Vertex Decompression in a Shader

Dean Calver

Introduction

Vertex compression can help save two valuable resources, memory and bandwidth. As you can never have enough of either, compression is almost always a good thing. The main reasons to not use compression are if the processing time to decompress is too high or if the quality is noticeably lower than the original data (for lossy compression schemes). The methods described here are designed to avoid both problems with reasonable compression ratios.

Graphics literature has a wide range of vertex compression schemes, but most of these require a complex decompression step that cannot be executed inside a vertex shader.

Vertex Compression Overview

Our aim is to get the entire decompression step in the vertex shader itself. This rules out many of the standard compressors that require either large compression tables or access to other vertices (predictor methods). The amount of data we can use for the decompression algorithm (almost all vertex shader constant space may already be used, i.e., skinning matrices, etc.) is very limited. We generally have a few per-object constants to upload into vertex constant memory.

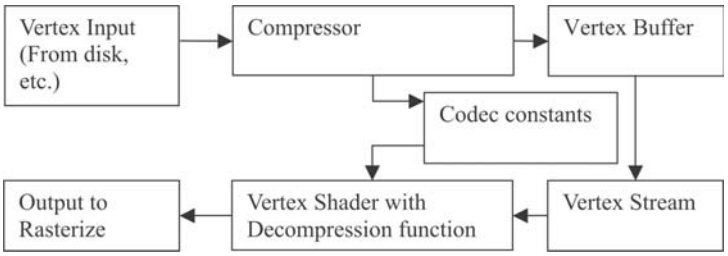

Figure 1: Vertex compression block diagram

A number of techniques can be used inside a vertex shader. These include:

- Quantization — The simplest form of compression; we simply lose the least significant bits.

- Scaled offset — Actually a special case of linear equations method, but limiting the equation to a scaled offset from a point.
- Compression transform — A method that calculates linear equations to transform by where the variables (the vertex values) need less precision.
- Multiple compression transforms — Each vertex selects from a palette of compression transforms.
- Sliding compression transform — Extends the precision of the compression transforms by putting spare padding bytes to good use.
- Displacement mapping and vector fields — Only for special kinds of data, but by storing geometry like a texture, we can save space.

All these compression methods rely on the same basic technique — storing the vertex components in fewer bits. Floating-point is designed with a huge dynamic range; usually we don't need much so we can save space. DirectX helps us by having a number of non-floating-point data types that are converted to floating-point before entry into the vertex shader.

Vertex Shader Data Types

As of DirectX 8.1, there are four different data types for us to work with (excluding the float data types):

- D3DVSDT_UBYTE4 — Four unsigned 8-bit integers that map in the vertex shader in the range 0.0 to 255.0.
- D3DVSDT_D3DCOLOR — Four unsigned 8-bit integers that map in the vertex shader in the range 0.0 to 1.0. Also, the mapping of each component into the vertex shader register is different from D3DVSDT_UBYTE4.
- D3DVSDT_SHORT2 — Two signed 16-bit integers that map in the vertex shader in the range –32767.0 to +32768.0.
- D3DVSDT_SHORT4 — Four signed 16-bit integers that map in the vertex shader in the range –32767.0 to +32768.0.

Some video cards don't support D3DVSDT_UBYTE4, but this isn't a problem in practice since D3DVSDT_D3DCOLOR can be used as a replacement with a few minor modifications. The two changes needed are accounting for the change in range (the vertex shader code has to multiply the incoming values by 255.0) and swapping the vertex stream components so they appear in the correct places when used. This can be achieved by either changing the input data (i.e., swapping the data in the vertex buffer and multiplying a transformation matrix by 255.0) or adding the following vertex shader code:

```
; v1 = D3DVSDT_D3DCOLOR4 to be used a D3DVSDT_UBYTE4
; c0 = <255.0, 255.0, 255.0, 255.0>
mul r0.zyxw, v1, c0       ; multiply color4 by 255 and swizzle
```

If we are going to use these types instead of standard float vector, then we are going to have to insert dummy pad values for odd vector sizes. The pad values show up a lot with vertex data because most components (positions, normals, etc.) are 3D vectors. Even with pad spaces, the new types are much smaller than floats, and often we can use these pad values to store other data.

As different vertex components have different numerical ranges and requirements, it is quite common to have varying compression methods in a single stream. We set up our stream declarations with the new types and fill vertex buffers with the packed values.

Compressed Vertex Stream Declaration Example

A lit vertex with a single uv channel format before compression:

```
D3DVSD_STREAM(0),
D3DVSD_REG( 0, D3DVSDT_FLOAT3),   // position
D3DVSD_REG( 1, D3DVSDT_FLOAT3),   // normal
D3DVSD_REG( 2, D3DVSDT_FLOAT2),   // texture coordinate
D3DVSD_END()
```

After compression, the position is stored in three signed 16-bit integers, the normal is stored in three unsigned 8-bit integers, and the texture coordinate is stored in two signed 16-bit integers. We have a signed 16-bit integer and an unsigned 8-bit integer that are unused (padding). The original size is 32 bytes, and the compressed size is 16 bytes, so we have a 50% savings.

```
D3DVSD_STREAM(0),
D3DVSD_REG( 0, D3DVSDT_SHORT4),   // position
D3DVSD_REG( 1, D3DVSDT_UBYTE4),   // normal
D3DVSD_REG( 2, D3DVSDT_SHORT2),   // texture coordinate
D3DVSD_END()
```

Basic Compression

Several simple forms of vertex compression are discussed in this section.

Quantization

The simplest form of vertex compression is to store fewer bits; as there is no floating-point format of less than 32 bits, this also implies a shift to integer or fixed-point data representation.

This has problems with almost all data sets but is a reasonable solution for data with a very limited dynamic range. The obvious candidates among standard vertex components are normalized values (normals and tangents are usually normalized for lighting calculations).

Compression

I choose a policy of clamping to a minimum and maximum range of the data type, but if this happens, we shouldn't use quantization as a compression method.

We also have to choose how many bits to use for the integer part and how many for the fractional part. This is very data-sensitive because there is no reasonable default with so few bits in total.

```
fracScale = number of bits for fractional values

Quantise( originalVal )
{
    return ClampAndFloatToInteger ( originalVal * fracScale );
}
```

Decompression

Decompression just reverses the compression process, taking the compressed values and restoring any fractional part.

```
Constants
{
    fracScale = number of bits for fractional vales
}
Decompress( compressedVal )
{
    return IntegerToFloat( compressedVal ) / fracScale
}
```

Practical Example

Normalized normals or tangents are usually stored as three floats (D3DVSDT_FLOAT3), but this precision is not needed. Each component has a range of –1.0 to 1.0 (due to normalization), ideal for quantization. We don't need any integer components, so we can devote all bits to the fractional scale and a sign bit. Biasing the floats makes them fit nicely into an unsigned byte.

For most normals, 8 bits will be enough (that's 16.7 million values over the surface of a unit sphere). Using D3DVSDT_UBYTE4, we have to change the vertex shader to multiply by 1.0/127.5 and subtract 1.0 to return to signed floats. I'll come back to the optimizations that you may have noticed, but for now, I'll accept the one cycle cost for a reduction in memory and bandwidth saving of one-fourth for normals and tangents.

Quantization vertex shader example:

```
; v1 = normal in range 0 to 255 (integer only)
; c0 = <1.0/127.5, -1.0, ???? , ????>
mad  r0, v1, c0.xxxx, c0.yyyy  ; multiply compressed normal by 1/127.5, subtract 1
```

Scaled Offset

This method produces better results than quantization by redistributing the bits to a more reasonable range for this object. By choosing a scale that covers the entire object, we ensure the object is correct to the accuracy of the quantization level.

A good candidate is to use the axis-aligned bounding box (AABB) as scale and translation vector. Each vector is defined in object space, so we use the entire compressed range to cover the AABB. This requires us to translate and then scale the input vectors. You can use either the center of the AABB or a corner. I usually use a corner, which gives each compressed vertex a range of 0.0 to 1.0.

Compression

For static objects, we determine the minimum and maximum values for each axis and use these to calculate the offset point and scales. For dynamic objects, we have to choose the maximum and minimum that will encompass all dynamic changes. We maximize our dynamic range by allowing separate scale and offset per axis.

```
DetermineScaleAndOffset()
{
    For Every Vertex
        LowerRange  = Minimum (LowerRange, Vertex)
        UpperRange  = Maximum (UpperRange, Vertex)
}

offset = LowerRange
scale = (UpperRange – LowerRange)

ScaledOffset( originalVal )
{
    scaledOffset =  (originalVal – offset) / scale;
    return FloatToInteger (scaledOffset );
}
```

Decompression

The incoming values are in the range 0.0 to 1.0. We simply have to multiply by the scale value and add the offset to get the original vertex coordinate back.

```
Constants
{
    scale = scale used in the compression
    offset = corner used in the compression
}

Decompress( compressedVal )
{
    return (IntegerToFloat( compressedVal ) * scale) + centroid
}
```

Practical Example

Texture coordinate data is a good candidate for this form of compression. 8 bits is usually too low for texture data (only enough for 1 texel accuracy with a 256x256 texture), but 16 bits is often enough (unless tiling textures often). We apply the compression and then upload the scale and offset constants into vertex constants; the scale is itself scaled by 65,535, and the offset has an additional 0.5 to convert the short back into the 0.0 to 1.0 range. We use a mad instruction to perform the decompression, so it costs us one cycle but saves us 50% in size.

Scaled offset vertex shader example:

```
; v1 = uv texture coordinate in the range -32767.0 to 32768.0 (integer only)
; c0 = <u scale / 65535.0, v scale / 65535.0, u offset + 0.5, v offset + 0.5, >
mad r0.xy, v1.xy, c0.xy, c0.zw ; multiply uv by scale and add offset
```

Compression Transform

This is the superset of the previous two compression methods. We transform our input data into a space that compresses better. With the scaled offset method, we took our uncompressed data and applied a scale and offset. This can be represented by applying a scale matrix and a translation matrix to our input vector. This method further extends the idea by applying another linear transform (i.e., a matrix).

To see why this method can produce better results than the other two, look at the case in Figure 2. We have a simple two-polygon square that is oriented at 45 degrees (A). After scaled offset compression, we might be left with (B) which has clearly been distorted by the loss of precision. While higher precision would reduce the error, we may not wish/be able to spend more precision on it. Looking at (C), it is clear that if the object was rotated by 45 degrees, the existing precision would produce an exact result.

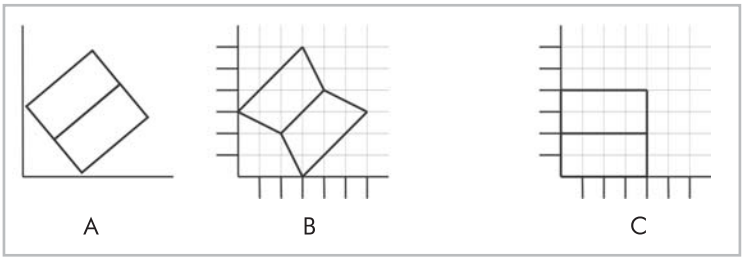

Figure 2: A square quantized in both the rotated and non-rotated case

The compression stage mainly consists of finding a transformation that will minimize the error when quantization occurs. The method I use is based on finding an optimal oriented bounding box for the vertices and using the basis matrix of this as my compression space. While it is possible to find better compression matrices for some objects (this method doesn't consider transforms such as shears), for most objects it finds a near-optimal linear quantization space. The compression space I usually use consists of a rotate, a scale, and a translation (a standard affine transformation), but in theory, any linear transformations could be used. A similar idea is used in image compression systems, such as MPEG, that use the DCT transform (a DCT is essentially an 8D rotation matrix).

Compression

Given a set of vertices, we want to find a compression space that minimizes the artifacts caused by the quantization. The problem has parallels in the collision detection field of trying to find an optimal oriented bounding box, so I use algorithms developed there to compute the compression transformation. It uses the vertex data from all triangles to set up a covariance matrix that we then extract the eigenvectors from.

This gives us a basis system that is aligned along the dominant axes of the data. Combined with the correct scale and translation, this gives excellent results for almost all data. From a geometrical point of view, compression transforms each input vector by the OBB basis space and then translates it so the minimum vertex in AABB space becomes the vector <0.0, 0.0, 0.0> and scales it so the maximum vertex becomes <1.0, 1.0, 1.0>.

```
CalculateRotationMatrix()
{
    n = Number of triangles
    m = vector size of data to be compressed
    p^i = 1st vertex of triangle i
    q^i = 2nd vertex of triangle i
    r^i = 3rd vertex of triangle i

      1/3n  Σ_{i=0}^{n}  p^i   q^i   r^i
```

$\overline{p^i}$ p^i

$\overline{q}^i$ q^i

$\overline{r}^i$ r^i

R_{jk} $\frac{1}{3n}\sum_{i\ 0}^{n}$ $\overline{p}_j^i$ $\overline{p}_k^i$ $\overline{q}_j^i$ $\overline{q}_k^i$ $\overline{r}_j^i$ $\overline{r}_k^i$ 1 j, k m

```
ExtractNormalizedEigenvectors(R)
}

DetermineCompressionMatrix()
{
    CalculateRotationMatrix();

    For Every Vertex
        v' = R^-1 * v
        LowerRange = Minimum (LowerRange, v')
        UpperRange = Maximum (UpperRange, v')

    O = TranslationMatrix( LowerRange )
    S = ScaleMatrix( UpperRange - LowerRange )

    C = R^-1 * O^-1 * S^-1
}

CompressionTransform( originalVal )
{
    v = originalVal
    v" = C*v
    return FloatToInteger(v")
}
```

Decompression

We transform the incoming values by the decompression matrix. This has the effect of transforming the compressed vector from compression space back into normal space.

```
Constants
{
    D = C^-1
}

Decompress( compressedVal )
{
    return (IntegerToFloat( compressedVal ) * D)
}
```

Practical Example

This compression method is ideal for position data, except for huge objects. Using 16-bit data types gives 65,535 values across the longest distance of the object; in most cases, it will provide no visual loss in quality from the original float version. We have to compensate for the data type range by changing the scale and translation matrix (same as scaled offset) and then

transpose the matrix (as usual for vertex shader matrices). This gives us a 25% savings in vertex size for four cycles of vertex code.

Compression transform shader example:

```
; v1 = position in the range -32767.0 to 32768.0 (integer only)
; c0 - c3 = Transpose of the decompression matrix
m4x4 r0, v1, c0    ; decompress input vertex
```

Using the same data stream from the earlier example, we can now see what the vertex shader might look like.

Compressed vertex example:

```
    D3DVSD_STREAM(0),
    D3DVSD_REG( 0, D3DVSDT_SHORT4),   // position
    D3DVSD_REG( 1, D3DVSDT_UBYTE4),   // normal
    D3DVSD_REG( 2, D3DVSDT_SHORT2),   // texture coordinate
    D3DVSD_END()
```

Compressed vertex shader example:

```
; v1.xyz = position in the range -32767.0 to 32768.0 (integer only)
; v2.xyz = normal in the range 0.0 to 255.0 (integer only)
; v3.xy = uv in the range -32767.0 to 32768.0 (integer only)
; c0 - c3 = Transpose of the world view projection matrix
; c4 - c7 = Transpose of the decompression matrix
; c8 = <1.0/255.0, 1.0/255.0, 1.0/255.0, 1.0/255.0>
; c9 = <u scale / 65535.0, v scale / 65535.0, u offset + 0.5, v offset + 0.5>
m4x4    r0, v1, c4                    ; decompress compress position
mul     r1, v2, c8                    ; multiply compressed normal by 1/255
mad     r2.xy, v3.xy, c9.xy, c9.zw    ; multiply uv by scale and add offset

; now the normal vertex shader code, this example just transforms
; the position and copies the normal and uv coordinate to the rasterizer
m4x4    oPos, r0, c0                  ; transform position into HCLIP space
mov oT0, r2                           ; copy uv into texture coordinate set 0
mov oT1, r1                           ; copy normal into texture coordinate set 1
```

For an extra six cycles of vertex shader code, we have reduced vertex data size by 50%.

Optimizations

We can usually eliminate most of the decompression instructions, reducing the shader execution time to roughly the same as the uncompressed version.

The first optimization is noting that anywhere we are already doing a 4x4 matrix transform, we can get any of the previous compressions for free. By incorporating the decompression step into the transform matrix, the decompression will occur when the matrix vector multiply is performed (this works because all the compressions involve linear transforms). This is particularly important for position; we always do a 4x4 matrix transform (to take us from local space into HCLIP space) so we can decompress any of the above for free!

Another optimization is the usual vertex shader rule of never moving a temporary register into an output register. Wherever you are, simply output directly into the output register.

Anywhere you use quantized D3DVSDT_UBYTE4, you might as well use D3DVSDT_D3DCOLOR instead, as it automatically does the divide by 255.0 and swizzling is free in the vertex shader. This also applies the other way; if you are scaling by a

D3DVSDT_D3DCOLOR by 255.0 (and your device supports both), use D3DVSDT_UBYTE4.

Practical Example

By applying these three rules to the same data as before, we achieve 50% savings of memory and bandwidth for zero cycles.

Optimized compressed vertex example:

```
D3DVSD_STREAM(0),
D3DVSD_REG( D3DVSDE_POSITION,     D3DVSDT_SHORT4),
D3DVSD_REG( D3DVSDE_NORMAL,       D3DVSDT_D3DCOLOR),
D3DVSD_REG( D3DVSDE_TEXCOORD0,    D3DVSDT_SHORT2),
D3DVSD_END()
```

Optimized compressed vertex shader example:

```
; v1.xyz = position in the range -32767.0 to 32768.0 (integer only)
; v2.xyz = normal in the range 0.0 to 255.0 (integer only) (swizzled)
; v3.xy = uv in the range -32767.0 to 32768.0 (integer only)
; c0 - c3 = Transpose of the decompression world view projection matrix
; c4 = <u scale / 65535.0, v scale / 65535.0, u offset + 0.5, v offset + 0.5>
m4x4    oPos, v1, c0              ; decompress and transform position
mad     oT0.xy, v3, c4.xy, c4.zw  ; multiply uv by scale and add offset
mov     oT1.xyz, v2.zyx           ; swizzle and output normal
```

Advanced Compression

There are several methods that extend and improve on the basic techniques described in the previous section for special situations.

Multiple Compression Transforms

The main reason that you may not be able to use a compression transform for position is if the object is huge but still has fine details (these are classic objects like large buildings, huge space ships, or terrain). This technique trades vertex constant space for an increase in precision across the object. It doesn't mix with palette skinning very well, but for static data it can increase precision significantly.

Essentially, it breaks the object up into separate compressed areas and each vertex picks which areas it belongs to. The main issue is making sure the compressor doesn't cause gaps in the object.

We load the address register with the matrix to use, which we store by using a spare component (position w is usually spare with compressed data).

Note: This is the same technique used to speed up rendering of lots of small objects (things like cubes): Every vertex is treated as if it were skinned with a single index and a weight of 1.0. This allows a single render call to have multiple local matrices. We just use it to select compression matrices rather than local space matrices.

Compression

The object is broken into chunks beforehand and the standard compressor is applied to each chunk, storing the chunk number in the vertex.

The exact method of breaking the object into chunks will be very data-dependent. A simple method might be to subdivide the object into equally spaced chunks; another might be to choose an area of high detail.

```
MultipleCompressionTransforms
{

    BreakObjectIntoChunks()
    For Each Chunk
        DetermineCompressionMatrix()

        compressionMatrixArray[chunkNum] = C

        For Each Vertex v in Chunk
        {
            v" = C * v
            Store FloatToInteger(v") && chunkNum
        }
}
```

Decompression

The only change from the standard compression transform decompression is to use the stored chunkNum to index into the compression matrix first.

```
Constants
{
    D[MAX_CHUNKS] = C^-1 [MAX_CHUNKS]
}

Decompress( compressedVal, chunkNum )
{
    return (IntegerToFloat( compressedVal ) * D[chunkNum] )
}
```

Practical Example

This compression method is good for position data, even for large objects, but isn't usable for skinned or multi-matrix objects. Each matrix takes four constants, so for equally spaced compression matrices, it quickly fills up constant space. The only extra cost is a single instruction to fill the index register.

Multiple compression transform shader example:

```
; v1.xyz = position in the range -32767.0 to 32768.0 (integer only)
; v1.w = (compression matrix index * 4) in the range 0 to MAX_CHUNKS*4
; c0 = < 1.f, ?, ?, ?>
; array of decompression matrices from c1 up to MAX_CHUNKS*4
; i.e.,  c0 - c3 = Transpose of the 1st decompression matrix
;        c4 - c7 = Transpose of the 2nd decompression matrix
mov a0.x, v1.w            ; choose which decompression matrix to use
mov r0.xyz, v1.xyz        ; for replacing w
```

```
mov r0.w, c0.x              ; set w =1
m4x4 r0, r0 c[a0.x + 0]     ; decompress input vertex using selected matrix
```

Sliding Compression Transform

Rather than pack more data into the already used vector components, this time we put the extra "dummy" padding to good use. By encoding displacement values in the padding, we displace the entire compression matrix, allowing each vertex to slide the compression space to a better place for it.

The number and size of the spare padding ultimately decides how much extra precision is needed. Vertex shaders are not very good at extracting multiple numbers from a single padding, so it is best not to, but we can split values in the compressor and reconstruct the original value in the vertex shader.

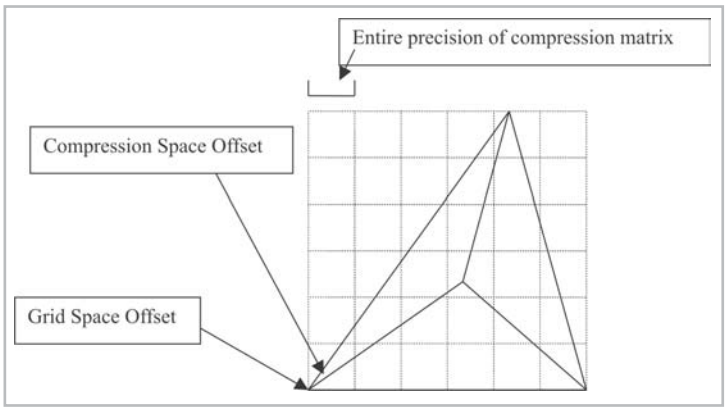

Figure 3: Sliding compression space example

Vertices store offset from local grid origin and store the grid number in the padding. This example has six times more precision along both world axes than the usual compression transform. The grid is aligned with the compression space, thus preserving maximum precision. Another way to look at it is that you break the range into discrete chunks and they are treated separately until decompression.

Compression

The first calculation is to decide how much extra precision you can get out of the padding. There is a different execution cost depending on how or what you have to extract, and this is likely to be the most important factor. The ideal situation is three padding values with enough range. The worst is to extract three values from a single padding. A typical case is one short pad and one byte pad (from a position and a normal); for this case, we can rearrange the data so we have three 8-bit values for extra precision.

In all cases, there is an extra precision that has to be factored into the compressor. This can be done by dividing the range that produces values greater than 1.f and then taking the integer components as the displacement scalars and the remainder as the compression values. The other method is to treat the slide as adding directly to the precision.

```
Sliding Compression Transform
{

    CalculateRotationMatrix()
    For Each Vertex v
    {
        v' = R^-1 * v
        LowerRange = Minimum (LowerRange, v')
        UpperRange = Maximum (UpperRange, v')

        O = TranslationMatrix( LowerRange )
        S = ScaleMatrix( UpperRange - LowerRange )
    }

    S = S / size of slide data-type
    C = R^-1 * O^-1 * S^-1

    For Each Vertex v
    {
        v'' = C * v
        c = frac(v'')
        g = floor(v'')
        store FloatToInteger( c )
        store PackPadding ( g )
    }
}
```

The packing into a spare short and a byte is done by changing the four shorts into two lots of 4 bytes. The space used is the same, but we can access it differently.

Original stream definition:

```
D3DVSD_REG(0, D3DVSDT_SHORT4),
D3DVSD_REG(1, D3DVSDT_UBYTE4),
```

New stream definition:

```
D3DVSD_REG( 0, D3DVSDT_UBYTE4),
D3DVSD_REG( 1, D3DVSDT_UBYTE4),
D3DVSD_REG( 2, D3DVSDT_UBYTE4),
```

We place the lower 8 bits of each position component in register 0 and the upper 8 bits in register 1. This leaves the w component of each register for our slide value. One thing to watch for is the change from signed 16-bit values to unsigned 16-bit values caused by the split; this means the decompression matrix may have to change a little.

Decompression

To decompress, we recombine the position, get the slide values, and scale this by the grid size, which is then added to the compressed value.

```
Constants
{
    D = C^-1
    GridSize = size of data-type
    S = <255.0, 255.0, 255.0, 0.0>

}
```

```
Decompress( lowerVal, upperVal, slideVal )
{
    compressedVal = (IntegerToFloat(upperVal) * S) + IntegerToFloat( lowerVal )
    return ( compressedVal + (slideVal * GridSize)) * D)
}

; v1.xyz = lower half of position in the range 0 to 255.0 (integer only)
; v1.w = x slide
; v2.xyz = upper half of position in the range 0 to 255.0 (integer only)
; v2.w = y slide
; v3.xyz = normal in the range 0 to 255.0 (integer only)
; v3.w = z slide
; c1 = <255.0, 255.0, 255, 0.0>
; c2 = <65535.0, 1.0, 0.0, 0.0>
mov r1.xyz, v1.xyz                    ; due to 1 vertex register per instruction
mad r0.xyz, v2.xyz, c1.xyz, r1.xyz    ; r0.xyz = position (upper * 255.0) + lower
mad r0.x, v1.w, c2.x, r0.x            ; (x slide * 65535.0) + position x
mad r0.y, v2.w, c2.x, r0.y            ; (y slide * 65535.0) + position y
mad r0.z, v3.w, c2.x, r0.z            ; (z slide * 65535.0) + position z
mov r0.w, c2.y                        ; set w = 1
; now decompress using a compression matrix
```

A cost of six cycles has improved our precision 255 times in all axes by reusing the padding bytes. This effectively gives us 24 bits across the entire object, matching the precision (but not the range) of the original floating-point version.

Displacement Maps and Vector Fields

Displacements maps (and to a lesser degree their superset vector fields) are generally considered to be a good thing. It's likely that the future hardware and APIs will support this primitive directly, but by using vertex shaders, we can do a lot today.

First some background. What are displacement maps and vector fields? A displacement map is a texture with each texel representing distance along the surface normal. If that sounds a lot like a height field, it's because displacement maps are the superset of height fields (height fields always displace along the world up direction). Vector fields store a vector at each texel, with each component representing the distance along a basis vector.

Basic Displacement Maps

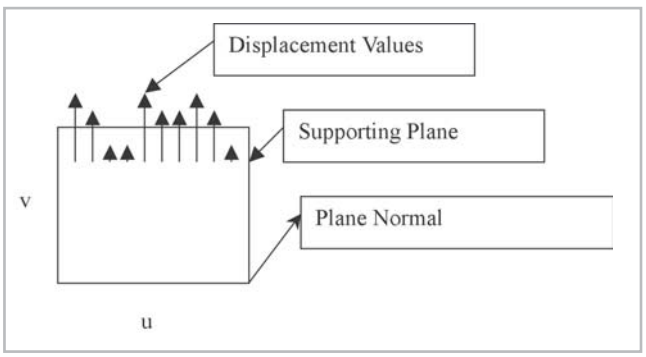

Figure 4: Displacement map example

We have a supporting plane which is constant for the entire displacement map and the displacement along the plane normal at every vertex. For true hardware support, that would be all that is needed, but we need more information.

To do displacement maps in a vertex shader requires an index list, whereas true displacement map hardware could deduce the index list from the rectilinear shape. Since the index data is constant for all displacement maps (only one index list per displacement map size is needed), the overhead is very small.

We also have to store the displacement along the u and v direction, as we can't carry information from one vertex to another. This is constant across all displacement maps of the same size, and can be passed in through a different stream. This makes the vector fields and displacement maps the same for a vertex shader program; the only difference is whether the u and v displacement is constant and stored in a separate stream (displacement maps) or arbitrarily changes per vertex (vector fields).

We can't automatically compute the normal and tangent (we would need per-primitive information), so we have to send them along with a displacement value (if needed; world space per-pixel lighting removes the need).

The easiest way (conceptually) is to store in vertex constants the three basis vectors representing the vector field. One basis vector is along the u edge, another the v edge, and the last is the normal. We also need to store maximum distance along each vector and a translation vector to the <0.0, 0.0, 0.0> corner of the displacement map.

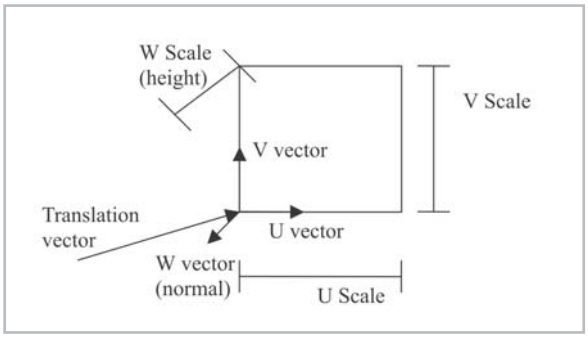

Figure 5: Displacement vector diagram

For each vertex, we take the displacement along each vector and multiply it by the appropriate scale and basis vector. We then add them to the translation vector to get our displaced point.

Basis displacement map/vector field shader example:

```
; v1.x = displacement w basis vector (normal) (dW)
; v2.xy = displacements along u and v basis vectors (dU and dV)
; c0.xyz = U basis vector (normalized) (U)
; c1.xyz = V basis vector (normalized) (V)
; c2.xyz = W basis vector (normalized) (W)
; c3.xyz = translation to <0.0, 0.0, 0.0> corner of map (T)
; c4.xyz = U V W scale
; c5 - c8 = world view projection matrix
; multiple normalized vectors
mul r0.xyz, c0.xyz, c4.x          ; scale U vector
mul r1.xyz, c1.xyz, c4.y          ; scale V vector
mul r2.xyz, c2.xyz, c4.z          ; scale W vector
; generate displaced point
mul r0.xyz, v2.x, r0.xyz          ; r0 = dU * U
```

```
mad r0.xyz, v2.y, r1.xyz, r0.xyz  ; r0 = dV * V + dU * U
mad r0.xyz, v1.x, r2.xyz, r0.xyz  ; r0 = dW * W + dV * V + dU * U
add r0.xyz, r0.xyz, c3.xyz        ; r0 = T + dW * W + dV * V + dU * U
; standard vector shader code
m4x4    oPos, r0, c5              ; transform position into HCLIP space
```

Hopefully, you can see this is just an alternative formulation of the compression matrix method (a rotate, scale, and translation transform). Rather than optimize it in the same way (pre-multiply the transform matrix by the compression transform), we follow it through in the other way, moving the displacement vector into HCLIP space.

Entering Hyperspace

Moving the displacement vector into HCLIP space involves understanding four-dimensional space. By using this 4D hyperspace, we can calculate a displacement vector that works directly in HCLIP space.

It sounds more difficult than it really is. HCLIP space is the space before perspective division. The non-linear transform of perspective is stored in the fourth dimension, and after the vertex shader is finished (and clipping has finished), the usual three dimensions will be divided by w.

The most important rule is that everything works as it does in 3D, just with an extra component. We treat it as we would any other coordinate space; it just has an extra axis.

All we have to do is transform (before entry into the vertex shader) the basis and translation vectors into HCLIP space and we use them as before (only we now use four components), and the displaced point is generated in HCLIP so we don't need to transform them again. We also pre-multiply the basis scales before transforming the basis vector into HCLIP space.

HCLIP displacement map/vector field shader example:

```
; v1.x = displacement w basis vector (normal) (dW)
; v2.xy = displacements along u and v basis vectors (dU and dV)
; c0.xyzw = Scaled U basis vector in HCLIP space (U)
; c1.xyzw = Scaled V basis vector in HCLIP space (V)
; c2.xyzw = Scaled W basis vector in HCLIP space (W)
; c3.xyzw = translation to <0.0, 0.0, 0.0> corner of map in HCLIP space (T)
; generate displaced point in HCLIP space
mul r0.xyzw, v2.x, c0.xyzw           ; r0 = dU * U
mad r0.xyzw, v2.y, c1.xyzw, r0.xyzw  ; r0 = dV * V + dU * U
mad r0.xyzw, v1.x, c2.xyzw, r0.xyzw  ; r0 = dW * W + dV * V + dU * U
add oPos, r0.xyzw, c3.xyzw           ; r0 = T + dW * W + dV * V + dU * U
```

This is a matrix transform. The only difference from the m4x4 (4 dp4) or the mul/mad methods is that the m4x4 is using the transpose of a matrix, whereas mul/mad uses the matrix directly. Clearly there is no instruction saving (if we could have two constants per instruction we could get down to three instructions) using vertex shaders for displacement mapping over compression transform, but we can reuse our index buffers and our u v displacements and make savings if we have lots of displacement maps.

This allows us to get down to only 4 bytes (either D3DVSDT_UBYTE4 or D3DVSDT_SHORT2) for every displacement position past the first map. For vector fields, the only savings we make over compression transform is reusing the index buffers.

Conclusion

I've demonstrated several methods for reducing the size of vertex data. The best method obviously depends on your particular situation, but you can expect a 50% reduction in size quite easily. That's good in many ways, as models will load faster (especially over slow paths, like modems), memory bandwidth is conserved, and memory footprint is kept low.

Acknowledgments

- S. Gottschalk, M.C. Lin, D. Manocha — "OBBTree: A Hierarchical Structure for Rapid Interference Detection," http://citeseer.nj.nec.com/gottschalk96obbtree.html.
- C. Gotsman — "Compression of 3D Mesh Geometry," http://www.cs.technion.ac.il/~gotsman/primus/Documents/primus-lecture-slides.pdf.
- Tom Forsyth — WGDC and Meltdown 2001 Displaced Subdivision
- Oscar Cooper — General sanity checking