Top 5 things that kill performance under DX7

1. Vertex Buffer Locking problems - If you are not using vertex buffers correctly, the driver will stall and wait for the hardware to finish using the buffer. . A “spin lock” is used to describe when the CPU is wait for a shared resource. When the application is requesting a shared resource (via a lock) and the GPU is using that resource, the CPU must stall until the resource is free.

   The key to avoid spin lock is to manage shared resources properly. Vertex buffers are managed differently depending what you are trying to accomplish.

   If you are constantly updating the vertex data in the vertex buffer (dynamic vertex buffer) then you should use the D3DLOCK_DISCARDCONTENTS and D3DLOCK_NOOVERWRITE methodology. The correct way to use this is in the following way:

   Create Vertex Buffer with D3DVBCAPS_WRITEONLY, approximately 1000-20000 are good numbers for the buffer sizes, with 10000-20000 vertices being the optimal size. For this size vertex buffers you will also have to sort the indices.

   When adding vertices to the vertex buffer, check if there is room in the allocated vertex buffer. If so, use the flag D3DLOCK_NOOVERWRITE and add vertices to the end of the buffer. Use the offset you placed the vertices in the buffer when you call DrawIndexedPrimitiveVB.

   If there is no room in the vertex buffer, then call the lock function with the D3DLOCK_DISCARDCONTENTS flag set. This will tell the driver to give you another vertex buffer and keeps the driver from stalling waiting for the hardware to finish with the vertex buffer.

   See the paper “Using Vertex Buffers” for more detail about vertex buffers.

2. Flip/Blit - A lot of games complain about the throughput of the hardware when they sit and spin in a loop to govern the frame rate. If your flow is as follows:

   ```
   Physics();
   GameAI();
   RenderTriangles();
   Flip();
   WaitTillFrameTime();
   ```

   You are stalling the CPU waiting for the frame to go by. Concurrency is the key to achieving lightning fast games. You must treat the GPU as a second processor and try to achieve as much concurrency as possible. A quick fix, that isn't 100% effective, but will help some is to rewrite the main game loop as follows:

   ```
   Physics();
   GameAI();
   WaitTillFrameTime();
   RenderTriangles();
   Flip();
   ```

   That way at least you get some concurrency going. The real trick is to architect your game engine from the beginning with concurrency in mind.

3. Texture Map Locking - This can happen for a variety of reasons. Some of the things that will absolutely kill performance are locking the rendering target before the hardware has finished with it. This happens a lot with 2D operations such as using the GDI to write text to the backbuffer. A better solution is to write the text to a separate buffer and then blit it to the backbuffer. An even better solution is to blit individual characters to a texture surface at the beginning of the program and blit using offsets from that texture to the backbuffer, this avoids any stalls in the driver.

4. DrawTriangles - If you application spends a lot of time drawing triangles, this can be a good thing or a bad thing. The way you can tell if it is a good thing is if the total amount of time spent drawing triangles is very low when compared to the time spent in your application. Typical numbers should be less than 10% spent drawing triangles. If this number is high, say 40% or more of the CPU cycles, then something can be amiss. Things to check in this case would be calling DrawPrimitive calls without vertex buffers. D3D copies these to vertex buffers, internally so you should put them in vertex buffers anyway. Another common mistake is to send a lot of small batches of vertices to the driver. The more you in small batches the more calls that happen internally. You should also note the D3D will batch up vertices that are sent in batches of less than 24. There is a lot of overhead with small batches of vertices sent to D3D and/or the video card driver.

5. Reading from AGP memory/ video memory. This is by far the most time consuming problem. Several factors affect the speed of reading/writing from AGP/video memory. If the hardware is using the surface/buffer the driver stalls waiting for the hardware to be done with it. Secondly, reading from video memory requires the data to be copied across the bus, so the CPU can have access to it. AGP memory reading can be done directly, but when the AGP memory is allocated for the video card the memory is Non-Cached/Write Combining memory, which effectively disables the L1/L2 caches. If you really must read vertex data, then it is suggested that you keep a copy of the data around, since all manipulations will be done in a cached environment. When writing AGP memory (like a vertex buffer) you should write your data sequentially to get maximum AGP write performance. Scattered writing of your data is slower than writing sequentially.

6. Vertex Data ordering - We have found that the size of your FVF vertex buffers and ordering of the indices makes a huge difference on the cards caching mechanisms. Look for an upcoming paper of the NVIDIA website. For indexed primitives, sort your vertex buffer vertices by the order they appear in the index list.