DirectX 10 Architecture

for
Chrome 400/500 Series
Discrete Graphics Processors

A
S3 Graphics
White Paper

© 2008 S3 Graphics. All rights reserved.
www.s3graphics.com

WP016-B.0
11/11/2008

Revision History

B.0	11/11/2008	Added Chrome 500 Series GPU Support	BT/KG
A.0	7/21/2007	Initial Version	BT/KG

CHROME

Introduction

This White Paper provides an overview of Microsoft's DirectX 10 architecture. DirectX 10 compatible features provide an important component of the enhanced experience available to users of systems featuring S3 Graphics Chrome 400/500 Series graphics processors. These processors are designed especially for Microsoft DirectX 10 and Windows Vista.

DirectX 3D is the standard API (application programming interface) that allows graphics hardware to render and support graphics on Microsoft Windows platforms. This API is a common interface or middleware that provides a hardware abstraction layer which allows developers of an application, such as a 3D game or CAD program, to access the graphics hardware via programming calls to the operating system (OS). When the application makes a request to draw an image on the screen, the API calls the OS, which in turn will invoke the graphics processor (GPU) driver to communicate with the graphics hardware to draw the corresponding image and output the result to the display. By using the standard DirectX interface, application developers need only need be concerned about their specific application, without needing to be concerned about details of the underlying hardware implementation. This allows developers to quickly create many visually stunning images and realistic detail, by providing fast access to the advanced hardware capabilities of today's leading edge GPUs.

Previous generations of Microsoft's DirectX (DX) 3D, had significant changes. Fixed function hardware units were used in DirectX generations up to DX7. Then programmable hardware shader units with new user-defined programming capabilities appeared for DX8. DX9 featured added hardware functionality and programmability. The latest release from Microsoft is DX10, which introduces a new architecture that is the subject of this white paper. S3 Graphics continues to work closely with Microsoft's DirectX team to extend its leadership in graphics technology with high performance parts, including the Chrome S20 Series processors based on DX9.0c, and the Chrome 400/500 Series graphics processors with advanced DX10 support.

Microsoft's latest DirectX release, DX10, extends the API beyond the limitations of previous generations. The DX10 API is the first redesign to the underlying architecture of DirectX 3D. New are optimized run-time features, CPU off-loading during state changes, a hardware geometry shader, texture arrays, and other graphics rendering enhancements that allow cinematic-like image quality. S3 Graphics Chrome 400/500 Series product lines fully support these new capabilities of DX10.

DirectX 9 Hardware Pipeline

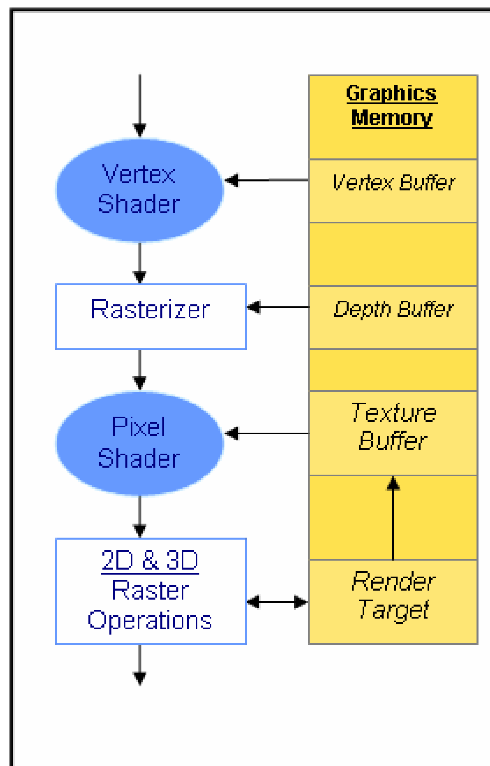

Figure 1. DirectX 8/9 Graphics Pipeline

The DirectX 8/9 pipeline is diagrammed in the above figure, which shows the basic features of a DX8/9 GPU. With the introduction of DX8/9 GPUs, such as S3 Graphics' Chrome 20 Series, the transition from a fixed to a programmable pipeline changed the graphics landscape. Fixed function pipelines meant the hardware blocks were hard-coded with specific graphics algorithms. Application developers had to limit their development to what the hardware supported. DirectX 8 introduced and DirectX 9 expanded programmable pipelines which provided an additional programmable API layer closer to the graphics hardware. Developers can take advantage of this layer by using shader assembly language to creatively write specific code to control the different shaders and elements of the programmable pipeline. The main parts of the pipeline are as follows:

- ***Vertex shaders*** **(VS)** replace the "Transform and lighting engine" logic prevalent in previous generations of graphics hardware. The VS can only manipulate vertices and transform the shapes of objects from the 3D model space to be displayed on a 2D screen. The VS also does per-vertex lighting based on computed color to give the vertex more detail. The VS cannot create or destroy any vertices and the unit can only work on one vertex at a time (in the API level). Actual graphics hardware may process batches or packets of vertices in parallel to increase throughput.
- ***Rasterization*** is the process of mapping a triangle from object to image space (combining vertices from the VS output) and determining which screen pixels cover the triangle. All pixels inside the triangle are tested for visibility using the depth buffer and are kept if the triangle being rasterized is closer from a viewer perspective than other triangles. All invisible triangles and pixels are discarded since they will not be seen onscreen. This step prepares the object to be modified at the pixel level by the pixel shader.
- ***Pixel shaders*** **(PS)** calculate color and texture on each individual pixel. They give flexibility to developers by allowing high quality details to be shown on each object.
- The 2D and 3D ***Raster Operation Pipeline*** **(ROP)** is responsible for outputting the rendered object to the render target buffer from the pipeline after textures and blending have been applied.
- The graphics memory stores vertex and texture data in addition to the final object and frame that is to be drawn onscreen.

Limitations of DirectX 8/9

Microsoft designed DirectX 10 to address some of the following key disadvantages identified in the DirectX 8/9 architectures. These include the following.

1. API overhead is high for DX8/9

- When a DX9 application requires use of the graphics hardware to draw an object onscreen, the application needs to perform a call to the DX8/9 API to tell the OS what to do. The OS would then call the graphics driver, which would instruct the graphics hardware to perform the assigned task for the application.
- The DX8/9 API runtime provides resource management like allocation, virtualization, and initialization for the graphics hardware for vertex buffers, texture maps, and state changes. With the introduction of programmable shaders, runtime allocation tends to be harder to manage since there are more levels of abstraction, control and detail per scene.
- As Figure 2 illustrates, all DX8/9 functional or runtime calls from an application to the graphics hardware were done by the CPU (once per object), causing CPU bottlenecks whenever many objects need to be rendered for the current frame. The high API overhead also limits the number of objects per scene, causing potential loss of detail in each frame.
- State changes within the GPU for the shaders and textures generate additional overhead as the CPU had to decode state change instructions in order to implement visual details for realistic rendering to object surfaces and textures. If multiple visual effects need to be performed on an object, multiple passes through the hardware could be required. That translates into multiple state changes per pass performed by the CPU.
- The CPU overhead required to direct the GPU, in essence, became the bottleneck between the DX8/9 application and DX8/9 hardware, which limits the overall capacity for creating stunning visuals.

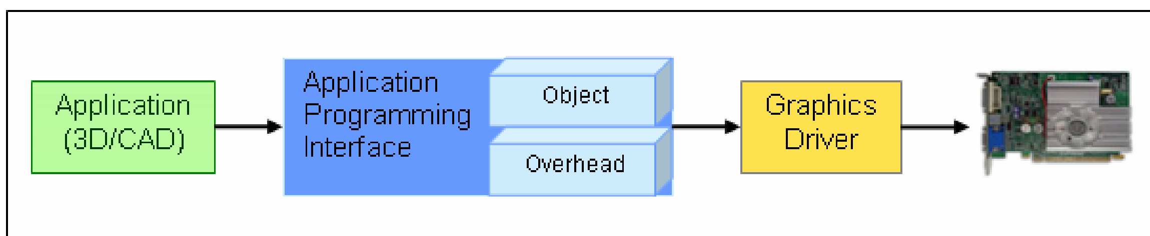

Figure 2. DirectX 8/9 Graphics API Interface

2. Vendor Variations in DirectX features

- Differences in DirectX feature support across GPU vendors, and even within a vendor's product lines, frequently cause problems for application development, because of the numerous levels of support that must be implemented across these multiple hardware platforms. Typical issues that must be addressed include allowing for the resource limitations of different hardware. The lack of support for optional features may prevent an application from running at its highest level on a particular platform. Differences in arithmetic precision in the hardware shaders, instruction and data types, and variations in storage of intermediate data formats may all affect the rendering process.

3. Hardware resource limitations

- The number of separate VS and PS units in the GPU are fixed. Applications requiring heavy use of one type of shader will cause the other shader to be idle. For example, large triangles create heavy loading for the PS, idling the VS. Since there is the possibility of an application being shader-limited, throughput of the rendering pipeline will be limited by the number and type of shaders. The DirectX 10 API allows graphics hardware to overcome this resource bottleneck by introducing a novel architecture which has been incorporated into all S3 Graphics Chrome 400/500 Series graphics processors.
- Another drawback of DX8/9 generation GPU architectures is their specialized focus on one task, that of 3D graphics rendering. A DirectX 8/9 GPU was designed to optimize that task, which in effect limited its ability to perform additional computing tasks. With the introduction of GPU architecture designed for DirectX 10, the GPU can now take on additional capabilities and complexity. The processor now becomes more of a general compute processor, capable of offloading the CPU from some basic tasks, and thus takes a significant step towards becoming a general purpose GPU (GPGPU).

DirectX 10 New Capabilities and Benefits

The main objective for redesigning the entire DirectX 10 API and hardware architecture was to provide a solution for the CPU overhead problem and the hardware capability issues. This re-design also incorporated benefits from the application development, API, and hardware perspectives. The improvements based on this extensive research and re-design are as follows:

1. Efficient runtime (lower API overhead)

- In DirectX 10 the number of possible states that need to be tracked by the system has been reduced, minimizing the overhead related to state changes.
- Overhead per object has been reduced which allows more objects per frame. This produces better graphic realism and a higher level of detail than was possible with previous generations of the API.
- The validation of objects has been redefined and the process is now more efficient. Validation checks the format of commands and the integrity of data sent by the application to make sure there are no interoperability issues with the hardware. The drawback of validation is a large CPU overhead at runtime. DX10 uses this feature minimally by only validating each object once when it is created, rather than every time the object is used, as was the case in DX9.

2. Reduced CPU loading

Rendering an object or applying multiple textures to an object in a repeated manner uses up valuable CPU cycles and overhead. DX10 has introduced several new instructions and hardware capabilities to help overcome rendering limitations.

- A new 3D pipeline unit called the geometry shader (GS) has been introduced with this iteration of DirectX. The GS can modify, create, or destroy primitive vertex data from the VS without CPU intervention, so no resource-intensive state changes or associated overhead is required by the API. In the past, any changes to the vertex data needed CPU-GPU coordination and state changes.
- The new hardware model in the DX10 pipeline gives more capability to the GPU to handle state changes and instructions. The DX10 GPU now includes built-in arithmetic and flow control logic, thereby providing flexibility in primitive shading and state change handling, and offloading the tasks once performed by the CPU.
- ***Stream out*** is a new feature that allows the VS/GS to output data directly into graphics memory where the data can be accessed automatically and repeatedly by the shader units. This is a great new feature controlled entirely by the GPU (with no CPU overhead), for recursive rendering on objects that require multiple passes through the pipeline. In addition, data from any step in the pipeline can go directly to memory. By avoiding the need to send data completely through the pipeline, resources are not wasted on processing intermediate vertices or pixels.
- ***Arrayed resources*** allow texture maps to be created as a linear array of up to 512 elements. Developers now have index instructions to access elements within the array in a single pass, so the GPU can work on multiple elements without any static switching overhead. For example, an environmental cube map can be stored in an

array as six elements (one for each face of the cube), and the GPU can work on all six elements concurrently in one pass.

- ***Multiple render targets*** allow the GPU to create different versions of a scene in a single pass. DX10 has the ability to create up to eight render targets at a time.
- ***High dynamic-range rendering*** is another feature that brings realistic graphics rendering to the user experience. Formats used in the past to represent color in floating-point representation took at least twice the amount of storage compared to integer formats with half the precision. DX10 provides more efficient mechanism for storage of color components by providing floating point format RGB 11:11:10 (R/G 11-bits each, B 10-bits) and RGBE format (5-bit shared exponent for R/G/B with 9-bit mantissas for each). These formats allow a wider range of color and more vivid detail to be represented as seen in the examples below.

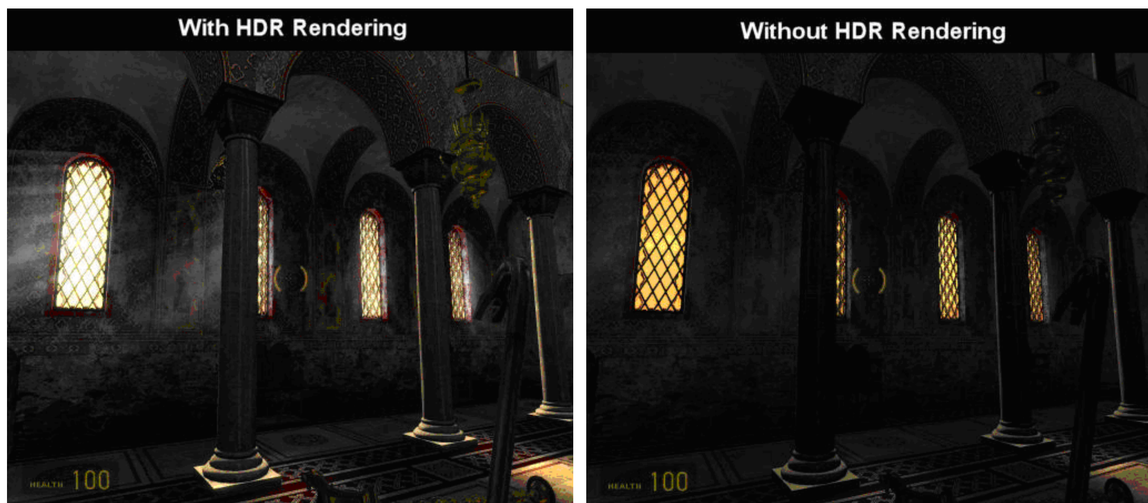

Figure 3. High Dynamic Range Rendering in Half-Life 2: Lost Coast

- A more complex method of utilizing occlusion query has also been implemented to conserve valuable GPU and CPU resources. *Occlusion query* is a method where non-visible primitives and objects in the Z-buffer are not rendered to save hardware computational resources. Because frame-to-frame rendering is very dynamic and implementation may vary with the application, occlusion query is not guaranteed to omit unseen pixels. DX10 takes it one step further by allowing the GPU to render complex objects in simple line drawing approximations. If the object needs to be drawn onscreen, the GPU already has the framework ready. If the object is not needed or is invisible onscreen, the GPU can throw away the object approximation, without wasting many CPU and GPU cycles.
- ***Data and resource mapping*** enhancements improve the ability of the GPU to access data in a timely manner. As an example, vertex buffer data for an application needs to be mapped to its memory address space, since that data can only be used by the application. The API and driver will allocate this buffer space at runtime either from the graphics memory (frame buffer) or system memory. While access to frame buffer is almost instantaneous, access to system memory is many magnitudes slower because of the communication lag between the GPU and system memory via the chipset. In DX10 resources are mapped according to how frequently they are used (reads/writes) with four resource classes: default, immutable, dynamic, and staging. Using these new resource classes, developers can optimize performance by putting frequently used data in the frame buffer to be closer to the GPU and putting seldom-used data in system memory.

3. Additional constructs to improve efficiency

In graphics, multiple iterations of textures and blending usually take place to produce realistic images, such as re-creating hair moving in the wind or the ripples on the surface of a lake. These multiple loops previously required state changes and extensive CPU work to be performed. With DX10, state objects and constant buffers are now available to manage multiple loops in the rendering pipeline and increase the range of processing that can be done in one pass.

- ***State objects*** define what the graphics pipeline units should do as an object travels through the pipeline. The state objects have information to tell the pipeline which textures to blend or to tell the GS to create more detail for specific vertices in a part of the rendered frame. DX10 handles all of these details by introducing five state object commands, and programmers can work using a high-level language, instead of low-level constructs where they would need to keep track of all the pipeline stage units. The commands ***InputLayout***, ***Sampler***, ***Rasterizer***, ***DepthStencil***, and ***Blend*** are performed in the GPU, with minimal CPU intervention for state changes.
- ***Constant buffers*** store large amounts of predefined values (data) for items in a scene, so the CPU or GPU does not have to keep track of those values constantly. Each buffer stores up to 4096 constants hold information such as camera view/projection and light source color/position/intensity. Since these items have update intervals which may be once per frame or once per object, doing several hundred of these constant updates one at a time required significant CPU overhead when done using DirectX 9 or earlier. In DX10, the constant buffer groups the constants based on frequency of use and does batch processing to update the constants, which significantly reduces CPU use and dependence.

4. DX10 hardware specification set

- In pre-DX10 revisions of the API, hardware vendors were able to provide capability bits to inform the system about what features were supported in hardware. DX10 has changed this scheme. DX10 binds the 3D hardware feature set with a DirectX version number, so consistency across all hardware vendors exists and identifies support for a set of same basic features. Implementation of these features is the key for differentiating the graphics quality seen across vendors. S3 Graphics Chrome 400/500 Series processors have proprietary design implementations that give them an edge over the competition in visual quality and rendering capability.
- The basic programmable and fixed function pipelines of the past have been redesigned or eliminated. New and powerful hardware is capable of extending the implementation scope beyond the limits of rendering-only applications to provide high-throughput computing implementations for physics and AI.

The additional capabilities available in DX10 hardware and software offload most of the runtime events associated with rendering an object from the CPU to the GPU. With new and even more powerful functional units, an expanded instruction set, a new architecture that streamlines the graphics pipeline, efficient memory access methods, and multi-pass rendering capability, DX10 has introduced an astounding ability to generate graphics realism into our computing lives today and in the immediate future.

DX10 Pipeline Introduction

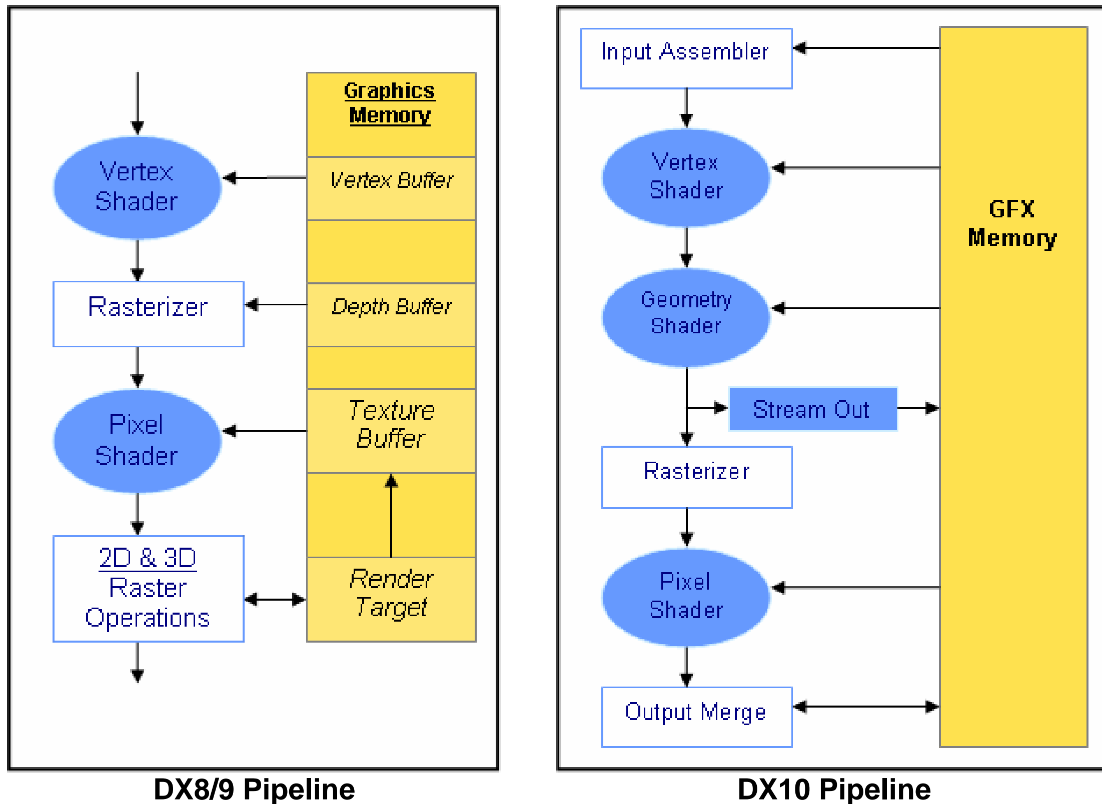

Figure 4. DX8/9 Pipeline Compared to DX10 Pipeline

Figure 4 shows a DX8/9 pipeline (left) compared to the pipeline provided in DX10 hardware. Common functional units, such as the vertex/pixel shaders, rasterizer, and ROP/Output Merge (OM) block, exist in both architectures and perform similar functions. The Input Assembler (IA) converts or replicates input vertex data from incoming streams (vertex structure) to be used by the pipeline. The key difference is the introduction of the Geometry Shader (GS) and Stream Output (SO) described below.

- The ***Geometry Shader*** (**GS**) takes the vertices of a primitive, such as a line, point, or triangle, and will either create additional vertices (generate data) or destroy the vertices. The GS increases the number of vertices by creating additional primitives composed of up to 1024 32-bit vertex data per instance. If a vertex is not needed, the GS can delete it from the rendering pipeline. The GS can also add additional elements to a primitive without needing to create a new vertex stream. In the past, the pipeline could not create or destroy vertices, only modify them. DX10 moves one step ahead by allowing even more flexibility and power in hardware, where the GS performs per-primitive modifications and also accesses adjacent primitive information. For example, in a GS implementation for a realistic shadow rendering, the GS can control a point or line and its neighboring primitive, as well as control the displacement mapping, where more detail can be shown with the creation of new primitives based on height maps.

- *Stream output* can write vertex or primitive information from the VS/GS to a stream buffer in memory immediately after the GS stage. In the past a primitive had to exit the PS and

then it could be written to the render target buffer in memory. Now data in the stream buffer can be used recursively or iteratively by other functional blocks in the pipeline on an as-needed basis to improve data re-use efficiency. Other uses of this feature are physics calculation support such as used in particle systems where ongoing calculations are needed to generate and destroy primitives to simulate water, smoke, and clouds.

- Graphics memory has been changed from an area that stored vertex and texture data separately, to memory where each independent shader unit can access the same data. The data storage formats have also been updated to allow the pipeline to store and use multiple format types to increase flexibility. The memory can store data in arrays which allows recirculation of data and texture fetches by the VS, GS, and PS.

DX10 introduces a unified architecture that builds upon the pipeline diagramed in Figure 4. The DX10 pipeline combines three types of shaders into unified execution units capable of handling VS, GS, and PS instructions. The DX10 pipeline architectural design has solved many issues seen in previous generations of DirectX. It has added significant changes to the software, instruction set, instruction support/capability, as well as requiring new hardware blocks and features.

This new design still has a limitation in the utilization rate for each shader type based on the application (see Figure 5 below). If complex geometry calculations and processing are needed, then the VS may be fully utilized while the PS might remain idle. If an application is pixel processing heavy, then the PS may be running at full speed while the VS will only be partially loaded.

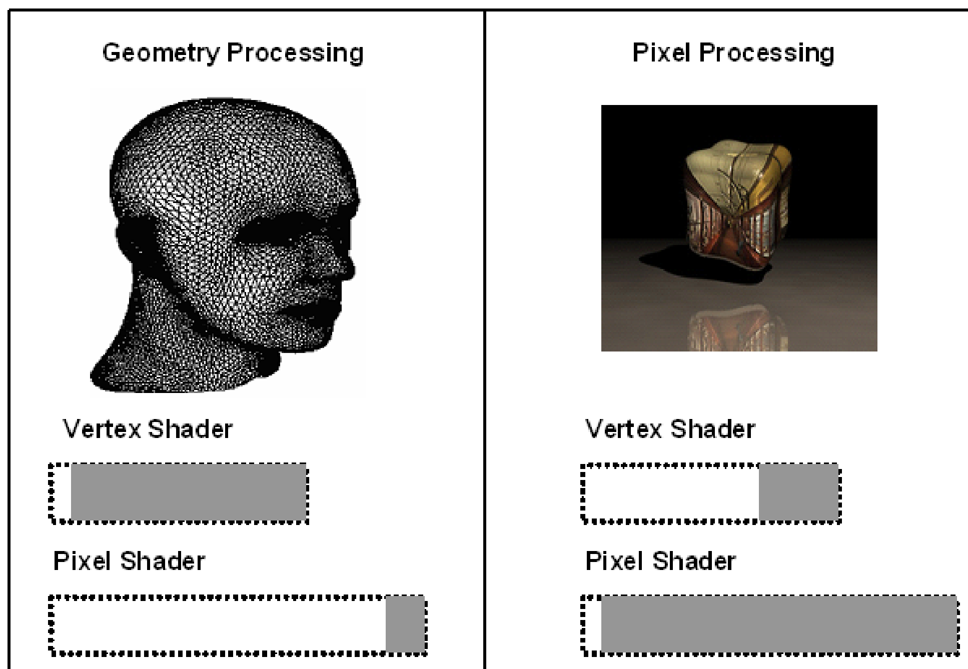

Figure 5. Vertex and Pixel Shader Resource Utilization for Different Applications

DX10 Shader Model 4.0

Shader Model 4.0 (SM4.0) is the new instruction set architecture (ISA) for DX10 that looks at the graphics in a unified way. Some keys advantages of SM4.0 are:

- **Easy Programmability:** Developers do not need to be bogged down with the low-level details of the hardware. In the past, programmers needed to control and write different low-level program code for each individual shader (VS/PS). Each shader was also considered an individual virtual machine that had separate input/output/general registers that had to be tracked for shader I/O, memory transfers, and intermediate data storage. SM4.0 instructions hide the low-level implementation details and incorporate all the pipeline flow control.
- **Flexible Load Balancing:** The new unified ISA allows flexibility. Developers can now look at these shaders units as one cohesive block (single common core virtual machine) instead of separate blocks, as shown in Figure 6. The unified shader is made up of shader blocks that can handle all vertex, pixel, and geometry instructions, so the GPU is fully utilized without concern for shader loading imbalances (geometry processing vs. pixel processing, as shown in Figure 5). There is also additional logic to load balance the shader units to keep all functional units fully utilized. If more pixel processing is needed, then more of the unified shader blocks can be allocated to pixel processing to increase throughput. The same shader-type allocation can be done with the VS and GS, as seen in Figure 7.
- **Unified Shader Code**: Developers code in "shader" instructions, not VS/PS/GS specific code.
- **Programmable Offloading:** The unified model helps offload the CPU state change overhead by incorporating flow control logic that programmers can control.

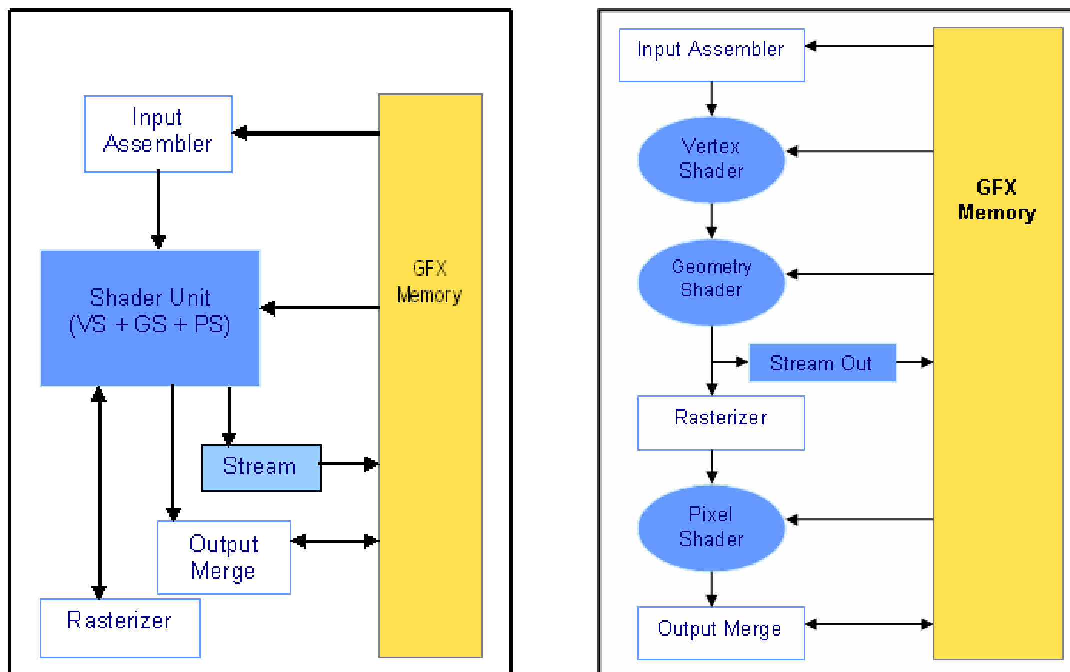

Figure 6. Unified Shader Model (Left) Compared to Basic DX10 Pipeline (Right)

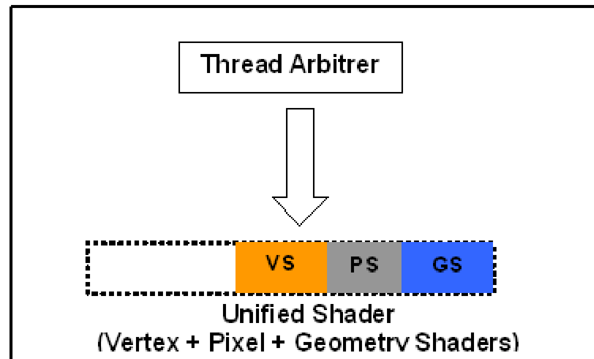

Figure 7. Unified Shader Utilized for Different Shader Types

- Additional resources, data formats, and instructions are available to the programmer for more efficient use and coupling of the graphics hardware to the application level.

Feature	DX9	DX10
Instruction Slots	512	64K
Constant Registers	256	4096 (x16)
Temporary Registers	32	4096
Render Targets	4	8
Textures	16	128
Texture Size	2K x 2K	8K x 8K
Load Operations	No	Yes
Sample Offsets	No	Yes
Flow Control	Static/Dynamic	Dynamic

Table 1. Basic Comparison Table of DX9 and DX10

- ***Full integer and bitwise instructions*** allow the GPU to compute complex algorithms more efficiently in integer format instead of converting between floating point and integer.
- ***Switch statements*** are another great addition because they provide multiple paths/options when rendering objects on a per-primitive basis. This means the GPU can replicate objects (instancing) and also provide unique characteristics for each object independently of the other objects in the scene.
- Increased texture support and size greatly enhance visual quality.
    - In DX9, developers only had 16 textures to work with at a given time and their size limitation was 4096 x 4096. Since the application of multiple textures required multiple texture changes and multiple state changes (large CPU overhead), developers were limited in what they could do. If developers needed multiple textures, they created a texture atlas that combined many smaller textures that could be accessed by indexing into the atlas. This method proved very inefficient since the boundaries between smaller textures were not as clearly defined and the atlas could only hold a certain amount of textures so there was a trade-off between storing fewer (larger) textures or more (smaller) textures.
    - SM4.0 has new instructions and indexed texture arrays, which can store 512 textures with resolutions of up to 8192 x 8192. This method effectively replaces the texture atlas with a large array that can be indexed into easy-to-access multiple textures. In addition, the number of textures a shader can use has increased from 16 to 128, allowing hardware to take advantage of the texture array to add more detail to all objects in a frame.

High-Level Shader Language (HLSL 10)

HLSL 10 is the name given to the programming language developers use to take advantage of DX10 shader and hardware capabilities. The advantages of HLSL are many.

- Application developers do not need to worry about using assembly-like instructions to control the shader and pipeline at the hardware level.
- Applications can offload the task of resource management.
- ***Bind-by-name*** to ***bind-by-position*** allows less overhead at runtime. Bind-by-name in DX9 performs checks like matching input/output between hardware functional units and matching vertex buffer format with the vertex shader. Any type mismatch would cause huge overhead since the hardware was not as flexible. In DX10 shader units have associated signatures with their inputs and outputs. As long as the output of the preceding stage is compatible with the input of the following stage, then the data type mismatch is allowed since DX10 allows multiple data formats to be used at any stage in the pipeline.
- DX10 has a "view" method for representing resources such as vertex buffers or texture maps, which can be read in many different formats, so that they are not type-set. This allows resources to be used in multiple parts of the pipeline. Data from one shader can be used in another for on-the-fly updates, regardless of the format type of the intermediate data.

Examples of DX10 Visual Effects

Figure 8. DX9 Screenshot

Figure 9. DX10 Screenshot

(Source: Flight Simulator X game)

Figure 10. DX9 Screenshot

Figure 11. DX10 Screenshot

(Source: Crysis game)

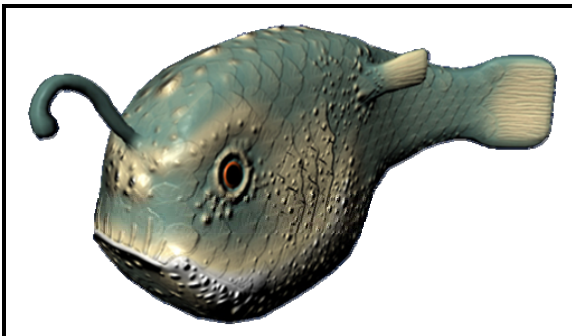

Figure 12.
DX9 (normal mapping)

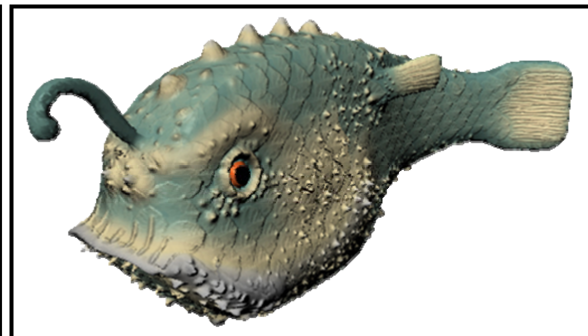

Figure 13.
DX10 (displacement mapping)

(Source: "DirectX 10: The Next Generation in Gaming" – http://windowsvistablog.com)

Figure 14. DX10 Morphing using the Geometry Shader and Stream Output
(Source: Microsoft MSDN Direct3D 10 Samples, http://msdn2.microsoft.com)

Figure 15. DX10 Alpha to Coverage
(Source: Microsoft "Intro to Direct3D 10" presentation by Sam Glassenberg)

Figure 16. DX10 Instancing
(Source: Microsoft MSDN Direct3D 10 Samples, http://msdn2.microsoft.com)

Conclusion

The introduction of DirectX 10 brings an inflection point to the graphics market where the rendering capabilities of advanced hardware and the creativity of software developers can now bring real-life 3D graphics to our daily lives. Features, which once were found only in luxury high-end graphics products costing several hundreds of dollars, can now be achieved using the new DirectX 10 companion GPUs of the S3 Graphics Chrome 400/500 Series product lines.

S3 Graphics Chrome 400/500 Series graphics processors are technological marvels with their multiple programmable DX10/SM4.0 execution units which provide a unified shader architecture. With support for all the new DX10 features such as ***stream processing***, ***geometry shaders***, and ***HLSL 10 programming***, S3 Graphics continues its position as an industry leader in visual computing for current and future generations of the DirectX 3D API.