# OPENGL

[The OpenGL Extension Wrangler (GLEW)](http://glew.sourceforge.net/)是用来访问OpenGL 3.2 API函数的。不幸的是你不能简单的使用#include <GL/gl.h>来访问OpenGL接口，除非你想用旧版本的OpenGL。在现代OpenGL中，API函数是在运行时（run time）确定的，而非编译期（compile time）。GLEW可以在运行时加载OpenGL API。

Shaders在现代OpenGL中是个很重要的概念。应用程序离不开它，除非你理解了，否则这些代码也没有任何意义。

Shaders是一段GLSL小程序，运行在GPU上而非CPU。它们使用[OpenGL Shading Language (GLSL)](http://en.wikipedia.org/wiki/GLSL" \t "_blank)语言编写，看上去像C或C++，但却是另外一种不同的语言。使用shader就像你写个普通程序一样：写代码，编译，最后链接在一起才生成最终的程序。

Shaders并不是个很好的名字，因为它不仅仅只做着色。只要记得它们是个用不同的语言写的，运行在显卡上的小程序就行。

在旧版本的OpenGL中，shaders是可选的。在现代OpenGL中，为了能在屏幕上显示出物体，shaders是必须的。

当shaders运行在GPU，其它代码运行在CPU时，你需要有种方式将数据从CPU传给GPU。在本文中，我们传送了一个三角的三个顶点数据，但在更大的工程中3D模型会有成千上万个顶点，颜色，贴图坐标和其它东西。

这就是我们为什么需要Vertex Buffer Objects (VBOs)和Vertex Array Objects (VAOs)。VBO和VAO用来将C++程序的数据传给shaders来渲染。

**Vertex Buffer Objects (VBOs)**

第一步我们需要从内存里上传三角形的三个顶点到显存中。这就是VBO该干的事。VBO其实就是显存的“缓冲区（buffers）” - 一串包含各种二进制数据的字节区域。你能上传3D坐标，颜色，甚至是你喜欢的音乐和诗歌。VBO不关心这些数据是啥，因为它只是对内存进行复制。

**Vertex Array Objects (VAOs)**

第二步我们要用VBO的数据在shaders中渲染三角形。请记住VBO只是一块数据，它不清楚这些数据的类型。而告诉OpenGL这缓冲区里是啥类型数据，这事就归VAO管。

VAO对VBO和shader变量进行了连接。它描述了VBO所包含的数据类型，还有该传递数据给哪个shader变量。在OpenGL所有不准确的技术名词中，“Vertex Array Object”是最烂的一个，因为它根本没有解释VAO该干的事。

你回头看下本文的vertex shader（在文章的前面），你就能发现我们只有一个输入变量vert。在本文中，我们用VAO来说明“hi，OpenGL，这里的VBO有3D顶点，我想要你在vertex shader时，发三个顶点数据给vert变量。”

在后续的文章中，我们会用VAO来说“hi，OpenGL，这里的VBO有3D顶点，颜色，贴图坐标，我想要你在shader时，发顶点数据给vert变量，发颜色数据给vertColor变量，发贴图坐标给vertTexCoord变量。”

OpenGL的缓冲对象提供了一种高效的数据操作及吞吐方式，这此操作涉及整个渲染管线的多个环节，用来提高OpenGL的实现效率。常见的缓冲对象有：顶点缓冲对象(VBO)、像素缓冲对象(PBO)和帧缓冲对象(FBO)，各种缓冲对象的操作方式及应用场合不尽相同，但主要都是在关系到内存与显存之间需进行数据传输与交互时而进行的优化操作，因为内存与显存之间的数据交互在当前的硬件渲染**[架构](http://lib.csdn.net/base/architecture" \o "大型网站架构知识库" \t "_blank)**下是必须的，但效率却又受限于PCI的传输速度。缓冲区对象的作用就是让这种数据交互操作在一种最优的方式下进行，使所有的操作从Client到显存之间的距离尽可能地缩短，从而提高效率。

**Vertex Buffer Object:**

1.      在Client与Server之间快速地操作几何顶点数据，能够很快地对顶点数据进行更新等操作。这里顶点数据的实质可以根据具体需要以其它数据填充。

2.      开辟的实际数据空间的位置不确定，根据VBO的属性及CPU调用的次数，由显存逐渐向内存转移。

**Pixel Buffer Object:**

1.      在Client与Server之间快速地操作各类像素数据

2.      高效并不在于传输更快，而在于与硬件相关的异步调用方式，调用之后CPU即返回执行其它操作（使用DMA方式的传输、由OpenGL直接控制）

3.      在单个PBO情况下并不能得到很好的效果，毕竟传输过程仍然存在（但速度可能变快，比如显存内部的数据传输），但其异步性就提供了双PBO实现的可能性，用双PBO来进行加速。

4.      PBO关联的实际数据空间位置不确定，主要由PBO的属性及OpenGL来确定。

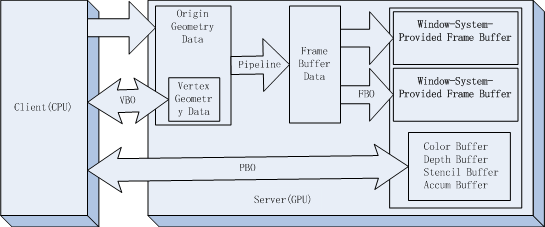
**Frame Buffer Object:**

1.      提供了一个与FrameBuffer同类型的离屏渲染操作对象，可直接控制管线最终数据的去向。

2.      分配的空间是在显存之中，因此用FBO来对显存进行涉及GPU运算的交互操作效率较高。

3.      主要用于显存内部数据的转化、交互、操作的场合。

整个过程可用如下图所示：



注意：所有的Buffer Object都只是一个中间的管理平台，与实际的空间相分离。其中，除了FBO所关联的数据空间在显存之外，PBO和VBO所关联的数据空间位置均不一定（system memory, shared memory, AGP, Video Memory均有可能）

关于更加详细的Buffer Object使用方法可以参见：

《OpenGL Super Bible》(chapter11, chapter18)

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

<http://www.songho.ca/opengl/gl_pbo.html>

<http://www.songho.ca/opengl/gl_fbo.html>

PS 更详细PBO参考<http://blog.csdn.net/u012501459/article/details/12945165>

# NVIDIA VIDEO SDK