Best Practices for Working with Vertex Data

使用顶点数据的最佳实践

To render a frame using OpenGL ES your app configures the graphics pipeline and submits graphics primitives to be drawn. In some apps, all primitives are drawn using the same pipeline configuration; other apps may render different elements of the frame using different techniques. But no matter which primitives you use in your app or how the pipeline is configured, your app provides vertices to OpenGL ES. This chapter provides a refresher on vertex data and follows it with targeted advice for how to efficiently process vertex data.

应用程序配置图形管线参数并且提交图形元素来使得OpenGL ES来渲染一帧的内容。在某些应用程序中，所有的图形元素时使用相同的管线配置参数，还有一些应用程序使用各种不同的技术来渲染一帧中的各个元素。但是不管应用程序绘制什么元素或者怎么配置管线的参数，你的应用程序都需要提供顶点数据给OpenGL ES。这一章对于顶点数据会提出一些针对性的建议来说明如何高效的处理顶点数据。

A vertex consists of one or more **attributes**, such as the position, the color, the normal, or texture coordinates. An OpenGL ES 2.0 or 3.0 app is free to define its own attributes; each attribute in the vertex data corresponds to an attribute variable that acts as an input to the vertex shader. An OpenGL 1.1 app uses attributes defined by the fixed-function pipeline

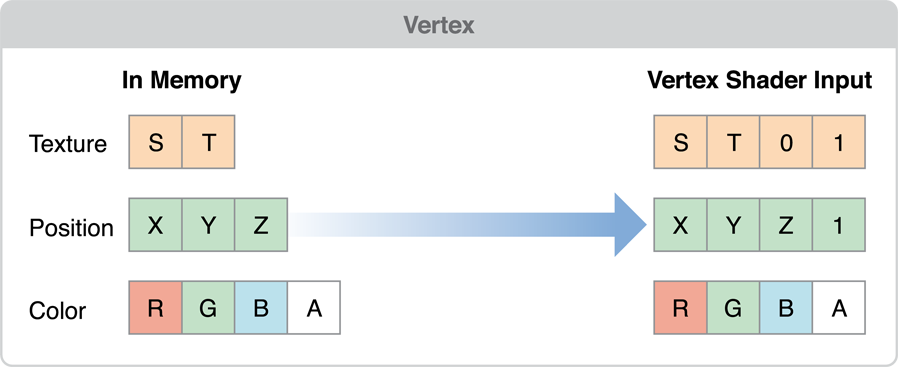
一个顶点是由一个或多个属性组成的，比如位置、颜色、法线或者纹理坐标。OpenGL ES2.0、OpenGL ES3.0的程序可以自由的定义各种属性。顶点中的每一个属性都关联到一个属性变量输入到顶点着色器中。OpenGL 1.1的程序使用固定管线中定义的属性。

You define an attribute as a vector consisting of one to four **components**. All components in the attribute share a common data type. For example, a color might be defined as four GLubyte components (red, green, blue, alpha). When an attribute is loaded into a shader variable, any components that are not provided in the app data are filled in with default values by OpenGL ES. The last component is filled with 1, and other unspecified components are filled with 0, as illustrated in Figure 8-1.

你为顶点定义一个由1到4个分量组成的属性。一个属性里的所有分量使用相同的数据类型。例如，颜色属性被定义为四个GLubyte（红、黄、蓝、透明）。当一个属性被加载到着色器中的变量的时候，任何一个你没有在程序中明确指定的分量都会被设定为OpenGL ES的默认值。最后一个分量被填充为1，其他未指定的分量被0填充，如图8-1所示。

**Figure 8-1**  Conversion of attribute data to shader variables

把属性数据转换成着色器变量



Your app may configure an attribute to be a *constant*, which means the same values are used for all vertices submitted as part of a draw command, or an *array*, which means that each vertex a value for that attribute. When your app calls a function in OpenGL ES to draw a set of vertices, the vertex data is copied from your app to the graphics hardware. The graphics hardware than acts on the vertex data, processing each vertex in the shader, assembling primitives and rasterizing them out into the framebuffer. One advantage of OpenGL ES is that it standardizes on a single set of functions to submit vertex data to OpenGL ES, removing older and less efficient mechanisms that were provided by OpenGL.

应用程序可以配置常量来作为属性，这意味着在一个绘图命令里提交的所有顶点都使用相同的值，也可以用数组来作为属性，这就是说每一个顶点的值都是数组中的一个。当应用程序调用OpenGL ES的函数来画已组顶点时，顶点的数据会被复制到图形硬件。图形硬件会处理顶点数据，在着色器中传输顶点，装配图形元素，并且光栅化到帧缓冲。OpenGL ES的一个优点是标准化了一个函数集合来提交顶点数据，移除了OpenGL中的旧的且没有效率的机制。

Apps that must submit a large number of primitives to render a frame need to carefully manage their vertex data and how they provide it to OpenGL ES. The practices described in this chapter can be summarized in a few basic principles:

应用程序渲染一帧时必须提交大量图形元素，所以需要仔细的管理顶点数据，并且提交给OpenGL ES。这一章就会总结一些基本的原则：

* Reduce the size of your vertex data.

减少顶点数据的尺寸。

* Reduce the pre-processing that must occur before OpenGL ES can transfer the vertex data to the graphics hardware.

在OpenGL ES可以传送顶点数据到图形硬件之前，减少必须产生的预处理。

* Reduce the time spent copying vertex data to the graphics hardware.

减少复制顶点到图形硬件的时间消耗。

* Reduce computations performed for each vertex

减少每个顶点的计算。.

Simplify Your Models

简化模型

The graphics hardware of iOS-based devices is very powerful, but the images it displays are often very small. You don’t need extremely complex models to present compelling graphics on iOS. Reducing the number of vertices used to draw a model directly reduces the size of the vertex data and the calculations performed on your vertex data.

IOS设备的图形硬件是非常强大的，但是它现实的图形通常比较小。所以往往并不需要非常复杂的模型来表现精细的图像。使用删减顶点后的模型，减少顶点数据的尺寸和计算量。

You can reduce the complexity of a model by using some of the following techniques:

可以使用下面的方法来较少模型的复杂度：

* Provide multiple versions of your model at different levels of detail, and choose an appropriate model at runtime based on the distance of the object from the camera and the dimensions of the display.

在不同的细节等级下提供多个版本的模型，并且在运行时根据物体距离摄像机的距离和显示的尺寸来选择一个合适的模型。

* Use textures to eliminate the need for some vertex information. For example, a bump map can be used to add detail to a model without adding more vertex data.

使用纹理来消除一些顶点信息。例如凹凸贴图可以在不增加顶点的同时增加物体的细节。

* Some models add vertices to improve lighting details or rendering quality. This is usually done when values are calculated for each vertex and interpolated across the triangle during the rasterization stage. For example, if you directed a spotlight at the center of a triangle, its effect might go unnoticed because the brightest part of the spotlight is not directed at a vertex. By adding vertices, you provide additional interpolant points, at the cost of increasing the size of your vertex data and the calculations performed on the model. Instead of adding additional vertices, consider moving calculations into the fragment stage of the pipeline instead:

一些模型增加顶点是为了增加光照的细节和渲染质量。通常会这样处理，数值在顶点阶段计算并且插值到三角形光栅化时。例如，如果把一个聚光灯对准三角形的中心，最终的效果或许会不明显，因为聚光灯明亮的部分没有直接照在顶点上。通过增加顶点，来增加额外的插值点，但同时也增加了顶点数据的尺寸和计算量。一种替代增加额外顶点的方法是将计算放到管线的片段阶段：

* + If your app uses OpenGL ES 2.0 or later, then your app performs the calculation in the vertex shader and assigns it to a varying variable. The varying value is interpolated by the graphics hardware and passed to the fragment shader as an input. Instead, assign the calculation’s inputs to varying variables and perform the calculation in the fragment shader. Doing this changes the cost of performing that calculation from a per-vertex cost to a per-fragment cost, reduces pressure on the vertex stage and more pressure on the fragment stage of the pipeline. Do this when your app is blocked on vertex processing, the calculation is inexpensive and the vertex count can be significantly reduced by the change.
  + If your app uses OpenGL ES 1.1, you can perform per-fragment lighting using DOT3 lighting. You do this by adding a bump map texture to hold normal information and applying the bump map using a texture combine operation with the GL\_DOT3\_RGB mode.

Avoid Storing Constants in Attribute Arrays

避免在属性数组里存储常量

If your models include attributes that uses data that remains constant across the entire model, do not duplicate that data for each vertex. OpenGL ES 2.0 and 3.0 apps can either set a constant vertex attribute or use a uniform shader value to hold the value instead. OpenGL ES 1.1 app should use a per-vertex attribute function such as glColor4ub orglTexCoord2f instead.

如果模型包含含有在整个模型中都要使用到的常量，就不要在每个顶点数据中都重复这个常量数据。OpenGL ES 2.0和3.0程序可以设置常量顶点属性或者使用uniform的着色器值来代替。OpenGL ES 1.1程序应该使用为每个顶点设置属性的函数，例如glColor4ub或者glTexCoord2f来代替。

Use the Smallest Acceptable Types for Attributes

将属性的类型设置为可接受的最小的数据类型

When specifying the size of each of your attribute’s components, choose the smallest data type that provides acceptable results. Here are some guidelines:

当为属性分量指定尺寸时，选择最小的数据类型来达到可接受的表现效果。这里是一些指导建议：

* Specify vertex colors using four unsigned byte components (GL\_UNSIGNED\_BYTE).

顶点颜色使用4个unsigned byte（GL\_UNSIGNED\_BYTE）分量。

* Specify texture coordinates using 2 or 4 unsigned bytes (GL\_UNSIGNED\_BYTE) or unsigned short (GL\_UNSIGNED\_SHORT). Do not pack multiple sets of texture coordinates into a single attribute.

纹理坐标使用2或4个unsigned byte（GL\_UNSIGNED\_BYTE）或unsigned short（GL\_UNSIGNED\_SHORT）分量。不要将多套纹理坐标包装到一个属性中。

* Avoid using the OpenGL ES GL\_FIXED data type. It requires the same amount of memory as GL\_FLOAT, but provides a smaller range of values. All iOS devices support hardware floating-point units, so floating point values can be processed more quickly.

避免使用OpenGL ES的GL\_FIXED数据类型。这中数据类型和GL\_FLOAT所占用的内存是一样的，但是能表示的数据范围更小。所有的IOS设备都支持硬件浮点单元，所以浮点值能更快的被处理。

* OpenGL ES 3.0 contexts support a wider range of small data types, such as GL\_HALF\_FLOAT and GL\_INT\_2\_10\_10\_10\_REV. These often provide sufficient precision for attributes such as normals, with a smaller footprint than GL\_FLOAT.

Opengl ES 3.0为小数据类型提供更高的精度，比如GL\_HALF\_FLOAT和GL\_INT\_2\_10\_10\_10\_REV。这些数据类型为属性值提供更高的精度，比如说法线值，但是比GL\_FLOAT更小。

If you specify smaller components, be sure you reorder your vertex format to avoid misaligning your vertex data. See [Avoid Misaligned Vertex Data](https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html#//apple_ref/doc/uid/TP40008793-CH107-SW7).

如果你为属性分量使用了更小的数据类型，请确保顶点数据的格式不会造成错误的对齐。参见避免顶点数据的错误对齐。

Use Interleaved Vertex Data

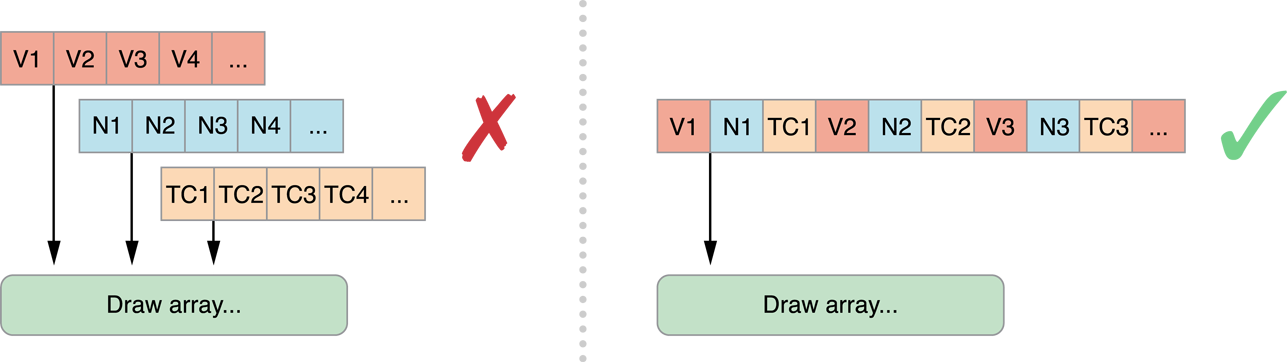
使用交错的顶点数据

You can specify vertex data as a series of arrays (also known as a *struct of arrays*) or as an array where each element includes multiple attributes (an *array of structs*). The preferred format on iOS is an array of structs with a single interleaved vertex format. Interleaved data provides better memory locality for each vertex.

你可以使用多个数组（每个数组只含有一个属性）或者一个数组（数组中的每一个元素是一个含有多个属性的struct）来设置顶点数据。在IOS上更好的选择是第二种。这种交错的数据格式提供更好的局部内存访问。

**Figure 8-2**  Interleaved memory structures place all data for a vertex together in memory

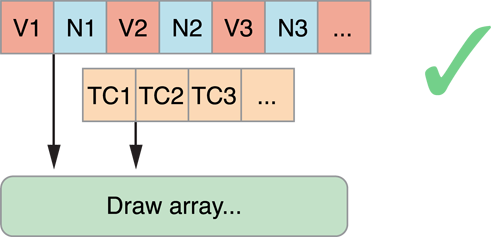
交错的内存结构使得顶点数据在内存中是连续的



An exception to this rule is when your app needs to update some vertex data at a rate different from the rest of the vertex data, or if some data can be shared between two or more models. In either case, you may want to separate the attribute data into two or more structures.

这个规则有一个例外，就是当你的程序需要更新一部分的顶点数据，或者多个模型共享一部分的顶点数据时。这两种情况时，你需要将属性数据分割成两个或者多块。

**Figure 8-3**  Use multiple vertex structures when some data is used differently



Avoid Misaligned Vertex Data

避免顶点数据的错误对齐

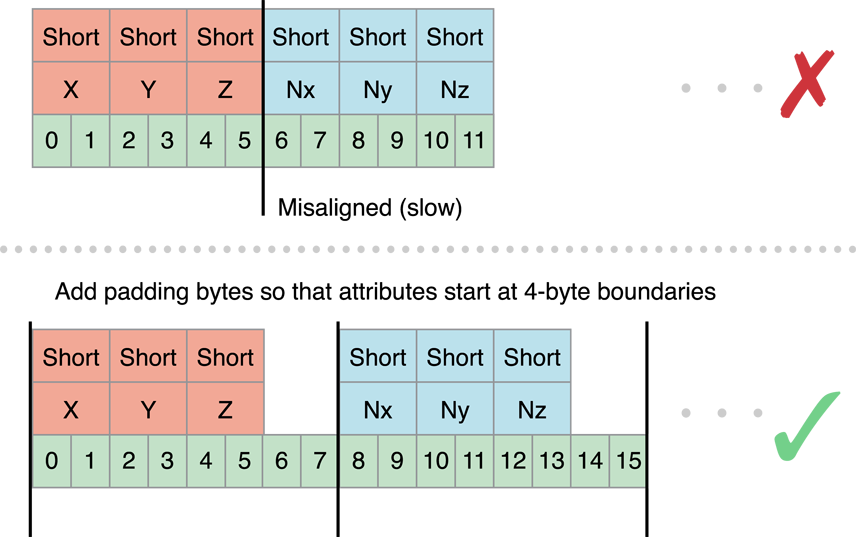
When you are designing your vertex structure, align the beginning of each attribute to an offset that is either a multiple of its component size or 4 bytes, whichever is larger. When an attribute is misaligned, iOS must perform additional processing before passing the data to the graphics hardware.

当你在设计顶点结构时，将每一个属性的起始位置对齐到分量尺寸或者4字节的倍数（取分量尺寸或者4字节中较大的）。当属性的对齐错误的时候，IOS必须在将数据传递到图形硬件前进行往外的处理。

In [Figure 8-4](https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html#//apple_ref/doc/uid/TP40008793-CH107-SW15), the position and normal data are each defined as three short integers, for a total of six bytes. The normal data begins at offset 6, which is a multiple of the native size (2 bytes), but is not a multiple of 4 bytes. If this vertex data were submitted to iOS, iOS would have to take additional time to copy and align the data before passing it to the hardware. To fix this, explicitly add two bytes of padding after each attribute.

如图8-4所示，坐标和法线数据被定义为三个short ingeger，一个分量总共为6个byte。法线数据的起始位置在6的偏移量处，这个值不是4的倍数。如果将这个顶点数据提交到IOS，IOS会在将数据传递到硬件前消耗额外的时间来拷贝和对齐数据。想要修正这个问题，需要显示的增加两个字节来填充在每个属性的最后。

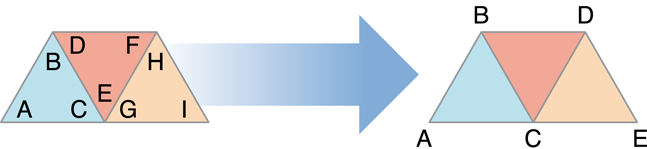
**Figure 8-4**  Align Vertex Data to avoid additional processing



Use Triangle Strips to Batch Vertex Data

Using triangle strips significantly reduces the number of vertex calculations that OpenGL ES must perform on your models. On the left side of Figure 8-5, three triangles are specified using a total of nine vertices. C, E and G actually specify the same vertex! By specifying the data as a triangle strip, you can reduce the number of vertices from nine to five.

**Figure 8-5**  Triangle strip

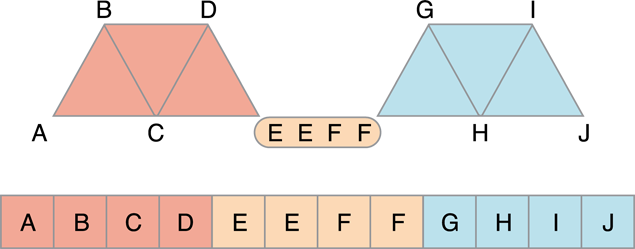


Sometimes, your app can combine more than one triangle strip into a single larger triangle strip. All of the strips must share the same rendering requirements. This means:

* You must use the same shader to draw all of the triangle strips.
* You must be able to render all of the triangle strips without changing any OpenGL state.
* The triangle strips must share the same vertex attributes.

To merge two triangle strips, duplicate the last vertex of the first strip and the first vertex of the second strip, as shown in Figure 8-6. When this strip is submitted to OpenGL ES, triangles DEE, EEF, EFF, and FFG are considered degenerate and not processed or rasterized.

**Figure 8-6**  Use degenerate triangles to merge triangle strips



For best performance, your models should be submitted as a single indexed triangle strip. To avoid specifying data for the same vertex multiple times in the vertex buffer, use a separate index buffer and draw the triangle strip using the glDrawElements function (or the glDrawElementsInstanced or glDrawRangeElements functions, if appropriate).

In OpenGL ES 3.0, you can use the primitive restart feature to merge triangle strips without using degenerate triangles. When this feature is enabled, OpenGL ES treats the largest possible value in an index buffer as a command to finish one triangle strip and start another. Listing 8-1 demonstrates this approach.

**Listing 8-1**  Using primitive restart in OpenGL ES 3.0

|  |
| --- |
| // Prepare index buffer data (not shown: vertex buffer data, loading vertex and index buffers) |
| GLushort indexData[11] = { |
| 0, 1, 2, 3, 4, // triangle strip ABCDE |
| 0xFFFF, // primitive restart index (largest possible GLushort value) |
| 5, 6, 7, 8, 9, // triangle strip FGHIJ |
| }; |
|  |
| // Draw triangle strips |
| glEnable(GL\_PRIMITIVE\_RESTART\_FIXED\_INDEX); |
| glDrawElements(GL\_TRIANGLE\_STRIP, 11, GL\_UNSIGNED\_SHORT, 0); |

Where possible, sort vertex and index data so triangles that share common vertices are drawn reasonably close to each other in the triangle strip. Graphics hardware often caches recent vertex calculations to avoid recalculating a vertex.

Use Vertex Buffer Objects to Manage Copying Vertex Data

Listing 8-2 provides a function that a simple app might use to provide position and color data to the vertex shader. It enables two attributes and configures each to point at the interleaved vertex structure. Finally, it calls the glDrawElements function to render the model as a single triangle strip.

**Listing 8-2**  Submitting vertex data to a shader program

|  |
| --- |
| typedef struct \_vertexStruct |
| { |
| GLfloat position[2]; |
| GLubyte color[4]; |
| } vertexStruct; |
|  |
| void DrawModel() |
| { |
| const vertexStruct vertices[] = {...}; |
| const GLubyte indices[] = {...}; |
|  |
| glVertexAttribPointer(GLKVertexAttribPosition, 2, GL\_FLOAT, GL\_FALSE, |
| sizeof(vertexStruct), &vertices[0].position); |
| glEnableVertexAttribArray(GLKVertexAttribPosition); |
| glVertexAttribPointer(GLKVertexAttribColor, 4, GL\_UNSIGNED\_BYTE, GL\_TRUE, |
| sizeof(vertexStruct), &vertices[0].color); |
| glEnableVertexAttribArray(GLKVertexAttribColor); |
|  |
| glDrawElements(GL\_TRIANGLE\_STRIP, sizeof(indices)/sizeof(GLubyte), GL\_UNSIGNED\_BYTE, indices); |
| } |

This code works, but is inefficient. Each time DrawModel is called, the index and vertex data are copied to OpenGL ES, and transferred to the graphics hardware. If the vertex data does not change between invocations, these unnecessary copies can impact performance. To avoid unnecessary copies, your app should store its vertex data in a **vertex buffer object**(VBO). Because OpenGL ES owns the vertex buffer object’s memory, it can store the buffer in memory that is more accessible to the graphics hardware, or pre-process the data into the preferred format for the graphics hardware.

**Note:** When using vertex array objects in OpenGL ES 3.0, you must also use vertex buffer objects.

Listing 8-3 creates a pair of vertex buffer objects, one to hold the vertex data and the second for the strip’s indices. In each case, the code generates a new object, binds it to be the current buffer, and fills the buffer. CreateVertexBuffers would be called when the app is initialized.

**Listing 8-3**  Creating a vertex buffer object

|  |
| --- |
| GLuint vertexBuffer; |
| GLuint indexBuffer; |
|  |
| void CreateVertexBuffers() |
| { |
|  |
| glGenBuffers(1, &vertexBuffer); |
| glBindBuffer(GL\_ARRAY\_BUFFER, vertexBuffer); |
| glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW); |
|  |
| glGenBuffers(1, &indexBuffer); |
| glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, indexBuffer); |
| glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(indices), indices, GL\_STATIC\_DRAW); |
|  |
| } |

Listing 8-4 modifies [Listing 8-2](https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html#//apple_ref/doc/uid/TP40008793-CH107-SW2) to use the vertex buffer objects. The key difference in Listing 8-4 is that the parameters to the glVertexAttribPointer functions no longer point to the vertex arrays. Instead, each is an offset into the vertex buffer object.

**Listing 8-4**  Drawing with a vertex buffer object

|  |
| --- |
| void DrawModelUsingVertexBuffers() |
| { |
| glBindBuffer(GL\_ARRAY\_BUFFER, vertexBuffer); |
| glVertexAttribPointer(GLKVertexAttribPosition, 2, GL\_FLOAT, GL\_FALSE, |
| sizeof(vertexStruct), (void \*)offsetof(vertexStruct, position)); |
| glEnableVertexAttribArray(GLKVertexAttribPosition); |
| glVertexAttribPointer(GLKVertexAttribColor, 4, GL\_UNSIGNED\_BYTE, GL\_TRUE, |
| sizeof(vertexStruct), (void \*)offsetof(vertexStruct, color)); |
| glEnableVertexAttribArray(GLKVertexAttribColor); |
|  |
| glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, indexBuffer); |
| glDrawElements(GL\_TRIANGLE\_STRIP, sizeof(indices)/sizeof(GLubyte), GL\_UNSIGNED\_BYTE, (void\*)0); |
| } |

Buffer Usage Hints

The previous example initialized the vertex buffer once and never changed its contents afterwards. You can change the contents of a vertex buffer. A key part of the design of vertex buffer objects is that the app can inform OpenGL ES how it uses the data stored in the buffer. An OpenGL ES implementation can use this hint to alter the strategy it uses for storing the vertex data. In [Listing 8-3](https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html#//apple_ref/doc/uid/TP40008793-CH107-SW3), each call to the glBufferData function provides a usage hint as the last parameter. Passing GL\_STATIC\_DRAW into glBufferData tells OpenGL ES that the contents of both buffers are never expected to change, which gives OpenGL ES more opportunities to optimize how and where the data is stored.

The OpenGL ES specification defines the following usage cases:

* GL\_STATIC\_DRAW is for vertex buffers that are rendered many times, and whose contents are specified once and never change.
* GL\_DYNAMIC\_DRAW is for vertex buffers that are rendered many times, and whose contents change during the rendering loop.
* GL\_STREAM\_DRAW is for vertex buffers that are rendered a small number of times and then discarded.

In iOS, GL\_DYNAMIC\_DRAW and GL\_STREAM\_DRAW are equivalent. You can use the glBufferSubData function to update buffer contents, but doing so incurs a performance penalty because it flushes the command buffer and waits for all commands to complete. Double or triple buffering can reduce this performance cost somewhat. (See [Use Double Buffering to Avoid Resource Conflicts](https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/OpenGLESApplicationDesign/OpenGLESApplicationDesign.html#//apple_ref/doc/uid/TP40008793-CH6-SW5).) For better performance, use the glMapBufferRange function in OpenGL ES 3.0 or the corresponding function provided by the [EXT\_map\_buffer\_range](http://www.khronos.org/registry/gles/extensions/EXT/EXT_map_buffer_range.txt)extension in OpenGL ES 2.0 or 1.1.

If different attributes inside your vertex format require different usage patterns, split the vertex data into multiple structures and allocate a separate vertex buffer object for each collection of attributes that share common usage characteristics. Listing 8-5 modifies the previous example to use a separate buffer to hold the color data. By allocating the color buffer using the GL\_DYNAMIC\_DRAW hint, OpenGL ES can allocate that buffer so that your app maintains reasonable performance.

**Listing 8-5**  Drawing a model with multiple vertex buffer objects

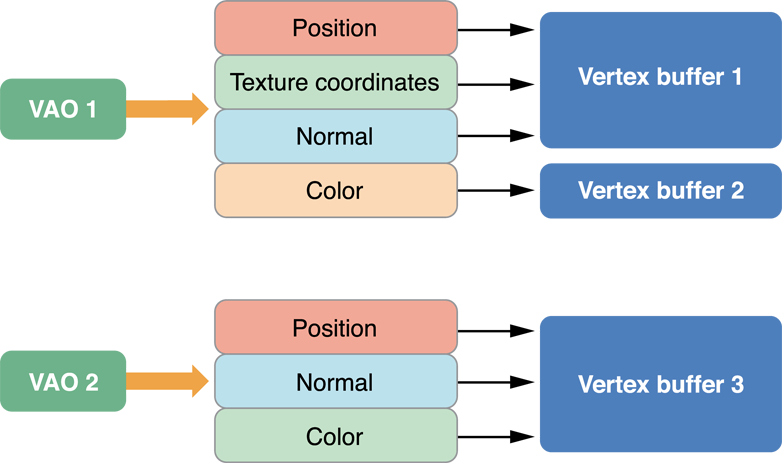
|  |
| --- |
| typedef struct \_vertexStatic |
| { |
| GLfloat position[2]; |
| } vertexStatic; |
|  |
| typedef struct \_vertexDynamic |
| { |
| GLubyte color[4]; |
| } vertexDynamic; |
|  |
| // Separate buffers for static and dynamic data. |
| GLuint staticBuffer; |
| GLuint dynamicBuffer; |
| GLuint indexBuffer; |
|  |
| const vertexStatic staticVertexData[] = {...}; |
| vertexDynamic dynamicVertexData[] = {...}; |
| const GLubyte indices[] = {...}; |
|  |
| void CreateBuffers() |
| { |
| // Static position data |
| glGenBuffers(1, &staticBuffer); |
| glBindBuffer(GL\_ARRAY\_BUFFER, staticBuffer); |
| glBufferData(GL\_ARRAY\_BUFFER, sizeof(staticVertexData), staticVertexData, GL\_STATIC\_DRAW); |
|  |
| // Dynamic color data |
| // While not shown here, the expectation is that the data in this buffer changes between frames. |
| glGenBuffers(1, &dynamicBuffer); |
| glBindBuffer(GL\_ARRAY\_BUFFER, dynamicBuffer); |
| glBufferData(GL\_ARRAY\_BUFFER, sizeof(dynamicVertexData), dynamicVertexData, GL\_DYNAMIC\_DRAW); |
|  |
| // Static index data |
| glGenBuffers(1, &indexBuffer); |
| glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, indexBuffer); |
| glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(indices), indices, GL\_STATIC\_DRAW); |
| } |
|  |
| void DrawModelUsingMultipleVertexBuffers() |
| { |
| glBindBuffer(GL\_ARRAY\_BUFFER, staticBuffer); |
| glVertexAttribPointer(GLKVertexAttribPosition, 2, GL\_FLOAT, GL\_FALSE, |
| sizeof(vertexStruct), (void \*)offsetof(vertexStruct, position)); |
| glEnableVertexAttribArray(GLKVertexAttribPosition); |
|  |
| glBindBuffer(GL\_ARRAY\_BUFFER, dynamicBuffer); |
| glVertexAttribPointer(GLKVertexAttribColor, 4, GL\_UNSIGNED\_BYTE, GL\_TRUE, |
| sizeof(vertexStruct), (void \*)offsetof(vertexStruct, color)); |
| glEnableVertexAttribArray(GLKVertexAttribColor); |
|  |
| glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, indexBuffer); |
| glDrawElements(GL\_TRIANGLE\_STRIP, sizeof(indices)/sizeof(GLubyte), GL\_UNSIGNED\_BYTE, (void\*)0); |
| } |

## Consolidate Vertex Array State Changes Using Vertex Array Objects

Take a closer look at the DrawModelUsingMultipleVertexBuffers function in [Listing 8-5](https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html#//apple_ref/doc/uid/TP40008793-CH107-SW4). It enables many attributes, binds multiple vertex buffer objects, and configures attributes to point into the buffers. All of that initialization code is essentially static; none of the parameters change from frame to frame. If this function is called every time the app renders a frame, there’s a lot of unnecessary overhead reconfiguring the graphics pipeline. If the app draws many different kinds of models, reconfiguring the pipeline may become a bottleneck. Instead, use a vertex array object to store a complete attribute configuration. Vertex array objects are part of the core OpenGL ES 3.0 specification and are available in OpenGL ES 2.0 and 1.1 through the [OES\_vertex\_array\_object](http://www.khronos.org/registry/gles/extensions/OES/OES_vertex_array_object.txt) extension.

Figure 8-7 shows an example configuration with two vertex array objects. Each configuration is independent of the other; each vertex array object can reference a different set of vertex attributes, which can be stored in the same vertex buffer object or split across several vertex buffer objects.

**Figure 8-7**  Vertex array object configuration



Listing 8-6 provides the code used to configure first vertex array object shown above. It generates an identifier for the new vertex array object and then binds the vertex array object to the context. After this, it makes the same calls to configure vertex attributes as it would if the code were not using vertex array objects. The configuration is stored to the bound vertex array object instead of to the context.

**Listing 8-6**  Configuring a vertex array object

|  |
| --- |
| void ConfigureVertexArrayObject() |
| { |
| // Create and bind the vertex array object. |
| glGenVertexArrays(1,&vao1); |
| glBindVertexArray(vao1); |
| // Configure the attributes in the VAO. |
| glBindBuffer(GL\_ARRAY\_BUFFER, vbo1); |
| glVertexAttribPointer(GLKVertexAttribPosition, 3, GL\_FLOAT, GL\_FALSE, |
| sizeof(staticFmt), (void\*)offsetof(staticFmt,position)); |
| glEnableVertexAttribArray(GLKVertexAttribPosition); |
| glVertexAttribPointer(GLKVertexAttribTexCoord0, 2, GL\_UNSIGNED\_SHORT, GL\_TRUE, |
| sizeof(staticFmt), (void\*)offsetof(staticFmt,texcoord)); |
| glEnableVertexAttribArray(GLKVertexAttribTexCoord0); |
| glVertexAttribPointer(GLKVertexAttribNormal, 3, GL\_FLOAT, GL\_FALSE, |
| sizeof(staticFmt), (void\*)offsetof(staticFmt,normal)); |
| glEnableVertexAttribArray(GLKVertexAttribNormal); |
|  |
| glBindBuffer(GL\_ARRAY\_BUFFER, vbo2); |
| glVertexAttribPointer(GLKVertexAttribColor, 4, GL\_UNSIGNED\_BYTE, GL\_TRUE, |
| sizeof(dynamicFmt), (void\*)offsetof(dynamicFmt,color)); |
| glEnableVertexAttribArray(GLKVertexAttribColor); |
|  |
| // Bind back to the default state. |
| glBindBuffer(GL\_ARRAY\_BUFFER,0); |
| glBindVertexArray(0); } |

To draw, the code binds the vertex array object and then submits drawing commands as before.

**Note:** In OpenGL ES 3.0, client storage of vertex array data is not allowed—vertex array objects must use vertex buffer objects.

For best performance, your app should configure each vertex array object once, and never change it at runtime. If you need to change a vertex array object in every frame, create multiple vertex array objects instead. For example, an app that uses double buffering might configure one set of vertex array objects for odd-numbered frames, and a second set for even numbered frames. Each set of vertex array objects would point at the vertex buffer objects used to render that frame. When a vertex array object’s configuration does not change, OpenGL ES can cache information about the vertex format and improve how it processes those vertex attributes.

## Map Buffers into Client Memory for Fast Updates

One of the more challenging problems in OpenGL ES app design is working with dynamic resources, especially if your vertex data needs to change every frame. Efficiently balancing parallelism between the CPU and GPU requires carefully managing data transfers between your app’s memory space and OpenGL ES memory. Traditional techniques, such as using theglBufferSubData function, can reduce performance because they force the GPU to wait while data is transferred, even if it could otherwise be rendering from data elsewhere in the same buffer.

For example, you may want to both modify a vertex buffer and draw its contents on each pass through a high frame rate rendering loop. A draw command from the last frame rendered may still be utilizing the GPU while the CPU is attempting to access buffer memory to prepare for drawing the next frame—causing the buffer update call to block further CPU work until the GPU is done. You can improve performance in such scenarios by manually synchronizing CPU and GPU access to a buffer.

The glMapBufferRange function provides a more efficient way to dynamically update vertex buffers. (This function is available as core API in OpenGL ES 3.0 and through the[EXT\_map\_buffer\_range](http://www.khronos.org/registry/gles/extensions/EXT/EXT_map_buffer_range.txt) extension in OpenGL ES 1.1 and 2.0.) Use this function to retrieve a pointer to a region of OpenGL ES memory, which you can then use to write new data. The glMapBufferRange function allows mapping of any subrange of the buffer’s data storage into client memory. It also supports hints that allow for asynchronous buffer modification when you use the function together with a OpenGL sync object, as shown in Listing 8-7.

**Listing 8-7**  Dynamically updating a vertex buffer with manual synchronization

|  |
| --- |
| GLsync fence; |
| GLboolean UpdateAndDraw(GLuint vbo, GLuint offset, GLuint length, void \*data) { |
| GLboolean success; |
|  |
| // Bind and map buffer. |
| glBindBuffer(GL\_ARRAY\_BUFFER, vbo); |
| void \*old\_data = glMapBufferRange(GL\_ARRAY\_BUFFER, offset, length, |
| GL\_MAP\_WRITE\_BIT | GL\_MAP\_FLUSH\_EXPLICIT\_BIT | |
| GL\_MAP\_UNSYNCHRONIZED\_BIT ); |
|  |
| // Wait for fence (set below) before modifying buffer. |
| glClientWaitSync(fence, GL\_SYNC\_FLUSH\_COMMANDS\_BIT, |
| GL\_TIMEOUT\_IGNORED); |
|  |
| // Modify buffer, flush, and unmap. |
| memcpy(old\_data, data, length); |
| glFlushMappedBufferRange(GL\_ARRAY\_BUFFER, offset, length); |
| success = glUnmapBuffer(GL\_ARRAY\_BUFFER); |
|  |
| // Issue other OpenGL ES commands that use other ranges of the VBO's data. |
|  |
| // Issue draw commands that use this range of the VBO's data. |
| DrawMyVBO(vbo); |
|  |
| // Create a fence that the next frame will wait for. |
| fence = glFenceSync(GL\_SYNC\_GPU\_COMMANDS\_COMPLETE, 0); |
| return success; |
| } |

The UpdateAndDraw function in this example uses the glFenceSync function to establish a synchronization point, or fence, immediately after submitting drawing commands that use a particular buffer object. It then uses the glClientWaitSync function (on the next pass through the rendering loop) to check that synchronization point before modifying the buffer object. If those drawing commands finish executing on the GPU before the rendering loop comes back around, CPU execution does not block and the UpdateAndDraw function continues to modify the buffer and draw the next frame. If the GPU has not finished executing those commands, the glClientWaitSync function blocks further CPU execution until the GPU reaches the fence. By manually placing synchronization points only around the sections of your code with potential resource conflicts, you can minimize how long the CPU waits for the GPU.