Render To Vertex Buffer Programming

Emil Persson
ATI Technologies, Inc.
epersson@ati.com

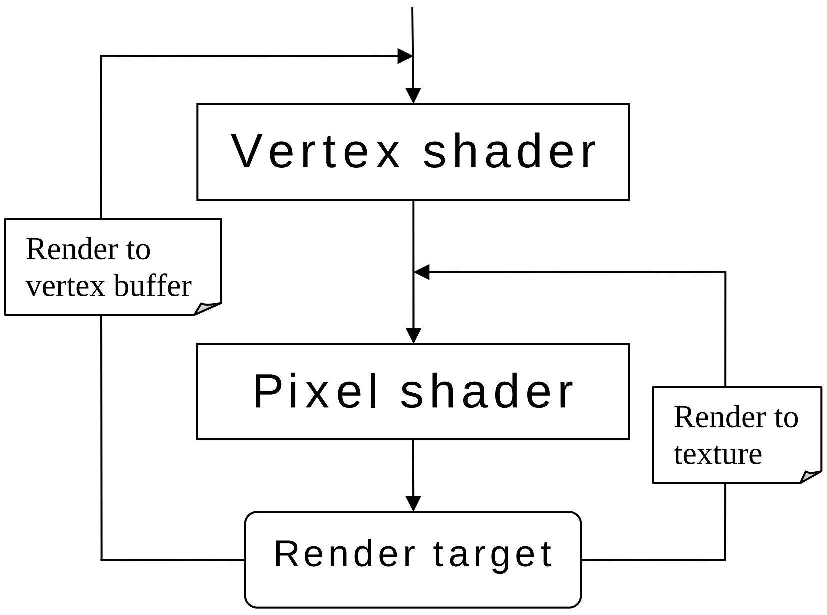
Introduction

Render to vertex buffer is a new feature available in ATI drivers starting with Catalyst 5.9. It is available on all DirectX 9 class ATI hardware from the Radeon 9500 and up, with some extra abilities for the X1x00 series. Render to Vertex Buffer, or R2VB for short, can in some ways be seen as a precursor to DirectX 10. It allows some of the effects of DirectX 10 to be efficiently implemented on DirectX 9 hardware. It also gives an interesting insight in the way of thinking you'll have to get used to when some of the DirectX 10 features become mainstream.

How it works

Basic idea

The basic idea is that you render to a texture as usual. In a second pass you use this texture as a vertex buffer. What is the benefit of this ability compared to regular render-to-texture? What does this feature bring us? Well, it creates a data loop on the GPU. We can first render an image, and then reuse this rendered data in a later pass. Essentially we're looping it back to the beginning of the pixel pipe. Data computed on the GPU can then be reused later on. By doing this we can create interesting effects such as reflections, heat haze, shadows etc. We can also do various simulations on the GPU without burdening the CPU, like for instance water. However, we're still stuck with only pixel processing having this ability. In the water simulation case, we can dynamically simulate a normal map to put over the surface, but we cannot use this method to alter the actual geometry. R2VB solves the problem by looping the data back even further, all the way to the vertex shader. This simple diagram demonstrates the difference.

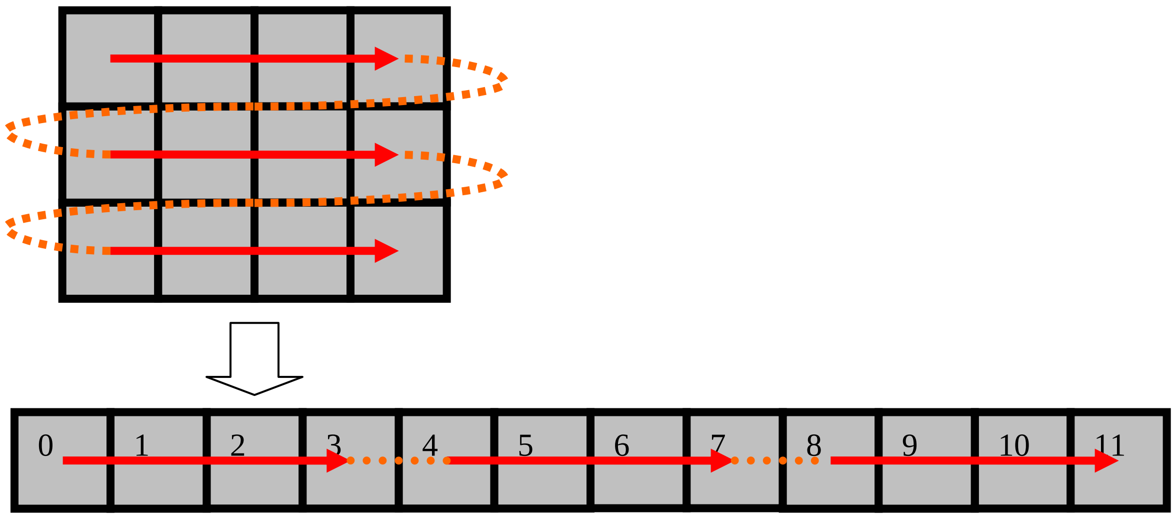


The ability to feed computed data back into the vertex shader allows us to modify actual geometry. It also allows for interesting, and perhaps slightly counterintuitive, ways of solving problems. For instance with the Radeon X1900 hardware we have 48 ALU units in the pixel pipes, while there are only 8 ALU units for vertex processing. Ignoring the slight differences in ALU setup, that's roughly 6x the amount of ALU power in the pixel pipes. By doing vertex shading in the pixel pipes, we can thus achieve significantly better performance. Where to logically place some of the workload becomes a bit fuzzier. In addition to the potential performance boost from R2VB, one can enjoy all the functionality of the pixel shader in the vertex shader – very efficient dynamic flow control, a plethora of texture formats, including compressed textures and extremely fast texture access, which allows using textures as very large constant buffers. In a sense this can also be viewed a precursor to unified shading.

Implementation

Let's take a look how rendered data be used in the vertex pipe. While it's called "render to vertex buffer" it's really more like "render to texture then reinterpret the texture as a vertex buffer", but that's of course way too long for daily language.
What do a texture and a vertex buffer have in common? Both of them are memory arrays addressable by the GPU. The difference is in how the GPU interprets the data in these arrays. For instance the layout of texture elements (texels) is described by texture dimensions, texel format and so on. The layout of data in a vertex buffer is determined by the vertex declaration. To illustrate a decoupling of the data stored in a buffer and its interpretation, one can at any time change the vertex declaration used with a certain area in the vertex buffer, thus reinterpreting the data, still as vertices, but with a completely different format. Of course, that typically doesn't make much sense unless you also change the data in the buffer, but the point is that data are just data, and the GPU could easily be programmed to read vertices from a memory area that belongs to a render target. This is exactly what R2VB does.

For this to make sense, we of course need to ensure that the data we're storing in a render target are laid out in such a manner that interpreting them as vertices would line up well with the usage of a vertex buffer. Normally the application doesn't know what format a texture is in, and the driver and hardware are free to use any swizzle they prefer. Since textures aren't typically accessed in a linear fashion, textures are normally swizzled to optimize for spatially close accesses in texture space. Also, render targets are two-dimensional. Vertex buffers on the other hand are one-dimensional and are accessed sequentially. So to make this work we'll need the render target to be in a linear format such that it can be interpreted as a long vertex array, line by line, like in the illustration below. By creating a render texture with `D3DUSAGE_DMAP` usage flag the application instructs the driver to disable texture swizzling so that the data can be linearly fetched by the vertex fetcher.
The following diagram shows how a 4x3 texture can be interpreted as a 12 element one-dimensional vertex array. This is analogous to how a 2D array in C/C++ would be accessed if you read it sequentially in memory as a 1D array.



Note that it's not necessary that we map pixels to vertices 1:1, even though this is by far the most common case. You could for instance render to an RGBA32F texture, where each of the four components map to the same single float attribute of four different vertices.
On SM 2.0 and SM 2.x ATI hardware such as Radeon 9500 to Radeon X850 only one vertex stream can be mapped to a render target, while on Radeon X1x00 series up to 5 R2VB streams can be used.

API usage

The first thing you need to do in order to use R2VB is to check that it's available. This is done by checking for the "R2VB" FourCC format.

```
bool supportsR2VB = d3d->CheckDeviceFormat(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
                    D3DFMT_X8R8G8B8, 0, D3DRTYPE_SURFACE,
                    (D3DFORMAT) MAKEFOURCC('R','2','V','B')) == D3D_OK;
```

To create a render target intended to be used with R2VB we pass the `D3DUSAGE_DMAP` flag when creating the texture. This tells the driver that we need a linear memory layout as discussed.

```
dev->CreateTexture(width, height, 1, D3DUSAGE_RENDERTARGET | D3DUSAGE_DMAP,
                   D3DFMT_A32B32G32R32F, D3DPOOL_DEFAULT, &d3drt, NULL);
```

The created render target can now be used as usual. It will behave in every way like any other render target would do; only now the data is stored in an intuitive linear fashion, which will only matter once we use it as a vertex buffer. Note that it's not necessary to use it exclusively as a vertex buffer. It's still a regular render target that can be used for texturing as well.

Since the texture is accessed as a linear memory area there's not that much to say about the vertex declaration. It behaves in every way as with any vertex buffer. But naturally in the vast majority of the cases one will probably want to match the texture format with the equivalent vertex declaration types. If the texture is D3DFMT_A32B32G32R32F you naturally would want to match this with a D3DDECLTYPE_FLOAT4 for the stream in question in your vertex declaration. For a direct mapping, these formats should line up perfectly.
The following table matches render target texture formats to all available vertex declaration types. However trickier mapping (e.g. D3DFMT_A32B32G32R32F treated as 2 D3DDECLTYPE_FLOAT2 vectors belonging to different vertices) can also be applied. The driver performs absolutely no validation and it's even possible to map floats to integers and vice versa.

Texture format	Vertex declaration type
D3DFMT_R32F	D3DDECLTYPE_FLOAT1
D3DFMT_G32R32F	D3DDECLTYPE_FLOAT2
D3DFMT_A32B32G32R32F	D3DDECLTYPE_FLOAT4
D3DFMT_G16R16F	D3DDECLTYPE_FLOAT16_2
D3DFMT_A16B16G16R16F	D3DDECLTYPE_FLOAT16_4
D3DFMT_G16R16	D3DDECLTYPE_SHORT2 D3DDECLTYPE_SHORT2N D3DDECLTYPE_USHORT2N
D3DFMT_A16B16G16R16	D3DDECLTYPE_SHORT4 D3DDECLTYPE_SHORT4N D3DDECLTYPE_USHORT4N
D3DFMT_A8R8G8B8	D3DDECLTYPE_D3DCOLOR D3DDECLTYPE_UBYTE4 (**BGRA**) D3DDECLTYPE_UBYTE4N (**BGRA**)

Once the render target has been updated with the desired vertex data we need to tell the driver about our intentions. The D3DRS_POINTSIZE state is used for this purpose. This state normally holds fairly small values as the maximum point size on current ATI hardware is 256. A part of the invalid range of point size values is used to communicate R2VB commands to the driver. In order to make all the commands more readable and avoid the need to understand intricacies of the special command token formatting, the following few utility functions can be used:

```
#define R2VB_GLB_ENA_CMD        0x0
#define R2VB_VS2SM_CMD          0x1

// R2VB Command Token
#define R2VB_TOK_CMD_SHFT       24
#define R2VB_TOK_CMD_MSK        0x0F000000
#define R2VB_TOK_CMD_MAG        0x70FF0000
#define R2VB_TOK_CMD_MAT        0xFFFF0000
#define R2VB_TOK_PLD_MSK        0x0000FFFF
```

```
#define R2VB_GLB_ENA_MSK         0x1
#define R2VB_VS2SM_STRM_MSK      0xF
#define R2VB_VS2SM_SMP_SHFT      0x4
#define R2VB_VS2SM_SMP_MSK       0x7

// R2VB enums
#define R2VB_VSMP_OVR_DMAP    0     // override stream with dmap sampler
#define R2VB_VSMP_OVR_VTX0    1     // override stream with vertex texture 0 sampler
#define R2VB_VSMP_OVR_VTX1    2     // override stream with vertex texture 1 sampler
#define R2VB_VSMP_OVR_VTX2    3     // override stream with vertex texture 2 sampler
#define R2VB_VSMP_OVR_VTX3    4     // override stream with vertex texture 3 sampler
#define R2VB_VSMP_OVR_DIS     5     // disable stream override
#define R2VB_VSMP_OVR_NUM     6     //
#define R2VB_VSMP_NUM         5     // 5 available texture samplers

__inline DWORD r2vbToken_Set(DWORD cmd, DWORD payload)
{   DWORD cmd_token = (cmd << R2VB_TOK_CMD_SHFT) & R2VB_TOK_CMD_MSK;
    DWORD pld_data = payload & R2VB_TOK_PLD_MSK;
    return (R2VB_TOK_CMD_MAG | cmd_token | pld_data);
}

__inline DWORD r2vbGlbEnable_Set(BOOL ena)
{   DWORD payload = ena & R2VB_GLB_ENA_MSK;
    DWORD dw = r2vbToken_Set(R2VB_GLB_ENA_CMD, payload);
    return dw;
}

__inline DWORD r2vbVStrm2SmpMap_Set(DWORD str, DWORD smp)
{   DWORD sampler = (smp & R2VB_VS2SM_SMP_MSK) << R2VB_VS2SM_SMP_SHFT;
    DWORD stream = (str & R2VB_VS2SM_STRM_MSK);
    DWORD payload = sampler | stream;
    DWORD dw = r2vbToken_Set(R2VB_VS2SM_CMD, payload);
    return dw;
}
```

These functions are declared in the atir2vb.h file included in the ATI SDK (located in Samples\Framework\Direct3D). The `r2vbGlbEnable()` function is used for globally enabling and disabling R2VB while `r2vbVStrm2SmpMap_Set()` is used for mapping samplers to vertex streams.

For any R2VB vertex stream mapping to take effect we must first globally enable the R2VB extension. This is done as follows:

```
dev->SetRenderState(D3DRS_POINTSIZE, r2vbGlbEnable_Set(TRUE));
```

Naturally, disabling it is done with the call:

```
dev->SetRenderState(D3DRS_POINTSIZE, r2vbGlbEnable_Set(FALSE));
```

After R2VB has been enabled, the next step is to bind a render target to a vertex stream. Unfortunately DirectX 9 API doesn't allow directly mapping texture surface in video memory to the vertex streams. The only way to specify a texture is to use the SetTexture API call. This means we need to use SetTexture with `D3DDMAPSAMPLER` for passing the texture down to the driver. The good thing about `D3DDMAPSAMPLER` is that it can be used on all SM2.0 hardware. On the Radeon X1x00 series, which support VS3.0, we also have four `D3DVERTEXTEXTURESAMPLER[n]` samplers available, allowing a total of five different vertex streams to be fetched from textures simultaneously. This is how you bind a render target to a DMAP sampler:

```
dev->SetTexture(D3DDMAPSAMPLER, renderTarget);
```

At this point mapping a texture to `D3DDMAPSAMPLER` sampler (or to `D3DVERTEXTEXTURESAMPLER[n]`sampler on Radeon X1x00) will not cause the texture to be used as a displacement map or vertex texture since displacement maps and vertex textures aren't supported by ATI hardware. We also need to tell the driver that the streams should be fetched from these samplers instead of from the vertex buffer. This can be done as shown in the following code snippet:

```
// Tell the driver that stream 1 is to be fetched from the DMAP texture
dev->SetRenderState(D3DRS_POINTSIZE, r2vbVStrm2SmpMap_Set(1, R2VB_VSMP_OVR_DMAP));
```

The first parameter of r2vbVStrm2SmpMap_Set helper function specifies which vertex stream should be overloaded with R2VB data, and the second parameters tell the driver where that data will be coming from. Here `R2VB_VSMP_OVR_DMAP` signals that the data will come from texture bound to `D3DDMAPSAMPLER` and similarly we could use for instance `R2VB_VSMP_OVR_VTX2` to fetch vertices from `D3DVERTEXTEXTURESAMPLER2`. Once we're done with using R2VB data we should restore the vertex stream to fetch from the vertex buffer as usual.

```
// Stream 1 restored to regular vertex buffer mode
dev->SetRenderState(D3DRS_POINTSIZE, r2vbVStrm2SmpMap_Set(1, R2VB_VSMP_OVR_DIS));
```

Note that even though we fetch the vertices from the texture, and not the vertex buffer, we still need to call `SetStreamSource`. We're not interested in the actual vertex buffer, but we need the stride and offset parameters. Instead of using the actual vertex buffer with real vertex data we could use a dummy vertex buffer. This can be any vertex buffer and the content doesn't matter, so any existing buffer could be used. Many developers will prefer to use the DirectX debug runtime at least some of the time during development, so it should be mentioned that the debug runtime has stricter validation which in this case means that you have to ensure that the dummy vertex buffer is large enough to contain all the vertices you specify in the draw call. In some cases this is not a problem. For instance if you use R2VB to update a water surface, and you have a static buffer containing two floats, the X and Z, and the water surface is a dynamically updated render target in R32F format containing the Y component. In that case, you can just use the static buffer as the dummy as it's guaranteed to always be large enough (in fact, even twice as big). In other cases where no available vertex buffer is larger than the largest stream fetched from a render target, the easiest solution may be to just create a separate dummy vertex buffer. In order not to waste too much memory it's best to detect whether you're running with debug or release runtime. Since most end users will have the release runtime it's best to create a minimal vertex buffer for this case (1 byte will suffice), while you could create a large one to cover the biggest stream used in the lifetime of the application if the debug runtime is detected.

The final code to set up rendering using the render target as a vertex buffer could look something like this:

```
// Enable render to vertex buffer extension
dev->SetRenderState(D3DRS_POINTSIZE, r2vbGlbEnable_Set(TRUE));

// Setup stream 0 – regular VB data
dev->SetStreamSource(0, staticVertexBuffer, 0, 2 * sizeof(float))

// Setup stream 1 – R2VB data
dev->SetTexture(D3DDMAPSAMPLER, renderTarget);
dev->SetRenderState(D3DRS_POINTSIZE, r2vbVStrm2SmpMap_Set(1, R2VB_VSMP_OVR_DMAP));
dev->SetStreamSource(1, dummy, 0, 4 * sizeof(float))

// Draw ...
dev->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0, nVertices, 0, nPrimitives);

// Restore stream 1 to regular vertex buffer mode
dev->SetRenderState(D3DRS_POINTSIZE, r2vbVStrm2SmpMap_Set(1, R2VB_VSMP_OVR_DIS));
dev->SetTexture(D3DDMAPSAMPLER, NULL);

// Disable render to vertex buffer extension
dev->SetRenderState(D3DRS_POINTSIZE, r2vbGlbEnable_Set(FALSE));
```

Finally, a detail worth noting is that the driver will not allow the same sampler to be mapped to multiple vertex streams. When you update the mapping of a certain stream, the driver will disable any previous binding that used that sampler. This means that the stream that sampler was previously mapped to will become mapped to a regular vertex buffer set with `SetStreamSource`. This behavior will not be a concern for the vast majority of the applications.

```
dev->SetRenderState(D3DRS_POINTSIZE, r2vbVStrm2SmpMap_Set(0, R2VB_VSMP_OVR_DMAP));
dev->SetRenderState(D3DRS_POINTSIZE, r2vbVStrm2SmpMap_Set(1, R2VB_VSMP_OVR_VTX0));
```

```
dev->SetRenderState(D3DRS_POINTSIZE, r2vbVStrm2SmpMap_Set(2, R2VB_VSMP_OVR_VTX1));
// Will implicitly disable the previous VTX0->stream1 binding
dev->SetRenderState(D3DRS_POINTSIZE, r2vbVStrm2SmpMap_Set(3, R2VB_VSMP_OVR_VTX0));
```

After this code has run, stream0 will read from the DMAP sampler, stream1 will read from the regular vertex buffer, stream2 from vertex texture 1 and stream3 from vertex texture 0.

Usage cases

So now that we have this functionality, what can we do with it? The idea to render to a vertex buffer diverts a fair amount from traditional techniques and it may not be immediately obvious what this is useful for. Similarly, regular render-to-texture felt somewhat alien when it was first introduced. Initial uses tended to be for rearview mirrors and other stuff that were intuitive enough, but soon more abstract techniques like shadow mapping were developed. Today render-to-texture is an indispensable tool in a wide range of rendering techniques. Similarly, techniques like displacement mapping on rectangular terrain patches may be an intuitive use for R2VB, but the possibilities go much further than that. Learning R2VB today will give you an early insight into the way of thinking that will become mainstream concepts in the DirectX 10 timeframe.

In the March release of the ATI SDK there is a large set of R2VB samples that illustrate a range of effects that can be implemented with R2VB. As a good “getting started” sample, take a look at the R2VB-Terrain sample. It does a simple terrain morphing. Other effects that have been implemented in the SDK include:

- Characters leaving footprints in the snow.
- Inverse kinematics.
- Cloth simulation.
- Collision detection.
- Water simulation.
- Particle system.
- Sorting.

All these are done entirely on the GPU with a minimal amount of CPU aid, where in many cases traditional solutions would have required the CPU to solve the bigger part of the problem. Other samples show how to use R2VB to optimize performance of certain tasks such as:

- Character animation.
- Shadow volume extrusion.

Or even reintroduce hardware acceleration of something that no longer has dedicated hardware:

- N-patches.

Looking into any of these samples and their documentation will explain the implementation details of each of these particular techniques. Other possibilities where render to vertex buffer could be advantageous include:

- Infinite terrain based on noise functions.
- Massive terrain (like the entire surface of Mars) compressed in ATI1N format.
- Various forms of tessellation.
- Destructible geometry.
- Physics.

The sky is the limit.

Performance considerations

R2VB doesn't require many new aspects of performance tuning over the general advice presented in *The Radeon X1x00 programming guide*. But there are a few things to remember that are particularly relevant in the context of R2VB.

Exploit parallelism. If you're processing a single channel attribute, like for example a vertical displacement on terrain, you could process four vertices in parallel. Instead of processing one element in the shader and outputting it to an R32F texture, you could process four vertices in parallel and output to an RGBA32F texture.

Don't use larger than necessary render target and vertex buffer data types. In many cases 16-bit floats and integers work fine without precision problems and give a healthy performance improvement. In some cases you could even use RGBA8. In many cases where you're maintaining some kind of state in the render targets, for example position and direction for particles in a particle system, you'll be reading current state into the shader as you're updating it, writing the results back to another render target, and then reading that as a vertex buffer. If you're using wide data types the bandwidth required for this could become a bottleneck. Also keep in mind that wide data types not only require extra bandwidth, but also need additional cycles when you sample them in the pixel shader, even if all data is in the cache. An RGBA32F texture requires 4 cycles per sample, whereas RGBA16F only requires 2 cycles. With the Radeon X1900 and Radeon X1600 providing a 3:1 ALU:TEX ratio in pixel shaders it becomes increasingly more important to keep the amount of cycles spent on texture sampling down.

Since many of the uses of R2VB are related to animating stuff on the GPU, like particles systems, cloth simulation, physics etc. it's common that you need to keep track of and process a larger amount of data per pass than usual. You may for instance want to keep track of position, direction, normals, etc, and if all need to be updated in the same processing pass this tends to amount to more than four components. For this reason you'll probably find that MRTs (multiple render targets) come into play much more frequently when you work with R2VB than otherwise. Unfortunately, one of the limitations of MRTs is that all render targets need to be the same bit-depth. This can be particularly annoying if you need 5 components. In order to accompany the 5th component you'd have to use another render target of the same size, which could mean three unused components. In some cases you can mix and match same size formats to make it less painful, for instance as RGBA16F and RG32F which would give you six components, where two have the luxury of additional precision. Another solution that can be considered in cases like this is to pack two or more components together in one. If for instance the 5th component can be merged with another component and the two later unpacked in the vertex shader in the final pass, this can in many cases be faster than reading and writing twice as much data in the pixel shader.

Another reason why you'd sometimes want to pack components together is that you only have the DMAP sampler available on pre-X1x00 hardware. This limits the amount of R2VB data you can read as a vertex buffer to four components. Fortunately, not all data in flight will necessarily be used for actual rendering, like for instance with a particle system you keep track of position, direction and particle life time, but the direction vector is relevant only for animation and not for rendering, leaving us with only four components of the seven actually needed to be read as a vertex buffer in the end. However, we're not always that lucky. A common case is if you need both a position and a normal. That's 6 components. In that case the normal is a good candidate for packing into the .w component to squeeze it into a four component vector since its components are limited to values between -1 and 1. The code below packs a normal with roughly 8 bit precision per component when using a 32bit float[*], at a cost of four instructions.

```
Out.Pos_Normal.w = dot(floor(normal * 127.5 + 127.5), float3(1 / 256.0, 1, 256.0));
```

Decoding this in the vertex shader can be done with the following code, which comes at a cost of three instructions.

```
float3 normal = frac(Pos_Normal.w * float3(1, 1 / 256.0, 1 / 65536.0)) * 2 - 1;
```

Conclusion

Render to vertex buffer adds a lot of interesting opportunities. By providing a data loop-back all the way to be beginning of the pipeline it allows us to manage and process data and states fully contained on the GPU in ways that were previously not possible. With R2VB many more tasks can now be done entirely by the GPU without burdening the CPU. This gives us a great chance to get started with DirectX 10 rendering algorithms on a DirectX 9 platform.

[*] The mantissa is 23bits, meaning that the x component may only receive the equivalent of 7 bits depending on the value of the z component. More elaborate math can be used to extract more precision at the cost of more instructions, for instance by using 7 bits plus sign for z.