The Direct3D 10 System

David Blythe
Microsoft Corporation

Abstract

We present a system architecture for the 4th generation of PC-class programmable graphics processing units (GPUs). The new pipeline features significant additions and changes to the prior generation pipeline including a new programmable stage capable of generating additional primitives and streaming primitive data to memory, an expanded, common feature set for all of the programmable stages, generalizations to vertex and image memory resources, and new storage formats. We also describe structural modifications to the API, runtime, and shading language to complement the new pipeline. We motivate the design with descriptions of frequently encountered obstacles in current systems. Throughout the paper we present rationale behind prominent design choices and alternatives that were ultimately rejected, drawing on insights collected during a multi-year collaboration with application developers and hardware designers.

Keywords: graphics systems, programmable graphics hardware, programmable shading

1. Introduction

The rendering pipeline architecture embodied by the OpenGL [Segal and Akeley 2004] and Direct3D [Gray 2003] systems have evolved substantially over the past decade. More dramatic changes have occurred over the last five years with the transition from a fixed-function pipeline to a programmable pipeline. While the evolutionary progress has been rapid, each step has reflected a compromise between generality, performance, and cost.

We have made efforts to understand and build a system to address requirements in the many applications for graphics accelerators (presentation graphics, CAD, multimedia processing, etc). However, we direct significant attention to the demands made by interactive entertainment applications. These applications manage gigabytes of artwork in the form of geometry, textures, animation data, and shader programs (content), consuming large amounts of system resources (processing, memory, transfer bandwidth) to render rich and detailed images at interactive rates. The combination of agile rendering control and large amounts of data poses significant system design challenges. Solutions to these challenges are reflected in many aspects of the system design.

Like previous versions of Direct3D, Direct3D 10 was designed in three-way collaboration between application developers, hardware designers, and the API/runtime architects. Detailed communication between the parties over the three-year design process was invaluable in fostering a deeper understanding of application issues with API and hardware and to similarly disseminate cost and complexity tradeoffs in various hardware proposals. Surveys during the development of Direct3D 10 revealed that application developers frequently struggled with the following limitations and addressed them with the accompanying mitigation strategies:

1. High state-change overhead. Changing any type of state (vertex formats, textures, shaders, shader parameters, blending modes, etc.) incurs a high overhead. The work-around is to reduce API state changes by sorting objects by state, reducing appearance variation, or using shader-based techniques to resolve state within the shader . An example of the latter is packing multiple texture images into a single texture map (sometimes called a *texture atlas*), combined with texture coordinate transformations to index the desired subimage.
2. Excessive variation in hardware accelerator capabilities. Applications respond by writing a small number of "code paths" customized to a few families of implementations plus a single minimum-common-feature-set path to cover the remaining implementations. This problem encompasses variations in feature sets, resource limits, arithmetic precision, and storage formats.
3. Frequent CPU and GPU synchronization. The traditional pipeline allows limited re-use of data generated within the pipeline as input to a subsequent processing step. Render-to-texture techniques are an example of a mechanism through which a rendered image can be later used as a texture map with minimal CPU intervention. However, generating new vertex data or building a cube map requires more coordination or communication between the CPU and GPU, reducing efficiency.
4. Instruction set and data type limitations. The vertex shader has led the way in terms of precision requirements and support for expressiveness in terms of traditional flow control constructs. The pixel shader has followed, but neither pixel or vertex shader supports integer instructions and the pixel shader accuracy requirements for floating-point arithmetic have been weakly specified. Applications either live without the extra capabilities or emulate them using, for example, table-based function evaluation.
5. Resource limitations. The number of dependent texture reads, bound textures, program instructions, etc. have been modest. Applications must scale algorithms back or split them into multiple shader passes. This has inspired research into automatically partitioning shader programs [Chen et al. 2002; Riffel et al. 2004].

2. Background

Our system builds on the application-programmable rendering pipeline now prevalent on PCs, workstations, and game consoles. The contemporary pipeline has two programmable stages, one for processing vertex data (vertex shader) and a second for processing pixel fragments (fragment or pixel shader).

The motivations and tradeoffs in designing an early programmable vertex shader are described in [Lindholm et al. 2001]. Programmable pixel shading has followed a similar trajectory to vertex shading with a few deviations. The evolution of programmable vertex and pixel shaders can be divided into four generations (including Direct3D 10), summarized in Table 1.

The evolutionary trajectories are largely a reflection of benefit (or necessity) vs. hardware cost. The slightly differing requirements surrounding dynamic range, precision, and texture mapping resulted in different trajectories for the two shader types with a general trend towards feature-set convergence. [Haines 2006] describes additional details regarding the feature set evolution.

Hardware pipeline implementations achieve high processing

Feature	1.1 2001	2.0 2002	3.0 2004[†]	4.0 2006
instruction slots	128	256	≥512	≥64K
	4+8[‡]	32+64[‡]	≥512	
constant registers	≥96	≥256	≥256	16x4096
	8	32	224	
tmp registers	12	12	32	4096
	2	12	32	
input registers	16	16	16	16
	4+2[§]	8+2[§]	10	32
render targets	1	4	4	8
samplers	8	16	16	16
textures			4	128
	8	16	16	
2D tex size			2Kx2K	8Kx8K
integer ops				✓
load op				✓
sample offsets				✓
transcendental ops	✓	✓	✓	✓
		✓	✓	
derivative op			✓	✓
flow control		static	stat/dyn	dynamic
			stat/dyn	

Table 1: Shader model feature comparison summary
[†]specification released in 2002, hardware in 2004; [‡]texture load + arithmetic instructions; [§]texture + color registers; dashed line separates vertex shader (above) from pixel shader (below)

throughput by exploiting the natural independence between vertices and between pixels fragments. Multiple instances of vertex and pixel shaders are used to process independent vertices and pixel fragments in parallel. Hardware implementations typically include a larger number of pixel shaders than vertex shaders reflecting the higher ratio of pixels to vertices in a typical rendering workload [Montrym and Moreton 2005]. This characteristic also influences the cost of pixel shaders relative to vertex shaders since pixel shaders are more heavily replicated.

The programmable pipeline is directed using a low-level abstraction layer such as OpenGL or Direct3D. The abstraction layer serves to hide the differences between varying implementations of the pipeline and provide a more convenient programming abstraction. Fixed platforms, such as consoles, differ from PCs in that there is only one hardware implementation, so often low-level details of the hardware are exposed through the abstraction layer.

We refer to the abstraction layer as a *runtime* and it is controlled through its API. The runtime provides device independent resource management (allocation, lifetime, initialization, virtualization, etc) for texture maps, vertex buffers, and other state and it communicates with the hardware accelerator through device-dependent driver software. The transition to a programmable pipeline has added the task of abstracting and managing shader programs to the runtime.

The limited instruction store of early programmable processors made the choice of programming in an assembly-like language [Gray 2003] both practical and in many cases necessary to maximize control of the limited resources. However, modest increases in available hardware resources created a need for a higher-level programming abstraction to maximize programmer productivity. C-like programming languages with some customizations to match the underlying rendering pipeline (4-vectors, intrinsics, I/O registers) answered this need [Proudfoot et al. 2001; Microsoft 2002; Mark 2003; Kessenich 2004; McCool and Du Toit 2004;] Additionally, other languages have been developed to explore the use of the substantial floating-point processing and memory bandwidth of GPUs for application domains other than rendering [Buck et al. 2004; McCormick et al. 2004], but we will not address this latter subject further in this paper.

While there are similarities to imperative CPU programming languages (notably C), there are some significant departures. For example, the machine and compilation model is more virtual machine-like, with the shader assembly language serving as a machine-independent intermediate language (IL) rather than a specific machine language[1]. Though a high-level language like Microsoft's HLSL can be compiled to IL offline, the translation to the target hardware occurs just in time (JIT) at run-time with the translator implemented as part of the driver infrastructure for the GPU. We note that the OpenGL Shading Language takes a different approach with the entire compilation process occuring at run-time.

Another significant difference is that shading programs are not standalone applications, and are instead executing in concert with a program executing on the CPU that orchestrates the rendering pipeline. The CPU program also supplies parameters to the shading program in the form of texture maps or by populating on-chip registers called constants.

While this paper does not describe a specific hardware embodiment of the new pipeline architecture, the pipeline design is shaped significantly by hardware practicalities and was designed concurrently with multiple hardware implementations. Many of the structural underpinnings from current hardware implementations [ATI 2005; Doggett 2005; Montrym and Moreton 2005] continue to be both relevant and influential in this design.

3. The Pipeline

The Direct3D 10 pipeline retains the structure of the traditional hardware-accelerated 3D pipeline. Two new stages have been added and other stages have been either simplified or further generalized. The basic pipeline is illustrated in Figure 1. For consistency we describe each of the pipeline stages, rather than just the additions. We use traditional terms such as vertex, texture, and pixel for continuity with prior nomenclature, but acknowledge that this terminology reflects a specific usage of a more general processing capability.

Input Assembler (IA) gathers 1D vertex data from up to 8 input streams attached to vertex buffers and converts data items to a canonical format (e.g., float32). Each stream specifies an independent vertex structure containing up to 16 fields (called elements). An element is a homogenous tuple of 1 to 4 data items (e.g., float32s). A vertex is assembled by reading from the currently enabled streams. Normally vertex data is read sequentially from each vertex buffer; however, if an index buffer is specified then each stream uses a shared index to compute the offset into each vertex buffer. Indexing allows additional performance optimizations in that the vertex processor computes a result that is completely determined by the index value, therefore recomputation of results for the same index can be avoided using a result cache indexed by the index value.

The IA also supports a mechanism that allows the IA to effectively replicate an object *n* times. This mechanism is an addressing mode referred to as *instancing* in which a repeat count *n* is associated with block of *k* vertices (corresponding to an object). At the same time, the primitive data is "tagged" with a current instance, primitive, and vertex id and these ids can be accessed in the programmable stages to compute values such as transformations or material parameters based on these ids.

[1] This does contradict the notion that the assembly-level shader programmer has absolute control.

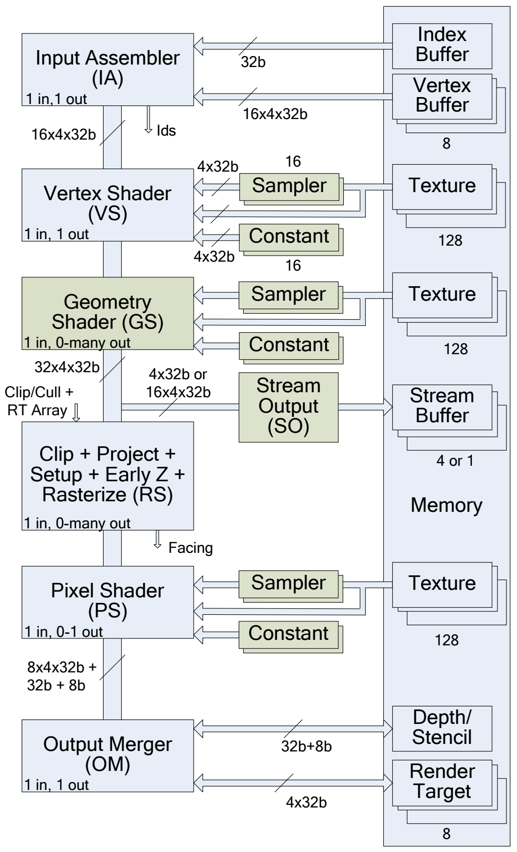

Figure 1: Direct3D 10 pipeline
Major additions are highlighted

Vertex Shader (VS) is most commonly used to transform vertices from object space to clip space. The VS reads a single vertex and produces a single vertex as output. The VS and other programmable stages share a common feature set that includes an expanded set of floating-point, integer, control, and memory read instructions allowing access to up to 128 memory buffers (textures) and 16 parameter (constant) buffers. This common core is described in more detail in Section 4.

Geometry Shader (GS) takes the vertices of a single primitive (point, line segment, or triangle) as input and generates the vertices of zero or more primitives. The input and output primitive types need not match, but they are fixed for the shader program. A GS program can amplify the number of input primitives by emitting additional primitives subject to a per-invocation limit of 1024 32-bit values of vertex data. Triangles and lines are output as connected strips of vertices. A GS program can output more than one strip in a single invocation or it can effectively delete an input primitive by not producing an output. A GS program can also simply affix additional attributes to a primitive without generating additional geometry, for example, computing additional uniform-valued attributes for each primitive. Since all of the primitive vertices are available, geometric attributes such as a triangle's plane equation can be readily computed.

In addition to the traditional input primitives, triangle and line primitives may also be processed with their adjacent vertices. A triangle comprises 3 vertices plus 3 adjacent vertices while a line has 2 vertices with 2 adjacent vertices as shown in Figure 2. Adjacent vertices are included as part of the vertex buffer formats for triangle and line primitives and are extracted by the IA when a primitive topology with adjacency is specified (rendered).

Stream Output (SO) copies a subset of the vertex information output by the GS to up to 4 1D output buffers in sequential order. Ideally the SO should have symmetric output capabilities with the (non-indexed) input capabilities of the IA (8 streams x 16 elements), but the hardware costs were not justified. The SO is limited to either 1 multi-element output stream of up to 16 elements or up to 4 single-element output streams. While the IA can support reading from 8- and 16-bit data types and converting to float32, the SO can only write raw 32-bit data types. However, data conversion and packing can be easily implemented in a GS program reducing the need for fixed-function support.

Set-up and Rasterization Stage (RS) is a fixed-function stage handling clipping, culling, perspective divide, viewport transform, primitive set-up, scissoring, depth offset, and fragment generation. Modern GPU designs invariably include some form of early depth processing (z-cull, hierarchical-z) [ATI 2005; Montrym and Moreton 2005] as well. We explicitly mention this optimization as it is becoming less transparent to application developers. The input of the RS is the vertices and attributes of a single primitive and the output is a series of pixel fragments.

The pixel shader program specifies the manner in which vertex attributes are interpolated to produce fragment attributes (no interpolation, non-perspective-corrected interpolation, or perspective-corrected interpolation). Modern GPUs usually support multisample antialiasing [Akeley 1993]. Multisampling requires additional care in specifying attribute evaluation behavior when a fragment does not include the pixel center, since center evaluation may result in an out-of-gamut value. An additional evaluation qualifier (centroid) can be specified to request evaluation within the fragment boundaries.

Pixel Shader (PS) reads the attributes of a single pixel fragment and produces a single output fragment consisting of 1 to 8 attribute (color) values and optionally a depth value. The attribute values (elements) are each written to a separate color buffer (termed a *render target*) or the entire result may be discarded (no fragment is output). Normally depth and stencil values are forwarded from the RS. However, the PS can replace the depth value with a computed value, but not the stencil value. Both discarding pixels and replacing the depth value may defeat depth-processing optimizations in the RS since they can change the fragment's visibility.

Output Merger (OM)[2] takes a fragment from the PS and performs traditional stencil and depth testing operations as well as render target blending. The OM specifies bind points for a single unified depth/stencil buffer and up to 8 other render targets (attribute buffers). The pixel shader must output a separate value for each render target (there is no multicast). While a single blending function is shared across all of the render targets, blending can be enabled or disabled independently for each render target.

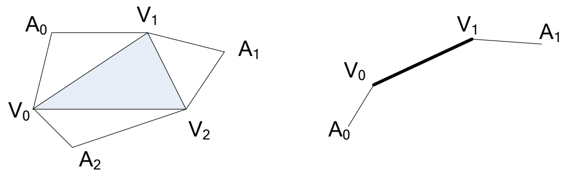

Figure 2: Triangle and line segment with adjacent vertices

3.1 Memory Structure and Data Flow

Modern GPUs rely heavily on processing retained data structures in the form of vertex and index buffers, texture maps, render targets and depth/stencil buffers. GPUs typically store these in a high-performance memory system attached directly to the GPU. The range of structures includes homogeneous 1D through 3D

[2] Some implementations traditionally refer to this functionality as "ROP" for raster operations.

images plus 2D cube map images (all with optional mipmap pyramids), and 1D homogenous and heterogeneous index and vertex buffers. Direct3D 10 generalizes these structure types, called *resources*, with the goal of improving efficiency. Efficiency improvements come about in two ways: one, by increasing the range of processing that can be performed in a single rendering pass and second, by allowing greater flexibility in generating data to a resource in one pass followed by using that data in a subsequent rendering pass.

The efficiency enhancements for single-pass rendering come from several sources. One is the addition of *arrayed* resources. Texture maps and render targets can be created as linear arrays (of up to 512 elements) of a homogenous resource and bound to the pipeline as a texture or a render target. Shader instructions used to address texture maps are extended to include a shader-computed array index. This alleviates some pressure for application developers to pack multiple images into a texture map and the extra computation to manipulate texture coordinates to retrieve the subimage. However, texture arrays do not necessarily help for the case of (un)packing inconsistently-sized images since the array elements must all have the same size.

When an array of render targets is bound to the OM, the target array index is computed for each primitive in the GS. This allows the GS to sort (or replicate) primitives into different array elements. One example of this is rendering an environment to a cube map in a single rendering pass by treating the cube map as an array of 6 2D render targets. As the environment geometry is processed, the GS determines to which cube faces a primitive should be rendered and issues the primitive once for each face. Note that the GS render target array selection mechanism is independent (orthogonal) to the multiple render target outputs of the PS.

To enable render-to-cube-map and to simplify the use of arrays, resources are extended with the notion of a *view* that either selects a subset of the resource (e.g., a single array element) or binds additional type information to a *partially-typed* resource. In the latter case, Direct3D 10 allows a resource to be created without binding the specific element data type (e.g., float16, snorm16, etc). This allows a very limited form of storage type "casting" where the data type can change but not the size of the data type (e.g., an element with 2 float16s cannot be treated as an element with a single float32). Resources cannot be directly bound to the pipeline; instead they are bound to the pipeline using a view. Two different views of a resource can be bound simultaneously to the pipeline.

Using the rendered cube map in a subsequent pass is an example of a multipass or iterative rendering scenario enabled by the more flexible Direct3D 10 resource model. A similarly useful capability is render-to-vertex-buffer. One approach is to attach a vertex buffer as a render target (using a view), compute new vertex data in one of the shader stages, and pass it through the remainder of the pipeline as color attribute data to the render target. Complications with this approach include limitations in the size and homogeneity of a vertex element (4xfloat32), the need to use multiple stages of the pipeline (e.g., VS and PS) for relatively simple processing, and the manner in which a 1D vertex buffer is mapped to a 2D render target. A Direct3D 10 application uses a view to select a contiguous subregion of the buffer as an *n*-wide x 1-high 2D render target, with a maximum subregion width of 8K elements.

The stream output capability provides an attractive alternative for performing sequential 1D output. An advantage of stream output is its support of richer output formats, for example, being able to write the equivalent of 16x4xfloat32 elements per vertex as well as a much larger buffer (128MB vs. 128KB). Stream output does not support random access (scatter) into the output stream, whereas the render target approach does since address can be controlled by drawing points and using the vertex shader to modify the render target coordinates of the point. For these reasons both approaches are useful.

To support iterative computation, the constraints on resources have been relaxed with respect to where resources can be attached to the pipeline resource bind points (i.e., IA buffers, VS/GS/PS textures, SO buffers, and OM render targets) using views as a form of "adapter" (e.g., rendering to a single mipmap level or a 2D slice of a 3D texture). However these adapters are far from universal: a 2D resource cannot be treated as a 1D resource and homogeneous resources comprising a single element type can't be interchanged with non-homogeneous resources such as multi-element vertex buffers. These restrictions largely prevent arbitrary reinterpretation of the gross structure of a resource, allowing hardware implementations to optimize storage layout. Similarly, the long-standing constraint that a resource cannot be simultaneously bound as both an input and an output resource remains.

3.2 Storage Formats

While data operated on within a shader is 32-bit (interpreted as either floating-point or integer), a richer set of storage data types is provided to reduce memory footprint and bandwidth. Data types that are not an integer multiple of 8 bits are packed with other types to produce a format that is an integer multiple. Almost all formats can be used in vertex buffers, textures, stream outputs (using manual conversion and packing), and render targets. The data types are summarized in Table 2.

name	**widths**	**range**
unormN, snormN	8, 16	[0,1] and [-1,1]
floatN	32, 16, 11, 10	s23e8, s10e5, 6e5, 5e5
uintN, sintN	8, 16, 32	$[0,2^n-1]$ and $[-2^{n-1},2^{n-1}-1]$
RGBE	32	9/9/9e5
SRGB	8	[0,1] non-linear

Table 2: Storage Data Types

The unorm, snorm, and float16 (half) formats are already widely used. While float16 is attractive for high-dynamic range imaging applications, it consumes too much storage space and memory bandwidth. Direct3D 10 provides two alternative 32-bit packed representations: two float11s (R, G) combined with a float10 (B) and a shared-exponent format (RGBE) with 9-bit mantissas for R, G, and B plus a 5-bit exponent. These formats are restricted to positive values and provide the same dynamic range as float16 (10 orders of magnitude). The lower precision, 11-11-10 format works well as a destination format for rendering HDR color data, whereas the shared exponent format is restricted to use as a source format for texturing operations. The shared-exponent format requires a more sophisticated encoder to avoid artifacts, hence the read-only restriction.

In addition to these simple forms of compression, Direct3D 10 augments the 4x4 texel block compression (S3TC) formats from previous versions of Direct3D [McCabe and Brothers 1998]. One through four component versions of the format are defined with compression ratios of 8:1, 6:1, and 4:1. The three and four (correlated) color component formats are suitable for low-dynamic-range color, whereas the two (uncorrelated) component format is suitable for tangent-space normal map data.

3.3 Design Considerations

There are several architecture decisions worth noting or that warrant additional discussion.

In contrast to prior generations, Direct3D 10 requires that essentially all features be supported by the hardware. The only two exceptions are the more costly filtering support for 32-bit floating-point textures and render target formats involving multisample antialiasing. This increases the burden on implementation providers but reflects the application developer preference to scale on the performance axis rather than the more difficult problem of scaling across features. Emphasizing hardware support for important storage formats increases the likelihood of their adoption across a wider range of applications.

All traditional fixed-function capability that is expressible in terms of programmable constructs has been eliminated from the pipeline and the core API. This includes vertex transform and lighting, point sprites, fog, and alpha testing. While the fixed-function capabilities can be easily emulated in software, we believe to reduce complexity this should be exposed through a separate software library rather than the core API.

Despite our changes to further generalize the pipeline, several large fixed-function blocks remain. Early on we considered making the IA fully programmable to allow more complex indexing schemes or vertex layouts, but ultimately we could not justify the extra complexity. Conversely, the memory read capability of the VS allows more complex indexing schemes to be implemented in the VS; in fact, one could eliminate all of the memory reads from the IA and just compute the id values. However, we retained the more complex IA for performance reasons with the observation that its position and function in the pipeline can enable hardware to do a better job of scheduling vertex-related memory traffic.

There are many complicated design issues around the GS. One of the more important pipeline tradeoffs is enabling parallelism while preserving order of operations. The GS is defined to preserve the order of its inputs, so multiple GS units executing in parallel cannot emit primitives out of order. This requirement translates into parallel implementations that must buffer their outputs and process the completed buffers in input order. Efficient buffer management motivated an upper bound on the output and the ability of the GS program to specify a smaller bound. The ceiling of 1024 32-bit values is a compromise between hardware cost and allowing useful amplification, for example, extruding the edges of a triangle. Undoubtedly there will be a temptation to employ the GS for larger scale amplification such as tessellation and our expectation is that performance will degrade quickly.

The GS subsumes as much of the RS functionality as is practical. While it would be desirable to have the GS perform the homogeneous divide, viewport transform, etc, the clipping operation that precedes these operations is impractical thus the GS-RS partition at clipping. The GS does perform some of the clipping set-up, e.g. computing vertex to model clip-plane distances and passing those to the RS in addition to the clip-space coordinates of the vertex. Since the precision and exact set of operations for the fixed-function RS vertex processing are not precisely defined, it is not possible for the GS to exactly mimic the transformations to produce image-space (window) coordinates, though it can come close. This limits some of the utility of the GS in implementing algorithms that rely on manipulating the image-space coordinates of a primitive.

Finally, the fixed-function limitations of the OM are a frequent source of discussion. The OM unit is the only stage where memory read-modify-write operations are supported and this is one of the features frequently requested for the programmable units. One proposal is to merge the OM functionality into the PS. However, the complexities in managing pipeline hazards and maximizing memory system efficiency do not yet lend themselves to a justifiable cost. Multisampling further complicates the structure since PS computations are performed on pixel fragments whereas blending operations are performed on samples. Promoting the PS to execute at sample granularity has significant performance ramifications. Furthermore, notable performance gains are achieved using early depth and stencil rejection optimizations. Success relies on the predictability of the outcome of shading operations, creating an argument against migrating that functionality to the PS.

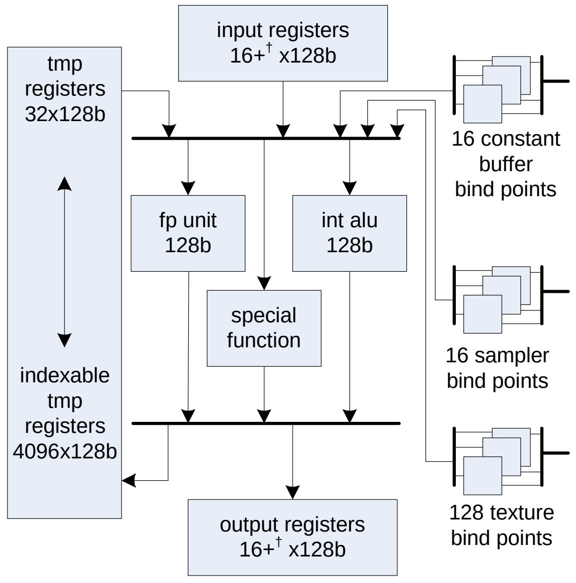
input registers 16+† x128b
tmp registers 32x128b
indexable tmp registers 4096x128b
fp unit 128b
int alu 128b
special function
output registers 16+† x128b
16 constant buffer bind points
16 sampler bind points
128 texture bind points

Figure 3: Programmable core
†input and output registers are further specialized in each stage

4. Shader Model 4.0

In previous versions of Direct3D, the programmable pipeline stages are described by a separate virtual machine for each programmable stage, i.e., vertex and fragment processors. Each virtual machine is described by a register-based Von Neumann-style processor with an assembly language-like instruction set, input and output registers for inter-stage communication, general purpose registers (sometimes called temporary registers), and a set of resource binding points for attaching memory resources such as texture images.

Direct3D 10 defines a single "common core" virtual machine as the base for each of the programmable stages, illustrated in Figure 3. This virtual machine retains many of the features of the previous models, for example, the basic 4-tuple register and floating-point arithmetic operations, while adding the following:

- 32-bit integer (arithmetic, bitwise, and conversion) instructions
- unified pool of general purpose and indexable registers (4096x4)
- separate unfiltered and filtered memory read instructions (load and sample instructions)
- decoupled texture bind points (128) and sampler state (16)
- shadow map sampling support
- multiple banks (16) of constant (parameter) buffers (4096x4)

Together, these constitute a major increase in capability. This unified model is considerably closer to providing all of the arithmetic, logic, and flow control constructs available on a CPU. Resources such as registers, texture bind points and instruction store, have been substantially increased to the point where they should not be a developer impediment for our target market for the next several years. Hardware implementations are expected to exhibit roughly linear degradation with increasing resource consumption as opposed to falling off performance cliffs.

With the increase in number of textures bindings, it became apparent that the number of unique texture filtering combinations did not require a commensurate increase. Instead the sampler state is decoupled into a separate object and the sampling instructions specify both a texture and a sampler to apply. The same texture can also be read, unfiltered, via a load instruction that operates using non-normalized addresses.

The demand for both an increase in constant storage and for more efficient update of constants presented a challenge. Current systems exhibited problems with both efficiently updating individual constant locations and with pipelining the update operations. The problem was made worse by switching between shader programs and needing to reload the constants associated with the new shader. One observation was that groups of constants are updated at different frequencies (e.g., once per frame, once per object, once per object instance). This led us to partition the constant store into separate buffers and to separate the operations of updating constants from binding them to the pipeline. Separating these operations allows implementations to do a better job of pipelining both types of operations. Since these operations are identical to how texture resource are handled we considered removing or unifying constants with textures.

However, the manner in which constants and textures are referenced are different in important ways. Constants are typically accessed at much higher frequencies than textures and often using indices that are uniform across sets of vertices or pixels, whereas textures are accessed at lower frequencies and with different indexes (texture coordinate values). This suggested that there were hardware implementation advantages to keeping constants and textures distinct.

A less visible, but equally significant change to the processing core is that the data representations, arithmetic accuracy, and behavior are much more rigorously specified than in the past. Much of this comes from recognition that shading programs, constants, and other pipeline state are actually part of the art content in an application rather than part of the execution engine. In this respect, there is increasing pressure to make this content more portable between implementations as well as preserving compatibility between generations of content. This is a reflection of the success of programmable shading.

Where possible we have avoided inventing custom behavior and follow CPU norms. We transitioned to the IEEE-754 [IEEE 1985] single-precision floating-point representation with a goal to converge to the exact behavior within a few years. In this generation, the basic arithmetic operations (add, subtract, multiply) are accurate to 1 ulp (rather than .5 ulp required in IEEE-754). Divide and square root are accurate to 2 ulp. Denormalized numbers are flushed to zero (but are defined and required for float16 operations) and IEEE-754 specials (NaNs, infinities) are fully implemented. These tolerances are driven by cost versus benefit considerations, where our first priority is to create well-defined, consistent behavior between hardware implementations and second to improve accuracy as hardware costs permit.

One of the more controversial design decisions has been the explicit adoption of IEEE-754 special behavior. This new behavior was introduced in the previous generation (shader model 3.0) and has caused some problems with portability of shading programs that relied on NaNs being flushed to zero. While we considered adding a unique mode to allow suppression of specials, we ultimately decided this would ease shader development in the short term at the expense of a greater long term maintenance cost in supporting this mode in all future hardware.

This rigor in specification is not solely limited to the programmable units; it extends to the definitions of the rules for filtering, rasterization, subpixel precision, data conversion, blending operations, etc. Our goal is twofold: to achieve both behavioral consistency and predictability for application developers. Pursuit of these objectives necessitated detailed discussions about NaN-propagation, optimizations of arithmetic or memory operations involving coefficients of 0 and 1, etc. Generally, we tried to arrive at compromises that allow for important performance optimizations.

4.1 Stage-Specific Functionality

Shader stages may have stage-specific specializations that augment the common behavior. These specializations include the configuration of the input and output attribute registers and additional instructions. The VS, with input and output configuration of 16x4 floating-point data elements, defines the common core.

The GS can input up to 6 times that number since it must read all of the vertices for a triangle and its 3 adjacent vertices. Since the GS can output more than one vertex or primitive, it cannot use the stream model of the other stages, where values are accumulated in output registers, with the actual output signaled by the end of the program. Instead there is an explicit *emit* instruction to signal output of an accumulated result to the next stage. There is also a *cut* instruction that signals the end of a strip primitive. The GS uses a compile-time configuration directive to specify the maximum output that any invocation will produce. Each output vertex can include up to 32 4xfloat32 elements for input by the RS and subsequent input into the PS. This is twice the number output by the vertex shader. These extra resources are used to communicate clipping and culling information to the RS as well as additional per-primitive data to the PS.

The PS has up to 32x4 input registers, but can only use them all if the GS is active. If the GS is not active, then the PS can only input the 16x4 values produced by the VS. These input values include compile-time directives to specify the evaluation (interpolation) method for each attribute. The PS outputs directly to up to 8 render targets, so it has 8x4 output registers plus a register for depth. The PS includes an instruction to discard a pixel with no effect on the render target and instructions for computing image-space derivatives ($\partial/\partial x$, $\partial/\partial y$). Only the PS stage has a built-in notion of screen-space, so while the VS and GS include instructions for sampling textures, they do not include the instructions that implicitly compute level-of-detail for mipmap filtering or the derivative instructions. Derivative behavior inside flow control is ill-defined (implementation specific) when the conditional expression varies between pixels. Therefore derivative instructions are disallowed within non-uniform flow control using compile-time enforcement.

A final set of specializations supports communication of values that are produced or consumed by fixed-function stages such as the IA and RS with the other programmable stages. For example, the IA produces a set of *system-generated values*: vertex, instance, and primitive ids and the RS produces a value indicating whether a polygon is front or back facing. Similarly, the RS accepts the *system interpreted values* output by a shader such as the position coordinates for a primitive, clip and cull distances, and render target array index. System-generated values are input into a programmable stage by declaring one of the inputs with the corresponding system-generated name, and conversely a system-interpreted value is driven by a programmable stage by declaring an output with the name of the system-interpreted value. These values count against the number of input and output registers used by a shader and in some cases they must be defined or an error results (e.g., the RS needs to distinguish position from other attributes). Sharing with the other registers keeps the input and output architecture more regular and avoids wasting resources that may go idle if additional dedicated input and output registers were used.

While a general purpose mechanism to manipulate other fixed-function state from the programmable stages (e.g., blend modes, depth or stencil configurations) is attractive it is currently only practical to do this for limited amounts of state.

5. Core API and Runtime

We separate the API and runtime into separate, but integral pieces: the core API/runtime and the shading language/state management system. We describe some of the larger parts of the new runtime and how they have changed relative to current systems.

The core API and runtime serves as the low-overhead, thin abstraction layer above the hardware. The transition to the programmable pipeline and removal of redundant fixed-function has dramatically simplified API and runtime. The API and runtime provide services for allocating and modifying resources, creating views and binding them to different parts of the pipeline, creating shaders and binding them to the pipeline, manipulating state for the non-programmable parts of the pipeline, initiating rendering operations, and querying information from the pipeline either by retrieving statistics or the contents of resources.

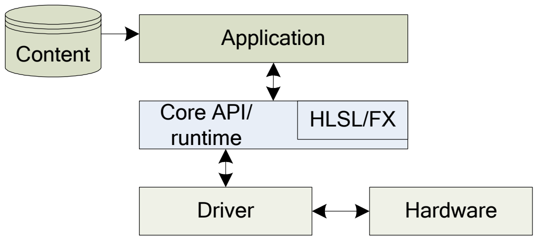

Figure 4: System layering

5.1 State Management

One of the big problems we set out to solve is reducing the end-to-end overhead of transferring commands from the application to the hardware. We partition commands into two classes: those that allocate or free resources and those that alter pipeline state. We are most concerned with the latter since they are the ones that necessarily occur most frequently in an application. We have a simple model for how those commands are delivered to the pipeline. The runtime allocates a memory buffer in which to append commands. Each API command calls through the runtime to the driver to add the hardware-specific translation of the command to this buffer. The buffer is transmitted to the hardware when it is full or when another operation requires the rendering state to be synchronized (e.g., reading the contents of a render target).

Little has changed in this runtime model on the PC over the past 10 years. Our goal is that commands be appended to the buffer with no need for extra processing. In the past this expectation has been unrealistic, so we endeavored to understand why and to look for design changes to bring our model closer to this ideal.

We found that there were several contributors to extra processing in both the runtime and drivers:

- mismatch between API and hardware
- deferred processing style
- miscommunication over application requirements

Of the three the third was the easiest to resolve, amounting to reaching agreement among application developers, runtime, driver, and hardware providers that the problem needed to be addressed in a significant way (i.e., not a 10% improvement but a tenfold improvement).

The second describes a traditional implementation strategy in which state changes are accumulated and resolved when primitives are issued to the pipeline. This has advantages that a set of state changes can be processed in bulk and state implementations that are interdependent (non-orthogonal) can be processed together rather than evaluating the interdependent state each time one of the dependencies change. It also allows redundant state changes to be trivially discarded. However, this comes at the cost of extra CPU cycles to record the changes and to do this global processing. An example of a catastrophic non-orthogonality is a change in a texture binding requiring a shader program recompile to match the new format. We advocate eliminating interdependencies in the hardware state implementations as much as possible and to relegate redundant state change suppression to a separate optional layer in the runtime (rather than redundantly attempting to solve it in multiple parts of the software stack).

The first category covers several kinds of mismatches. One example is orthogonality mismatches as exemplified by the shader recompile example, but there are many others. A related issue concerns granularity of a state change. Both OpenGL and prior versions of Direct3D define state changes at very fine granularity, e.g., change a blend factor, or a sampler filtering mode. There have been various attempts to aggregate state changes to make them more efficient, for example, using display lists in OpenGL or state blocks in Direct3D 9. While it is possible that those solutions could be made to work with a much more concerted effort, we chose a simpler approach. The elimination of redundant fixed pipeline functions has reduced the overall amount of state. Our analysis failed to show any significant advantage in retaining fine grain changes on the remaining state, so we collected the fine grain state into larger, related, immutable aggregates called *state objects*. This has the advantage of establishing an unambiguous model for which pieces of state should and should not be independent and reducing the number of API calls required to substantially reconfigure the pipeline. This model provides a better match for the way we have observed applications using the API.

Direct3D 10 defines 5 state objects: InputLayout (vertex buffer layout), Sampler, Rasterizer, DepthStencil, and Blend. The partitioning reflects state that logically belongs together (stage-wise) and is only further subdivided if an application is likely to change it independently at a significant frequency. At object creation time the driver can create the hardware representation for this state (e.g., a set of register values) and when the object is bound to the pipeline, the corresponding commands are copied into the command buffer. Some implementations may choose to retain (cache) the state representation in hardware and reduce the implementation of the API command to a hardware command to activate this retained state.

In Section 4 we described an issue involving pipelining constant updates. This is an instance of a more general problem involving pipeline hazards. Some types of hazards occur when a value is about to be updated, but the previous one is still being used. These are typically solved using extra storage to hold the new value and redirecting references to the new value to use the new buffer (renaming).

Another type of hazard occurs when switching a resource from being written to, to being read from. For example, switching a texture that was previously used as a render target, to a texture source requires that the outstanding rendering commands complete and be written to the render target before texels can be fetched from it. Unlike the update hazard described above, read-after-write hazards are more difficult or impossible to eliminate from the API and runtime. To avoid stalling the pipeline in such cases, the application should be structured to perform rendering work that does not immediately require reading from the previous render target whenever possible.

5.2 Validation and Error Handling

Some of the tenets of the API design are to avoid creating situations where errors may arise or to do error checking during more-expensive low-frequency operations such as object creation rather than object use. While our performance objectives do not permit having copious error checking throughout the API for a deployed application, we appreciate the value of such error checking during application development. The strategy we use for error detection and reporting involves partitioning errors into two categories, critical and non-critical. Critical errors are always checked and reported in all versions of the runtime. The non-critical errors are detected in a separate interception layer that can be transparently pushed onto the runtime. This validation layer is primarily used during application development, with the developer disabling it when the application is deployed. In addition to detecting errors, it can also look for and report other types of non-ideal API use. The validation layer includes additional controls to enable a developer to make it more selective over what it detects or reports.

The error partitioning approach does create ambiguity over which errors are always detected and what the behavior should be for errors that are not detected. Undefined error behavior may later appear as an unintended but relied upon (defacto) behavior. Or, if the runtime doesn't detect a particular error, the driver may need to anyway to avoid a catastrophic hardware error, negating any performance win. We have made efforts to identify such errors and move their detection to the runtime. Another criterion that we imposed is that no error detection should happen at render time, e.g., error detection shouldn't be deferred until a Draw command is issued. The list of critical errors includes mismatched depth buffer and render target sizes, simultaneous binding a resource for read and write operations, etc, whereas examples of less-critical errors include: mismatched shader type linkages (signatures) and mismatched data format declarations.

At this stage in the shader programming model evolution it is costly to trap run-time errors in shader programs. Instead we have specified well-defined behavior, for example, returning 0 when reading outside of an array, to achieve consistent behavior. Longer term we anticipate that hardware implementations will support exception mechanisms.

5.3 Resource Mapping and Access

One of the more complex issues with the API and pipeline design concerns shared access to resources between the CPU and the GPU. For example, both Direct3D and OpenGL allow vertex buffers to be mapped into the application's address space irrespective of whether the buffer is allocated from system memory or dedicated accelerator memory. However, the location can have a dramatic effect on performance. The memory bandwidth between accelerator and attached memory can be more than 50GB/s on a modern accelerator, whereas a PCI Express interconnect can provide no more than 2.8GB/s of bandwidth to the GPU from system memory.

There are other degrees of freedom as well. For example, CPU access may be cached or uncached having a dramatic effect on read performance, or write-combined or not, having a similarly dramatic effect on CPU write performance. Additionally, a memory resource may be reordered from row-order to another (e.g., Morton, boustrophedon, or pi orders), or tiled [Blinn 1990; Hakura and Gupta 1997; Igehy et al. 1999] to improve spatial coherence when accessed from the GPU. We believe it essential to keep the application unaware of the tiling patterns, therefore when a resource is mapped to the CPU it must appear in a linear organization. Since tiling patterns are also dependent on the access pattern, they are different for 2D vs. a 3D texture and this issue is not purely one of CPU vs. GPU access. There are compelling performance reasons to try to expose all of these capabilities, but the tradeoff complexity can make it exceedingly difficult for an application to achieve the performance improvement. We have tried to provide as much of the functionality as possible within a straightforward model.

We classify the readers and writers of a particular resource, e.g., CPU vs. GPU and read vs. write. If the resource is primarily used by one client, then the problem is simpler since placement can be biased to the principal consumer. Similarly, knowing whether the resource is used exclusively as a read source or write destination also helps. Fortunately, this describes a large number of typical application usages. For example, render targets and textures are primarily accessed by the GPU and limited to write destinations and read sources. On the other hand, vertex buffers can have more complicated usage. While static geometry is read primarily by the GPU, dynamic geometry is often generated on the CPU as part of animation or synthesis and subsequently processed on the GPU. This amounts to frequent CPU writes and frequent GPU reads.

Direct3D 10 partitions resources into 4 classes according to usage: default, immutable, dynamic, and staging. Default corresponds to simple texture, render target, or static vertex buffer GPU-only access. Resources of type default are initialized by copying the contents from another resource. Immutable disallows the copy operation but provides a way to initialize the resource at creation time. Default and immutable resources cannot be mapped for CPU access. Dynamic resources can be bound to the pipeline while allowing write-only CPU mappings. This handles CPU generated vertex data, video decoders, etc. Finally, staging resources allow CPU-only mapping but can be copied to and from other resources. Staging resources are useful for initializing or retrieving data from GPU-only resources.

Resource placement and encoding is further qualified at creation time as to where on the pipeline the resource can be bound. These categorizations include vertex buffer, index buffer, constant buffer, shader resource (texture), stream output buffer, render target, or depth/stencil buffer. This typing serves two purposes: it provides additional placement hints to the driver and it simplifies error checking when the resources are used.

5.4 HLSL 10

The broad and rapid adoption of high-level shading language has left no doubt about the importance of such languages. In addition to supporting the new pipeline features, we have several additional objectives for our high-level shading language – HLSL. Simply put, we want an application developer to program efficiently in HLSL without needing to know arcane details of the underlying virtual machine, for example, register names or constant buffer indexes. We refined this statement to these sub-goals:

1. Eliminate the need and desire for applications to explicitly control resource allocation and assignments.
2. Transition from bind-by-name to bind-by-position as the principal binding mechanism.
3. Eliminate the need to author in the intermediate (assembly) language.

The first goal has several manifestations. In current systems application developers exercised control over placement of parameters in the constant store. This facilitated sharing of variables between multiple shader programs, by having the developer perform global allocation and placement amongst multiple shaders. With the addition of multiple constant buffers per pipeline stage, we believe that the compiler has sufficient information to perform placement within a buffer, but the application developer should control assignment of parameters to constant buffers. We extended the language to allow specification of the buffer name as part of the parameter declaration.

The second goal is a larger departure from previous directions and is related to our concerns around performance and future evolution. Bind-by-name can occur in several places: matching input and outputs between shaders, matching vertex buffer layout with the vertex shader, etc. While the run-time cost for doing matching of names between source and destination can be made moderately efficient and an implementation can cache source-destination pairs, we believe this is unwarranted complexity and encourages additional overhead in the runtime system. The changes are manifested in several places in the new system. Shader inputs and outputs have associated *signatures* that are similar to function prototypes in C. A pipeline configuration is legal only if the output signature of the preceding stage is *compatible* with the input of the next. Compatibility means there is an element-by-element match between the inputs and outputs. The one concession we make is that the following stage need not consume the trailing outputs from the preceding stage.

Bind-by-position is also reflected in the vertex buffer bindings to the IA and SO stages. However, in these stages we create separate objects to encapsulate the bindings, allowing the more expensive matching operation to be performed once at create time.

The third objective will also be considered controversial, since it strongly manifests itself in our implementation by not supporting input of shader programs authored in intermediate language. We have reached the point where shader programs are sufficiently complex that equivalents hand-authored in IL are consistently less efficient than those produced by the compiler. Furthermore we cannot justify the extra work in supporting and maintaining compatibility with a hand-authored IL path as we increase the sophistication of optimization, linkage, and interaction with driver compilers. The system does support generation of IL output from the compiler as a diagnostic technique, but we do not allow an application developer to modify the compiler output and inject that into the runtime.

This desire to achieve maximum performance out of compiled code raises a number of questions. Foremost is the scope of optimizations that a driver may be permitted to do while mapping the IL to the machine-specific language. As shaders increase in complexity it becomes vital to ensure that application developers have sufficient control over optimizations that change the order of executed operations. Invariance must be achievable for critical code fragments, so that multipass algorithms produce the same intermediate values for duplicate computations in separate shaders. We considered several solutions for how to specify invariance requirements in the source code itself, for example, requiring that subroutines be compiled in an invariant fashion even if they are inlined. However, our search ultimately led us to the more traditional route of providing selectable, well-defined optimization levels that must also be respected by the driver compiler.

We should also note that our preferred usage model is to perform author-time compilation of the HLSL code and only perform driver compilation of the opaque IL code at application runtime. This stems from a desire to avoid increasing delays in application response by adding additional compilation time. While this behavior is preferred whenever possible we also acknowledge that content authoring and other applications need to do run-time compilation and this is fully supported by the runtime.

5.5 HLSL-FX 10

We have already noted that the success of the programmable pipeline has led to a shift in treating the shader program and the remainder of the pipeline configuration from being part of the execution engine to part of the (portable) art assets for an application. To accommodate this practice, the Effects (FX) system extends the HLSL language to allow initialization of other fixed-function parts of the pipeline. A similar approach is described for CgFX and Cg [Mark 2003]. Though the approaches share common roots, we have diverged in the evolution of HLSL-FX. Our objective with FX is to meet the needs of runtime asset representation first and as an asset interchange or general content authoring tool second. In many ways these two are historically at odds since authoring tools are often willing to trade increased flexibility for reduced performance, whereas we have made run-time performance one of our most important goals.

The FX system has evolved substantially both as a result of our experiences with how application developers have used it and in response to our efforts to improve run-time performance. To this latter end the FX, HLSL, API, runtime, and the pipeline are all examined together to arrive at complementary solutions.

Similar changes were made to decouple expensive name lookup and matching operations from frequently issued state manipulation functions.

A recurring theme is the manner in which state is processed. One way to structure an application is to render a set of geometric objects, where each object has a pipeline configuration (an *Effect*) applied to it. An Effect is parameterized by setting shader parameters in constant buffers, texture bindings, and by setting other fixed-function state. For maximum efficiency, an application should draw all objects that use the same Effect together. This is the traditional state-sorting solution to efficiency used by scene management systems [Rohlf and Helman 1994]. However, within an Effect there may be several levels of parameterization, for example, per-frame state such as current time, view position; character static state such as texture maps or vertex data; character dynamic state such as position, pose; etc. A separate constant buffer is used to store the shader parameterization at each level and as the objects are drawn constant buffers holding static parameters are simply bound, and constant buffers holding dynamic parameters are updated and bound.

As a practical matter, applications cannot always sort objects by Effect. There may be other constraints that govern the order of drawing objects, such as distance from the viewer, opacity, etc. We have designed the Direct3D 10 system to substantially reduce the cost of state changes in general so that reconfiguring the entire pipeline can be done efficiently.

6. System Experience

This project started in early 2003 and the runtime system and first hardware implementations are slated to ship sometime in 2006. We shared our first software prototypes in late 2004 and the first publicly available reference software implementation in late 2005.

Performance has received considerable attention throughout the project. Early on we made many measurements of current systems using microbenchmarks. We performed similar measurements on the new runtime using a simulated driver (faithfully implementing all of the driver interfaces) and the reference pipeline implementation. An abbreviated set of results are summarized in Table 3. It is difficult to make a perfect comparison as the Direct3D 10 commands operate at larger granularity. Our measurements look promising, but we will not know the final answer until we have hardware and drivers later in 2006.

Operation	**Direct3D 9**	**Direct3D 10 (reference)**
Draw	1470	154
Bind VS Shader	6636	416
Set Constant	3297	916
Set Blend Function	787	530

Table 3: Command cycle counts on Pentium IV

During the course of development we have written many applications to validate ideas for using the geometry shader, stream out, render target arrays and other new features. These samples include soft shadow volume algorithms, procedural content generation, higher-order normal interpolation, motion blur, morphing with sparse targets, shell and fin extrusions, etc (see Appendix A). These prototypes were critical to uncovering early design problems and now serve as validation for hardware implementations and educational examples for application developers.

With the reference implementation available we have been working with outside application developers to port several large Direct3D 9 applications to the new API to help track down "rough edges" in the API design. Concurrently we are working with several outside developers to create "from the ground up" applications using the new API and we anticipate that these will be commercially available later this year.

7. Future Work

We are currently analyzing development experience with our design and using that to inform a minor update to the API and runtime a year after the release of Direct3D 10. Since the application landscape is constantly changing, we encountered new problems with simple solutions, but too late for inclusion in the hardware. These features are also under consideration for the minor update.

Longer term we are looking at somewhat larger additions to the pipeline and programming model. Much of our attention is directed to the growing bottleneck in content production. One solution we are investigating is hardware-assisted procedural and quasi-procedural techniques for synthesis of detail. Examples include traditional procedural texture generation or tessellation of curved surfaces followed by displacement mapping. We believe these additions are possible with minimal disruption to the architecture. With respect to programming model, shader program composition and specialization are increasingly becoming a development and management burden. We are examining variations on traditional programming language constructs such as subroutines with dynamic binding. We believe this can be supported through minor hardware additions to the common shader core with the bulk of the work in the software system.

8. Conclusions

Direct3D 10 constitutes a large step forward from previous generations of the rendering pipeline. We chose to depart from the annual revision cycle of previous versions, taking more than 3 years to go from concept to hardware to leverage a larger net improvement in raw hardware capability. We were also more willing to make broader changes, breaking compatibility with previous versions of the pipeline and API to both solve the problems we were presented with and to provide a solid foundation for future evolution.

We anticipate the geometry shader and stream output becoming a rich source of new ideas. We are confident that the changes we made to extend and homogenize the programmable shading core are good long term steps for both hardware implementations and application developers. Ultimately we are satisfied that we arrived at good solutions for the problems we set out to address. We hope that this paper provides insights into the system design and that it will provide helpful background for those thinking about the next set of challenges and solutions for similar systems.

9. Acknowledgements

We would like to thank the companies who contributed to the Direct3D 10 design: ATI, NVIDIA, Intel, S3, and the members of the DirectX Graphics Advisory Board. Bob Drebin and Tom Frisinger at ATI and Henry Moreton at NVIDIA provided critical insights into hardware implementation. We also thank the design engineers in Microsoft's DirectX group: Amar Patel, Brian Klamik, John Ketchpaw, Jeff Noyle, Iouri Tarassov, Relja Markovic, Dan Baker, Craig Peeper, Peter-Pike Sloan, Mike Toelle, Sam Glassenberg, and Chas Boyd. Mike Toelle, Sam Glassenberg, Henry Moreton, and the anonymous reviewers made many constructive suggestions to improve the clarity of the paper.

References

ATI. 2005. Radeon X800 3D Architecture White Paper. `http://www.ati.com/products/radeonx800/RadeonX800ArchitectureWhitePaper.pdf`.

AKELEY, K. 1993. RealityEngine graphics. In *Proceedings of ACM SIGGRAPH 1993*. ACM Press, New York, NY, 109-116.

BLINN, J. F. 1990. The truth about texture mapping. *IEEE Computer Graphics and Applications 10*, 2, 78-83.

BUCK, I. FOLEY, T., HORN, D., SUGERMAN, D., FATAHALIAN, K., HOUSTIN, M., AND HANRAHAN, P. 2004. Brook for GPUs: Stream computing on graphics hardware. *Transactions on Graphics 23*, 3, 777-786.

CHAN, E., NG, R., SEN, P., PROUDFOOT, K., AND HANRAHAN, P. 2002. Efficient Partitioning of Fragment Shaders for Multipass Rendering on Programmable Graphics Hardware, In *Graphics Hardware*, 69-78.

DOGGETT, M., 2005. Xenos: XBox 360 GPU. GDC-E 2005, `http://www.ati.com/developer/eg05-xenos-doggett-final.pdf`.

GRAY, K. 2003. *The Microsoft DirectX 9 Programmable Graphics Pipeline*. Microsoft Press.

HAINES, E. 2006. An Introductory Tour of Rendering. *IEEE Computer Graphics and Applications 26*, 1, 76-87.

HAKURA, Z. S., AND GUPTA, A. 1997. The design and analysis of a cache architecture for texture mapping. *ACM SIGARCH Computer Architecture News 25*, 2, 108-120.

IEEE COMPUTER SOCIETY .1985. IEEE Standard for Binary Floating-Point Arithmetic. IEEE Std 754-1985.

IGEHY, H., ELDRIDGE, M., AND HANRAHAN, P. 1999. Parallel Texture Caching. In *Graphics Hardware*, ACM Press, New York, NY, 95-106.

KESSENICH, J., BALDWIN, D., and ROST, R. 2004. The OpenGL Shading Language version 1.10.59. `http://www.opengl.org/documentation/oglsl.html`.

LINDHOLM, E., KILGARD, M. J., AND MORETON, H. 2001. A User-programmable vertex engine. In *Proc. of SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, 149-158.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. Cg: A system for programming graphics in a C-like language. *Transactions on Graphics 22*, 3, 2003, 896-907.

MCCABE, D., AND BROTHERS, J. 1998. DirectX 6 Texture Map Compression. *Game Developer Magazine 5*, 8. 42-46.

MCCOOL, M. and DU TOIT, S. 2004. *Metaprogramming GPUs with Sh*. A K Peters.

MCCORMICK P. S., INMAN, J., AHRENS, J. P., HANSEN, C., AND ROTH, G. 2004, Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In *Proc. of IEEE Visualization*, 171-178.

MICROSOFT CORP. 2002. High-level shader language. In *DirectX 9.0 graphics*. `http://msdn.microsoft.com/directx`.

MICROSOFT CORP, 2006, Direct3D 10 Reference. In *Direct3D 10 graphics*. `http://msdn.microsoft.com/directx`.

MONTRYM, J., and MORETON, H. 2005. The GeForce 6800. *IEEE Micro 25*, 2, 41-51.

PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proc. of SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, 159-170.

RIFFEL, A., LEFOHN, A. E., VIDIMCE, K., LEONE, M., AND OWENS, J. D. 2004. Mio: Fast Multipass Partitioning via Priority-Based Instruction Scheduling. In *Graphics Hardware*, 35-44.

ROHLF, J. AND HELMAN, J. 1994. IRIS Performer: a high performance multiprocessing toolkit for real-time 3D graphics. In *Proc.* of SIGGRAPH '94. ACM Press, New York, NY, 381-394.

SEGAL, M., and AKELEY, K. 2004. The OpenGL Graphics System: A Specification (Version 2.0). `http://www.opengl.org/documentation/spec.html`.

TARDITI, D., PURI, S., AND OGLESBY, J. 2005. Accelerator : simplified programming of graphics units for general-purpose uses via data parallelism. Technical Report, MSR-TR-2005-184.

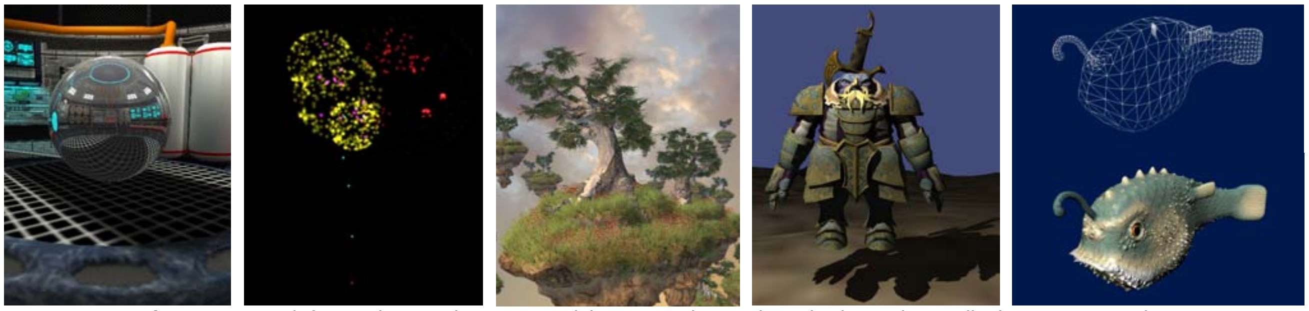
Figure 5: From left - render to cube map, particle system, instancing, shadow volume, displacement mapping

Appendix A: Examples

Figure 5 shows images from several Direct3D 10 sample applications. Each of these applications uses pipeline features to substantially reduce the amount of preprocessing or CPU operations compared to current rendering systems.

Example 1 renders a scene to a cube map and then applies the result as a reflection map in a second rendering pass. The cube map is generated in a single pass using a GS program to replicate each input triangle 6 times, applying a separate model-view transformation for each face of the cube. Each of the resulting triangles is directed to the appropriate element of a render target array view of the cube map.

Example 2 demonstrates a particle system using the GS and stream output. A pair of vertex buffers holds the current and next particle state, consisting of position, time, and type (emitters, non-emitters). In each pass the array of particles are processed drawing them as a list of point primitives. The GS evaluates the current state of each particle and writes zero or more updated particles to the stream output depending on this state. The number of output particles is tracked as part of the vertex buffer state and is used as the primitive count in subsequent drawing commands. The new list of particles is rendered in a second pass using the GS to convert each particle to a "point sprite" consisting of a pair of triangles.

Example 3 uses the instancing and geometry amplification features to draw a scene composed of 50 islands using a total of 6 drawing commands to draw the sky (1), lower island bases (50), upper island bases (50), trees (50), leaves (125000), and grass blades (500000). Instance and primitive ids generated by the IA are used to index arrays of modeling transforms and texture images to apply distinct transformations and shading to each of the instances. A GS program is used to dynamically generate the 10,000 blades of grass per island from a smaller number of seed primitives.

Example 4 demonstrates lighting with shadowing using a shadow volume technique [Everitt and Kilgard 2002]. A shadow volume consisting of front and back caps and side faces is created from the caster geometry in a single rendering pass using a GS program. The front cap is created using the surface normal and light vector to select those triangles that face the light source and draw them. The back cap is created by duplicating the front cap triangles and translating them along the light direction. Side faces are created by using adjacency information for each triangle to determine which triangle edges participate in the silhouette and extruding a quadrilateral (triangle pair) along each of the silhouette edges connecting the front and back caps.

Example 5 demonstrates a per-pixel displacement mapping technique [Hirche et al. 2004] used to add additional geometric detail to a base mesh of 1480 triangles. A GS program is used to extrude a prism at each base triangle along the surface normal. The prisms are each decomposed to 3 tetrahedra resulting in an additional 17760 triangles. Ray casting against a height field is performed in a PS program to test whether pixels generated by the extrusions are part of the displaced surface. Pixels are either discarded or shaded depending on the outcome of the test.

EVERITT, C. AND KILGARD, M. 2002. Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering. `http://developer.nvidia.com`.

HIRCHE, J., EHLERT, A. GUTHE, S. AND DOGGETT, M. 2004.Hardware accelerated per-pixel displacement mapping. In *Proc. of Graphics Interface 2004*, 153-160.