Vertex Buffer Object

Christophe [Groove] Riccio
www.g-truc.net
Creation: 01 Mai 2006
Revision 1: 30 June 2006
Revision 2: 28 September 2006

Contents

Introduction

Vertex buffer object (VBO) is a new method used to describe geometric primitives. It had been introduced by NV_vertex_array_range and ATI_vertex_array_object extensions promoted to ARB_vertex_buffer_object and finally integrated in OpenGL 1.5 specification. VBO should be the only way to describe geometric primitives with OpenGL LM.

OpenGL LM is the ARB project about simplification of OpenGL API for making easier graphics card drivers optimisations. Its goal is to compete with Direct Graphics in video games area and it would be based on OpenGL 3.0 specifications. ARB wishes to simplify OpenGL API that many developments had hoped for OpenGL 2.0. As a result, immediate mode, vertex arrays and specific cases as glRect should disappear.

VBOs are supported by nVidia TNT and first ATI Radeon. However, all features aren't always available and even some are supported by any today's graphic card.

Features, efficiency and longevity are three reasons to use VBOs as soon as possible.

This document is published with a sample programs available at the following addresses:
www.g-truc.net/article/vbo.zip (9 Mo)
www.g-truc.net/article/vbo.7z (2.5 Mo)

1. Description

1.1. OpenGL legacy

OpenGL beginners like its simplicity through the immediate mode that comes from OpenGL 1.0. This mode presents like this:

```
glBegin(GL_QUADS);
      glColor3f(1.0f, 0.5f, 0.0f);
      glVertex2f(0.0f, 0.0f);
      glVertex2f(1.0f, 0.0f);
      glVertex2f(1.0f, 1.0f);
      glVertex2f(0.0f, 1.0f);
glEnd();
```

This code is quiet easy to understand but it could become very slow whether the goal is to describe complex geometric primitive composed by thousand of vertices just because of the high number of functions calls.

OpenGL 1.1 includes vertex arrays that allow rendering a large amount of data with few function calls. This mode looks like this:

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);

float ColorArray[] = {…};
float VertexArray[] = {…};

glColorPointer(3, GL_FLOAT, 0, ColorArray);
glVertexPointer(3, GL_FLOAT, 0, VertexArray);

glDrawArrays(GL_TRIANGLES, 0, sizeof(VertexArray) / sizeof(float));

glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

Vertex buffer objects are similar in their principles but they provide lots of improvements.

1.2. VBOs interests

VBO provides 3 data transfer modes instead of one for vertex arrays.

The first one called ***GL_STREAM_DRAW*** provide the same behaviours than vertex arrays witch means that data are sand for each called of ***glDrawArrays***, ***glDrawElements***, ***glDrawRangeElements*** (EXT_draw_range_elements, OpenGL 1.2) ***glMultiDrawArrays*** or ***glMultiDrawElements*** (EXT_multi_draw_array, OpenGL 1.4). This mode is particularly efficient for animated object as characters for examples.

A second mode called ***GL_STATIC_DRAW*** inherit of features from the EXT_compiled_vertex_array extension. It allows sending vertices information only one time to the graphics card because these data are saved in the graphic card memory. This mode is suitable for non deformable objects that are to say for all geometries witch rest the same during several frames. This mode is the one witch could provide the higher performances.

The last mode is ***GL_DYNAMIC_DRAW***. With this mode, graphics card drivers are in charge of the choice of data location. This mode is recommended for animated object rendered several times per frame, for example in case of multi passes rendering.

To summarize, use the mode:
- ***GL_STREAM_DRAW*** when vertices data could be updated between each rendering.
- ***GL_DYNAMIC_DRAW*** when vertices data could be updated between each frames.
- ***GL_STATIC_DRAW*** when vertices data are never or almost never updated

Notice that these three modes are more and more considered by the graphics drivers as a recommendation and sometime they will choose for you without taking care of your advice. Consequently, it could happen that you don't observed frame rate variations when you are toggling modes.

The second improvement that characterise VBO sis called Vertex mapping. It represents the ability of does't using temporary array to store vertices data but directly an array allocated by OpenGL. Take notice that VBOs are also available with OpenGL ES 2.0 but this feature is optional. This feature comes from ATI_map_buffer_object extension available since first ATI Radeon. nVidia has also define an extension for this feature witch is called NV_vertex_array_range however the memory allocation is manually done using the OpenGL context… With VBOs this allocation is transparent.

Finally, VBOs provide an excellent alternative to interleaved arrays. As a recalling, interleaved arrays allow sending vertices data to the graphics card using a single array decribes by the function ***glInterleavedArrays***. With vertex arrays, we must using predefined data structures with defined order for each element of this structure. With GPU programming arising, programmer custom attributes aren't managed by this solution. VBOs solve this issue and allow updated of only a part of the data.

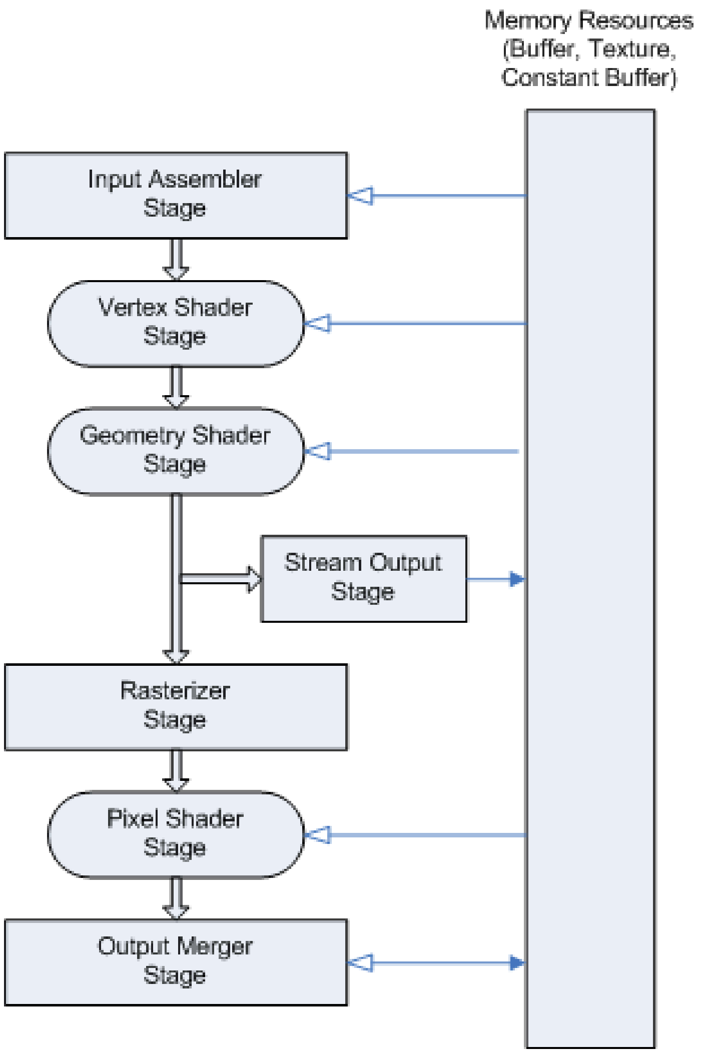
1.3. Partial support

VBOs provide a robust API that matches with needs of current and future graphics cards. Thus, it provides 6 other modes of transfer called ***GL_STREAM_READ***, ***GL_STREAM_COPY***, ***GL_DYNAMIC_READ***, ***GL_DYNAMIC_COPY***, ***GL_STATIC_READ*** and ***GL_STATIC_COPY***.

Unfortunately, no one current graphics card supports these features according my knowledge. ATI Render to Vertex Buffer (R2VB) technology from Radeon X1*00 nearly match to feature provides by this 6 modes. However, ATI hasn't communicated on the OpenGL side but it seams like Pixel Buffer Object (ARB_pixel_buffer_object) so it doesn't provide any new stuff for OpenGL.

Modes finishing by ***GL_*_READ*** allow reading data managed by OpenGL. Rules on ***GL_STREAM_**** , ***GL_DYNAMIC_**** and ***GL_STATIC_**** terms are the same but now, they deal with the number of reading of data by the program. Modes using the term ***GL_*_COPY*** allow displaying geometric from source managed by OpenGL.

In concrete terms, what's the used of these modes? For the coming graphics card generations knowed under code named G80 for nVidia and R600 for ATI, new features will be available. They are brought into disrepute under DirectX names as Shader Model 4.



Picture from DirectX 10 documentation that represents the graphic pipeline

Geometry shaders, called primitive shaders under OpenGL, will allow vertices instancing from the GPU. As the previous figure shows, the graphic pipeline will change by allowing the transfert of data, here indicated under the term “stream output stage”. This stream of data required an API, ***GL_*_COPY*** and ***GL_*_READ*** modes of VBOs are dedicated on answer to this new need. ***GL_*_COPY*** modes seams suitable to manage recursif processing on vertices witch could really increase resources required by the vertex pipelines. The marketing action by ATI with R2VB is based on an analogy with this new “stream output stage” using pixels as stream output. However there is no way for pixel creation with Radeon X1*00.

2. Practice

For each sub-part of the practice part, there is an associated class in the samples program of this document. Two others examples none described by this document show use of VBOs with GLSL and Cg. The single goal of this class implementation is to make different method interchangeable.

2.1. VBO basic use as Vertex Array method (class CTest1)

To make ease the understanding, let start by an example witch match with vertex arrays features. VBOs have a similar API of texture objects for their management.

```
GLvoid glGenBuffers(GLsizei n, GLuint* buffers);
GLvoid glDeleteBuffers(GLsizei n, const GLuint* buffers);
```

buffers is an array created by the user within are store VBOs identifiers. ***n*** objects are created or deleted so take care of ***buffers*** size.

Admit that we wish to render a quad on the screen using two triangles. Are sources could be for example:

```
static const GLsizeiptr PositionSize = 6 * 2 * sizeof(GLfloat);
static const GLfloat PositionData[] =
{
	-1.0f,-1.0f,
	 1.0f,-1.0f,
	 1.0f, 1.0f,
	 1.0f, 1.0f,
	-1.0f, 1.0f,
	-1.0f,-1.0f,
};

static const GLsizeiptr ColorSize = 6 * 3 * sizeof(GLubyte);
static const GLubyte ColorData[] =
{
	255,   0,   0,
	255, 255,   0,
	  0, 255,   0,
	  0, 255,   0,
	  0,   0, 255,
	255,   0,   0
};
```

We are using two VBOs to render this 6 vertices described by the previous array. Arrays are identified by ***POSITION_OBJECT*** and ***COLOR_OBJECT***. Creation and destruction of VBOs are done using ***glGenBuffers*** and ***glDeleteBuffers*** functions. The function ***glBindBuffer*** allows selecting active VBO.

```
static const int BufferSize = 2;
static GLuint BufferName[BufferSize];

static const GLsizei VertexCount = 6;

enum
{
    POSITION_OBJECT = 0,
    COLOR_OBJECT = 1
};
```

The C++ code to render this quad is:

```
glBindBuffer(GL_ARRAY_BUFFER, BufferName[COLOR_OBJECT]);
glBufferData(GL_ARRAY_BUFFER, ColorSize, ColorData, GL_STREAM_DRAW);
glColorPointer(3, GL_UNSIGNED_BYTE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, BufferName[POSITION_OBJECT]);
glBufferData(GL_ARRAY_BUFFER, PositionSize, PositionData, GL_STREAM_DRAW);
glVertexPointer(2, GL_FLOAT, 0, 0);

glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);

glDrawArrays(GL_TRIANGLES, 0, VertexCount);

glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

glBufferData initialises data storage of VBOs. The last parameter specifies VBO usage as presented in section 1.2 and 1.3. The list of all usages is available in section 3.1. Functions ***glColorPointer*** and ***glVertexPointer*** allows specifying the location where OpenGL will respectively find colours and spatial coordinates of the vertices.

The order of these three types of function is particularly important. It is intuitive that we must first select the active VBO for its setup, however the order of ***glBufferData*** and ***gl*Pointer*** is important as well. Actually, ***gl*Pointer*** indicated the data sources when a VBO is active, the source is the one described by ***glBufferData***.

Remark:
- Often, we will prefer to separate the task of loading vertices data and the one of data description for flexible issues of uses. The following solution is perfectly correct:

```
glBindBuffer(GL_ARRAY_BUFFER, BufferName[POSITION_OBJECT]);
glBufferData(GL_ARRAY_BUFFER, PositionSize, PositionData, GL_STREAM_DRAW);
...
glBindBuffer(GL_ARRAY_BUFFER, BufferName[POSITION_OBJECT]);
glVertexPointer(3, GL_FLOAT, 0, 0);
```

- When the function ***glBindBuffer*** is called with a valide VBOs name, OpenGL toggle in VBO mode. To come back in vertex array mode, we have to use ***glBindBuffer*** with the value 0 as object name.

The rendering is carried out by one of the functions dedicated to array rendering: ***glDrawArrays*** or ***glMultiDrawArrays***.

In the specific case where the whole memory size of the graphic card is lower that the size we are asking to reserve for a single VBO, a ***GL_OUT_OF_MEMORY*** error is thrown and could be retrieving with the common function ***glGetError***.

2.2. Indexed arrays (class CTest2)

During the first example we had used the target ***GL_ARRAY_BUFFER***. It is used for all types of data excepted index arrays witch have their dedicated target, ***GL_ELEMENT_ARRAY_BUFFER***.

Index array initialisation is done by the following code:

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, BufferName[INDEX_OBJECT]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, IndexSize, IndexData, GL_STREAM_DRAW);
```

It’s the part og VBO API that differ the most from vertex arrays. Essentially, it is useless to call ***glIndexPointer*** or activate ***GL_INDEX_ARRAY*** state. If the rendering function by element is used with the null value instead of a pointer to an index array then the active VBO with the target ***GL_ELEMENT_ARRAY_BUFFER*** is used as source of indexes.

The rendering is carried out with one of the function dedicated to index arrays: ***glDrawElements***, ***glDrawRangeElements*** or ***glMultiDrawElements***.

2.3. Interleaved arrays alternative (class CTest3)

Update of the function ***glInterleavedArrays*** [3.4.2] never happen since it has been included in OpenGL 1.1. This function was often used for interleaved arrays but even if we still could used it with VBOs, there is a better option based on ***gl*Pointer*** functions. The principle is to specify for each attribute of the interleaved array the source of the data using the ***stride*** parameter.

```
#pragma pack(push, 1)
struct SVertex
{
    GLubyte r;
    GLubyte g;
    GLubyte b;
    GLfloat x;
    GLfloat y;
};
#pragma pack(pop)

glBindBuffer(GL_ARRAY_BUFFER, BufferName);
glBufferData(GL_ARRAY_BUFFER, VertexSize, VertexData, GL_STREAM_DRAW);

glColorPointer(3, GL_UNSIGNED_BYTE, sizeof(SVertex),
BUFFER_OFFSET(ColorOffset));
glVertexPointer(2, GL_FLOAT, sizeof(SVertex), BUFFER_OFFSET(VertexOffset));

glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);

glDrawArrays(GL_TRIANGLES, 0, VertexCount);

glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

For this sample, we are using a structure witch allow us to interleave vertex data. The structure previously defined is surrounded by the standard pre-processor instructions ***#pragma pack***. In fact, if we get the size of this structure with the sizeof instruction, there are all chances that the returned value is 12 or maybe 16 octets instead of 11 in this case. The common size of GLfloat is usually 4 bytes and the one of GLubyte is usually 1 byte so the required size if 11 bytes. However, compilers align data in memory because processors are optimized to handle data of the size of their registers: 4 bytes for 32 bits CPUs and 8 bytes for 64 bits CPUs. That's a good initiative but when the memory space is expensive it's sometime better to forget this optimisation. Actually, in our case, the aligned structure cost 1/12 of extra memory but also 1/12 of extra data transfer to the graphic card. Finally, further problem could happen about the management of additional bytes, especially in this case. Where are they?

glColorPointer and ***glVertexPointer*** functions must still indicated the sources and the types of the data stored by the VBO. To proceed, specifications suggest a macro called ***BUFFER_OFFSET***:

```
#define BUFFER_OFFSET(i) ((char*)NULL + (i))
```

With VBOs, the purpose isn't to give the address of the data source because the source is stored somewhere by the VBO. An offset indicate where the OpenGL drivers is suppose to start reading the data in the VBO memory. ***sizeof(SVertex)*** is in the sample the ***stride*** value, that is to say it indicates the number of bytes between two vertices for a same attribute. Usually, this value is null to simplify the OpenGL API. Null means that the values are adjoining, the VBO only contains a single attribute by vertex and no one space. Consequently, if we create an array that only contains the spatial 3D coordinates of vertices, then the following function calls are equivalent:

```
glVertexPointer(3, GL_FLOAT, 0, 0);
glVertexPointer(3, GL_FLOAT, sizeof(float) * 3, 0);
```

The macro ***BUFFER_OFFSET*** also allows preventing a warning about a conversion of an integer to a pointer.

2.4. Serialized arrays (classe CTest4)

For lot of cases, data used to describe geometric primitive have no reason to be interleaved and it could even become a bad choice. It often happens that we just want to update a part of the data of each vertex. For example, with the case of animated meshes, as a personage, texture coordinates never need to be updated but vertex positions and vertex normals will be.

As a result, we just have a single VBO within we insert many types of data using the function ***glBufferSubData***.

First, we have to reserve memory space for the full data storage with the function ***glBufferData***. We don't have to pass the data source in the third parameter anymore; instead we use the value 0.

Then, we use the function ***glBufferSubData*** to fill the array. The second parameter is the VBO data offset. The third one indicated the size of the source data we want to add and the last one is the data source itself.

```
glBindBuffer(GL_ARRAY_BUFFER, BufferName);
glBufferData(GL_ARRAY_BUFFER, ColorSize + PositionSize, 0, GL_STREAM_DRAW);

glBufferSubData(GL_ARRAY_BUFFER, 0, ColorSize, ColorData);
glBufferSubData(GL_ARRAY_BUFFER, ColorSize, PositionSize, PositionData);

glColorPointer(3, GL_UNSIGNED_BYTE, 0, 0);
glVertexPointer(2, GL_FLOAT, 0, BUFFER_OFFSET(ColorSize));

glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);

glDrawArrays(GL_TRIANGLES, 0, VertexCount);

glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
```

Finally, the ***glBufferSubData*** function could be used for update of just a par of the whole data, for example in case of partial animated models or if more than one model is stored in the array which could be very efficient.

2.5. Vertex mapping (classe CTest5)

In some cases, we would like to avoid the use of an intermediate array to store geometry data. This could accelerate the rendering by avoiding a useless copy of data. The vertex mapping use the ***glMapBuffer*** function to access by a pointer to the memory space reserved by the VBO.

```
glBindBuffer(GL_ARRAY_BUFFER, BufferName[POSITION_OBJECT]);
glBufferData(GL_ARRAY_BUFFER, PositionSize, NULL, GL_STREAM_DRAW);
GLvoid* PositionBuffer = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
memcpy(PositionBuffer, PositionData, PositionSize);
glUnmapBuffer(GL_ARRAY_BUFFER);
glVertexPointer(2, GL_FLOAT, 0, 0);
```

One time more, the ***glBufferData*** function is just used to reserve the memory space only, the initialisation is done by the programmer thank to the ***glMapBuffer*** function. There is three types of acces to VBO data: ***GL_WRITE_ONLY***, ***GL_READ_ONLY*** and ***GL_READ_WRITE***. Names are particularly explicites. Modes that allow reading are also very useful because they avoid data duplication for none graphical uses. The function ***glUnmapBuffer*** invalids the pointer. It's preferable to call ***glUnmapBuffer*** as soon as possible because vertex mapping implicate CPU and GPU synchronisation.

Whem many VBOs are used, a good optimisation consists in parallel initialisation because this process decrease the number of CPU/GPU synchronisation. Here an example:

```
glBindBuffer(GL_ARRAY_BUFFER, BufferName[COLOR_OBJECT]);
glBufferData(GL_ARRAY_BUFFER, ColorSize, NULL, GL_STREAM_DRAW);
```

```
GLvoid* ColorBuffer = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

glBindBuffer(GL_ARRAY_BUFFER, BufferName[POSITION_OBJECT]);
glBufferData(GL_ARRAY_BUFFER, PositionSize, NULL, GL_STREAM_DRAW);
GLvoid* PositionBuffer = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

memcpy(ColorBuffer, ColorData, ColorSize);
memcpy(PositionBuffer, PositionData, PositionSize);

glBindBuffer(GL_ARRAY_BUFFER, BufferName[COLOR_OBJECT]);
glUnmapBuffer(GL_ARRAY_BUFFER);
glColorPointer(3, GL_UNSIGNED_BYTE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, BufferName[POSITION_OBJECT]);
glUnmapBuffer(GL_ARRAY_BUFFER);
glVertexPointer(2, GL_FLOAT, 0, 0);
```

3. References

3.1. Buffers usage

	STREAM [1]	DYNAMIC [1]	STATIC [1]
DRAW [1]	GL_STREAM_DRAW	GL_DYNAMIC_DRAW	GL_STATIC_DRAW
READ [2]	GL_STREAM_READ	GL_DYNAMIC_READ	GL_STATIC_ READ
COPY [2]	GL_STREAM_COPY	GL_DYNAMIC_COPY	GL_STATIC_COPY

[1] It is available with all OpenGL 1.5 graphics card and all the ones that support GL_ARB_vertex_buffer_object extension.
[2] There are none available yet; this could change with the release of nVidia G80 and ATI R600.

3.2. Rendering functions

3.2.1. glArrayElement (Obsolete)

This function is obsolete because it comes from the immediate mode.

```
GLvoid glArrayElement(GLint i);
```

Draw the ***i***th vertex of an array and could be used in immediate mode like this:

```
glColorPointer(3, GL_FLOAT, 0, Colors);
glVertexPointer(3, GL_FLOAT, 0, Positions);

glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

glBegin(GL_TRIANGLES);
    glArrayElement(0);
    glArrayElement(1);
    glArrayElement(2);
glEnd();

glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
```

Witch corresponds to the following sequence of functions:

```
glBegin(GL_TRIANGLES);
    glColor3fv(Colors + 0 * 3);
    glVertex3fv(Positions + 0 * 3);
    glColor3fv(Colors + 1 * 3);
    glVertex3fv(Positions + 1 * 3);
    glColor3fv(Colors + 2 * 3);
    glVertex3fv(Positions + 2 * 3);
glEnd();
```

3.2.2. glDrawElements

Draw a geometric primitive according an index array composed by ***count*** indexes.

```
GLvoid glDrawElements(GLenum mode, GLsizei count, GLenum type, GLvoid* indices)
{
    glBegin(mode);
    for(GLint i = 0; i < count; ++i)
        glArrayElement(indices[i]);
    glEnd();
}
```

3.2.3. glDrawRangeElements

Draw a geometric primitive according an index array by keeping only indexes between ***start*** and ***end*** included.

```
GLvoid glDrawRangeElements(GLenum mode, GLuint start, GLuint end,
       GLsizei count, GLenum type, GLvoid* indices)
{
    glBegin(mode);
    for(GLint i = 0; i < count; ++i)
        if(indices[i] >= start && indices[i] <= end)
            glArrayElement(indices[i]);
    glEnd();
}
```

3.2.4. glDrawArrays

Draw a geometric primitive composed by ***count*** vertices starting by the element ***first*** included.

```
GLvoid glDrawArrays(GLenum mode, GLint first, GLsizei count)
{
    glBegin(mode);
    for(GLint i = 0; i < count; ++i)
        glArrayElement(first + i);
    glEnd();
}
```

3.2.5. glMultiDrawArrays

Draw multiples geometric primitives composed by ***count*** vertices starting by the element ***first*** included.

```
GLvoid glMultiDrawArrays(GLenum mode, GLint* first, GLsizei* count,
       GLsizei primcount)
{
    for(GLint i = 0; i < primcount; ++i)
    {
        if(count[i] > 0)
            glDrawArrays(mode, first[i], count[i]);
    }
}
```

3.2.6. glMultiDrawElements

Draw multiple geometric primitives according index arrays.

```
GLvoid glMultiDrawElements(GLenum mode, GLsizei* count, GLenum type,
       GLvoid** indices, GLsizei primcount)
{
    for(GLint i = 0; i < primcount; ++i)
    {
        if(count[i]) > 0)
            glDrawElements(mode, count[i], type, indices[i]);
    }
}
```

3.3. Array types

3.3.1. Fixed pipeline types of array

Specify the fixed pipeline array that we want enable or disable:

```
GLvoid glEnableClientState(GLenum array);
GLvoid glDisableClientState(GLenum array);
```

List of predefined array available:
- ***GL_VERTEX_ARRAY***
- ***GL_NORMAL_ARRAY***
- ***GL_COLOR_ARRAY***
- ***GL_SECONDARY_COLOR_ARRAY***
- ***GL_INDEX_ARRAY***
- ***GL_EDGE_FLAG_ARRAY***
- ***GL_FOG_COORD_ARRAY***
- ***GL_TEXTURE_COORD_ARRAY***

3.3.2. Shader pipeline types of array

Specify the shader pipeline array that we want enable or disable:

```
GLvoid glEnableVertexAttribArray(GLuint index);
GLvoid glDisableVertexAttribArray(GLuint index);
```

index is an identifier of attribute variable.

3.4. Arrays data

3.4.1. Description of fixed pipeline data

Since OpenGL 1.1 and GL_EXT_vertex_array and GL_ARB_vertex_buffer_object :

```
GLvoid glVertexPointer(GLint size, GLenum type, GLsizei stride, GLvoid* pointer);
GLvoid glNormalPointer(GLenum type, GLsizei stride, GLvoid* pointer);
GLvoid glColorPointer(GLint size, GLenum type, GLsizei stride, GLvoid* pointer);
GLvoid glEdgeFlagPointer(GLsizei stride, GLvoid* pointer);
GLvoid glTexCoordPointer(GLint size, GLenum type, GLsizei stride, GLvoid* pointer);
```

The function ***glIndexPointer*** is useless with Vertex Buffer Objects.

```
GLvoid glIndexPointer(GLenum type, GLsizei stride, GLvoid* pointer);
```

Since OpenGL 1.4 and GL_EXT_secondary_color :

```
GLvoid glSecondaryColorPointer(GLint size, GLenum type, GLsizei stride,
       GLvoid* pointer);
```

Since OpenGL 1.4 and GL_EXT_fog_coord :

```
GLvoid glFogCoordPointer(GLenum type, GLsizei stride, GLvoid* pointer);
```

3.4.2. Description of the whole data (Obsolete)

The function ***glInterleavedArrays*** implicated the use of array with predefined structure witch is annoying and none adapted to rending based on multitexture or shaders. It is obsolete since OpenGL 1.3 and GL_ARB_multitexture. Moreover, the VBOs solution to manage interleaved array is really better because it fixes the both issues.

```
GLvoid glInterleavedArrays(GLenum format, GLsizei stride, GLvoid* pointer);
```

3.4.3. Description of shader pipeline data

All attribute define by the fixed pipeline are available in the OpenGL vertex shaders. Is the user want to sand custom attributes, he can create new attribute variables and data will be send to vertex shaders using the function ***glVertexAttribPointer***.

```
GLvoid glVertexAttribPointer(GLuint index, GLint size, GLenum type,
       GLboolean normalized, GLsizei stride, const GLvoid* pointer);
```

OpenGL ES doesn't support fixed pipeline attributes. Consequently, it's necessary to use custom attributes and ***glVertexAttribPointer*** for all attributes. This approach can be used with OpenGL as well and it seams that it will be generalized in the future especially with OpenGL LM. This method increase flexibility and sturdiness of code for a whole shader design. nVidia drivers are ready but ATI ones aren't yet. The class ***CTest5*** from example present this aspect.

3.5. Types de primitive

GL_POINTS: Points.
GL_LINES: List of independent segments.
GL_LINE_STRIP: Single line formed by a list of segments.
GL_LINE_LOOP: Single line formed by a list of segments that loops.
GL_TRIANGLES: List of independent triangles.
GL_TRIANGLE_STRIP: List of triangles that forms a strip.
GL_TRIANGLE_FAN: Create a fan using the first vertex as reference for all triangles.
GL_QUADS: List of independent quadrilateral.
GL_QUAD_STRIP: List of quadrilateral that forms a strip.
GL_POLYGON: Convex polygon.