Technical Brief

Microsoft DirectX 10:
The Next-Generation Graphics API

November 2006
TB-02820-001_v01

Microsoft DirectX 10: The Next-Generation Graphics API

Introduction

Microsoft's release of DirectX 10 represents the most significant step forward in 3D graphics API since the birth of programmable shaders. Completely built from the ground up, DirectX 10 features a highly optimized runtime, powerful geometry shaders, texture arrays, and numerous other features that unlock a whole new world of graphical effects.

DirectX has evolved steadily in the past decade to become the API of choice for game development on the Microsoft Windows platform. Each generation of DirectX brought support for new hardware features, allowing game developers to innovate at an amazing pace. NVIDIA has led the 3D graphics industry by being the first to launch new graphics processors to provide full support for each generation of DirectX. We are proud to continue this tradition for DirectX 10.

NVIDIA was the first company to introduce support for DirectX 7's hardware-accelerated transform and lighting engine with its award-winning NVIDIA® GeForce® 256 graphics processor. When DirectX 8 introduced programmable shaders in 2000, NVIDIA led the way with the world's first programmable GPU, the GeForce 3. The GeForce FX, introduced in 2003, was the first GPU to support 32-bit floating-point colors, a key feature of DirectX 9. When Shader Model 3.0 was announced, NVIDIA once again led the way with its popular GeForce 6 and GeForce 7 series of graphics processors.

DirectX 10 is the first complete redesign of DirectX since its birth. To carry on the tradition of serving as the premier DirectX platform, we designed a new GPU architecture from scratch specifically for DirectX 10. This new architecture, which we refer to as the GeForce 8800 series architecture, is the result of over three years of intensive research and development with intimate collaboration from Microsoft. The first product based on this new architecture is the GeForce 8800 GTX—the world's first DirectX 10–compliant GPU.

The GeForce 8800 GTX is a GPU of many firsts. It is simultaneously the world's largest, most complex, and most powerful GPU. With a massive array of 128-stream processors operating at 1.35 GHz, the GeForce 8800 GTX has no peer in performance. Built with image quality as well as speed in mind, its new 16× antialiasing, 128-bit HDR rendering and angle-independent anisotropic filtering engines produce pixels that rival Hollywood films. This paper will discuss the new features behind DirectX 10 and how the GeForce 8800 architecture will bring them to life.

How This Paper Is Organized

This paper is organized into the following six sections.

- **A New Architecture Designed for High Performance**
  This section discusses the problem of high CPU overhead for graphics APIs and how DirectX 10 addresses this problem.
- **Shader Model 4.0**
  This section discusses how the new unified shading core and vastly improved resources affect graphics.
- **Geometry Shader + Stream Output**
  This section explores the geometry shader and the stream output function.
- **HLSL 10**
  This section discusses the new features behind the latest version of the DirectX 10's high-level shader language (HLSL), such as constant buffers and views.
- **Other Improvements**
  This section reviews other improvements in the API that are not part of any of the prior sections, but nevertheless have significant value for next-generation titles.
- **Next-Generation Effects**
  This section takes a glimpse at the future by showcasing three next-generation effects powered by DirectX 10.

A New Architecture Designed for High Performance

Overcoming High API Overhead

DirectX has enjoyed great popularity with developers thanks to its rich features and ease of use. However, the API has always suffered one major problem—a high CPU overhead.

Before graphics-programming APIs were introduced, 3D applications issued their commands directly to the graphics hardware. While this was very fast, the application had to hand out different commands to different hardware, making development difficult and error prone. As the range of graphics hardware grew, this soon became infeasible.

Graphics APIs like DirectX and OpenGL act as a middle layer between the application and the graphics hardware. Using this model, applications write one set of code and the API does the job of translating this code to instructions that can be understood by the underlying hardware. This greatly eases the development process by allowing developers to concentrate on making great games instead of writing code to talk to a vast assortment of hardware.

The problem with this model is that every time DirectX receives a command from the application, it has to process the command before knowing how to issue it to the hardware. Since this processing is done on the CPU, it means all 3D commands now carry a CPU overhead. This overhead causes two problems for 3D graphics: it limits the number of objects that can be rendered and it limits the number of unique effects that can be applied to a scene.

In the first case, since each draw call carries a fixed API overhead, only a certain number of draw calls can be used before the system is completely CPU bound. This imposes a limit on the number of objects that can be drawn. To combat this problem, developers use a technique called *batching*, where multiple objects are drawn as a group. But when objects differ in material properties, batching cannot be applied.

A high API overhead not only limits rendering performance, it also limits the visual richness of the application. State change commands (as well as draw calls) produce significant API overhead. This includes changing textures, shaders, vertex formats, and blending modes. These state change operations are crucial in providing unique appearances to the world; without them, every object's surface would look the same. However, since state change commands are accessed via the API, they carry a CPU overhead. State changes also occur much more often than draw calls because multiple effects may be applied to a single object. Due to the high cost of state changes, developers avoid using a large variety of textures and unique materials. The result is that games are not as visually rich as they should be.

DirectX 10—A New 'Ground Up' Architecture

One of the chief objectives of DirectX 10 is to significantly reduce the CPU overhead of rendering. DirectX 10 attacks the overhead problem in three ways. First, the cost of draw calls and state changes is reduced by completely redesigning the performance-critical parts of the core API. Second, new features are introduced to reduce CPU dependence. Third, new features are added to allow more work to be done in one command.

New Runtime

DirectX 10 introduces a new runtime that significantly reduces the cost of draw calls and state changes. The new runtime has been redesigned to map much closer to graphics hardware, allowing it to perform far more efficiently then before. Legacy fixed-function commands from previous versions of DirectX have been removed. This reduces the number of states that need to be tracked, providing a cleaner and lighter runtime. To support this new runtime, we designed the GeForce 8800 architecture with all these changes in mind. Our new driver, supporting the new Windows Vista Driver Model, is tuned for optimal performance on DirectX 10.

A key runtime change that greatly enhances performance is the treatment of validation. Validation is a process that occurs before any draw call is executed. The validation process ensures that commands and data sent by the application are correctly formatted and will not cause problems for the graphics card. Validation also helps maintain data integrity, but unfortunately introduces a significant overhead.

Table 1. DirectX 9 vs. DirectX 10 Validation. DirectX 9 validates resources for every use. DirectX 10 only needs to validate resource once during creation, greatly reducing validation overhead.

DirectX 9 Validation	DirectX 10 Validation
Application starts Create Resource Game loop (executed millions of times) • Validate Resource • Use Resource • Show frame Loop End App ends	Application starts Create Resource Validate Resource (executed once) Game loop (executed millions of times) • Use Resource • Show frame Loop End App ends

In DirectX 10, objects are validated when they are created rather than when they are used. Since objects are only crated once, validation only occurs once. Compared to DirectX 9 where objects are validated once for each use, this represents a huge saving.

Less CPU Intervention

DirectX 10 introduces several new features that greatly reduce the amount of CPU intervention. These include texture arrays, predicated draw, and stream out.

Traditionally, switching between multiple textures incurred a high state-change cost. As a workaround, artists stitched together several small textures into a single large texture called a *texture atlas*, allowing them to use multiple textures without paying the cost of creating and managing multiple textures. However, since the largest texture size permitted in DirectX 9 is 4048 × 4048, this approach was fairly limited.

DirectX 10 introduces a new construct called *texture arrays*, which allow up to 512 textures to be stored in an array structure. Also included are new instructions that allow a shader program to dynamically index into the texture array. Since these instructions are handled by the GPU, the amount of CPU overhead associated with managing multiple textures is greatly reduced.

Predicated draw is another feature that no longer requires CPU intervention. In typical 3D scenes, many objects are often entirely overlapped by other objects. In such cases, drawing the occluded object takes up rendering resources, but has no effect on the final image. Advanced GPUs use various hardware-based culling methods to detect these conditions to avoid processing pixels that will never be visible. But nevertheless, some redundant overdraw still occurs. To prevent this waste, developers use a technique called predicated draw**,** where complex objects are first drawn using a simple box approximation. If drawing the box has no effect on the final image, the complex object is not drawn at all. This is also known as an *occlusion query*. In previous versions of DirectX, solving the occlusion query required using both the CPU and the GPU. With DirectX 10, this process is done entirely on the GPU, eliminating all CPU intervention.

Lastly, DirectX 10 introduces a new function called *stream out* that allows the vertex or geometry shader to output their results directly into graphics memory. This is a significant improvement compared to previous versions of DirectX, where results must pass through to the pixel shader before they can exit the pipeline. With stream output, results can be iteratively processed on the GPU with no CPU intervention.

Do More with Each Command

State management has always been a costly affair with DirectX 9. The task of repeatedly setting up textures, constants, and blending modes incurs a significant CPU overhead. Typically, applications use these commands in rapid succession. But because DirectX 9 does not have any way of batching these operations, their accumulated overhead greatly limits rendering performance.

DirectX 10 introduces two new constructs—*state objects* and *constant buffers*—permitting common operations to be performed in batch mode, greatly reducing the cost of state management.

State Objects

Prior to DirectX 10, states were managed in a very fine-grained manner. States define the behavior of various parts of the graphics pipeline. For example, in the vertex shader, the vertex buffer layout state defines the format of input vertices. In the output merger, the blend state determines which blend function is applied to the new frame. In general, states help define various vertex and texture formats and the behavior of fixed-function parts of the pipeline. In DirectX 9's state management model, the programmer manages state at low level—often many state changes were required to reconfigure the pipeline. To make state changes more efficient, DirectX 10 implements a new, higher-level state management model using *state objects*.

The huge range of states in DirectX 9 is consolidated into five state objects in DirectX 10: InputLayout (vertex buffer layout), Sampler, Rasterizer, DepthStencil, and Blend. These state objects capture the essential properties of various pipeline stages. Leveraging them, state changes that used to require multiple commands can be performed using only one call, greatly reducing the state change overhead.

Constant Buffers

Another major feature being introduced is the use of *constant buffers*. Constants are predefined values used as parameters in all shader programs. For example, the number of lights in a scene along with their intensity, color, and position are all defined by constants. In a game, constants often require updating to reflect world changes. Because of the large number of constants and their frequency of update, constant updates produce a significant API overhead.

Constant buffers allow up to 4096 constants to be stored in a buffer that can be updated in one function call. This batch mode of updating greatly alleviates the overhead cost of updating a large number of constants.

Image courtesy of Microsoft's DirectX 10 SDK

Figure 1. DirectX 10's drastically reduced CPU overhead makes it possible to render a huge number of objects with incredible detail.

In Summary: Faster, Lighter, Smarter

To sum up the improvements outlined in this section: DirectX 10 has been rebuilt from the ground up to offer the highest performance by mapping closer to the hardware and leveraging creation time validation. It requires less CPU intervention—thanks to new features like texture arrays, predicated draw, and stream output. With state objects and constant buffers, the task of managing state and constants is more efficient and streamlined. Together, these contribute to a major reduction in the overhead required to render using the DirectX API.

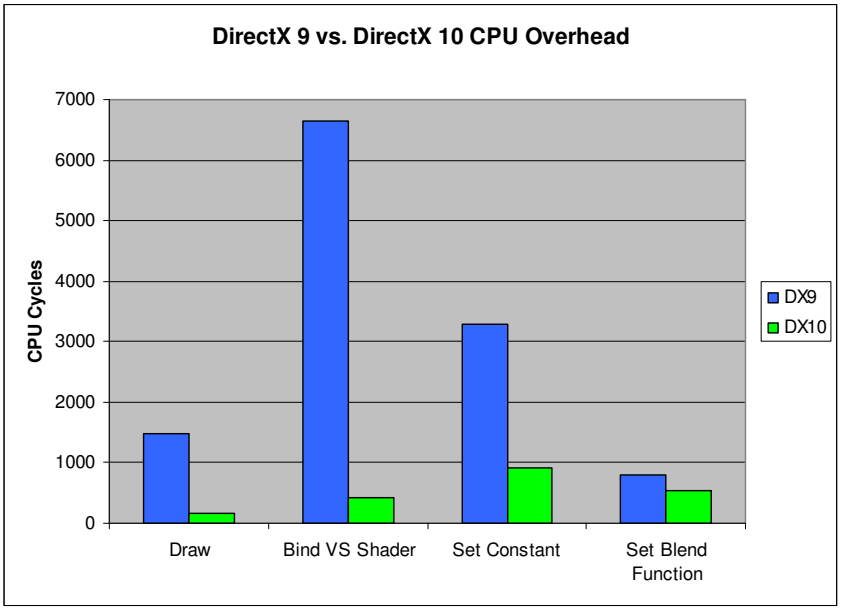


Figure 2. DirectX 9 vs. DirectX 10 CPU overhead

Shader Model 4.0

DirectX 10 introduces Shader Model 4.0, which provides several key innovations:

- A new programmable stage called the *geometry shader*, which allows per-primitive manipulation
- A unified shading architecture that uses a unified instruction set and common resources across vertex, geometry, and pixel shaders
- Shader resources that are vastly approved across the board

Geometry shaders represent a major step forward in the programmable graphics pipeline, allowing for the first time the generation and destruction of geometry data on the GPU. Coupled with the new stream out function, algorithms that were once out of reach can now be mapped to the GPU. Geometry shaders are discussed in the next section of this paper.

Unified Shading Architecture

In prior versions of DirectX, pixel shaders lagged behind vertex shaders in constant registers, available instructions, and instruction limits. As such, programmers had to learn how to use vertex and pixel shaders as separate entities.

Shader Model 4.0 differs from prior versions by providing a unified instruction set with the same number of registers (temporary and constant) and inputs across the programmable pipeline*. Games developed under DirectX 10 do not need to spend time working around stage-specific limitations; all shaders are able to tap into the entire resources of the GPU.

More Than a Hundred Times the Resources of DirectX 9

Shader Model 4.0 provides an astounding increase in resources for shader programs. In previous versions of DirectX, developers were forced to carefully manage scarce register resources. DirectX 10 provides over two orders of magnitude increases in register resources: temporary registers are up from 32 to 4096, and constant registers are up from 256 to 65,536 (sixteen constant buffers of 4096 registers). Needless to say, the GeForce 8800 architecture provides all these DirectX 10 resources.

Table 2. DirectX 9 vs. DirectX 10 Resources

Resources	DirectX 9	DirectX 10
Temporary registers	32	4096
Constant registers	256	16 × 4096
Textures	16	128
Render targets	4	8
Maximum texture size	4048 × 4048	8096 × 8096

More Textures

Shader Model 4.0 brings support for texture arrays, liberating artists from the tedious work of creating texture atlases. Prior to Shader Model 4.0, the overhead cost associated with changing textures meant that it was infeasible to use more than a few unique textures per shader. To help combat this problem, artists packed small individual textures into a large texture called a *texture atlas*. At runtime, the shader performed an additional address calculation to find the right texture within the texture atlas.

Texture atlases have two major issues. First, the boundaries between textures within a texture atlas receive incorrect filtering. Second, since the largest texture size is 4096 × 4096 in DirectX 9, texture atlases can only hold a modest collection of small textures or a few large textures.

Texture arrays solve both problems by formally allowing textures to be stored in an array format. Each texture array can store up to 512 equally sized textures. The maximum texture resolution has also been extended to 8192 × 8192. To facilitate their use, the maximum number of textures that can be used by a shader has been increased to 128, an eight-fold increase from DirectX 9. Together, these features represent an unprecedented leap in texturing power.

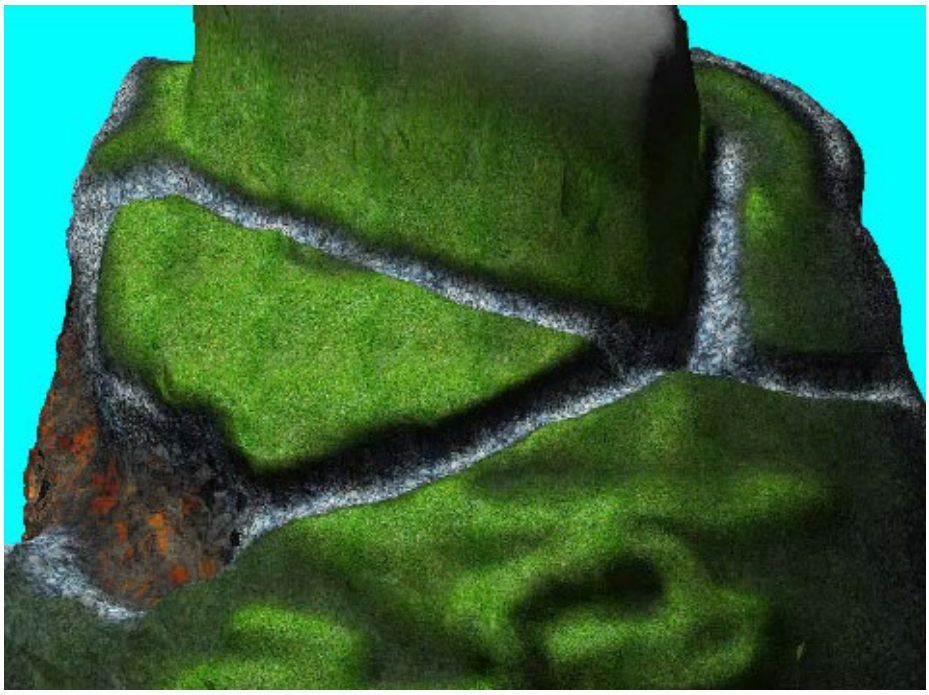


Figure 3. Using texture arrays, much greater detail can be applied to objects.

More Render Targets

Multiple render targets, a popular feature of DirectX 9, allow a single pass of the pixel shader to output four unique rendering results, effectively rendering four interpretations of the scene in one pass. DirectX 10 takes this further by supporting eight render targets. This greatly increases the complexity of shaders that can be used. Deferred rendering and other image space algorithms will benefit immensely.

Two New HDR Formats

High dynamic-range rendering became popular thanks to the support of floating-point color formats in DirectX 9. Unfortunately, floating-point representation takes up more space than integer representation, limiting performance and accessibility. For example, the popular FP16 format takes up 16 bits per color component—twice the storage of standard rendering using an 8-bit integer.

Image courtesy of Futuremark

Figure 4. High dynamic-range rendering

DirectX 10 introduces two new HDR formats that offer the same dynamic range as FP16 but at half the storage. The first format, R11G11B10, is optimized to be used as a floating-point render target. It uses 11 bits for red and green, and 10 bits for blue. The second floating-point format, RGBE, is designed for floating-point textures. It uses a 5-bit shared exponent across all colors with 9 bits of mantissas for each component. These new formats allow high dynamic-range rendering without the high storage and bandwidth costs.

For the highest level of precision, DirectX 10 supports 32 bits of data per component. The GeForce 8800 GTX fully supports this feature, which can be used from high-precision rendering to scientific computing applications.

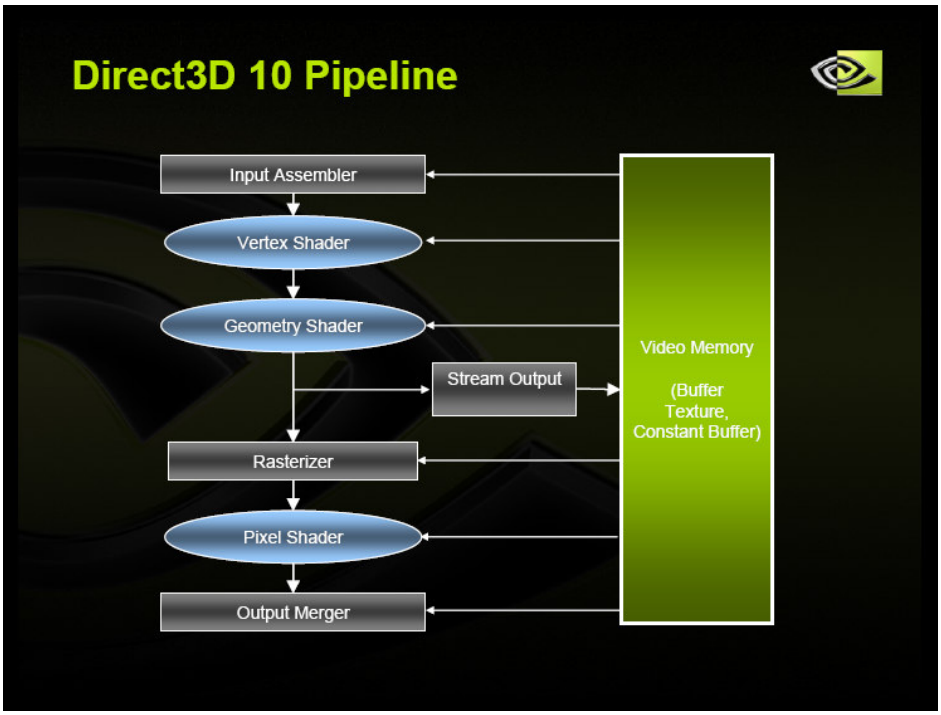
Summary

The huge increase in register space, texturing ability, and new rendering options will allow developers to completely rethink their graphics engines, and paves the way for creating infinitely more detailed worlds.

Geometry Shader + Stream Output

Up until now, graphics hardware only had the ability to manipulate existing data on the GPU. Both the vertex and the pixel shader apply their program to data that already exists in memory. This model has been very successful in performing complex skinning on existing meshes, as well as doing accurate per-pixel calculations on existing pixels. However, this model does not permit the generation of new data on the graphics processor (*instancing* is an exceptions). When objects are created dynamically in a game (such as the appearance of new weapons or power-ups), it is done by invoking the CPU. Since most games are already CPU limited, little opportunity exists for generating large amounts of new data dynamically on the CPU during gameplay.

The geometry shader, introduced in Shader Model 4.0, will for the first time allow data to be generated on the graphics processor. This revolutionizes the role of the GPU in the system from being a processor that can only process existing data to one that can generate and manipulate data at incredible speeds. A whole new spectrum of algorithms—out of reach on previous graphics systems—is now possible. Using DirectX 10 and the GeForce 8800 GTX, popular algorithms such as stencil shadows, dynamic cube maps, and displacement mapping which have all relied either on the CPU or multipass rendering can now be performed with much greater efficiency.



Adds the geometry shader and stream output, allowing for iterative calculations on the GPU with no CPU intervention

Figure 5. The DirectX 10 pipeline

The geometry shader sits in the middle of the DirectX 10 graphics pipeline between the vertex shader and the rasterizer. It takes vertices transformed by the vertex shader as input, and for each vertex the geometry shader emits up to 1024 vertices as output. The ability to generate huge amounts of new data is also known as *data amplification*. Likewise the geometry shader can remove data on the GPU by outputting fewer vertexes than it received, thus doing *data minimization*. Together these two features make the geometry shader exceptionally powerful in how it can influence the flow of data within the GPU.

Displacement Mapping with Tessellation

The geometry shader will finally allow generalized displacement mapping on the GPU. Displacement mapping is a popular technique used in offline rendering systems that allows very complex models to be rendered using a simple model and special textures called *height maps*. A height map is a grayscale texture that expresses the height of vertices at each point in the model. During rendering, the low-polygon model is tessellated with additional polygons and, based on information encoded in the height map, the polygons are extruded to give the representation of the full detailed model.

Since no part of the GPU could generate additional data in DirectX 9, the low-polygon model could not be tessellated. Hence only a limited implementation of displacement mapping was possible. Now with DirectX 10 and the power of the GeForce 8800 GTX, thousands of additional vertices can be generated on the fly, allowing true displacement mapping with tessellation to be rendered in real time.

New Algorithms Based on Adjacency

The geometry shader can take in three types of primitives: vertices, lines, and triangles. Likewise it can output any of these primitives, though it cannot output more than one type per shader. In the case of lines and triangles, the geometry shader also has the ability to access adjacency information. Using adjacent vertices of lines and triangles allows many powerful algorithms to be implemented. For example, adjacency information can be used to calculate silhouette edges for objects used for cartoon rendering and realistic fur rendering. See Figure 7 on the following page.

Figure 6. Nonphotorealistic rendering (NPR) using the geometry shader

Stream Output

Prior to DirectX 10, geometry needed to be rasterized and sent to the pixel shader before any results could be written to memory. DirectX 10 introduces a new feature called *stream output,* which allows data to be passed directly from either the vertex or the geometry shader straight into frame buffer memory. The output can then be fed back to the pipeline to allow iterative processing. When the geometry shader is used in conjunction with stream output, the GPU can process not only new graphical algorithms, but be far more effective with physics and general computations as well.

With the ability to generate, destroy, and stream data, a fully featured particle system can now be run exclusively on the GPU. Particles begin their life in the geometry shader and are amplified by the generation of additional points. The new particles are streamed out to memory and fed back to the vertex shader for animation. After a set period of time, their brightness starts to fade and eventually the geometry shader deletes them from the system.

HLSL 10

DirectX 10 brings several key additions to the popular high-level shader language (HLSL) that was first released with DirectX 9. These include constant buffers for fast constant updates, views for binding resources to the pipeline, integer and bitwise instructions for more generalized computation, and switch statements for flexible shader-based instancing.

Constant Buffers

Shader programs require the use of constants to define various parameters such as the position and color of lights, the position of the camera, view and projection matrices, and material parameters like specularity. During rendering, these constants often require updating. When hundreds of constants are in use, updating them creates significant CPU overhead. Constant buffers are a new DirectX 10 construct that groups constants into buffers and updates them in unison based on the frequency of use.

DirectX 10 supports a total of 16 constant buffers per shader program, each capable of holding 4096 constants. In contrast, DirectX 9 supports only 256 constants per shader program.

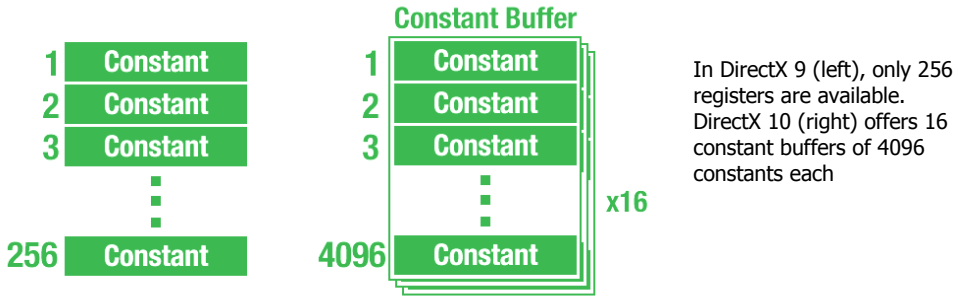


Figure 7. Comparing DirectX 9 with DirectX 10

Not only does DirectX 10 offer many more constants, the speed with which they can be updated is dramatically faster than DirectX 9. When grouped in a constant buffer, constants can be updated using one call, rather than individually.

Because various constants are updated at different intervals, organizing them by frequency of use is much more efficient. For example, the camera and view projection matrix will change every frame, whereas material parameters such as texture may change for every primitive. To optimize for this pattern of usage, constant buffers are designed to be updated depending on the frequency of use—per-primitive constant buffers update for every new primitive, while per-frame constant buffers update per frame. This greatly minimizes the overhead cost of updating constants, allowing shaders to run more efficiently than before in DirectX 9.

Views

Shader resources are strongly typed in DirectX 9. For example, a vertex buffer used in the vertex shader cannot be reinterpreted as a texture in the pixel shader. This formed a strong bond between specific resources and specific stages in the pipeline, limiting the scope for resource sharing across different stages of the pipeline.

DirectX 10 removes the notion of strongly typed resources. When a resource is created, it is simply treated as a bit field in memory. To use these untyped bit fields in memory, a *view* construct is used. The view of a resource is an interpretation of that resource into a particular format. Using views, the same resource can be read in many different ways. DirectX 10 supports two views of the same resource at any one time.

Using multiple views, it is possible to reuse the same resource for different purposes in different parts of the pipeline. For example, one could use the pixel shader to render new geometric data into a texture. In a subsequent pass, the vertex shader can use a view to interpret the texture as a vertex buffer and use the rendering result as geometry Views enable a more flexible treatment of resources and allow developers to create more interesting effects by reusing resources in the pipeline.

Integer and Bitwise Instructions

The new high-level shader language adds support for integer and bitwise instructions. The ability to work natively with integer instructions makes many algorithms much easier to implement on the GPU. Developers can finally compute exact solutions using integers rather than converting from floating-point. Array indices can now be easily calculated. When used with the unfiltered load instruction, integer calculations can now be performed on large arrays of numbers, enhancing the graphics processor's general purpose computing ability.

Switch Statement

HLSL 10 includes support for the commonly used switch statement in programming. This makes programming easier for shaders that execute a large number of paths. One use would be über-shaders—large shader programs that encapsulate many smaller shader programs. An über-shader using the switch statement can allow different effects to be applied on a per-primitive basis by switching based on the material ID. Entire armies can now be rendered with unique effects on each character.

Other Improvements

In addition to the major changes in the runtime—a new shader model and HLSL 10—DirectX 10 includes many less notable but important changes that will benefit next-generation titles.

Alpha to Coverage

Games often use polygons with transparent portions (alpha textures) to represent complex geometry such as leaves, chain-linked fences, and railings. Unfortunately when alpha textures are rendered, the intersection between the opaque and transparent portions show heavy aliasing. Alpha blending can solve this problem, but requires that polygons be sorted and rendered in back-to-front order. This is currently infeasible to do without a large performance penalty.

DirectX 10 includes support for a technique called *alpha to coverage*. Using this scheme, partially transparent alpha values are resolved using multisample antialiasing. This gives the transition a smooth, antialiased edge without the performance penalty of alpha blending. Outdoor games with heavy use of foliage will benefit greatly from this feature.

Figure 8. Using alpha to coverage, leaf edges in alpha textures receive the smoothing effects of antialiasing.

Shadow Map Filtering

Shadow maps have become the most popular algorithm for rendering realistic shadows. However, shadow maps have always suffered heavy aliasing due to their limited resolution. To alleviate this, DirectX 10 adds official support for shadow map filtering. With shadow map filtering, arbitrary samples of the shadow map can be read and blended to create soft, realistic shadows.

Access to Multisampling Subsamples

The current method of resolving multisample antialiasing (MSAA) is done at a very late stage in the graphics pipeline called *scan out*. At this stage, the subsamples cannot be recorded in memory for subsequent use. This limitation means MSAA cannot be used in deferred shading graphics engines.

In DirectX 10 it is possible to bind an MSAA render target as a texture, where each subsample can be accessed individually. This gives the programmer much better control over how subsamples are used. Deferred rendering can now benefit from MSAA. Custom resolves can also be computed.

Next-Generation Effects

DirectX 10 offers many new features for developers to create next-generation effects. However, due to DirectX 10's highly evolved programmability, it's not immediately obvious what these features will translate to visually. In fact, some of the most compelling effects enabled by DirectX 10 are not the result of a single feature, but the result of many features working in unison. In this section we look at three next-generation graphics techniques in detail and show how they leverage the new DirectX 10 pipeline.

Next-Generation Instancing

DirectX 9 introduced a feature known as *instancing*, where one object can have its instance drawn in different locations and orientations by using only one draw call. This featured was designed to lower CPU overhead for drawing large number of objects and was popular with developers for drawing lots of objects of similar appearance. For example, littered rocks or armies of identical soldiers could be drawn very efficiently by using instancing. The drawback was that the instanced objects were effectively just clones; they could not use different textures or shaders.

All of the objects in the scene are instanced. Even the blades of grass are uniquely generated by the geometry shader and instanced to cover the island.

Figure 9. Image rendered using only a few draw calls

DirectX 10 liberates instancing from these restrictions. Instanced objects no longer need to use the same textures thanks to texture arrays; each object can now reference a unique texture by indexing into a texture array. Likewise, instanced objects can use different shaders thanks to HSLS 10's support for switch statements. An über-shader can be written that describes a dozen materials. During rendering, a forest of trees can all reference this über-shader but point to different shaders within, allowing unique effects to be applied to instanced objects.

Even animation can be customized thanks to the switchable vertex shaders and constant buffers. With sixteen buffers of 4096 constants available, a huge number of matrix palate animations can be stored and referenced by the vertex shader.

Gone are the days where instancing meant a world of clones. DirectX 10's advanced instancing features will allow each object to have its own personality. Be it textures, pixel shaders, or vertex shaders, each object will breathe and live a life of its own.

Per-Pixel Displacement Mapping

Earlier on we discussed the benefits of displacement mapping by generating new vertices in the geometry shader. Using the combined power of the geometry shader and pixel shader, it is also possible to perform displacement mapping without generating and displacing vertices. By not producing new geometry, this method offers higher performance than standard displacement mapping.

Per-pixel displacement mapping works by sampling the height map with rays cast from the camera. Based on the sampled value, the pixel shader then renders the color value of the displaced surface in screen space.

The first step requires the geometry shader to extrude an imaginary prism for every triangle. The height of this prism represents the maximum displacement distance. Inside the prism, the height map encodes the displacement of the surface. During rendering, the pixel shader fires rays into the prism and calculates the intersection between the ray and the height map. Based on this value, the pixel is rendered.

Per-pixel displacement mapping improves upon previous screen space techniques (parallax, relief, and offset mapping) by offering correct results at object silhouettes. It is also very robust, working on static as well as dynamic geometry.

Thanks to the combined power of DirectX 10 and the GeForce 8800 GTX, per-pixel displacement mapping can be rendered in real time, bring this lizard to life.

Image courtesy of Microsoft's DirectX 10 SDK

Figure 10. Combined power of DirectX 10 and GeForce 8800 GTX

Procedural Growth Simulation

When we introduced programmable shaders with the GeForce 3 graphics processor, many developers were excited about the possibility of procedurally generated effects. Procedural effects are based on real-time calculations rather than pre-made vertex or texture data. However, due to the inability to generate new data on the GPU, procedural effects have largely been used for animating existing data like fire, water, and particle effects.

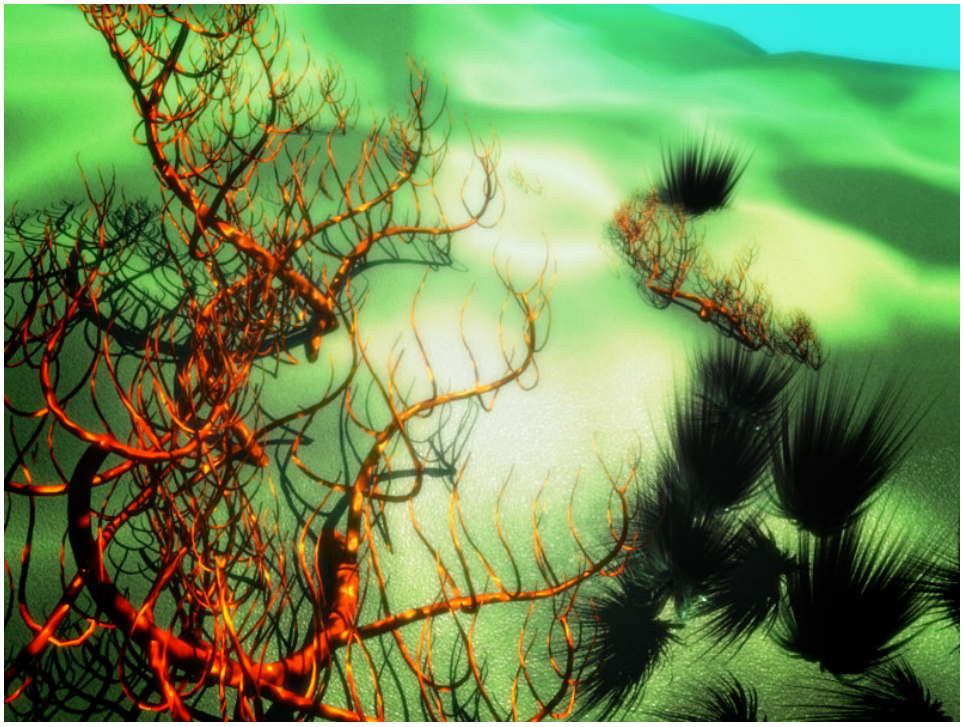


Image courtesy of Niko Suni

Figure 11. Procedurally simulated coral growth

Now with DirectX 10 and the geometry shader, the world can be rendered with life—growing, changing, and decaying. The image in Figure 12, rendered with DirectX 10, shows the simulation of a whole ecosystem underwater. The red coral is defined by an L-system, which is the formal grammar used to describe the growth of plants in botany. Using the L-system as a model, the coral begins its life as a few seed branches in the vertex shader. The geometry shader takes the initial branches and simulates the next phase in the coral's life based on the L-system. Older branches grow thicker and intricate new branches are generated thanks to its ability to introduce additional vertices into the model. The coral's texture is given definition and detail in the pixel shader. Physics calculations simulating the flow of water and its effects on the leaves are also computed. As a final touch, the result is tone-mapped to give the scene depth and richness.

With procedural growth, the game world will not be forced to use static, immutable objects. Games today cannot depict the intricate change of plants and animals in real time due to high-CPU overhead and limited GPU programmability. With DirectX 10, both problems are given generous solutions. The next generation of titles will depict worlds ever changing and filled with living, breathing life.

Conclusion

DirectX 10 is the most significant step forward in many years for graphics. By being the first to fully support all the features of DirectX 10, NVIDIA has once again proven to be the definite platform for DirectX. With powerful features like geometry shaders, stream output, and texture arrays, it will be now possible to render scenes of unprecedented scale, detail, and dynamism.

Together, Microsoft's DirectX 10 and NVIDIA's GeForce 8800 GTX represent the ultimate graphics platform.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com