The Efficient Use of Vertex Buffers
by
Richard Huddy
RichardH@nvidia.com

Problem statement:
• Vertex Buffer misuse is the single most common architectural problem in DX7 apps

Aim:
• Good throughput from all data types

Complications
• Dynamic vs. Static
• Optimal Vertex Buffer sizes
• Optimal FVF
• To index or not to index?
• Strips vs. lists
• The meaning of “Optimize”
• Which caches are at work and how do they affect things
• What does it mean to Lock a Vertex Buffer?
• When is it inappropriate to use Vertex Buffers?

Problem Resolution
• The way to handle all cases
  • Static Data
  • Dynamic Data
  • Large data sets

Essential thinking… or “Why it should all make sense”.
• Why the Vertex Buffer is good in the first case
• Register renaming in CPUs
• Thinking about the GPU caches
• How DX8 changes things

Appendix 1
• Pseudo code for updating dynamic Vertex Buffers.

Appendix 2
• Six simple rules to take home.
Terminology and Assumptions

It is assumed that the reader is familiar with DirectX 7.

- **“HEL”** The software emulation layer available on DX7 which handles transform, lighting and rasterization (all on the CPU).
- **“Regular HAL”** The software HAL on DX7. All transforms and lighting handled on the CPU. Hardware accelerated rasterization only.
- **“TnL HAL”** The hardware accelerated path in DX7 which handles transform, clipping, lighting and rasterization.
- **“VB”** A Vertex Buffer as defined in DX7.
- **“GPU”** The Graphics Processing Unit. A single chip containing all the necessary logic to handle transformation, lighting, clipping, and rasterization.
- **“FVF”** Flexible Vertex Format – as defined in DX7.

The following uses are typically made of memory on the PC:

- **Local Video Memory:** Frame Buffers, Z buffers, most textures
- **AGP memory:** Overspill textures, Vertex Buffers
- **System Memory:** Program code, Vertex Buffers which are explicitly loaded into system memory, all system memory surfaces and copies.

Problem Statement

**Vertex Buffer misuse is the single most common architectural problem in DX7 apps**

In the day-to-day work of the Technical Developer Relations staff, the single most common tough problem that we meet is the improper use of VBs. On DX7 the penalty for non-optimal use can range from significant to very severe.

Making the right decision early on in development typically pays off many times over. Re-architecting apps which abuse Vertex Buffers can be a painful process.

**Aim**

**Getting good throughput from all data types**

The objective for any performance sensitive app should be to produce a polygon throughput that stresses the platform on which it is running. One of the main difficulties with achieving this is the wide spread of capabilities from low-end legacy hardware to top performance modern hardware. In spite of this spread, there are a few basic rules that you can apply. These rules will help you to maximize the performance on each and every one of the machines that your app will end up running on. The key notion that is examined in this paper is Vertex Buffers - and their efficient use.
We will target high poly throughputs with both static and dynamic data and show that, unlike previous hardware generations, it is now possible to write apps that actually get within a few percentage points of the ‘theoretical’ maximum throughput.

**Complications**

**Dynamic vs. Static data**
One of the simplest misconceptions about hardware transform and lighting (“H/W TnL”) is that this is designed exclusively to handle static data – and that dynamic data suffers such a severe performance penalty as to eliminate the advantage of hardware acceleration.

In fact, correct and careful use of the API is capable of showing very high throughputs for dynamically generated data (far higher than using the CPU alone). Throughputs as high as 11 million triangles per second have been achieved on GeForce 256 with dynamic data (which is around 70% of the peak theoretical figure for static data!).

Static data is easier to handle and many benchmarks exist which are able to demonstrate polygon throughputs that actually match the hardware specified rates.

It is necessary to use different data management approaches for dynamic and static data – but both can be handled within the same application without great difficulty.

**Optimal Vertex Buffer sizes**
For the GeForce family of GPUs there is no best size for a VB but the following rules apply:

1. VB switches are expensive (much more expensive that on the regular HAL) so packing multiple objects into one VB is a good thing – simply because it saves the cost of the transition.
2. Batches of primitives made up of fewer than about 200 triangles are necessarily sub-optimal. Very small batches (say 10 or fewer) should be regarded as a main target for optimization.
3. Redundant information in the vertex format can bloat the VB and dramatically reduce the rate at which vertex data can be read across the AGP bus.

Note that (1) and (3) can produce conflicting demands. Since switching VB is so expensive it can be advantageous to use a common vertex format throughout all or much of your app so that all vertex data can live in a small number of VBs. This is quite typical of optimization issues and makes the task significantly more complex.

Microsoft have recommended a size of “around 1000 vertices” for most of the lifetime of DX7, but recent evidence suggests that bigger is better and sizes in excess of 2000 vertices are measurably superior.
Optimal FVF
The general rule is: Prefer compact FVFs. Redundant data tends to reduce the efficiency of the bus usage when transferring data – and is usually a bad thing. Sometimes it can be justified on the basis of saving VB transitions or eliminating duplicate vertices, but most usually it simply slows the system down and should be avoided. If including a small amount of redundant data (say, a second texture coordinate pair) allows you to use just one dynamic VB then it’s almost certain to be a win because there’s a bug in the DX7 runtime which means that there are special performance benefits to using one, and only one, dynamic VB.

That said there are certain highly efficient data types which the GeForce family of GPUs handle very quickly. These are typically 32 bytes or 64 bytes in length.

To index or not to index?
Generally, indexed primitives are to be preferred over non-indexed primitives. The single best format in which to send data to the API is as indexed strips, a close second is indexed lists. Indexing can substantially reduce the total bandwidth consumed when handling long sequences of triangles; as can be seen from the optimal case in which a strip ends with a final index that refers to a vertex that is already in the GPU cache. In such a case the total amount of data that needs to be loaded into the GPU to setup the final triangle is just two bytes (since all indices in DX7 are 16 bit quantities).

As bandwidth demands increase in future chips, it is reasonable to expect indexing to be preferred well into the future. In addition, the introduction in DX8 of index buffers is a step that will most certainly help.

Strips vs. lists
Because strips represent each additional triangle by adding just one extra vertex, it should come as no surprise to discover that strips are the preferred way of passing data to the API. In the limit, rectangular grids approach the ideal throughput of two triangles for every additional vertex. In such cases, indexed strips can send just 1 word per triangle (indexed lists require at least 3 words per triangle).

The meaning of “Optimize”
Calling Optimize() on a VB has three main effects.
(1) It means that the contents of the VB are re-arranged in such a way as they can no longer be used with any other rendering device. So a VB allocated in AGP memory which is passed to Optimize can no longer be read directly by the CPU. Locks, and calls to ProcessVertices will both fail.
(2) Rendering will, in some cases, be noticeably faster. For hardware TnL devices you should expect only a minimal speed up, but for software devices Optimize is a significant opportunity.
(3) VBs that have been optimized can never be locked thereafter.
We recommend using Optimize in all cases where your data is highly persistent. You should not call Optimize on dynamic data (because you’ll lose the right to lock that VB) and you should not call Optimize on data that lasts for only a few renders.

If calling Optimize would force you to Create or Destroy VBs in any time critical code then you should avoid it at all costs. You should never call Optimize in time-critical code.

**Which caches are at work and how do they affect things**

On the GeForce family of GPU’s there are two separate caches and effective use of both can make a big difference.

**The GPU Memory Cache**

There is apure memory cache which simply stores the most recently used lines of AGP memory which were read when fetching vertices from VBs. The size of the cache line is 32 bytes so in cases where your FVF isn’t an exact multiple of 32 then it’s in your interest to access vertex data in a roughly sequential way (because otherwise reads into the cache will often fetch data which will not be used). True random access into a VB effectively eliminates this cache and therefore should be avoided. As with all caches “locality of reference” is the basic principle that brings rewards.

[“Locality of reference” is a term which simply means tending to make memory accesses as close to the previous access as possible.]

**The GPU Vertex Cache**

The second GPU cache (which is better known) is a cache of post-transform and light data. The last ten distinct vertices that the chip has processed are maintained in a cache that is driven as a FIFO (first in first out). The value of this cache is greater the higher the load on the GPU. The fact that the cache is run as a FIFO is not intuitive – but in practice it has shown to be highly effective. Just like the memory cache the essential lesson is to go for locality of reference. Vertex data that has just been handled should be used again as soon as possible. As with the memory cache, random access is a bad policy.

It should be obvious that the Vertex Cache is only able to help you if you use indexed primitives. If that’s not obvious, think about it for a short while…

**The best way to create the Vertex Buffer**

When VBs are going to be used for rendering (i.e. not submitted to ProcessVertices) then their optimal placement is an important matter.

Software devices (e.g. the regular HAL) need their VBs to be placed in system memory at creation time. Hardware devices (i.e. the TnL HAL) need their VBs to reside in non-system memory. Typically this means that the driver will put them into AGP memory.
AGP memory is fast for the CPU to write, and fast for the GPU to read – but very slow for the CPU to read.

For a GPU the ideal create call will use only WRITEONLY. Generally, you should not specify SYSTEMMEMORY for VBs that will be used for the GPU rendering methods.

The underlying logic is surprisingly simple. You should apply the following rule. “VBs should be created with the SYSTEMMEMORY flag if and only if they will be read by the CPU”. That includes the app using the CPU directly to access data, and it includes the runtime accessing data via ProcessVertices. There’s no gray area here. If the CPU is going to read from them, then your VBs belong in system memory.

Naturally, you should avoid creating ‘pathological’ situations where you use the CPU to read from a VB very rarely, accompanied by very frequent access by the GPU. In such cases, you probably do better to maintain a system memory copy for the CPU to use and an AGP copy for the GPU.

It’s also worth noting that by far the most typical usage pattern for VB data is that the app creates it and only ever writes to it. Using the CPU to read from a VB is relatively unusual…

**What does it mean to Lock a Vertex Buffer?**

Locking a VB is the way by which you get direct access to the memory within a VB via a pointer which is returned from the Lock call. It is important to note that this pointer is only valid for the duration of the Lock. Like all DirectX surfaces there is no reason to expect the pointer returned from successive calls to Lock to remain valid. This means that you should never attempt to retain a pointer into a VB after releasing the lock. Using that pointer is almost certain to lead to major system instabilities that can be very hard to track down.

As with all surfaces in DirectX, locking a VB can be an arbitrarily slow process, and for that reason Lock should be used with great care.

**Using the flags to aid performance.**

Critically, locking a VB that is currently in use by the GPU can stall the whole graphics pipeline and cause severe performance impact. When locking VBs care should be taken to make sure that the flags WRITEONLY, DISCARDCONTENT and NOOVERWRITE are always used in the correct way.

The flags are clearly described in the DX7 help files and you should take great care to read and understand the pseudo code that is provided at the end of this document in Appendix 1.

**What happens if I use both of DISCARDCONTENT and NOOVERWRITE?**
Many people (including the author) have wrongly assumed that DISCARDCONTENTS is the ‘winning’ flag if both are set. This is not the case. The implicit promise given when using NOOVERWRITE clearly indicates that the app is guaranteeing that it will not trample on any data that is currently in use – and for that reason DISCARDCONTENTS is ignored if NOOVERWRITE is also supplied.

It is recommended that you avoid passing both flags to a single call to Lock.

Using WRITEONLY for both creation and locking is highly advantageous as it allows the driver to return AGP memory. CPU reads from AGP memory are very slow, but GPU reads from AGP are several times faster than from CPU-cached system memory.

**When is it a bad idea to use Vertex Buffers?**

Never!

If you are coding for a hardware accelerated transform and lighting device then you should *always* use vertex buffers. This applies to all vertex data, even for non-3D operations like HUD art or text. If you fail to use a vertex buffer the runtime will copy your data into it’s own internal VB but will not apply the optimal rules for using DISCARDCONTENTS and NOOVERWRITE.

In severe cases we have seen performance cut by more than one half as a consequence of introducing non-VB based rendering. Don’t make this mistake!

**Problem Resolution**

**The way to handle all cases:**

**Static Data**

Static data is any vertex data that does not change for hundreds of frames, possibly for the life of the game (or level). Static data should be handled by committing it to AGP memory (by specifying WRITEONLY and *not* specifying SYSTEMMEMORY). Because there is typically much more AGP memory than free video memory this can work even if you have many megabytes that need to be held. Most current games don’t use in excess of 32Mb of vertex data – if you are considering doing so then you should be careful to ensure that you have good justification. Remember also that AGP allocations always consume real physical memory that cannot be swapped out by the Windows virtual memory manager.

For each VB of static data the following rules should be applied.

1. Create the VB using only the flag WRITEONLY.
2. Lock the VB once (again using WRITEONLY) and fill the VB. If this code is likely to be executed in a performance sensitive situation then try to make your
writes sequential to allow the write combining capabilities of the CPU to work for you. Random access writes to AGP memory are much slower.

(3) Unlock the VB
(4) Optimize the VB

**Dynamic Data**
Dynamic data is data which can be varied or ‘written’ to (even if it’s only changed infrequently) in the course of the game (or level). As such, it will require locking from time to time and therefore can never be the subject of a call to Optimize.

Dynamic data falls into two distinct categories. Data which is only the subject of write operations is most efficiently rendered and this should generally be your aim with dynamic data. The alternative is to have data which can be both read and written by the CPU. For performance reasons this should not be kept in AGP memory as CPU reads from AGP are very slow. These two dynamic data types are referred to as W/O (Write Only) and R/W (read write) in the remainder of this section.

For each VB of dynamic data the following rules should be applied.

**W/O:**
Create the VB using only the flag WRITEONLY and then every time the data is updated…

(1) Lock the VB (using WRITEONLY and one of DISCARDCONTENTS and NOVERWRITE) and fill the VB. See the pseudo code in Appendix 1 for guidance on which flags to use and when. If this code is likely to be executed in a performance sensitive situation then try to make your writes sequential to allow the write combining capabilities of the CPU to work for you.

(2) Unlock the VB
(3) Submit the VB to DIPVB

**R/W:**
Create the VB with the flag SYSTEMMEMORY and then every time the data is updated…

(1) Lock the VB and fill the VB with any read/modify/write operations required. [Note that you will be unable to specify any of the DISCARDCONTENTS, NOOVERWRITE and WRITEONLY flags.]

(2) Unlock the VB
(4) Submit the VB to DIPVB

N.B. Just to complicate matters, there’s a bug in the DX7 runtime which means that there are special performance benefits to using exactly one dynamic VB.

**Large data sets**
Large data sets are typically best broken down into subsets and each subset treated as static or dynamic on it’s own merits. If you are thinking of creating very many large VBs or substantial numbers of VBs you should be aware of the direct consequences.
Firstly, for every VB which you create there is a system memory overhead of around 2K which is used by the runtime for managing it’s own resources.

If you have large numbers of VBs it’s usually an indication that you will be switching your rendering from one VB to another frequently. Since this transition is one of the most expensive operations under DX7 you should strive to avoid this penalty.

Note that although the maximum number of vertices in a VB is 65535 it’s not usually a great idea to approach this kind of size. Since the VB renaming scheme in DX7 requires the driver to find a contiguous free block of AGP memory which the same size as the original buffer it can be easier for the driver to satisfy these requirements if the request is for a smaller chunk of memory. Typically that’s more likely in cases where the VBs are themselves smaller.

Data sets which are as large or larger than the actual physical memory of the machine are typically best handled as in the following case:

**Situation:**

Physical memory: 128Mb  
AGP Heap size: 44Mb  
Vertex Data: 256Mb

**Suggested arrangement:**

For each FVF type that you need to support:
- Create a WRITEONLY VB in AGP memory which is large enough to hold 4K of vertices.  
- Create system memory copy of all the data which matches that FVF. Even when this exhausts all physical memory and forces Windows to virtualize your vertex data.

**Then at render time:**

BeginScene()  
Lock the appropriate VB using WRITEONLY and NOOVERWRITE for the first lock, and using both WRITEONLY and DISCARDCONTENTS on subsequent locks.  
Copy your virtualized vertex data into the matched VB until the VB is full.  
Unlock the VB, and immediately render from it. Continue with this VB and FVF until all data of this format is rendered.  
Cycle through the full set of FVFs until they’ve all been rendered.

EndScene()  
Flip() or Blt()
It is advantageous to reverse the order of rendering from one frame to the next as this allows data which is left in physical memory at the end of one frame to be used at the start of the next without requiring too much intervention from Window’s virtual memory manager.

As before it’s also helpful to arrange for your writes to the VB to visit successive memory locations. Any other access pattern will show poorer performance because of the way the CPU interacts with AGP memory.

**Essential thinking… or “Why it should all make sense”**.

**Why the Vertex Buffer is good in the first case**

VBs are good because they have semantics of ownership and because the driver is able to place them in optimal memory. The correct use of flags when creating and locking is critical to getting best performance.

Without clear semantics of ownership the driver would need to copy the contents of the VB into driver-owned memory with the consequential loss in performance and waste of bandwidth.

**Register renaming in CPUs**

The technique applied by the driver known as “VB renaming” is a technique borrowed from modern high-performance CPU design. For those who have not met the technique before I’ll describe the situation in which a highly parallel register-based processor is able to gain benefits from register renaming.

Suppose the object code contains the following sequence of instructions.

```
  mov   a,1
  mov   b,a
  mov   a,2
```

The first instruction poses no problem. A value of 1 is placed in ‘a’.
The second instruction copies the content of ‘a’ into ‘b’.
The third instruction waits for the second to complete and then a value of 2 placed in ‘a’.

The problem which we seek to address is how to remove the stall implied in the execution of the third instruction. We cannot allow the value of 2 to be written to ‘a’ before the previous instruction completes…

The trick is to introduce a 3rd register which is not directly accessible to the code and which is managed by the processor it-self.
Since the 3\textsuperscript{rd} instruction is guaranteed to destroy the previous contents of ‘a’ we don’t really care where the value is put provided that we subsequently take all mention of ‘a’ as referring to that value. In this case we’ll introduce a shadow register ‘S’ which effectively takes the place of ‘a’ in subsequent instructions. So we can substitute the following code and gain the same effect.

\begin{verbatim}
  mov  a,1
  mov  b,a
  mov  $,2
\end{verbatim}

Now, provided that we subsequently direct all references to ‘a’ towards ‘S’ then we can completely paralellize the 2\textsuperscript{nd} and 3\textsuperscript{rd} instructions. If we achieve this then we have doubled the number of instructions which can be handled at one time.

VB renaming works in the same way but generalizes the renaming method by allowing the driver to perform renaming as many times as the app (and free memory) allows.

Without renaming performance can be pretty unimpressive.

With constructive use of the Lock and Create flags performance can be several times higher.

\textbf{Thinking about the GPU caches}

The post-transform cache is only able to work for you at all when you use the indexed rendering interfaces (DrawIndexedPrimitiveVB using strips or lists).

The memory cache is only working for you when you use high locality of reference.

For these reasons, for maximum performance you should used indexed triangle strips where the vertices have been arranged in the VB in such a way as to make locality of reference of the vertex data implicit in the index sequence.

If it proves impractical to use strips then use indexed lists but otherwise observe all of the same rules.

\textbf{How DX8 changes things}

The two main changes to the handling of VBs under DX8 are the introduction of Index Buffers and lightweight VB changes.

Index Buffers (or ‘IB’s) are a highly efficient way of passing indexing data to the API. Index sets which are constant, or which change only infrequently and which are regularly associated with specific VBs offer excellent optimization opportunities.

DX8 also introduces 32 bit indices and allows VBs to be much larger than before.
In DX8 it is now harder to render without using Vertex Buffers – which reduces the number of opportunities for getting things wrong.

And, for advanced users, there is a new idea of separate DMA streams which allow you to take different components of the vertex data from different VBs. This has the additional benefit of allowing the unchanging parts of the vertex data to be placed in a separate, static VB and therefore allowing the app to access and change only that part of the data which actually needs to change.

It’s fair to say that DX8 makes it easier to get things right, and harder to get things wrong. If you have not considered upgrading to DX8 then you should seriously consider it now.
Appendix 1

Pseudo code for updating dynamic Vertex Buffers.

```
• CreateVB(WRITEONLY, 4K-ish);
• I = 0;

Add:
• Space in VB for N vertices?
  • Yes: { Flag = NOOVERWRITE; }
  • Else
    • No: { Flag = DISCARDCONTENTS; I = 0; }
  • Lock(Flag | WRITEONLY);
  • Fill in N vertices at index I
  • Unlock();
  • DIPVB(I);
  • I += N;
  • GOTO Add;
```

Appendix 2

Six simple rules to take home.

- Always use VBs, for everything.
- If you’re locking a VB then usually you should be using WRITEONLY and one of DISCARDCONTENTS and NOOVERWRITE.
- VBs shouldn’t be in system memory unless the CPU is going to read from them.
- CPUs shouldn’t read from the same VBs that GPUs read from.
- Don’t use ProcessVertices to speed up TnL hardware— it’s almost certain to slow it down instead.
- Use the resources that NVIDIA makes available to developers on it’s web site. We supply header files for both C and C++ users which offer a sample implementation of well handled Vertex Buffers. These can be found on the public part of NVIDIA’s developer web site under the heading “Dynamic Vertex Buffer Header Files” and are at NVIDIA's developer web page.