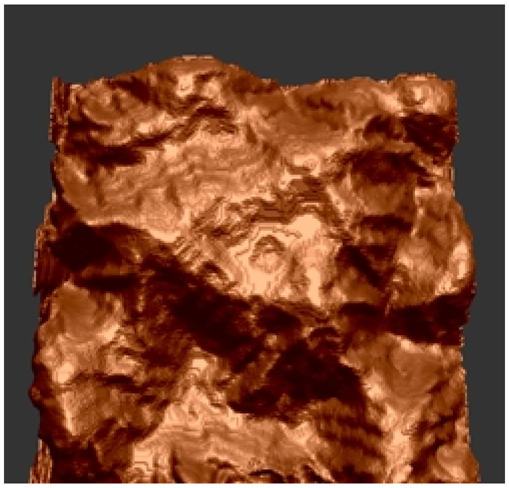
Elevation Maps

NVIDIA Corporation
Sim Dietrich
Please send me comments/questions/suggestions
mailto:sim.dietrich@nvidia.com

Introduction

Most applications running on modern 3D accelerators such as the RIVA TNT2 and the GeForce 256 use two features to add visual detail to their scenes, textures and triangle geometry. This paper describes Elevation Maps, another approach that combines some of the advantages of both textures and triangles. Elevation maps allow extremely detailed visuals at the cost of a few polygons. This screenshot requires only 10 quads to render.



Geometry is specified with triangles. More triangles can potentially mean more visual cmpplexity, if used properly. However, there is a limit to how many triangles even the GeForce 256 GPU can process in a frame and still achieve real-time frame rates. To maintain a high-level of visual complexity, some triangle scalability solution must typically be employed to add or remove triangles on a frame by frame basis. Scalability solutions often tend to work on a per-object basis, and do not always lend themselves to pixel-level detail control. Thus, to achieve detail up close, many triangles must be generated throughout a given model, thus wasting throughput by drawing triangles that may span less that one pixel in area.

Textures allow per-pixel level of detail, but have no depth information, so they can look artificially flat up close. Bump maps can compensate to some degree, but don't display parallax behavior. Additionally, bump maps don't show up on silhouette edges. Also bump maps are not Z correct, so don't interact properly with a depth buffer.

It would be terrific if there were some way to combine texture-style level of detail with the spatially correct aspect of triangle geometry. Elevation Maps provide such a solution.

Elevation Maps are similar to Relief Textures, developed at UNC, and Nailboards. However, they are distinct enough from these to warrant their own name to avoid confusion.

The following pages will give an overview of the technique, as well as a discussion of all three techniques, triangles, textures and Elevation Maps.

Figure 1 contains a height field, represented as a gray scale image. This image corresponds to the alpha channel of the Elevation Map.

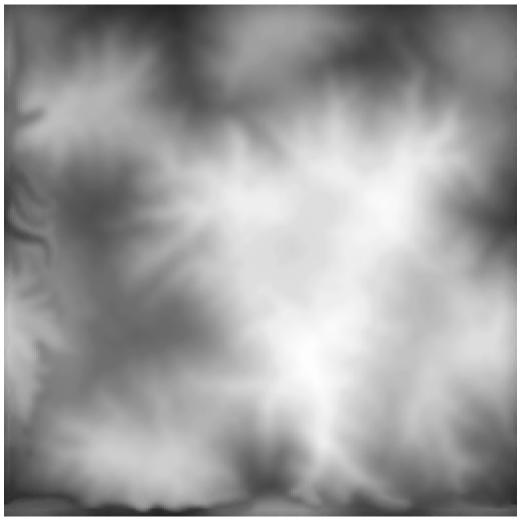


Figure 1.

To represent this height field with geometry, we could take three neighboring texel height values, form a triangle, compute vertex normals, and render each triangle. We could use a binary triangle tree or other level-of-detail approach to avoid rendering flat or distant areas with too many triangles.

To render this as a texture, we could simply draw it on a flat quad as shown above. Obviously the texture would not be Z correct with respect to the height values and would appear flat. It would only look nearly correct when viewed from directly above the polygon on which it is mapped. The advantage of this approach compared to drawing the height field as triangles is that only two triangles need to be drawn, rather than the thousands necessary to draw the terrain as triangles.

With Elevation Maps, we can have the advantages of both approaches.

How do Elevation Maps Work?

The basic idea behind Elevation Maps is to render multiple slices of the object being modeled, like an MRI medical scan, while using a texture's alpha channel to represent the height of each texel. The height is measured from some basis surface, such as a plane, cylinder or sphere. For the purposes of discussion, it's easiest to begin with a basis plane.

To get true volumetric rendering, one could simply render slices of an object, with each slice represented by a separate texture. Of course, that could be prohibitively expensive to store so many textures. Also it is wasteful because the interior of the volume is not typically visible. Elevation Maps leverage alpha testing and multi-texture in order to achieve a volumetric effect with a single 2D texture.

For the above height field, which represents mountains, we would draw a certain number of quads, representing slices through the data set. The quads would be stacked vertically.

Think of the dataset as existing in a transparent glass box. The quad slices drawn would represent a horizontal scan through the box.

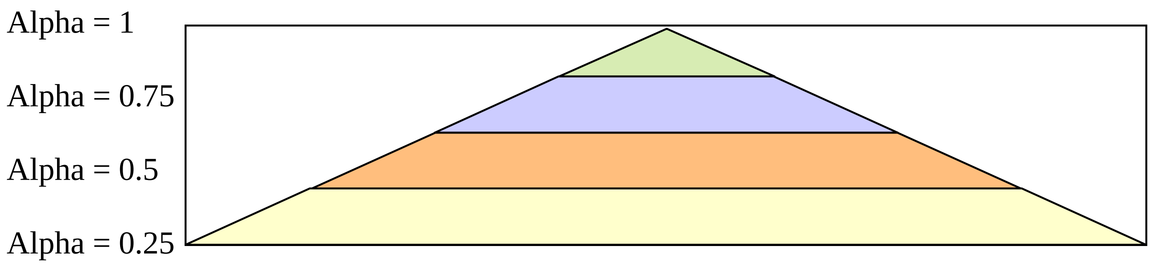


Figure 2 – Visualizing the Alpha Channel

Each color represents what would be drawn at each scan through the box. In this example, four quads are drawn to represent the four distinct height values.

For instance, the lowest slice would only draw the lowest texels, the slice above it would draw the lower and some higher texels, and so on.

Since we are rendering geometry via the quad slices, X,Y & Z are not faked but actually pixel perfect. By using alpha test to throw away transparent portions of surfaces, we also get the correct depth values in the Z buffer. Because Elevation Maps use both geometry and textures together in one primitive, they have correct spatial coherency, can be mip-mapped and filtered, and when combined with normal maps, can be dynamically lit on a per-pixel basis.

How does alpha test allow us to store height values?

What we really would like the GPU to compute is :

```
if ( Height of Terrain > Height Of Current Slice ) then Draw Pixel
```

We can use a combination of multi-texture and alpha test to perform this test on a per-pixel basis.

If we store the terrain height values in the texture alpha and the height of the current slice in diffuse alpha, we would set the alpha test function to perform :

TEXTURE_ALPHA > DIFFUSE_ALPHA

In both Direct3D and OpenGL, there is no way to explicitly test alpha this way. The alpha test is always compared to a constant factor, set to a value ranging from 0 to 255.

But, we can perform a little algebra to get :

TEXTURE_ALPHA - DIFFUSE_ALPHA > 0

Now the test is in terms of a constant. We can set up the Direct3D texture stages to perform the subtraction, using the D3DTOP_SUBTRACT texture blend mode, and use alpha test against an alpha factor of zero. Thus alpha test not only achieves transparency, but also prevents the low texel heights from being drawn on the high quad slices.

```
pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );
pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SUBTRACT );
pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE );
```

Elevation Map Example

Figure 3 is an example of the height texture method in use. Note how it appears to be extremely visually complex, although Figure 1 represents only **14 quads**. Note the correct Elevations that really help give it a sense of dimension along the silhouettes. This could not be achieved with only a single bump mapped quad.

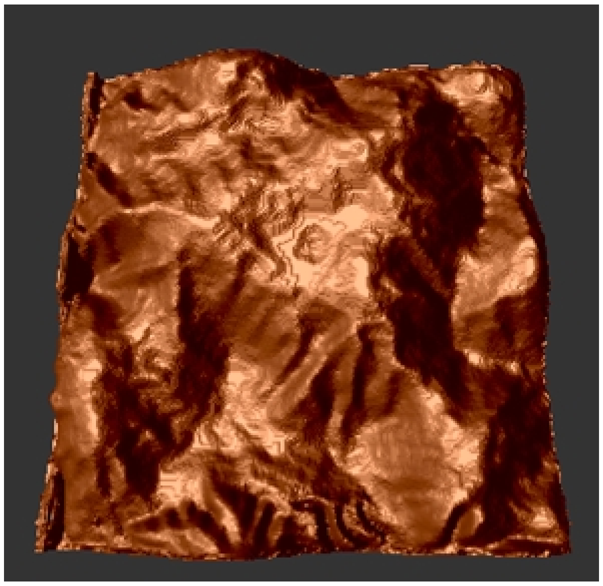


Figure 3 – Elevation Map

Elevation Map Wireframe

In Figure 4, we see that the height texture is actually only 14 quads, generating much more visual complexity than would seem possible with such a low number of triangles.

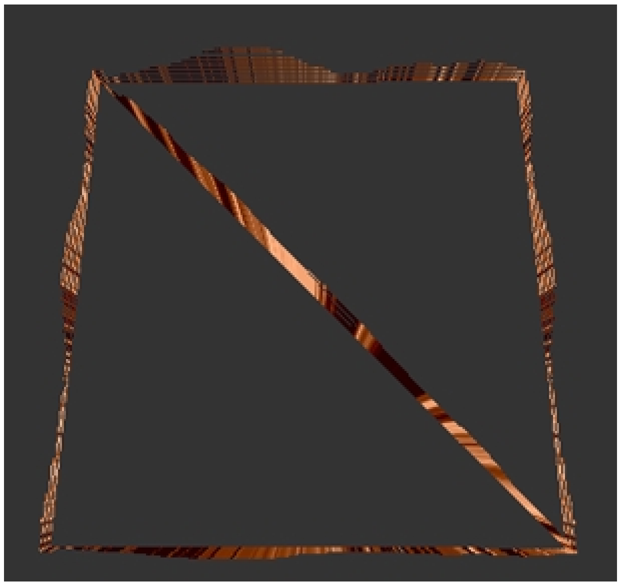


Figure 4 – Elevation Map Wireframe

What about Color?

To achieve the mountain effect shown above, only the alpha channel of the texture is needed for height testing. The RGB color channels are available for a color or normal map.

The mountain example program actually takes the alpha height field and replaces any RGB colors in the texture with a normal map for the surface. After the conversion process, each RGB color represents the direction of the surface normal at that point in space. This allows each pixel to be dynamically and individually lit. Below is an example showing the mountain's unaltered, unlit normal map.

Example showing unlit Normal Map

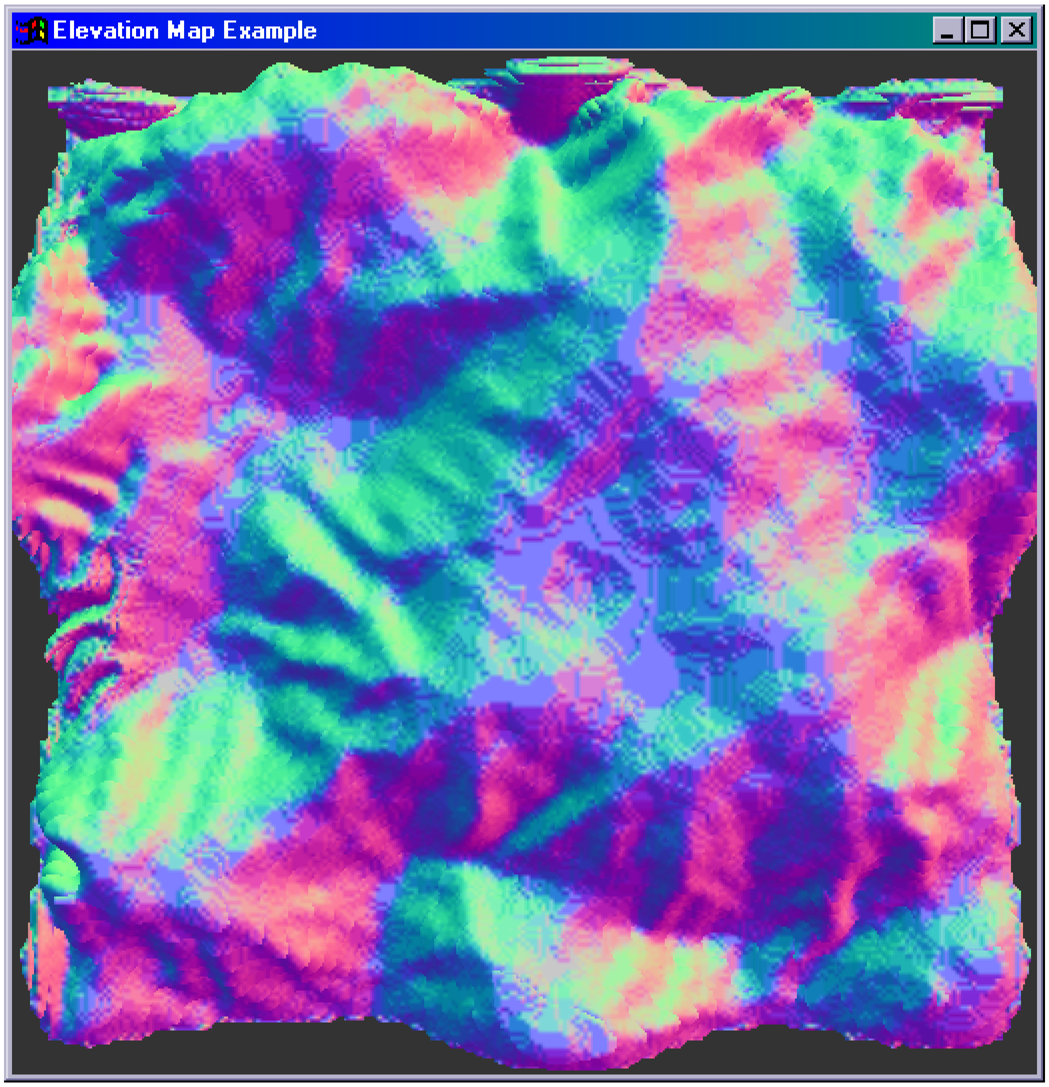


Figure 5 – Normal Map

How are the Mountains Lit?

The mountain terrain is lit by performing the dot product between a directional light vector, stored in D3DTA_TFACTOR and the normal map (which represents surface normals) stored in D3DTA_TEXTURE.

```
pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_DOTPRODUCT3 );
pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_TFACTOR );
```

The result of the dot product is a grayscale replicated into RGB, and optionally Alpha. For our purposes, we are already using alpha for the height testing, so we don't replicate the dot product into Alpha.

In the next multi-texture stage, we bring in the diffuse color, either by modulating or adding to the lighting calculated in stage zero.

```
pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_DIFFUSE );
pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_MODULATE );
pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_CURRENT );
```

Or, for a pseudo-specular effect, add in the color contribution as a signed value :

```
pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_DIFFUSE );
pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_ADDSIGNED );
pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_CURRENT );
```

What else can Elevation Maps be used for?

Figure 6 shows Elevation Map-style grass blades. The alpha at each texel in the source texture represents the height of each grass blade. The color can be used either for the grass blade's color or for a surface normal. This way each blade could be dynamically lit.

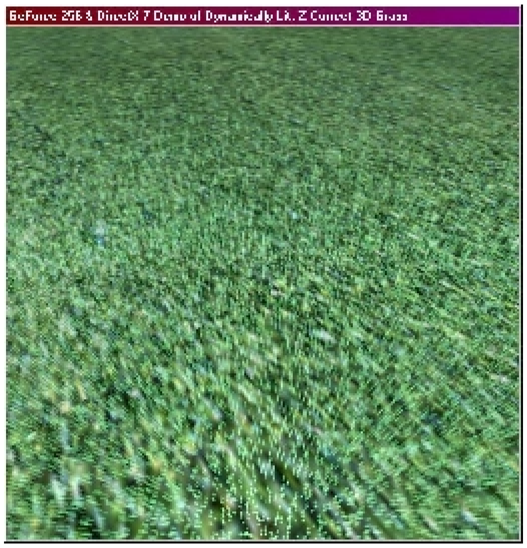

GeForce 256 & DirectX 7 Demo of Dynamically Lit, Z Correct 3D Grass

Figure 6

Cloud Layers using Elevation Maps

Elevation Maps can also be used to make layered clouds. Again, the color channels are used for a normal map, thus allowing dynamically lit clouds.

Here is an example screen shot :

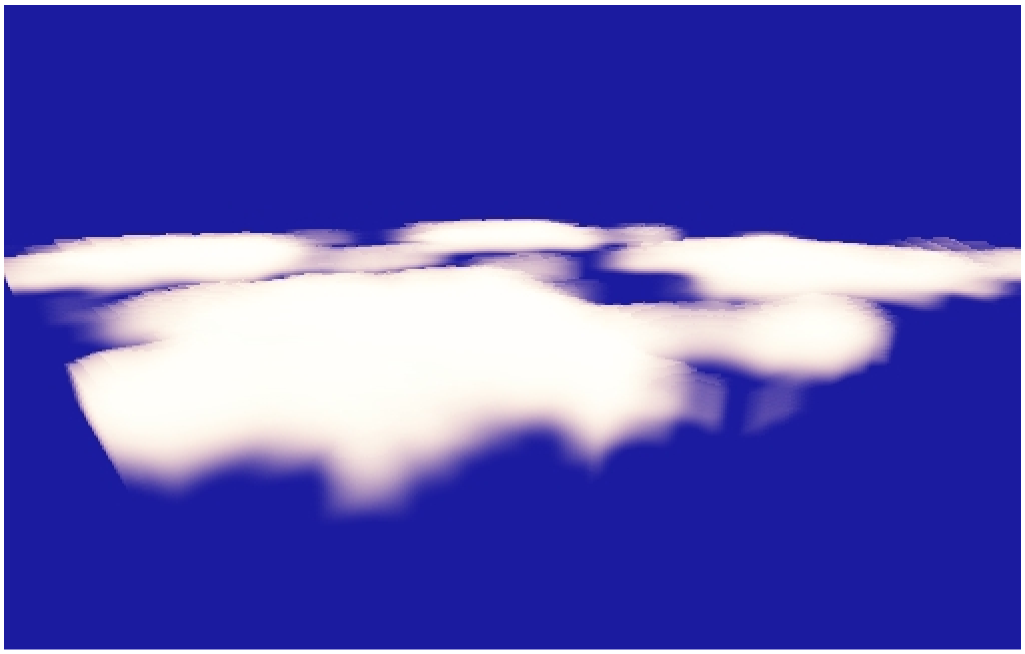


Figure 7 – Volumetric, Lit Clouds

When do Elevation Maps make the most sense?

Elevation Maps are a particular way to draw one or more objects via multiple layers. They can cost fill rate, because they tend to increase the depth complexity of the scene. On the other hand, they can save geometry in some situations, and can actually be roughly equal in depth complexity for extremely complex and dense objects, such as a field of flowers or mushrooms or a particle system.

One could just use separate textures, rather than the alpha height technique discussed in this paper, but it costs more texture memory. Alpha Elevation Maps are a good technique when trying to accurately represent an extremely complex surface area. They can be considered a form of voxel, and voxel representations are most useful when representing data whose surface can't be efficiently represented by a modest number of triangles.

Another example of Elevation Maps is as a form of detail texture or bump map. Here is an example of brick implemented with height maps:

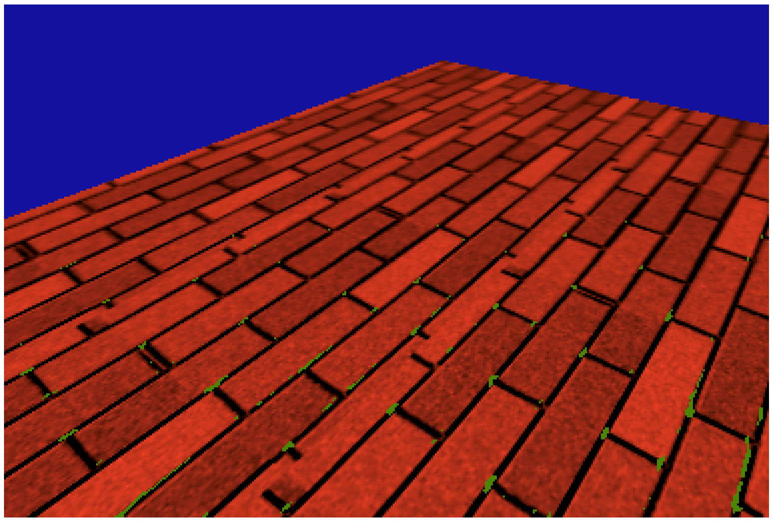


Figure 8 – Volumetric Bricks

Alternate uses of Elevation Maps

There are other uses for the alpha comparison concept, including offset elevation maps and shadow buffers.

Standard elevation maps could be used to represent a hemisphere. However, a single elevation map could not be used for an entire sphere, due to the fact that two height samples are required, one for the height above the reference plane of the sphere bottom, and the second for the height of the sphere top. It turns out we can use the alpha from two textures to achieve this effect using Offset Elevation Maps. In general, this technique allows one to displace the volume away from the basis plane on a per-pixel basis. The alpha test expression can be written:

```
if ( Height of Bottom of Sphere < Height Of Slice ) and
   ( Height of Top of Sphere > Height Of Slice ) then Draw Pixel
```

This test can be reworked into the following expression:

```
if (ClampTo0(sliceHeight – bottomHeight) * ClampTo0(topHeight – sliceHeight) > 0)
        DrawPixel();
```

Review the following logic table:

		Slice below bottom	Slice above bottom
		ClampTo0(sliceHeight-bottomHeight) == 0	ClampTo0(sliceHeight-bottomHeight) > 0
Slice above top	ClampTo0(topHeight-sliceHeight) == 0	Shouldn’t ever happen if top > bottom	Slice too high
Slice below top	ClampTo0(topHeight-sliceHeight) > 0	Slice too low	Slice within volume

Because the subtractions clamp to zero, the expression will be non-zero if the slice is between bottomHeight and topHeight, and zero otherwise. This gives the effect of a logical AND. Unfortunately, there is no way to express this in a single pass with Direct3D 6 or 7.

It can be accomplished with the stencil buffer in two passes, however.

Under OpenGL this technique can be accomplished using the NV_register_combiners extension in a single pass.

Note that with this approach normal maps won’t work in general because each vertical sample of the volume will have the same color. Therefore any RGB normals on the top and bottom sections would have to be the same. This works out fine for vertically symmetrical objects such as upright cylinders, but not in general.

Alpha Maps for Depth Masks and Shadow Buffers

One way to get fast, accurate shadows is to render a scene from the point of view of a light-source, capture the depth buffer, and then perform projected depth comparisons when rendering the scene to test whether each pixel drawn is the closest to the light-source. A major advantage of this technique over stencil shadows and projected shadows is that it can be used to ensure only the closest object to the light is lit.

With current mainstream accelerators, however, there is no way to use the depth buffer in this way.

It turns out that rather than performing a depth or stencil test, we can use the alpha test instead. We can render the scene into a 32 bit texture from the light viewpoint. The color drawn will correspond to a grayscale, where 0x01010101 corresponds to exactly on the light’s viewplane, and greater values indicate greater distances away. The distribution of values can be controlled by using a texture lookup table containing an alpha ramp.

Next, when objects that may be lit by the light or may be in shadow are rendered, the light’s alpha texture is selected as texture 0, and the object’s projected position into the lookup alpha texture are calculated via a combination of camera space position texture coordinate generation and the texture matrix. The two alpha values are then compared using the subtraction technique described above. A resulting alpha value of zero indicates either that the pixel’s position is behind the light’s viewplane, or that another point was closer to the light, therefore the pixel should not be affected by the light.

Thus one can use the alpha test to determine whether to light an object with a particular light. For a point light, this essentially means creating a cube map from the light position. To avoid performing detailed culling or clipping to the light’s view volume, one can use clamp texture addressing with zero alpha along all borders. This will correctly avoid lighting pixels that are outside of the light’s effective area.

One could use the same idea to test against not only an alpha shadow buffer texture, but an Elevation Map of an arbitrary design. This would allow objects to be intersected with any volume that could be described via a planar Elevation Map on a per-pixel basis, allowing CSG operations.

For instance, the hemisphere map described earlier could be used as half of a point light source’s attenuation function. When drawing an object near the point light, generate alpha values corresponding to the distance from the light’s reference plane and test against the alpha values in the hemisphere map. If the alpha test passes, then the pixel drawn is within the volume of the hemisphere and can be lit.

In Summary

To summarize the advantages of this Elevation Maps :

1. Uses multi-texture to perform height testing, thus avoiding having to change the alpha test reference value on a per-quad basis which would be very slow
2. Uses alpha height testing to leverage the same texture, rather than a texture per quad layer
3. Uses alpha only, leaving the color channels available for other lighting and effects
4. Can represent complex geometry with just a few simple shapes such as quads
5. Runs on any card with support for alpha test and alpha subtract multi-texture
6. Makes a great proxy for far away objects such as terrain sections
7. They make a better billboard – use several layers rather than a single billboard
8. They can be filtered, effectively giving anti-aliasing within the primitive for free, at least along the surface of the basis primitive

9. Makes a great bump map, giving accurate Z values and parallax effects close to the viewer
10. When combined with the D3DTOP_DOTPRODUCT3 blending mode, can be used to create dynamically lit volumetric effects
11. Could be used to port a voxel-based engine to 3D hardware acceleration
12. Can be combined with projected textures to provide depth-correct lighting and shadows
13. The number of slices drawn can be dynamically scaled based on view distance, thus only paying for the extra overdraw where the detail is noticeable.

The disadvantages are :

1. Potentially high depth complexity cost. Although many pixels may get alpha tested away and thus not drawn, they still have some cost.
2. Not appropriate for shapes that could easily be represented with triangles
3. It is not filtered in the height dimension – this would require 3d textures.