Windowing System on a 3D Pipeline

Nick Triantos (nick ‘at’ nvidia.com), NVIDIA Corporation

Feb 11, 2005

How 3D primitives are drawn

The overwhelming majority of 3D primitives are drawn on current graphics hardware by decomposition into triangles (or triangle lists, triangle strips, etc.) Quads, points, and lines are the other primitives used. The first stage of the graphics pipeline breaks down these primitives into individual vertices. These input vertices are run through a programmable vertex engine, which determines the positions in screen space. The resultant vertices are reconstructed into the originally-input primitive (triangle, point, line), and fed to a rasterizer. The rasterizer selects the fragments[1] affected by the primitive, interpolates the per-vertex attributes, and feeds these fragments with their interpolated attributes through to the programmable fragment processor. The fragment processor determines a color (or colors, see below) for each fragment. Finally, the ROP (Raster Op) portion of the pipeline determines if the resultant fragment should be drawn into the frame buffer, and if so, blends the fragment’s color with the color in the frame buffer.

Vertex processing pipeline

The vertex processors are typically configured to DMA the vertex positions, and optionally colors, and texture coordinates, from arrays of vertex data. The GeForce hardware supports float (fp32) data types, plus uint8 for colors (RGBA). The vertex data can be fed as a stream of data, or can be stored in an array which is indexed by a vertex index list.

Once the vertex data is fetched by the GPU, a vertex program is run on each vertex. The program determines the position in homogeneous clip space of the vertex, and can also modify or generate the color and texture coordinate attributes. The GeForce 6 processors have up to 6 vertex engines that all run in parallel. Each vertex engine

[1] A fragment is similar to a pixel, but also contains state such as position, depth, color, texture coordinates, and other attributes

executes up to 5 math operations (float4 + float) per cycle. In addition, the vertex engines are able to sample from 4 texture maps. The instruction set is similar to many SIMD extensions found in modern CPUs.[2] Programs can be up to 512 instructions in length, though up to 64k instructions can execute (by looping).

Below the vertex engines, the vertices are reconstructed into quads, triangles, points, or lines, which a rasterizer then breaks into fragments.

Fragment (pixel) processing pipeline

These fragments are then fed into the programmable fragment engines. The purpose of these engines is to determine a final color (or colors[3]) for the fragment. The fragment engines can also optionally modify the fragment’s depth. The GeForce 6 processors have up to 16 fragment engines on each chip, to support a maximum throughput of up to 16 fragments per clock.

The instruction set is similar to that of the vertex engines, though there are some differences between the two instruction sets.[4] However, the fragment engines perform quite differently from the vertex engines. The fragment engines are more optimized for accessing texture maps, and have a more robust texture caching system. The execution units are also quite different; it is possible to execute up to 12 math operations (3 * float4) per fragment engine per clock. Programs may be up to 2048 instructions in length, though up to 64k instructions can execute (by looping).

Raster Ops (ROP)

The last stage of the pipeline is the ROP. These units grab the output colors from the fragment engines, coalesce the data into memory-word-sized chunks, and then blend this data into the frame buffer, if the fragment passes depth and stencil tests. Many

[2] For more info, see the ARB_vertex_program OpenGL extension specification at http://www.nvidia.com/dev_content/nvopenglspecs/GL_ARB_vertex_program.txt

[3] GeForce 6 supports writing up to 4 colors to 4 separate buffers, per-fragment. For more info, see the ARB_draw_buffers OpenGL extension specification at http://www.nvidia.com/dev_content/nvopenglspecs/GL_ARB_draw_buffers.txt

[4] For more info, see the NV_fragment_program2 OpenGL extension specification at http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_fragment_program2.txt

Year	Product	Core Clk (Mhz)	Mem Clk (Mhz)	Mtri/sec	Mtri/sec per-year Increase	Mvert/sec	Mvert/sec per-year Increase	Single Texture Fill Mpix/sec	Single Texture Fill per-year Increase
1998	Riva ZX	100	100	3	-	1	-	100	-
1999	Riva TNT2	175	200	9	300%	2	200%	350	350%
2000	GeForce2 GTS	166	333	25	278%	24	1186%	664	190%
2001	GeForce3	200	460	30	120%	33	141%	800	120%
2002	GeForce4 Ti 4600	300	650	60	200%	100	300%	1200	150%
2003	GeForce FX	500	1000	167	278%	375	375%	2000	167%
2004	GeForce 6800 Ultra	425	1100	170	102%	638	170%	6800	340%
		Increase over 6 years		56.7		637.5		68.0	

Table 1: Growth of various performance metrics from 1998 - 2004

blend modes are available, but this unit is not programmable in the same way as the vertex and fragment units.

Strengths of the 3D pipeline

Since 2000, the amount of horsepower applied to processing 3D vertices and fragments has been growing at a staggering rate (see Table 1).

This trend is likely to continue, for as long as new applications demand more performance.

The texture mapping hardware provides for very fast mag-/minification of images, as well as arbitrary rotation, shearing, and other distortions. Many texture formats are supported (including YCbCr formats, in some GPUs), though palettized formats are no longer directly supported in many 3D graphics pipelines.

The blending modes are flexible, and become more flexible with each new generation of GPU. Standard alpha blending (α * src + (1- α) * dst) is supported, as well as over 100 other blending modes.

The programmability of the vertex and fragment engines also allows for huge flexibility in the types of effects. Everything from eye-candy transitions, to per-window gamma adjustments and color space conversions are possible.

Challenges with layering a windowing system on a 3D pipeline

There are numerous challenges which must be overcome by a windowing system design, if it hopes to integrate well with the 3D pipeline.

Resource Management

Today, it is not uncommon for a 3D application to greedily consume video memory with requested textures and geometry. For a full-screen game like Quake or Doom, this might not be a problem, as no other applications should need access to video memory. But when an application is running within a window, the system must facilitate applications cooperatively sharing video memory.

Maintaining interactivity

The 3D pipeline is designed for high throughput, but is not typically optimized for low latency. It is now possible to create very simple 3D geometry which, when run through complex programs, can run for many seconds to draw just one triangle. Careful planning is needed to ensure that the desktop can remain responsive in these situations. Synchronization between the CPU and completion of some rendering operations also needs to be properly architected.

Backward compatibility

Several features are difficult to emulate when rendering through a 3D pipeline and compositing desktop system. For example, video overlays can not be used, since other layers may need to be rendered above the video YUV data. Another example is the “feature” on some operating systems that requires that console messages be rendered onto the server’s attached display, even if X is running. These messages are rendered by FCODE which must be made aware of the possible double-buffering of X.

Best performance comes from a full pipeline

Typically, the best performance is achieved when 3D pipeline state changes can be amortized across the rendering of many primitives and many pixels. Performance can be several orders of magnitude below peak in cases where applications change state too frequently.

Conclusion

Obviously, several modern platforms exist already which prove that it is possible to build an interactive, flexible, and attractive windowing system. But moving to the 3D pipeline is not a panacea; there are many new challenges that must be overcome in order to design a successful 3D-based windowing system. NVIDIA is very interested in participating in this process. We look forward to collaborating with you in designing and implementing this new system.