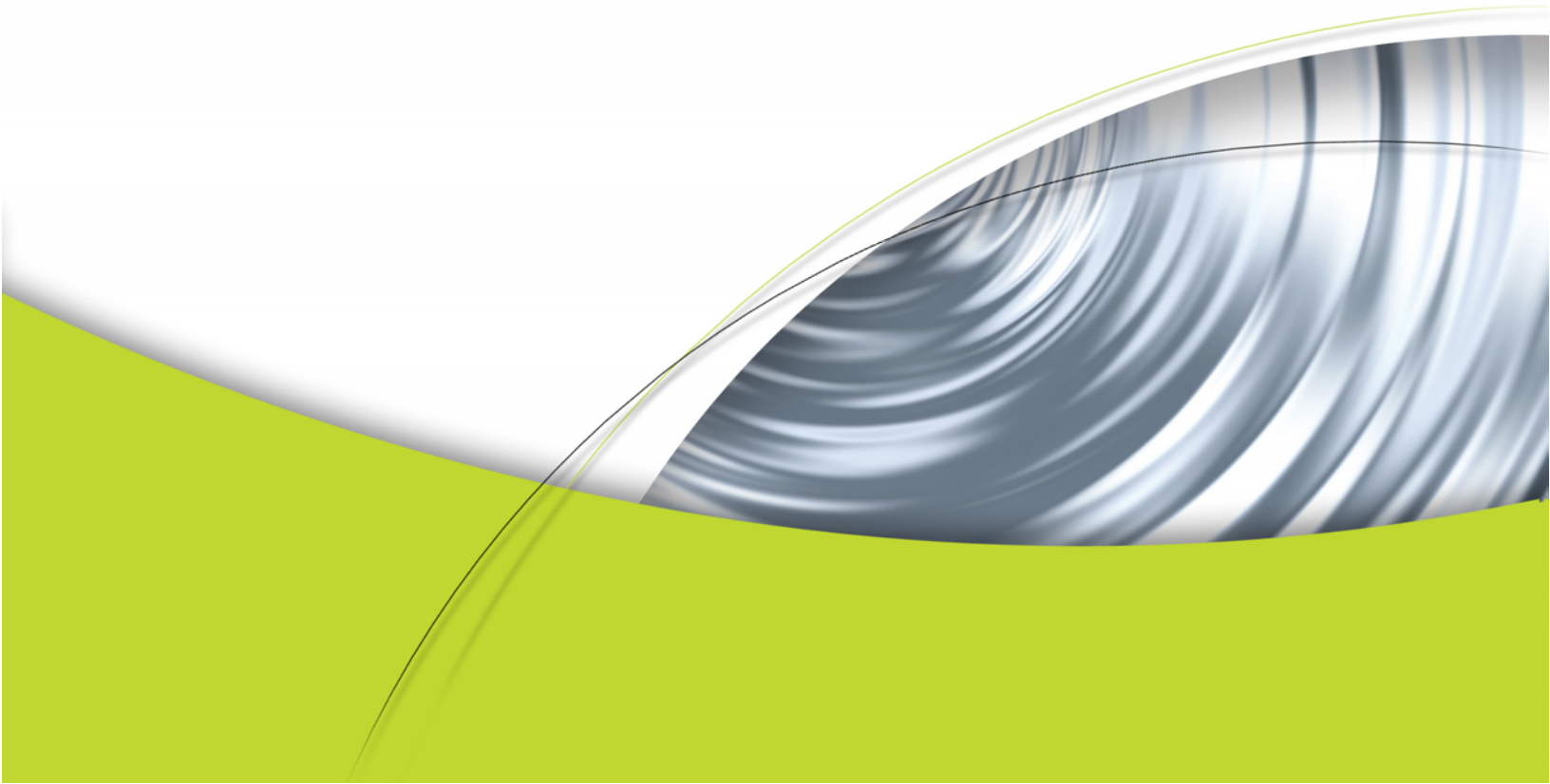
White Paper

Using Vertex Buffer Objects (VBOs)



Document Change History

Version	Date	Responsible	Reason for Change
01	10/16/03	CK	Initial release

Using Vertex Buffer Objects (VBOs)

Overview

A vertex buffer object (VBO) is a powerful feature that allows us to store certain data in high-performance memory on the server side.

This feature proposes a mechanism—encapsulating data within "buffer objects"—for handling these data without having to take them out from the server side, thereby increasing the rate of data transfers.

VBOs help with:

- Any data that would be pointed to by a client/state function. Typically we're talking about glVertexPointer(), glColorPointer(), glNormalPointer(), and so on.
- Arrays of indices for drawing a set of elements (glDraw[Range]Elements()).

The basic idea of this mechanism is to provide some chunks of memory (buffers) that will be available through identifiers. As with any display list or texture, we can bind such a buffer so that it becomes active.

This binding operation turns every pointer in every client/state function into offsets, because we will work in a memory area that is relative to the current bound buffer. In other words, this extension turns a client/state function into a server/state function.

We all know that a client/state function deals with data whose scope is only accessible for the client itself. Another client won't be able to access any of this data. As a consequence of passing these functions on the server's side, it is now possible to share this data between various clients. Many clients will be able to bind common buffers, and everything is dealt with just like texture or display list identifiers.

Problems with VAR

The previous extension dedicated to this kind of task was the vertex array range (VAR). Although this extension is still available, we advocate you use VBOs instead.

VAR is fully functional, but raises some issues that developers tend to dislike:

- It breaks the server/client paradigm, because the client takes control of the memory management (server's side).
- It doesn't provide an internal memory management; the only thing VAR provides is the ability to allocate a big chunk of memory in the server's memory.
- When allocating buffers for VAR, the developer needs to specific whether to use AGP, system, or video memory, which can be troublesome.
- Developers still need to create their own memory allocator to optimize the chunk of memory they borrowed for VAR.
- Efficient memory management had to go through the semaphore-like Fence system.

To simplify things, we can say that VBO now handles a kind of VAR and memory management for you (Figure 1).

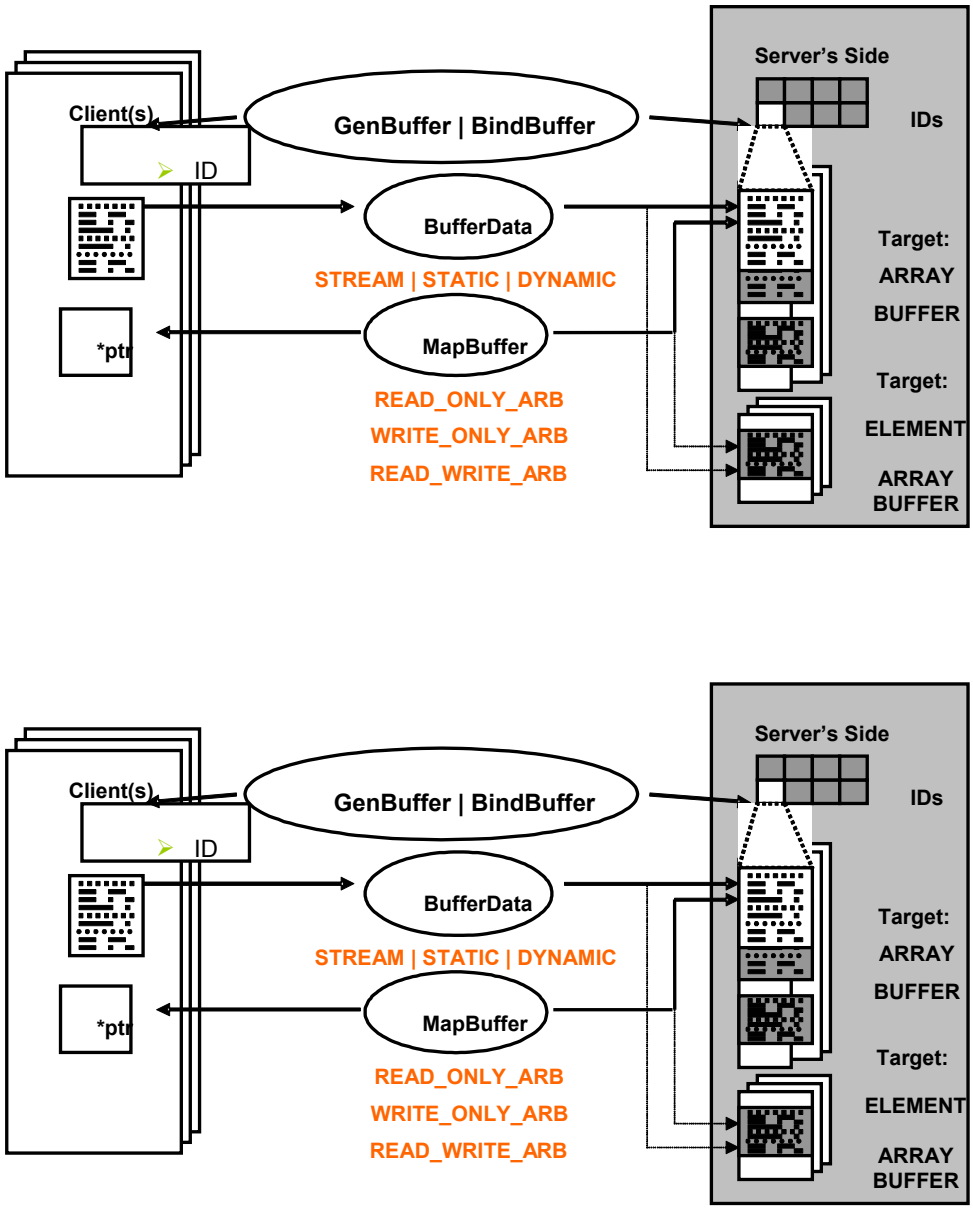


Figure 1. Using VBOUsing VBO

Memory Management

OpenGL has acquired the same level of functionality in the AGP/video memory management as Direct3D provides for vertex arrays. The way it works in OpenGL

is very similar: It has the ability to map the buffer in memory, thereby defining the usage of various buffers, mappings, and unmappings (Lock/Unlock in Direct3D).

Internal memory management can choose the best type of memory (system, video, or AGP), depending on the way we want to use the buffers.

VBO provides different ways to interact with buffer objects.

- **Binding the buffer (glBindBuffer):** This allows client/state functions to work in this area of memory instead of working in absolute memory on the client side. Binding the buffer #0 switches off VBO, so we go back to the usual client state mode with absolute pointers.
- **Using the VBO API to load data in these buffers (glBufferData, glBufferSubData, and glGetBufferSubData):** These functions let you make copies between a client's area and a buffer object in the server side.
- **Using the technique of buffer mapping (glMapBuffer and glUnmapBuffer):** This works similar to Direct3D's Lock and Unlock. You get a temporary pointer as an entry to the beginning of the buffer, meaning the buffer is mapped in the client's memory. OpenGL is responsible for how this mapping into the client's absolute memory is done. For this reason, mapping must be done for a short operation and the pointer must not be stored for further use.

Targets

VBO works with these two targets:

- **Array buffers (ARRAY_BUFFER_ARB):** These buffers contain vertex attributes, such as vertex coordinates, texture coordinates data, per vertex-color data, and normals. You can either interleave this data (using the stride parameter), or write one array after the other one (write 1,000 vertices, then 1,000 normals, and so on). glVertexPointer and glNormalPointer (and so on) have to point to the right offsets.
- **Element array buffers (ELEMENT_ARRAY_BUFFER_ARB):** This type of buffer is used mainly for the element pointer in glDraw[Range]Elements(). It may contain only indices of elements.

These two targets are set up in parallel because element arrays must be available at the same time as array buffers in glDraw[Range]Elements().

An interesting point about using these targets is the ability to switch between various element buffers while keeping the same vertex array buffer. We may be able to implement LOD or any other effect by changing the elements table while working on the same database of vertices.

A Few Words About PBO

Another extension that is supposed to add more targets to a VBO is ARB_pixel_buffer_object (PBO).

Although it isn't available in our Release 50.xx drivers, this extension will allow us to work on textures, frame buffers, and offscreen buffers by adding two new targets.

In other words, it will provide the same mechanism for vertex, normals, elements, and so on, but for arrays of bytes.

The two new targets are

- **PIXEL_PACK_BUFFER:** This target will bring a buffer for various read operations, such as glReadPixels and glGetTexImage. These commands will write their data into the currently bound buffer object.
- **PIXEL_UNPACK_BUFFER:** This target will bring a buffer for various write operations, such as glBitmap, glDrawPixels, and glTexImage2D. These commands will read their data from a buffer object.

Some interesting optimizations can be done when mixing VBOs with PBOs (Figure 2):

- **Render to vertex array:** If in a first pass we intend to create a special vertex array (for skinning, displacement, and so on), we can avoid copying the pbuffer on the client's side and putting it back on the server's side as an input for a vertex program. VBO/PBO would keep all the data flow inside the server.
- **Streaming textures:** This operation is similar to what we would do with Pixel Data Range (PDR); we would use MapBuffer/UnmapBuffer to change the texture data, depending on a video stream, then call TexSubImage to update the texture.

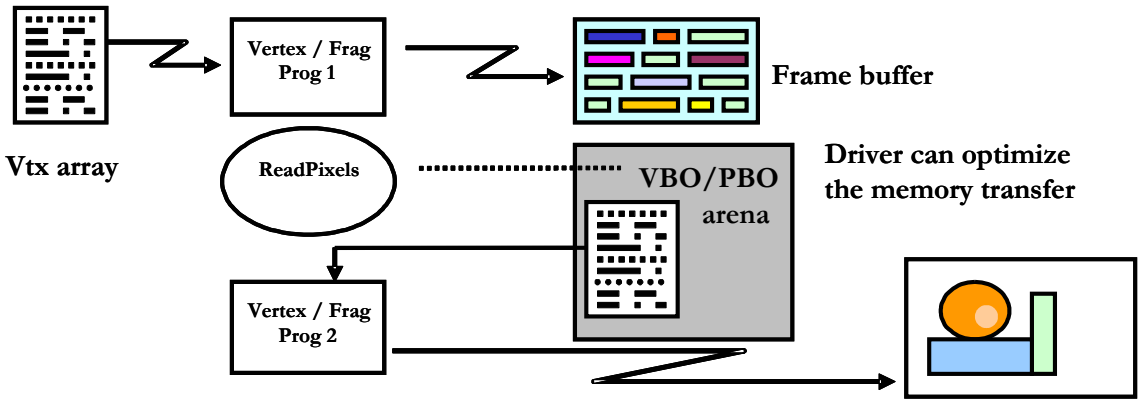


Figure 1. Example of VBO/PBO combination

New Procedures, Functions, and Token

Usage Flags

- STREAM_DRAW_ARB
- STREAM_READ_ARB
- STREAM_COPY_ARB
- STATIC_DRAW_ARB
- STATIC_READ_ARB
- STATIC_COPY_ARB
- DYNAMIC_DRAW_ARB
- DYNAMIC_READ_ARB
- DYNAMIC_COPY_ARB

Access Flags

- READ_ONLY_ARB
- WRITE_ONLY_ARB
- READ_WRITE_ARB

Targets

- ARRAY_BUFFER_ARB
- ELEMENT_ARRAY_BUFFER_ARB

void BindBufferARB(enum target, uint buffer):

The BindBufferARB function is used to bind a buffer ID as the actual buffer to use. It switches off the use of buffers if the ID is zero.

void *MapBufferARB(enum target, enum access);
boolean UnmapBufferARB(enum target);

The function MapBufferARB provides a pointer corresponding to the mapped area of the current buffer object. UnmapBufferARB releases the mapping.

void BufferDataARB(enum target, sizeiptrARB size, const void *data, enum usage);

The BufferDataARB function can be used two ways:

- To simply set up the amount and usage of memory for the current buffer object with data set to NULL. In this case, you can map the buffer later in order to set up its data.
- To allocate memory, set the usage, and copy some data; typically used when dealing with a static memory model.

void BufferSubDataARB(enum target, intptrARB offset, sizeiptrARB size, const void *data);

The BufferSubDataARB function copies data in a specific range inside the buffer object.

void GetBufferSubDataARB(enum target, intptrARB offset, sizeiptrARB size, void *data);

The GetBufferSubDataARB function retrieves some data from a specific range in the current buffer object.

void DeleteBuffersARB(sizei n, const uint *buffers);

void GenBuffersARB(sizei n, uint *buffers);

boolean IsBufferARB(uint buffer);

These three functions are similar to any display list/textures identifiers; they can allocate, free, or query some identifiers for buffer objects.

void GetBufferParameterivARB(enum target, enum pname, int *params);

The GetBufferParameterivARB function returns various parameters concerning the current buffer object. Pname can be:

- **BUFFER_SIZE_ARB:** Returns the size of the buffer object.
- **BUFFER_USAGE_ARB:** Returns the usage of the buffer object.
- **BUFFER_ACCESS_ARB:** Returns the access flag of the buffer object.
- **BUFFER_MAPPED_ARB:** Tells you if we mapped this buffer.

void GetBufferPointervARB(enum target, enum pname, void **params);

The GetBufferPointervARB function returns the actual pointer of the buffer if it has been mapped (MapBufferARB). Pname can only be BUFFER_MAP_POINTER_ARB for this time.

Tokens for Get{Boolean, Integer, Float, Double}v

The buffer object ID zero is reserved, and when buffer object zero is bound to a given target, the commands affected by that buffer binding behave normally. When a nonzero buffer ID is bound, then the pointer represents an offset and will go through VBO management.

You can use these token to know which buffers are bound as VBO offsets.

- ARRAY_BUFFER_BINDING_ARB
- ELEMENT_ARRAY_BUFFER_BINDING_ARB
- VERTEX_ARRAY_BUFFER_BINDING_ARB
- NORMAL_ARRAY_BUFFER_BINDING_ARB
- COLOR_ARRAY_BUFFER_BINDING_ARB
- INDEX_ARRAY_BUFFER_BINDING_ARB
- TEXTURE_COORD_ARRAY_BUFFER_BINDING_ARB
- EDGE_FLAG_ARRAY_BUFFER_BINDING_ARB

- SECONDARY_COLOR_ARRAY_BUFFER_BINDING_ARB
- FOG_COORDINATE_ARRAY_BUFFER_BINDING_ARB
- WEIGHT_ARRAY_BUFFER_BINDING_ARB

Token for GetVertexAttribivARB:

When working with VBOs and vertex programs, some attributes can have arbitrary meanings: an array of normals for example may be used to store information other than just normals. Instead of using token from the previous section, you may want to use the index of the attribute. This token allows you to query which attribute number is being used by VBOs through an offset system.

- VERTEX_ATTRIB_ARRAY_BUFFER_BINDING_ARB

Purposes of Various Functions

glBufferDataARB()

This function is an abstraction layer between the memory and the application. But behind each buffer object is a complex memory management system.

Basically, the function does the following:

- Checks whether the size or usage type of the data store has changed.
- If the size is zero, frees the memory attached to this buffer object.
- If the size and storage type didn't change and if this buffer isn't used by the GPU, we'll use it. Everything is already set up for use.
- On the other hand, if the GPU is using it or is about to use it, or if the storage type changed, we'll have to request another chunk of memory for this buffer to be ready.
- If the data pointer isn't NULL, we'll copy the data into this new memory area.

  We can see that the memory we had before a second call to BufferDataARB isn't necessarily the same exact memory we had afterward. However, it's still the same from the application's point of view (same buffer object). But on the driver's side, *we're optimizing and allowing the application to not wait for the GPU*.

  Internally, we've allocated a large pool of memory that we suballocate from. When we call BufferDataARB, we reserve a chunk of it for the current buffer object. Then we fill it with data and draw with it, and we mark that memory as being used (similar to the glFence function) .

  If we call BufferDataARB again before the GPU is done, we can simply assign the buffer object a new chunk of the large pool. This is possible because BufferDataARB says we're going to re-specify all the data in the buffer (as opposed to BufferSubDataARB).

Usage Flag

The usage argument is a key value for helping the VBO memory manager fully optimize your buffers.

Name of Flag	Definition
STATIC_…	Assumed to be a 1-to-n update-to-draw. Means the data is specified once (during initialization).
DYNAMIC_…	Assumed to be a n-to-n update-to-draw. Generally, it means data that is updated frequently, but is drawn multiple times per update. For example, this is any dynamic data that is updated every few frames or so.
STREAM_…	Assumed to be a 1-to-1 update-to-draw. Can be thought of as data that is updated about once each time it's drawn. STREAM is like DYNAMIC: Data will be changed over time. However, data is expected to change all the time, so it may end up somewhere more volatile (like video memory…), where if it disappears (during a modeswitch for instance), it'll be quickly replaced.
…_READ_…	Means we must have an easy access to read the data: AGP or system memory would be fine.
…_COPY_…	Means we are about to do some _READ_ and _DRAW_ operations.
…_DRAW_…	Means the buffer will be used for sending data to the GPU. We may want to use the video here (for STATIC|STREAM _DRAW_ARB,) or AGP (DYNAMIC _DRAW_ARB) memory.

Table 1. List of Usage Flags

This combination of memory usage can help the memory manager balance between three kinds of memory: system, AGP and video. On the other hand, it has to figure out how areas of memory can be recycled for the other buffers. STATIC, STREAM, and DYNAMIC are here for that purpose.

But STATIC, STREAM, and DYNAMIC are simply suggestions or hints about potential usage patterns. They do not force the driver to do anything in particular, but they help us make decisions about buffer memory placement and mapping behavior. We can assume that the data allocated and used will always be available, until you expressly release it by either deleting the buffer (a heavy weight approach if you plan to create another one in its place) or by calling BufferDataARB(…, NULL, …);. The latter method is preferred.

On the client's side, these are not hard restrictions! They are suggestions that help us decide where to put the data and how to manage it. Nothing prevents you from creating a STATIC data store and then updating it every frame. Nor is there any reason you can't create a STREAMING one and never modify it. However, we strongly discourage this kind of behavior.

glBufferSubDataARB()

This function gives you a way to replace a range of data into an existing buffer. Note that in order to avoid conflicts, we may have to wait for the GPU if ever the GPU is working with this area. As a consequence there could be a loss of performance.

glBindBufferARB()

This sets up the internal parameters so that the next operations on vertex arrays, or any other VBO functions, work on this current buffer object. Note that this binding operation is cheap: it is a kind of prebinding, waiting for other operations to seriously change internal states of the VBO manager.

glMapBufferARB()

This function maps the buffer object into the client's memory, depending on the access flag. In the best case, there isn't any data transfer: the driver may just "reveal" the actual pointer into the AGP or system memory.

In other cases, some data transfers may happen if it isn't possible to satisfy the access flag. For example, asking a pointer to read a buffer that may be in video memory requires the driver to demote this buffer to AGP or system memory.

Using an access flag prepares the memory to be as efficient as it can, depending on the work we want to do with it. For example, we can afford to read a mapped buffer with a WRITE_ONLY flag. It is only a hint for the driver, not a restriction. The driver is, however, allowed to be very slow if we try to read from a WRITE_ONLY buffer (writing would still be fast).

glVertexPointer()

This function sets up the offset (originally a pointer), depending on the current buffer object. A big part of the job in the VBO memory management is done here.

Caveats in VBO

Here are some ideas for keeping the VBO efficient.

glBufferDataARB() with glMapBufferARB()

Sometimes we know that we'll update all data in a buffer object, and we don't need to retrieve the old values from the buffer. This is typically what happens when we are call glBufferData(), as mentioned before.

However, we may want to use glMapBuffer() in order to update the whole set of data. Unfortunately this operation is more expensive than glBufferData(). We must keep in mind that the driver cannot guess what we want to do with the memory pointer returned by glMapBuffer(): will we just change a few bytes, or will we update the whole buffer?

The pointer returned by with glMapBuffer() refers to the actual location of the data. It is possible that the GPU could be working with these data, so requesting it for an update will force the driver to wait for the GPU to finish its task.

To solve this conflict we you just need to call glBufferDataARB() *with a NULL pointer*. Then calling call glMapBuffer() tells the driver that the previous data are aren't valid. As a consequence, if the GPU is still working on them, there won't be a conflict because we invalidated these data. The function glMapBuffer() returns a new pointer that we can use *while the GPU is working on the previous set of data.*.

Note that Microsoft® Direct3D® solves this problem by via the D3DLOCK_DISCARD providing a flag parameter to the Lock() method.

Avoid Calling glVertexPointer() more than once per VBO

The "glVertexPointer" function does a lot of setup in VBO, so to avoid redundancy.

The most efficient way to do is to bind the VBO buffer, setup various array pointers (glNormalPointer etc) and then call glVertexPointer(). glVertexPointer should be called one time for one VBO.

You might think the essentials of VBO management are done in glBindBufferARB(), but it's the opposite. VBO systems wait for the next upcoming important function (like glVertexPointer).

The binding operation is cheap compared to the setup of various pointers.

This advice fits any other function working in the same manner as glVertexPointer().

Use "First" in glDrawArrays() Value Instead of Changing glVertexPointer

In the function:

glDrawArrays (GLenum mode, GLint first, GLsizei count);

Instead of changing glVertexPointer() to a specific offset and leaving 'first' to NULL, it is more efficient to change the 'first' argument of glDrawArrays. As we said previously, this will prevent having to go through the VBO manager and performing an additional setup step.

Use glDrawRangeElements Instead of glDrawElements

Using range elements is more efficient for two reasons:

- If the specified range can fit into a 16-bit integer, the driver can optimize the format of indices to pass to the GPU. It can turn a 32-bit integer format into a 16-bit integer format. In this case, there is a gain of 2×.
- The range is precious information for the VBO manager, which can use it to optimize its internal memory configuration.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com