Direct3D 10

David Blythe
Microsoft Corporation

Direct3D 10 is a new GPU system architecture. The architecture consists of a pipeline abstraction, an API for manipulating the pipeline, and a unified programming abstraction for the programmable stages of the pipeline. While the architecture is new, it is similar to previous Direct3D system architectures.

This chapter includes material that describes details of the architecture as well as sample programs using this new architecture.

The Direct3D System: This paper appears in the SIGGRAPH 2006 proceedings and describes the system architecture and some of the motivation for various architectural choices.

Pipeline : Provides an overview of the pipeline and the programmable stages, describing the common shader core.

Resources: Describes the memory resource types and how they are mapped to the pipeline using views.

Samples: This is a collection of sample techniques that highlight architectural features of the new pipeline – notably the Geometry Shader and Stream Output stages.

Much of this material is drawn from (a Beta release of) the Direct3D 10 SDK. The source code , updates to the sample documentation, as well as more comprehensive reference documentation are available from the SDK at `http://msdn.microsoft.com/directx`.

1. The Pipeline

The Direct3D 10 programmable pipeline is designed for generating graphics for real-time gaming applications. The conceptual diagram below illustrates the data flow from input to output through each of the piprlinr stages.

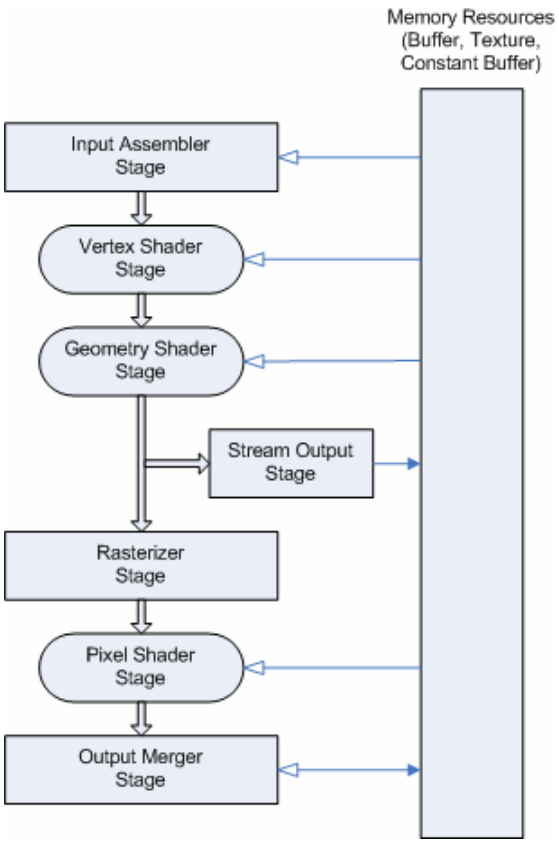


Figure 1: Direct3D 10 Pipeline

All of the stages are configured using the Direct3D 10 API. Stages featuring common shader cores (the rounded rectangular blocks) are programmed using the HLSL programming language. As you will see, this makes the pipeline extremely flexible and adaptable. The purpose of each of the stages is listed below.

- **Input Assembler Stage** - The input assembler stage is responsible for supplying data (triangles, lines and points) to the pipeline.
- **Vertex Shader Stage** - The vertex shader stage processes vertices, typically performing operations such as transformations, skinning, and lighting. A vertex shader always takes a single input vertex and produces a single output vertex.
- **Geometry Shader Stage** - The geometry shader processes entire primitives. Its input is a full primitive (which is three vertices for a triangle, two vertices for a line, or a single vertex for a point). In addition, each primitive can also include the vertex data for any edge-adjacent primitives. This could include at most an additional three vertices for a triangle or an additional two vertices for a line. The Geometry Shader also supports limited geometry amplification and de-

amplification. Given an input primitive, the Geometry Shader can discard the primitive, or emit one or more new primitives.

- **Stream Output Stage** - The stream output stage is designed for streaming primitive data from the pipeline to memory on its way to the rasterizer. Data can be streamed out and/or passed into the rasterizer. Data streamed out to memory can be recirculated back into the pipeline as input data or read-back from the CPU.
- **Rasterizer Stage** - The rasterizer is responsible for clipping primitives, preparing primitives for the pixel shader and determining how to invoke pixel shaders.
- **Pixel Shader Stage** - The pixel shader stage receives interpolated data for a primitive and generates per-pixel data such as color.
- **Output Merger Stage** - The output merger stage is responsible for combining various types of output data (pixel shader values, depth and stencil information) with the contents of the render target and depth/stencil buffers to generate the final pipeline result.

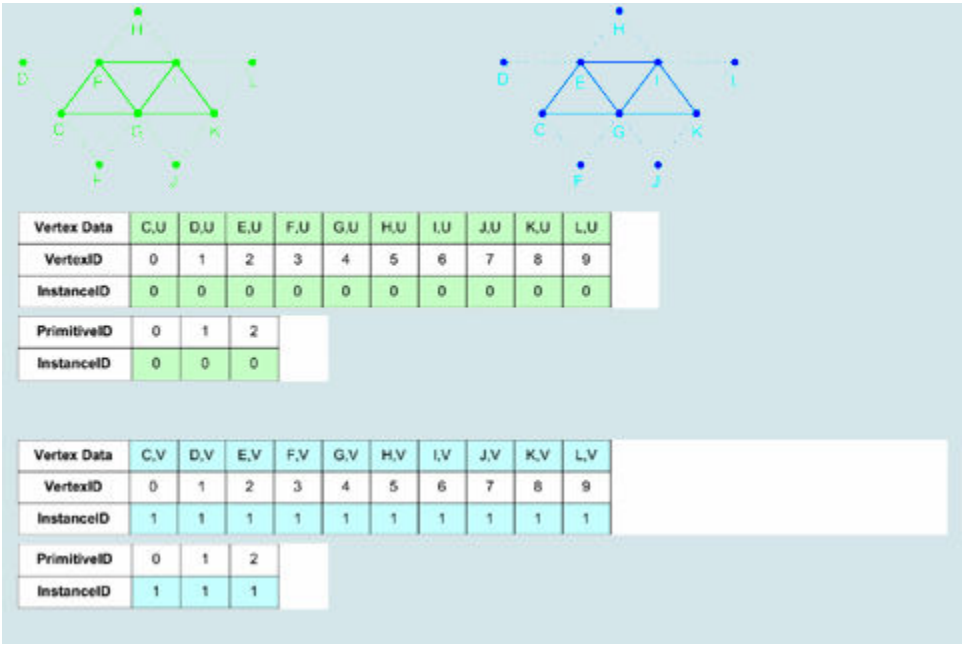
1.1.1 System Generated Values

As stated earlier, system values are generated by the IA to allow certain efficiencies in shader operations. By attaching data such as:

- InstanceID (visible to VS)
- VertexID (visible to VS)
- PrimitiveID (visible to GS/PS)

A subsequent shade stage may look for these system values to optimize processing in that stage. For instance, the VS stage may look for the InstanceID to grab additional per-vertex data for the shader or to perform other operations; the GS and PS stages may use the PrimitiveID to grab per-primitive data in the same way.

Here's an example of the IA stage showing how system values may be attached to an instanced triangle strip:



Vertex Data	C,U	D,U	E,U	F,U	G,U	H,U	I,U	J,U	K,U	L,U
VertexID	0	1	2	3	4	5	6	7	8	9
InstanceID	0	0	0	0	0	0	0	0	0	0

PrimitiveID	0	1	2
InstanceID	0	0	0

Vertex Data	C,V	D,V	E,V	F,V	G,V	H,V	I,V	J,V	K,V	L,V
VertexID	0	1	2	3	4	5	6	7	8	9
InstanceID	1	1	1	1	1	1	1	1	1	1

PrimitiveID	0	1	2
InstanceID	1	1	1

Figure 2: IA Example

This example shows two instances of geometry that share vertices. The figure at the top left shows the first instance (U) of the geometry - the first two tables show the data that the IA generates to describe instance U. The input assembler generates the VertexID, PrimitiveID, and the InstanceID to label this primitive. The data ends with the strip cut, which separates this triangle strip from the next one.

The rest of the figure pertains to the second instance (V) of geometry that shares vertices E, H, G, and I. Notice the corresponding InstanceID, VertexID and PrimitiveIDs that are generated.

VertexID

A VertexID is used by each shader stage to identify each vertex. It is a 32bit unsigned integer whose default value is 0. It is assigned to a vertex when the primitive is processed by the IA stage. Attach the VertexID semantic to the shader input declaration to inform the IA stage to attach the per-vertex VertexID semantic.

The IA will add a VertexID to each vertex for use by shader stages. For each draw call, VertexID is incremented by 1. Across indexed draw calls, the count resets back to the start value. For DrawIndexed() and DrawIndexedInstanced(), VertexID represents the index value. If VertexID overflows (exceeds 2^{32}), it wraps to 0.

For all topologies, vertices have a VertexID associated with them (regardless of adjacency).

PrimitiveID

A PrimitiveID is used by each shader stage to identify each primitive. It is a 32bit unsigned integer whose default value is 0. It is assigned to a primitive when the primitive is processed by the IA stage. Attach the PrimitiveID semantic to the shader input declaration to inform the IA stage to attach the per-primitive PrimitiveID semantic.

The IA will add a PrimitiveID to each primitive for use by the geometry shader or the pixel shader stage (whichever is the first stage active after the IA). For each indexed draw call (), PrimitiveID is incremented by 1, however, the PrimitiveID resets to 0 whenever a new instance begins. All other Draw calls do not change the value of InstanceID. If InstanceID overflows (exceeds 2^{32}), it wraps to 0.

The geometry shader stage uses a special register vPrim (to decouple the value from the other per-vertex inputs).

The pixel shader stage does not have a separate input for PrimitiveID; however, any pixel shader input that specifies PrimitiveID used a constant interpolation mode.

There is not support for automatically generating PrimitiveID for adjacent primitives. For primitive topologies with adjacency, such as a triangle strip w/adjacency, PrimitiveID is only maintained for the interior primitives in the topology (the non-adjacent primitives), just like the set of primitives in a triangle strip without adjacency.

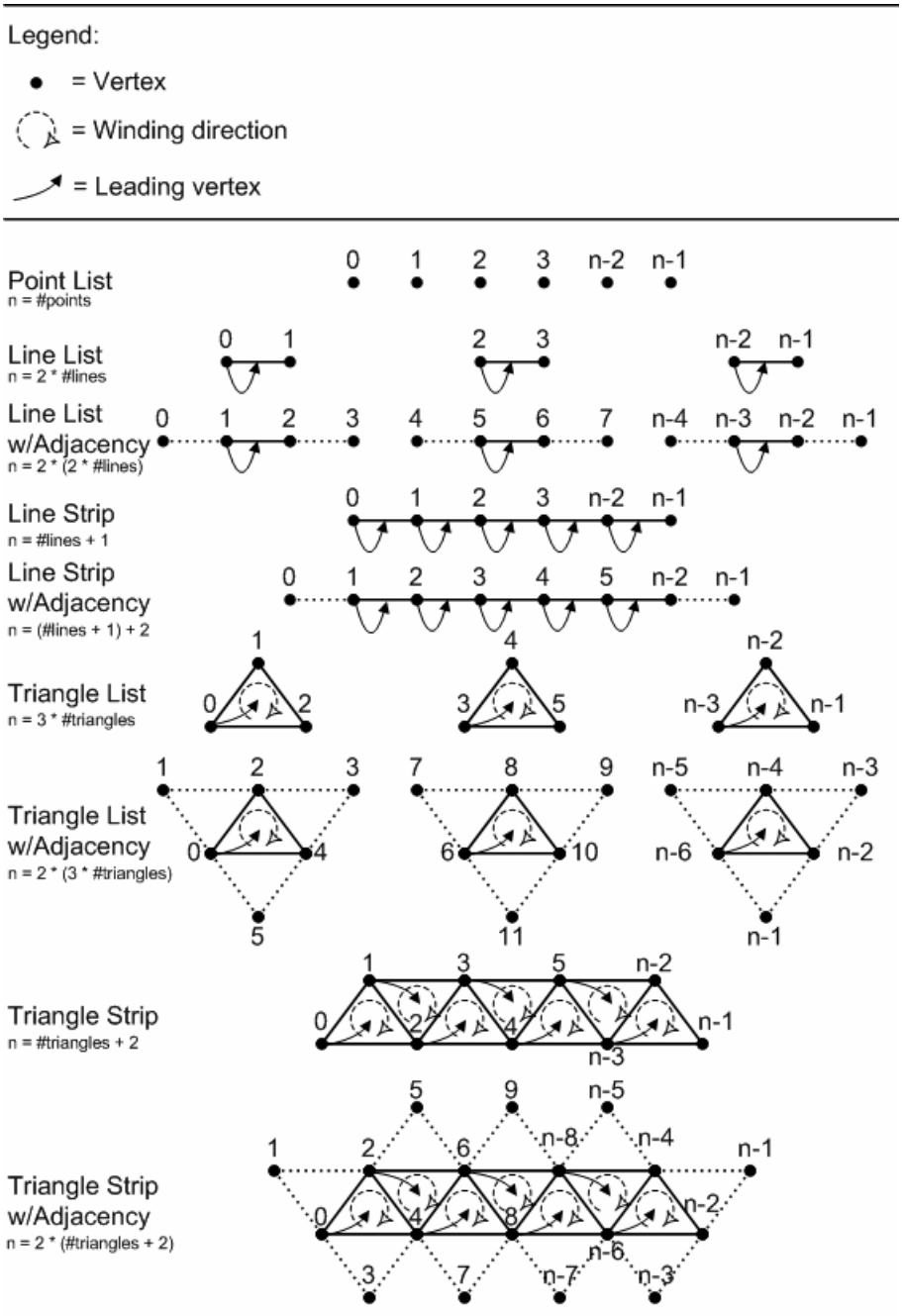
InstanceID

An InstanceID is used by each shader stage to identify the instance of the geometry that is currently being processed. It is a 32bit unsigned integer whose default value is 0.

The IA will add an InstanceID to each vertex if the vertex shader input declaration includes the InstanceID semantic. For each indexed draw call (), InstanceID is incremented by 1. All other Draw calls do not change the value of InstanceID. If InstanceID overflows (exceeds 2^{32}), it wraps to 0.

1.1.2 Primitive Toplogies

Primitive topologies describe how to data is organized into primitives. Direct3D supports the following primitive types:



The winding direction of a triangle indicates the direction in which the vertices are ordered. It can either be clockwise or counter clockwise.

A leading vertex is the first vertex in a sequence of three vertices.

1.2 Shader Stages

The Direct3D 10 pipeline contains 3 programmable-shader stages (shown as the rounded blocks in the pipeline functional diagram):

- Vertex Shader Stage
- Geometry Shader Stage
- Pixel Shader Stage

Each shader stage exposes its own unique functionality, built on the shader model 4.0 common shader core.

1.2.1 Vertex Shader Stage

The vertex shader stage processes vertices from the input assembler, performing per-vertex operations such as transformations, skinning, morphing, and per-vertex lighting. Vertex shaders always operate on a single input vertex and produce a single output vertex. The vertex shader stage must always be active for the pipeline to execute. If no vertex modification or transformation is required, a pass-through vertex shader must be created and set to the pipeline.

Each vertex shader input vertex can be comprised of up to 16 32-bit vectors (up to 4 components each) and each output vertex can be comprised of as many as 16 32-bit 4-component vectors. All vertex shaders must have a minimum of one input and one output, which can be as little as one scalar value.

The vertex shader stage can consume two system generated values from the input assembler: VertexID and InstanceID (see System Values and Semantics). Since VertexID and InstanceID are both meaningful at a vertex level, and IDs generated by hardware can only be fed into the first stage that understands them, these ID values can only be fed into the vertex shader stage.

Vertex shaders are always run on all vertices, including adjacent vertices in input primitive topologies with adjacency. The number of times that the vertex shader has been executed can be queried from the CPU using the VSInvocations pipeline statistic.

The vertex shader can perform load and texture sampling operations where screen-space derivatives are not required (using HLSL intrinsic functions samplelevel, samplecmplevelzero, samplegrad).

1.2.2 Geometry Shader Stage

The geometry shader runs application-specified shader code with vertices as input and the ability to generate vertices on output. Unlike vertex shaders, which operate on a single vertex, the geometry shader's inputs are the vertices for a full primitive (two vertices for lines, three vertices for triangles, or single vertex for point). Geometry shaders can also bring in the vertex data for the edge-adjacent primitives as input (an additional two vertices for a line, an additional three for a triangle).

The geometry shader stage can consume the SV_PrimitiveID System Value that is auto-generated by the IA. This allows per-primitive data to be fetched or computed if desired.

The geometry shader stage is capable of outputting multiple vertices forming a single selected topology (GS output topologies available are: tristrip, linestrip, and pointlist). The number of primitives emitted can vary freely within any invocation of the geometry shader, though the maximum number of vertices that could be emitted must be declared statically. Strip lengths emitted from a GS invocation can be arbitrary, and new strips can be created via the RestartStrip HLSL intrinsic function.

Geometry shader output may be fed to the rasterizer stage and/or to a vertex buffer in memory via the stream output stage. Output fed to memory is expanded to individual point/line/triangle lists (exactly as they would be passed to the rasterizer).

When a geometry shader is active, it is invoked once for every primitive passed down or generated earlier in the pipeline. Each invocation of the geometry shader sees as input the data for the invoking primitive, whether that is a single point, a single line, or a single triangle. A triangle strip from earlier in the pipeline would result in an invocation of the geometry shader for each individual triangle in the strip (as if the strip were expanded out into a triangle list). All the input data for each vertex in the individual primitive is available (i.e. 3 vertices for triangle), plus adjacent vertex data if applicable/available.

A geometry shader outputs data one vertex at a time by appending vertices to an output stream object. The topology of the streams is determined by a fixed declaration, choosing one of: PointStream, LineStream, or TriangleStream as the output for the GS stage. There are three types of stream objects available, PointStream, LineStream and TriangleStream which are all templated objects. The topology of the output is determined by their respective object type, while the format of the vertices appended to the stream is determined by the template type. Execution of a geometry shader instance is atomic from other invocations, except that data added to the streams is serial. The outputs of a given invocation of a geometry shader are independent of other invocations (though ordering is respected). A geometry shader generating triangle strips will start a new strip on every invocation.

When a geometry shader output is identified as a System Interpreted Value (e.g. SV_RenderTargetArrayIndex or SV_Position), hardware looks at this data and performs some behavior dependent on the value, in addition to being able to pass the data itself to the next shader stage for input. When such data output from the geometry shader has meaning to the hardware on a per-primitive basis (such as SV_RenderTargetArrayIndex or SV_ViewportArrayIndex), rather than on a per-vertex basis (such as SV_ClipDistance[n] or SV_Position), the per-primitive data is taken from the leading vertex emitted for the primitive.

Partially completed primitives could be generated by the geometry shader if the geometry shader ends and the primitive is incomplete. Incomplete primitives are silently discarded. This is similar to the way the IA treats partially completed primitives.

The geometry shader can perform load and texture sampling operations where screen-space derivatives are not required (samplelevel, samplecmplevelzero, samplegrad).

Algorithms that can be implemented in the geometry shader include:

- Point Sprite Expansion
- Dynamic Particle Systems
- Fur/Fin Generation
- Shadow Volume Generation
- Single Pass Render-to-Cubemap
- Per-Primitive Material Swapping
- Per-Primitive Material Setup - Including generation of barycentric coordinates as primitive data so that a pixel shader can perform custom attribute interpolation.

1.2.3 Pixel Shader Stage

A pixel shader is invoked by the rasterizer stage, to calculate a per-pixel value for each pixel in a primitive that gets rendered. The pixel shader enables rich shading techniques such as per-pixel lighting and post-processing. A pixel shader is a program that combines constant variables, texture values, interpolated per-vertex values, and other data to produce per-pixel outputs. The stage preceding the rasterizer stage (GS stage or the VS stage is the geometry shader is NULL) must output vertex positions in homogenous clip space.

A pixel shader can input up to 32 32-bit 4-component data for the current pixel location. It is only when the geometry shader is active that all 32 inputs can be fed with data from above in the pipeline. In the absence of the geometry shader, only up to 16 4-component elements of data can be input from upstream in the pipeline.

Input data available to the pixel shader includes vertex attributes that can be chosen, on a per-element basis, to be interpolated with or without perspective correction, or be treated as per-primitive constants. In addition, declarations in a pixel shader can indicate which attributes to apply centroid evaluation rules to. Centroid evaluation is relevant only when multisampling is enabled, since cases arise where the pixel center may not be covered by the primitive (though subpixel center(s) are covered, hence causing the pixel shader to run once for the pixel). Attributes declared with centroid mode must be evaluated at a location covered by the primitive, preferably at a location as close as possible to the (non-covered) pixel center.

A pixel shader can output up to 8 32-bit 4-component data for the current pixel location to be combined with the render target(s), or no color (if the pixel is discarded). A pixel shader can also output an optional 32-bit float scalar depth value for the depth test (SV_Depth).

For each primitive entering the rasterizer, the pixel shader is invoked once for each pixel covered by the primitive. When multisampling, the pixel shader is invoked once per covered pixel, though depth/stencil tests occur for each covered multisample, and multisamples that pass the tests are updated with the pixel shader output color(s).

If there is no geometry shader, the IA is capable of producing one scalar per-primitive system-generated value to the pixel shader, the SV_PrimitiveID, which can be read as input to the pixel shader. The pixel shader can also retrieve the the SV_IsFrontFace value, generated by the rasterizer stage.

One of the inputs to the pixel shader can be declared with the name SV_Position, which means it will be initialized with the pixel's float32 xyzw position. Note that w is the reciprocal of the linearly interpolated 1/w value. When the rendertarget is a multisample buffer or a standard rendertarget, the xy components of position contain pixel center coordinates (which have a fraction of 0.5f).

The pixel shader instruction set includes several instructions that produce or use derivatives of quantities with respect to screen space x and y. The most common use for derivatives is to compute level-of-detail calculations for texture sampling and in the case of anisotropic filtering, selecting samples along the axis of anisotropy. Typically, hardware implementations run a pixel shader on multiple pixels (for example a 2x2 grid) simultaneously, so that derivatives of quantities computed in the pixel shader can be reasonably approximated as deltas of the values at the same point of execution in adjacent pixels.

1.3 Common Shader Core

In Direct3D 10, all shader stages offer the same base functionality, which is implemented by the Shader Model 4.0 Common Shader Core. In additional to the base each of the three shader stages (vertex, geometry, and pixel) offer some unique functionality only to that stage, such as the ability to generate new primitives from the geometry shader stage or to discard a specific pixel in the pixel shader stage. Here is a

conceptual diagram of how data flows through a shader stage, and the relationship of the shader common core with shader memory resources:

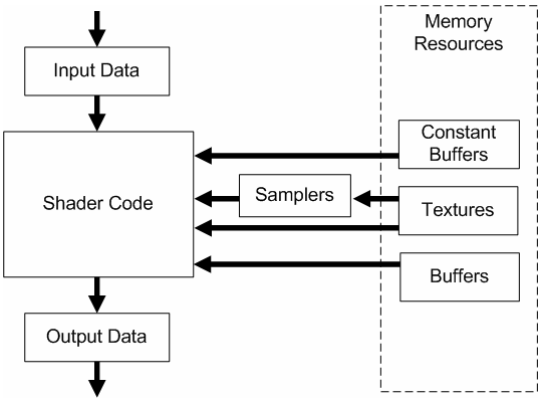


Figure 3: Shader Common Core Flow Diagram

Input Data: Every shader processes numeric inputs from the previous stage in the pipeline. The vertex shader receives its inputs from the input assembler stage; other shaders receive their inputs from the previous shader stage. Additional inputs include system-generated values, which are consumable by the first unit in the pipeline to which they are applicable.

Output Data: Shaders generate output results to be passed onto the subsequent stage in the pipeline. In the case of the Geometry Shader, the amount of data output from a single invocation can vary. Some specific outputs are system-interpreted (examples include vertex position and rendertarget array index), the rest serve as generic data to be interpreted by the application.

Shader Code: Shaders can perform vector floating point and integer arithmetic operations, flow control operations, and read from memory. There is no instruction count limit for these shaders.

Samplers: Samplers define how to sample and filter textures. As many as 16 samplers can be bound to a given shader simultaneously.

Textures: Textures can be filtered via samplers or read on a per-texel basis directly via the Load() HLSL intrinsic.

Buffers: Buffers can be read from memory on a per-element basis directly via the Load() HLSL intrinsic. They cannot be filtered. As many as 128 texture and buffer resources (combined) can be bound to a given shader simultaneously.

Constant Buffers: Constant buffers are buffer resources that are optimized for shader constant-variables. As many as 16 constant buffers can be bound to a shader stage simultaneously. They are designed for lower-latency access and more frequent update from the CPU. For this reason, additional size, layout, and access restrictions apply to constant buffers.

Differences between Direct3D 9 and Direct3D 10: In Direct3D 9, each shader unit had a single, small constant register file to store all constant shader variables. Accommodating all shaders with this limited constant space required frequent recycling of constants by the CPU. In Direct3D 10, constants are stored in immutable buffers in memory and are managed like any other

resource. There is no limit to the number of constant buffers an application can create. By organizing constants into buffers by frequency of update and usage, the amount of bandwidth required to update constants to accommodate all shaders can be significantly reduced.

1.3.1 Integer and Bitwise Support

The common shader core provides a full set of IEEE-compliant 32-bit integer and bitwise operations. These operations enable a new class of algorithms in graphics hardware - examples include compression and packing techniques, FFT's, and bitfield program-flow control.

The int and uint data types in Direct3D 10 HLSL map to 32 bit integers in hardware.

Differences between Direct3D 9 and Direct3D 10:

In Direct3D 9 stream inputs marked as integer in HLSL were interpreted as floating-point. In Direct3D 10, stream inputs marked as integer are interpreted as a 32 bit integer.

In addition, boolean values are now all bits set or all bits unset. Data converted to bool will be interpreted as TRUE if the value is not equal to 0.0f (both positive and negative zero are allowed to be FALSE) and FALSE otherwise.

1.3.2 Bitwise operators

The common shader core supports the following bitwise operators:

Operator	Function
~	Logical Not
<<	Left Shift
>>	Right Shift
&	Logical And
|	Logical Or
^	Logical Xor
&&=	Left shift Equal
>>=	Right Shift Equal
&=	And Equal
|=	Or Equal
^=	Xor Equal

Bitwise operators are defined to operate only on Int and UInt data types. Attempting to use bitwise operators on float, or struct data types will result in an error. Bitwise operators follow the same precedence as C with regard to other operators.

1.3.3 Binary Casts

Casting operation between an int and a float type will convert the numeric value following C rules for truncation of int data types. Casting a value from a float, to an int, back to a float result in a lossy conversion according to defined precision of the target.

Binary casts may also be performed using HLSL intrinsic function. These cause the compiler to reinterpret the bit representation of a number into the target data type. Here are a few examples:

```
asfloat() //Input data is aliased to float
asint()//Input data is aliased to int
asuint() //Input data is aliased to Uint
```

1.4 Shader Constant Variables

In Direct3D 10, HLSL constant variables are stored in one or more buffer resources in memory.

Shader constants can be organized into two types of buffers: constant buffers (cbuffers) and texture buffers (tbuffers). Constant buffers are optimized for shader-constant-variable usage: lower-latency access and more frequent update from the CPU. For this reason, additional size, layout, and access restrictions apply to these resources. Texture buffers utilize the texture access pipe and can have better performance for arbitrarily indexed data. Regardless of which type of resource you use, there is no limit to the number of cbuffers or tbuffers an application can create.

Declaring a cbuffer or tbuffer in HLSL looks very much like a structure declaration in C. Define a variable in a constant buffer similar to the way a struct is defined:

```
cbuffer name
{
  variable declaration;
  ...
};
```

where

- name - the constant buffer name
- variable declaration - any HLSL or effect non-object declaration (except texture and sampler)

The register modifier can be applied to a cbuffer/tbuffer namespace, which overrides HLSL auto assignment and causes a user specified binding of the named cbuffer/tbuffer to a given constant buffer/texture slot.

Differences between Direct3D 9 and Direct3D 10:

Unlike the auto-allocation of constants in Direct3D 9 which did not perform packing and instead assigned each variable to a set of float4 registers, HLSL constant variables follow packing rules in Direct3D 10.

A cbuffer or tbuffer definition is not a declaration because there is no type checking done, the definition only acts as a mechanism for naming a constant/texture buffer. It is similar to a namespace in C. All variables defined in constant and texture buffers must have unique names within their namespace. Any variable without a namespace defined, is in the global namespace (called the $Globals constant buffer, which is defined in all shaders). The register specifier can be used for assignment within a cbuffer or tbuffer, in which case the assignment relative to that cbuffer/tbuffer namespace. The register specification overrides the packing derived location.

1.4.1 Organizing Constant Variables

To make efficient use of bandwidth and maximize performance, the application can organize its constant variables into buffers by frequency of update and usage. For instance, data that needs to be updated per object, should be grouped into a different buffer than data used by all objects and materials in the scene.

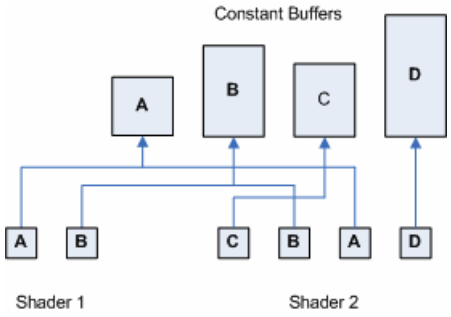


Figure 4: Binding Constant Buffers to Shaders

The first shader uses only two of the constant buffers. The second shader may use the same or different constant buffers.

As another example:

```
cbuffer myObject
{
  float4x4 matWorld;
  float3   vObjectPosition;
  int      arrayIndex;
}

cbuffer myScene
{
  float3   vSunPosition;
  float4x4 matView;
}
```

This example declares two constant buffers and organizes the data in each based on their frequency of update: data that needs to be updated on a per-object basis (like a world matrix) is grouped into myObject which could be updated for each object. This is separate from data that characterizes a scene and is therefore likely to be updated much less often (when the scene changes).

1.4.2 Constant Buffers

A shader constant buffer is a specialized buffer resource. A single constant buffer can hold up to 4096 4-channel 32-bit elements. As many as 16 constant buffers can be bound to a shader stage simultaneously.

A constant buffer is designed to minimize the overhead of setting shader constants. The Direct3D10 Effects system will manage updating of tbuffers and cbuffers for you. Alternatively, the buffer can be updated directly by the application via API calls: Map with D3D10_MAP_WRITE_DISCARD, or UpdateSubresource. The application can also copy data from another buffer (including a buffer used as a render target or stream out target) into a CB. A constant buffer is the only resource that does not use a view to bind it to the pipeline, therefore, you cannot use a view to reinterpret the data.

You can store up to 4096 4-component values in each constant buffer, and you can bind up to 16 constant buffers to a shader at the same time.

1.4.3 Texture Buffers

A texture buffer is a buffer of shader constants that is read from texture loads (as opposed to a buffer load). Texture loads can have better performance for arbitrarily indexed data than constant buffer reads.

Define a variable in a texture buffer similarly to a constant buffer:

```
tbuffer name
{
  variable declaration;
  ...
};
```

where

- name - the texture buffer name
- variable declaration - any HLSL or effect non-object declaration (except texture or sampler)

2. Resources

A resource is an area in memory that can be accessed by the Direct3D pipeline. In order for the pipeline to access memory efficiently, data that is provided to the pipeline (such as input geometry, shader resources, textures etc) must be stored in a resource. There are two types of resources from which all Direct3D resources derive: a buffer or a texture.

Each application will typically create many resources. Examples of resource include: vertex buffers, index buffer, constant buffer, textures, and shader resources. There are several options that determine how resources can be used. You can create resources that are strongly typed or typeless; you can control whether resources have both read and write access; you can make resources accessible to only the CPU, GPU, or both. Naturally, there will be speed vs. functionality tradeoff - the more functionality you allow a resource to have, the less performance you should expect.

Since an application often uses many textures, Direct3D also introduces the concept of a texture array to simplify texture management. A texture array contains one or more textures (all of the same type and dimensions) that can be indexed from within an application or by shaders. Texture arrays allow you to use a single interface with multiple indexes to access many textures. You can create as many texture arrays to manage different texture types as you need.

Once you have created the resources that your application will use, you connect or bind each resource to the pipeline stages that will use them. This is accomplished by calling a bind API, which takes a pointer to the resource. Since more than one pipeline stage may need access to the same resource, Direct3D 10 introduces the concept of a resource view. A view identifies the portion of a resource that can be accessed. You can create m views or a resource and bind them to n pipeline stages, assuming you follow binding rules for shared resource (the runtime will generate errors at compile time if you don't).

A resource view provides a general model for access to a resource (textures, buffers, etc.). Because you can use a view to tell the runtime what data to access and how to access it, resource views allow you create typeless resources. That is, you can create resources for a given size at compile time, and then declare the data type within the resource when the resource gets bound to the pipeline. Views expose many new

capabilities for using resources, such as the ability to read back depth/stencil surfaces in the shader, generating dynamic cubemaps in a single pass, and rendering simultaneously to multiple slices of a volume.

2.1 Resource Types

All resources used by the Direct3D pipeline derive from two basic resource types:

- a buffer resource
- a texture resource

A buffer resource essentially contains an array of elements (similar to an array in C++).

Textures, on the other hand, can be more complex: the data in a texture resource is made up of one or more subresources which are organized into arrays. Unlike buffers, textures can be filtered by texture samplers as they are read by shader units.

There are some restrictions for what types of resource can be bound to particular pipeline stages. This is explained in below Resource Types and Pipeline stages. Most resources can be created as untyped; however, its memory interpretation must provided when the resource is bound to the pipeline. For example, when the resource is bound as a shader resource, a view must be provided which describes how to interpret the resource.

2.1.1 Buffer Resources

A buffer resource essentially contains an array of elements (similar to an array in C++). A buffer can be visualized like this:

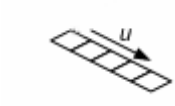


Figure 5: Buffer Resource Architecture

There are 5 elements shown in Figure 5, therefore this buffer can store up to 5 elements of data. The size of each element in a buffer is dependent on the type of data that is stored in that element, each elements has a data format defined by DXGI_FORMAT. This means you could create a buffer whose elements are all the same size or a buffer whose elements are each a different size. Furthermore, the runtime allows you to either format each element when the resource is created or create an unstructured buffer (in this case you need to know the buffer size to create).

The u vector is a 1D vector that is used to look up the elements in the array. This could have been called an index, but you will see when you look at the texture resource, how this index gets extended to two or three dimensions (where u, v, and w will be used to a texture resource).

When you create a structured buffer, you define the type (and therefore) size of each element at resource creation time. This allows the runtime to perform type checking when the resource is created. For an unstructured buffer, you must provide the format of the elements for an unstructured buffer (including element types, element offsets and an overall stride) when the unstructured buffer is bound to the graphics pipeline type. This allows the runtime to type check an unstructured buffer when the resource is bound.

When the buffer is bound to the graphics pipeline, its memory interpretation must be provided as a DXGI format. This describes types and offsets for the element(s) in the resource.

A buffer cannot be filtered, and it does not contain multiple miplevels, multiple subresources, or multisamples.

A Vertex Buffer Resource

A vertex buffer stores per-vertex data such as position, texture coordinates, normals, etc. This data will be assembled into primitives by the IA stage, and streamed into a vertex shader. To see the data that must be stored in a vertex buffer, look at the vertex shader input declaration that will use the data. For example, here is the vertex shader input declaration for the BasicHLSL sample.

```
VS_OUTPUT RenderSceneVS( float4 vPos : POSITION,
                         float3 vNormal : NORMAL,
                         float2 vTexCoord0 : TEXCOORD0,
                         uniform int nNumLights,
                         uniform bool bTexture,
                         uniform bool bAnimate )
```

This vertex shader function takes six input parameters. The first three parameters are per-vertex data which comes from the vertex buffer. The last three input parameters are uniform inputs which means that they do not vary per-vertex. Uniform inputs are generally defined once like a #define, and do not need to be included in the vertex buffer.

A vertex buffer stores data for each of the per-vertex elements. Conceptually it can be viewed like this.

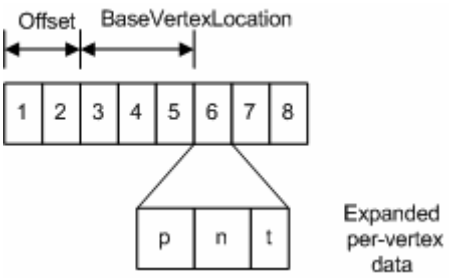


Figure 6: Vertex Buffer Data Organization

This vertex buffer has enough space allocated for eight vertices. The data for each vertex is made up of three elements (labeled p for position, n for normal, and t for texture coordinates). The three elements are shown in the expanded view. Each of the vertices (labels 1-8) in Figure 6 contains these three elements. All vertex buffers contain a series of vertices; each vertex contains one of more elements. Every vertex in a vertex buffer has an identical data structure to every other vertex.

To access data from a vertex buffer you need to know which vertex to access and these other buffer parameters:

- Offset - the number of bytes from the start of the buffer to the first vertex data. The offset needs to be supplied to the API when the buffer is bound to the pipeline with IASetVertexBuffers.
- BaseVertexLocation - the number of bytes from the offset to the first vertex data used by the appropriate Draw call.

An Index Buffer Resource

An index buffer is a buffer that contains a sequential set of indices. Each index is used to look up a vertex in a vertex buffer. Using an index buffer with one or more vertex buffers to supply data to the IA stage is called indexing.

An index buffer stores index data. It is simpler than a vertex buffer because it can only store one type of data, that is, integer indices. Conceptually an index buffer can be viewed like this.

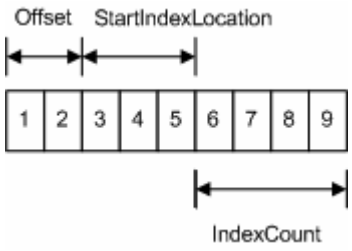


Figure 7: Index Buffer Data Organization

The sequential indices stored in an index buffer are located with the following parameters:

- Offset - the number of bytes from the start of the buffer to the first vertex data. The offset needs to be supplied to the API when the buffer is bound to the pipeline with IASetIndexBuffer.
- StartIndexLocation - the number of bytes from the offset to the first vertex data used by the appropriate Draw call (see Executing the IA Stage).
- IndexCount - the number of indices to render.

An index buffer contains 16-bit or 32-bit indices.

A buffer created with the D3D10_BIND_CONSTANT_BUFFER flag can be used to store shader constant data. Constant buffers must be created with type DXGI_FORMAT_R32G32B32A32_TYPELESS. The size of a constant buffer is restricted to hold a maximum of 4096 elements.

A buffer created with the D3D10_BIND_SHADER_RESOURCE flag can be used as a shader resource input, accessed in the shader via the Load() method. The buffer must be bound at one of the available 128 slots for input resources, by first creating the appropriate shader resource view.

A buffer created with the D3D10_BIND_SHADER_RESOURCE flag can be used to store the results of the Stream Output stage. There are two types of bindings available for stream output buffers, one that treats a single output buffer as a multiple-element buffer (array-of-structures), while the other permits multiple output buffers each treated as single-element buffers (structure-of-arrays). If the resource also has the D3D10_BIND_VERTEX_BUFFER flag specified, the resource may also be used with DrawAuto.

2.1.2 Texture Resources

A texture resource is a structured collection of data designed to store texture data. The data in a texture resource is made up of one or more subresources which are organized into arrays and mipchains. Unlike buffers, textures can be filtered by texture samplers as they are read by shader units. The type of resource impacts how the texture is filtered - for example, TextureCubes are filtered across edges.

- Texture1D and Texture1DArray Resource

- Texture2D and Texture2DArray Resource
- Texture3D Resource

Texture1D and Texture1DArray Resource

Like the buffer resource, a Texture1D contains a 1D array of elements. Unlike the buffer resource, a texture1D resource is filtered and may contain one or more mip levels. The 1D texture resource looks like this:

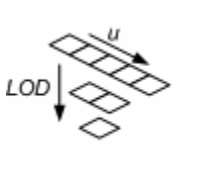


Figure 8: Texture1D Resource Architecture

Figure 8 illustrates a Texture1D resource with 3 mip levels. The top-most mip level is the largest level; each successive level is a power of 2 (on each side) smaller. In this example, since the top-level texture width is 5 elements, there are two mip levels before the texture width is reduced to 1. Each element in each mip level is addressable by the u vector (which is commonly called a texture coordinate).

Each element in each mip level contains a single texel, or texture value. The data type of each element is defined by the texel format which is once again a DXGI_FORMAT value. A texture resource may be typed or typeless at resource-creation time, but when bound to the pipeline, its interpretation must be provided in a view.

A Texture1DArray resource is a homogenous array of 1D textures. It looks like an array of 1D textures.

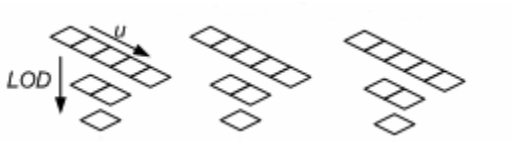


Figure 9: Texture1DArray Resource Architecture

This texture array contains three textures. Each of the three textures has a texture width of 5 (which is the number of elements in the 1st layer). Each texture also contains a 3 layer mip-map.

All texture arrays in Direct3D are a homogenous array of textures; this means that every texture in a texture array must have the same data format and size (including texture width and number of miplevels). You may create texture arrays of different sizes, as long as all the textures in each array match in size.

Subresources

One interesting feature with Direct3D texture resources (including textures and texture arrays) is that they are made up of subresources. A subresource is defined as single mip level and texture index. In other words, for a single texture, a subresource is a single mip level. For instance, here is a valid subresource for a 1D texture.

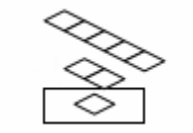


Figure 10: Texture1D Subresource

A subresource is a texture and mip-level combination. This 1D texture is made up of 3 subresources. For a texture array, a subresource can be extended to an array of single mip levels. Here is an example of subresources, within a 2DTextureArray.

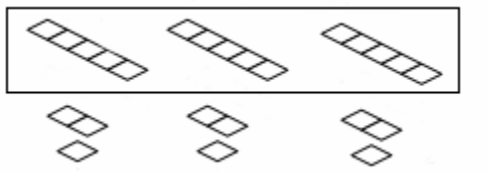


Figure 11: Texture1DArray Subresource

This subresource is an array of the top mip level of all three textures in the texture array resource. You could have specified any mip level in the subresource. Direct3D uses a resource view to access this array of texture subresources in a texture array.

The pipeline uses several objects derived from a Texture1D resource to read data from or write data to. This table shows the types of objects that can be created from texture resources, and where they can be bound to the pipeline:

Indexing Subresources

A texture array can contain multiple textures, each with multiple mip-levels. Each subresource is a single texture mip-level. When accessing a particular subresource, this is how the subresources are iondexed within a texture array, with multiple mip levels.

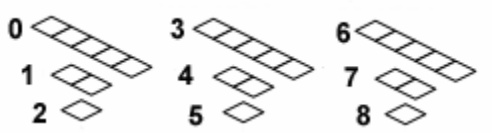


Figure 12: Indexing Subrsources in a Texture1DArray

The index starts at zero in the first texture in the array, and increments through the mip levels for that texture. Simply increment the index to go to the next texture.

Texture2D and Texture2DArray Resource

A Texture2D resource contains a 2D grid of texels. Each texel is addressable by a u, v vector. Since it is a texture resource, it may contain mip levels, and subresources. A fully populated 2D texture resource looks like this:

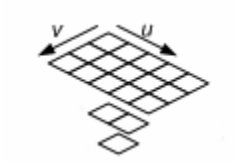


Figure 13: Texture2D Resource Architecture

This texture resource contains a single 3x5 texture with two mip levels.

A Texture2D resource is a homogeneous array of 2D textures; that is, each texture has the same data format and dimensions (including mip levels). It has a similar layout as the Texture1D resource except that the textures now contain 2D data, and therefore looks like this:

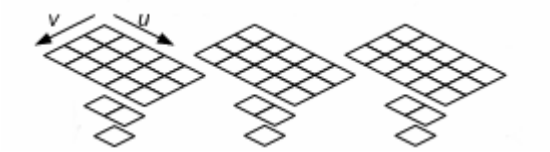


Figure 14: Texture2DArray Resource Architecture

This texture array contains three textures; each texture is 3x5 with two mip levels.

Texture2DArray Resource as a Texture Cube

A texture cube is a Texture2DArray that contains 6 textures, one for each face of the cube. A fully populated texture cube looks like this:

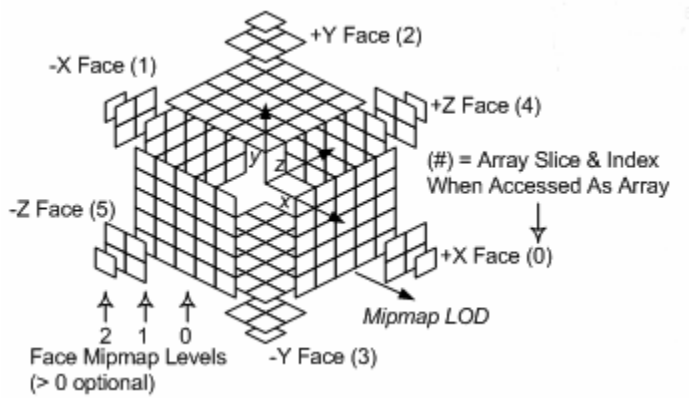


Figure 15: TextureCube Resource Architecture

A Texture2DArray that contains 6 textures may be read from within shaders with the cube map sample instructions, after they are bound to the pipeline with a cube-texture view. Texture cubes are addressed from the shader with a 3D vector pointing out from the center of the texture cube.

Texture3D Resource

A Texture3D resource (also known as a volume texture) contains a 3D volume of texels. Since it is a texture resource, it may contain mip levels. A fully populated Texture3D resource looks like this:

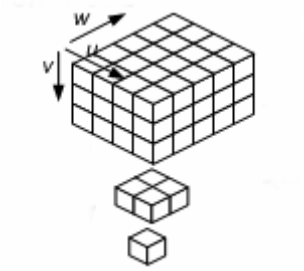


Figure 16: Texture3D Resource Architecture

When a Texture3D mip-slice is bound as a rendertarget output, (by creating a rendertarget view), the Texture3D behaves identically to a Texture2DArray with n array slices where n is the depth (3rd dimension) of the Texture3D. The particular slice in the Texture3D to render is chosen from the geometry shader stage, by declaring a scalar component of output data as the SV_RenderTargetArrayIndex system-value.

There is no concept of an array of Texture3D resources; therefore a Texture3D subresource is a single mip level.

2.1.3 Resource Types and Pipeline stages

Resources must be bound to pipeline stages before they are used. There are several pipeline stages that read or write to resources. This table identifies the different resources that can be bound to each pipeline stage.

Under certain conditions, you may use resource views to bind a resource to more than one pipeline stage. A view allows you to read (or write) a portion of a resource. In general, you can read a resource by as many stages as necessary using different views. However, you may only bind a resource for writing by one stage at a time during any Draw call.

Input Assembler Resource Types

Pipeline Stage	In/Out	Resource Object	Resource Type	Bind API
Input Assembler	In	Vertex buffer	Buffer	IASetVertexBuffers
Input Assembler	In	Index buffer	Buffer	IASetIndexBuffer

Shader Resource Types

Pipeline Stage	In/Out	Resource Object	Resource Type	Bind API
Vertex Shader	In	Shader Resource Input	Texture1D, Texture2D, Texture3D, TextureCube	VSSetShaderResources, GSSetShaderResources, PSSetShaderResources
Vertex Shader	In	Shader Constant Buffer Input	Buffer, Texture1D, Texture2D, Texture3D, TextureCube	VSSetConstantBuffers, GSSetConstantBuffers, PSSetConstantBuffers
Geometry Shader	In	Shader Resource Input	Texture1D, Texture2D, Texture3D, TextureCube	VSSetShaderResources, GSSetShaderResources,

				PSSetShaderResources
Geometry Shader	In	Shader Constant Buffer Input	Buffer, Texture1D, Texture2D, Texture3D, TextureCube	VSSetConstantBuffers, GSSetConstantBuffers, PSSetConstantBuffers
Pixel Shader	In	Shader Resource Input	Texture1D, Texture2D, Texture3D, TextureCube	VSSetShaderResources, GSSetShaderResources, PSSetShaderResources
Pixel Shader	In	Shader Constant Buffer Input	Buffer, Texture1D, Texture2D, Texture3D, TextureCube	VSSetConstantBuffers, GSSetConstantBuffers, PSSetConstantBuffers

Stream Output Resource Types

Pipeline Stage	In/Out	Resource Object	Resource Type	Bind API
Stream Output	Out	Stream Output	Buffer	SOSetTargets

Output Merger Resource Types

Pipeline Stage	In/Out	Resource Object	Resource Type	Bind API
Output Merger	Out	Rendertarget Output	Buffer, Texture1D, Texture2D, Texture3D, TextureCube	OMSetRenderTargets
Output Merger	Out	Depth/Stencil Output	Texture1D, Texture2D, TextureCube	OMSetDepthStencilState

2.2 Resource Creation Flags

Resource creation flags specify how a resource is to be used, where the resource is allowed to bind, and which upload and download methods are available. The flags are broken up into these categories:

- Resource Creation Usage Flags
- Resource Creation Binding Flags
- Resource Creation CPU Access Flags
- Resource Creation Miscellaneous Flags
- Resource Flag Combinations

2.2.1 Resource Creation Usage Flags

A resource usage flag specifies how often your resource will be changing. This update frequency depends on how often a resource is expected to change relative to each rendered frame. For instance:

- Never - the contents never change. Once the resource is created, it cannot be changed.
- Infrequently - less than once per frame
- Frequently - once or more per frame. Frequently describes a resource whose contents are expected to change so frequently that the upload of resource data (by the CPU) is expected to be a bottleneck.
- Staging - this is a special case for a resource that can copy its contents to another resource (or vice versa).

Usage	Update Frequency	Limitations

D3D10_USAGE_DEFAULT	Infrequently (less than once per frame)	This is the most likely usage setting. • Mapping: not allowed, the resource can only be changed with UpdateSubresource. • Binding flag: Any, none
D3D10_USAGE_DYNAMIC	Frequently	A dynamic resource is limited to one that contains a single subresource. • Mapping: Use Map(CPU writes directly to the resource) with the D3D10_CPU_ACCESS_WRITE flag. Optionally use D3D10_MAP_WRITE_DISCARD or D3D10_MAP_WRITE_NO_OVERWRITE. • Binding flag: at least one GPU input flag, GPU output flags are not allowed. You must use D3D10_MAP_WRITE_DISCARD and D3D10_MAP_WRITE_NO_OVERWRITE if the resource is a vertex or index buffer (uses D3D10_BIND_VERTEX_BUFFER or D3D10_BIND_INDEX_BUFFER).
D3D10_USAGE_IMMUTABLE	Never	• Mapping: not allowed • Binding flag: at least one GPU input flag, GPU output flags are not allowed
D3D10_USAGE_STAGING	n/a	This resource cannot be bound to the pipeline directly; you may copy the contents of a resource to (or from) another resource that can be bound to the pipeline. Use this to download data from the GPU. • Mapping: Use CopyResource or CopySubresource with either/both D3D10_CPU_ACCESS_WRITE and D3D10_CPU_ACCESS_READ to copy the contents of this resource to any resource with one of the other usage flags (which allows them to be bound to the pipeline). • Binding flag: None are valid

2.2.2 Resource Creation Binding Flags

Resources are bound to a pipeline stage through an API call. Resources may be bound at more than one location in the pipeline (even simultaneously within certain restrictions) as long as each resource satisfies any restrictions that pipeline stage has for resource properties (memory structure, usage flags, binding flags, cpu access flags).

It is possible to bind a resource as an input and an output simultaneously, as long as the input view and the output view do not share the same subresources.

Binding a resource is a design choice affecting how that resource needs to interact with the GPU. If you can, try to design resources that will be reused for the same purpose; this will most likely result in better performance.

For example, if a render target is to be used as a texture, consider how it will be updated. Suppose multiple primitives will be rendered to the rendertarget and then the rendertarget will be used as a texture. For this scenario, it may be faster having two resources: the first would be a rendertarget to render to, the second would be a shader resource to supply the texture to the shader. Each resource would specify a single binding flag (the first would use D3D10_BIND_RENDER_TARGET and the second would use D3D10_BIND_SHADER_RESOURCE). If there is some reason that you cannot have two resources, you can specify both bind flags for a single resource and you will need to accept the performance trade-off. Of course, the only way to actually understand the performance implication is to measure it.

The bind flags are broken into two groups, those that allow a resource to be bound to GPU inputs and those that allow a resource to be bound to GPU outputs.

GPU Input Flags

Flag	Resource Type	API Call
D3D10_BIND_VERTEX_BUFFER	unstructured buffer	IASetVertexBuffers
D3D10_BIND_INDEX_BUFFER	unstructured buffer	IASetIndexBuffer
D3D10_BIND_CONSTANT_BUFFER	unstructured buffer	VSSetConstantBuffers, GSSetConstantBuffers, PSSetConstantBuffers restricts the buffer width (in bytes) to be less than or equal to 4096 * sizeof(R32G32B32A32). The resource must also be a multiple of sizeof(R32G32B32A32). Use UpdateSubresource to update the entire buffer; D3D10_MAP_WRITE_NO_OVERWRITE is not allowed.
D3D10_SHADER_RESOURCE	shader resource (vertex buffer, index buffer, texture)	VSSetShaderResources, or GSSetShaderResources, or PSSetShaderResources; this resource may not use D3D10_MAP_WRITE_NO_OVERWRITE.

GPU Output Flags

Flag	Resource Type	API Call
D3D10_BIND_STREAM_OUTPUT	unstructured buffer	SOSetTargets
D3D10_BIND_RENDER_TARGET	any resource (or subresource) with a rendertarget view	OMSetRenderTargets using a render target
D3D10_BIND_DEPTH_STENCIL	a Texture1D, Texture2D, or TectureCube resource (or subresource) with a depth stencil view	OMSetRenderTargets using a depth-stencil parameter

Resources bound to an output stage may not be:

- Mapped with ID3D10Buffer::Map or ID3D10TextureXXX::Map
- Loaded with UpdateSubresource
- Copied with CopyResource

2.2.3 Resource Creation CPU Access Flags

These flags allow the CPU to read or write (or both) a resource. Reading or writing a resource requires that the resource be mapped (which is analogous to Lock in Direct3D 9) so that the resource cannot be simultaneously be read and written to.

Flag	Limitations
D3D10_MAP_READ	• If the writeable resource does not have the D3D10_MAP_WRITE bind flag also, then the application can only write to the memory address retrieved from Map.
D3D10_MAP_WRITE	• If the writeable resource does not have the D3D10_MAP_READ bind flag also, then the application can only write to the memory address retrieved from Map.

Any resource that has either the read or write flag specified:

- Cannot be updated with UpdateSubresource.
- May not use any GPU output bind flag.
- Must use either the DYNAMIC or STAGING usage flag.

2.2.4 Resource Creation Miscellaneous Flags

Flag	Limitations
D3D10_RESOURCE_MISC_MIPGEN	Use this flag to allow a shader resource view to use GenerateMips to generate mip maps; note that this requires that the shade resource view also is capable of being bound as a rendertarget (D3D10_BIND_RENDERTARGET). You may not generate mip maps for a buffer resource.

Every resource can be used with CopySubresourceRegion, and CopyResource (as a source). However, the primary advantage of not specifying this flag (when it could be used), is related to STAGING Resources and their interaction with DEVICEREMOVED. After DEVICEREMOVED, Map on STAGING Resources will fail with DEVICEREMOVED when the application specified COPY_DESTINATION. However, if the application will not use the STAGING Resource as a destination for Copy commands, it can continue to Map such Resources after DEVICEREMOVED. See the following table to determine which Resources can set this flag.

2.2.5 Resource Flag Combinations

Resource Type and Usage	GPU Input Bind	GPU Output Bind	Map(READ and/ or WRITE)	Map(WRITE_DISCARD or WRITE_NOOVERWRITE)	Update Subresource	Copy Dest
IMMUTABLE	R					
DEFAULT (GPU Input)	C				E	C
DEFAULT (GPU Output)	C	R				
DYNAMIC	R			D		C
STAGING			R			C

- R = Requires at least one bit set.
- C = Compatible to use.
- D = Compatible, but WRITE_NOOVERWRITE may only be used if Bind flags are restricted to VERTEX_BUFFER and INDEX_BUFFER.
- E = Compatible, but CONSTANT_BUFFER prevents partial CPU updates.
- empty = Cannot be used together.

2.3 Resource Access and Views

In Direct3D 10, resources are accessed with a view, which is a mechanism for hardware interpretation of a resource in memory. A view allows a particular pipeline stage to access only the subresources it needs, in the representation desired by the application.

A view supports the notion of a typeless resource - that is, you can create a resource that is of certain size but whose type is interpreted as a uint, float, unorm, etc. only when it is bound to a pipeline stage. This makes a resource re-interpretable by different pipeline stages.

Here is an example of binding a Texture2DArray resource with 6 textures two different ways through two different views. (Note: a subresource cannot be bound as both input and output to the pipeline simultaneously.)

The Texture2DArray can be used as a shader resource by creating a shader resource view for it. The resource is then addressed as an array of textures.

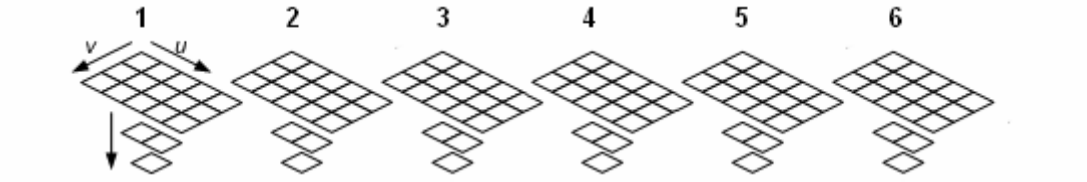


Figure 17: Texture2DArray Viewed as 6 2D Textures

Create this view object by calling CreateShaderResourceView. Then set the view object to the pipeline stage (the particular shader) by calling SetShaderResources (VSSetShaderResources, PSSetShaderResources, GSSetShaderResources). Use an HLSL texture instrinsic function to sample the texture.

The Texture2DArray can also be used in a shader as a cube map with a cube-map view. The resource is then addressed as a cube-mapped texture that is filtered correctly across edges and corners by the sampler.

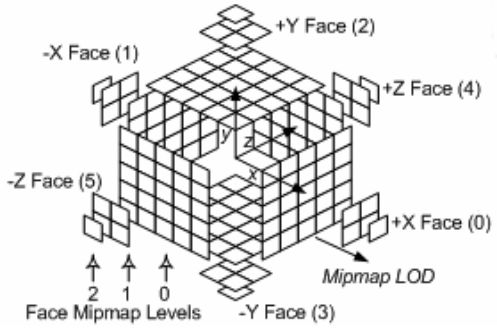


Figure 18: Texture2DArray Resource - Viewed as a texture cube for sampling

Using a Texture2DArray as a render target. The resource can be viewed as an array of 2D textures (6 in this case) with mip levels (3 in this case).

Create a view object for a rendertarget by calling calling CreateRenderTargetView. Then call OMSetRenderTargets to set the rendertarget view to the pipeline. Render into the rendertargets by calling Draw and using the RenderTargetArrayIndex to index into the proper texture in the array. You can use a subresource (a mip level, array index combination) to bind to any array of subresources. So you could bind to the second mip level and only update this particular mip level if you wanted like this:

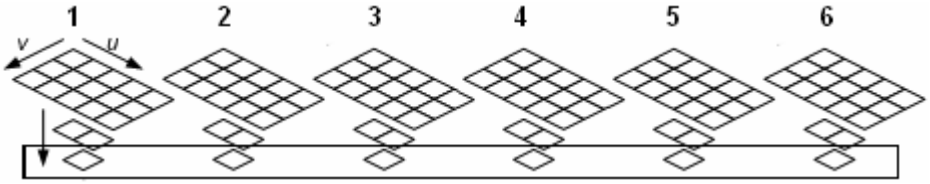


Figure 19: Views can access an array of Subresources

2.3.1 Render Target Views

Creating a rendertarget view for a resource will enable the user to bind the resource to the pipeline as a rendertarget. A rendertarget is a resource that will be written to by the output merger stage at the end of a render pass; after a render pass the rendertargets contain the color information of a rendered image. When creating a rendertarget view, only one mip level of a texture resource may be used as a rendertarget. When binding a rendertarget view to the pipeline, there will be a corresponding depth stencil view that will be used with the rendertarget in the output merger stage.

2.3.2 Depth Stencil Views

Creating a depth stencil view for a resource will enable the user to use the resource as depth stencil. A depth stencil view is bound to the output merger stage along with a corresponding rendertarget view.

2.3.3 Shader Resource Views

A shader resource view enables the user to bind a resource to a shader stage. A shader resource view can interpret a resource as something different from when it was created. For example, a shader resource view may interpret a Texture2DArray with six textures as a cube map. By binding this cube map view to a shader stage, the shader will sample the Texture2DArray as a cube map.

Differences between Direct3D 9 and Direct3D 10:

In Direct3D 10, you no longer bind a resource directly to the pipeline, you create a view of a resource, and then set the view to the pipeline. This allows validation and mapping in the runtime and driver to occur at view creation, minimizing type checking at bind-time.

2.4 New Resource Formats

Direct3D 10 offers new data compression formats for compressing high-dynamic range (HDR) lighting data, normal maps and heightfields to a fraction of their original size. These compression types include:

- Shared-Exponent high-dynamic range (HDR) format (RGBE)
- New Block-Compressed 1-2 channel UNORM/SNORM formats

The block compression formats can be used for any of the 2D or 3D texture types (Texture2D, Texture2DArray, Texture3D, or TextureCube) including mip-map surfaces. The block compression techniques require texture dimensions to be a multiple of 4 (since the implementation compresses on blocks of 4x4 texels). In the texture sampler, compressed formats are always decompressed before texture filtering.

3. Samples

3.1 CubeMapGS Sample

The CubeMapGS sample demonstrates rendering a cubic texture render target with a single DrawIndexed() call using two new features in Direct3D 10: render target array and geometry shader. A render target array allows multiple render target and depth stencil textures to be active simultaneously. By using an array of six render targets, one for each face of the cube texture, all six faces of the cube can be rendered together. When the geometry shader emits a triangle, it can control which render target in the array the triangle gets rasterized on. For every triangle that gets passed to the geometry shader, six triangles are generated in the shader and output to the pixel shader, one triangle for each render target.

3.1.1 Sample Overview

Environment mapping is a popular and well-supported technique in 3D graphics. Traditionally, dynamic cubic environment maps are created by obtaining a surface for each face of a cube texture and setting that surface as the render target to render the scene once for each face of the cube. The cube texture can then be used to render environment-mapped objects. This method, while it works, increases the number of rendering passes from one to seven, greatly reducing the application frame rate. In Direct3D 10, applications can use geometry shaders and render target arrays to alleviate this problem.

3.1.2 How the Sample Works

A geometry shader in Direct3D 10 is a piece of code that runs for each primitive to be rendered. In each invocation, the geometry shader can output zero or more primitives to be rasterized and processed in the pixel shader. For each output primitive, the geometry shader can also control to which element slice of the render target array the primitive gets emitted.

The render target array in Direct3D 10 is a feature that enables an application to render onto multiple render targets simultaneously at the primitive level. The application uses a geometry shader to output a primitive and select which render target in the array should receive the primitive. This sample uses a render target array of 6 elements for the 6 faces of the cube texture. The following code fragment creates the render target view used for rendering to the cube texture.

```
// Create the 6-face render target view
D3D10_RENDER_TARGET_VIEW_DESC DescRT;
DescRT.Format = dstex.Format;
DescRT.ResourceType = D3D10_RESOURCE_TEXTURECUBE;
DescRT.TextureCube.FirstArraySlice = 0;
DescRT.TextureCube.ArraySize = 6;
DescRT.TextureCube.MipSlice = 0;
m_pD3D10Device->CreateRenderTargetView(m_pEnvMap, &DescRT, &m_pEnvMapRTV);
```

By setting DescRT.TextureCube.FirstArraySlice to 0 and DescRT.TextureCube.ArraySize to 6, this render target view represents an array of 6 render targets, one for each face of the cube texture. When the sample renders onto the cube map, it sets this render target view as the active view by calling ID3D10Device::OMSetRenderTargets() so that all 6 faces of the texture can be rendered at the same time.

```
// Set the env map render target and depth stencil buffer
ID3D10RenderTargetView* aRTViews[ 1 ] = { m_pEnvMapRTV };
m_pD3D10Device->OMSetRenderTargets(sizeof(aRTViews) / sizeof(aRTViews[0]), aRTViews,
m_pEnvMapDSV);
```

3.1.3 Rendering the CubeMap

At the top level, rendering begins in Render(), which calls RenderSceneIntoCubeMap() and RenderScene(), in that order. RenderScene() takes a view matrix a projection matrix, then renders the scene onto the current render target. RenderSceneIntoCubeMap() handles rendering of the scene onto a cube texture. This texture is then used in RenderScene() to render the environment-mapped object.

In RenderSceneIntoCubeMap(), the first necessary task is to compute the 6 view matrices for rendering to the 6 faces of the cube texture. The matrices have the eye point at the camera position and the viewing directions along the -X, +X, -Y, +Y, -Z, and +Z directions. A boolean flag, m_bUseRenderTargetArray, indicates the technique to use for rendering the cube map. If false, a for loop iterates through the faces of the cube map and renders the scene by calling RenderScene() once for each cube map face. No geometry shader is used for rendering. This technique is essentially the legacy method used in Direct3D 9 and prior. If m_bUseRenderTargetArray is true, the cube map is rendered with the RenderCubeMap effect technique. This technique uses a geometry shader to output each primitive to all 6 render targets. Therefore, only one call to RenderScene() is required to draw all 6 faces of the cube map.

The vertex shader that is used for rendering onto the cube texture is VS_CubeMap, as shown below. This shader does minimal work of transforming vertex positions from object space to world space. The world space position will be needed in the geometry shader.

```
struct VS_OUTPUT_CUBEMAP {
    float4 Pos : SV_POSITION;    // World position
    float2 Tex : TEXCOORD0;      // Texture coord
};
```

```
VS_OUTPUT_CUBEMAP VS_CubeMap(float4 Pos : POSITION, float3 Normal : NORMAL, float2 Tex :
TEXCOORD) {
    VS_OUTPUT_CUBEMAP o = (VS_OUTPUT_CUBEMAP)0.0f;

    // Compute world position
    o.Pos = mul(Pos, mWorld);

    // Propagate tex coord
    o.Tex = Tex;

    return o;
}
```

One of the two geometry shaders in this sample, GS_CubeMap, is shown below. This shader is run once per primitive that VS_CubeMap has processed. The vertex format that the geometry shader outputs is GS_OUTPUT_CUBEMAP. The RTIndex field of this struct has a special semantic: SV_RenderTargetArrayIndex. This semantic enables the field RTIndex to control the render target to which the primitive is emitted. Note that only the leading vertex of a primitive can specify a render target array index. For all other vertices of the same primitive, RTIndex is ignored and the value from the leading vertex is used. As an example, if the geometry shader constructs and emits 3 vertices with RTIndex equal to 2, then this primitive goes to element 2 in the render target array.

At the top level, the shader consists of a for loop that loops 6 times, once for each cube face. Inside the loop, another loop runs 3 times per cube map to construct and emit three vertices for the triangle primitive. The RTIndex field is set to f, the outer loop control variable. This ensures that in each iteration of the outer loop, the primitive is emitted to a distinct render target in the array. Another task that must be done before emitting a vertex is to compute the Pos field of the output vertex struct. The semantic of Pos is SV_POSITION, which represents the projected coordinates of the vertex that the rasterizer needs to properly rasterize the triangle. Because the vertex shader outputs position in world space, the geometry shader needs to transform that by the view and projection matrices. In the loop, the view matrix used to transform the vertices is g_mViewCM[f]. This matrix array is filled by the sample and contains the view matrices for rendering the 6 cube map faces from the environment-mapped object's perspective. Thus, each iteration uses a different view transformation matrix and emits vertices to a different render target. This renders one triangle onto 6 render target textures in a single pass, without calling DrawIndexed() multiple times.

```
struct GS_OUTPUT_CUBEMAP {
    float4 Pos : SV_POSITION;     // Projection coord
    float2 Tex : TEXCOORD0;       // Texture coord
    uint RTIndex : SV_RenderTargetArrayIndex;
};

[maxvertexcount(18)]
void GS_CubeMap(triangle VS_OUTPUT_CUBEMAP In[3], inout TriangleStream<GS_OUTPUT_CUBEMAP>
CubeMapStream) {
    for ( int f = 0; f < 6; ++f ) {
        // Compute screen coordinates
        GS_OUTPUT_CUBEMAP Out;
        Out.RTIndex = f;
        for (int v = 0; v < 3; v++) {
            Out.Pos = mul(In[v].Pos, g_mViewCM[f]);
            Out.Pos = mul(Out.Pos, mProj);
            Out.Tex = In[v].Tex;
            CubeMapStream.Append(Out);
        }
        CubeMapStream.RestartStrip();
    }
}
```

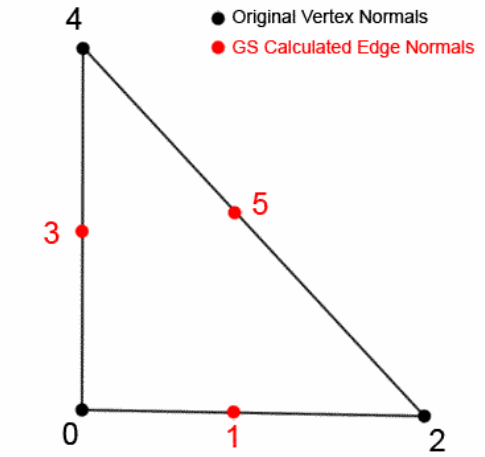
The pixel shader, PS_CubeMap, is rather straight-forward. It fetches the diffuse texture and applies it to the mesh. Because the lighting is baked into this texture, no lighting is performed in the pixel shader.

3.1.4 Rendering the Reflective Object

Three techniques are used to render the reflective object in the center of the scene. All three fetch texels from the cubemap just as they would from a cubemap that wasn't rendered in a single pass. In short, this technique is orthogonal to the way in which the resulting cubemap is used. The three techniques differ mainly in how they use the cubemap texels. RenderEnvMappedScene uses an approximated fresnel reflection function to blend the colors of the car paint with reflection from the cubemap. RenderEnvMappedScene_NoTexture does the same, but without the paint material. RenderEnvMappedGlass adds transparency to RenderEnvMappedScene_NoTexture.

3.1.5 Higher Order Normal Interpolation

Traditional normal interpolation has been linear. This means that the normals calculated in the vertex shader are linearly interpolated across the face of the triangle. This causes the reflections in the car to appear to slide when the direction of the normal changes rapidly across the face of a polygon. To mitigate this, this sample uses a quadratic normal interpolation. In this case, 3 extra normals are calculated in the geometry shader. These normals are placed in the center of each triangle edge and are the average of the two normals at the vertices that make up the edge. In addition, the geometry shader calculates a barycentric coordinate for each vertex that is passed down to the pixel shader and used to interpolate between the six normals.

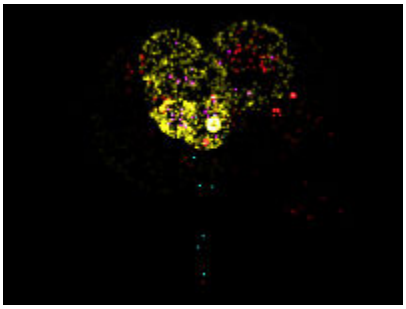


In the pixel shader, these 6 normals weighted by six basis functions and added together to create the normal for that particular pixel. The basis functions are as follows.

```
2x^2 + 2y^2 + 4xy -3x -3y + 1
-4x^2 -4xy + 4x
2x^2 - x
-4y^2 -4xy + 4y
2y^2 - y
4xy
```

3.2 ParticlesGS Sample

This sample implements a complete particle system on the GPU using the Direct3D 10 Geometry Shader, Stream Output, and DrawAuto.



3.2.1 How the Sample Works

Particle system computation has traditionally been performed on the CPU with the GPU rendering the particles as point sprites for visualization. With Geometry Shaders, the ability to output arbitrary amounts of data to a stream allows the GPU to create new geometry and to store the result of computations on existing geometry.

This sample uses the stream out capabilities of the geometry shader to store the results of particles calculations into a buffer. Additionally, the Geometry Shader controls particle birth and death by either outputting new geometry to the stream or by avoiding writing existing geometry to the stream. A Geometry Shader that streams output to a buffer must be constructed differently from a normal Geometry Shader.

When used inside an FX file

```
//--------------------------------------------------------------------------------------
// Construct StreamOut Geometry Shader
//--------------------------------------------------------------------------------------
geometryshader gsStreamOut = ConstructGSWithSO(compile gs_4_0 GSAdvanceParticlesMain(),
                                               "POSITION.xyz;
                                               NORMAL.xyz;
                                               TIMER.x;
                                               TYPE.x" );
```

When used without FX

```
//--------------------------------------------------------------------------------------
// Construct StreamOut Geometry Shader
//--------------------------------------------------------------------------------------
D3D10_STREAM_OUTPUT_DECLARATION_ENTRY pDecl[] =
{
          // semantic name, semantic index, start component, component count, output slot
          { L"POSITION", 0, 0, 3, 0 }, // output first 3 components of "POSITION"
          { L"NORMAL", 0, 0, 3, 0 }, // output the first 3 components of "NORMAL"
          { L"TIMER", 0, 0, 1, 0 }, //  output the first component of "TIMER"
          { L"TYPE", 0, 0, 1, 0 }, //  output the first component of "TYPE"
};

CreateGeometryShaderWithStreamOut( pShaderData, pDecl, 4, sizeof(PARTICLE_VERTEX),
&pGS );
```

3.2.2 Particle Types

This particle system is composed of 5 different particle types with varying properties. Each particle type has its own velocity and behavior and may or may not emit other particles.

Launcher Particles

Launcher particles do not move and do not die. They simply count down until they can emit a Shell particle. Once they have emitted a Shell particle, they reset their timer.

Shell Particles

Shell particles are single particles that are given random velocities and launched into the air by Launcher particles. They are meant to represent fireworks shells before they explode. When a Shell particle reaches the end of it's lifespan it does not re-emit itself into the system. Instead, it emits several Ember1 and Ember2 type particles.

Ember1 Particles

Ember1 particles are emitted from Shell particles when they explode. Ember1 particles have short lifespans and fade out as their timer counts down. When their timer reaches zero, these particles do not re-emit themselves into the system and effectively "die."

Ember2 Particles

Ember2 particles are also emitted from Shell particles when they explode. Unlike Ember1 particles, when Ember2 particles reach the end of their lifespans they emit Ember3 particles. These are the source of the secondary explosions in the system.

Ember3 Particles

Ember3 particles are similar to Ember1 particles except that they are of a different color and have a shorter lifespan.

3.2.3 Handling Particles in the Geometry Shader

Particles are handled entirely on the GPU by the Geometry Shader. Instead of going to the rasterizer, the vertex data passed into the geometry shader is output to another vertex buffer. After an initial seeding of the vertex buffer with LAUNCHER type particles, the system can sustain itself on the GPU with only per-frame timing information coming from the CPU.

The sample uses 3 buffers consisting of vertex data to facilitate a fireworks particle system. The first stream contains the initial particles needed to "seed" the system. It is used only once during the first frame that the particle system is active. The second and third buffers are ping-pong buffers and trade off being streamed to and rendered from every other frame.

The particle system works in the following manner:

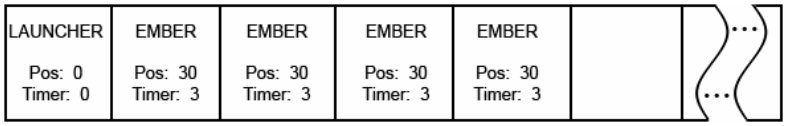
1. The seed buffer is filled with an initial launcher particle

2. The first time through the GS, the GS sees that the LAUNCHER is at 0 and emits a SHELL at the launcher position. NOTE: Because the particle system is rebuilt every pass through the GS, any particles that are necessary for the next frame need to be emitted, not just new particles.

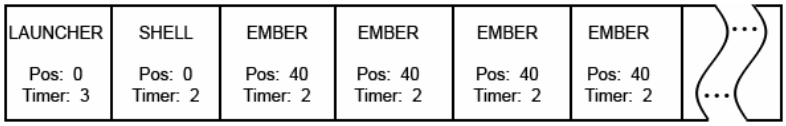
3. The second time through the GS, the LAUNCHER and SHELL timers are decremented and the SHELL is moved to a new position.

4. The SHELL timer is decremented to 0, which means that this is the last frame for this SHELL.

5. Because its timer is at 0, the SHELL is not emitted again. In its place, 4 EMBER particles are emitted.



6. The LAUNCHER is at zero, and therefore must emit another SHELL particle. The EMBER particles are moved to a new position and have their timers decremented as well. The LAUNCHER timer is reset.



The Geometry shader is organized as one main shader with a subroutine for each of the six particle types.

```
[maxvertexcount(256)]
void GSAdvanceParticlesMain(point VSParticleIn input[1], inout PointStream<VSParticleIn>
ParticleOutputStream) {
    if (input[0].Type == PT_LAUNCHER)
        GSLauncherHandler(input[0], ParticleOutputStream);
    else if (input[0].Type == PT_SHELL)
        GSShellHandler( input[0], ParticleOutputStream);
    else if (input[0].Type == PT_EMBER1 ||
             input[0].Type == PT_EMBER3)
        GSEmber1Handler(input[0], ParticleOutputStream);
    else if(input[0].Type == PT_EMBER2)
        GSEmber2Handler(input[0], ParticleOutputStream);
}
```

The subroutine for handling LAUNCHER particles looks like this:

```
void GSLauncherHandler(VSParticleIn input, inout PointStream<VSParticleIn>
ParticleOutputStream) {
    if (input.Timer <= 0) {
        float3 vRandom = normalize(RandomDir(input.Type));
        //time to emit a new SHELL
        VSParticleIn output;
        output.pos = input.pos + input.vel*g_fElapsedTime;
        output.vel = input.vel + vRandom*8.0;
        output.Timer = P_SHELLLIFE + vRandom.y*0.5;
        output.Type = PT_SHELL;
        ParticleOutputStream.Append(output);

        //reset our timer
        input.Timer = g_fSecondsPerFirework + vRandom.x*0.4;
    } else {
        input.Timer -= g_fElapsedTime;
    }

    //emit ourselves to keep us alive
    ParticleOutputStream.Append(input);
}
```

3.2.4 Knowing How Many Particles Were Output

Geometry Shaders can emit a variable amount of data each frame. Because of this, the sample has no way of knowing how many particles are in the buffer at any given time. Using standard Draw calls, the sample would have to guess at the number of particles to tell the GPU to draw. Fortunately, DrawAuto is designed to handle this situation. DrawAuto allows the dynamic amount of data written to streamout buffer to be used as the input amount of data for the draw call. Because this happens on the GPU, the CPU can advance and draw the particle system with no knowledge of how many particles actually comprise the system.

3.2.5 Rendering Particles

After the particles are advanced by the gsStreamOut Geometry Shader, the buffer that just received the output is used in a second pass for rendering the particles as point sprites. The VSSceneMain Vertex Shader takes care of assigning size and color to the particles based upon their type and age. GSSceneMain constructs point sprites from the points by emitting a 2 triangle strip for every point that is passed in. In this pass, the Geometry Shader output is passed to the rasterizer and does not stream out to any buffer.

```
[maxvertexcount(4)]
void GSScenemain(point VSParticleDrawOut input[1], inout TriangleStream<PSSceneIn>
SpriteStream) {
    PSSceneIn output;

    //
    // Emit two new triangles
    //
    for (int i=0; i<4; i++) {
        float3 position = g_positions[i]*input[0].radius;
        position = mul(position, (float3x3)g_mInvView) + input[0].pos;
        output.pos = mul(float4(position,1.0), g_mWorldViewProj);

        output.color = input[0].color;
        output.tex = g_texcoords[i];
        SpriteStream.Append(output);
    }
    SpriteStream.RestartStrip();
}
```

3.3 Instancing Sample

This sample demonstrates the use of the Instancing and Texture Arrays to reduce the number of draw calls required to render a complex scene. In addition, it uses AlphaToCoverage to avoid sorting semi-transparent primitives.

3.3.1 How the Sample Works

Reducing the number of draw calls made in any given frame is one way to improve graphics performance for a 3D application. The need for multiple draw calls in a scene arises from the different states required by different parts of the scene. These states often include matrices and material properties. One way to combat these issues is to use Instancing and Texture Arrays. In this sample, instancing enables the application to draw the same object multiple times in multiple places without the need for the CPU to update the world matrix for each object. Texture arrays allow multiple textures to be loaded into same resource and to be indexed by an extra texture coordinate, thus eliminating the need to change texture resources when a new object is drawn.

The Instancing sample draws several trees, each with many leaves, and several blades of grass using 3 draw calls. To achieve this, the sample uses one tree mesh, one leaf mesh, and one blade mesh instanced many time throughout the scene and drawn with DrawIndexedInstanced. To achieve variation in the leaf and grass appearance, texture arrays are used to hold different textures for both the leaf and grass instances. AlphaToCoverage allows the sample to further unburden the CPU and draw the leaves and blades of grass in no particular order. The rest of the environment is drawn in 6 draw calls.

Instancing the Tree

In order to replicate a tree the sample needs two pieces of information. The first is the mesh information. In this case, the mesh is loaded from tree_super.x. The second piece of information is a buffer containing a list of matrices that describe the locations of all tree instances. The sample uses IASetVertexBuffers to bind the mesh information to vertex stream 0 and the matrices to stream 1. To get this information into the shader correctly, the following InputLayout is used:

```
const D3D10_INPUT_ELEMENT_DESC instlayout[] =
{
    { L"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"TEXTURE0", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"mTransform", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 0,
D3D10_INPUT_PER_INSTANCE_DATA, 1 },
    { L"mTransform", 1, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 16,
D3D10_INPUT_PER_INSTANCE_DATA, 1 },
    { L"mTransform", 2, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 32,
D3D10_INPUT_PER_INSTANCE_DATA, 1 },
    { L"mTransform", 3, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 48,
D3D10_INPUT_PER_INSTANCE_DATA, 1 },
```

```
};
```

The vertex shader will be called (number of vertices in mesh)*(number of instance matrices) times. Because the matrix is a shader input, the shader can position the vertex at the correct location according to which instance it happens to be processing.

Instancing the Leaves

Because one leaf is instanced over an entire tree and one tree is instanced several times throughout the sample, the leaves must be handled differently than the tree and grass meshes. The matrices for the trees are loaded into a constant buffer. The InputLayout is setup to make sure the shader sees the leaf mesh data m_iNumTreeInstances time before stepping to the next leaf matrix. The last element, fOcc, is a baked occlusion term used to shade the leaves.

```
const D3D10_INPUT_ELEMENT_DESC leaflayout[] =
{
    { L"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"TEXTURE0", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"mTransform", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 0,
D3D10_INPUT_PER_INSTANCE_DATA, m_iNumTreeInstances },
    { L"mTransform", 1, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 16,
D3D10_INPUT_PER_INSTANCE_DATA, m_iNumTreeInstances },
    { L"mTransform", 2, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 32,
D3D10_INPUT_PER_INSTANCE_DATA, m_iNumTreeInstances },
    { L"mTransform", 3, DXGI_FORMAT_R32G32B32A32_FLOAT, 1, 48,
D3D10_INPUT_PER_INSTANCE_DATA, m_iNumTreeInstances },
    { L"fOcc", 0, DXGI_FORMAT_R32_FLOAT, 1, 64, D3D10_INPUT_PER_INSTANCE_DATA,
m_iNumTreeInstances },
};
```

The Input Assembler automatically generates an InstanceID which can be passed into the shader. The following snippet of shader code demonstrates how the leaves are positioned.

```
    int iTree = input.InstanceId%g_iNumTrees;
    float4 vInstancePos = mul(float4(input.pos, 1), input.mTransform);
    float4 InstancePosition = mul(vInstancePos, g_mTreeMatrices[iTree]);
```

If there were 3 trees in the scene, the leaves would be drawn in the following order: Tree1, leaf1; Tree2, leaf1; Tree3, leaf1; Tree1, leaf2; Tree2, leaf2; etc...

Instancing the Grass

Grass rendering is handled differently than the Tree and Leaves. Instead of using the input assembler to instance the grass using a separate stream of matrices, the grass is dynamically generated in the geometry shader. The top of the island mesh is passed to the vertex shader, which passes this information directly to the GSGrassmain geometry shader. Depending on the grass density specified, the GSGrassmain calculates psuedo-random positions on the current triangle that correspond to grass positions. These positions are then passed to a helper function that creates a blade of grass at the point. An 1D texture of random floating point values is used to provide the psuedo-random numbers. It is indexed by vertex ids of the input mesh. This ensures that the random distribution doesn't change from frame to frame.

```
struct VSGrassOut {
    float3 pos : POSITION;
    float3 norm : NORMAL;
    uint VertexID : VERTID;
};
struct PSQuadIn{
    float4 pos : SV_Position;
    float3 tex : TEXTURE0;
```

```
    float4 color : COLOR0;
};

float3 RandomDir(float fOffset) {
    float4 tCoord = float4((fOffset) / 300.0, 0, 0, 0);
    return g_txRandom.SampleLevel(g_samPoint, tCoord);
}

VSGrassOut CalcMidPoint(VSGrassOut A, VSGrassOut B, VSGrassOut C) {
    VSGrassOut MidPoint;

    MidPoint.pos = (A.pos + B.pos + C.pos)/3.0f;
    MidPoint.norm = (A.norm + B.norm + C.norm)/3.0f;
    MidPoint.VertexID = A.VertexID + B.VertexID + C.VertexID;

    return MidPoint;
}

void OutputGrassBlade(VSGrassOut midPoint, inout TriangleStream<PSQuadIn> GrassStream) {
    PSQuadIn output;

    float4x4 mWorld = GetRandomOrientation(midPoint.pos, midPoint.norm,
(float)midPoint.VertexID);
    int iGrassTex = midPoint.VertexID%4;

    float3 grassNorm = mWorld._m20_m21_m22;
    float4 color1 = saturate(dot(g_sunDir,grassNorm))*g_sunColor;
    color1 += saturate(dot(g_bottomDir,grassNorm))*g_bottomColor;
    color1 += g_quadambient;

    for (int v=0; v<6; v++) {
        float3 pos = g_positions[v];
        pos.x *= g_GrassWidth;
        pos.y *= g_GrassHeight;

        output.pos = mul(float4(pos,1), mWorld );
        output.pos = mul(output.pos, g_mWorldViewProj);
        output.tex = float3(g_texcoords[v], iGrassTex);
        output.color = color1;

        GrassStream.Append(output);
    }

    GrassStream.RestartStrip();
}

 [maxvertexcount(512)]
void GSGrassmain(triangle VSGrassOut input[3], inout TriangleStream<PSQuadIn>
GrassStream) {
    VSGrassOut MidPoint = CalcMidPoint(input[0], input[1], input[2]);

    for (int i=0; i<g_iGrassCoverage; i++) {
        float3 Tan = RandomDir(MidPoint.pos.x + i);
        float3 Len = normalize(RandomDir(MidPoint.pos.z + i));
        float3 Shift = Len.x*g_GrassMessiness*normalize(cross(Tan, MidPoint.norm));
        VSGrassOut grassPoint = MidPoint;
        grassPoint.VertexID += i;
        grassPoint.pos += Shift;

        //uncomment this to make the grass strictly conform to the mesh
        //if (IsInTriangle(grassPoint.pos, input[0].pos, input[1].pos, input[2].pos))
            OutputGrassBlade(grassPoint, GrassStream);
    }
}
```

Alternatively, the grass can be rendered in the same way as the tree by placing the vertices of a quad into the first vertex stream and the matrices for each blade of grass in the second vertex stream. The fOcc element of the second stream can be used to place precalculated shadows on the blades of grass (just as it is used to precalculate shadows on the leaves). However, the storage space for a stream of several hundred

thousand matrices is a concern even on modern graphics hardware. The grass generation method of using the geometry shader, while lacking built-in shadows, uses far less storage.

Changing Leaves with Texture Arrays

Texture arrays are just what the name implies. They are arrays of textures, each with full mip-chains. For a texture2D array, the array is indexed by the z coordinate. Because the InstanceID is passed into the shader, the sample uses InstanceID%numArrayIndices to determine which texture in the array to use for rendering that specific leaf or blade of grass.

Drawing Transparent Objects with Alpha To Coverage

The number of transparent leave and blades of grass in the sample makes sorting these objects on the CPU expensive. Alpha to coverage helps solve this problem by allowing the Instancing sample to produce convincing results without the need to sort leaves and grass back to front. Alpha to coverage must be used with multisample anti-aliasing (MSAA). MSAA is a method to get edge anti-aliasing by evaluating triangle coverage at a higher-frequency on a higher resolution z-buffer. With alpha to coverage, the MSAA mechanism can be tricked into creating psuedo order independant transparency. Alpha to coverage generates a MSAA coverage mask for a pixel based upon the pixel shader output alpha. That result gets AND'ed with the coverage mask for the triangle. This process is similar to screen-door transparency, but at the MSAA level.

Alpha to coverage is not designed for true order independent transparency like windows, but works great for cases where alpha is being used to represent coverage, like in a mipmapped leaf texture.

3.4 Shadow Volume Sample

3.4.1 How the Sample Works

A shadow volume of an object is the region in the scene that is covered by the shadow of the object caused by a particular light source. When rendering the scene, all geometry that lies within the shadow volume should not be lit by the particular light source. A closed shadow volume consists of three parts: a front cap, a back cap, and the side. The front and back caps can be created from the shadow-casting geometry: the front cap from light-facing faces and the back cap from faces facing away from light. In addition, the back cap is formed by translating the front facing faces a large distance away from the light, to make the shadow volume long enough to cover enough geometry in the scene. The side is usually created by first determining the silhouette edges of the shadow-casting geometry then generating faces that represent the silhouette edges extruded for a large distance away from the light direction. Figure 20 shows different parts of a shadow volume.

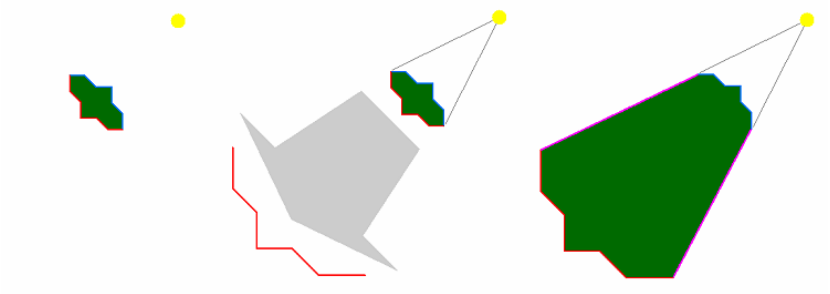
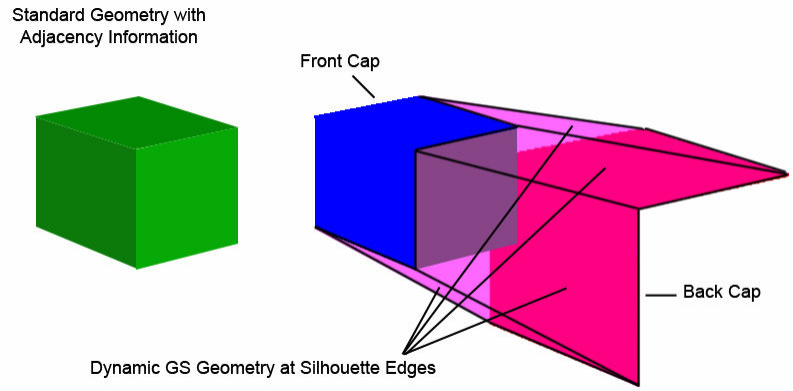


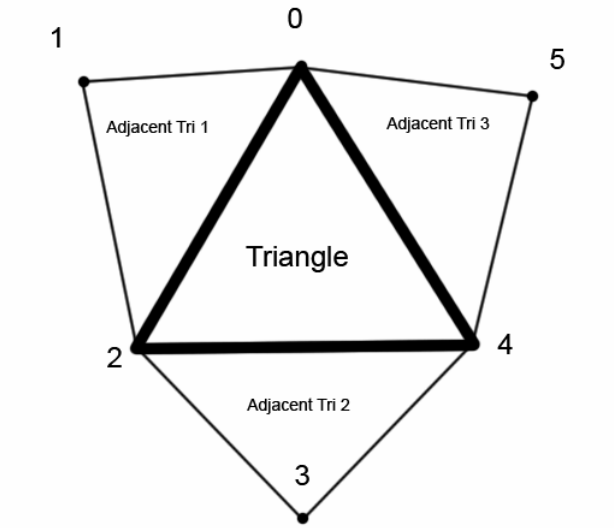
Figure 20: Creation of a shadow volume.

The front cap (blue) and back cap (red) are created from the occluder's geometry. The back cap is translated to prolong the shadow volume, and the side faces (purple) are generated to enclose the shadow volume.

This sample demonstrates a specific implementation of shadow volumes. Many traditional shadow volume approaches determine the silhouette and generate the shadow volume geometry on the CPU. This sample determines the silhouette in the geometry shader and uses the fact that the geometry shader can send a variable amount of data to the rasterizer to create new shadow volume geometry on the fly. The underlying idea is that for triangles that face the light, we can use them as-is for the front cap of the shadow volume. The back cap is generated from the front facing triangles translated a large distance along the light direction at each vertex, then they can be used as the back cap. However, a problem occurs at silhouette edges where one triangle faces the light and its neighbor faces away from the light. In this situation, the geometry shader extrudes two new triangles to create a quad to match up between the front cap of the shadow volume and the back cap.



In order for the geometry shader to find silhouette edges it must know which faces are adjacent to each other. Fortunately, the geometry shader has support for a new type of input primitive, triangleadj. Triangleadj assumes that every other vertex is an adjacent vertex. The index buffer of the geometry must be modified to reflect the adjacency information of the mesh. The CDXUTMesh10::ConvertToAdjacencyIndices handles this by creating an index buffer in which every other value is the index of the adjacent vertex of the triangle that shares an edge with the current triangle. The index buffer will double in size due to the extra information being stored. The figure below demonstrates the ordering of the adjacency information.



The Geometry shader that generates the shadow volume is structured as a main shader that processes the triangles and a helper subroutine that performs the side extrusions along silhouette edges:

```
void DetectAndProcessSilhouette(float3 N,                  // Un-normalized triangle
normal
                                GSShadowIn v1,    // Shared vertex
                                GSShadowIn v2,    // Shared vertex
                                GSShadowIn vAdj,  // Adjacent triangle vertex
                                inout TriangleStream<PSShadowIn> ShadowTriangleStream) {
    float3 NAdj = cross(v2.pos - vAdj.pos, v1.pos - vAdj.pos);

    // we should be not facing the light
    float fDot = dot(N, NAdj);
    if (fDot < 0.0) {
        float3 outpos[4];
        float3 extrude1 = normalize(v1.pos - g_vLightPos);
        float3 extrude2 = normalize(v2.pos - g_vLightPos);

        outpos[0] = v1.pos + g_fExtrudeBias*extrude1;
        outpos[1] = v1.pos + g_fExtrudeAmt*extrude1;
        outpos[2] = v2.pos + g_fExtrudeBias*extrude2;
        outpos[3] = v2.pos + g_fExtrudeAmt*extrude2;

        // Extrude silhouette to create two new triangles
        PSShadowIn Out;
        for(int v=0; v<4; v++) {
            Out.pos = mul(float4(outpos[v],1), g_mViewProj);
            ShadowTriangleStream.Append(Out);
        }
        ShadowTriangleStream.RestartStrip();
    }
}

[maxvertexcount(18)]
void GSShadowmain(triangleadj GSShadowIn In[6], inout TriangleStream<PSShadowIn>
ShadowTriangleStream) {
    // Compute un-normalized triangle normal
    float3 N = cross(In[2].pos - In[0].pos, In[4].pos - In[0].pos);

    // Compute direction from this triangle to the light
    float3 lightDir = g_vLightPos - In[0].pos;

    //if we're facing the light
    if (dot(N, lightDir) > 0.0f) {
        // for each edge of the triangle, determine if it is a silhouette edge
        DetectAndProcessSilhouette(lightDir, In[0], In[2], In[1], ShadowTriangleStream);
```

```
        DetectAndProcessSilhouette(lightDir, In[2], In[4], In[3], ShadowTriangleStream);
        DetectAndProcessSilhouette(lightDir, In[4], In[0], In[5], ShadowTriangleStream);

        //near cap
        PSShadowIn Out;
        for(int v=0; v<6; v+=2) {
            float3 extrude = normalize(In[v].pos - g_vLightPos);

            float3 pos = In[v].pos + g_fExtrudeBias*extrude;
            Out.pos = mul(float4(pos,1), g_mViewProj);
            ShadowTriangleStream.Append(Out);
        }
        ShadowTriangleStream.RestartStrip();

        //far cap (reverse the order)
        for (int v=4; v>=0; v-=2) {
            float3 extrude = normalize(In[v].pos - g_vLightPos);

            float3 pos = In[v].pos + g_fExtrudeAmt*extrude;
            Out.pos = mul(float4(pos,1), g_mViewProj);
            ShadowTriangleStream.Append(Out);
        }
        ShadowTriangleStream.RestartStrip();
    }
}
```

3.4.2 Rendering Shadows

At the top level, the rendering steps look like the following:

- If ambient lighting is enabled, render the entire scene with ambient only.
- For each light in the scene, do these:
    - Disable depth-buffer and frame-buffer writing.
    - Prepare the stencil buffer render states for rendering the shadow volume.
    - Render the shadow volume mesh with a vertex extruding shader. This sets up the stencil buffer according to whether or not the pixels are in the shadow volume.
    - Prepare the stencil buffer render states for lighting.
    - Prepare the additive blending mode.
    - Render the scene for lighting with only the light being processed.

The lights in the scene must be processed separately because different light positions require different shadow volumes, and thus different stencil bits get updated. Here is how the code processes each light in the scene in details. First, it renders the shadow volume mesh without writing to the depth buffer and frame buffer. These buffers need to be disabled because the purpose of rendering the shadow volume is merely setting the stencil bits for pixels covered by the shadow, and the shadow volume mesh itself should not be visible in the scene. The shadow mesh is rendered using the depth-fail stencil shadow technique and a vertex extruding shader. In the shader, the vertex's normal is examined. If the normal points toward the light, the vertex is left where it is. However, if the normal points away from the light, the vertex is extruded to infinity. This is done by making the vertex's world coordinates the same as the light-to-vertex vector with a W value of 0. The effect of this operation is that all faces facing away from the light get projected to infinity along the light direction. Since faces are connected by quads, when one face gets projected and its neighbor does not, the quad between them is no longer degenerate. It is stretched to become the side of the shadow volume. Figure 23 shows this.

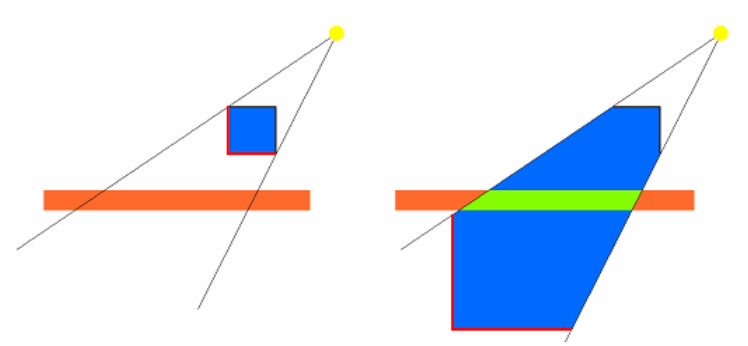


Figure 21: The faces of the shadow volume that face away from light (left, shown in red) have their vertices extruded by the vertex shader to create a volume that encloses area covered by shadows (right).

When rendering the shadow mesh with the depth-fail technique, the code first renders all back-facing triangles of the shadow mesh. If a pixel's depth value fails the depth comparison (usually this means the pixel's depth is greater than the value in the depth buffer), the stencil value is incremented for that pixel. Next, the code renders all front-facing triangles, and if a pixel's depth fails the depth comparison, the stencil value for the pixel is decremented. When the entire shadow volume mesh has been rendered in this fashion, the pixels in the scene that are covered by the shadow volume have a non-zero stencil value while all other pixels have a zero stencil. Lighting for the light being processed can then be done by rendering the entire scene and writing out pixels only if their stencil values are zero.

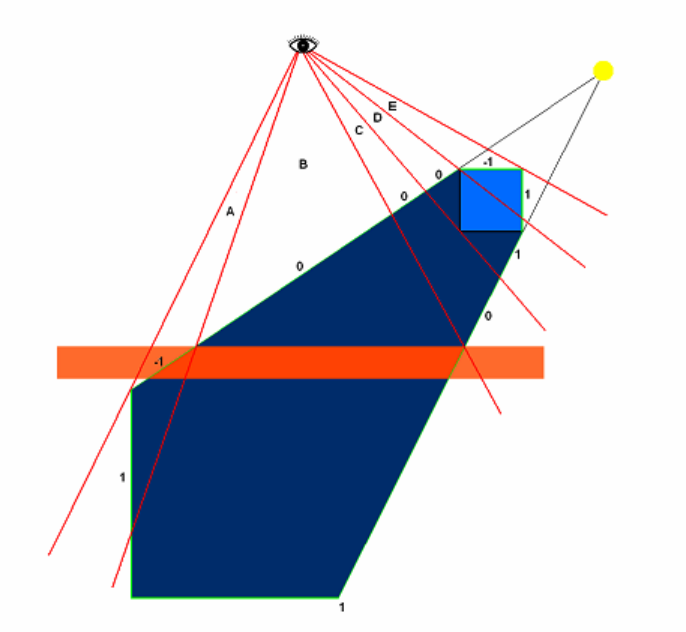


Figure 22: The depth-fail technique.

Figure 22 illustrates the depth-fail technique. The orange block represents the shadow receiver geometry. Regions A, B, C, D and E are five areas in the frame buffer where the shadow volume is rendered. The numbers indicate the stencil value changes as the front and back faces of the shadow volume are rendered. In region A and E, both the front and back faces of the shadow volume fail the depth test, and therefore both cause the stencil value to change. For region A, the orange shadow receiver is causing the depth test to fail, and for region E, the cube's geometry is failing the test. The net result is that stencil values in this

region stay at 0 after all faces are rendered. In region B and D, the front faces pass the depth test while the back faces fail, so the stencil values are not changed with the front faces, and the net stencil changes are 1. In region C, both the front and back faces pass the depth test, and so neither causes the stencil values to change, and the stencil values stay at 0 in this region. When the shadow volume is completely rendered, only the stencil values in regions B and D are non-zero, which correctly indicates that regions B and D are the only shadowed areas for this particular light.

3.4.3 Performance Considerations

The current sample performs normal rendering and shadow rendering using the same mesh. Because the mesh class tracks only one index buffer at a time, adjacency information is sent to the shader even when it is not needed for shadow calculations. The shader must do extra work to remove this adjacency information in the geometry shader. To improve performance the application could keep two index buffers for each mesh. One would be the standard index buffer and would be used for non-shadow rendering. The second would contain adjacency information and only be used when extruding shadow volumes.

Finally, there is another area that could call for some performance optimization. As shown earlier, the rendering algorithm with shadow volumes requires that the scene be rendered in multiple passes (one plus the number of lights in the scene, to be precise). Every time the scene is rendered, the same vertices get sent to the device and processed by the vertex shaders. This can be avoided if the application employs deferred lighting with multiple rendertargets. With this technique, the application renders the scene once and outputs a color map, a normal map, and a position map. Then, in subsequent passes, it can retrieve the values in these maps in the pixel shader and apply lighting based on the color, normal and position data it reads. The benefit of doing this is tremendous. Each vertex in the scene only has to be processed once (during the first pass), and each pixel is processed exactly once in subsequent passes, thus ensuring that no overdraw happens in these passes.

3.4.4 Shadow Volume Artifacts

It is worth noting that a shadow volume is not a flawless shadow technique. Aside from the high fill-rate requirement and silhouette determination, the image rendered by the technique can sometimes contain artifacts near the silhouette edges, as shown in Figure 23. The prime source of this artifact lies in the fact that when a geometry is rendered to cast shadow onto itself, its faces are usually entirely in shadow or entirely lit, depending on whether the face's normal points toward the light. Lighting computation, however, uses vertex normals instead of face normals. Therefore, for a face that is near-parallel to the light direction, it will either be all lit or all shadowed, when in truth, only part of it might really be in shadow. This is an inherent flaw of stencil shadow volume technique, and should be a consideration when implementing shadow support. The artifact can be reduced by increasing mesh details, at the cost of higher rendering time for the mesh. The closer to the face normal that the vertex normals get, the less apparent the artifact will be. If the application cannot reduce the artifact down to an acceptable level, it should also consider using other types of shadow technique, such as shadow mapping or pre-computed radiance transfer.

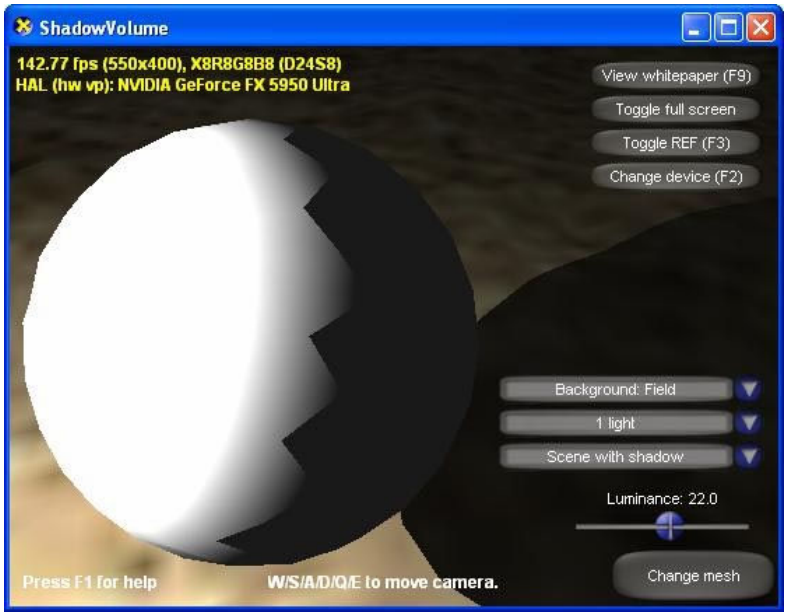


Figure 23: Shadow volume artifact near the silhouette edges.

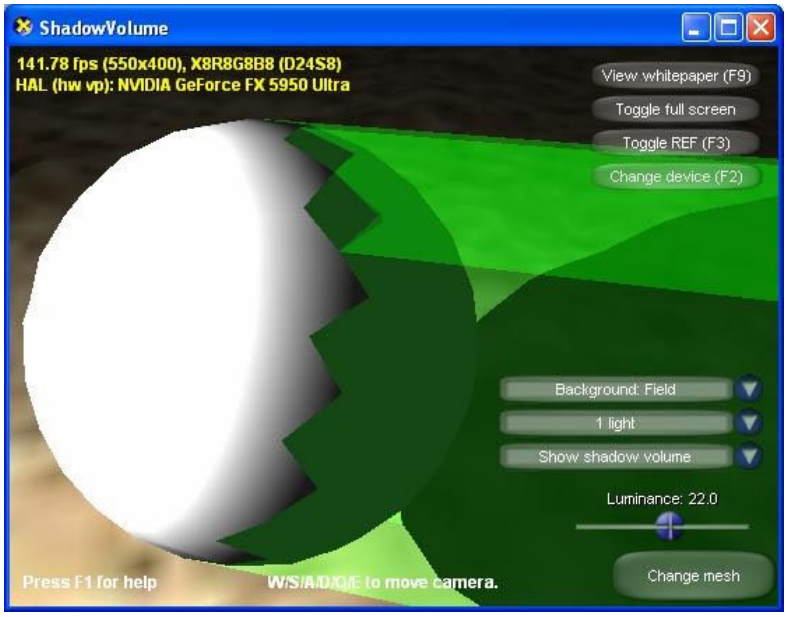


Figure 24: Displaying the shadow volume reveals that the jagged edge is caused by faces that are treated as being entirely in shadow when they are actually only partially shadowed.

3.5 DisplacementMapping10 Sample

This sample implements displacement mapping by ray tracing through geometry extruded from a base mesh. The sample uses Direct3D 10 and the geometry shader to extrude prisms which are then decomposed into three tetrahedra for each triangle in the mesh. The pixel shader ray traces through the tetrahedra to intersect the displacement map. The diffuse color and normal values are then calculated for this point and used to set the final color of the pixel on screen. This sample is an adaptation of a paper from Microsoft

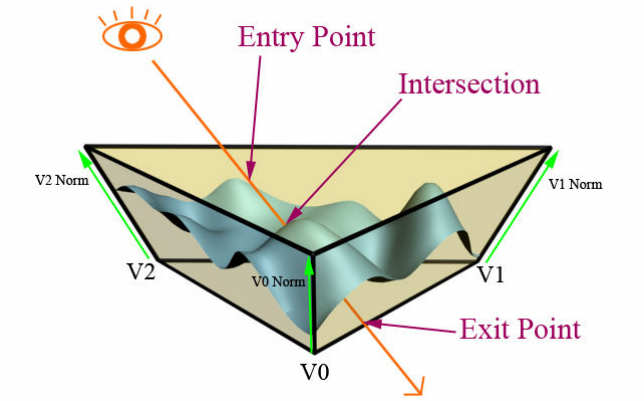
Research Asia (Wang, Xi, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. 2004. Generalized Displacement Maps. In Eurographics Symposium on Rendering 2004, pp. 227-234.)

3.5.1 Sample Overview

There have been many approaches to displacement mapping in recent history. Many rely on tessellating the geometry to a high level of detail and then displacing this geometry based upon a height value stored in a height map. DisplacementMapping10 uses a different approach. Instead of producing detail by displacing highly tessellated geometry, the sample creates three tetrahedra for every input triangle and uses a per-pixel ray tracing approach to evaluate the geometry.

3.5.2 Extruding Geometry

For every triangle on the mesh, a triangular prism is extruded. The prism is constructed by extruding the vertices of the triangle along the directions of their normals by an amount specified to be the maximum displacement. The XY texture coordinates at the top of the prism remain the same as the bottom; however, the Z coordinate becomes 1 to represent the maximum displacement. A naive approach would be to ray trace through this extruded prism by finding the texture coordinates where the eye ray enters and exits the prism. By taking an evenly spaced sampling of a height map texture between the entry and exit points, one could easily determine where the ray intersects the height map.



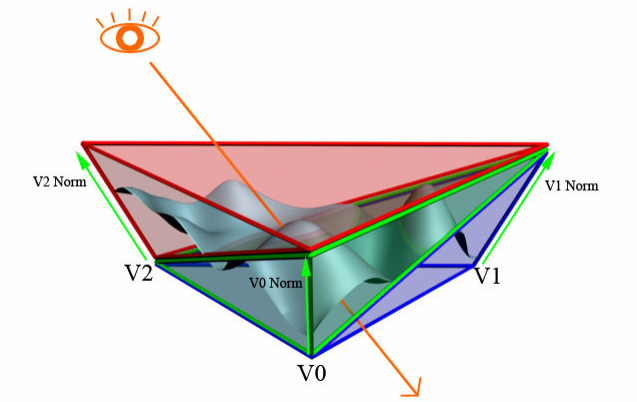
Unfortunately, there are two problems with this approach.

Problem 1: Because of varying curvature in the mesh, the four points that make up each side of the prism are not guaranteed to be coplanar. When extruding the prism, each side will be split into two triangles. Any neighboring face will also share this edge as an edge of its extruded prism. If the neighboring triangle does not create the sides of its prism in a way that correlates to how the current triangle creates the same edge, cracks may appear in the final rendering.

Problem 2: It's very hard to calculating the exit texture coordinate for an eye ray traveling through a triangular prism. The entry point, on the other hand is easy. Because of depth-buffering, the nearest triangle to the eye will always be drawn. The graphics hardware will automatically interpolate the entry texture coordinate across the draw faces. The rear exit point is a different story. One could do ray-plane intersections to get the distance to the rear of the prism. This is a good start, but still doesn't give us the texture coordinates. Calculating barycentrics for a triangle are costly in the shader. One could pass down an un-normalized tangent basis frame (TBN matrix) and transform the view vector into this frame, add it to the input texture coordinate, and achieve the exit texture coordinate. Unfortunately, this breaks down when the mesh deforms since under high deformation, the un-normalized tangent frame is non-constant across the prism.

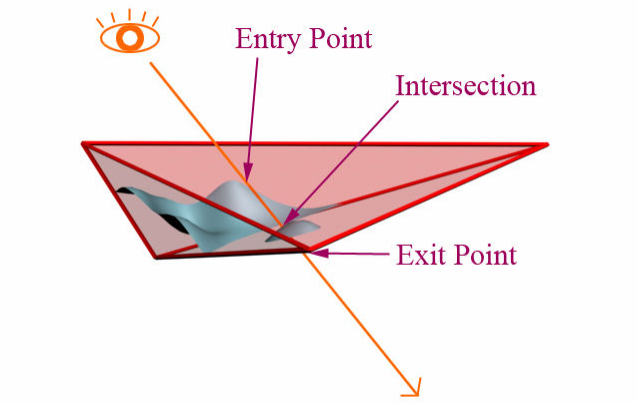
3.5.3 Using Tetrahedra

The two problems above can be solve by decomposing the the prism into three tetrahedra and by using the VertexID generated by the input assembler to order the vertices when creating the tetrahedra.



The distance to the rear of a tetrahedron is computed by intersecting the eye ray with the planes of the tetrahedron facing away from the eye and choosing the smallest distance. Now, the only problem left is to determine the texture coordinates at the exit point.

A nice property of a tetrahedron is that by using the texture coordinates of the 4 points, a constant texture gradient can be found across the tetrahedron. By using this gradient, the entry texture coordinate (computed easily during rasterization), and the distance to the rear of the tetrahedron, the shader can compute the exit texture coordinates.



The Geometry shader used to create the tetrahedra is structured as a main shader and several helper subroutines used to create the individual tetrahedra:

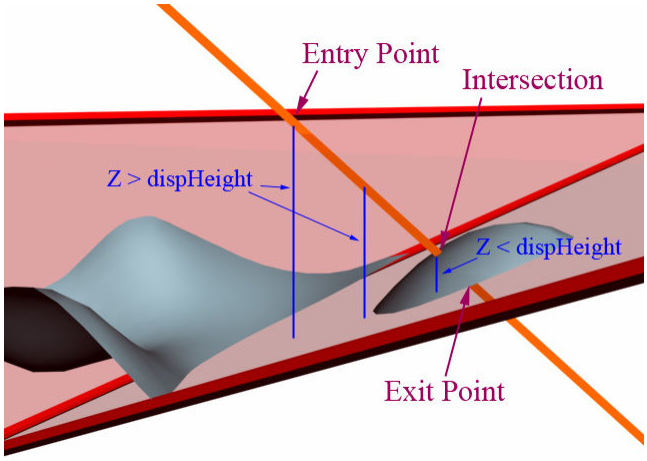
```
struct VSDisplaceOut {
        float4 Pos       : POS;         //Position
        float3 vPos      : POSVIEW;     //view pos
        float3 Norm      : NORMAL;      //Normal
        float3 Tex       : TEXCOORD0;   //Texture coordinate
        float3 Tangent   : TANGENT;     //Normalized Tangent vector
        float3 BiTangent : BITANGENT;   //Normalized BiTangent vector
        uint   VertID    : VertID; //verTex ID, used for consistent tetrahedron
generation
};
struct PSDisplaceIn {
        float4 Pos       : SV_Position;
        float4 planeDist : TEXCOORD0;
        float3 Norm      : TEXCOORD1;
        float3 vPos      : TEXCOORD2;
        float3 TanT      : TEXCOORD3;    // Normalized Texture space Tangent vector
        float3 BiTanT    : TEXCOORD4;    // Normalized Texture space BiTangent vector
        float3 Tex       : TEXCOORD5;    // Texture Coordinate of the first vert
        float3 Gx        : TEXCOORD6;    // Gradient of the tetrahedron for X texcoord
        float3 Gy        : TEXCOORD7;    // Gradient of the tetrahedron for Y texcoord
        float3 Gz        : TEXCOORD8;    // Gradient of the tetrahedron for Z texcoord
        float3 pos0      : TEXCOORD9;
};

float RayDistToPlane(float3 vPoint, float3 vDir, float3 A, float3 planeNorm) {
        float Nom = dot(planeNorm, float3(A - vPoint));
        float DeNom = dot(planeNorm, vDir);
        return Nom/DeNom;
}

void CalcGradients(inout PSDisplaceIn V0, inout PSDisplaceIn V1, inout PSDisplaceIn V2,
inout PSDisplaceIn V3, float3 N0, float3 N1, float3 N2, float3 N3) {
        float dotN0 = dot(N0, V0.vPos - V3.vPos);
        float dotN1 = dot(N1, V1.vPos - V2.vPos);
        float dotN2 = dot(N2, V2.vPos - V1.vPos);
        float dotN3 = dot(N3, V3.vPos - V0.vPos);

        float3  Gx =  (V0.Tex.x / dotN0 )*N0;
                      Gx += (V1.Tex.x / dotN1)*N1;
                      Gx += (V2.Tex.x / dotN2)*N2;
                      Gx += (V3.Tex.x / dotN3)*N3;

        float3  Gy =  (V0.Tex.y / dotN0 )*N0;
                      Gy += (V1.Tex.y / dotN1)*N1;
                      Gy += (V2.Tex.y / dotN2)*N2;
                      Gy += (V3.Tex.y / dotN3)*N3;

        float3  Gz =  (V0.Tex.z / dotN0 )*N0;
                      Gz += (V1.Tex.z / dotN1)*N1;
                      Gz += (V2.Tex.z / dotN2)*N2;
                      Gz += (V3.Tex.z / dotN3)*N3;

       V0.Tex = V0.Tex;   V1.Tex = V0.Tex;   V2.Tex = V0.Tex;   V3.Tex = V0.Tex;
       V0.pos0 = V0.vPos; V1.pos0 = V0.vPos; V2.pos0 = V0.vPos; V3.pos0 = V0.vPos;
       V0.Gx = Gx;        V1.Gx = Gx;        V2.Gx = Gx;        V3.Gx = Gx;
       V0.Gy = Gy;        V1.Gy = Gy;        V2.Gy = Gy;        V3.Gy = Gy;
       V0.Gz = Gz;        V1.Gz = Gz;        V2.Gz = Gz;        V3.Gz = Gz;
 }

void GSCreateTetra(in VSDisplaceOut A, in VSDisplaceOut B, in VSDisplaceOut C, in
VSDisplaceOut D, inout TriangleStream<PSDisplaceIn> DisplaceStream) {
        float3 AView = normalize(A.vPos - g_vEyePt);
        float3 BView = normalize(B.vPos - g_vEyePt);
        float3 CView = normalize(C.vPos - g_vEyePt);
        float3 DView = normalize(D.vPos - g_vEyePt);

       PSDisplaceIn Aout;    PSDisplaceIn Bout;    PSDisplaceIn Cout;    PSDisplaceIn Dout;
```

```
        Aout.Pos = A.Pos;      Bout.Pos = B.Pos;      Cout.Pos = C.Pos;      Dout.Pos = D.Pos;
        Aout.vPos = A.vPos;    Bout.vPos = B.vPos;    Cout.vPos = C.vPos;    Dout.vPos = D.vPos;
        Aout.Norm = A.Norm;    Bout.Norm = B.Norm;    Cout.Norm = C.Norm;    Dout.Norm = D.Norm;
        Aout.Tex = A.Tex;      Bout.Tex = B.Tex;      Cout.Tex = C.Tex;      Dout.Tex = D.Tex;
        Aout.TanT = A.Tangent; Bout.TanT = B.Tangent; Cout.TanT = C.Tangent; Dout.TanT = D.Tangent;
        Aout.BiTanT =          Bout.BiTanT =          Cout.BiTanT =          Dout.BiTanT =
        A.BiTangent ;          B.BiTangent;           C.BiTangent;           D.BiTangent;

         float3 AB = C.vPos-B.vPos;
         float3 AC = D.vPos-B.vPos;
         float3 planeNormA = normalize(cross(AC, AB));
                    AB = D.vPos-A.vPos;
                    AC = C.vPos-A.vPos;
         float3 planeNormB = normalize(cross(AC, AB));
                    AB = B.vPos-A.vPos;
                    AC = D.vPos-A.vPos;
         float3 planeNormC = normalize(cross(AC, AB));
                    AB = C.vPos-A.vPos;
                    AC = B.vPos-A.vPos;
         float3 planeNormD = normalize(cross(AC, AB));

         Aout.planeDist.x = Aout.planeDist.y = Aout.planeDist.z = 0.0f;
         Aout.planeDist.w = RayDistToPlane(A.vPos, AView, B.vPos, planeNormA);

         Bout.planeDist.x = Bout.planeDist.z = Bout.planeDist.w = 0.0f;
         Bout.planeDist.x = Bout.planeDist.z = Bout.planeDist.w = 0.0f;
         Bout.planeDist.y = RayDistToPlane(B.vPos, BView, A.vPos, planeNormB);

         Cout.planeDist.x = Cout.planeDist.y = Cout.planeDist.w = 0.0f;
         Cout.planeDist.z = RayDistToPlane(C.vPos, CView, A.vPos, planeNormC);

         Dout.planeDist.y = Dout.planeDist.z = Dout.planeDist.w = 0.0f;
         Dout.planeDist.x = RayDistToPlane(D.vPos, DView, A.vPos, planeNormD);

         CalcGradients(Aout, Bout, Cout, Dout, planeNormA, planeNormB, planeNorm C,
planeNormD);

         DisplaceStream.Append(Cout);
         DisplaceStream.Append(Bout);
         DisplaceStream.Append(Aout);
         DisplaceStream.Append(Dout);
         DisplaceStream.Append(Cout);
         DisplaceStream.Append(Bout);
         DisplaceStream.RestartStrip();
}

void SWAP(inout VSDisplaceOut A, inout VSDisplaceOut B){
         VSDisplaceOut Temp = A;
         A = B;
         B = Temp;
}

[maxvertexcount(18)]
void GSDisplaceMain(triangle VSDisplaceOut In[3], inout TriangleStream<PSDisplaceIn>
DisplaceStream) {
         //Don't extrude anything that's facing too far away from us
         //Just saves geometry generation
         float3 AB = In[1].Pos - In[0].Pos;
         float3 AC = In[2].Pos - In[0].Pos;
         float3 triNorm = normalize(cross(AB, AC));
         float3 view = normalize(In[0].Pos - g_vEyePt);
         if (dot(triNorm, view) < 0.5) {

                  //Extrude along the Normals
                  VSDisplaceOut v[6];
                  for(int i=0; i<3; i++) {
                           float4 PosNew = In[i].Pos;
                           float4 PosExt = PosNew + float4(In[i].Norm*g_MaxDisplacement,0);
                           v[i].vPos = PosNew.xyz;
                           v[i+3].vPos = PosExt.xyz;
```

```
                        v[i].Pos = mul(PosNew, g_mViewProj);
                        v[i+3].Pos = mul(PosExt, g_mViewProj);

                        v[i].Tex = float3(In[i].Tex.xy,0);
                        v[i+3].Tex = float3(In[i].Tex.xy,1);

                        v[i].Norm = In[i].Norm;

                        v[i].Norm = In[i].Norm;
                        v[i+3].Norm = In[i].Norm;

                        v[i].Tangent = In[i].Tangent;
                        v[i+3].Tangent = In[i].Tangent;

                        v[i].BiTangent = In[i].BiTangent;
                        v[i+3].BiTangent = In[i].BiTangent;
                }

                // Make sure that our prism hasn't "flipped" on itself after the extrusion
                AB = v[4].vPos - v[3].vPos;
                AC = v[5].vPos - v[3].vPos;
                float3 topNorm = cross(AB, AC);
                float lenTop = length( topNorm );
                if (dot( topNorm, triNorm ) < 0 || lenTop < 0.005) {
                        for (int i=0; i<3; i++) {
                                float4 PosNew = In[i].Pos;
                                float4 PosExt = PosNew +
float4(In[i].Norm*g_Minisplacement,0);
                                v[i].vPos = PosNew.xyz;
                                v[i+3].vPos = PosExt.xyz;

                                v[i].Pos = mul(PosNew, g_mViewProj);
                                v[i+3].Pos = mul(PosExt, g_mViewProj);
                        }
                }

                               // Create 3 tetrahedra
                GSCreateTetra(v[4], v[5], v[0], v[3], DisplaceStream);
                GSCreateTetra(v[5], v[0], v[1], v[4], DisplaceStream);
                GSCreateTetra(v[0], v[1], v[2], v[5], DisplaceStream);
        }
```

3.5.4 Intersecting the Displacement Map

Once the entry and exit texture coordinates are known, it's only a matter of stepping from one to the other. While stepping along the ray from the entry to the exit coordinates, the XY components are used to lookup into the displacement map. The Z component represents the height from the base triangle. This is compared against the height value returned by the lookup into the displacement map. If the Z component of the ray is less than the height returned by the displacement map, the sample assumes that an intersection has taken place and stop searching. If no intersection is found before reaching the exit texture coordinate, the pixel is discarded.



Once the intersection point is found, all that is left is to determine the color and normal at the point of intersection. Again, if no intersection point is found, the pixel is simply discarded. The diffuse color at the intersection point can be found by sampling the diffuse texture at the XY coordinates of the ray where the ray intersected the displacement map. While the displacement map provides enough information to determine the normal at the point of intersection (using finite differencing), this sample encodes a normal map as the rgb components of the displacement texture (displacement is stored in alpha). Thus, the last sample along the entry/exit ray contains the normal for the point of intersection.

```
#define MAX_DIST 1000000000.0f
#define MAX_STEPS 16.0
float4 PSDisplaceMain(PSDisplaceIn input) : SV_Target {
        float4 modDist = float4(0,0,0,0);
        modDist.x = input.planeDist.x > 0 ? input.planeDist.x : MAX_DIST;
        modDist.y = input.planeDist.y > 0 ? input.planeDist.y : MAX_DIST;
        modDist.z = input.planeDist.z > 0 ? input.planeDist.z : MAX_DIST;
        modDist.w = input.planeDist.w > 0 ? input.planeDist.w : MAX_DIST;

        // find distance to the rear of the tetrahedron
        float fDist = min(modDist.x, modDist.y);
        fDist = min(fDist, modDist.z);
        fDist = min(fDist, modDist.w);

        // find the texture coords of the entrance point
        float3 texEnter;
        texEnter.x = dot(input.Gx,input.vPos-input.pos0) + input.Tex.x;
        texEnter.y = dot(input.Gy,input.vPos-input.pos0) + input.Tex.y;
        texEnter.z = dot(input.Gz,input.vPos-input.pos0) + input.Tex.z;

        // find the exit position
        float3 viewExitDir = normalize( input.vPos - g_vEyePt )*fDist;
        float3 viewExit = input.vPos + viewExitDir;

        // find the texture coords of the entrance point
        float3 texExit;
        texExit.x = dot(input.Gx,viewExit-input.pos0) + input.Tex.x;
        texExit.y = dot(input.Gy,viewExit-input.pos0) + input.Tex.y;
        texExit.z = dot(input.Gz,viewExit-input.pos0) + input.Tex.z;

        // March along the Texture space view ray until we either hit something
        // or we exit the tetrahedral prism
        float3 tanGrad = texExit - texEnter;
        tanGrad /= (MAX_STEPS-1.0);
        float3 TexCoord;
        float4 Norm_height;
        bool bFound = false;
        for (float i=0; i<MAX_STEPS; i+=1) {
                if (!bFound) {
                        TexCoord = texEnter + i*tanGrad;
```

```
                        Norm_height = g_txNormal.Sample(g_samPoint, TexCoord.xy);
                        float height = Norm_height.a;
                        if (TexCoord.z < height) {
                                bFound = true;
                        }
                }
        }
        if (!bFound)
                discard;

        // Move the light orientation into Texture space
        float3 lightDir = normalize(g_vLightPos - input.vPos);
        float3 TexLight;
        float3 nBiTanT = input.BiTanT;
        float3 TexLight;
        float3 nBiTanT = input.BiTanT;
        float3 nTanT = input.TanT;
        float3 nNormT = input.Norm;
        TexLight.x = dot(lightDir, nBiTanT);
        TexLight.y = dot(lightDir, nTanT);
        TexLight.z = dot(lightDir, nNormT);

        // unbias the Normal map
        Norm_height.xyz *= 2.0;
        Norm_height.xyz -= float3(1,1,1);

        // dot with Texture space light vector
        float light = saturate(dot(TexLight, Norm_height.xyz));

        // Get the Diffuse and Specular Textures
        float4 diffuse = g_txDiffuse.Sample(g_samLinear, TexCoord.xy);
        float specular = diffuse.a;

        // Move the viewDir into Texture space
        float3 viewDir = normalize(-viewExitDir);
        float3 tanView;
        tanView.x = dot(viewDir, nBiTanT);
        tanView.y = dot(viewDir, nTanT);
        tanView.z = dot(viewDir, nNormT);

        // Calculate specular power
        float3 halfAngle = normalize(tanView + TexLight);
        float4 spec = saturate(pow(dot(halfAngle, Norm_height.xyz), 64));

        // Return combined lighting
        return light*diffuse*kDiffuse + spec*kSpecular*specular;
}
```

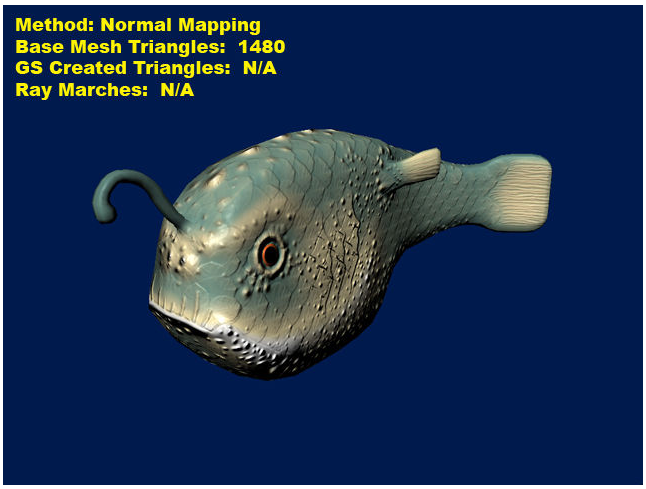
3.5.5 Problems

One problem with this technique is also a problem with most displacement mapping techniques. In areas of high concave curvature, the extruded prism can actually fold in on itself and flip. This causes the tetrahedra and resulting ray trace to be incorrect. The solution that the sample employs is to detect these cases and reduce the amount of displacement for the offending triangle.

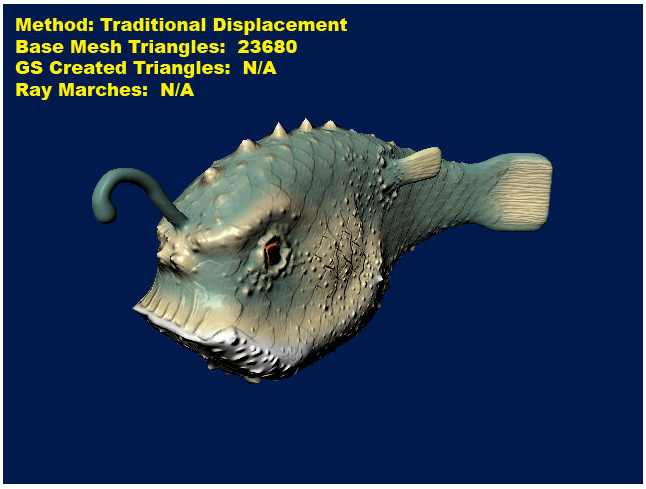
Another problem with this technique is sampling artifacts. If the number of steps per ray is too small, some high-frequency details can be missed. For very low sampling rates, this can cause details to pop in and out as the camera or object moves. Although geometry based displacement mapping also has frequency issues, the connectivity between the triangles ensure that there is always a smooth transition between different height values. Furthemore, geometry based methods are not dependent on the orientation of the view ray, so no popping of detail occurs when the object or camera moves.

3.5.6 Comparison with Other Techniques

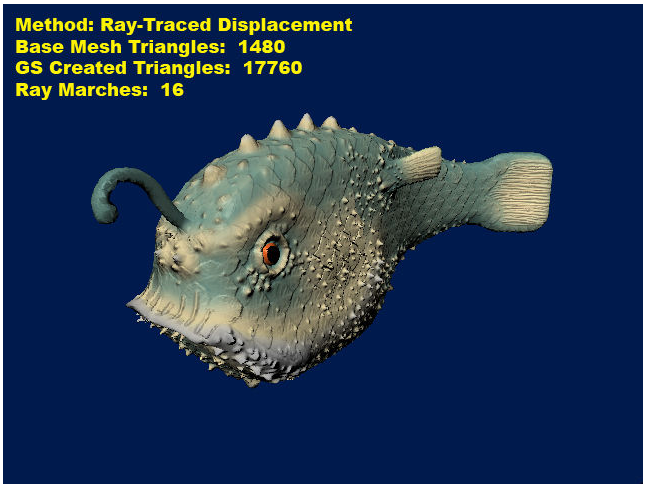
In this section, we compare traditional methods of achieving higher detail with the method this sample uses. The base mesh is 1480 polygons. For normal-mapping, the polygon count stays at 1480. For traditional displacement mapping, the mesh is uniformly tessellated to a high number of polygons and then the vertices are displaced according to the height of the displacement map. For ray-traced displacement (this sample), the mesh is converted into 3 tetrahedra (4 triangles each) in the geometry shader, such that 17760 triangles are actually passed to the pixel shader.



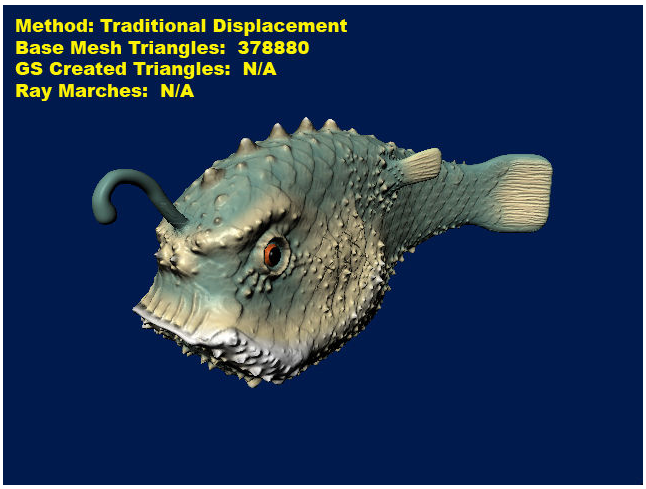
Notice the lack of self occlusion or definition on the silhouette of the object.



At 23680 triangles, the mesh still lacks some details. Notice the precision issues around the eyes.



This is a screenshot from the DisplacementMapping10 sample. It uses the ray traced displacement method described above. The eye ray is traced through the geometry shader created tetrahedra and intersected with the displacement map. This is from the base mesh of 1480 triangles that is expanded to 17760 triangles in the GS. Notice the level of detail compared to the traditional method using 23680 triangles.



This mesh was rendered using the traditional method of displacement by tessellating the mesh to 378880 triangles. Compare this to the image above rendered with 1480 (17760 triangles after the GS).

3.6 SparseMorphTargets Sample

This sample implements mesh animation using morph targets.

3.6.1 Morph Targets

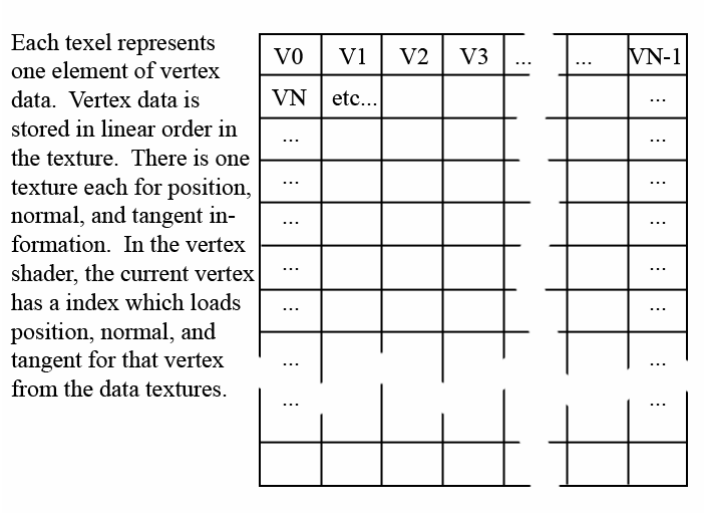
The SparseMorphTargets sample demonstrates facial animation by combining multiple morph targets on top of a base mesh to create different facial expressions.

Morph targets are different poses of the same mesh. Each of the minor variations below is a deformation of the base mesh located in the upper left. By combining different poses together at different intensities, the sample can effectively create many different facial poses.

3.6.2 How the Sample Works

Preprocessing

In a preprocessing step, the vertex data for the base pose is stored into three 2D textures. These textures store position, normal, and tangent information respectively.



This base texture is stored into the .mt file along with index buffer data and vertex buffer data that contains only texture coordinates. Then, for each additional morph target, the mesh is converted to three 2D textures. The base mesh textures are subtracted from the three morph target textures. The smallest subrect that can fully contain all of the differences between the two textures is then stored into the .mt file. By only storing the smallest texture subrect that can hold all of the differences between the two meshes, the sample cuts down on the storage requirements for the morph targets.

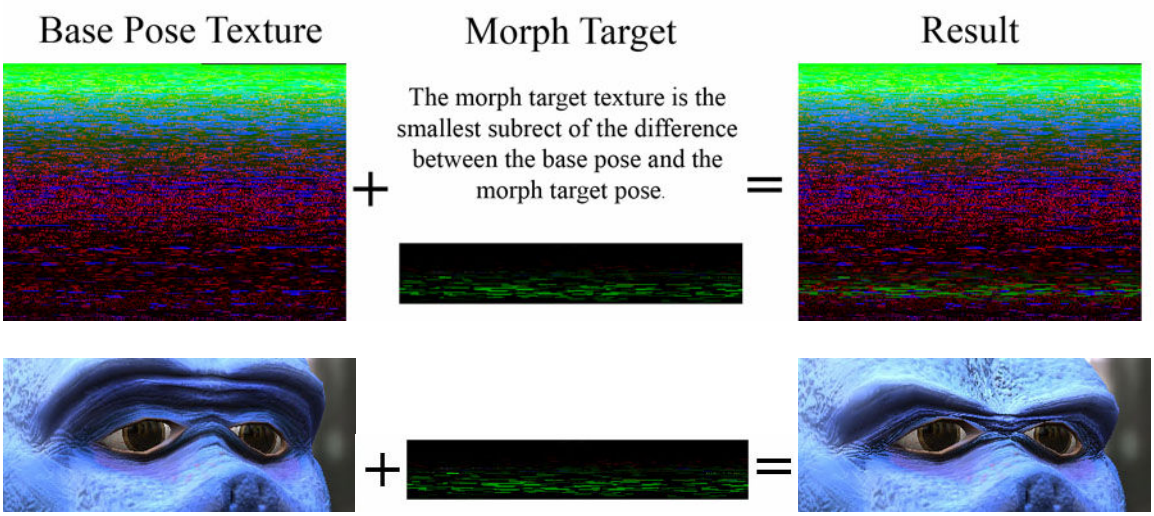
Applying Morph Targets at Runtime

Morph targets are handled in the MorphTarget.cpp and MorphTarget.h files. These contain classes that help with the application of the morph targets. To apply the different morph targets to the mesh, the sample sets up a texture2Darray render target with three array indices. The first holds position data, while the second and third hold normal and tangent information respectively.

The first part of the process involves filling the render targets with the position, normal, and tangent information from the base pose. Then for any morph targets that need to be added, the following occurs:

- The alpha blending mode is set to additive blending
- The viewport is set to the exact size of the pos, norm, and tangent render targets
- A quad covering only the pixels in the render target that will change for the given morph target is drawn
- This quad is drawn with the morph target position, normal, and tangent texture bound as a textured2darray
- A special Geometry Shader replicates the quad three times and sends one to each render target
- The Pixel Shader sets the output alpha to the blend amount. The blend amount is the amount of influence this morph target should have in the final image.

Effectively, the above uses the alpha blending hardware of the GPU to add up a series of sparse morph targets to a base mesh. The position, normal, and tangent render targets now contain the vertex data of the final deformed mesh. These are bound as inputs to the technique that renders the final morphed mesh onscreen.



Rendering

The input vertex stream contains a vertex reference that tells the VSRefScenemain shader which texel in the position, normal, and tangent maps contains our vertex data. In the shader, the uiVertexRef is converted to a 2D texture coordinate. The position, normal, and tangent data is then loaded from textures and transformed as they would be as if they had been passed in via the vertex stream.

```
struct VSSceneRefIn {
    uint uiVertexRef    : REFERENCE;
    float2 tex          : TEXTURE;
    float4 redblock     : COEFFSET0;
    float4 greenblock   : COEFFSET1;
    float4 blueblock    : COEFFSET2;
    float4 rgrest       : COEFFSET3;
    float2 brest        : COEFFSET4;
};
struct GSRefMeshIn {
    float4 pos          : SV_Position;
    float2 tex          : TEXTURE0;
    float3 wTan         : TEXTURE1;
    float3 wNorm        : TEXTURE2;
    float3 posOrig      : TEXTURE3;
    float4 redblock     : TEXTURE4;
    float4 greenblock   : TEXTURE5;
    float4 blueblock    : TEXTURE6;
    float4 rgrest       : TEXTURE7;
    float2 brest        : TEXTURE8;
};

GSRefMeshIn VSRefScenemain(VSSceneRefIn input) {
    GSRefMeshIn output = (GSRefMeshIn)0.0;

    // Find out which texel holds our data
    uint iYCoord = input.uiVertexRef / g_DataTexSize;
    // workaround for modulus // TODO: remove?
    uint iXCoord = input.uiVertexRef - (input.uiVertexRef/g_DataTexSize)*g_DataTexSize;
    float4 dataTexcoord = float4(iXCoord, iYCoord, 0, 0);
    dataTexcoord += float4(0.5,0.5,0,0);
    dataTexcoord.x /= (float)g_DataTexSize;
    dataTexcoord.y /= (float)g_DataTexSize;

    // Find our original position (used later for the wrinkle map)
    float3 OrigPos = g_txVertDataOrig.SampleLevel(g_samPointClamp, dataTexcoord);
    dataTexcoord.y = 1.0f - dataTexcoord.y;

    // Find our position, normal, and tangent
```

```
    float3 pos = g_txVertData.SampleLevel(g_samPointClamp, dataTexcoord);
    dataTexcoord.z = 1.0f;
    float3 norm = g_txVertData.SampleLevel(g_samPointClamp, dataTexcoord);
    dataTexcoord.z = 2.0f;
    float3 tangent = g_txVertData.SampleLevel(g_samPointClamp, dataTexcoord);

    // Output our final positions in clipspace
    output.pos = mul(float4(pos, 1), g_mWorldViewProj);
    output.posOrig = mul(float4(OrigPos, 1), g_mWorldViewProj);

    // Normal and tangent in world space
    output.wNorm = normalize(mul(norm, (float3x3)g_mWorld ));
    output.wTan = normalize(mul(tangent, (float3x3)g_mWorld ));

    // Just copy ldprt coefficients
    output.redblock = input.redblock;
    output.greenblock = input.greenblock;
    output.blueblock = input.blueblock;
    output.rgrest = input.rgrest;
    output.brest = input.brest;

    // Prop texture coordinates
    output.tex = input.tex;

    return output;
}
```

3.6.3 Adding a Simple Wrinkle Model

To add a bit of realism the sample uses a simple wrinkle model to modulate the influence of the normal map in the final rendered image. When rendering the final morphed image, the Geometry Shader calculates the difference between the triangle areas of the base mesh and the triangle areas of the current morphed mesh. These differences are used to determine whether the triangle grew or shrank during the deformation. If it shrank, the influence of the normal map increases. If it grew, the influence of the normal map decreases.

```
struct PSRefMeshIn {
    float4 pos          : SV_Position;
    float2 tex          : TEXTURE0;
    float3 wTan         : TEXTURE1;
    float3 wNorm        : TEXTURE2;
    float wrinkle       : WRINKLE;
    float4 redblock     : TEXTURE3;
    float4 greenblock   : TEXTURE4;
    float4 blueblock    : TEXTURE5;
    float4 rgrest       : TEXTURE6;
    float2 brest        : TEXTURE7;
};

[maxvertexcount(3)]
void GSCalcWrinkles(triangle GSRefMeshIn input[3], inout TriangleStream<PSRefMeshIn>
RefStream ) {
    // Find the area of our triangle
    float3 vortho = cross(input[1].pos-input[0].pos, input[2].pos-input[0].pos);
    float areaNow = length( vortho ) / 2.0;

    vortho = cross(input[1].posOrig-input[0].posOrig, input[2].posOrig-input[0].posOrig);
    float areaOrig = length( vortho ) / 2.0;

    for (int v=0; v<3; v++) {
        PSRefMeshIn output;
        output.pos = input[v].pos;
        output.tex = input[v].tex;
        output.wTan = input[v].wTan;
        output.wNorm = input[v].wNorm;
        output.redblock = input[v].redblock;
```

```
        output.greenblock = input[v].greenblock;
        output.blueblock = input[v].blueblock;
        output.rgrest = input[v].rgrest;
        output.brest = input[v].brest;

        float w = ((areaOrig-areaNow)/ areaOrig)*1.0;
        if (w < 0)
            w *= 0.005f;
        output.wrinkle = saturate( 0.3 + w );

        RefStream.Append(output);
    }

    RefStream.RestartStrip();
}
```

3.6.4 Adding Illumination Using LDPRT

Local Deformable Precomputed Radiance Transfer is used to light the head. The PRT simulation is done on the base mesh and the 4th order LDPRT coefficients are saves to a .ldprt file. The sample loads this file into memory and merges it with the vertex buffer at load time. The bulk of the LDPRT lighting calculations are handled by GetLDPRTColor. The current implementation only uses 4th order coefficients. However, uncommenting the last half of GetLDPRTColor, re-running the PRT simulation to generate 6th order coefficients, and changing dwOrder to 6 in UpdateLightingEnvironment may give a better lighting approximation for less convex meshes.

For a more detailed explanation of Local Deformable Precomputed Radiance Transfer, refer to the Direct3D 9 LocalDeformablePRT sample or the PRTDemo sample.

Acknowledgements

This material is drawn from a number of sources including contributions from the Microsoft DirectX development, sample, and documentation teams: Mike Toelle, Sam Glassenberg, John Ketchpaw, Iouri Tarassov, Brian Klamik, Jeff Noyle, Relja Markovic, John Rapp, Craig Peeper, Peter-Pike Sloan, Jason Sandlin, Shanon Drone, and Kris Gray.