**Opengl的VBO**

顶点数组的数据是存储在内存中的，顶点缓冲对象的数据是存在显存中的

顶点数组渲染时数据从内存送入渲染管线，而VBO则从显存送入渲染管线

顶点数组通过**glDrawEliments或glDrawArrays从cpu主存中拷贝到gpu**中进行运算与渲染

Opengl顶点缓冲区对象。GL\_ARB\_vertex\_buffer\_object扩展可以提升opengl的性能，它提供了顶点数组和显示列表，这避免了低效实现这些功能。**Vertex buffer object(VBO)允许顶点数组储存在高性能显卡上，即服务器端的内存中，改善数据传输效率，如果缓冲区对象保存了像素数据，它就被称为Pixel Buffer Object(PBO)**

使用顶点数组可以减少函数调用次数及复用共享顶点，然而，顶点数组的缺点是顶点函数以及顶点数据在客户端（对于opengl来说，显卡为服务器端，其他为客户端），每次引用顶点数组时，都必须将顶点数据从客户端（CPU）发送到服务器端（GPU），另一方面，显示列表是服务端函数，它不会再重头传送数据，但是，一旦显示列表被编译了，显示列表中的数据就不能修改了。

VBO为顶点创建了一个缓冲区对象。缓冲区对象在服务器端的高性能内存中，并提供了相同的函数，引用这些数组，如glVertexPointer，glNormalPointer，glTexCoorPointer等等。顶点缓冲区内存管理器将缓冲区对象放在存储器中的最佳位置，与显示列表不同的是，在顶点缓冲区对象中的数据可以读也可以将它映射到服务器端的内存空间中，然后更新它的数据。

VBO的另外一个重要优点是，可以在许多客户端中共享缓冲区对象，就像显示列表和纹理那样，由于vbo在服务器端，多个客户端可以通过对应的标识符访问同一个缓冲区。

**创建VBO的三个步骤**

1）使用glGenBuffersARB得到一个新的缓冲区对象

2）使用glBindBufferARB绑定一个缓冲区对象

3）使用glBufferDataARB复制顶点数据到缓冲区对象

glGenBuffersARB(GLsizei n, GLuint\* buffers)

glGenBuffersARB创建缓冲区对象并返回缓冲区对象的标识符。

参数n：表示需要创建顶点缓存对象的个数

参数buffers：用于存储创建好的顶点缓存对象句柄，句柄使用时大于0的正整数，0是opengl es保留

glBindBufferARB(GLenum target, GLunit buffer)

一旦缓冲区对象被创建，我们要再使用缓冲区对象前将缓冲区对象与ID绑定

参数target：指定绑定的对象，取值为GL\_ARRAY\_BUFFER（用于顶点数据）或GL\_ELEMENT\_ARRAY\_BUFFER（用于索引数据）

参数buffer：顶点缓存对象句柄

glBufferData(GLenum target, GLsizeiptr size, const GLvoid\* data, GLenum usage)

为顶点缓存对象分配空间，加载缓冲区对象

参数target：与glBindBufferARB的参数target相同

参数size：指定点点缓存区的大小，以字节为单位计数

参数data：用于初始化顶点缓存区的数据，可以为NULL，表示只分配空间，之后再由glBufferSubData进行初始化

参数usage：表示该缓存区域将会被如何使用，它的主要目的是用于提示opengl该对该缓存区域做何种程序的优化

其参数为以下三个之一：

GL\_STATIC\_DRAW：表示该缓冲区不会被修改

GL\_DYNAMIC\_DRAW：表示该缓存区的数据可能会经常变化

GL\_STREAM\_DRAW：表示该缓存区的数据可能不太会变化

如果顶点数据一经初始化就不会再被修改，那么就应该尽量使用GL\_STATIC\_DRAW，这样

能获得更好的性能

glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size, const GLvoid\* data)

参数offset：表示需要更新的数据的起始偏移量

参数size：表示要更新的数据的个数，也是以字节为计数单位

参数data：用于更新的数据

glDeleteBuffers(GLsizei , const GLunit\* buffers)

参数与glGenBuffers类似，用于删除顶点缓存对象，释放顶点缓存

也就是分为两个阶段（初始化阶段、使用阶段和收尾阶段）

1）初始化阶段

glGenBufferARB

glBindBufferARB

glBufferDataARB

2）使用阶段

glEnableClientState(GL\_VERTEX\_ARRAY) 开始使用VBO

glBindBufferARB(GL\_ARRAY\_BUFFER\_ARB,nVBOVertices) 选择当前VBO

glVertexPointer() 指定vbo顶点格式

glDrawArrays() 绘制

glDisableClientState(GL\_VERTEX\_ARRAY) 停止使用vbo

3）收尾阶段

glDeleteBufferARB(1, &nVBOVertices) 删除句柄，同时删除server端顶点缓冲

上面的过程与顶点数组的调用有一些变化：

1. 需要调用vbo初始化的三个函数
2. 需要调用glBindBuffer函数

上面的过程和纹理缓冲对象很相似：

1）初始化阶段

glGenTextures 创建句柄

glBindTexture(GL\_TEXTURE\_2D, texID) 选择当前使用的纹理缓冲

发送顶点和纹理坐标，绘制

glDisable(GL\_TEXTURE\_2D) 停止使用纹理

2）收尾阶段

glDeleteTextures 删除句柄，同时阐述server端缓冲

这个概念都很相似：

1. 创建句柄
2. 设置句柄类型
3. 上传数据
4. 开始使用缓冲
5. 选择句柄
6. 使用缓冲
7. 停止使用缓冲
8. 删除句柄和缓冲

glEnable glEnableClientState一个是针对服务器端的，一个是针对客户端的

**VBO与VA性能的比较：**

1）直接使用VA：

auto start = chrono::system\_clock::now();

for(int i = 0, size = 10000000; i < size; i++)

{

glColor4f(1.0f, 0.0f, 0.0f, 1.0f);

glEnableClientState(GL\_VERTEX\_ARRAY);

glVertexPointer(2, GL\_FLOAT, 0, vertices);

glDrawElements(GL\_LINES, 4, GL\_UNSIGNED\_INT, index);

}

auto end = chrono::system\_clock::now();

cout << "elapse time = " << chrono::duration\_cast<chrono::milliseconds>(end - start).count() << endl;



2）使用VBO

初始化：

glGenBuffersARB(1, &vboId);

glBindBufferARB(GL\_ARRAY\_BUFFER\_ARB, vboId);

glBufferDataARB(GL\_ARRAY\_BUFFER\_ARB, sizeof(vertices) \* sizeof(GL\_FLOAT), vertices, GL\_STATIC\_DRAW\_ARB);

delete[] vertices;

调用：

glEnableClientState(GL\_VERTEX\_ARRAY);

glBindBufferARB(GL\_ARRAY\_BUFFER\_ARB, vboId);

glVertexPointer(2, GL\_FLOAT, 0, NULL);

glDrawArrays(GL\_LINES, 0, NUMBER \* 4);

注意：在数据被复制到缓冲区对象之后，原指针就不再需要了，所有的工作空间缓冲区都被删除，这就产生了三个影响：首先，它会释放客户端内存，其次，它会消耗图形硬件的内存，最后，我们不再能够对数据进行修改，因为无法再访问它

下面的例子是开启多个缓冲区（其中顶点坐标、颜色与纹理为一个，边界标志为一个）：

static GLfloat vertices[] =

{

-100.0f, -100.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,

100.0f, -100.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f,

100.0f, 100.0f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f,

-100.0f, 100.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f

};

static GLboolean edgeFlag[] =

{

GL\_TRUE, GL\_FALSE, GL\_TRUE, GL\_TRUE

};

初始化：

glGenBuffersARB(1, &vboId);

glBindBufferARB(GL\_ARRAY\_BUFFER, vboId);

glBufferDataARB(GL\_ARRAY\_BUFFER, sizeof(GL\_FLOAT)\* sizeof(vertices), vertices, GL\_STATIC\_DRAW);

glGenBuffersARB(1, &edgeFlagId);

glBindBufferARB(GL\_ARRAY\_BUFFER, edgeFlagId);

glBufferDataARB(GL\_ARRAY\_BUFFER, sizeof(GL\_UNSIGNED\_BYTE)\* sizeof(edgeFlag), edgeFlag, GL\_STATIC\_DRAW);

// 纹理1

GLint width, height, component;

GLenum format;

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);

glGenTextures(1, &textureId1);

glBindTexture(GL\_TEXTURE\_2D, textureId1);

GLbyte \*bytes = gltLoadTGA("fire.tga", &width, &height, &component, &format);

if (bytes == NULL) cout << "load tga picture failed" << endl;

glTexImage2D(GL\_TEXTURE\_2D, 0, component, width, height, 0, format, GL\_UNSIGNED\_BYTE, bytes);

free(bytes);

// 设置纹理参数

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

// 设置纹理环绕

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

// 设置纹理环境

glTexEnvf(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_MODULATE);

// 纹理2

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);

glGenTextures(1, &textureId2);

glBindTexture(GL\_TEXTURE\_2D, textureId2);

bytes = gltLoadTGA("..\\logo.tga", &width, &height, &component, &format);

if (bytes == NULL) cout << "load tga picture failed" << endl;

glTexImage2D(GL\_TEXTURE\_2D, 0, component, width, height, 0, format, GL\_UNSIGNED\_BYTE, bytes);

free(bytes);

// 设置纹理参数

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

// 设置纹理环绕

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

// 设置纹理环境

glTexEnvf(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_MODULATE);

// 开启纹理对象

glEnable(GL\_TEXTURE\_2D);

绘制：

// 顶点坐标与颜色

GLsizei stride = 9 \* sizeof(GL\_FLOAT);

GLuint \*vertexPointer = NULL;

GLvoid \*colorPointer = ((GLubyte\*)NULL) + 3 \* sizeof(GL\_FLOAT);

GLvoid \*texPointer = ((GLubyte\*)NULL) + 7 \* sizeof(GL\_FLOAT);

glBindBufferARB(GL\_ARRAY\_BUFFER, vboId);

if (index % 2 == 0)

{

glBindTexture(GL\_TEXTURE\_2D, textureId1);

}

else

{

glBindTexture(GL\_TEXTURE\_2D, textureId2);

}

index++;

glEnableClientState(GL\_VERTEX\_ARRAY);

glEnableClientState(GL\_COLOR\_ARRAY);

glEnableClientState(GL\_TEXTURE\_COORD\_ARRAY);

glVertexPointer(3, GL\_FLOAT, stride, NULL);

glColorPointer(4, GL\_FLOAT, stride, colorPointer);

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

// 顶点边界标志

glBindBufferARB(GL\_ARRAY\_BUFFER, edgeFlagId);

glEnableClientState(GL\_EDGE\_FLAG\_ARRAY);

glEdgeFlagPointer(0, NULL);

glDrawElements(GL\_TRIANGLES, sizeof(indices), GL\_UNSIGNED\_BYTE, indices);

glBindBufferARB(GL\_ARRAY\_BUFFER, NULL);



注意：glDrawElements中的最后一个参数：这个参数也是个多面手，如果没有使用 VBO，它指向 CPU 内存中的索引数据数组；如果使用 VBO 绑定到 GL\_ELEMENT\_ARRAY\_BUFFER，那么它表示索引数据在 VBO 中的偏移量。

也就是我们可以这样使用ebo

创建：

glGenBuffersARB(1, &eboId);

glBindBufferARB(GL\_ELEMENT\_ARRAY\_BUFFER, eboId);

glBufferDataARB(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(GL\_FLOAT)\* indicesData.size(), indicesData.data(), GL\_STATIC\_DRAW);

绑定：

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

绘制：

glDrawElements(GL\_TRIANGLES, sizeof(GL\_FLOAT)\* indicesData.size(), GL\_UNSIGNED\_INT, NULL);

**如果对于动态数据的修改可以调用：**

glMapBufferARB与glUnmapBufferARB

glEnableClientState(GL\_VERTEX\_ARRAY);

glBindBufferARB(GL\_ARRAY\_BUFFER\_ARB, vboId);

glVertexPointer(2, GL\_FLOAT, 0, NULL);

glDrawArrays(GL\_LINES, 0, NUMBER \* 4);

GLfloat\* bufferData = (GLfloat\*)glMapBufferARB(GL\_ARRAY\_BUFFER, GL\_READ\_WRITE\_ARB);

if(bufferData != nullptr)

{

for(int i = 0, size = NUMBER \* 4; i < size; i++)

{

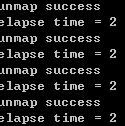
bufferData[i]++;

}

}

boolean b = glUnmapBufferARB(GL\_ARRAY\_BUFFER\_ARB);

cout << (b ? "unmap success" : "unmap failed") << endl;



实际的表现结果就是发现一条直线在向右上方移动（正确）

glMapBufferARB 用来将一个缓冲区对象中的数据映射为客户端中的地址空间

glUnmapBufferARB用来释放缓冲区对象与客户端地址空间的关系，如果一个映射关系在使用之后没有使用glUnmapBufferARB来释放，那么任何访问缓冲区对象的命令都会导致错误，当映射关系被解除后，使用映射得到的指针就会失效

注意：glMapBuffer()提供了对缓冲区对象中包含的整个数据集合的访问。如果需要修改

缓冲区中的大多数数据，这种方法很有用，但是，如果有一个很大的缓冲区并且只需要更新

很小的一部分值，这种方法效率很低。

C++代码

1. glBindBuffer(GL\_ARRAY\_BUFFER,m\_nPositionVBO);
2. glEnableClientState(GL\_VERTEX\_ARRAY);
3. glVertexPointer(2, GL\_FLOAT, 0, NULL);
4. glBindBuffer(GL\_ARRAY\_BUFFER,m\_nTexcoordVBO);
5. glEnableClientState(GL\_TEXTURE\_COORD\_ARRAY);
6. glTexCoordPointer(2, GL\_FLOAT, 0, NULL);
7. glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER,m\_nIndexVBO);
8. glDrawElements(GL\_TRIANGLES, 6, GL\_UNSIGNED\_SHORT,NULL);
9. glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, NULL);
10. glDisableClientState(GL\_TEXTURE\_COORD\_ARRAY);
11. glDisableClientState(GL\_VERTEX\_ARRAY);
12. glBindBuffer(GL\_ARRAY\_BUFFER, NULL);

C++代码2

1. glBindBuffer(GL\_ARRAY\_BUFFER,m\_nQuadPositionVBO);
2. glEnableVertexAttribArray(VAT\_POSITION);
3. glVertexAttribPointer(VAT\_POSITION, 2, GL\_INT, GL\_FALSE,0, NULL);
4. glBindBuffer(GL\_ARRAY\_BUFFER,m\_nQuadTexcoordVBO);
5. glEnableVertexAttribArray(VAT\_TEXCOORD);
6. glVertexAttribPointer(VAT\_TEXCOORD, 2, GL\_INT, GL\_FALSE,0, NULL);
7. glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER,m\_nQuadIndexVBO);
8. glDrawElements(GL\_TRIANGLES, 6, GL\_UNSIGNED\_SHORT,NULL);
9. glDisableVertexAttribArray(VAT\_POSITION);
10. glDisableVertexAttribArray(VAT\_TEXCOORD);
11. glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, NULL);
12. glBindBuffer(GL\_ARRAY\_BUFFER, NULL);

以上两端是效果一致的VBO渲染部分的代码，尽量使用第二种，使用第二种的前提是使用shader来进行顶点处理。VAT\_POSITION / VAT\_TEXCOORD需要与shader里代表顶点/纹理坐标的attribute变量建立联系。