# VBO, PBO与FBO

从应用程序的内存（CPU）中通过CPU连接到GPU（通常是通过PCIe接口）的总线，传递到GPU本地内存，这样会花费很多时间，以至于大大降低应用程序的运行速度。

如果对每帧，要进行渲染的数据基本相同，或者如果在单个帧中对同样数据的多个副本进行渲染，那么一次性将这些数据复制到GPU的本地内存中，再很多次重复使用这个副本是非常高效的。

使用缓冲区存储顶点数据

由于一个复杂的应用程序可能会需要几个VBO和许多顶点属性，所以OpenGL有一个叫做VAO (vertex array object)的特殊容器对象，用来管理所有这些状态。但是，由于不存在默认的VAO，就需要创建并绑定一个VAO，然后才能使用这一部分的任何代码。创建VAO：

glGenVertexArrays(1, &vao);

glBindVertexArray(vao);

## 简介

*VBO*（*vertex buffer object*）是*GPU*上存储顶点数据的高速缓存。在应用程序初始化阶段，顶点数据被直接传送到显卡中的高速缓存上，在绘制时可以直接从高速缓存中获取，除非几何数据需要修改，否则*VBO*数据不需变化。除了*VBO*技术外，*OpenGL*还提供顶点数组和显示列表的绘制方式。顶点数组可以降低函数调用次数与降低共享顶点的重复使用，但顶点数组函数位于客户端状态中，且每次引用都须向服务端重新发送数据。显示列表为服务端函数，并不受限于数据传输的开销。不过，一旦显示列表编译完成，显示列表中的数据不能够修改。*VBO*技术使用*OpenGL*的*vertex\_buffer\_object*扩展，可以实现显示列表方式的高速数据传递，同时又能像使用顶点数组那样，绘制过程中随时修改数据（可以通过映射缓存到客户端内存空间的方式读取与更新顶点缓存对象中的数据）。

*OpenGL*的三种绘制方式：

*1)* 直接绘制*: glBegin(),glEnd();*

*2)*使用顶点数组*; glDrawArrays()*或*glDrawElements()*或*glDrawRangeElements()*；

*3) VBO: glBindBuffer(), glDrawArrays()*或*glDrawElements();*

## VBO，Vertex Buffer Array

为了加快显示速度，显卡增加了一个扩展，即VBO。它本质上是存储几何数据的缓存。它直接把顶点数据放置到显卡中的高速缓存，极大提高了绘制速度。

这个扩展用到ARB\_vertex\_buffer\_object，它可以直接像顶点数组那样使用。唯一不同的地方在于它需要将数据载入显卡的高效缓存，因此需要占用渲染时间。

初始化阶段：  
1.glGenBuffers (1, &nVBOVertices); //生成1个缓冲区对象

或者glGenBuffers (10, nVBOVertices); // 生成10个缓冲区对象

2.glBindBuffer (GL\_ARRAY\_BUFFER, nVBOVertices); //声明该句柄为一个vbo句柄，并选择之  
3.glBufferData (GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW); //将顶点集上传至server端  
使用阶段：  
1.glEnableClientState(GL\_VERTEX\_ARRAY); //开始使用vbo  
2.glBindBuffer (GL\_ARRAY\_BUFFER, nVBOVertices); //选择当前使用的vbo  
3.glVertexPointer(3, GL\_FLOAT, 0, BUFFER\_OFFSET(0)); //指定vbo顶点格式  
4.glDrawArrays( GL\_TRIANGLES, 0, g\_pMesh->m\_nVertexCount );   
5.glDisableClientState(GL\_VERTEX\_ARRAY); //停止使用vbo  
收尾阶段：  
1.glDeleteBuffers (1, &nVBOVertices); //删除句柄，同时删除server端顶点缓冲

## VBO使用的详细解说(An Songho)

转自：<http://www.songho.ca/opengl/gl_vbo.html>

创建VBO

需要3步：

1. Generate a new buffer object with **glGenBuffers()**.
2. Bind the buffer object with **glBindBuffer()**.
3. Copy vertex data to the buffer object with **glBufferData()**.

（1）glGenBuffers()

glGenBuffers() creates buffer objects and returns the identifiers of the buffer objects. It requires 2 parameters: the first one is the number of buffer objects to create, and the second parameter is the address of a GLuint variable or array to store a single ID or multiple IDs.

void glGenBuffers(GLsizei n, GLuint\* ids)

（2）glBindBuffer()

Once the buffer object has been created, we need to hook the buffer object with the corresponding ID before using the buffer object. glBindBuffer() takes 2 parameters: *target* and *ID*.

void glBindBuffer(GLenum target, GLuint id)

*Target* is a hint to tell VBO whether this buffer object will store vertex array data or index array data: GL\_ARRAY\_BUFFER, or GL\_ELEMENT\_ARRAY\_BUFFER. Any vertex attributes, such as vertex coordinates, texture coordinates, normals and color component arrays should use GL\_ARRAY\_BUFFER. Index array which is used for glDraw[Range]Elements() should be tied with GL\_ELEMENT\_ARRAY\_BUFFER. Note that this *target* flag assists VBO to decide the most efficient locations of buffer objects, for example, some systems may prefer indices in AGP or system memory, and vertices in video memory.

Once glBindBuffer() is first called, VBO initializes the buffer with a zero-sized memory buffer and set the initial VBO states, such as usage and access properties.

（3）glBufferData()

You can copy the data into the buffer object with glBufferData() when the buffer has been initialized.

void glBufferData(GLenum target, GLsizei size, const void\* data, GLenum usage)

Again, the first parameter, *target* would be GL\_ARRAY\_BUFFER or GL\_ELEMENT\_ARRAY\_BUFFER. *Size*is the number of bytes of data to transfer. The third parameter is the pointer to the array of source data. If *data* is NULL pointer, then VBO reserves only memory space with the given data size. The last parameter, *"usage"* flag is another performance hint for VBO to provide how the buffer object is going to be used: *static*, *dynamic* or *stream*, and *read*, *copy* or *draw*.

VBO specifies 9 enumerated values for *usage* flags;

GL\_STATIC\_DRAW

GL\_STATIC\_READ

GL\_STATIC\_COPY

GL\_DYNAMIC\_DRAW

GL\_DYNAMIC\_READ

GL\_DYNAMIC\_COPY

GL\_STREAM\_DRAW

GL\_STREAM\_READ

GL\_STREAM\_COPY

*"Static"* means the data in VBO will not be changed (specified once and used many times), *"dynamic"* means the data will be changed frequently (specified and used repeatedly), and *"stream"* means the data will be changed every frame (specified once and used once). *"Draw"* means the data will be sent to GPU in order to draw (application to GL), *"read"* means the data will be read by the client's application (GL to application), and *"copy"* means the data will be used both drawing and reading (GL to GL).

Note that only *draw* token is useful for VBO, and *copy* and *read* token will be become meaningful only for pixel/frame buffer object ([PBO](http://www.songho.ca/opengl/gl_pbo.html) or [FBO](http://www.songho.ca/opengl/gl_fbo.html)).

VBO memory manager will choose the best memory places for the buffer object based on these usage flags, for example, GL\_STATIC\_DRAW and GL\_STREAM\_DRAW may use video memory, and GL\_DYNAMIC\_DRAW may use AGP memory. Any \_READ\_ related buffers would be fine in system or AGP memory because the data should be easy to access.

（4）glBufferSubData()

void glBufferSubData(GLenum target, GLint offset, GLsizei size, void\* data)

Like glBufferData(), glBufferSubData() is used to copy data into VBO, but it only replaces a range of data into *the existing buffer*, starting from the given offset. (The total size of the buffer must be set by glBufferData() before using glBufferSubData().)

（5）glDeleteBuffers()

void glDeleteBuffers(GLsizei n, const GLuint\* ids)

You can delete a single VBO or multiple VBOs with glDeleteBuffers() if they are not used anymore. After a buffer object is deleted, its contents will be lost.

The following code is an example of creating a single VBO for vertex coordinates. Notice that you can delete the memory allocation for vertex array in your application after you copy data into VBO.

GLuint vboId; // ID of VBO

GLfloat\* vertices = new GLfloat[vCount\*3]; // create vertex array

...

// generate a new VBO and get the associated ID

glGenBuffers(1, &vboId);

// bind VBO in order to use

glBindBuffer(GL\_ARRAY\_BUFFER, vboId);

// upload data to VBO

glBufferData(GL\_ARRAY\_BUFFER, dataSize, vertices, GL\_STATIC\_DRAW);

// it is safe to delete after copying data to VBO

delete [] vertices;

...

// delete VBO when program terminated

glDeleteBuffers(1, &vboId);

绘制VBO

Because VBO sits on top of the existing vertex array implementation, rendering VBO is almost same as using vertex array. Only difference is that the pointer to the vertex array is now as an offset into a currently bound buffer object. Therefore, no additional APIs are required to draw a VBO except glBindBuffer().

Binding the buffer object with 0 switchs off VBO operation. It is a good idea to turn VBO off after use, so normal vertex array operations with absolute pointers will be re-activated.

// bind VBOs for vertex array and index array

glBindBuffer(GL\_ARRAY\_BUFFER, vboId1); // for vertex attributes

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, vboId2); // for indices

glEnableClientState(GL\_VERTEX\_ARRAY); // activate vertex position array

glEnableClientState(GL\_NORMAL\_ARRAY); // activate vertex normal array

glEnableClientState(GL\_TEXTURE\_COORD\_ARRAY); //activate texture coord array

// do same as vertex array except pointer

glVertexPointer(3, GL\_FLOAT, stride, offset1); // last param is offset, not ptr

glNormalPointer(GL\_FLOAT, stride, offset2);

glTexCoordPointer(2, GL\_FLOAT, stride, offset3);

// draw 6 faces using offset of index array

glDrawElements(GL\_TRIANGLES, 36, GL\_UNSIGNED\_BYTE, 0);

glDisableClientState(GL\_VERTEX\_ARRAY); // deactivate vertex position array

glDisableClientState(GL\_NORMAL\_ARRAY); // deactivate vertex normal array

glDisableClientState(GL\_TEXTURE\_COORD\_ARRAY); //deactivate vertex tex coord array

// bind with 0, so, switch back to normal pointer operation

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);

OpenGL version 2.0 added **glVertexAttribPointer()**, **glEnableVertexAttribArray()** and **glDisableVertexAttribArray()** functions to specify generic vertex attributes. Therefore, you can specify all vertex attributes; position, normal, colour and texture coordinate, using single API.

// bind VBOs for vertex array and index array

glBindBuffer(GL\_ARRAY\_BUFFER, vboId1); // for vertex coordinates

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, vboId2); // for indices

glEnableVertexAttribArray(attribVertex); // activate vertex position array

glEnableVertexAttribArray(attribNormal); // activate vertex normal array

glEnableVertexAttribArray(attribTexCoord); // activate texture coords array

// set vertex arrays with generic API

glVertexAttribPointer(attribVertex, 3, GL\_FLOAT, false, stride, offset1);

glVertexAttribPointer(attribNormal, 3, GL\_FLOAT, false, stride, offset2);

glVertexAttribPointer(attribTexCoord, 2, GL\_FLOAT, false, stride, offset3);

// draw 6 faces using offset of index array

glDrawElements(GL\_TRIANGLES, 36, GL\_UNSIGNED\_BYTE, 0);

glDisableVertexAttribArray(attribVertex); // deactivate vertex position

glDisableVertexAttribArray(attribNormal); // deactivate vertex normal

glDisableVertexAttribArray(attribTexCoord); // deactivate texture coords

// bind with 0, so, switch back to normal pointer operation

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);

更新VBO

The advantage of VBO over display list is the client can read and modify the buffer object data, but display list cannot. The simplest method of updating VBO is copying again new data into the bound VBO with glBufferData() or glBufferSubData(). For this case, your application should have a valid vertex array all the time in your application. That means that you must always have 2 copies of vertex data: one in your application and the other in VBO.

The other way to modify buffer object is to map the buffer object into client's memory, and the client can update data with the pointer to the mapped buffer. The following describes how to map VBO into client's memory and how to access the mapped data.

glMapBuffer()

VBO provides glMapBuffer() in order to map the buffer object into client's memory.

void\* glMapBuffer(GLenum target, GLenum access)

If OpenGL is able to map the buffer object into client's address space, glMapBuffer() returns the pointer to the buffer. Otherwise it returns NULL.

The first parameter, *target* is mentioned earlier at glBindBuffer() and the second parameter, *access* flag specifies what to do with the mapped data: read, write or both.

GL\_READ\_ONLY

GL\_WRITE\_ONLY

GL\_READ\_WRITE

Note that glMapBuffer() causes a synchronizing issue. If GPU is still working with the buffer object, glMapBuffer() will not return until GPU finishes its job with the corresponding buffer object.

To avoid waiting (idle), you can call first glBufferData() with NULL pointer, then call glMapBuffer(). In this case, the previous data will be discarded and glMapBuffer() returns a new allocated pointer immediately even if GPU is still working with the previous data.

However, this method is valid only if you want to update entire data set because you discard the previous data. If you want to change only portion of data or to read data, you better not release the previous data.

glUnmapBuffer()

GLboolean glUnmapBuffer(GLenum target)

After modifying the data of VBO, it must be unmapped the buffer object from the client's memory. glUnmapBuffer() returns GL\_TRUE if success. When it returns GL\_FALSE, the contents of VBO become corrupted while the buffer was mapped. The corruption results from screen resolution change or window system specific events. In this case, the data must be resubmitted.

Here is a sample code to modify VBO with mapping method.

// bind then map the VBO

glBindBuffer(GL\_ARRAY\_BUFFER, vboId);

float\* ptr = (float\*)glMapBuffer(GL\_ARRAY\_BUFFER, GL\_WRITE\_ONLY);

// if the pointer is valid(mapped), update VBO

if(ptr)

{

updateMyVBO(ptr, ...); // modify buffer data

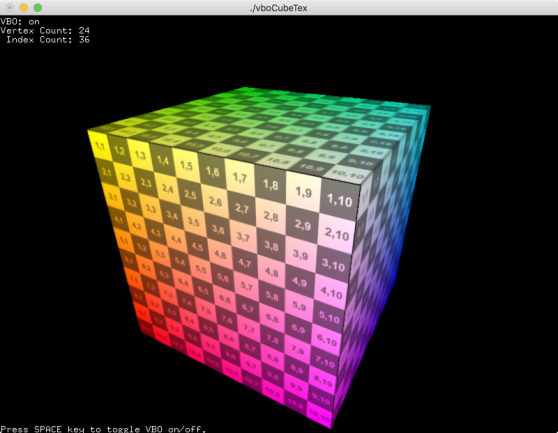
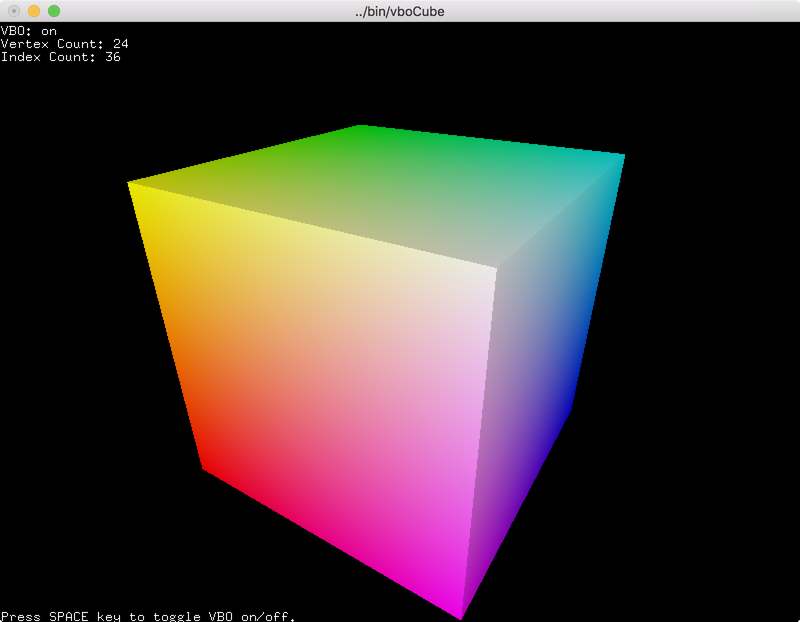
glUnmapBuffer(GL\_ARRAY\_BUFFER); // unmap it after use

}

// you can draw the updated VBO

...

例子：绘制一个立方体



This example is to draw a unit cube with glDrawElements().

Download: [vboCube.zip](http://www.songho.ca/opengl/files/vboCube.zip), [vboCubeTex.zip](http://www.songho.ca/opengl/files/vboCubeTex.zip)

A unit cube can be defined with the following arrays; vertices, normals and colors. A cube has 6 faces and 4 vertices per face, so the number of elements in each array is 24 (6 sides × 4 vertices).

// unit cube

// A cube has 6 sides and each side has 4 vertices, therefore, the total number

// of vertices is 24 (6 sides \* 4 verts), and 72 floats in the vertex array

// since each vertex has 3 components (x,y,z) (= 24 \* 3)

// v6----- v5

// /| /|

// v1------v0|

// | | | |

// | v7----|-v4

// |/ |/

// v2------v3

// vertex position array

GLfloat vertices[] = {

.5f, .5f, .5f, -.5f, .5f, .5f, -.5f,-.5f, .5f, .5f,-.5f, .5f, // v0,v1,v2,v3 (front)

.5f, .5f, .5f, .5f,-.5f, .5f, .5f,-.5f,-.5f, .5f, .5f,-.5f, // v0,v3,v4,v5 (right)

.5f, .5f, .5f, .5f, .5f,-.5f, -.5f, .5f,-.5f, -.5f, .5f, .5f, // v0,v5,v6,v1 (top)

-.5f, .5f, .5f, -.5f, .5f,-.5f, -.5f,-.5f,-.5f, -.5f,-.5f, .5f, // v1,v6,v7,v2 (left)

-.5f,-.5f,-.5f, .5f,-.5f,-.5f, .5f,-.5f, .5f, -.5f,-.5f, .5f, // v7,v4,v3,v2 (bottom)

.5f,-.5f,-.5f, -.5f,-.5f,-.5f, -.5f, .5f,-.5f, .5f, .5f,-.5f // v4,v7,v6,v5 (back)

};

// normal array

GLfloat normals[] = {

0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, // v0,v1,v2,v3 (front)

1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, // v0,v3,v4,v5 (right)

0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, // v0,v5,v6,v1 (top)

-1, 0, 0, -1, 0, 0, -1, 0, 0, -1, 0, 0, // v1,v6,v7,v2 (left)

0,-1, 0, 0,-1, 0, 0,-1, 0, 0,-1, 0, // v7,v4,v3,v2 (bottom)

0, 0,-1, 0, 0,-1, 0, 0,-1, 0, 0,-1 // v4,v7,v6,v5 (back)

};

// colour array

GLfloat colors[] = {

1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, // v0,v1,v2,v3 (front)

1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, // v0,v3,v4,v5 (right)

1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, // v0,v5,v6,v1 (top)

1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, // v1,v6,v7,v2 (left)

0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, // v7,v4,v3,v2 (bottom)

0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1 // v4,v7,v6,v5 (back)

};

// texture coord array

GLfloat texCoords[] = {

1, 0, 0, 0, 0, 1, 1, 1, // v0,v1,v2,v3 (front)

0, 0, 0, 1, 1, 1, 1, 0, // v0,v3,v4,v5 (right)

1, 1, 1, 0, 0, 0, 0, 1, // v0,v5,v6,v1 (top)

1, 0, 0, 0, 0, 1, 1, 1, // v1,v6,v7,v2 (left)

0, 1, 1, 1, 1, 0, 0, 0, // v7,v4,v3,v2 (bottom)

0, 1, 1, 1, 1, 0, 0, 0 // v4,v7,v6,v5 (back)

};

// index array for glDrawElements()

// A cube requires 36 indices = 6 sides \* 2 tris \* 3 verts

GLuint indices[] = {

0, 1, 2, 2, 3, 0, // v0-v1-v2, v2-v3-v0 (front)

4, 5, 6, 6, 7, 4, // v0-v3-v4, v4-v5-v0 (right)

8, 9,10, 10,11, 8, // v0-v5-v6, v6-v1-v0 (top)

12,13,14, 14,15,12, // v1-v6-v7, v7-v2-v1 (left)

16,17,18, 18,19,16, // v7-v4-v3, v3-v2-v7 (bottom)

20,21,22, 22,23,20 // v4-v7-v6, v6-v5-v4 (back)

};

After the vertex attribute arrays and index array was defined, they can be copied into VBOs. One VBO is used to store all vertex attribute arrays one after the other by using glBufferSubData() with the array size and offsets, for example, [VVV...NNN...CCC...TTT...]. The second VBO is for the index data only.

// create VBOs

glGenBuffers(1, &vboId); // for vertex buffer

glGenBuffers(1, &iboId); // for index buffer

size\_t vSize = sizeof vertices;

size\_t nSize = sizeof normals;

size\_t cSize = sizeof colors;

size\_t tSize = sizeof texCoords;

// copy vertex attribs data to VBO

glBindBuffer(GL\_ARRAY\_BUFFER, vboId);

glBufferData(GL\_ARRAY\_BUFFER, vSize+nSize+cSize+tSize, 0, GL\_STATIC\_DRAW); // reserve space

glBufferSubData(GL\_ARRAY\_BUFFER, 0, vSize, vertices); // copy verts at offset 0

glBufferSubData(GL\_ARRAY\_BUFFER, vSize, nSize, normals); // copy norms after verts

glBufferSubData(GL\_ARRAY\_BUFFER, vSize+nSize, cSize, colors); // copy cols after norms

glBufferSubData(GL\_ARRAY\_BUFFER, vSize+nSize+cSize, tSize, texCoords); // copy texs after cols

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

// copy index data to VBO

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, iboId);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(indices), indices, GL\_STATIC\_DRAW);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);

Drawing VBO using OpenGL fixed pipeline is almost identical to [Vertex Array](http://www.songho.ca/opengl/gl_vertexarray.html). The only difference is to specify the memory offsets where the data are stored, instead of the pointers to the arrays. For OpenGL programmable pipeline using vertex/fragment shaders, please refer to the next example, [Drawing a Cube with Shader](http://www.songho.ca/opengl/gl_vbo.html#example2).

// bind VBOs before drawing

glBindBuffer(GL\_ARRAY\_BUFFER, vboId);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, iboId);

// enable vertex arrays

glEnableClientState(GL\_VERTEX\_ARRAY);

glEnableClientState(GL\_NORMAL\_ARRAY);

glEnableClientState(GL\_COLOR\_ARRAY);

glEnableClientState(GL\_TEXTURE\_COORD\_ARRAY);

size\_t nOffset = sizeof vertices;

size\_t cOffset = nOffset + sizeof normals;

size\_t tOffset = cOffset + sizeof colors;

// specify vertex arrays with their offsets

glVertexPointer(3, GL\_FLOAT, 0, (void\*)0);

glNormalPointer(GL\_FLOAT, 0, (void\*)nOffset);

glColorPointer(3, GL\_FLOAT, 0, (void\*)cOffset);

glTexCoordPointer(2, GL\_FLOAT, 0, (void\*)tOffset);

// finally draw a cube with glDrawElements()

glDrawElements(GL\_TRIANGLES, // primitive type

36, // # of indices

GL\_UNSIGNED\_INT, // data type

(void\*)0); // offset to indices

// disable vertex arrays

glDisableClientState(GL\_VERTEX\_ARRAY);

glDisableClientState(GL\_NORMAL\_ARRAY);

glDisableClientState(GL\_COLOR\_ARRAY);

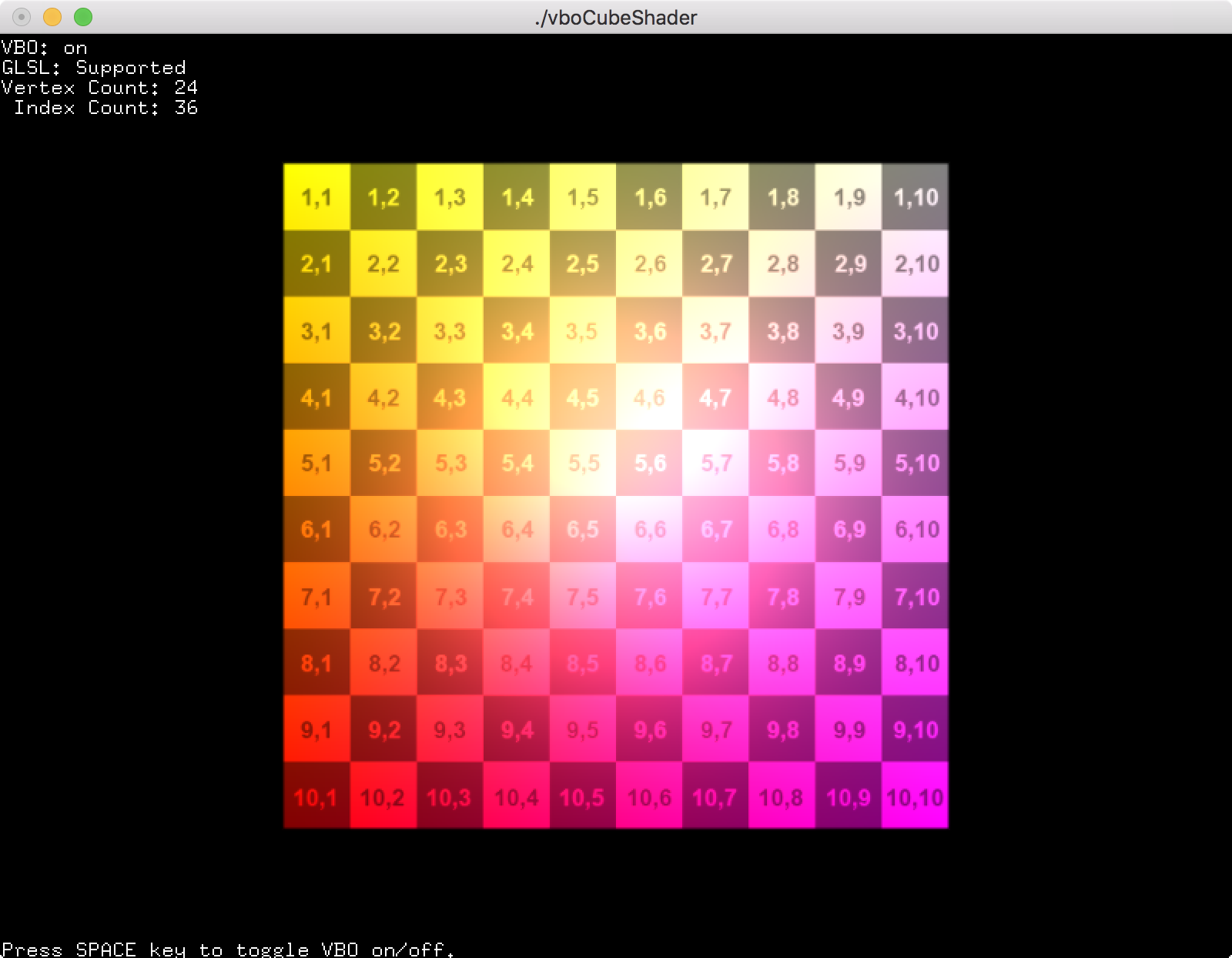
glDisableClientState(GL\_TEXTURE\_COORD\_ARRAY);

// unbind VBOs

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);

Example: Drawing a Cube with Shader



**Download:** [vboCubeShader.zip](http://www.songho.ca/opengl/files/vboCubeShader.zip)

This example is to draw a cube with GLSL shader. Creating and storing data to VBOs is the same as Vertex Array mode. The only difference in GLSL mode is specifying the offset to the vertex buffer data to draw VBOs. OpenGL v2.0 introduces a new generic function, **glVertexAttribPointer()** to specify the offset for any vertex attribute type, instead of using type-specific functions; glVertexPointer, glNormalPointer(), glColorPointer(), etc.

// bind VBOs and GLSL program

glUseProgram(progId);

glBindBuffer(GL\_ARRAY\_BUFFER, vboId);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, iboId);

// activate vertex attribs

glEnableVertexAttribArray(attribVertex);

glEnableVertexAttribArray(attribNormal);

glEnableVertexAttribArray(attribColor);

// set attrib offsets using glVertexAttribPointer()

glVertexAttribPointer(attribVertex, 3, GL\_FLOAT, GL\_FALSE, 0, (void\*)vertexOffset);

glVertexAttribPointer(attribNormal, 3, GL\_FLOAT, GL\_FALSE, 0, (void\*)normalOffset);

glVertexAttribPointer(attribColor, 3, GL\_FLOAT, FL\_FALSE, 0, (void\*)colorOffset);

// draw VBO

glDrawElements(GL\_TRIANGLES, // primitive type

36, // # of indices

GL\_UNSIGNED\_INT, // data type

(void\*)0); // offset to indices

// deactivate vertex attribs

glDisableVertexAttribArray(attribVertex);

glDisableVertexAttribArray(attribNormal);

glDisableVertexAttribArray(attribColor);

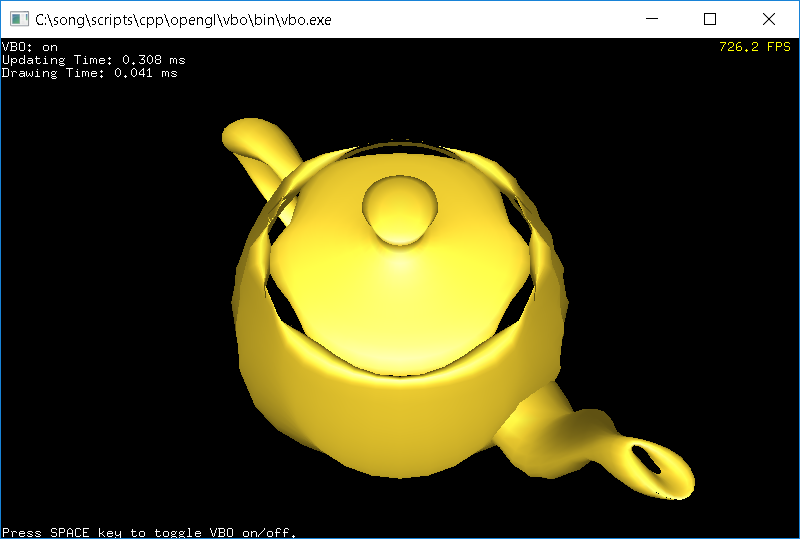
// unbind

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);

glUseProgram(0);

Example: Updating Vertex Data using glMapBuffer()



This demo application makes a VBO wobbling in and out along normals. It maps a VBO and updates its vertices every frame with the pointer to the mapped buffer. You can compare the performace with a traditional vertex array implementation. It uses 2 vertex buffers; one for both the vertex positions and normals, and the other stores the index array only.

Download: [vbo.zip](http://www.songho.ca/opengl/files/vbo.zip) (Updated: 2018-08-15).

// bind VBO

glBindBuffer(GL\_ARRAY\_BUFFER, vboId);

// map the current VBO into client's memory,

// so, this application can directly access VBO

// Note that glMapBuffer() causes sync issue.

// If GPU is working with this buffer, glMapBufferARB() will wait(stall)

// for GPU to finish its job.

float\* ptr = (float\*)glMapBuffer(GL\_ARRAY\_BUFFER, GL\_READ\_WRITE);

if(ptr)

{

// wobble vertex in and out along normal

updateVertices(ptr, srcVertices, teapotNormals, vertexCount, timer.getElapsedTime());

// release pointer to mapped buffer after use

glUnmapBuffer(GL\_ARRAY\_BUFFER);

}

// unbind VBO

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

Every example includes Code::Blocks project file for Windows, and the makefiles (Makefile.linux and Makefile.mac) for linux system and macOS in *src* folder, so you can build an executable on your system, for example:

> make -f Makefile.linux

> make -f Makefile.mac

## PBO, Pixel Buffer Object

*PBO*（*pixel buffer object*）是*GPU*上存储像素数据的高速缓存，类似于*VBO*存储顶点数据。*PBO*的优势是像素数据的快速传递，还可以使*CPU*与*GPU*异步执行。图*1*是用传统的方法从图像源（如图像文件或视频）载入图像数据到纹理对象的过程。像素数据首先存到系统内存中，接着使用*glTexImage2D*将数据从系统内存拷贝到纹理对象。包含的两个子过程均需要有*CPU*执行。而从图*2*中，可以看到像素数据直接载入到*PBO*中，这个过程仍需要*CPU*来执行，但是从数据从*PBO*到纹理对象的过程则由*GPU*来执行*DMA*，而不需要*CPU*参与。另外，*OpenGL*可以进行异步*DMA*，不必等像素数据传递完毕*CPU*就可以继续执行其他操作。

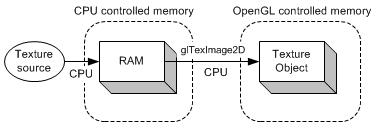


图1

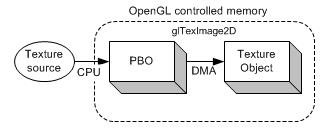


图2

再来看看纹理缓冲(PBO)是怎么使用的，其实差不多：

初始化阶段：  
1.glGenTextures(1, &texID); //创建句柄  
2.glBindTexture(GL\_TEXTURE\_2D, texID); //设置句柄类型  
3.glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA, img->GetWidth(), img->GetHeight(), 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, raw\_rgba); //上传纹理缓冲

使用阶段：  
1.glEnable(GL\_TEXTURE\_2D); //开始使用纹理缓冲  
2.glBindTexture(GL\_TEXTURE\_2D, texID); //选择当前使用的纹理缓冲  
3.发送顶点和纹理坐标，画吧...省略  
4.glDisable(GL\_TEXTURE\_2D); //停止使用纹理

收尾阶段：  
1. glDeleteTextures(1,&texID); //删除句柄，同时删除server端缓冲

## FBO, frame buffer object

*FBO (frame buffer object)*是*OpenGL*扩展*GL\_EXT\_framebuffer\_object*提供的不能显示的帧缓存接口。和*Windows*系统提供的帧缓存一样，*FBO*也有一组相应存储颜色、深度和模板（注意没有累积）数据的缓存区域。*FBO*中存储这些数据的区域称之为*“*缓存关联图像*”*（*frame buffer-attached image*）。缓存关联图像分为两类：纹理缓存和渲染（显示）缓存（*renderbuffer*）。如果纹理对象的图像数据关联到帧缓存，*opengl*执行的将是*“*渲染到纹理*”*（*render to texture*）操作。如果渲染缓存对象的图像数据关联到帧缓存，*opengl*执行的将是*“*离线渲染*”*（*offscreen rendering*）。

