Optimizing Direct3D for the GeForce 256

Douglas H. Rogers
Please send me your comments/questions/suggestions
drogers@nvidia.com

Transform and Lighting (T&L) Acceleration under Direct3D

To enable hardware Transform, Lighting and Clipping support under Direct3D, applications must :

1. Use Direct3D version 7. (Earlier versions do not support HW T&L)
2. Enumerate and select the TnL HAL, not the normal HAL or refrast.
3. Use Vertex Buffers to store the vertex data in model, world or view space
4. Call DrawIndexedPrimitiveVB or DrawPrimitiveVB. DrawIndexedPrimitive and DrawPrimitive will give you T&L, but at slower speeds

Applications will **NOT** get hardware T&L support if :

1. ProcessVertices is used. This is always performed in software.
2. D3D_TLVERTICES are used. They are already transformed and lit.
3. DirectX 6 or lower is used. T&L is not supported in this API.
4. The HAL, RGB, Ramp or refrast devices are selected. Only the TnL device supports transform and lighting.

There are many performance factors involved in achieving peak triangle rates on the GeForce256, but here are a few quick things to keep in mind :

1. There is no access to post transformed and lit vertices. Transformed vertex, texture coordinate, color and normal information is internal to the hardware and is not stored in memory. There is no read-back of screen space vertices, view space, normals, clip extents, outcodes or any other output except for screen pixels.
2. Never specify D3DDP_WAIT when calling DrawIndexedPrimitive or DrawIndexedPrimitiveVB, or you will stall the hardware
3. When Locking vertex buffers or DirectDraw surfaces, always specify DDLOCK_WRITEONLY and DDLOCK_DISCARDCONTENTS or DDLOCK_NOOVERWRITE as appropriate.
4. Indexed primitives are faster than inline vertices (indexed vs. non-indexed)
5. Indexed strips are the fastest primitive, even if you must create degenerate triangles. The downside is potentially greater function call overhead when compared to indexed triangle lists.
6. Do not specify DDLOCK_NOSYSLOCK in your lock. There are some issues with this flag.
7. Do not use only a small number of vertices in the vertex buffer and never use less than 24. There is an overhead of 1.5 to 2 kilobytes for each vertex buffer.

8. The GeForce 256 transfomes and lights 15M *vertices* per second. To achieve high triangle rates, you must re-use the vertices. If you specify completely independent triangles, the maximum triangle rate you will achieve is 5M. You get vertex reuse only with index lists. Re-specifying the same vertex twice is **not** reusing it.
9. Do not duplicate render state commands. Worse is useless renderstates. Do not set a renderstate unless it is needed.
10. Only send new matrices and lights if they are needed.
11. The graphics chip is another processor running in parallel. Try not to stall it with synchronization. If you render a vertex buffer, then immediately lock it, the rendering must complete before the lock returns successfully. Lock on vertex buffers will always wait, even if you specify DONOTWAIT. If you must do this, specify DDLOCK_DISCARDCONTENTS or DDLOCK_NOOVERWRITE.
12. Use only the lights you need. The hardware lights are applied per triangle. Most of the time, two or three lights is sufficient. Adding more will probably not add visually to the scene and will slow down the rendering unnecessarily.
13. Trivially reject all the geometry that is not in the view frustum.

HAL vs. TnL

There are two devices that are enumerated in the GeForce 256, the Hardware Abstraction Layer (HAL) and the Transform and Lighting (TnL). The HAL device is a rasterizer only and does not support hardware TnL, even if the hardware is capable of supporting it. You must use the TnL device to get hardware transform and lighting.

The major blocks of the GeForce hardware pipeline are:

- Transform
- Lighting
- Projection
- Clipping
- Divide by W

Transformation

The GeForce has a transform engine that transforms vertex data, but does not write out intermediate transformed information. You do not have access to transformed vertex data before the geometry is rendered as pixels. There are several ramifications to this:

If you need to have access to transformed data, you will have to perform the transformations in software. Performing transformation in software should be done on vertex data that is in system memory. You may have to duplicate your data in system and video memory. For collision detection, for example, you may transform bounding boxes in software, then render the object using hardware transformations.

Mixing Software and Hardware

You may perform some of the functions in software and use the hardware pipeline to perform the remaining tasks. For example, you might want to transform your vertices in software, but you can light, project, clip and w divide in hardware. As another example, you can perform your own lighting calculations, and perform transform, project, clip and w divide tasks in hardware. This should be done only when you are attempting to do something that is beyond what the hardware can do, and should only be used as a fallback.

Use as much of the 3D hardware pipeline as you can. If you do transformations is software, do the rest of the 3D pipeline in hardware.

Culling
Clipping on GeForce 256 is extremely fast and is very efficient when combined with a guard band. Applications should still perform trivial rejection of large blocks of data in software, however. Objects or terrain blocks that are completely off screen should be determined by a bounding box or sphere and not sent down the pipeline. Any bounding box information should be kept in system memory.

Perform trivial rejection where possible.

Creating Vertex Buffers
Indexed Vertex buffers will yield the most performance from the GeForce. There are some dependencies and flags are helpful to understand.

When you create a vertex buffer there are several options. One is the destination memory and the other is how it is managed.

D3DVBCAPS_SYSTEMMEMORY will place the vertex buffer in system memory;
Set this flags:

- When you are performing software transforms on the vertex buffer (even if you are using the TnL device). If the vertices are placed in system memory, transforming them in hardware will be slower.
- You are using the HAL

Do not specify this flag:

- You are using the TnL device and want the vertices to be transformed in hardware

D3DVBCAPS_WRITEONLY
Set this flag

- If you do not need to read the contents of the vertex buffer. Currently this bit must be set for the driver to perform vertex buffer renaming (see below).
- Whenever you can.

Do not set this flags

- If you need to read the contents of the vertex buffer. This will kill performance, though. Keep a system copy of your data if you need to perform partial updates of a vertex buffer, then copy the whole thing over. Better yet, keep the dynamic data in one vertex buffer and update the whole thing and keep another vertex buffer for the static geometry.

Do not create vertex buffers at runtime. These take a long time to create.
Do not make a lot of small vertex buffers, each takes about 2K of overhead.

Locking Vertex Buffers
Locking your vertex buffers is how you get access to it, but the flags you specify can drastically change your performance.

Locking vertex buffers will always return a valid pointer;
DDLOCK_WAIT and DDLOCK_DONOTWAIT are ignored and should not be specified.
Locking vertex buffers always waits.

DDLOCK_WRITEONLY - Indicates that the surface being locked will only be written to.
Same as CreateVertexBuffers

DDLOCK_NOOVERWRITE - Indicates that no vertices that were referred to in Draw*PrimtiveVB calls since the start of the frame (or the last lock without this flag) will be modified during the lock. This can be useful when one is only appending data to the vertex buffer, or writing to portions of the buffer (even the middle or beginning) that you know won't be interfering with pending drawing commands this frame.

Set this flag:
- If you re-lock a vertex buffer that is being modified and you are not changing data specified is a previous DrawPrimitive command. In other words, you are indicating to the system that you are not modifying anything that you previously asked it to render.
- Whenever you can

Do not set this flag:
- If you lock a vertex buffer that has a pending Draw against it.

DDLOCK_DISCARDCONTENTS.   Indicates that no assumptions will be made about the contents of the surface or vertex buffer during this lock. Direct3D or the driver may provide an alternative memory area as the vertex buffer.

Set this flag:
- When you intend to overwrite the entire vertex buffer during the lock
- Whenever you can

Do not set this flag.
- When you will make only partial updates to a vertex buffer during the lock.
- When you intent to read the data in the vertex buffer

Using the create and lock flags has a tremendous impact on performance.

Static Geometry
Static geometry is geometry that is defined once and only manipulated by matrix operations. Basically, any vertex buffer that you create/lock and unlock only once.
This is data such as rigid object or perhaps terrain. For static geometry, store it all in non-system memory at load time. If your static geometry is too big, treat it like dynamic geometry.
Optimize should be called, but it does not perform any function at this time. Once you call Optimize on a surface, you cannot lock it, so you cannot access the data.

Vertex Buffer Renaming
Vertex buffer renaming may occur when you have created vertex buffer with the DDLOCK_WRITEONLY and DDLOCK_DISCARDCONTENTS attributes and you lock a vertex buffer that has an outstanding render applied to it. A new vertex buffer from the free pool may be substituted in its place. This so you will not wait for the buffer to become free, but only works if you indicate that you don't care what the contents are via the DDLOCK_DISCARDCONTENTS flags. If you do not specify DDLOCK_DISCARDCONTENTS, you will wait for any outstanding rendering to occur before the lock finishes. This is to be strongly avoided.

Dynamic Geometry
Dynamic geometry is all geometry that is constantly written during runtime. This can include terrain geometry that is too big to fit into memory so only sections will be resident in non-system memory. This also includes geometry that is transformed in software and written to a vertex buffer.

Dynamic geometry is handled by creating reusable pages of vertex buffers that allocated to object data dynamically. You can place more than one object in a page (remember to use DDLOCK_NOOVERWRITE if you use more than one lock on a page). If you need to re-use the vertex buffers within a frame, then treat the buffers in a round robin manner and use the DDLOCK_DISCARDCONTENTS flag.
The sequence round robin is:

- Lock the vertex buffer
- Completely fill it
- Draw
- Move on to the next one.
- After you have finished with the last buffer, move back to the first. Vertex buffer renaming may occur if the buffer you are requesting is still rendering.

The flags will allow the vertex buffer pages to be managed by the driver and Dorect3D to give you the best performance. There is no "right size" of vertex buffer page, but too small and there is a lot of overhead. Too big, and you will wait longer and possibly use up more memory. Microsoft recommends 500-2000 vertices, but much greater than 64KB of total size.

Losing and Restoring Vertex Buffers

The application is responsible for restoring the contents of the vertex buffer when it is lost. The app checks the return value of EndScene call and if it returns DDERR_SURFACELOST or DDERR_SURFACEBUSY, you have lost all your surfaces, including and vertex buffers that are in device memory.

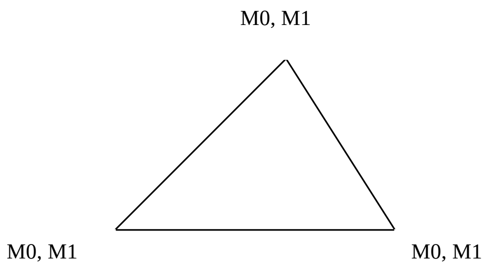
The app should then proceed and do restore all surfaces to restore everything. For unoptimized vertex buffers, the app then should lock and fill the vertex buffers with the vertex data again. In case of optimized vertex buffers, the app needs to destroy and re-create them and then re-optimize them.

You will lose you vertex buffers if the use performs and ALT+TAB sequence. You can disable this by capturing WM_SYSCOMMAND messages and do not send them on for processing by DefWindowProc().

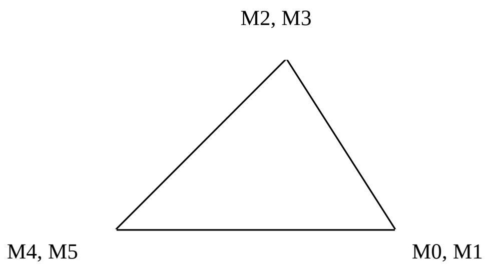
Progressive Meshes and Dynamic Terrain

Some apps recompute mesh complexity on the fly. Rather than recreating or overwriting vertex buffers with the new mesh data, create a vertex buffer with the highest level of detail. Create different index lists that reference different geometry within this one list for each level-of-detail.

Vertex Blending



Possible



Not supported in Direct3D

The GeForce 256 supports 2 blending matrices per vertex. These are not divisible within a triangle. In Direct3D, blending matrices are applied on a per vertex basis and can be changed per primitive basis. You cannot reassign the blending matrices within the triangle. If you require more sophisticated vertex blending than this, you have several options :

1. Use the two most important matrices and discard the others
2. Create a two matrix approximation of what you are trying to perform
3. Perform the skinning transform in software. You can still use the rest of the TnL pipeline, though. Pass in ‘root model space’ or world or view space geometry.

Vertex Format
Select the smallest vertex footprint that you can. Use the smallest flexible vertex format you can, but also remember that you can have multiple texture coordinate sets in a vertex.
You can select the between the two texture coordinate sets rather than duplicating vertices. If you have two textures for one vertex, for example, you can use a different texture set for each texture, rather than duplicating vertices.
Use whichever flexible vertex format (FVF) is smaller.

When you change the FVF format, the driver must flush out the rendering pipeline, so this is an expensive operation and should be kept to a minimum.

Matrices
Do not set a matrix unless it has been changed, keep a dirty bit for modifications to matrices and only perform a SetTransform call if the matrix has changed.

Lights
The GeForce has 8 hardware lights. This is per primitive. You may have many more than eight in a scene and they can all be created through a SetLight call. You must make sure that only 8 lights are active at one time, though. A directional light is very fast and the spotlight is the most expensive light to calculate.

The first two lights have only slight impact on performance, but there is not a linear degradation, however. More than two lights cause performance to falloff quickly. Artists say that the most important lights are the first two, so where you can, use the closest two lights only. Disable any lights that do not affect the surface that you are rendering. Experiment how the lights affect performance. Use bounding tests compared to the light range to turn on and off lights as needed.

Set specular enable to false whenever possible (ie if you have lighting off.). Also turning D3DRENDERSTATE_LOCALVIEWER to false saves transform cycles when exact specular highlights are not necessary.

The D3DTOP_DOTPRODUCT3 texture blending operation is very cool. You can get some very good per pixel effects with this. You can also use it to perform per pixel bump mapping with surface normal perturbation.

The GeForce cannot map the vertex specular component to other colors so the following calls will not work as expected :

```
SetRenderState(device, D3DRENDERSTATE_AMBIENTMATERIALSOURCE, D3DMCS_COLOR2);
SetRenderState(device, D3DRENDERSTATE_DIFFUSEMATERIALSOURCE, D3DMCS_COLOR2);
SetRenderState(device, D3DRENDERSTATE_EMISSIVEMATERIALSOURCE, D3DMCS_COLOR2);
```

Apps should be sure to turn off per-vertex color material support when appropriate for increased performance.

WindowsProc and WM_ERASEBKGND

If you use the WindowProc callback, trap the *WM_ERASEBKGND* and do not call DefWindowProc when this event occurs to avoid an unnecessary memory clear from windows. You can prevent this message from occurring when you register your window class. Set *hbrBackground* to GetStockObject(NULL_BRUSH). This is the brush used to paint the background, so NULL_BRUSH means don't paint.

Textures

The supports two simultaneous textures. The best texture performance will be using compressed textures, either 8-bit palletized or DXTn.

Compressed Textures

The GeForce supports all the Direct3D compressed texture formats. Direct3D compressed textures use 4 bits / pixel and are very compact. They should be the default texture format that you use unless the compression scheme yields undesired results. To create the compressed textures, you can create your uncompressed textures in system memory, then create a compressed texture in video memory then blt from system to video memory. Compressed textures are NOT swizzled.

Cube Environment Maps

You do not need to completely re-render all your cube maps if you want to reflect a scene. You can save the statically render parts of the scene, like walls and table in a static map. To render the dynamic map, copy the static map (and possibly the Z buffer) into a new cube map, then render the moving objects into the static scene. Rinse. Repeat.

You should render your cube map faces with low-detail versions of the environment, also using low res maps if possible.

Cube maps can be compressed, although this will not be worth it except for relatively static maps. The GeForce cannot render to compressed or palettized textures.

The render-to-texture performance was significantly enhanced in the last public driver release ~3.53 or so, so updating cubemaps with rendered geometry should perform significantly faster compared to earlier TNT2 drivers.

Procedural Textures

Bltting to the texture surface is preferred to the lock function for all texture updates, if you must lock a texture specify the flags WRITEONLY and DISCARDCONTENTS when you are able. Also, Load is even faster than BLT under Dx7.

General

- Disable specular and alpha blending until you need it.
- Set D3DRENDERSTATE_NORMALIZENORMALS to true and leave it there. The GeForce will normalize for you and the driver will be faster.

Direct3D 7.0

The new attenuation factor is inverted from Direct3D 6.0 $\frac{1.0}{A_0 + A_1 \cdot d + A_2 \cdot d^2}$

The default case for blt, flip and lock is to wait. If you do not want this behavior and wish to service the DDERR_WASSTILLDRAWING return code, specify the following flags:

DDBLT_DONOTWAIT
DDBLTFAST_DONOTWAIT
DDLOCK_DONOTWAIT - This does not apply to a vertex buffer lock.
DDFLIP_DONOTWAIT

Materials in Direct3D are more flexible, so you must set the source for the lighting equations. This code forces the material to provide all the lighting information.

```
SetRenderState(D3DRENDERSTATE_DIFFUSEMATERIALSOURCE , D3DMCS_MATERIAL);
SetRenderState(D3DRENDERSTATE_SPECULARMATERIALSOURCE, D3DMCS_MATERIAL);
SetRenderState(D3DRENDERSTATE_AMBIENTMATERIALSOURCE , D3DMCS_MATERIAL);
SetRenderState(D3DRENDERSTATE_EMISSIVEMATERIALSOURCE, D3DMCS_MATERIAL);
```

The default lighting state in enable, so if you are performing your own lighting, you must disable it or you may get black polygons.

Specular light has been added as a texture argument. Under our 3.53 drivers, you must enable specular (ie set D3DRENDERSTATE_SPECULARENABLE to TRUE) to apply it. This is a bug and should be fixed for our next web release.