Optimizing For Hardware Transform and Lighting

Sim Dietrich

NVIDIA Corporation

sdietrich@nvidia.com

HW T&L : The Good News

- Hardware T&L is extremely fast
  - GeForce2 GTS can achieve 22 million drawn triangles per second – Quadro2, Ultra even more
- Using Hardware T&L correctly is very easy
  - In DX7, it all happens through VertexBuffers

HW T&L : The Bad News

- Using HW T&L incorrectly is even *easier* than getting it right
  - Some apps are slower when first ported to T&L!
- Why? Because the obvious way to use VBs is NOT the right way
  - If you replace many DrawPrimitive calls with many DrawPrimitiveVB calls, you will be very disappointed

HW T&L : A New API Path

- The “D3D TnL HAL” Device is new for DX7
- It allows access to :
  - AGP and video memory vertex buffers
  - HW Texture Matrix
  - HW Texture Coordinate Generation “TexGen”
  - HW Fog
  - HW Lighting
  - HW Clipping
  - HW Transform & Projection

The D3D TnL HAL

- The TnL HAL is a different API and driver path than the HAL
- It has different Performance Characteristics
  - Even more oriented towards batching than the HAL
  - Higher memory overhead for VBs
    - They are DDraw Surfaces, so have a 2K memory overhead
  - Very expensive to create VBs
  - Has the potential to be lighter-weight and faster than the HAL

What is a Vertex Buffer, Anyway?

- **There are two answers to this question, one for Static VBs, and one for Dynamic VBs**

- **Static VBs are like textures. You create them at level load time in AGP or video memory and leave them there**
    - **Great for terrain, rigid-body objects**
    - **Not good for skinned, animated characters or procedural effects**
    - **NEVER create a VB at runtime – it can take 100s of milliseconds**

Vertex Buffers are Write Only

- They are not designed for getting results back with ProcessVertices()
- You can never get the result of T&L back
- But that's OK
  - If you need to do collision detection or culling, you'd do best to use a separate simpler database anyway
    - Case in point – Do you really need to walk through U,Vs & diffuse colors when doing collision work?
- VBs should always be WRITE_ONLY – even on non T&L devices

Dynamic VBs

- Dynamic VBs are sort of like like streaming DVD video
  - There is not enough space to hold every possible frame of animation, just like there wouldn’t be enough space to hold a DVD video in ram
  - Plus, many effects are truly dynamic and have an essentially infinite number of possible states
  - The focus is on getting the vertex data from the app to the card as efficiently as possible

The Myths Of Dynamic VBs

- If your data isn't static, you can't use T&L
  - Wrong, VBs were designed to handle Dynamic data, too
- Dynamic T&L is so slow as to be worthless
  - Totally incorrect, Dynamic T&L is still faster than static CPU T&L
- It is hard to manage Dynamic VBs
  - I have a single page of source code to prove this one wrong…

Shared Resources

- The GPU is a co-processor to the CPU
- If you can keep both processors busy, speed will be excellent
- However, to work together, the CPU and GPU must sometimes share resources
  - Textures
  - Frame Buffers
  - Vertex Buffers
- If the sharing is managed poorly, you will get no overlap between the GPU and CPU and performance will suffer

Keeping GPU & CPU Busy

- **Dynamic VBs are a shared resource**
- **CPU must write data into it**
- **GPU must read data out of it**
- **The API tries to ensure that both of these won't occur in the same place at the same time**
- **You can control how strictly access to the VB is managed**
- **Control is managed through three flags :**
  - **DDLOCK_WRITEONLY**
  - **DDLOCK_DISCARDCONTENTS**
  - **DDLOCK_NOOVERWRITE**

DDLOCK_WRITEONLY

- Use D3DVBCAPS_WRITEONLY when creating your VB
- Use ONLY this flag
- Do NOT USE DDVBCAPS_SYSTEMMEMORY, or you will not get AGP or video memory vertex buffers
  - This will require the driver to copy the data into AGP first
  - You could have just put it there yourself and saved the work
- If you specify this cap, you can only lock w/ DDLOCK_WRITEONLY

DDLOCK_DISCARDCONTENTS

- This flag tells D3D
  - "I just need more space, give me a pointer with junk in it, please"
  - Specifying this flag allows the driver to "rename" vertex buffers
  - You are saying that you don't want the object back that you just drew, you are saying that you are going to fill up part of this with new data
  - This prevents stalling the CPU & GPU

DDLOCK_NOOVERWRITE

- DDLOCK_NOOVERWRITE says “I am just appending data to the VB, no need to stall”
- This allows you to append data to a VB without incurring a stall of the GPU & CPU

Using These Flags Together

- Start of Frame – Lock your Dynamic VB with DDLOCK_DISCARDCONTENTS
  - Giving you an empty buffer
- Fill with data to render
- Call Unlock(), then DrawIndexedPrimitiveVB()
- Now, as long as there is room in the VB,
  - Lock with DDLOCK_NOOVERWRITE
  - Append Data into VB pointer
  - Unlock(), and DIPVB()
- If you run out of room, just lock the SAME VB with the DDLOCK_DISCARDCONTENTS and repeat

Other Dynamic VB tips

- Only use ONE dynamic VB
  - An issue with DX7 requires this for performance
  - This implies using the largest FVF you need
- Send triangles in large batches if you can
- NEVER use DrawPrimitive, or DrawIndexedPrimitive, even for Text
  - It will ALWAYS cause a stall of the GPU & CPU
- Check out your system's AGP perf with BenMark from our website
  - GeForce should get 14 million tps @ AGP2X
  - GeForce2 ~22 million w/ AGP 4x

Other VB Perf Tips

- Changing VB is more expensive than changing textures – this is an API thing, not the HW
- Never do your own VB “round robin” – that’s what the DDLOCK_DISCARDCONTENTS flag is for
- Never use ONLY DDLOCK_DISCARDCONTENTS, there are only so many “rename” buffers – use appending, too
- Use only one or two static VBs, and use index lists for different objects within them
- Write into DynamicVBs sequentially for AGP write-combining performance

Source Code

- I wrote an extremely lightweight wrapper for correct Dynamic VB functionality
- On NVIDIA’s Developer Website
- One for C++ heads (like me)
  - DynamicVB.hpp
- One for C types
  - DynamicVB.h

Other Optimizations : Culling

- The CPU is still needed for gross culling
  - View Frustum
    - Sphere, AABB, OBB, Cone, Cylinder
  - Occlusion
    - Don’t use span buffers or C-buffer – too much CPU work
  - Light Culling
    - Turn off lights that are too far away to affect the object
    - Turn point lights into directional if far away
  - Fog Culling
    - Turn off fog if objects are too far from the fog plane

Culling and Clipping

- Do gross culling on the CPU, but leave the Clipping to the GPU
- Expect H/W clipping to be fast (GeForce clipping is essentially free)
- Expect guard band clipping to be very fast
- Don’t cull individual polys unless you cull them very early and they are quite expensive
  - Culling should be at the model or hierarchy level
  - For world geometry at the BSP Leaf or OctTree cube level
- H/W will clip out 1.0 < z < 0.0

NVIDIA

Other Optimizations : LOD

- Use the CPU to perform gross LOD
  - For terrain, don’t use ROAM – too CPU heavy – cheaper to just draw the darn triangles than to figure out which ones to draw and which to skip
  - If you do adaptive terrain, do one where you
    - A) don’t track previous frame’s terrain
    - B) Don’t do screen space error for every triangle
    - C) Can ‘quit’ at a high enough level to keep large batch sizes – Quadtree approaches
- Don’t do View-dependent progressive meshes
  - Again, too much CPU work
- View Independent Progressive Meshes look great and are trivial to use with vertex buffers

Other Optimizations : LOD

- Never try to scale to frame rate by adding or removing triangles in small groups on a T&L card
  - You are just wasting CPU time
  - 90% of frame rate drops are CPU or fill-bound, not triangle bound
  - Do less LOD calculations when frame rate drops, not more, save the CPU time
  - Reduce depth of volumetric effects, especially when player is near
  - Reduce particle counts, especially when player is inside the particle system
    - Player won’t notice

Other Optimizations : Lighting

- If multi-pass, you often don't need it on for both passes
- Turn on & off lights per object based on distance from light
- Turn off per-vertex material properties if you don't need them
  - Using the per-vertex diffuse for the diffuse material is expensive – use it wisely
- Turn off local viewer for specular lighting if not needed
  - If you are not sure, you probably wouldn't notice
- Turn off SpecularEnable if you aren't using specular for this pass

Other Optimizations : Vertex Cache

- GeForce GPUS have a ~10 entry FIFO vertex cache
  - Post-transformed vertices
- If you reuse an indexed triangle within 10 vertices, you save the AGP B/W & transform cost
- If you don't index, or don't re-use, you pay both AGP & transform again
- The fastest primitive is indexed strips, sometimes only the cost of one short per triangle if all reside in cache
- Use the NVStripifer on our website to optimize your models

Other Optimizations : Triangle Size

- **Little known facts**
  - **Every app is fillbound**
  - **Every app is Xform or setup bound**
  - **- In different parts of the same scene**
  - **Two Engines in parallel – vertex and pixel**
- **Given fill rate, b/w and max xform/setup rate you can determine what the optimal triangle size is for a GPU**
  - **For GeForce, with a few lights on it's about 100 pixel triangles**
  - **Bigger Tris get you temporarily fill bound**
  - **Smaller Tris get you vertex bound**
  - **More expensive vertices (more lights or xform work) need bigger triangles to balance out**

Other Optimizations : Triangle Size

- If you are temporarily fill bound (Tri too big), you lose xform rate
- If you are xform bound (xformed vertex cache is full) you loose potential fill rate
- This is one reason why you may not see the optimal vertex or fill rate
  - If one engine is backed up, the other will eventually idle – and you never get this time back
  - When you are drawing the sky, you lose potential triangles
  - This means that you can tessellate down to the optimal triangle size in these cases for FREE

Other Optimizations : Stat Driver

- NVIDIA has provided a Statistics Driver for registered developers
  - Written by Ken Hurley
- You install two parts
  - A monitoring program
  - A special stats driver
- You start the monitoring and then run your app
  - Or, you can use a hotkey to toggle the stats collection
- Quit your app and see where you are forcing a SpinLock()
  - This means the CPU & GPU are idle

Stats Driver

- SpinLock() means the CPU is waiting on the GPU to finish with something
  - Usually a shared resource
- Most apps spend quite a bit of time here
  - This time is totally wasted!
- The Stat Driver monitor will tell you where your d3d & driver CPU time is going
- Your app should be spending > 60% of the Driver time in DrawIndexedPrimitiveVB
- SpinLock() should be < 5%

The log file can help you track down the culprit

Summary

- T&L is Faster, but it is different
  - The first time you port to DX7, you will almost certainly do it wrong! ;(
- Use Static VBs for static geometry
- Stream vertex data through DynamicVBs
- Use the stat driver often when working on rendering code
  - Take out stalls as soon as they are introduced
    - Texture Locks
    - FB or ZB Locks
    - VB Locks w/out proper flags
    - DrawPrimitive or DIP, not the VB calls

Questions…

?

Sim Dietrich

Sim.dietrich@nvidia.com