-day modern 3D graphics challenges
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'New Features of OpenGL ES 3.0'
- A synopsis of the book’s content
- More information on OpenGL ES 3.0 Cookbook

About the Author

Parminder Singh is a senior 3D graphics engineer at Continental Automotive, Singapore. He also works as a part-time freelancer. In 2006, Parminder obtained his CSE degree from Punjab Technical University.

He is a strong believer of design simplicity. In his opinion, this is a key factor that plays an important role in building scalable and manageable products. With this philosophy and as a passionate 3D architect, he has worked in the fields of network simulations, geomodeling, navigation, automotive, and infotainment systems. His research interests include GPU-based real-time rendering, geospatial terrain rendering, screen-spaced techniques, real-time dynamic shadows, scientific visualization, scene graphs, and anti-aliasing techniques.

He is an OpenGL ES trainer and a member of the Khronos Group. Parminder loves to take up challenges related to real-time rendering. His current research and work includes futuristic implementation for next-generation graphics in the automobile domain in order to create stunning data, user interface, and visualization effects (merging 2D and 3D concepts). His hobbies include cooking, traveling, sharing knowledge, and exploring the possibilities of applied physics and mathematics.

Feel free to reach Parminder at https://www.linkedin.com/in/parmindersingh18.
OpenGL ES 3.0 Cookbook

OpenGL ES 3.0 is a royalty free, hardware-accelerated graphics rendering application programming interface for embedded systems. It is used for visualizing 2D and 3D graphics with the modern programmable graphics pipeline. "Write once, use anywhere" is truly the power behind OpenGL ES, which has made it an embedded industry standard. OpenGL ES 3.0 is a cross-platform graphics library and opens its gates to many other cutting-edge technologies, such as parallel processing libraries (OpenCL) and digital image processing (OpenCV) that works in conjunction with many other open community solutions.

The main strength of this book is that it covers development of real-time rendering graphics and visualization development using OpenGL ES 3.0 from scratch. It lets the user know how to define the framework for the OpenGL ES 3.0 application. These are some of the techniques that make this book entirely different from other books available on the market. The idea behind this book is to give you an in-depth knowledge of this new version of graphics API and use it to implement computer graphics fundamentals and advance concepts from scratch using Android and iOS as embedded platforms. This book covers a lot of ground, from basic concepts of modern 3D graphics to advanced real-time rendering techniques using OpenGL ES 3.0.

What This Book Covers

Chapter 1, OpenGL ES 3.0 on Android/iOS, takes you through the process of how to develop the Android and iOS OpenGL ES 3.0 application. This chapter shows you how to load and compile a shader program apart from the process to program shaders in GL shading language 3.0.

Chapter 2, OpenGL ES 3.0 Essentials, provides you with a detailed description of the basic concepts that are required to understand 3D graphics and implement them using OpenGL ES 3.0. We will build prototypes using the GLPI framework and implement touch events and scene with model, view, and projection analogy.

Chapter 3, New Features of OpenGL ES 3.0, helps you understand the various new features introduced in OpenGL ES 3.0 and GL shading language 3.0. This chapter tells you how to manage variable attributes with qualifiers and render multiple objects with geometry instancing and primitives with primitive restart.

Chapter 4, Working with Meshes, teaches you how to create simple meshes using Blender, which is an open source 3D modeling tool. In addition, this chapter covers various aspects of the 3D mesh model that will be helpful to render them in 3D graphics. This chapter also teaches how to use the created mesh model in OpenGL ES 3.0 applications.
**Chapter 5, Light and Materials**, introduces the concepts of light and materials in 3D graphics. It also covers some important common illumination techniques, such as Phong and Gouraud shading, which will help you implement realistic looking lighting models in computer graphics.

**Chapter 6, Working with Shaders**, gives you an in-depth understanding on the shaders programming technique. It discusses various techniques that can be implemented using the vertex and fragment shader, revealing their capabilities. This chapter helps you play with fragments by programing them using procedural shaders.

**Chapter 7, Textures and Mapping Techniques**, sheds some light on textures, which is a very interesting part of the 3D computer graphics study. Texturing is a technique in which the surface of a 3D mesh model is painted with static images. This chapter is all about image texturing and explains its various applications in the field of 3D computer graphics. This chapter covers ample techniques on mapping, such as environment, bump, displacement mapping, and so on.

**Chapter 8, Font Rendering**, provides a detailed description on how to build the font engine and render different languages with Harfbuzz and text on Head Up Display (HUD).

**Chapter 9, Postscreen Processing and Image Effects**, unfolds the endless possibilities of scene-based effects and image-based effects, which are widely used in the field of data visualization and after effects. This includes applications such as edge detection, image blurring, real-time glow, emboss effect, and so on.

**Chapter 10, Scene Management with Scene Graphs**, introduces a scene graph paradigm that allows you to program and manage complex scenes efficiently. This chapter will help you create a small architecture that allows you to manage multiple scenes. Each scene consists of multiple lights, cameras, and models.

**Chapter 11, Anti-aliasing Techniques**, tells you how to implement fast approximate anti-aliasing (FXAA), adaptive anti-aliasing, and anti-aliased circle geometry.

**Chapter 12, Real-time Shadows and Particle System**, shows you how to implement shadows using shadow mapping and improve it with percentile closer filtering and variance shadow mapping technique. It also discusses the basics of particle rendering. This chapter teaches you the transform feedback with sync objects and fence, which help you implement high-performance, GPU-driven, and real-time graphics applications.

**Appendix, Supplementary Information on OpenGL ES 3.0**, covers all the basic requirements that we need to develop OpenGL ES 3.0 applications on the iOS and Android platforms. This chapter teaches you two ways of Android application development with Android ADT and Android Studio. This also provides you a simple overview of OpenGL ES 3.0 architectures. This overview also helps you understand the technical jargon of various computer graphics terminology.
Chapter 3

New Features of OpenGL ES 3.0

In this chapter, we will cover the following recipes:

- Managing variable attributes with qualifiers
- Grouping uniforms and creating buffer objects
- Managing VBO with Vertex Array Objects
- Reading and writing buffer objects with mapping
- Rendering multiple objects with geometry instancing
- Rendering multiple primitives with primitive restart

Introduction

OpenGL ES 3.0 was publicly released in August 2012. It brings the mobile 3D graphics to the next level. This release was focused to provide 3D-enriched features and enhanced the portability across diverse mobiles, embedded operating systems, and platforms. OpenGL ES 3.0 is fully backward compatible with OpenGL ES 2.0. This enables the applications to grow the graphics capabilities and visual features incrementally. OpenGL ES 3.0 also introduces a new version of GL Shading Language (GLSL) 3.0. The GLSL is used for programing shaders. The new shading language has also extended the capabilities in many directions, which you will learn in the next section.

This chapter will be helpful in understanding the new features introduced in OpenGL ES 3.0 and GL shading language 3.0. This book uses OpenGL ES 3.0 in conjunction with GLSL 3.0 for all its recipes.
The new features of OpenGL ES 3.0 can be broadly divided into the following five categories:

- **Geometry**: These features focus on the vertex attributes specifications, such as data storage, data transfer, attribute states, primitive assembly, and so on. They are explained as follows:
  - **Transform feedback**: This feature allows us to capture the vertex shader output to provide feedback to the GPU for next frame rendering. This way, it avoids CPU intervention and makes the rendering efficient.
  - **Occlusion query**: This enables fast hardware testing to check whether a pixel is going to appear on screen or whether it is occluded by another pixel. This kind of check is helpful in deciding whether to skip certain operations such as geometry processing because it's occluded.
  - **Geometry instancing**: This allows efficient rendering of an object multiple times without calling multiple render API's. This is very helpful in situations such as crowd simulation, trees rendering, and so on.
  - **Primitive restart**: This new feature allows us to render multiple disconnected primitives using a single drawing API call. The index array is used to pack multiple primitives (of the same type) in a single bundle. This array contains multiple disconnected primitives with a special marker that helps the GPU render disconnected primitives in one go.

- **Textures**: There are many new features added into OpenGL ES 3.0 for textures. The features are described here:
  - **Depth textures and shadow comparison**: Depth textures allow the storing of the depth buffer information into a texture. This is helpful in rendering shadows using the **percentile closest filtering (PCF)** technique in which depth information is explicitly stored from the depth buffer to a texture using the render-to-texture technique. Later, this information is used to test incoming fragments for whether they are a part of shadow or not. OpenGL ES 3.0 allows this comparison test implicitly.
  - **Seamless cube maps**: The cubemap rendering is improved to remove artifacts from the boundary edges of the images. Now, the filtering techniques take adjacent faces texture data into account to produce seamless texture boundaries on the face edges. You can refer to the *Implementing Skybox with seamless cube mapping* recipe in *Chapter 7, Textures and Mapping Techniques*. 


ETC2/EAC texture-compression formats: Before OpenGL ES 3.0, there was no standard compression format officially supported by OpenGL ES. Developers relied on the specific compression formats provided by different vendors, such as PVRTC by Imagination Technologies, Ericsson Texture Compression (ETC) by Sony Ericsson, ATC by Qualcomm, and so on. Now, ETC2 and EAC texture-compression formats are integrally supported in OpenGL ES 3.0. Refer to the Efficient rendering with ETC2 compressed texture recipe in Chapter 7, Textures and Mapping Techniques.

Nonpower of two (NPOT) texture: Now, textures with pixel dimensions of the nonpower of two texture are supported with full wrap mode and mipmapping. In earlier specifications of OpenGL ES, the textures had to be in the form of power of two (POT) dimensions. Therefore, external imaging tools were required to convert NPOT to POT format.

Texture swizzles: The GLSL provides a level of abstraction in accessing the components of texture, R, G, B, and A, irrespective of the order in which they are stored physically.

Increased 2D texture dimension: The dimension of 2D texture in OpenGL ES 3.0 is 2048, which is much more compared to OpenGL ES 2.0.

3D texture: OpenGL ES 3.0 supports 3D texture targets. 3D textures are widely used in medical imaging.

Arrays of 2D texture: This new features allows us to store multiple 2D textures in the form of an array. This is useful for animation purpose. Prior to this, texture sprites were used.

Shaders: These are the special small programs that are used in modern computer graphics programming to control geometry and pixel color shading. The features on shaders are as follows:

Program binaries: The vertex and fragment shaders are compiled and stored in a binary format. This binary format needs to be linked to the program at run time in OpenGL ES 2.0. OpenGL ES 3.0 allows an optimization by storing this binary into an offline binary format that does not require linking at run time. This optimization helps load the application faster by avoiding runtime linking.

Flat/smooth interpolators: In OpenGL ES 2.0, all the interpolators perform linear interpolation across the primitives. With the help of GLSL 3.0, in OpenGL ES 3.0, the interpolation can be explicitly declared to have flat and smooth shading.
Buffer objects: These allow us to store vertex data on the GPU memory. The new features have extended the capabilities of buffer objects to make them more efficient. Here are the new features:

- **Uniform blocks**: This allows to group related uniform values into a single manageable group. This increases the readability of the shader program.
- **Layout qualifiers**: The attributes defined in the vertex and fragment shaders can be directly bound to the user-defined locations. This way, the on-fly binding API calls are not required.
- **Vertex Array Objects**: This feature provides an efficient way to bind vertex array data and respective attributes. **Vertex Array Objects (VAO)** are used to encapsulate the VBO. When a VAO API is called, it efficiently switches the states stored in VBO without calling several APIs. This reduces the overhead in the switching of vertex array states.
- **Uniform buffer object**: This feature stores the uniform block in an efficient way as a buffer object. This uniform block object can be bound on fly time. This gives an opportunity to share the uniform data among multiple programs at once. Additionally, it allows us to set multiple uniform variables in one go.
- **Subrange buffer mapping**: Unlike mapping the complete buffer from the GPU to the CPU side, this mechanism provides an efficient way to access a range of memory contents from the GPU memory space. Sometimes, the intention is to update only a small section of the buffer. Therefore, mapping the complete buffer is inefficient. In such situations, subrange buffer mapping reduces the time of marshaling from GPU to CPU to GPU.
- **Buffer object copies**: This mechanism transfers the data of one buffer object to the other one without intervening the CPU.
- **Sync object**: This provides a synchronized mechanism between the applications and GPU. In this way, the application can assure completion of OpenGL ES operations on the GPU side.
- **Fencing**: This feature informs the GPU to wait for queuing up new OpenGL ES operations until the old operations are completely executed on the GPU.

Framebuffer: The new features also include enhancements related to off-screen rendering for the framebuffer. Here are the new features:

- **Multiple render target (MRT)**: This feature allows us to perform off-screen rendering simultaneously to several color buffers or textures at the same time. These textures can be used as input to other shaders or can be used on 3D models. MRTs are most commonly used to achieve deferred shading.
Multisample render buffer: This feature enables the application to perform off-screen framebuffer rendering with multisample anti-aliasing. This improves the visual quality of the generated image and reduces the jagged-line effect that appears in the lines or sharp geometry edges drawn diagonally to the screen.

This chapter will focus on the new features of geometry and buffer objects. As we progress with the upcoming chapters, we will also introduce the new features of shaders, textures, and framebuffers.


### Managing variable attributes with qualifiers

GLSL 3.0 has introduced two new qualifiers: storage and layout. Let's take a look at them in detail:

- **Storage qualifier**: This is a special keyword that specifies the storage or the behavior of a global or local variable. It is used in shader programming. It enables the communication bridge between the application and shaders. It is also used to share information from one shader stage to another. For example, a 3D light illumination technique requires an object's geometry information in order to create realistic light shading. This geometry information is calculated in the vertex shader and passed to the fragment shader, where this input is used to color the fragments of the geometric primitives.

  There are six types of storage qualifiers available in GLSL 3.0. They are described in the following table:

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>This is the value of variable does not alter compile time.</td>
</tr>
<tr>
<td>in</td>
<td>This is the copied input variable from the previous stage, which is linked to the current shader. If specified in a function argument, this is an input variable.</td>
</tr>
<tr>
<td>centroid in</td>
<td>This is the input type variable linked to the centroid interpolator.</td>
</tr>
<tr>
<td>out</td>
<td>This is the copied input variable from the previous stage, which is linked to the current shader. If specified in a function argument, this is an output variable.</td>
</tr>
<tr>
<td>centroid out</td>
<td>This is the output type variable that is linked to the centroid interpolator.</td>
</tr>
<tr>
<td>uniform</td>
<td>This is the value of the variables does not change across the primitives during the processing. The uniforms are shared across the shaders.</td>
</tr>
</tbody>
</table>
Layout qualifier: This influences the properties of a variable, such as storage, location, memory alignment, and so on. This qualifier is widely used in declaring the location of the variable(s) in shaders. Each variable or generic attribute declared in the shader is stored in an allocated memory location on the GPU. This memory location is used to store data in the variables as a result of runtime calculation or input data from the previous stage of the shader. Unlike C/C++ pointers, the shading language uses a location ID to access the variable. A location is an ID (numeric value(s)) of a variable that is used to connect the variable present in the shading language to the application program.

Getting ready

The next table specifies the syntax for the storage and layout qualifiers. The storage qualifiers are mentioned before the data type of the variable. The most commonly used qualifiers are in and out. These storage qualifiers tell us whether the vertex attribute is an incoming or outgoing variable.

The layout qualifier assigns an ID or location to the vertex attribute so that run the binding and querying of the location can be avoided. The layout qualifier is always mentioned before the storage qualifier.

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>(storage qualifier) [Data type] [Variable Name]</td>
</tr>
<tr>
<td>Layout</td>
<td>layout (qualifier1, qualifier2 = value, . . .) [Storage qualifier]</td>
</tr>
</tbody>
</table>

How to do it...

The variables in a shader are abstracted in the form of location IDs. Each variable or generic attribute is recognized using its location ID and used to bind the data in the OpenGL ES program. These location IDs/indexes can be defined using the location keyword in the layout qualifier.

In our first recipe, we will demonstrate the use of storage and layout qualifiers:

1. Create a vertex shader LayoutVertex.glsl, as shown here:

```glsl
#version 300 es
layout(location = 0) in vec4 VertexPosition;
layout(location = 1) in vec4 VertexColor;
out vec4 Color;
uniform mat4 MODELVIEWPROJECTIONMATRIX;

// Function with two input and one output storage qualifier
```
void calculatePosition(in mat4 MVP, in vec4 vp, out vec4 position){
    position = MVP * vp;
}

void main()
{
    vec4 position;
    calculatePosition(MODELVIEWPROJECTIONMATRIX, VertexPosition, position);
    gl_Position = position;
    Color = VertexColor;
}

2. Create the fragment shader LayoutFragment.glsl and modify it, as shown here:

```glsl
#version 300 es
precision mediump float;
in vec4 Color; //in variable receive from shader
float blendFactor = 0.8;
layout(location = 0) out vec4 outColor;
// Function with input argument and output as return type
vec4 addBlend( in vec4 colorOpaque )
{
    return vec4(colorOpaque.x, colorOpaque.y,
                colorOpaque.z, blendFactor);
}

void main() {
    outColor = addBlend( Color );
}
```

3. Reuse the Efficient rendering with Vertex Buffer Object recipe Chapter 2, OpenGL ES 3.0 Essentials and define the location index according to your choice in the application program, Cube.cpp. Make sure that the same index is specified in the shader program:

```cpp
#define VERTEX_LOCATION 0
#define COLOR_LOCATION 1
```

4. Create the VBO and IBO in the constructor and enable the following attributes like:

```cpp
glGenBuffers(1, &vId); // Create VBO and bind data
glGenBuffers(1, &iId); // Create IBO and bind data

// Enable the attribute locations
glEnableVertexAttribArray(VERTEX_LOCATION);
glEnableVertexAttribArray(COLOR_LOCATION);
```
5. Attach the VBO geometry data to the location ID. This will be used to send data from application to the GPU shader processor. Clearly, with the layout qualifier, the location query (glGetAttribLocation) for the vertex attribute can be avoided:

```cpp
void Cube::RenderCube() {
    ......
    glBindBuffer( GL_ARRAY_BUFFER, vId );
    glVertexAttribPointer(VERTEX_LOCATION, 3,
        GL_FLOAT, GL_FALSE, 0, (void*)0);
    glVertexAttribPointer(COLOR_LOCATION, 3, GL_FLOAT,
        GL_FALSE, 0, (void*)size);
    glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, iId );
    glDrawElements(GL_TRIANGLES, 36,
        GL_UNSIGNED_SHORT, (void*)0);
    ......
}
```

**How it works...**

The OpenGL ES program defines two index ID's in Cube.cpp, VERTEX_LOCATION and COLOR_LOCATION for vertex and color data, respectively. These indices will be used to define the attribute location in the shader program. The programmer must ensure that the layout location ID used in the shader program for the attribute must be same as the one used in the OpenGL ES program. This can be achieved by declaring the variable attributes using the layout qualifier. Prefixing the layout keyword in conjunction with the location qualifier allows the user-defined locations to attach with attribute variables. If some attribute variables are not specified by user-defined location indices, then the compiler would automatically generate and assign them.

In the shader program, VertexPosition and VertexColor are assigned to the same location indices, 0 and 1, respectively, what was defined in the OpenGL ES program. These two variable declarations are of the vec4 type, which is prefixed with the storage qualifier in. This gives information that these two variables are input to the vertex shader from the OpenGL ES program. The geometry data (vertex and color) is sent to the vertex shader by attaching the data to the location indexes of VertexPosition and VertexColor using the glVertexAttribPointer API in the RenderCube function. It should be noted that the generic attribute variables must be enabled before they are attached using the glEnableVertexAttribArray API. This recipe enables them in the Cube constructor.
When the vertex shader receives an input data for vertices in `VertexPosition` and transformation coordinates in the uniform `MODELVIEWPROJECTIONMATRIX`, it uses these two variables as an input argument to the `calculatePosition` function to calculate the transformed position of the incoming vertex. This calculated position returns to the main function as an output storage qualifier in the variable called `position`. The `calculatePosition` function is introduced in this recipe to demonstrate another possible use of storage qualifiers in the local scope of the shader program.

The `Color` variable uses the incoming value of `VertexColor` and passes it to the next stage in which the fragment shader consumes this value to assign the color to the fragments. In order to send data from the vertex shader to fragment shader, both shaders should use the same attribute variable name. The storage qualifier for the vertex shader must be defined as `out` since it is producing an output data for fragment shader. In contrast, the fragment shader must be specified with the `in` storage qualifier, as this receives the data from the previous stage. The fragment shader demonstrates another way of using return values from the shader programming functions.

**There’s more...**

In the current recipe, you learned how to bind the location indices of the generic attribute variables in OpenGL ES from the shader program using layout qualifiers. As an alternative, the `glBindAttribLocation` API can also be used to explicitly bind the location index.

**Syntax:**

```c
void glBindAttribLocation( GLuint program, GLuint index, const GLchar *name );
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>program</code></td>
<td>This is the program object handle</td>
</tr>
<tr>
<td><code>index</code></td>
<td>This is the index of the generic vertex attribute or variable</td>
</tr>
<tr>
<td><code>name</code></td>
<td>This is the vertex shader attribute variable that the index is to be bound</td>
</tr>
</tbody>
</table>

However, it is advisable to encourage layout qualifier as it does not produce the overhead of an API call for attaching the location index to shader program. The use of a layout location qualifier in the shader programing avoids the binding of attribute location at runtime in the OpenGL ES program.
See also

- Refer to the Using the per-vertex attribute to send data to a shader recipe in Chapter 1, OpenGL ES 3.0 on Android/iOS
- Refer to the Efficient rendering with Vertex Buffer Object recipe in Chapter 2, OpenGL ES 3.0 Essentials

Grouping uniforms and creating buffer objects

The interface block helps in grouping the uniform variables into one logical bunch. This is very useful in grouping the related variables in the shader programing. The interface block gives an opportunity to share the uniform data among multiple programs at once. This allows us to set multiple uniform variables in one go, which can be used many times.

A Uniform Buffer Object (UBO) is a buffer object for the interface blocks (containing uniform) similar to VBO, IBO, and so on. It stores the contents of the interface block in the GPU memory for quick data access at runtime. The UBO uses bind points that act as a mediator between the uniform block and uniform buffer. In this recipe, we will create a uniform block and learn how to program uniform buffer objects.

This recipe demonstrates the concept of interface block. In this recipe, we created an interface block to store transformation matrices. This block contain three uniforms. The interface block is stored as a buffer object using the UBO feature. This allows us to store the interface block as an OpenGL ES buffer object.

Getting ready

The syntax to create the uniform block is very simple. The following table shows the syntax and use test cases of the implementation:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Individual uniforms</th>
<th>Uniform blocks</th>
</tr>
</thead>
</table>
| uniform <block name>{  
  [Type] <variable name 1>;  
  [Type] <variable name 2>;  
  . . .  
}; | uniform mat4 ModelMatrix;  
uniform mat4 ViewMatrix;  
uniform mat4 ProjectionMatrix; | uniform Transformation{  
  mat4 ModelMatrix;  
  mat4 ViewMatrix;  
  mat4 ProjectionMatrix;  
}; |
How to do it...

Here is the step-by-step description that demonstrates the interface block and helps in programming the uniform block object:

1. Reuse the previous recipe, *Managing variable attributes with qualifiers*, and create the vertex shader (`UniformBlockVertex.glsl`) as shown here:

```glsl
#version 300 es
layout(location = 0) in vec4 VertexPosition;
layout(location = 1) in vec4 VertexColor;

text vec4 Color;
// Uniform Block Declaration
uniform Transformation{
    mat4 ModelMatrix;
    mat4 ViewMatrix;
    mat4 ProjectionMatrix;
};

void main()
{
    gl_Position = ProjectionMatrix * ViewMatrix * ModelMatrix * VertexPosition;
    Color = VertexColor;
}
```

2. Create the fragment shader, (`UniformBlockFragment.glsl`), as follows:

```glsl
#version 300 es
precision mediump float;
in vec4 Color;
layout(location = 0) out vec4 outColor;
void main()
{
    outColor = vec4(Color.x, Color.y, Color.z, 1.0);
}
```

3. In the `Cube::InitModel()` function, compile the given shader(s) and create the program object. Make sure that the program is in use (`glUseProgram`) before the UBO creation is attempted. In this recipe, we created the UBO in a separate class member function `CreateUniformBufferObject`. Follow these steps to understand this function:

```cpp
void Cube::CreateUniformBufferObject()
{
```
// Get the index of the uniform block
char blockIdx = glGetUniformBlockIndex(program->ProgramID, "Transformation");

// Query uniform block size
GLint blockSize;
glGetActiveUniformBlockiv(program->ProgramID, blockIdx, GL_UNIFORM_BLOCK_DATA_SIZE, &blockSize);

// Bind the block index to BindPoint
GLint bindingPoint = 0;
glUniformBlockBinding(program->ProgramID, blockIdx, bindingPoint);

// Create Uniform Buffer Object (UBO) Handle
 glGenBuffers(1, &UBO);
 glBindBuffer(GL_UNIFORM_BUFFER, UBO);
 glBufferStorage(GL_UNIFORM_BUFFER, blockSize, 0, GL_DYNAMIC_DRAW);

// Bind the UBO handle to BindPoint
 glBindBufferBase(GL_UNIFORM_BUFFER, bindingPoint, UBO);

4. Query the index of the uniform block that is defined in the vertex shader using the
   glGetUniformBlockIndex API into blockIdx. This API accepts the program ID
   and the name of the uniform block whose block index needs to be queried.

5. Use blockIdx and query the block data size in the blockSize variable with the
   help of the glGetActiveUniformBlockiv API. Bind the uniform block index to
   binding point bindingPoint with glUniformBlockBinding.

6. Create the object handle for uniform buffer block and bind it to the symbolic constant
   GL_UNIFORM_BUFFER, and allocate the required memory specified by blockSize.
   Finally, bind the UBO with binding point by using glBindBufferBase.

7. In the render function, make use of buffer object memory mapping to modify the
   content of UBO:

   void Cube::RenderCube()
   {

        // Bind the UBO
        glBindBuffer( GL_UNIFORM_BUFFER, UBO );

        // Map the buffer block for MVP matrix
        glm::mat4* matrixBuf = (glm::mat4*)glMapBufferRange(
                GL_UNIFORM_BUFFER, 0, sizeof(glm::mat4*)*(3),
GL_MAP_WRITE_BIT);
// Assign updated matrix
matrixBuf[0] = *TransformObj->TransformGetModelMatrix();
matrixBuf[1] = *TransformObj->TransformGetViewMatrix();
// UnMap the buffer block
glUnmapBuffer ( GL_UNIFORM_BUFFER );

// Draw Geometry using VBO..
.
.
.
}

### How it works...

The uniform block declaration in the vertex shader groups the model, view, and projection matrices into one logical block called **transformation**. When the shader program gets compiled, it assigns a unique ID/index to the block called block index. The user-defined location indexes are not permitted in uniform blocks. The following five steps are required to create a UBO:

1. Use the `glGetUniformBlockIndex` API to query the Transformation ID in the blockIdx variable.
2. In order to allocate the memory for the UBO, use the `glGetActiveUniformBlockiv` API to query the size of the Transformation uniform block in the blockSize variable.
3. Bind blockIdx (block index) to bindingPoint (binding point) using the `glUniformBlockBinding` API. UBO uses the concept of binding points to create a connection between the block object and the buffer object. Both must be bound to the binding point.
4. Unlike the buffer objects (VBO and IBO) are created in OpenGL ES, similarly create the uniform buffer object. The `glBindBuffer` and `glBufferData` APIs must use the GL_UNIFORM_BUFFER symbolic constant to ensure UBO buffer to the OpenGL ES state machine.
5. As mentioned in step 3, we need to attach the UBO with the respective binding point that is already attached to the block index. Use the `glBindBufferBase` API to bind UBO and bindingPoint.

The UBO can be used to set several values with the single UBO binding call. `RenderCube()` binds the UBO to set the uniform values for model, view, and projection matrices. The buffer object allows modifications to buffer elements using buffer-mapping techniques.
The OpenGL ES 3.0 release has introduced a new feature for range buffer mapping. This feature allows us to modify a subset of the buffer object. Unlike the old buffer-mapping technique, where the complete buffer needs to be mapped onto the CPU side, this technique appears to be much more efficient.

Use the `glMapBufferRange` API to map the UBO on the client side to modify the model, view, and projection matrices with updated values. Make sure that you unmap the buffer object after modification is completed by using the `glUnmapBufferAPI`. Use the existing code for VBO rendering.

**There's more...**

The following figure describes the concept of binding point in UBOs. Each uniform block is identified with a unique index within the shader program. This index is attached to a binding point. Similarly, the UBO is also attached to the binding point and provides a mechanism to share the same data among different programs.

In the preceding figure, `P1_2` and `P2_1` are pointing to the same binding point. Therefore, both share the same data.

**See also**

- Refer to the *Efficient rendering with Vertex Buffer Object* recipe in Chapter 2, *OpenGL ES 3.0 Essentials*
- *Reading and writing buffer objects with mapping*
Chapter 3

Managing VBO with Vertex Array Objects

In Chapter 2, *OpenGL ES 3.0 Essentials*, we introduced two features to load the vertex attributes using vertex arrays and *Vertex Buffer Object (VBO)*. Both these features allow us to load the vertex attribute in the OpenGL ES rendering pipeline. The VBO are considered efficient compared to vertex arrays because they store the vertex data in the GPU memory. This reduces the cost of data copy between CPU and GPU. In this recipe, we will understand a new feature: *Vertex Array Objects (VAO)* of OpenGL ES 3.0. This feature is more efficient compared to VBO.

When a vertex attribute is loaded, it requires some additional calls to set the attribute states in the OpenGL ES rendering pipeline. For example, prior to rendering, the buffer object is bound using the `glBindBuffer` API, the data array is assigned using the `glVertexAttribPointer` API, and the vertex attribute is enabled using the `glEnableVertexAttribArray` API. The VAO stores all such states into a single object in order to remove the overhead caused by these calls.

This allows the application to quickly switch among available vertex array buffers and set their respective states. This makes the rendering efficient and also helps keep the programming code compact and clean.

**How to do it...**

This recipe demonstrates a simple grid geometry rendering using VAO in conjunction with VBO. There is no change required in shaders for programming VAO. Perhaps previous recipes from this chapter can be used.

The steps to create VAO are very straightforward:

1. Create a `Grid` class and define the geometry in the `CreateGrid` function. This function takes the dimension and division of the grid. Inside this function, create a VBO, IBO, and VAO, as shown in the following code:

```c++
void Grid::CreateGrid(GLfloat XDim, GLfloat ZDim, int XDiv,
                      int ZDiv)
{
    // Define geometry using Dimension and divisions
    // Create VBO and IBO for grid geometry
    // Create Vertex Array Object
    // Enable VBO and set attribute parameters
    // Unbind VAO, VBO and IBO
}
```
New Features of OpenGL ES 3.0

2. Create a VBO, generate the buffer, and fill in the buffer object with the vertex information:

```c
// Create VBO ID
glGenBuffers(1, &vIdGrid);
glBindBuffer(GL_ARRAY_BUFFER, vIdGrid);
glBufferData(GL_ARRAY_BUFFER, size, 0, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, size, gridVertex);
```

3. Similarly, create an IBO and fill in the buffer with the element indexes:

```c
// Create IBO for Grid
unsigned short indexSize=sizeof(unsigned short)*indexNum;
glGenBuffers(1, &iIdGrid);
glBindBuffer(GL_ARRAY_BUFFER, iIdGrid);
glBufferData(GL_ARRAY_BUFFER, indexSize, 0, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, indexSize, gridIndices);
```

4. Generate the VAO ID using the `glGenVertexArrays` API. Bind this generated `Vertex_VAO_Id` using `glBindVertexArray`. The code written after the creation of the VAO is recorded in the state vector of the VAO object. Therefore, use the VBO and bind the data to the required vertex attribute for rendering purposes:

```c
// Create Vertex Array Object
glGenVertexArrays(1, &Vertex_VAO_Id);
glBindVertexArray(Vertex_VAO_Id);
// Create VBO and set attribute parameters
glBindBuffer(GL_ARRAY_BUFFER, vIdGrid);
glEnableVertexAttribArray(VERTEX_LOCATION);
glVertexAttribPointer(VERTEX_LOCATION, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
```

5. Unbind the VAO, VBO, and IBO, once the vertex states and attributes are set properly:

```c
glBindVertexArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

6. Render the geometry with VAO using the `Render()` function, as shown here:

```c
// void Grid::Render()
// Use shader program and apply transformation
....

glBindVertexArray(Vertex_VAO_Id); // Bind VAO
glDrawElements(GL_LINES, ((XDivision+1)+(ZDivision+1))*2,
GL_UNSIGNED_SHORT, (void*)0); }
```
How it works...

The VAO stores the vertex array client states and the buffer binding in a state vector. When the VAO ID is bound, the subsequent operation calls, such as calls to bind with VBO, enable client states, and attach data buffer to generic attributes, are stored in the state vector of the VAO. This way, when the VAO is bound, the state vector provides the full state of current settings, configurations, and client states of the vertex array. Instead of making several calls, this one binding call will be sufficient to enable vertex array configurations and states.

See also

- Refer to the Rendering primitives with vertex arrays recipe in Chapter 2, OpenGL ES 3.0 Essentials

Reading and writing buffer objects with mapping

The previous recipe introduced a new feature to access vertex arrays using VAO. This object minimizes the overhead of switch among vertex arrays and their respective states. This recipe will go one step ahead in order to teach you how to update the data of the buffer objects using buffer mapping. The VBO can be updated using `glBufferData` and `glBufferSubData` as demonstrated in many recipes. These APIs can be used to upload or download data to the device. In contrast, the buffer mapping is an efficient way to update the buffer objects that are residing in the GPU memory.
This recipe will demonstrate buffer object range mapping. In this recipe, we will reuse the cube geometry and render each vertex of the cube as a point primitive, instead of a triangle primitive. Each vertex of the cube is programmed to change its colors randomly using the buffer object range mapping feature after a fixed interval of time.

**Getting ready**

Before we start with a step-by-step description, here is the overview of buffer object range mapping:

1. Bind the buffer that needs to be mapped using `glBindBuffer`.
2. Get the pointer to the memory location from driver memory space using the `glMapBufferRange` API.
3. Use this pointer to perform any read/write operations on the acquired memory.
4. Invalidate the acquire pointer using the `glUnmapBuffer` API. This API allows us to send updated memory contents to the GPU memory space.

**How to do it...**

This recipe does not require any special change in the vertex and fragment shaders. For this recipe, we used a new GL shading language API called `glPointSize`. This API is used to specify the size of the `GL_POINTS` primitives. Make use of the Efficient rendering with Vertex Buffer Object recipe in Chapter 2, *OpenGL ES 3.0 Essentials*, and proceed with the following steps to program range mapping onto a buffer object:

1. First, create the VAO of the cube geometry using the previous VAO recipe.
2. Program the map range buffer inside the `Render()` function as shown here. The following steps will describe this function:

```cpp
void Cube::RenderCube(){
  if (clock() - last >= CLOCKS_PER_SEC * 0.1) {
    // Bind the Buffer Object for vertex Array.
    glBindBuffer( GL_ARRAY_BUFFER, vId );
    // Get the mapped memory pointer.
    GLfloat* colorBuf = (GLfloat* )glMapBufferRange(
      GL_ARRAY_BUFFER, size, size, GL_MAP_WRITE_BIT);
    for(int i=0; i<size/sizeof(GLfloat); i++){
      colorBuf[i] = float(rand()%255)/255;  
    }
    last = clock();
    // Invalidate the mapped memory.
    glUnmapBuffer ( GL_ARRAY_BUFFER );
  }
```

`112`
// Perform Transformation.

// Bind the VAO and Render the cube
// with Point primitive.
glBindVertexArray(Vertex_VAO_Id);
glDrawElements(GL_POINTS, 36, GL_UNSIGNED_SHORT, (void*)0);
}

3. First, bind the VBO in order to map the color buffer data using the `glBindBuffer` API. Map the pointer to the color data memory. The color data in the VBO starts from the size index and is also size bytes long:

colorBuf = (GLfloat*)glMapBufferRange (GL_ARRAY_BUFFER, size, size, GL_MAP_WRITE_BIT);

On successful mapping of the buffer object, it returns a valid pointer to the memory mapped location. If an error occurs, the API would return the NULL pointer.

- **Syntax:**

```c
void *glMapBufferRange(GLenum target, GLintptr offset, GLsizeiptr length, GLbitfield access);
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>This specifies the type of buffer, which is expected to bind for memory mapping, for example, GL_MAP_READ_BIT and GL_MAP_WRITE_BIT</td>
</tr>
<tr>
<td>offset</td>
<td>This specifies the starting offset within the buffer object that is the subject of interest for mapping</td>
</tr>
<tr>
<td>length</td>
<td>This specifies the range of the buffer that needs to be mapped</td>
</tr>
<tr>
<td>access</td>
<td>This is the symbol constant flag combination that indicates the desired access to the buffer range</td>
</tr>
</tbody>
</table>

4. Copy the new color values in this mapped memory buffer:

```c
// size/sizeof(GLfloat) gives total number of elements
// that needs to be updated with new color, the formula
// is- total size of buffer / unit item size
for(int i=0; i<size/sizeof(GLfloat); i++){
    colorBuf[i] = float(rand()%255)/255;
}
```
5. Unmap the memory mapped buffer to indicate the OpenGL ES rendering pipeline to transfer this data to the GPU memory space:

```c
glUnmapBuffer ( GL_ARRAY_BUFFER );
```

The `glUnmapBuffer` API returns the Boolean TRUE if it successfully unmaps the current mapped buffer. If some error occurs, it returns FALSE.

- **Syntax:**

  ```c
  GLboolean glUnmapBuffer(GLenum target);
  ```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>This specifies the type of the buffer that needs to unbound</td>
</tr>
</tbody>
</table>

6. Bind the VAO and render the geometry using the GL_POINTS primitive. The GL_POINTS primitive renders small dots on the screen. In order to increase the dimension of these dots, the `glPointSize` API can be used in the vertex shader, as shown in the next step:

```c
glBindVertexArray(Vertex_VAO_Id);
glDrawElements(GL_POINTS, 36, GL_UNSIGNED_SHORT, (void*)0);
```

7. Create `BufferMappingVertex.glsl` as follows:

```c
layout(location = 0) in vec4 VertexPosition;
layout(location = 1) in vec4 VertexColor;
uniform mat4 MODELVIEWPROJECTIONMATRIX;
out vec4 Color;
void main(){
  gl_Position = MODELVIEWPROJECTIONMATRIX * VertexPosition;
  gl_PointSize= 80.0; // Size of GL_POINTS primitive
  Color       = VertexColor;
}
```

**How it works...**

In the VBO, `glBufferData` and `glBufferSubData` use the user data and copy it into a hooked/pinned location in the device memory location. This hooked location can be accessed by the GPU. The user data is copied to this memory location like `memcpy` internally. As the data copying process gets completed, the driver starts direct memory allocation (DMA) without intervening the CPU cycles.

The target destination of the DMA depends upon the usage hints from the (GL_STREAM_DRAW, GL_STREAM_READ, GL_STREAM_COPY, GL_STATIC_DRAW, GL_STATIC_READ, GL_STATIC_COPY, GL_DYNAMIC_DRAW, GL_DYNAMIC_READ, or GL_DYNAMIC_COPY) APIs.
In contrast, the `glMapBufferRange` method is considered much more efficient. The API first hooks a memory location directly into the driver memory space. This pinned memory location is available through a pointer to the application. This pointer can be directly used to update the location for the uploading or downloading of data for read/write purposes. Once the operation on the mapped location for read/write is completed, the pointer can be made invalid by calling `glUnMapBuffer`. This API call hints the OpenGL ES pipeline to push the updated data to the GPU memory using DMA calls.

**See also**

- Refer to the Swizzling recipe in Appendix, *Supplementary Information on OpenGL ES 3.0*
- Refer to the Transform feedback particle system with sync objects and fences recipe in Chapter 12, *Real-time Shadows and Particle System*

**Render multiple objects with geometry instancing**

The geometry instancing allows us to render multiple instances of the same object in a single rendering API call. These multiple instances differ in their generic attributes, such as transformation matrices, color, scale, and so on. This feature is very useful to implement particle systems, crowd simulation, rendering of jungle trees, and so on. Compared to the traditional way of rendering multiple objects that use multiple rendering calls, this technique is very efficient as it requires a single API call. This reduces the overhead of CPU processing in sending multiple rendering calls to the OpenGL ES rendering engine.
This recipe demonstrates the rendering of 1000 cubes using geometric instancing. For this, we will use 1000 matrices in a VBO. Each matrix contains a transformation to place a cube in the 3D space. The information of the matrices are updated using the range map buffer feature as discussed in the previous recipe. This allows us to pass new transformation data on the fly at run time. The transformed data contains new rotation and translated positions.

**How to do it...**

So far, in our recipes, the model-view-projection matrix is always treated as uniform in the vertex shader. For this recipe, we will make use of the VAO and declare the model-view-projection matrix as a generic attribute instead of a uniform. Since the matrix is an attribute, a new VBO is required. This VBO is stored in the `matrixId` variable. `RenderCube()` uses the map buffer to update transformation matrix data.

Here are the steps to implement geometric instancing:

1. Create the vertex shader and add the following code. There is no change required for the fragment shader. It can be reused:

   ```glsl
   #version 300 es
   layout(location = 0) in vec4 VertexPosition;
   layout(location = 1) in vec4 VertexColor;
   ```
layout(location = 2) in mat4 MODELVIEWPROJECTIONMATRIX;
out vec4 Color;
void main() {
    gl_Position = MODELVIEWPROJECTIONMATRIX * VertexPosition;
    Color = VertexColor;
}

2. In Cube::InitModel(), use the existing code and add a new VBO for matrix transformation. Get the ID of the generated buffer object in matrixId:

    // Create VBO for transformation matrix
    glGenBuffers(1, &matrixId);
    glBindBuffer (GL_ARRAY_BUFFER, matrixId);

3. Allocate the memory to the VBO for matrix transformation. The dimension variable is initialized with 10. It gives the number of cubes along an axis. Therefore, along x, y, and z axes, 10 x 10 x 10 = 1000 cubes. The total size of the buffer would be size of (GLfloat) * 16 (16 float elements in mat4) * 1000 (cubes):

    glm::mat4 transformMatrix[dimension][dimension][dimension];
    glBufferData(GL_ARRAY_BUFFER, sizeof(transformMatrix) , 0, GL_DYNAMIC_DRAW);

    The glBufferData uses GL_DYNAMIC_DRAW. This symbolic constant specifies that the buffer is going to contain some data that is dynamic in nature. In other words, the data will require updates in the buffer. This symbolic constant helps the graphics driver to manage buffer memory in the best possible way to achieve high-performance graphics rendering.

4. In the same function, after creating the VAO (Vertex_VAO_Id), define the generic attribute states and configuration of the transformation matrix buffer object. This helps in saving the vertex array client states and the buffer binding in the VAO (Vertex_VAO_Id). The glVertexAttribDivisor calculates the instance ID from the total number of instances given. For more information, refer to the There's more... section in this recipe:

    // Create VBO for transformation matrix and set attributes
    glBindBuffer( GL_ARRAY_BUFFER, matrixId );
    glEnableVertexAttribArray(MATRIX1_LOCATION);
    glEnableVertexAttribArray(MATRIX2_LOCATION);
    glEnableVertexAttribArray(MATRIX3_LOCATION);
    glVertexAttribPointer(MATRIX1_LOCATION,4,GL_FLOAT,GL_FALSE,
                         sizeof(glm::mat4), (void*)(sizeof(float)*0));
    glVertexAttribPointer(MATRIX2_LOCATION,4,GL_FLOAT,GL_FALSE,
                         sizeof(glm::mat4), (void*)(sizeof(float)*4));
New Features of OpenGL ES 3.0

```c
glVertexAttribPointer(MATRIX3_LOCATION, 4, GL_FLOAT, GL_FALSE,
                     sizeof(glm::mat4), (void*) (sizeof(float)*8));
glVertexAttribPointer(MATRIX4_LOCATION, 4, GL_FLOAT, GL_FALSE,
                     sizeof(glm::mat4), (void*) (sizeof(float)*12));
glVertexAttribDivisor(MATRIX1_LOCATION, 1);
glVertexAttribDivisor(MATRIX2_LOCATION, 1);
glVertexAttribDivisor(MATRIX3_LOCATION, 1);
glVertexAttribDivisor(MATRIX4_LOCATION, 1);

5. In Cube::RenderCube(), use range buffer mapping to map the transformation buffer on the client-side memory. Update the data in the memory and unmap it. Use VAO and render the cube of cubes using the geometric instance API called glDrawElementsInstanced. This API's last argument specifies the number of instances the given primitive will be rendered:

```c
void Cube::RenderCube()
{
    glBindBuffer(GL_ARRAY_BUFFER, matrixId);
    glm::mat4* matrixBuf = (glm::mat4*)glMapBufferRange
                           (GL_ARRAY_BUFFER, 0, sizeof(glm::mat4*)*(dimension
                            *dimension*dimension), GL_MAP_WRITE_BIT);
    static float l = 0;
    TransformObj->TransformRotate(l++, 1, 1, 1);
    TransformObj->TransformTranslate
    (-distance*dimension/4, -distance*dimension/4, -distance*dimension/4);
    glm::mat4 projectionMatrix = *TransformObj->
                        TransformGetProjectionMatrix();
    glm::mat4 modelMatrix = *TransformObj->
                        TransformGetModelMatrix();
    glm::mat4 viewMatrix = *TransformObj->
                        TransformGetViewMatrix();
    int instance = 0;
    for ( int i = 0; i < dimension; i++ ){
        for ( int j = 0; j < dimension; j++ ){
            for ( int k = 0; k < dimension; k++ ){
                matrixBuf[instance++] = projectionMatrix *
                                viewMatrix * glm::translate(modelMatrix, glm::vec3(
                                i*distance, j*distance, k*distance)) *
                                glm::rotate(
                                modelMatrix, 1, glm::vec3(1.0, 0.0, 0.0));
            }
        }
    }
```
glUnmapBuffer ( GL_ARRAY_BUFFER );

glBindVertexArray(Vertex_VAO_Id);
glDrawElementsInstanced(GL_TRIANGLES, 36,
  GL_UNSIGNED_SHORT, (void*)0, dimension*dimension*dimension);

How it works...

The application first compiles the shader programs. This makes us aware of all the generic attribute locations used in the shader program. Create a VBO of 1000 matrix elements. Each element represents a transformation matrix. This matrix element is updated with new values of the transformation of every frame in the RenderCube function.

The generic attributes are first enabled using glEnableVertexAttribArray. The data array is attached to the generic location with glVertexAttribPointer. The following figure shows how the OpenGL ES program API is attached to the layout location of the vertex shader to send data:

```c
#define VERTEX_LOCATION 0
#define COLOR_LOCATION 1

layout(location = 0) in vec4 VertexPosition;
layout(location = 1) in vec4 VertexColor;
layout(location = 2) in mat4 MODELVIEWPROJECTIONMATRIX;
out vec4 Color;

void main()
{
    gl_Position = MODELVIEWPROJECTIONMATRIX * VertexPosition;
    Color = VertexColor;
}

glBindBuffer( GL_ARRAY_BUFFER, vbo );
glEnableVertexAttribArray(VERTEX_LOCATION);
glEnableVertexAttribArray(COLOR_LOCATION);
glVertexAttribPointer(VERTEX_LOCATION, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
glVertexAttribPointer(COLOR_LOCATION, 3, GL_FLOAT, GL_FALSE, 0, (void*)size);
```
New Features of OpenGL ES 3.0

Note that the generic attributes are sent as a group of four. Therefore, for a 4 x 4 matrix, we will need four attribute locations. The start location of the attribute should be mentioned into the vertex shader using a layout qualifier:

```
layout(location = 2) in mat4 MODELVIEWPROJECTIONMATRIX;
```

The following figure shows how the attribute locations are managed by the compiler:

![Generic attributes are sent as a group of 4. Therefore, a matrix needs 4 attribute locations.](image)

Similar to the other locations such as VERTEX_LOCATION (0) and COLOR_LOCATION (1), the transformation matrix locations (2, 3, 4, 5) also need to be enabled and attached to the array data.

The `glVertexAttribDivisor` API is responsible for controlling the rate at which OpenGL ES advances the data from an instanced array. The first parameter of this API specifies the generic attribute that needs to be treated as an instanced array. This tells the OpenGL ES pipeline to use this attribute per instance rendering. For example, in this example, the generic attributes, 2, 3, 4, 5, are instanced attributes. Therefore, OpenGL ES consumes the data from the transformation matrix array as an instance ID. We will see how this instance ID is calculated in a moment.

The default value of the divisor is 0 when it is not specified in the program explicitly. If the divisor is 0, the attribute index is advanced once per-vertex. If the divisor is not 0, the attribute advances once per divisor instance of the set(s) of the vertices being rendered.
Syntax:

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>This specifies generic attribute layout location</td>
</tr>
<tr>
<td>divisor</td>
<td>This specifies the number of instances that will pass between updates of the generic attribute at the index slot</td>
</tr>
</tbody>
</table>

The rendering of the geometric instancing requires special instanced-based drawing APIs from OpenGL ES 3.0, as mentioned here for array- and index-based geometric data.

Syntax:

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>This specifies the type of the primitive that needs to be rendered</td>
</tr>
<tr>
<td>count</td>
<td>This specifies the number of indices considered in the drawing</td>
</tr>
<tr>
<td>type</td>
<td>This is used by glDrawElementsInstanced, this specifies the data type of the indices stored</td>
</tr>
<tr>
<td>indices</td>
<td>This specifies the arrays containing the order of the indices</td>
</tr>
<tr>
<td>primcount</td>
<td>This specifies the number of copies to be rendered</td>
</tr>
</tbody>
</table>

In the present recipe, the `glDrawElementsInstanced` API is used to render multiple instances of the same object. This API works in conjunction with another API called `glVertexAttribDivisor`. In order to update the VBO matrix elements, buffer mapping is used, which is an efficient way to update the buffer elements. If the geometric data is not index based but array based, then `glDrawArraysInstanced` can be used. This API accepts almost the same parameters. Refer to the online *OpenGL ES 3.0 Reference Manual* for more information.
There's more...

The second attribute of `glVertexAttribDivisor` specifies the divisor. This divisor helps in calculating the instance ID from the total number of instances. The following figure shows a simple example of the working logic of this API. In this figure, we assumed that there are total five instances to be rendered, and the figure contains five matrices. When the divisor is 5, it produces 5 instance ID of the (0, 1, 2, 3, 4). This instance ID will be used as an index to the transformation matrix array. Similarly, when the divisor is 2, it generates three instances (0, 1, 2). It generates two instances (0, 1) when the divisor is 3.

<table>
<thead>
<tr>
<th>Divisor</th>
<th>INSTANCES</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0/1 = 0</td>
<td>1/1 = 1</td>
<td>2/1 = 2</td>
<td>3/1 = 3</td>
<td>4/1 = 4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0/2 = 0</td>
<td>1/2 = 0</td>
<td>2/2 = 1</td>
<td>3/2 = 1</td>
<td>4/2 = 2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0/3 = 0</td>
<td>1/3 = 0</td>
<td>2/3 = 0</td>
<td>3/3 = 1</td>
<td>4/3 = 1</td>
<td></td>
</tr>
</tbody>
</table>

instanceID = INSTANCES/Divisor

See also

- Managing VBO with Vertex Array Objects
- Refer to the Efficient rendering with ETC2 compressed texture and Implementing Skybox with seamless cube mapping recipes in Chapter 7, Texture and Mapping Techniques

Rendering multiple primitives with primitive restart

OpenGL ES 3.0 introduced a new feature called primitive restart, where multiple disconnected geometry primitives can be rendered using a single API. This feature uses a special marker in the vertex data or the index data to concatenate different geometries of the same drawing type into a single batch. The restart primitive feature executes on the GPU. Therefore, it eliminates the communication overhead per drawing call. This provides high-performance graphics by avoiding multiple drawing calls from CPU to GPU.
The recipe shows us how to use the primitive restart technique to render a cube using two sets of geometries, which are separated by a special marker.

**Getting ready**

The marker used by the restart primitive feature to separate geometries is the highest value of the data type with which the element index or vertex data array is specified. For instance, an index value of `GLushort` and `GLint` should be $0xFFF (65535)$ and $0xFFFFFFFF (4294967295)$, respectively.
New Features of OpenGL ES 3.0

How to do it...

To render multiple primitives, follow these steps:

1. Define the cube vertices and indices, as shown here:

<table>
<thead>
<tr>
<th>Cube vertices</th>
<th>Indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
<td>GLushort cubeIndices[] =</td>
</tr>
<tr>
<td>-1, -1, 1 , // V0</td>
<td>{</td>
</tr>
<tr>
<td>-1, 1, 1 , // V1</td>
<td>0,3,1, 3,2,1,</td>
</tr>
<tr>
<td>1, 1, 1 , // V2</td>
<td>7,4,6, 4,5,6,</td>
</tr>
<tr>
<td>1, -1, 1 , // V3</td>
<td>4,0,5, 0,1,5,</td>
</tr>
<tr>
<td>-1, -1, -1 ,// V4</td>
<td>0xFFFF, 3,7,2,</td>
</tr>
<tr>
<td>-1, 1, -1 , // V5</td>
<td>7,6,2, 1,2,5,</td>
</tr>
<tr>
<td>1, 1, -1 , // V6</td>
<td>2,6,5, 3,0,7,</td>
</tr>
<tr>
<td>1, -1, -1 // V7</td>
<td>0,4,7</td>
</tr>
<tr>
<td>};</td>
<td>};</td>
</tr>
</tbody>
</table>

2. In order to render the cube with primitive restart, it must first of all be enabled, using glEnable(GL_PRIMITIVE_RESTART_FIXED_INDEX). Specify the total size of the indice and include the number of markers that are used in the geometry indices:

   //Bind the VBO
   glBindBuffer( GL_ARRAY_BUFFER, vId );
   glVertexAttribPointer(VERTEX_LOCATION, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
   glVertexAttribPointer(COLOR_LOCATION, 3, GL_FLOAT, GL_FALSE, 0, (void*)size);
   glEnable(GL_PRIMITIVE_RESTART_FIXED_INDEX);
   glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, iId );
   // Plus 36 + 1 because it has 1 Primitive Restart Index.
   glDrawElements(GL_TRIANGLES, 36+1, GL_UNSIGNED_SHORT, (void*)0);
   glDisable(GL_PRIMITIVE_RESTART_FIXED_INDEX);
There's more...

The other way in which the disconnected geometry primitives can be rendered is called triangle degeneration. Triangle degeneration is the capability of the GPU to recognize disconnected primitives in the triangle strip or triangle fan index information on the basis of some special pattern.

For example, the following figure shows the special index pattern data that can be used to render degenerated triangles using the `glDrawElement` or `glDrawElementsInstanced` API.

The degeneration between the two geometries is achieved by repeating the last index of the previous geometry and the first index of the next primitive. This rule of degeneration is only applicable when the previous geometry contains an odd number of triangles. Behind the curtains, the triangle would be drawn in the following order: (0, 1, 2), (2, 1, 3), (2, 3, 3), (3, 3, 6), (3, 6, 6), (6, 6, 7), (6, 7, 8), (8, 7, 10). The repeated indices form an area equivalent to zero, allowing the GPU to discard the triangles. These zero area triangles are mentioned using the bold font.

The second type of degeneration case is where the first geometry contains an odd number of triangles. For instance, the following image demonstrates the first geometry with three (odd) triangles. As per this case rule, the last index of the first geometry is repeated twice, followed by the first index of the second geometry.
For instance, the indices specified for degenerate triangles (0, 1, 2, 3, 4, 4, 4, 8, 8, 9, 10, 11) generate the following triangles: (0, 1, 2), (2, 1, 3), (2, 3, 4), (4, 3, 4), (4, 4, 4), (4, 4, 6), (4, 6, 6), (6, 6, 9), (6, 7, 8), (8, 7, 9), (9, 8, 10).

See also

- Refer to the Using the per-vertex attribute to send data to a shader recipe in Chapter 1, OpenGL ES 3.0 on Android/iOS
- Refer to the Efficient rendering with Vertex Buffer Object recipe in Chapter 2, OpenGL ES 3.0 Essentials
Where to buy this book

You can buy OpenGL ES 3.0 Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.