SDK White Paper

Fog Polygon Volumes
Rendering Objects as Thick Volumes

WP-01416001_v01
July 2004

nVSDK

Rendering Objects as Thick Volumes

Introduction

This article presents a convenient and flexible technique for rendering ordinary polygon objects of any shape as thick volumes of light-scattering or light-absorbing material. Vertex and pixel shaders are used in multi-pass rendering to generate a measure of the object's thickness at each pixel. The thickness at each pixel is then used to produce the colors of the object on screen. Thickness information is computed each frame from the appropriate point of view, and the result is a true volumetric rendering of ordinary polygon objects. No preprocessing of object data is required, and the result is a volumetric technique suitable for interactive dynamic scenes. The appearance of the volume objects is easy to control, and artist-created color ramps can be used to map thickness to color. Several upcoming games are employing the technique, and it is a good way to bring practical volumetric effects to interactive real-time rendering.

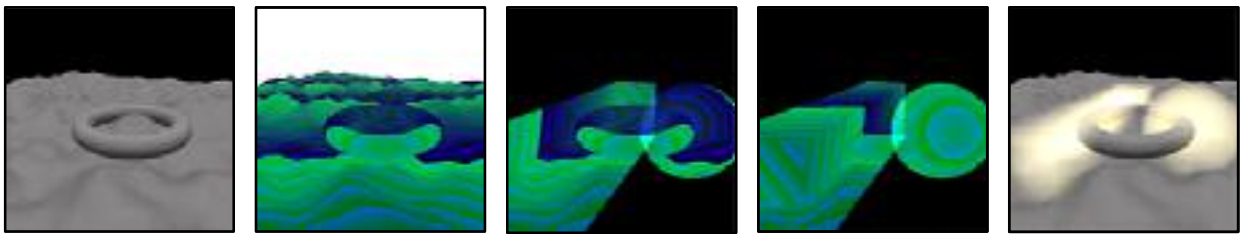
The technique can be implemented on hardware that supports Microsoft's Pixel Shaders version 1.3 or higher, and runs at real-time frame rates in complex scenes. Since no pre-processing or special treatment of the volume object geometry is required, it is trivial to animate and distort the volume objects. An efficient and simple method is given to properly render any closed hull mesh, convex or concave, as a thick volume and to handle any intersection cases where opaque objects penetrate the volumes. This paper also introduces a new method of dithering to overcome the effects of aliasing in the per-pixel thickness information. The dithering is accomplished using texture data, and it does not complicate the rendering or require additional passes.

This article focuses on rendering thickness from the scene camera's point of view. This is suitable for volumes of single-scattering material. In other words, material where light arriving at each pixel is the result of only one scattering interaction in the material, thus the total amount of light is a function only of thickness. As the visible thickness increases, the number of scattering particles increases and so does the probability of scattering. Scattering may both add light and attenuate transmitted light. More sophisticated models of scattering could be employed, but will not be presented here. Instead, refer to Hoffman and Preetham for an excellent introduction to various types of scattering [Hoffman02].

Rendering Overview

The technique for rendering objects as thick volumes is a departure from traditional 3D rendering. It involves rendering to off-screen render-target textures, rendering depth information as RGBA colors, using vertex shaders and textures to encode information, and using alpha blending to add and subtract high-precision encoded depth information. Before presenting each of those aspects in detail, let's start with an overview of the complete rendering process, so you can clearly see what's involved and how the technique compares to other approaches.

The full implementation of the technique for hardware supporting Direct3D Pixel Shaders 2.0 is illustrated in Figure 1. These steps render any volumetric shape, handle all solid objects intersecting the volumes, dither the thickness information, and handle any camera position in the scene, whether the camera is inside or outside of the volumes or solid objects. The technique involves five steps for ps.2.0 hardware. One additional pass (not shown) is required for ps.1.3 hardware. On hardware that supports Direct3D's Multiple Render Targets (MRTs), Figure 1 steps a) and b) can be combined into a single pass. Hardware that supports additive blending to floating point render targets can combine steps c) and d) into a single pass, reducing the implementation to three passes on GeForce 6800 series hardware.



a. **b. Texture 'O'** **c. Texture 'B'** **d. Texture 'F'** **e.**

a) Opaque objects are rendered to the ordinary back buffer
b) The view-space depth of opaque objects that might intersect the volume objects is rendered to a texture we label 'O.' Depth is encoded as a 32-bit RGBA color at each pixel.
c) All volume object back faces are rendered to texture 'B' using additive RGBA blending to sum the depths. A pixel shader samples 'O' while rendering each triangle in order to handle intersections.
d) All volume object front faces are rendered to texture 'F' while sampling 'O' to handle intersections.
e) Textures 'B' and 'F' are sampled to compute the volume thickness, the thickness is used to generate a color for the volume, and the color is blended to the scene rendered in a).

Figure 1. Rendering Steps of the Volume Technique

One advantage of this technique is that the rendering does not change in order to handle various intersection cases and camera positions. No extra passes or knowledge about the objects or scene is required as long as the depth complexity of the volume objects remains below a certain adjustable limit. A later section will present this in greater detail, but the depth complexity limit depends on the precision of the thickness information and the number of bits of each color channel used to hold the thickness information. The tradeoff between depth complexity and precision can be adjusted from frame to frame, though in practice, this adjustment is not necessary. A depth complexity of 16 or 32 volume object faces can be rendered with 15 or 12 bits of depth precision in a single pass. On hardware that supports blending to floating point render targets, there is no limit to the depth complexity that can be handled. Hardware without that support can also use multiple passes to avoid the tradeoff.

Computing Thickness

First, we need a way to get thickness information from ordinary polygon hulls. Dan Baker presented a technique for this in the Microsoft DirectX™ 8.1 SDK "Volume Fog" example [Baker02]. His approach can be extended in a number of ways, but the basic approach is to calculate thickness by subtracting the view-space depth of an object's front faces from the depth of the back faces. The polygonal object is rendered to off-screen render targets using some measure of depth interpolated across the polygons, and the thickness is computed at each rendered pixel. At any given pixel, if we sum the depths of all of an object's front faces at that pixel and sum the depths of all back faces, the thickness through the object is the back face sum minus the front face sum (Figure 2).

As shown in Figure 2, for a given pixel on screen, the thickness through the object is the sum of the depths of all front faces at that pixel subtracted from the sum of the depths of all back faces at that pixel.

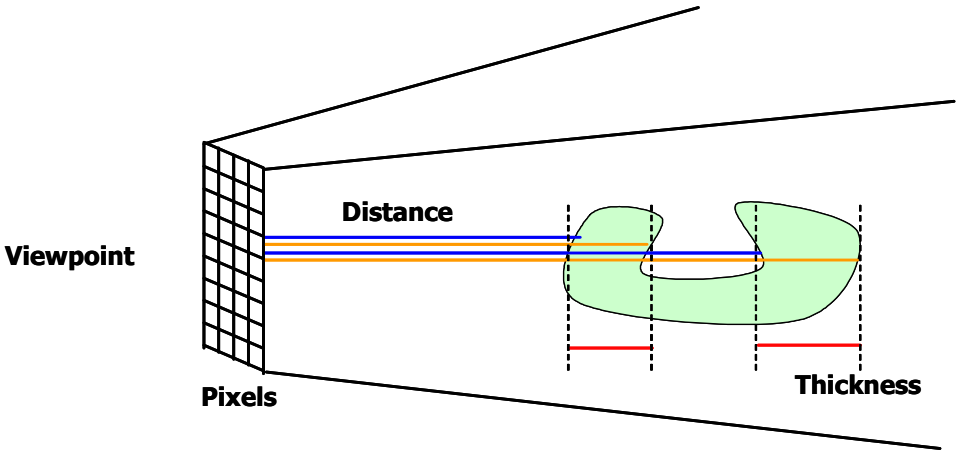


Figure 2. Computing Thickness

Depth is calculated at each vertex as part of the standard 3D view transform. This is interpolated for standard Z-buffer rendering, but the Z-buffer information is not practical to use for this technique. It is too costly in terms of performance, and the graphics APIs have no flexibility for summing and differencing the Z-buffer information. Attempting to manipulate the information on our own would require decompressing and copying the GPU data across to the CPU for processing. This would break the parallelism of the two processors, stall the GPU, and burden the CPU unnecessarily.

Instead, we can use standard RGBA 8-bit color rendering and additive blending to accomplish the thickness calculations entirely on the GPU. A high precision depth value can be split up and encoded across the color channels of an RGBA-8 color. I'll refer to this as RGB-encoding of the depth information. Standard blend operations can then sum the encoded values. This allows us to process and sum, for example, 12-bit or 18-bit depth information using commonplace RGBA-8 render targets.

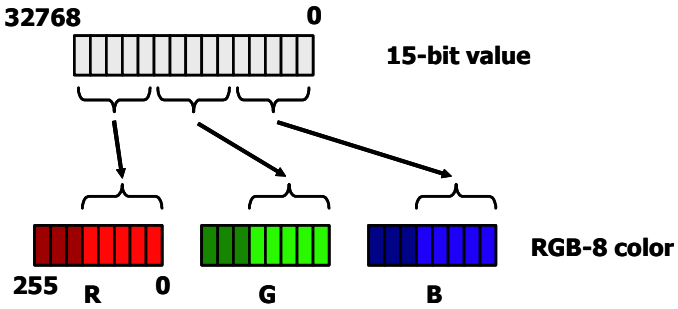
GeForce FX and Radeon 9800 series hardware has introduced support for rendering high precision color information with up to 32 bits per color component. This gives a total of 128-bits per pixel. Unfortunately, these chips do not support additive blending of high precision colors, so they are not capable of performing the depth sums as efficiently or quickly as with RGBA-8 additive blending. The GeForce 6000 series hardware does support blending of 64 bpp and 128 bpp colors, as well as signed floating point blending, which can be used to optimize Figure 1 steps c) and d) into a single pass.

RGB-Encoding of Values

A standard RGBA-8 render target can do a fantastic job of storing and accumulating high precision scalar (1D) values. The bits of a number can be split across the 8-bit red, green, blue, and alpha color channels using any number of the low bits of each channel. When the bits of a number are split across the R, G, and B colors, we call it an RGB-encoded value. A particular case is illustrated in Figure 3, where a 15-bit number is split into three 5-bit color values. The precision at which we can encode values is given by the number of low bits, L, we use in each color channel multiplied by the number of color channels. For example, if we use four low bits (L=4) from each R, G, and B channel, we can encode 12-bit values (3*4).

It's important to note that we use only a few of the lowest bits of each color channel to encode any single value. The remaining high bits are left empty so that when two or more values are added, the low bits can carry over into the unused high bits. RGB-encoded values can be added together using standard RGBA blend operations until all the bits of any color channel are full. At that point, any further additions will be lost, because the bits of one color channel do not carry into the other channels. The number of high 'carry' bits in each color channel is (8-L), and the number of RGB-encoded values we can add together without error is 28-L. There is a tradeoff between the precision we can encode and the number of encoded values that can be added together. For our case of encoding a 15-bit value (L=5), we have three carry bits, so we can sum at most 8 values into any given RGBA-8 color. Figure 3 includes a table relating precision to the number of values that can be safely added.

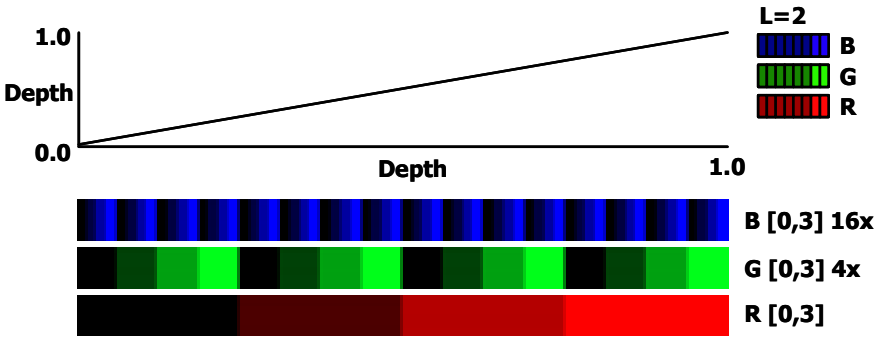
Figure 3 shows the encoding of a 15-bit value using five low bits (L=5) of each 8-bit R, G, and B color channel. The diagram relates the number of low bits, L, to the precision of each value and the number of encoded values which can be added into an RGB-8 color before error occurs due to saturating all of the bits of a particular color channel.



L Bits	Precision	#'s Added
3	9-bits	32
4	12-bits	16
5	15-bits	8
6	18-bits	4

Figure 3. Encoding a 15-bit Value

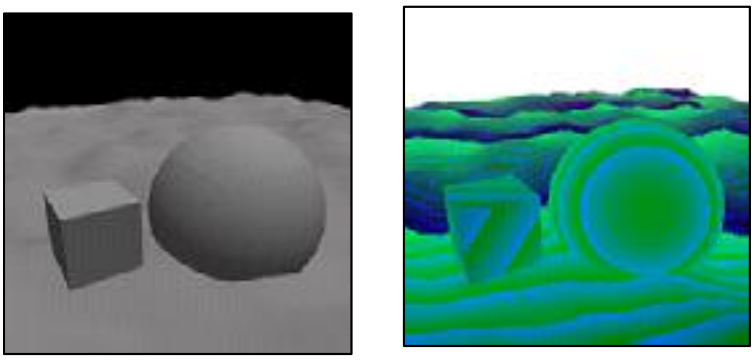
Figure 4 illustrates the RGB-encoding applied to a steadily increasing value. The RGB-encoded value is the sum of the R, G, and B ramps at a point along the axis. Only two bits per color channel are used to better illustrate the relationship of the colors, and a scheme is used where blue holds the least significant bits, green holds the middle significant bits, and red holds the most significant bits. Thus, the green values go through one cycle each time red increases by one bit, and blue cycles once for each green increment.



This figure illustrates encoding 6-bit depth values using two low bits (L=2) of each channel of an RGB-8 color. Depth varies from 0 to 1 from the near clipping plane to the far clipping plane, and is encoded by adding the blue, green, and red color ramps shown. At a depth of 1.0 the color is RGB=(3,3,3) out of the full (255, 255, 255) range, and at 0.75 the color is (3,0,0).

Figure 4. Encoding 6-bit Depth Values

Applying this RGB-encoding of depth to the simple scene shown in Figure 5a gives the result shown in Figure 5b. Here, four bits are used from each color channel. The RGB colors are displayed over-bright, because in practice the low bit values of each color would appear mostly black. Red values are too low to be noticeable in Figure 5b, but if the objects extended farther toward the far clip plane, the red values would become noticeable. In practice, the RGB-encoded depths are rendered to an off-screen texture render target. This allows us to read back the depths in later rendering operations, which is important for handling solid objects that intersect the volumes of fog.



a **b**

Objects rendered with traditional shading (a) and RGB-encoded depth rendering (b) at 12 bits of precision (L=4). The RGB-encoded colors are shown over bright, as their actual range, in this case from [0, 16] out of [0,255], would appear mostly black.

Figure 5. Applying RGB-encoding to Depth

RGB-encoding is easy to achieve using programmable vertex shaders and small color ramp textures. The encoding can be applied to any per-vertex scalar we compute in a vertex shader, but here all we care about is the per-vertex depth. The vertex shader computes a depth value at each vertex as part of the standard 3D transform from object space to homogenous clip space (HCLIP space) as shown in the following vertex shader assembly code `VS 1`:

```
(VS 1)
DP4 r1.x, V_POSITION, c[CV_WORLDVIEWPROJ_0]
DP4 r1.y, V_POSITION, c[CV_WORLDVIEWPROJ_1]
DP4 r1.z, V_POSITION, c[CV_WORLDVIEWPROJ_2]
DP4 r1.w, V_POSITION, c[CV_WORLDVIEWPROJ_3]
MOV oPos, r1
```

`V_POSITION` is the input vertex position in object space, and `CV_WORLDVIEW-PROJ_<N>` are the elements of the standard 4x4 transform-and-project matrix used in 3D rendering. The `r1.w` component is the vertex's distance to the camera plane (not the radial distance to the camera and not the distance to the near clip plane), so it behaves correctly when linearly interpolated in rasterization. This `W` component is easily turned into three texture coordinates which can access small color ramp textures to achieve the encoding of Figure 5. All we have to do is scale the `W` component so it varies from zero to one from the near to far plane, and scale that value by the number of times each color ramp repeats.

The color ramp textures are typically small, with one texel per color value, and they are created to match our choice of the number of low bits, L, we are using from each channel. The color value at each texel of the color ramps is simply an integer, L-bits in size, corresponding to the texel location. For example, if we choose a 12-bit encoding, then L = 4 bits from each color channel, so we use R, G, and B ramps 16 texels long (16 being 2L), with values ranging from 0 to 15 over the 16 texels. The lowest coordinate texel, which is at (0,0), has the value 0 out of 255, the second texel has the value 1 of 255, the eighth is 7, etc.

Texture repeating or wrapping is enabled so that one color ramp can repeat many times over the range of depths. The texture coordinate for the red texture ramp is the W depth scaled to [0,1] from the near to far clip plane. (The range notation [n,m] denotes all numbers from n to m, inclusive of the limit values n and m). The coordinate for green is this same coordinate multiplied by the number of values in the red color ramp, which is 2L, so that the green color ramp repeats 2L times or once for each bit increment of the red color. The texture coordinate for the blue color ramp is the red coordinate scaled by 2L * 2L, or 22L, so that the blue ramp repeats once for each increment of the green color ramp. For the case of Figure 5 where L=2, the texture coordinate for green ranges from zero to four, and the blue coordinate spans [0,16]. These texture coordinates are calculated and output by the vertex code fragment listed in VS 2, where 'near' and 'far' denote the distances to the near and far clip planes.

```
(VS 2)
  // CV_RAMPSCALE = ( 1.0, 2^L, 2^(2L), 0 )
  // CV_NEARFAR = ( 1/(far-near), -near/(far-near), 0, 0 )
  // Scale r1.w to [0,1] from near to far clip planes
MAD r1.w, r1.w, c[CV_NEARFAR].x, c[CV_NEARFAR].y
  // oT0 = ( [0,1], [0,2^L], 0, 0 ) + ( 0, 0, 1, 1 )
MAD oT0.xyzw, r1.w, c[CV_RAMPSCALE].xyww, c[CV_RAMPSCALE].wwxx
  // oT0 = ( [0,2^(2L)], 0, 1, 1 )
MAD oT1, r1.w, c[CV_RAMPSCALE].zwww, c[CV_RAMPSCALE].wwxx
```

Three separate color ramp textures could be used, but it is better to combine the red and green ramps into a single texture where red is accessed with the X coordinate and green is accessed with the Y coordinate. This saves a texture fetch and math operation in the pixel shader which fetches the color ramps. The blue ramp is not merged with red and green, because that would entail using a 3D volume texture. For simple color ramps, such a texture is wasteful. As you'll see later, the blue bits are special, and we can use a larger 2D dithered color ramp to dither the least significant bit of depth information. To form the RGB-encoded depth value and output it to a texture render target, we use the following pixel shader code, PS 1, which adds the red-green color from T0 to the blue color from T1:

```
(PS 1)
ps.1.1
TEX    t0           // read red+green texture ramp
TEX    t1           // read blue texture ramp
ADD    r0, t0, t1   // output RGB-encoded value
```

These simple shader fragments and the small RGB ramp textures are all we need to render functions of depth and distance for any object. Vertex positions can be animated in software or in a vertex shader, and the depth-encoding behaves correctly in all cases. In practice, more operations are added to these shaders to compare depth-encoded objects to previously rendered RGB-encoded values, but more about that later.

Decoding RGB-encoded Values

There are several advantages to this scheme of encoding high precision values. Unlike exponentiation (RGBE) or multiplicative (RGBM) encoding, RGB-encoded values can be added by simply adding each color channel. An offset can be added to each color channel in order to store negative values in a biased state, and decoding the values can be done in a single dot product operation. Hardware that supports Microsoft's D3D8 Pixel Shaders version 1.1 or higher can perform the decode in one shader instruction.

The RGB-encoding scheme spreads the bits into each color channel by dividing the middle and high bit ranges by a scale factor, or shifting the bits down to begin from zero. Decoding the values is simply a matter of multiplying each color channel by that same scale factor (shifting the bits back up to their original values) and adding the shifted values together. This is accomplished by a single dot product, as illustrated in equation 1.1, where **VDecoded** is the decoded value, **T0** is the RGB-encoded value, **C** is a constant vector of scale values for each channel, and **scale** is an arbitrary scale factor that can be used as a convenient control to adjust the output to some meaningful or more pleasant visual range.

```
(1.1)
( C.x, C.y, C.z )  =  scale * ( 1.0,  1 / 2L, 1 / 22L )
VDecoded =  C.x * T0.red + C,y * T0.green + C.z * T0.blue
VDecoded =  C  DOT  T0
```

The multipliers, **C,** for each channel depend on the number of bits, **L**, we chose from each channel to encode the values. Typically, four or five bits are used from each channel, so the values **1/2L** and **1/22L** may be small. For example, with **L=5** we have **C = (1, 1/32, 1/1024).** Since the values may be small, the dot product must be executed at high precision in the pixel shader.

Pixel shader versions 1.1 – 1.4 have two classes of operations: texture addressing operations and arithmetic operations. Arithmetic operations can be executed at 9-bit precision per color component. This is not sufficient to hold small scale values like **1 / 1024**, so we should use the texture addressing operations which must be performed at floating point precision. If we're working with pixel shaders 2.0 and higher, all operations are executed at floating point precision, so the **ps.2.0** arithmetic operations can be used. Shader program fragments that perform the decode for **ps.1.3** and **ps.2.0** are shown in listings **PS 2** and **PS 3**. In these shaders, a dot-product operation generates a texture coordinate which we use to access a color ramp texture.

This maps the RGB-encoded values (depth, or thickness, etc.) to any colors we like, and it provides a convenient way for artists to control the look of the volume objects.

```
(VS 1)
// vertex shader pseudo-code to generate inputs
// for the pixel shader below
vs.1.1
MOV    oT0, SCREEN_COORDS  // map texture to the screen
MOV    oT1, scale * ( 1, 2^(-L), 2^(-2L) )

(PS 2)
ps.1.3
TEX     t0          // read RGB-encoded value
TEXDP3  t1, t0      // decode and output color from t1 texture
MOV     r0, t1      // output the color value

(PS 3)
ps.2.0
// c0 = scale * ( 1, 2^(-L), 2^(-2L) )
dcl          t0.xyzw
dcl_2d       s0          // A texture with RGB-encoded values
dcl_2d       s1          // A color ramp mapping value to color
TEXLD     r0, t0, s0     // read the RGB-encoded value
DP3_SAT   r0, r0, c0     // decode the RGB-encoded value
TEXLD     r0, r0, s1     // convert value to color
MOV       oC0, r0        // output the color
```

These shader fragments work for RGB-encoded values with both positive and negative values in the color channels. They also work for RGB-encoded values that result from adding or subtracting two or more encoded values. As you'll see in the next section, two sums of RGB-encoded values are easily subtracted and decoded. The result of decode(A) – decode(B) is identical to the result of decode(A – B), which is very convenient! At each pixel we can easily calculate the front face and back face depth sums, subtract them to get an RGB-encoded thickness value and convert this into a color for the volume object.

Rendering Thick Volumes
With Nothing Intersecting the Volumes

Applying RGB-encoding to the method of computing an object's thickness gives us a way to render ordinary polygonal objects as thick volumes of material. This section gives a step-by-step discussion of the rendering for volumes in free space where no opaque objects intersect the volumes. A few issues related to each step are also presented, and I'll focus on the ps.2.0 implementation. The ps.1.3 implementation is almost identical, and information about the differences is included in the demo source code. Situations where objects intersect the volumes are far more common, and these will be covered in the next section.

Rendering Process

To render volumes in free space, two off-screen texture render targets are needed. These render targets could be a lower resolution than the ordinary back buffer, but if the rendered volumes have high contrast edges, the render targets should match the back buffer size to reduce aliasing. These color render targets may or may not need an associated depth buffer. It depends on whether or not the solid objects in the scene can occlude the volumes, and it depends on the geometry used to render the final volume object color into the back buffer. A simple approach which handles occlusion is to use a depth buffer and to render the final color with a large quad covering the entire screen. In that case, rendering proceeds as follows:

1) Render the scene to the color and depth buffers as you normally would with no volumetric objects.

2) Switch to an off-screen texture render target; call it the 'back faces' target.
   In Direct3D, the depth buffer can be shared between the back buffer and the texture render target if they are the same size and multisample type. Otherwise, the depth of occluders in the scene needs to be rendered again into a separate depth buffer that matches the texture render target.

   a) Clear the render target to black.
   b) Set the cull mode to render volume object back faces.
   c) Set the depth test to less-equal, and disable depth writes.
   d) Render the RGB-encoded depth of all back faces of the volume objects with additive blending to the color target.
      Where several back faces overlap, the encoded depths will be added to form a sum of all back face depths at each pixel. The result will be similar to Figure 1c., where colors are shown over-bright to better illustrate their values.

3) Switch the color render target to the other off-screen render target; call it the 'front faces' target.

   a) Clear it to black.
   b) Render the RGB-encoded depth of all volume object front faces to create the front face depth sum. The result is similar to Figure 1d.

4) If you are using hardware that supports pixel shaders 2.0, switch to the ordinary color and depth back buffers from step 1.

   a) Disable depth testing.

   b) Render a single quad covering the entire back buffer with the pixel shader listed in PS 4. This shader builds on the shader from listing PS 3. It samples the depth sums in the 'back faces' texture, samples the 'front faces' texture, computes the object thickness, and converts the thickness to the volume object color. It converts the thickness to a color value using an arbitrary color ramp texture bound to the D3D S2 sampler. The color ramp can be created by an artist or computed from a mathematical model of light scattering. For hardware that supports only pixel shaders 1.3, an extra pass and texture render target are needed to compute the RGB-encoded thickness value and supply this to the shader PS 2 listed above.

```
(PS 4)
ps.2.0
// c0 = scale * ( 1, 2^(-L), 2^(-2L) )
dcl          v0.xyzw
dcl          t0.xyzw
dcl_2d       s0                  // back face depth sum
dcl_2d       s1                  // front face depth sum
dcl_2d       s2                  // color ramp

TEXLD   r0, t0, s0      // back face depth sum
TEXLD   r1, t0, s1      // front face depth sum
ADD     r0, r0, -r1     // RGB-encoded thickness = back – front
DP3_SAT r0, r0, c0      // decode to floating-point coordinate

TEXLD   r0, r0, s2      // convert thickness to fog color
MOV     oC0, r0
```

An alternate approach is to use the volume object geometry instead of a full-screen quad to drive the computation of volume object color and rendering to the back buffer. The choice depends on the coverage and complexity of the volume objects, and you can switch between methods depending on the viewpoint and performance. This geometry provides the appropriate pixel coverage on screen. It creates pixels over an area so the pixel shader receives input and can perform the computations. If we use a simple full-screen quad, we waste fill rate rendering pixels where there is no volume thickness. If we use the volume objects themselves, we might reduce the fill rate by drawing pixels only where the volumes are, but we could spend more time transforming vertices or passing over pixels more than once where the depth complexity of the volume objects is greater than one. Since alpha blending is used to blend the volume's color into the scene, the depth complexity is important. If we use the volume object geometry, we need to enable a stencil or destination alpha test to avoid blending the volume color more than once at each pixel where the depth complexity might be greater than one.

If we use the volume objects to drive the processing, we need a shader which projects the front and back face depth sum textures from steps 2 and 3 onto the volume object geometry so that the pixel shader receives the correct values for each point on screen. This is simply a matter of turning the screen-space position into a texture coordinate from [0,1] across the full screen. A Direct3D vertex shader code fragment for this is listed in VS 3. This code is also used in handling solid objects that may intersect the volumes, since it can project rendered texture information at each pixel onto the same pixels as they are rendered again, regardless of the shape of the geometry. The code is useful in many multipass approaches, so it's good to keep in mind for other effects.

```
(VS 3)
vs.1.1
// Transform position to clip space and output it
DP4 r1.x, V_POSITION, c[CV_WORLDVIEWPROJ_0]
DP4 r1.y, V_POSITION, c[CV_WORLDVIEWPROJ_1]
DP4 r1.z, V_POSITION, c[CV_WORLDVIEWPROJ_2]
DP4 r1.w, V_POSITION, c[CV_WORLDVIEWPROJ_3]
MOV oPos, r1
  // Convert geometry screen position to a texture
  coordinate,
  // so we can project previously rendered textures to the
  same
  // pixels on screen for any geometry.
  // CV_CONSTS_1 = ( 0.0, 0.5, 1.0, 2.0 )
MUL r1.xy, r1.xy, c[CV_CONSTS_1].yyyy
  // Add w/2 to x,y to shift from (x/w,y/w) in the
  //  range [-1/2,1/2] to (x/w,y/w) in the range [0,1]
MAD r1.xy, r1.wwww, c[CV_CONSTS_1].yyyy, r1.xy
  // Invert y coordinate by setting y = 1-y
  // Remember, w!=1 so 1.0 really equals 1*w
  //  and we compute y = 1*w - y
ADD r1.y, r1.w, -r1.y
  // Add half-texel offset to sample from texel centers
  //  not texel corners (a D3D convention)
  // Multiply by w because w != 1
MAD r1.xy, r1.wwww, c[CV_HALF_TEXEL_SIZE], r1.xy
  // output to tex coord t0
MOV oT0, r1
```

The steps above work for any camera position in the scene. The camera can move through the volume objects and the rendering will remain correct, with a volume thickness contribution from only the part of the volume in front of the near clip plane. This is a consequence of choosing our RGB depth-encoding to start from 0 at the near clip plane. Without this, the volume object's polygons would have to be clamped or capped at the near plane so their depth values are not clipped away.

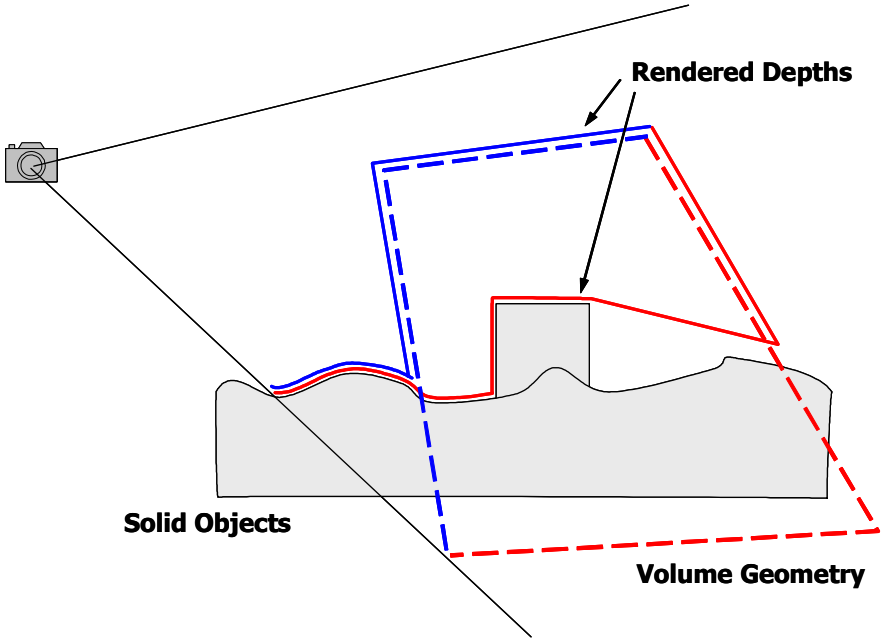
These steps require the volume objects to be closed hulls. There can be no back face without a corresponding front face, and vice versa. The depth complexity of the volume objects must also be kept below the limit for adding the RGB-encoded values together. The front and back faces are summed to separate render targets, so the depth complexity limit is for only the number of back or front faces, not for the total of front and back faces. In practice, using 12-bit encoding (`L=4`) allows 16 front and back faces (see the table of Figure 3), which is more than enough for interesting scenes. The approach also sums the thicknesses of overlapping volumes, so that where two volumes intersect, the thickness will be greater.

Everitt's technique of depth peeling [Everitt02] could be applied to eliminate the overlapping areas, but this would require several more passes and might be too costly in terms of performance.

Handling Solids Intersecting the Volumes

Often, we want opaque objects to pass through the volume objects, or we want to place the volumes in a scene without having to worry about their polygons being clipped away by solid objects. Handling the areas where solid objects intersect the volumes is key to making the technique easy to use. Fortunately, even complex intersection cases are easy to handle using one additional render target texture and a comparison of RGB-encoded values in the pixel shader.

If an opaque object cuts through a volume object's hull, we have to use the depth to the opaque object instead of the depth to the volume object faces that are occluded by the opaque object. This ensures that we get thickness contributions for only the part of the volume that is visible in front of the opaque object. This can be accomplished by doing a comparison of depth values in the pixel shader as each face of the volume objects are rendered. The pixel shader that created the RGB-encoded depth value (as in listing PS 1) can also sample a texture containing the solid object depth. This texture holds the RGB-encoded depth of the solid object closest to the near plane. The pixel shader compares the RGB-encoded depth of the volume object pixel being rendering to the encoded depth of the nearest solid object and outputs whichever value is the lesser depth. This allows both the back faces and front faces of volume objects to be occluded by the solid objects. It is an efficient way to handle any and all solids that might penetrate the volumes, and it handles complex volumes of any shape or depth complexity. Where a volume object face goes inside or behind a solid object, its depth contribution becomes the depth of the solid object. The volume object is effectively clamped to always begin from the solid object, and the varying depth complexity of concave or folded volume objects is handled correctly in all cases (Figure 6).



The volume geometry is shown with a dotted line. A pixel shader compares the volume object depth to the solid object depth read from a texture and outputs the lesser depth value. This results in depth information being taken from the geometry shown with solid lines. The depth comparison clamps the occluded volume object pixels to the nearest solid object depth, effectively limiting the volume object thickness to the proper amount for all intersection cases.

Figure 6. Handling Opaque Objects Intersecting the Volume Objects

To implement this approach, first render a texture to hold the RGB-encoded depth of the nearest part of any opaque objects that may penetrate or occlude the volume objects. There is no need to render all the opaque objects in the scene to this texture, and the regions of intersection do not have to be computed. Next, render the volume object faces according to steps 2 and 3 from the previous section, but in the pixel shader, sample the solid object depth texture, perform the depth comparison, and output the lesser depth. The depth is either the opaque object depth read from the texture or the depth of the volume object face at that pixel. Shaders to perform this on ps.1.3 and ps.2.0 hardware are shown in listings PS 5 and PS 6.

The depth comparison barely fits within the limits of a ps.1.3 shader. Unfortunately, Direct3D API restrictions require an additional instruction slot for the CMP instruction on ps.1.1 hardware, so this comparison can't be expressed in a ps.1.1 shader. Also note that the ps.1.3 shader can't decode the RGB-encoded values to a high precision scalar, so it relies on comparing each R, G, and B channel separately. It scales and clamps each R, G, and B difference to [-1,1]. The ps.1.3 comparison will not work for RGB-encoded values where the high carry bits are used, but this doesn't present a problem. Additional comments are provided in the demo's shader source code. Since the ps.2.0 shader operates at floating point precision, it is more simple and can handle values where the carry bits are on.

```
(PS 5)
ps.1.3
  // RGB-encoded depth comparison.
  // Outputs the lesser RGB-encoded value
  // Requires saturation to [-1,1] range
  // Weight for each of the RGB channels
DEF c7, 1.0, 0.66, 0.31, -0.66
  // CMP uses >= 0.0, so use this small bias to get a
  //  "less than zero" comparison
DEF c6, -0.01, -0.01, -0.01, -0.01

TEX t0           // red+green ramp texture
TEX t1           // blue ramp texture
TEX t3           // depth of solid objects
ADD t2, t0, t1   // Add R + G + B to make depth value

  // Difference the pixel depth vs the solid object depth
  // Use *4 to increase the contrast.  The goal is to saturate
  //  each R,G,B channel of the signed number the values
  //  -1, 0, or +1.
ADD_x4 r1, -t3, t2       // diff * 4
ADD_x4 r1, r1, r1        // diff * 32
ADD_x4 r1, r1, r1        // diff * 256
  // DP3 the saturated difference with the c7 weights.
  // The result is positive, negative, or zero depending on the
  //   difference between the high precision values that t3 and t2
  //   represent
DP3_x4 r1, r1, c7
  // Subtract a small value from r1
ADD r1, r1, c6
  // Compare r1 decision value to zero.  If r1 is positive,
  //  output t3, otherwise output t2
CMP r0, r1, t3, t2
```

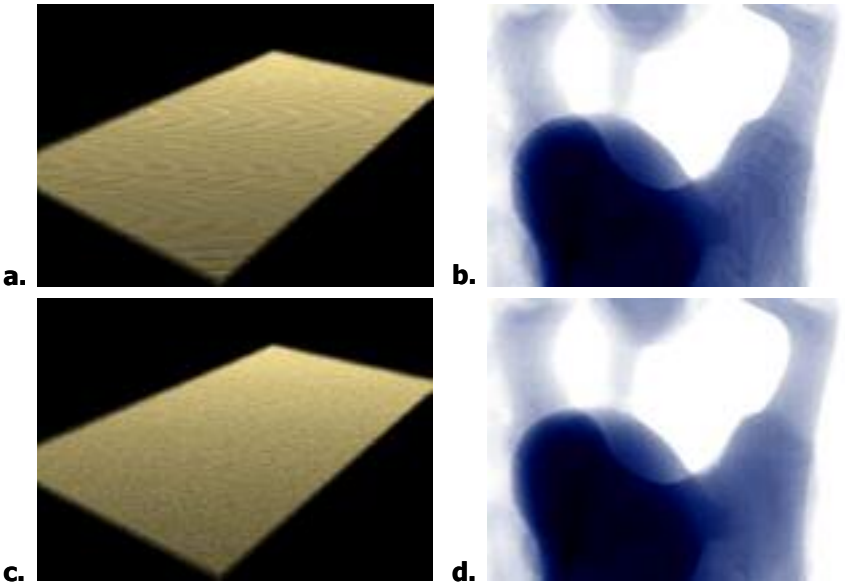
```
(PS 6)
ps.2.0
// Comparison of RGB-encoded depths
// Outputs the lesser RGB-encoded value
// c0 = scale * ( 1, 2^(-L), 2^(-2L) )
dcl          t0.xyzw
dcl          t1.xyzw
dcl          t3.xyzw
dcl_2d   s0         // red+green ramp texture for depth encode
dcl_2d   s1         // blue ramp texture for depth encode
dcl_2d   s3         // RGB-encoded depth value of nearest solid

TEXLD    r0, t0, s0   // red+green part of depth encoding
TEXLD    r1, t1, s1   // blue part of depth encoding
```

```
ADD      r0, r0, r1   // Depth of volume object's pixel
TEXLDP   r1, t3, s3   // RGB-encoded depth from texture at s2
  // RGB-encoded difference
ADD      r2, r0, -r1
  // Decode to positive or negative value
DP4      r2, r2, CPN_RGB_TEXADDR_WEIGHTS
  // Choose the lesser value:  r2 >= 0 ? r1 : r0
CMP      r3, r2.xxxx, r1, r0
MOV      oC0, r3
```

Dithering The Low Bit

In this technique, depth is represented by discreet values, so aliasing can appear in the thickness values. This aliasing is always present. Even at high precision, it can be noticeable, especially if thin objects have their thickness multiplied by a large scale factor in order to generate some visible contribution. Depth aliasing appears as sharp transitions between light and dark in the color of the rendered volume, shown in Figure 7, a and b. Luckily, there is a painless way to dither the lowest bit of depth information and this break the sharp bands into dithered transitions that appear smooth. The results are shown in Figure 7, c and d.



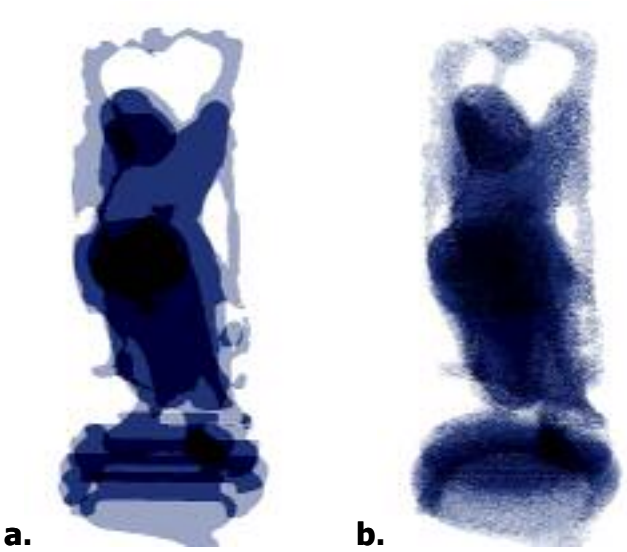
Artifacts of depth aliasing in severe cases are shown in **a** and **b**. Low depth precision is used to accentuate the effects. Dithering the depth information breaks the artifacts into gradual noisy transitions, which appear smooth as shown in **c** and **d**.

Figure 7. Artifacts of Depth aliasing in Severe Cases

The dithering is very easy to implement. The lowest bit of depth, held in the blue color channel, is read from a small color ramp texture as described above. To dither the depth information, all we have to do is dither the color ramp texture. As long as the texture coordinates used to access the color ramp is not aliased, this approach works very well. The texture coordinate is a high precision floating point value with more precision than the 12 or 15 bits of depth precision typically used. The demo source code has a few functions to create dithered color ramps.

To access the dither pattern, a 2D texture coordinate is used instead of the 1D coordinate oT1 (.x only) of listing VS 2. The dither pattern varies in the second (.y) coordinate direction, so the Y coordinate can vary randomly to create gradual dithered transitions. There is some subtlety involved in wrapping the dither pattern in the X direction from one end of the texture to the other, so that dithering continues where the color ramp repeats. This involves using the alpha channel of the blue ramp to hold a value which represents a negative increment to the next highest (green) bit.

The dithered depth information can be used to create interesting noise effects in the volume rendering. As the depth precision is lowered, the dithered thickness becomes progressively noisier. At just a few bits of depth precision, the volume shape and thickness remain recognizable, and the rendering appears as though it were coming from a noisy video source. An example is shown in Figure 8.



A curious volume noise effect is produced by an object spanning only three bits of depth precision. The depth-aliased rendering is shown in **a** where only a few bit-increments of thickness occur. Dithering the least significant bit of depth during depth encoding gives the result in **b** which is a close but noisy match to the actual volume

Figure 8. Producing Noise Effects

Negative Thicknesses

Objects can be treated as contributing negative thickness to the volume rendering. This could be used to render shafts of shadow cutting through a volume or to modulate the thickness of objects. It is accomplished by simply reversing the front and back faces for negative objects. To render an object as subtracting from the total thickness, it's front facing triangles are rendered to the 'back faces' depth sum texture, and the back facing triangles are rendered to the 'front faces' depth texture. This approach is not robust, and works correctly only if the negative objects lie entirely within a positive thickness. If two negative volumes overlap or if a negative volume extends outside of a positive volume, they will still subtract from the total thickness. This would create areas of over-subtraction where the total thickness ends up being too thin. For some situations, the effects of this are not problematic, but to properly handle such cases, Everitt's technique of depth-peeling [Everitt02] could be used. This requires several more rendering passes and would have a substantial impact on frame rate.

Additional Thoughts

For slower hardware, you may want to use a single texture for both the front faces and back faces depth sums. This texture would begin from a mid-range value, such as RGB=(128,128,128). The back face depths are added to this, and the front face depths are subtracted using subtractive framebuffer blending. The advantage is that one less non-power-of-2 texture needs to be read. The disadvantage is that the depth complexity that can be handled in a single pass for a given choice of RGB-encoding precision is half of what it is when two separate render target textures are used.

The method of RGB-encoding can be extended to higher precision and depth complexity by spreading the bits across multiple render targets. For example, you could use two RGBA-8 surfaces to encode 64-bit values. Older hardware will run out of precision in the texture coordinate interpolation used to read the color ramp textures, but newer hardware can calculate the color ramp values in the pixel shader itself.

The method of rendering RGB-encoded depths to a texture and projecting these back onto objects can be used to implement shadow mapping. Depths can be rendered from the point of view of a light, and any number of passes can be used to implement complex and customizable filtering of the shadow map comparisons. This could provide high quality hardware-accelerated shadow mapping for near-real-time applications.

Going a step further, the thickness through an object to a light source can be computed and used to render translucent materials like jade or to approximate the appearance of sub-surface scattering. A future article and demo will present this approach. It is also possible to render shadows from semi-transparent objects where the shadow darkness depends on the thickness through objects.

Conclusion

Programmable shaders and a few render-to-texture passes are all that is needed to render ordinary polygon objects as thick volumes of light scattering material. This article presents an efficient and direct means to render and accumulate high precision values using 8-bit-per-component render targets, to handle objects intersecting and occluding any volume object shape, and to eliminate aliasing artifacts. The approach works for any viewpoint in the scene, and it is trivial to animate the volume geometry. The technique can be used on the large installed base of Direct3D8 ps.1.3 hardware. When hardware supports additive blending to floating-point render targets, the method of RGB-encoding can be abandoned to simplify the implementation.

Using thickness to determine the appearance of objects offers exciting new possibilities for real-time interactive rendering. Scenes can be filled with dynamic wisps of fog, clouds, and truly volumetric beams of light that are easily created and controlled. Intuitive controls and color ramps govern the appearance of the volume objects, though more sophisticated treatments of scattering could also be employed.

Example Code

Example code and additional images for this technique and others are available from NVIDIA's Developer Website:

http://developer.nvidia.com

http://developer.nvidia.com/view.asp?IO=FogPolygonVolumes

References

[Baker02] Dan Baker, "VolumeFog" D3D example, Microsoft D3D8.1 and D3D9 SDKs, http://www.microsoft.com

[Everitt02] Cass Everitt, "Order Independent Transparency," http://developer.nvidia.com/view.asp?IO=order_independent_transparency

[Hoffman02] Naty Hoffman, Kenny Mitchell, "Photorealistic Real-Time Outdoor Light Scattering." in Game Developer Magazine, CMP Media, Inc., Vol 9 #8, August 2002, p32-38

[Mech01] Radomir Mech, "Hardware-Accelerated Real-Time Rendering of Gaseous Phenomena." Journal of Graphics Tools, 6(3):1-16, 2001, http://www.acm.org/jgt/papers/Mech01

Notice

Trademarks

Copyright

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