译者注：

最近翻找有关于 VBO 的资料，发现网上很少有这方面的中文资料。现将 nvidia 官方网站上的一篇文章《Using Vertex Buffer Objects(VBOs)》，特将其翻译出来，以供后人使用☺。翻译时间不到一天，匆忙之余，难免有误，敬请指正，欢迎来邮，我是 Neil。

联系邮箱：zhuhuazha@yahoo.com.cn
联系blog：Neil`s Blog

Using VBOs 是来自于 Nvidia 的官方网站上的白皮书。

来源网址：http://developer.nvidia.com/object/using_VBOs.html，如有转载请注明版权。

白皮书

Using Vertex Buffer Objects (VBOs)

使 用 VBOs

概要：

VBO 是一种强大的技术：它允许我们在服务器端的高速缓存上存储一定量的数据。

该要素提供这样一种机制：压缩数据到“缓存对象（buffer objects)”，使得处理这些数据时不必从服务器直接取出来，从而加快数据传输速率。

VBOs 可在以下几点上帮助我们：

- 由客户端或者状态函数指向的任意数据体。最典型的就是我们所讨论的 glVertexPointer(),glColorPointer(),glNormalPointer()等等。
- 画图元集的指示数组（glDraw[Range]Elements()）。

该机制的最初想法是提供一些能由标识符获取的内存块（缓存）。就像显示列表和纹理，我们可以通过绑定这样一个缓存来激活它。

该绑定操作将每个客户机/状态函数指针变成一定量的偏移，因为我们将会在内存区域里用到，该区域是相对于当前范围缓存的。换句话说，该扩展将客户机/状态函数变成了服务器/状态函数。

我们都知道客户机/状态函数处理数据的范围仅是对于客户机自身是可访问的。其它的任一个客户机都不可能访问到这些数据。通过在服务器端执行这些函数序列，才可以在不同的客户机上共享使用这些数据。许多客户机能绑定通用的缓存，它们都像纹理和显示列表一样通过标识符来处理。

VAR 的问题：

以前解决这类任务的扩展方法是用 VAR（Vertex Array Range 顶点数组序列）。尽管该扩展仍然可以用，但我们建议您用 VBO 来代替。

VAR 功能上完全够用，但在此不妨列举一下开发人员觉得它不好的地方：

- 它打破了服务器/客户机的模式，因为客户机部分控制了内存管理（本归于服务器）。
- 并没有提供一个内部的内存管理，VAR 所提供的唯一功能就是在服务器的内存上定位一大块内存区域。
- 当给 VAR 定位缓存时，开发者需要描述哪里要用到 AGP、系统、或者显卡缓存，而这些工作显得麻烦而棘手。
- 开发人员仍然需要为 VAR 在本地创建自己的内存定位以优化它们借来的缓存块。
- 高效的内存管理必须是像旗语一样的防卫系统。

为了简化问题，我们可以说 VBO 能像 VAR 样处理问题，并且能为你管理内存（图 1）。

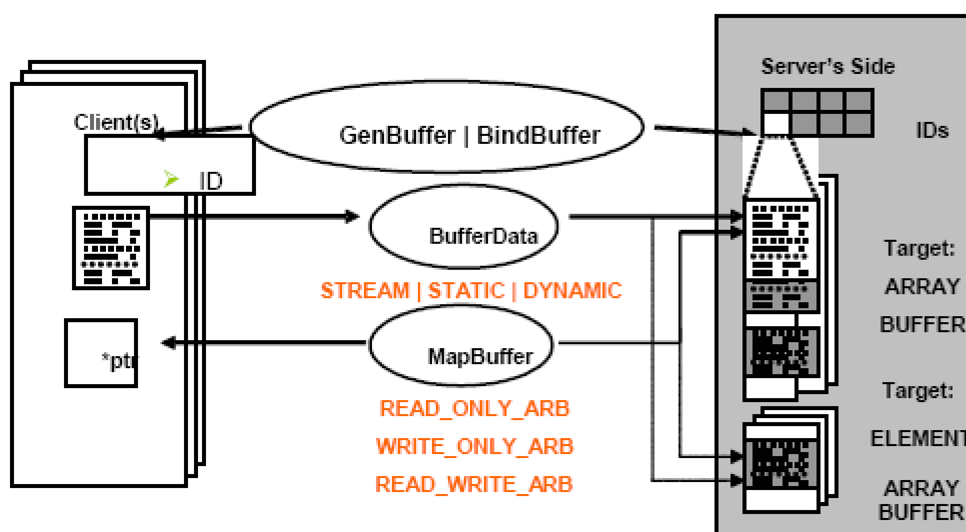

图 1　VBO 使用

内存管理

OpenGL 有着与 Direct3D 为顶点数组提供的 AGP/视频内存（显存）管理似的同一层次功能。OpenGL 里的工作方式也十分相似。它有映射到内存缓存的能力，从而定义各缓存的利用，映射和取消映射（在 D3D 中是 Lock/Unlock）。

内部内存管理能为我们选择最佳的内存类型（系统、视频卡或者 AGP），这取决于我们所用的缓存。

VBO 提供了不同的方式与缓存对象进行交互：

- 绑定缓存（glBindBuffer）：该操作允许客户机/状态函数直接在缓存区域工作，而不必在客户机的绝对（物理）内存上。绑定 0 号缓存则切断 VBO，这样就可以用绝对指针回到原有客户机的状态模式。
- 利用 VBO 的 API 加载数据到这些缓存对象上(glBufferData、glBufferSubData、glGetBufferSubData)：这些函数让你在客户机区域和服务器的缓存对象作一个复制）。
- 利用缓存映射技术（glMapBuffer 和 glUnmapBuffer）：这个类似于 D3D 的 Lock 与

UnLock。你可以得到一个临时指针作为缓存开始的入口，意味着缓存是映射到了客户机内存上。OpenGL 负责如何映射到客户机上。由于这一点，映射必须作为一个短的操作，并且指针不能存储给以后的应用。

对象

VBO 与以下两种对象一起工作：

- 数组缓存（ARRAY__BUFFER__ARB）：这些缓存包含顶点属性，例如顶点坐标、纹理坐标数据、各顶点着色数据和法向信息。你可以插入该数据（利用 stride 参数），或者在另一数组后再写入一个数组（比如先写 1000 个顶点，再写 1000 个法向量，等等）。GlVertexPointer glNormalPOinter（等等）必须有正确的指向偏移量。
- 图元数组缓存（ELEMENT__ARRAY__BUFFER__ARB）：该类型缓存主要在 glDraw[Range]Element()中用于图元指针。它可能仅包含图元的指示符。

这两种目标平行同步建立，因为图元数组在 glDraw[Range]Element()函数中必须与数组缓存同时获取。

用这两种对象的一个有意思的地方是，当保存同一顶点数组缓存时，交换各种图元缓存的能力。我们可以实施 LOD，或者当我们工作于同一顶点数据库时改变图元表所产生的其它任何效果。

关于 PBO 的几句话

建议给 VBO 增加更多的对象的另一扩展是 ARB_pixel_buffer_object(PBO，象素缓存对象)。

尽管它在我们的 50.XX 版本驱动里还没有（译者注：此为 nividia 的图形驱动），该扩展允许我们通过增加两个新的对象在纹理、帧缓存、离屏缓存里工作。

换句话说，它会对顶点、法向、图元等等提供同样的机制，但是对于字节数组却不同。

新的两种对象是：

- PIXEL_PACK_BUFFER：该对象给各种读操作提供缓存，例如 glReadPixels 和 glGetTexImage。这些命令将会把它们的数据写到当前范围缓存对象中。
- PIXEL_UNPACK_BUFFER：该对象给各种写操作提供缓存，例如 glBitmap,glDrawPixels 和 glTexImage2D。这些命令将会从缓存对象中读取数据。

在 VBOs 与 PBOs 混合时会有一些很有意思的优化措施（图 2）：

- 渲染到顶点数组：如果我们打算在第一个通道创建一个特殊的顶点数组（用于贴图、移位等等），我们可以避免作为一个顶点程序的输入而在客户机端复制 p 缓存(pBuffer)，并放回到服务器端。VBO/PBO 会保持所有数据流都在服务器端。
- 流纹理：该操作类似于我们操作 PDR（Pixel Data Range 象素数据范围）；我们用 MapBuffer/UnMapBuffer 来改变纹理数据，基于视频流，然后调用 TexSubImage 来更新纹理。

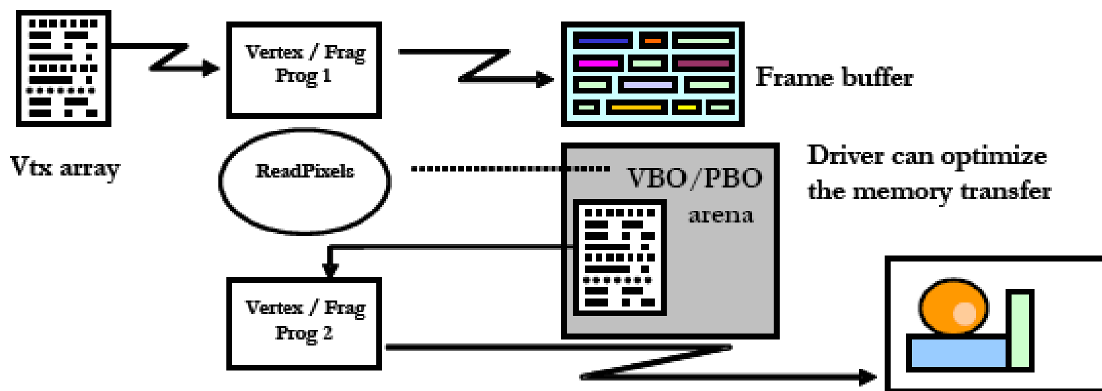

图 2　VBO/PBO 结合实例

新的步骤、功能函数和符号标志

使用标志：

- 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

访问标志：

- READ_ONLY_ARB
- WRITE_ONLY_ARB
- READ_WRITE_ARB

对象：

- ARRAY_BUFFER_ARB
- ELEMENT_BUFFER_ARB

void BindBufferARB(enum target, uint buffer)：

BindBufferARB 函数用于绑定对象 ID 作为使用的实际缓存。如果 ID 为 0 就关闭了缓存使用。

void *MapBufferARB(enum target, enum access)；

boolean UnmapBufferARB(enum target)；

MapBufferARB 函数提供一个指针指向当前缓存对象的映射区域，UnmapBufferARB 则解除映射。

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

BufferDataARB 有以下两种用法：

- 当数据集置为 NULL 时简化当前缓存对象的内存装载和使用量。这样，你后来就可以通过映射缓存来装载数据。
- 定位缓存，设置 usage，　复制一些数据；特别用于处理静态内存模型。

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

该函数用于在缓存对象的特定区域内复制数据。

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

该函数用于在当前缓存的特定区域获取数据。

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

void GenBufferARB(sizei n, uint *buffers);

boolean IsBufferARB(uint buffer);

这三组函数类似于显示列表/纹理标识符；它们能缓存对象定位、释放或者查询一些标识符。

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

该函数返回关于当前缓存对象的各种参数，pname 可能是：

- BUFFER_SIZE_ARB：返回缓存对象的大小。
- BUFFER_USAGE_ARB：返回缓存对象的用法。
- BUFFER_ACCESS_ARB：返回缓存对象的可访问标志。
- BUFFER_MAPPED_ARB：告诉你我们是否已经映射了该缓存。

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

该函数返回缓存的实际指针，如果该缓存已经被映射了的话(MapBufferARB)。Pname 这次只能是 BUFFER_MAP_POINTER_ARB。

Get{Boolean, Integer, Float, Double}v 的标签

缓存对象 ID0 是保留的，当 0 缓存对象与所给出的对象外时，该绑定缓存所影响的命令才能正常起作用。当非 0 缓存出界了，指针就表示一个偏移，并且将会超出 VBO 的管理。

你可以使用以下标签来得知哪个缓存是作为 VBO 偏移的：

- 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
- SECONDAR_COLOR_ ARRAY_BUFFER_BINDING_ARB
- FOG_COORDINATE_ ARRAY_BUFFER_BINDING_ARB
- WEIGTH_ ARRAY_BUFFER_BINDING_ARB

GetVertexAttribivARB 的标签：

当使用 VBO 和顶点程序工作时，一些属性会有一些任意的意思：例如一个法向数组可能不仅仅是用于存储法向信息。替代前一节使用标签的方法是：你可以使用属性的索引。该标签允许你通过偏移系统利用 VBO 来查询使用了哪个属性码。

- VERTEX_ATTRIB_ARRAY_BUFFER_BINDING_ARB

各函数的目的

glBufferDataARB()

该函数是应用与内存之间的抽象层。但在每个缓存对象的背后是复杂的内存管理系统。

基本上该函数主要有以下作用：

- 检查存储数据变化的大小和使用类型
- 如果大小为 0，释放该缓存对象的内存。
- 如果大小和存储类型没有变化，并且该缓存没有被 GPU 使用，我们就可以使用它。

所有的一切都已经为使用设置好。

- 另一方面，如果 GPU 在用它，或者将会用到它，或者存储类型有变化，我们就必须为该缓存申请另一块内存区域以作准备。
- 如果数据指针不用空，我们就会复制新的数据到内存区域。

从中我们可见到我们在第二次调用该函数前所拥有的内存不一定与调用后的内存是同一块。但是，从应用程序的角度来看它仍然是相同的(同一缓存对象)。但是在驱动层面上，我们*正在优化并且允许应用程序不必等候 GPU*。

实际上，我们已经定位了一大块内存池用于后面的程序定位。当我们调用 函数 glBufferDataARB 时，我们为当前缓存对象保留了其中一块。然后我们就用数据来填充，并用它来绘制，并标记该缓存为用过的(类似于 glFence 函数)。

如果我们在 GPU 动作完成之前再次调用 glBufferDataARB，我们可以简单的为该缓存对象从内存池中新开一块区域。这是可能的，因为 BufferDataARB 会认为我们准备重新描述缓存里面的数据(BufferSubDataARB 反对的)。

使用标志

使用参数是帮助 VBO 内存管理完全优化你的缓存的一个关键值。

标志名称	定义
STATIC_…	假定 1 到 n 是更新绘制。表明数据仅描述一次(在初始化中)
DYNAMIC_…	N 对 N 的绘制。通常表明数据经常更新，但每次更新都会绘制多次。例如，每少数几帧更新任一动态数据。
STREAM_…	1 对 1 的绘制。每绘制一次都会更新。STREAM 有点像 DYNAMIC：数据会发生变化。但是，数据会一直变化，因此它可能会在完成的时候变得不稳定(像显存)，当它消失时(比如模式变换)会立即被替换掉。
…_READ_…	指我们必须有一个读取数据的简单访问：AGP 或者系统内存会很适合。
…_COPY_…	意味我们即将执行_READ_和_DRAW_操作
…_DRAW_…	指缓存将向 GPU 发送数据。我们或许想利用视频(STATIC | STREAM_DRAW_ARB),或者 AGP（DYNAMIC__DRAW__ARB）内存

表 1　标签使用列表

这种内存使用的合并，能帮助内存管理者平衡三种内存：系统、AGP、显示。另一方面，它还得计算出能为其它缓存回收多少内存区域。STATIC,STREAM, 和 DYNAMIC 这里就是为此目的。

但是 STATIC,STREAM, 和 DYNAMIC 只是一些简单的建议和提示潜在的使用模式。它们并不特别强迫驱动器作任何事情，但是它们帮助我们决策缓存管理配置和映射行为。我们假定定位和利用的数据总是可获取的，直到你特意释放它，通过删除缓存（如果你打算在其上创建另一缓存时所用的重量级方法）或者调用 BufferDataARB(…,NULL, ..)；第二种方式更可取。

在服务器端，这些并不是严重约束。它们建议帮助我们确定在哪里放置数据并且如何管理它。没有任何东西阻碍你创建一个 STATIC 数据存储，然后每一帧都更新它。也没有任何理由限制你不能创建　STREAMING 数据而却不能修改它。尽管如此，我们仍然极力反对

这种行为。

glBufferSubDataARB()

该函数提供一个为已有缓存替换一定范围数据的方法。注意：为了避免冲突，我们必须等待 GPU 如果有 GPU 正在此块上工作的话，结果就有可能丧失一些效果。

GlBindBufferARB()

该函数加载内部参数，这样在顶点数组上的下一步操作，或者任一 VBO 函数，都可以在当前缓存对象上工作。注意：这种绑定操作是很廉价的：是一种预前绑定，等等其它操作来变化 VBO 管理者的内部状态。

glMapBufferARB()

该函数映射缓存对象到客户机内存中，依据访问标志。在最好的情况下，没有任何的数据传输：驱动器正好"展现"实际的指针到 AGP 或者系统内存中。

在其它情况下，如果访问标志条件不满足，则有可能会传输一些数据。例如，要求一个指针去读取一个缓存，该缓存存在于显存，这时就要求驱动器降级这个缓存到 AGP 或者系统缓存。

利用访问标志来尽量有效的准备内存，有赖于我们需要对它所做的事情。比如，我们可以用 WRITE__ONLY 读取映射缓存。这只是对于驱动器的一个提示符，而不是限制。尽管如此，驱动器会允许我们从一个 WRITE__ONLY 缓存（写仍然很快）非常慢的读取。

glVertexPointer()

该函数依据当前的缓存对象设置偏移量（初始为一个指针）。VBO 内存管理的大部分工作都是在此完成的。

VBO 使用建议

以下是一些保持 VBO 高效工作的建议。

用 glMapBufferARB（）的 glBufferDataARB()

有时我们需要更新缓存对象里的数据，而我们又不想再次访问原缓存里的旧数据。这种情况典型的发生在我们前面所提到的调用 **glBufferData** 中。

但是，我们可以使用 glMapBuffer 来更新整个数据集。不幸的是，这步操作比 glBufferData 开销更大。我们必须谨记：驱动器并不能猜到我们将对 glMapBuffer 所返回的内存指针会做什么：我们仅是改变其中少数字节？或者我们要更新所有数据？

由 glMapBuffer 返回的指针指向的是数据的实际地址。有可能 GPU 正在使用这些数据，因此申请更新数据将会强制要求驱动器等待 GPU 完成它的绘制任务。

为了解决该冲突，你只须用一个空指针来调用 glBufferDataARB()。然后调用 glMapBuffer，就会通知驱动器前面用的数据已经无效了。接下来的结果就是，如果 GPU 仍然工作于这些数据，它们将不会冲突，因此我们将这些数据置为无效了。GlMapBuffer 函数返回一个新的指针，我们可以用该指针而同时 GPU 正在工作于前面的数据。

注意：Microsoft®的 Direct3D®解决该问题的办法是：通过 D3DLOCK_DISCARD 提供一个参数标志给 Lock()方法。

避免每个 VBO 调用 glVertexPointer()一次以上

glVertexPointer()函数在 VBO 里做了很多装载工作，因此要避免冗余。

最有效的办法是绑定 VBO 缓存，装载各种数组指针（glNormalPointer etc）后再调用 glVertexPointer()。每个 VBO 应该只调用一次 glVertexPointer()。

你可能会认为 VBO 管理的本质工作是在 glBindBufferARB 中完成的，但实际上正好相反。VBO 系统等待着即将来临的输入函数（像 glVertexPointer）。

绑定操作相对于各种指针的装载来说开销是很小的。

该建议适用于任何像 glVertexPointer()一样工作的函数。

在 glDrawArrays()参数里面用“First”值代替改变 glVertexPointer 函数

在函数

glDrawArrays(Glenum mode, Glint first, Glsizei count);

通过代替改变 glVertexPointer 到一个特定的偏移位置并将其 first 置为 NULL 的方法，改变 glDrawArrays()里的“first”参数值将更为有效。正如我们前面所讲到的，这将会阻止通过 VBO 管理者执行另一装载步骤的过程。

用 glDrawRangeElements()代替 glDrawElements()来绘制

用范围图元来画更有效率是基于以下两个原因：

- 如果所描述的范围匹配于 16 位的整数，驱动器将会优化通过 GPU 指示数的格式。它可以将一个 32 位的整数转为 16 位的整数。这样，就会得到 2X 的提升。
- Range 是 VBO 管理者的精确信息，它可以用来优化它的内部内存配置。