1.1

Accessing and Modifying Topology on the GPU

Dean Calver

A vertex shader can only access a single vertex at a time; the hardware has a fixed function indexed triangle model that doesn't allow a vertex shader to manually do a vertex indirection. This seems to limit the extent that topology can be accessed and modified on the GPU. This article introduces several techniques that essentially bypass this limitation, allowing complex operations that involve accessing, and in some cases modifying, topology on the GPU.

The two major techniques are the related ideas of topology and vertex maps: texture maps that hold the mesh itself. The ability of vertex shaders to access video memory surfaces (textures) is the heart of the system and limits this technique to vertex shader 3.0 or higher. The article also discusses using the render to vertex map approaches to optimize many operations by caching data across shader executions. In some cases the vertex maps themselves are created on the GPU just prior to being used to generate the visuals.

Overview

Vertex shader 3.0 introduced vertex texturing—the ability of a vertex shader to access a texture for data. The canonical use is for displacement mapping, but the ability to use a texture as a 2D array allows for more exotic uses. Using this capability we can treat textures as a large, but slow memory pool, giving us the possibility of topology and vertex maps—texture maps that encode the actual physical mesh.

This article will discuss using this to construct shader programs to look at and modify the mesh topology at each vertex. Topology access requires the mesh structure to be uploaded into memory accessible at each vertex or pixel. This is either constant RAM or texture RAM. Constant RAM is fast but extremely limited in size. Texture RAM can be much larger, but has high latency in a vertex shader. However it's much cheaper in a pixel shader.

One of the more confusing aspects is that many of the standard names that D3D/OpenGL use for various parts of the pipeline are re-used for different usage's, i.e., vertex streams might hold indices and textures holding vertex data. To ease this it's sometimes helpful to drop the traditional view of the graphics pipeline and see it as a stream processor with different types of memory. Doing this and setting some standard sizes we can expect for vertex shader 3.0 hardware, we can see the real power of modern GPUs more clearly (see Figure 1.1.1).

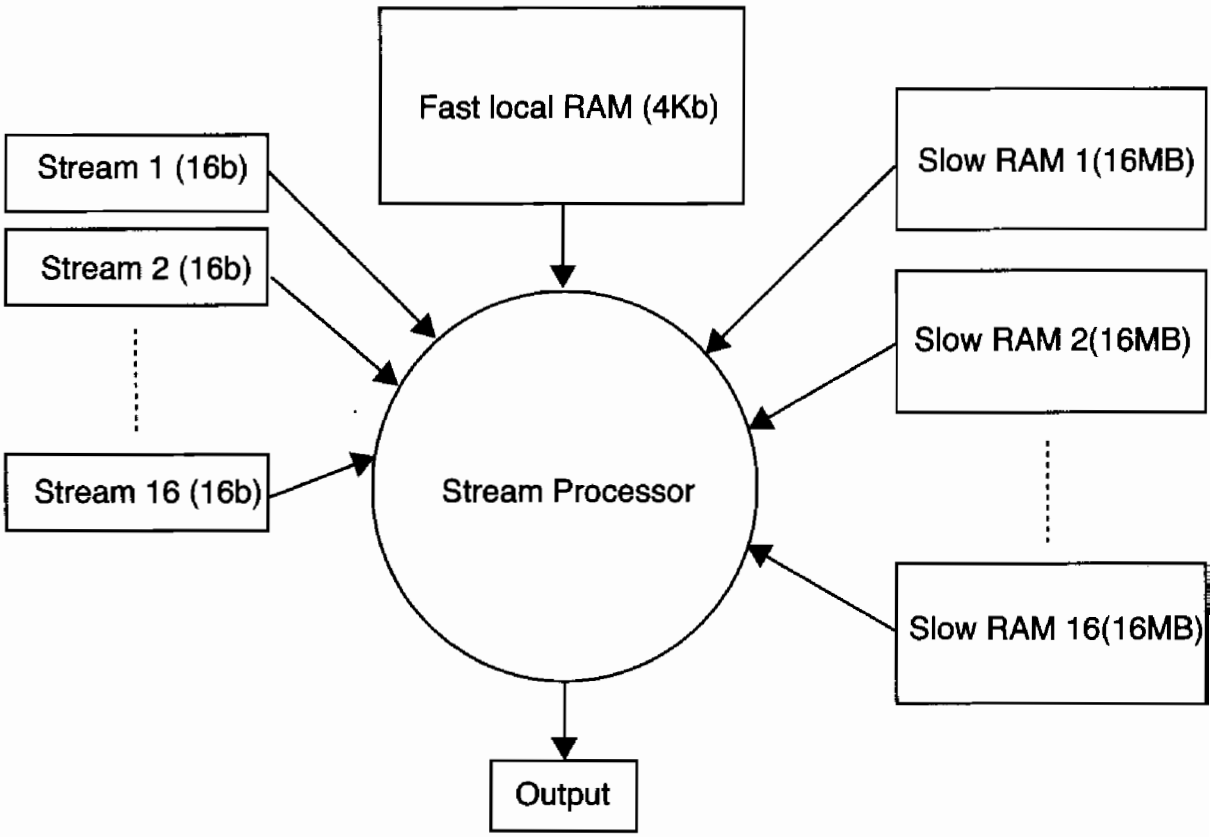

FIGURE 1.1.1 *Vertex Shader Model 3.0 as a Stream Processor.*

The estimated RAM sizes in Figure 1.1.1 are common memory configurations based on 256 vertex constants, 1024×1024 4-tuple float textures and 4-tuple float vertex streams.

The streams undergo a massive redeployment of function. They become your counters, primitives, or index data—in fact they become just about anything that can vary across the mesh. The RAM pools usually hold all of your vertex data and topology maps. This allows them to be read by both vertex and pixel shaders, and to be written to, by the rasteriser.

Implementation

Recreating What We Already Have

Before we can start doing complex operations we have to be able to handle the simple topology that standard meshes are made of.

An Indexed Triangle Mesh

First we need to encode the vertex data itself as vertex maps. Each texture can only hold a 4-tuple of data, and with current vertex shader 3.0, we only have 4 vertex texture samplers. But by treating textures as linear arrays we can encode more data at the expense of more texture look-ups and a complicated topology map.

Vertex Data As Vertex Maps

A simple vertex format might be position, normal and uv, represented as:

```
struct Vertex
{
        Vector3 position;
        Vector3 normal;
    Vector2 uv;
};
```

Without packing, this can be represented as three vertex maps each having a 4-tuples element. For high precision, the vertex map format is `D3DFMT_A32R32G32B32F`. The vertex maps are texture maps that hold the data accessed via vertex streams, but as textures they can be rendered to and even accessed in a pixel shader.

Vertex streams are 1D while textures are 2D or 3D. This requires us to rearrange our data. By using `D3DTADDRESS_WRAP` in the U dimension on the texture (which will effectively perform the modulus operation for free) we can do this function on the GPU as:

```
float2 GPUConvert1DAddressTo2D( float OneD )
{
    oneD = oneD * (1.f / TextureWidth);
    return( float2(OneD, OneD  * (1.f / TextureHeight) ) );
}
```

The texture width and height may be different for each vertex or topology map, so the actual function will probably take a `float2` constant per texture. It can also be optimized into a single multiple, so it's very cheap in practice.

Generally you want to use point sampling, so you get the exact vertex you're indexing, but you may be able to do some limited vertex tweening by using linear filtering.

Index Data As Vertex Streams

For an indexed triangle mesh, we have three indirections that point to the vertices this triangle is made of. Here we store this index data in a stream, but we could equally have placed it in a texture itself. We will discuss this use later when we encounter topology maps.

Depending on the number of vertices we need to index we can pick a variety of stream formats. But to replace the standard 16-bit index we can use `D3DDECLTYPE_SHORT4`.

Along with the per-primitive data, we need an indicator for the vertex we are currently working on. In this case this is a simple repeating 0, 1, 2 per triangle. As this data is repeating we can use the new vertex frequency functionality of vertex shader version 3.0 in order not to have a physical copy of this data for every primitive[1].

If the topology is continuous then vertices can be shared; however, if the topology (or the modifications) involves a discontinuous change then vertices will have to be duplicated. In other words, if the values calculated are the same for each vertex sharing a primitive regardless of the primitive it was calculated on, then you can share data. Otherwise you will need to have a unique vertex for each primitive.

For the simple case of replicating a standard indexed triangle mesh, the rules are exactly the same. Wherever the model is continuous in nD space (8D for the standard x, y, z, nx, ny, nz, u, v case) you can share data. It is worth noting that when we start modifying topology, sharing has to be considered post-modification. So for example, if you want to arbitrarily remove triangles you won't be able to share any vertex data.

One important thing to note is that in many cases you are dealing with floats pretending to be integers. In most cases this doesn't make much difference, but you have to alter all comparisons to accommodate. In the HLSL below we never use an equality (==) operator; instead we use < to achieve the same results.

Vertex Shader Primitive Assembly

The vertex shader retrieves the primitive data and vertex index from the stream. This is used to select the vertex data from the vertex maps, which is then processed as normal.

```
struct Vertex{
float3 position;
float3 normal;
float2 uv;
};

Vertex GetVertex( float3 primitiveData, float vertexIndicator )
{
   Vertex Data;

    float address1D = 0;
    // pick which of the 3 vertices of this triangle we are currently
processing
    if( vertexIndicator < 0.5 ) // vertexIndicator == 0
```

[1]At this writing the vertex frequency API is new and has a few "gotchas." Depending on what you want to do, you may have to expand the data fully, repeating the same data over and over, as there are some fairly simple things the current API doesn't allow.

```
    {
        address1D  = primitiveData.x;
    } else if( vertexIndicator < 1.5 )  // vertexIndicator == 1
    {
        address1D = primitiveData.y;
    } else if( vertexIndicator  < 2.5 )  // vertexIndicator == 2
    {
        address1D = primitiveData.z;
    }

    // convert vertex indice to texture UV
    // tex2Dlod takes a float4 with the w being the mip map, which is
always 0 for this article
    float4 texAddr = float4( GPUConvert1DAddressTo2D ( address1D ),
0, 0 );

    // grab the data from the vertex maps
    Data.position = tex2Dlod( posSampler,  texAddr ).xyz;
    Data.normal = tex2Dlod( normSampler,  texAddr ).xyz;
    Data.uv = tex2Dlod( uvSampler, texAddr ).xy;

    return Data;
}
```

Extending Things

By implementing indexed triangle mesh via this method, all we managed to do is to slow our rendering down. But now with this basic framework we can start accessing topology.

Topology Maps

We are used to thinking about texture maps holding color values, but as graphics hardware has advanced they are being used to hold arbitrary data. The leap to holding vertex data itself isn't that great, but the leap to storing the structure of the mesh itself requires closer scrutiny.

In a topology map, we use the texture to store indices and pointers to various parts of the mesh. A simple case is storing the standard triangle indices in a texture; a more complicated example is holding indices to the faces that surround each vertex.

By using textures to hold the mesh structure itself, we can perform operations on the GPU that usually would have to be done with a CPU. Any structure can be represented, but some (like linked lists) are very inefficient and should be avoided. Generally, we use arrays of indices. While this sometimes feels a bit alien to a generation brought up with C/C++ and its complex pointer types, this basic idea has been in use since the birth of high-level languages and it poses no real limitations. The only major problem is having to decide on a maximum array size for all data. If this poses a problem, an indexed-linked list can be constructed.

To demonstrate topology maps, we will calculate normals in the shader itself. Face normals are fairly trivial but smooth vertex normals are more complex as they

require not only data on the vertex and face we are currently rendering but also on the surrounding ones.

Calculating Face Normals in a Vertex Shader

Calculating face normals requires access to the vertices of the polygon being rendered. A vertex map will hold the vertex positions and a topology map will hold the structure of the mesh itself (we only need a fairly simple mesh structure for this function).

A triangle structure that can be used for face normal generation is:

```
struct TriangleFace
{
    int vertexIndex[3];
};
```

We simply store the vertex indices for this triangle in a texture. Then in the stream we store the triangle index of the triangle and the vertex indicator.

```
struct StreamElement
{
   int triangleIndex;       // 1D address of triangle face
   int vertexIndicator;     // 0,1,2
};
```

The triangle face topology map is looked up per vertex and then the vertex map is referenced to calculate the plane equation. The position is calculated in the same manner as before, but uses the triangle face indirection to obtain the vertex index instead of looking it up in the stream.

```
float3 CalcFaceNormal( float triangleIndex )
{
    float2 triIndex2D = GPUConvert1DAddressTo2D ( triangleIndex );
    // get vertex indices that make up this triangle
    float3 vertexIndices = tex2Dlod( TriangleFaceSampler, triIndex2D,
0 ).xyz;
    // convert indices in UV form
    float2 i0 = GPUConvert1DAddressTo2D(vertexIndices.x);
    float2 i1 = GPUConvert1DAddressTo2D(vertexIndices.y);
    float2 i2 = GPUConvert1DAddressTo2D(vertexIndices.z);

    // get the vertex position of all 3 vertices
    float3 v0 = tex2Dlod( PositionSampler, float4(i0,0,0) ).xyz;
    float3 v1 = tex2Dlod( PositionSampler, float4(i1,0,0) ).xyz;
    float3 v2 = tex2Dlod( PositionSampler, float4(i2,0,0) ).xyz;

    // calculate plane equation to get normal from
    return CalcPlaneEquationOf( v0, v1, v2 );
}
```

In practice, the types we can use hardware vertex texturing with are extremely limited on some hardware and you may have to use D3DFMT_A32B32G32R32F for all

your data. In the future, this hardware limitation is likely to become more relaxed, to the point that any texture you can use in the pixel shader can be used in the vertex shader.

Calculating Vertex Normals in a Vertex Shader

To generate smooth vertex normals we need the surrounding faces and vertices. In this case, accessing this data requires us to examine the vertex valency to see how much space we will need to allocate.

There is no actual limit to the vertex valency (each vertex can potentially have an unlimited number of faces attached), and while a linked list is possible in theory, a fixed-sized array is preferable. In practice, restricting the number of surrounding faces doesn't alter the visual result that much, as each face influences the normal, less than the greater, vertex valency. So, in this example, we limit the maximum vertex valency to 7, greatly simplifying the code.

The triangle face topology map is the same as for generating face normals, but now we add an extra topology map which has a list of the surrounding faces at each vertex in the model. There is a one-to-one mapping between the vertex map and vertex valency topology map, so we simply use the vertex indices to look up the vertex valency topology map.

```
#define MAX_VERTEX_VALANCY 7
struct vertexValency
{
   int valenceNum;
   int faceIndex[ MAX_VERTEX_VALANCY ];
};

#define SizeOfVertexValencyStructIn4Tuples 2

float3 CalcVertexNormal( float triangleIndex, float vertexIndicator )
{
    float2 triIndex2D = GPUConvert1DAddressTo2D ( triangleIndex );
    float3 vertexIndices = tex2Dlod( TriangleFaceSampler, triIndex2D,
0 ).xyz;

    float vertValencyAddr1D = 0;
    if( vertexIndicator < 0.5 )
    {
          vertValencyAddr1D  = vertexIndices.x;
    } else if( vertexIndicator < 1.5 )
    {
           vertValencyAddr1D  = vertexIndices.y;
    } else if( vertexIndicator  < 2.5 )
    {
            vertValencyAddr1D  = vertexIndices.z;
    }

    float2 index = GPUConvert1DAddressTo2D( vertValencyAddr1D );
    index = index *  SizeOfVertexValencyStructIn4Tuples;
```

```
    // get faces indices that surround this vertex
    vertexValency VVStruct;
    float4 tmp0 = tex2Dlod( VertexValencySampler, float4( index, 0,
0) ).xyzw;
    float4 tmp1 = tex2Dlod( VertexValencySampler, float4( index +
float2(1,0), 0, 0) ).xyzw;
    // copy indices into area accessable form
    VVStruct.valenceNum = tmp0.x;
    VVStruct.faceIndex[0] = tmp0.y;
    VVStruct.faceIndex[1] = tmp0.z;
    VVStruct.faceIndex[2] = tmp0.w;
    VVStruct.faceIndex[3] = tmp1.x;
    VVStruct.faceIndex[4] = tmp1.y;
    VVStruct.faceIndex[5] = tmp1.z;
    VVStruct.faceIndex[6] = tmp1.w;

    // accumulate face normals
    float3 vertexNorm(0,0,0);
    int curFaceIndex = 0;
    while( curFaceIndex < VVStruct.valenceNum )
    {
        vertexNorm += CalcFaceNormal( VVStruct.faceIndex[curFaceIn-
dex] );
        curFaceIndex++;
    };

    return Normalize( vertexNorm );
}
```

As you will notice, this is very expensive, but there are optimizations that can speed it up. The most important uses the fact that the vertex maps are just textures and can be generated on the fly by the GPU. But before we get to optimizations lets explore some other examples of uses for these techniques.

Removing Triangles

Removing a triangle using a topology map is trivial. The only caveat is that you can't share vertices between triangles, because to remove a triangle each of its vertices must be placed into a position the clipper will reject. This requires no sharing with a triangle that is visible. The `triangle kill` command just places a clip space vertex position that definitely will be clipped (for example, $w = -1$) for each vertex in the triangle and the triangle will be removed.

This can be used to easily remove groups of polygons (i.e., arms, heads, etc.). We give each primitive a group `id`, and then a constant list of whether or not each group is processed.

```
struct TriangleFace
```

```
{
    int vertexIndex[3];
    int groupID;
};
```

In the stream we store the triangle index of the triangle and the vertex indicator:

```
struct StreamElement
{
   int triangleIndex;        // 1D address of triangle face
   int vertexIndicator;      // 0,1,2
};
```

and then a quick check and dynamic branch will avoid processing too much code for removed triangles.

```
float TriangleGroups[ MAX_GROUPS ]; // bool as float
(false = 0.f, true = 1.f)

Output TriKiller( float vertexIndicator )
{
    float2 triIndex2D = GPUConvert1DAddressTo2D ( triangleIndex );
    float4 triangleFace = tex2Dlod( TriangleFaceSampler, float4( tri-
Index2D, 0 ,0) ).xyzw;
    if( TriangleGroups[ triangleFace.w ] > 0.5 ) // true
   {
        // process vertex as normal
   } else
   {
         // group is invisible kill the vertex
         // set the output to somewhere the clipper considers off
screen
         Output.position = float4(-1,-1,-1,-1);
   }
}
```

Sprite-Based Quads

You want to render a crazy particle system (or old school 2D shooter), that is limited by the speed you can get the data from CPU to GPU. Point sprites aren't good enough because the no-rotation limitation doesn't work in your situation. Is there a way to reduce bandwidth requirements to the GPU by using topology maps?

For this we use the stream input as a primitive stream not a vertex stream—the user supplies no data per vertex, just the sprite data itself. Also we use the fast local memory (constant) to reduce bandwidth into the vertex shader to an absolute minimum.

The only per-vertex information is the vertex indicator field (which, because we are using quads, now has values of 0, 1, 2, and 3). The vertex frequency API allows us to repeat this data for every primitive, reducing the actual external GPU bandwidth to

zero per vertex (the stream is so small it will easily fit into pre-vertex shader cache on the GPU).

We set up a sprite structure (this is obviously application specific, the one described here is the one used in the sample code on the disk).

```
struct SpritePrim
{
    float16 Position[2];      // 16 bit float is enough range for
                              // this sprite system
    byte    SpriteNum;        // Index to look-up size and texture data
    byte    Rotation;         // byte encoded rotation parameter
    byte    ScaleHi;          // scale is encoded as 8.8 fixed point
                              // number integer byte
    byte    ScaleLo;          // fraction byte
};
```

This is placed in stream 0, and stream 1 simply consists of the numbers 0, 1, 2, and 3. The vertex frequency system is then programmed to stream 0 only every fourth vertex and stream 1 to update every vertex but repeat after every fourth. We do this by using each primitive as an instance. But instancing changes the way we count, so the number of sprites we want to render is passed to `SetStreamSourceFreq` for the instanced parameter.

For DirectX 9.0c this is enabled via:

```
D3DDevice->SetStreamSourceFreq( 0, numSprites | D3DSTREAMSOURCE_
INDEXEDDATA );
// update the data updated every vertex repeated numSprites times
D3DDevice->SetStreamSourceFreq(1, 1 | D3DSTREAMSOURCE_INSTANCEDATA );
// update stream 1 per instance (2 triangles in this case)
```

then we just call `DrawPrimitive` with the number of triangles of 2 (2 triangles per quad).

In the actual vertex shader, instead of placing the sprite size and texture data in a texture, we place it in constant memory. This saves us bandwidth because constant memory is much faster to access.

Render to Vertex Map

Having the mesh as a texture allows us to use "render-to-vertex" techniques. These techniques use the rasteriser itself to modify mesh data. This allows caching of complex operations across multiple invocation of the complex operations.

Speeding Up Vertex Normal Generation

Previously, when calculating the vertex normal we had to call `GetFaceNormal()` multiple times per vertex, recalculating the face normal for every surrounding face. Using render to vertex we can do a single pass over the faces calculating the face normals, which can then be quickly accessed per vertex.

We need an intermediate face normal surface that we will calculate when the face normal changes. As this is normal data we can probably use an 8-bit integer format. Of course, if the precision isn't enough for your application, you can use a higher precision format.

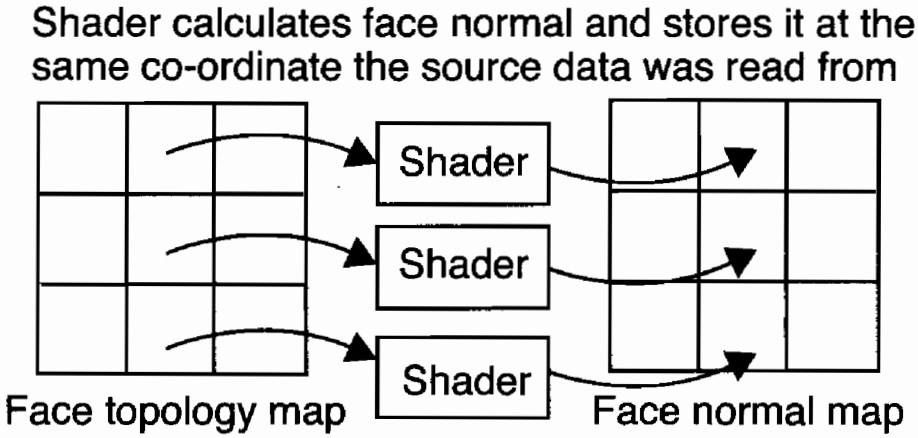

FIGURE 1.1.2 *Positioning in the face normal map.*

We render quads that cover each face in the triangle face topology map, rendering one-to-one with the face normal map. The shader accesses each face one by one, calculates the face normal, and then outputs it into the same position in the face normal map (see Figure 1.1.2).

But why stop at just the face normals? If the reason we are calculating vertex normals is something that can be cached, we can continue with this technique until the actual object renderer is fairly conventional (and fast). Just perform the complex topology operations when the cache becomes invalid. This approach is especially relevant where normals change infrequently. The actual cost is mainly contained in the update functions when the vertex used is low.

To do this we add another pass where every vertex is mapped one-to-one with a vertex normal texture. A quad is rendered that takes the vertex input, calculates the vertex normal, and outputs it into the vertex normal texture. Using the massively parallel pixel shader to process multiple vertex normals we can perform the job significantly faster than with most other approaches to the problem.

Putting It All Together for a Simple Terrain Renderer

To demonstrate the various techniques we use a completely GPU-modifiable terrain system, with datalike normals being calculated when the mesh changes. The terrain position itself is represented by a displacement map (note: the method of rendering

displacement maps isn't discussed here; see [Calver02] and [Calver03] for details) but it can easily be a full 3D world position (it's a displacement map more for the terrain modification algorithm than anything). This doesn't take advantage of any inherent topology from the displacement map, but it is advisable in real code (in a real displacement map renderer you can calculate the face indices directly).

The rendering is split into two portions, one is called when the mesh topology changes (i.e., the terrain is damaged), the other is called to actually render the terrain itself. We set up the data so that the terrain render is fairly fast with the expensive work taking place only when the mesh changes.

The actual terrain render is fairly simple: the vertex data is split between static and dynamic. Static data comes in via a standard vertex stream; in the example this is just the terrain texture *u,v* and the vertex index. The dynamic data is the dynamic vertex maps stored in textures and is computed by the update code.

Terrain Render Code:

```
SetRenderTarget( BackBuffer );
SetTexture( 0,  VertexPositionSurface  );
SetTexture( 1, VertexNormalSurface );
SetPixelShader( TerrainRenderVS );
RenderMesh();

// HLSL Shader vs_3_0
Struct RenderVSInput
{
     float2    texUV : TEXCOORD0;
     float    vertexIndex : TEXCOORD1;
};
struct RenderVSOutput
{
     float4 HclipPos : POSITION;  // position for the rasterisor
     float2 texUV : TEXCOORD0;    // pass the old fashioned texture
                                     // coordinate
     float3 normal : TEXCOORD0;   // pass the world space normal
                                     // through to the pixel shader
};
sampler2D sampPositionVec;
sampler2D sampNormalVec;
void TerrainRenderVS( in RenderVSInput inp,
      out RenderVSOutput outp )
{
    float4 vertUV =  float4( GPUConvert1DAddressTo2D( inp.vertexIn-
dex), 0, 0 );
    float3 position = tex2Dlod( sampPositionVec,  vertUV ).xyz;
    float3 normal = tex2Dlod( sampNormalVec,  vertUV ).xyz;
    outp.HclipPos = mul( float4(position,1), transformMatrix );
    outp.normal = normal;
    outp.texUv = inp.texUV;
}
```

The real work occurs in the update pass where we create two render-targets textures for position and normal maps. We also need the displacement map (also a render-target so we can modify it via the GPU) and a couple of topology maps hooking everything together. We also need a temporary render-target to hold the face normals (this isn't currently used outside the update cycle but potentially it could be used to optimize terrain damage).

The topology maps are basically the same as we previously used, one face to vertex map and the other encoding surrounding faces per vertex. For the displacement map grid we use a fixed vertex valancy of 6, as in the majority of cases this will be true (only edges of the map have a vertex valency not equal to 6).

```
// face to vertex map
struct TriangleFace
{
    int vertexIndex[3];
};

// vertex to surrounding faces
struct vertexValancy
{
   int faceIndex[ 8 ];
};
```

When the displacement maps are altered we need to update the vertex maps for the renderer. This is done in three passes: the first calculates the vertex positions from the displacement map, the second calculates face normals, and the third finally calculates the vertex normals.

First Pass: Vertex Positions

This is an easy one: we simply read each displacement value and output the world space position.

```
SetRenderTarget( VertexPositionSurface );
SetTexture( 0, DisplacementMap );
SetPixelShader( VertexPositionGenerator );
RenderFullSurfaceQuad();

// HLSL Shader ps_3_0
sampler2D sampDisplacementMap;
float4 VertexPositionGenarator( in float2 uv ) : COLOR0
{
  float displacement = tex2D( sampDisplacementMap, uv ).xyz;
  return WorldPositionFromDisplacement( displacement );
}
```

Second Pass: Face Normals

The face normal generator takes each `TriangleFace` structure and outputs the face normal. The vertex stream contains a u,v coordinate matching the source and destination pixels.

```
SetRenderTarget( FaceNormalMap );
SetTexture( 0, TriangleFaceTopoMap );
SetTexture( 1, VertexPositionSurface );
SetPixelShader( FaceNormalGenerator );
RenderFullSurfaceQuad();

// HLSL Shader ps_3_0
sampler2D sampTriangleFace;
sampler2D sampVertexPosition;
float4 FaceNormalGenerator( in float2 uv ) : COLOR0
{
    float3 vertexIndices = tex2D( sampTriangleFace, uv ).xyz;
    float2 i0 = GPUConvert1DAddressTo2D(vertexIndices.x);
    float2 i1 = GPUConvert1DAddressTo2D(vertexIndices.y);
    float2 i2 = GPUConvert1DAddressTo2D(vertexIndices.z);

    float3 v0 = tex2D( sampVertexPosition, i0 ).xyz;
    float3 v1 = tex2D( sampVertexPosition, i1 ).xyz;
    float3 v2 = tex2D( sampVertexPosition, i2 ).xyz;

    return Normalise( CalcPlaneEquationOf( v0, v1, v2 ) );
}
```

Third Pass: Vertex Normals

The vertex normal generators take each vertex and sum the surrounding faces (the edge cases can be handled in various ways and will be ignored here for clarity) before normalizing the normal. The use of a fixed vertex valency with a pixel shader process greatly simplifies the code. One extra complication over the last few passes is that the vertex valency structure takes two texture look-ups, so the `UV`s have to be adjusted.

```
SetRenderTarget( VertexNormalSurface );
SetTexture( 0, VertexValencyTopoMap );
SetTexture( 1, FaceNormalMap );
SetPixelShader( VertexNormalGenerator );
RenderFullSurfaceQuad();

// HLSL Shader ps_3_0
sampler2D sampFaceNormal
sampler2D sampVertexValency;
float4 VertexNormalGenerator( in float2 uv ) : COLOR0
{
    float4 vv0 = tex2D( VertexValancySampler, expUV ).xyzw;
    float4 vv1 = tex2D( VertexValancySampler, expUV +
float2(OneTexel,0) ).xyzw;
    float3 vertexNorm;
```

```
    vertexNorm = tex2D(  sampFaceNormal, GPUConvert1DAddressTo2D(
vv0.x ) );
    vertexNorm += tex2D( sampFaceNormal, GPUConvert1DAddressTo2D(
vv0.y ) );
    vertexNorm += tex2D( sampFaceNormal, GPUConvert1DAddressTo2D(
vv0.z ) );
    vertexNorm += tex2D( sampFaceNormal, GPUConvert1DAddressTo2D(
vv0.w ) );
    vertexNorm += tex2D( sampFaceNormal, GPUConvert1DAddressTo2D(
vv1.x ) );
    vertexNorm += tex2D( sampFaceNormal, GPUConvert1DAddressTo2D(
vv1.y ) );

    return normalise( vertexNorm );
}
```

Conclusion

The technique shows that Pixel Shader 3.0 is almost a complete graphics pipeline capable of techniques well beyond the classic vertex- and pixel-level effects. By using Topology Maps and Vertex Maps, we have fundamentally changed the classic idea of how meshes are rendered and processed by GPUs. The use of the pixel shader hardware as a method for mesh manipulation is something that will be used a lot in the future as we move to treating the GPU as another general purpose processor.

The examples presented here are mainly illustrative, and in many cases (like the terrain renderer) they can be mixed with more conventional rendering techniques for faster throughput. In practice, losing the cost of vertex texture fetches isn't easy but real rendering generally has more optimization potential than the simple example code here. In the terrain example, the `UV` coordinates could have had a texture matrix applied while waiting for one of the fetches to finish.

The techniques demonstrated here are merely the tip of the iceberg. Some obvious future techniques that could benefit from these techniques are caching of skinning data, advanced animation techniques, and subdivision surfaces. There are probably many other systems, that with future development, will benefit from the unification of topology, vertex, and pixel data.

References

[Calver02] D. Calver, "Vertex Decompression Using Vertex Shaders" in *ShaderX*, Engel, Wolfgang, ed., Wordware, May 2002.

[Calver03] D. Calver, "Vertex Decompression Using Vertex Shaders Part 2" in *ShaderX*^{2}, Engel, Wolfgang, ed., Wordware, August 2003.