5.4

Generating Procedural Clouds Using 3D Hardware

Kim Pallister, Intel

kim.pallister@intel.com

A great number of the games we play take place in outdoor environments, and most of those take place in environments that attempt to look much like the world around us. Because of this, the realistic rendering of terrain has become something of a holy grail for many game developers. Unfortunately, the modeling of the sky, and the clouds therein, does not get nearly as much attention, and is often added as an afterthought. Clouds usually end up being modeled with one or more layers of static scrolling textures. At first glance, these look OK, but they quickly give away their repetitive nature when looked at for an extended period.

In this gem, we'll set out to procedurally generate cloud textures that possess some of the properties that real clouds exhibit. In addition, because textures typically reside in graphics subsystem memory, we'll aim to generate the procedural clouds almost entirely using the graphics hardware. Finally, we'll address some ways to scale the technique's quality and performance requirements in order to accommodate a range of target machines.

Properties of Clouds

By taking note of the characteristics of real clouds, it's possible to come up with a prioritized feature list for the simulation. As with all real-time techniques, we may need to forsake some of these features in the interest of speed, but we'll worry about that later.

Here are a few things that we notice at first glance:

- Clouds are animated. They both move across the sky as the wind pushes them along, and change shape or "evolve" with time (as can be seen in any time-lapse footage of clouds). Additionally, the amount of cloud in the sky changes with time. This gives us a hint that there are three variables we'll need: the simulation rate, the wind speed, and something describing how overcast it is.
- As the clouds change shape over time, small details change frequently, while larger details take longer to change.
- Clouds exhibit a great deal of self-similarity in their structure.

- The level of cloud cover can also vary from a completely overcast sky to a clear blue sky with only a few isolated clouds (or none at all). As the level of cloud cover changes, so does the appearance of the clouds. In overcast conditions, clouds become gray and dark; in clear skies, clouds tend to be bright white against the blue backdrop of the clear sky.
- Clouds are three-dimensional entities. They are lit by the sun on one side and fall in shadow on the other. On a smaller scale, its much more complex than that, with clouds self shadowing and scattering light in all directions.
- Clouds are lit much differently at sunrise and sunset, as the light of the sun tends to illuminate them from an orthogonal angle, or even from underneath.
- Clouds tend to float at a common altitude, forming a cloud layer. Sometimes there are multiple layers. Since each of these layers is a constant altitude above the earths curved surface, we'll likely use some kind of curved geometry for the layers of clouds.

. . . and this is just some of what we see from the ground. In an application such as a flight simulator, where we could fly up and into the clouds, a whole new set of difficulties comes up. For this gem, we'll stick to the view from the ground.

Random Number Generation

As with almost any procedural texture, a good place to start is with the generation of some noise. *Noise* is a term used to describe a primitive that is random in nature. The noise primitive is a function that for a given input series (e.g., 1, 2, 3...), produces a *seemingly* random series of results (e.g., 0.52,—0.13,—0.38...). I say *seemingly* random because it must always produce the same result for a given input. This makes it *pseudo-random,* allowing us to recreate the same result for a given input seed value. Additionally, noise is often referred to as having a certain dimensionality (e.g., 2D noise, 3D noise, etc). This just refers to the number of inputs that are used to map into the random number function, much as one does a lookup into a multidimensional array. One of the dimensions is scaled by some factor (usually a prime number) large enough to keep repetitive patterns of results from showing up in an obvious fashion.

The generation of random numbers (or pseudo-random numbers) is a subject of much study. All approaches contain trade-offs between the complexity of the function and the quality of the result. A random number generator of high quality generates a good distribution of the random values, and does not repeat for a very long time.

Luckily, for the purposes of this gem, the results of a very simple random number generator will suffice. When we cover the specifics of the technique later, we'll be calling the random number generator multiple times per pixel with different input "seed" values. The cumulative result of this will mask any obvious repetition that occurs.

The pseudo-random number generator (PRNG) we'll start with is shown in Listing 5.4.1. It works by using the input parameter x as the variable in a polynomial,

where each term of the polynomial is multiplied by large prime number. By scaling each term of the polynomial differently, we get a long series before it repeats. The sign bit is then masked off, and the number is divided down to the 0—2 range, and is subtracted from 1, to give us a range of—1 to 1.

Listing 5.4.1 A simple pseudo-random number generator

```
float PRNG( int x)
{
    x = (x«13)*x;

    int Primel = 15731;
    int Prime2 = 789221;
    int Primes = 1376312589;


    return (1.0f - ((x * (x*x*Prime1 + Prime2) + Primes)
    & 7fffffff ) / 1073741824.0)
}
```

In cases where multiple octaves are being used, you can increase the "randomness" of the PRNG by constructing an array of prime numbers and using different ones for Primel and Primel, depending on the octave. For the purposes of this gem, using the single set of primes in Listing 5.4.1 is sufficient

Since we want to implement this in graphics hardware, and graphics hardware doesn't allow us to do all of the code shown in Listing 5.4.1, we'll create a lookup table of the function in a 512x512 texture map. Using a 512x512 texture map gives us -260 thousand entries before the function repeats. We'll use this texture as a random number generator by copying part of this texture into a target texture, using a software-generated random number to offset the texture coordinates. This is illustrated in Figure 5.4.1. The copying is done using the graphics hardware to render our random number series texture to a quad in the target texture.

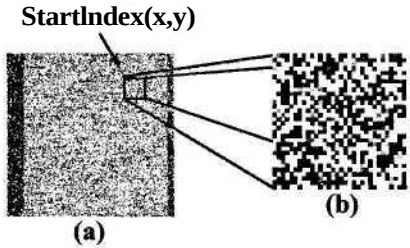

FIGURE 5.4.1 *A) The random number lookup table texture. B) The generated 32x32 noise texture.*

Band-Limited Noise

Ken Perlin pioneered the technique of using *band-limited* noise as a rendering primitive. Band-limited means that the noise is limited to a certain frequency range, which can be thought of as its maximum rate of change between samples. For our purposes, this means that we want to be able to produce a series of random values spaced out according to the frequency range, with intermediate points interpolating between these random samples. We do this simply by indexing into our random number lookup table and upsampling by the desired amount, using the graphics hardware's bilinear filtering to do the interpolation for us. Getting rid of the high frequency components will make for a procedural primitive with smoother, more natural-looking transitions. This is illustrated in Figure 5.4.2.

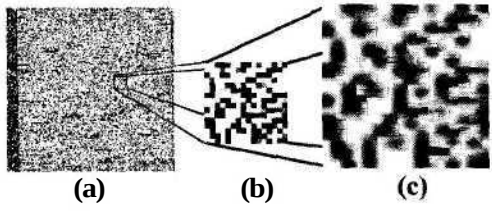

FIGURE 5.4.2 *A) Sampling the random number lookup table texture to create an array of noise. B, C) Using upsampling and filtering to create band-limited noise at the target resolution (the target frequency range).*

It's worth noting here than in some cases when using noise to create procedural content, bilinear interpolation is insufficient, and a higher order filtering is needed to produce adequate results. Luckily, this is one case where the lower quality bilinear filtering produces sufficient results.

An array of noise created in this fashion gives us a single noise primitive of a given frequency. We'll refer to this as an octave of noise, since we will later combine multiple octaves (multiples of the same frequency) together. First, though, we need to be able to animate our array of noise.

Animating an Octave of Noise

If we want to animate an octave of noise, we can look at time as a third dimension that we can use to index into our random number generator. The way we'll implement this using the graphics hardware is to periodically save off our noise texture, create a new one, and then interpolate between the two from one update to the next. The rate at which we update the texture determines the frequency in this third dimension.

The interpolation is illustrated in Figure 5.4.3. This is one case where using linear interpolation results in some artifacts, as the noise becomes more "focused" at the

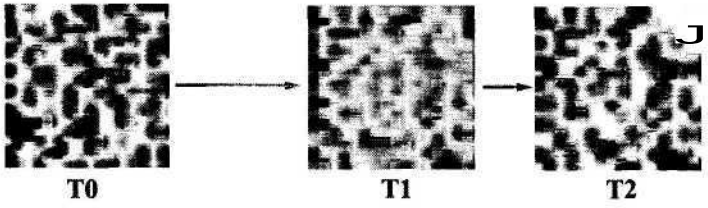

FIGURE 5.4.3. *Interpolating between noise updates to animate an octave of noise. Result = PreviousUpdate * (1-(T1-TO)/(T2-TO)) + NewUpdate * (Tl-TO)/(T2-TO). TO is the previous update time, T2 is the new update time, and Tl is the current time between updates.*

actual sample points. Again, this artifact is not apparent after the noise is incorporated into the final effect, as we'll see. Still, it's worth keeping in mind. If we were willing to sacrifice enough fill rate, we could maintain octaves for multiple points in time and use a higher order interpolation scheme to achieve a better result.

Summing Octaves into Turbulent Noise

A commonly used technique in procedural texture generation is to use a fractal sum, which is a summation of scaled harmonics of the basic noise function. This is referred to as fractional Brownian motion, or fBm, and is used in all types of procedural rendering techniques, such as fractal terrain, many types of procedural textures, and so forth. The fractal sum is shown in Equation 5.4.1. In it, *a* determines how we step across the frequency spectrum *(a* is most commonly 2, giving us f, 2f, 4f, ...), and *b* determines the amplitude of the contribution of each component in the series.

$$Noise(x) = \sum_{k=0}^{JV-1} \frac{Noise(a^k x)}{b^k} \qquad (5.4.1)$$

Fortunately, a value of 2 for *a* is both commonly used in the generation of procedural textures, *and* is well suited to our hardware implementation, as we can simply implement the octaves of noise as a series of texture maps whose sizes are a series of powers of 2.

Using a value of 2 for b is also both commonly used and makes our implementation easier. The composition of the octaves of noise can be done by performing multiple render passes using a simple blend, as shown in the pseudocode in Listing 5.4.2.

Listing 5.4.2 Noise octave composition

```
//note that ScaleFactor is needed to keep results in the
// 0-1 range, it changes {1 + 1/2 + 1/4 + ...} to {1/2 + 1/4 + ...}
```

```
float ScaleFactor  =0.5
float FractalSum = 0.0
for i = 1 to NumOctaves
{
     FractalSum = FractalSum*0.5 + ScaleFactor *
     NoiseOctave(NumOctaves-i)
}
```

Figure 5.4.4 illustrates the process of animating the different octaves between updates, and building the composited turbulent noise texture.

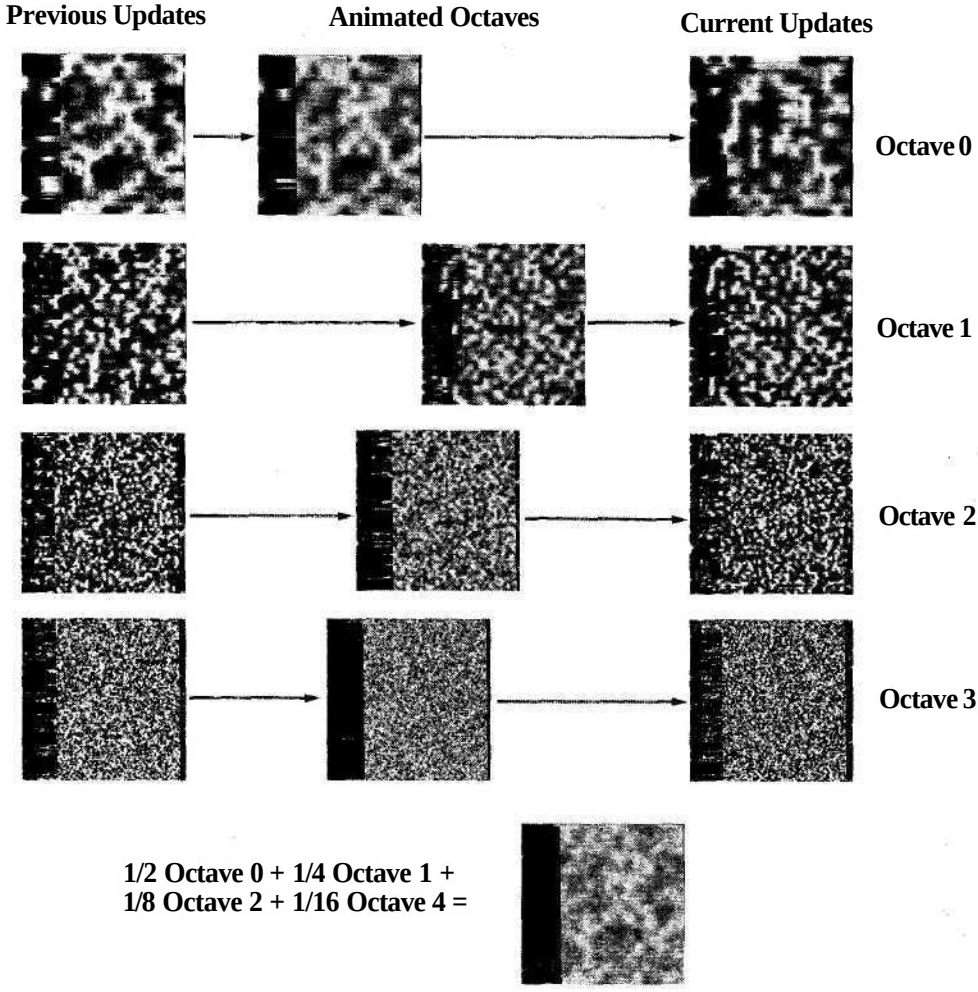

FIGURE **5.4.4** *Composition of several octaves of animated noise.*

Making Clouds from "Vapor"

Now that we have a texture map of animated, turbulent-looking noise, we need to do a few steps to turn it into clouds. Ideally, we'd like to use the noise as input to some sort of exponential function, giving us a sharp "condensation level" above which clouds would be visible, below which they would not. However, since this is an operation not available to us in a graphics chip, we'll have to use another method. There are several ways to tackle this.

The simplest way is to do a subtraction. Subtracting a fixed value from the texture clamps the texture at 0 where the noise was below that fixed value, isolating some clouds. This is illustrated in Figure 5-4.5.

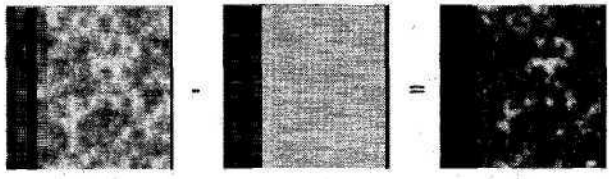

FIGURE 5.4.5 *Subtracting a fixed value to get isolated clouds.*

Unfortunately, we lose some of the dynamic range in the remaining clouds. We can compensate for this, however, by oversaturating and modulating with vertex colors and varying this. This is the method we use in the accompanying sample code.

Another way would be to have all the layers of texture that we have talked about so far have an alpha channel. This would let you do the subtraction and the clamping as in the previous test, but only on the alpha, and then do an alpha test to mask off the region without clouds without losing any dynamic range in die clouds themselves. A different problem occurs here, however. Since the alpha test is done before any filtering, rough jagged edges occur unless we use a really high-resolution destination texture.

Another way that may be better still would be to use some sort of dependent texture lookup, where one texture's color value affects the location from which a texel is fetched in a subsequent stage. One example of a texture- dependent lookup is the BumpEnvMap render state supported by some hardware under DirectX. In the future, as more hardware supports this type of operation, it may become feasible to encode an exponential function in a texture map and simply look up the result.

Mapping It to the Sky Geometry

Having prepared the cloud texture in Figure 5.4.5, we can map it to the sky geometry. Your choice of geometry depends on your application. The sample application uses something we've called a "skyplane," which is simply a rectangular grid of triangles, with the vertices pulled down over die radius of a sphere. You can imagine this as dropping a piece of cloth over the surface of a beach ball. It's not perfect, but makes mapping the texture easy. The skyplane is illustrated in Figure 5.4.6.

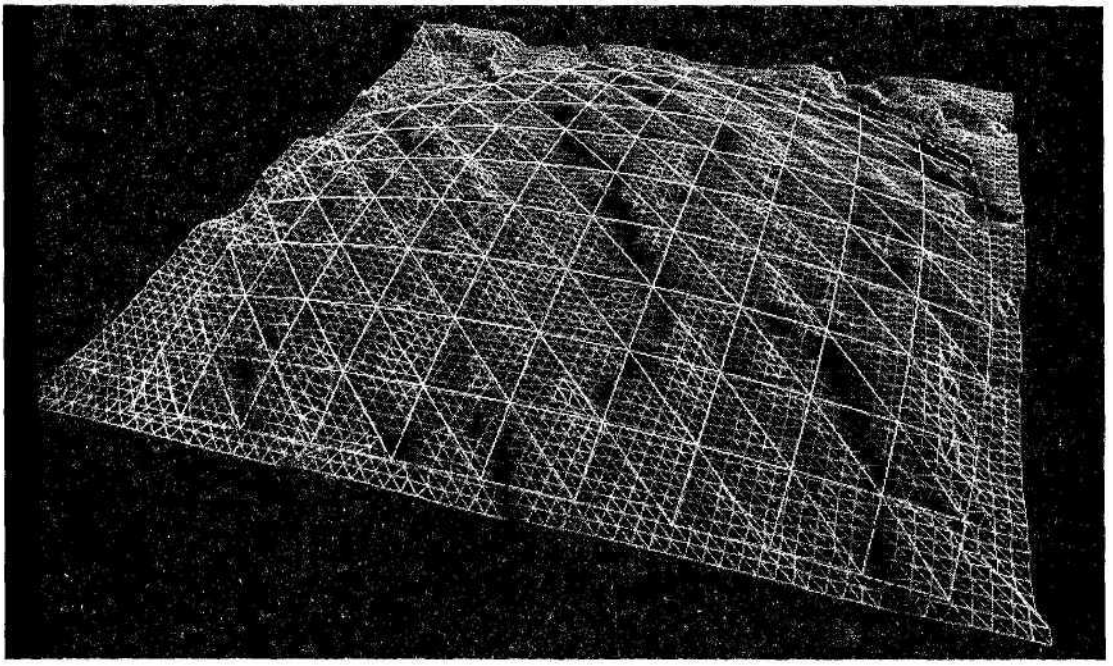

FIGURE 5.4.6 *Terrain geometry and the skyplane usedfor the cloud texture.*

Feature Creep

At this point, we can do many things to the clouds, depending on how much application time and graphics hardware fill-rate we want to consume. Some of these are implemented in the sample code; others are listed to provide you with ideas for your own implementation. Some of the things that can be added include:

- Noise-driven wind direction, cloud cover. Some dramatic results can be achieved by using other noise functions to modify some of our other variables over time, so that, for example, the wind can change direction and the level of cloud can increase or decrease over hours or days.
- Embossing the clouds to give them a 3D appearance. This will require an extra pass, and will require some modification of the vertex UV values. The vertices are given a second set of texture coordinates and are offset/scaled according to the direction of the sun. The clouds are darkened on the side away from the sun, and brightened on the side toward the sun.
- The clouds can cast shadows on the ground simply by using the end result texture with a subtractive blend. Your terrain will need to have another set of texture coordinates, or will have to use texture projection in order to do this.
- Modify lighting and/or lens flare intensity. Since we know how the texture is mapped to the geometry, and we know the angle of our sun, we can calculate the

exact texel or group of texels corresponding to where the sun is in our line of sight. We can use this to vary the level of lighting on our terrain, or to fade out a lens flare temporarily. Note that when modifying lighting, increased cloud cover should decrease the intensity of the directional light of your "sun" in the scene, but should *increase* the ambient level, since in reality the clouds will scatter the sunlight in all directions.

Hardware Limitations

Allowing the graphics hardware to do much of the grunt work in this technique allows us to achieve better performance than we'd be able to achieve if we had to manually lock and modify the texture at every frame. Unfortunately, it comes with some drawbacks as well. The drawbacks fall in four areas:

- **Render-to-texture support.** The technique spelled out in this gem uses this capability extensively. However, support for rendering to texture surfaces is far from ubiquitous. Detecting support for the capability has become easier with more recent API releases, but can still be tricky. In cases where rendering to texture surfaces is not supported, a workaround of copying from frame buffer to texture can be used, but performance may suffer.
- **Limited precision.** Current hardware stores texture and render target values as integers, usually in 32 or 16 bits per pixel. Because we are essentially working in "monochrome," generating a texture in shades of gray, we are limited to 8 bits per pixel, and in the worst case, 4! The latter gives noticeable artifacts. Note that this means that the high-frequency octaves are only contributing a few bits of color data to the final result due to this problem.
- **Limited dynamic range.** Because the integer values represent data in the 0-1 range, we are forced to scale and offset values to fit in this range. Since our noise function originally returned values in the -1 to 1 range, we had to compress these to fit them into our available range. This requires extra work, and also amplifies the artifacts from the limited precision
- **Limited hardware instruction set and capabilities.** The limited instruction set of the graphics hardware limits us in terms of what we are able to do. It would have been nice to feed the turbulent noise texture into a power function, perhaps as the exponent, to generate some interesting results, but we are instead limited to simple arithmetic operations. As well, we make much use of render-to-texture, which isn't available on all hardware. In the future, as hardware capabilities increase, this will be less of an issue.

Making It Scale

If we want to use this technique in a commercial game project, and are targeting a platform such as the PC, where performance can vary from one machine to the next, then we need to worry about scalability of the technique. Even if targeting a fixed

platform, the game may have a varying load, and we may want to scale back the quality and performance requirements of the technique in heavy load situations.

There are several ways in which the technique can be scaled:

- **Texture resolution.** The resolution of the textures used at the various intermediary stages could be reduced, saving memory and fill-rate consumption. Note that scaling color depth is most likely not an option here, since the artifacts show up pretty quickly when using 16bpp textures.
- **Texture update rate.** Not every texture need be updated per frame. Particularly if the simulation rate is slow, the interpolating textures do not need to be updated at every frame as we did in the sample code.
- **Number of octaves used.** The sample code uses four octaves of noise, but using three might provide an acceptable result. Similarly, five octaves might give slightly better quality on systems that have the horsepower.

Conclusion

Figure 5.4.7 displays the final result of the procedural cloud example program.

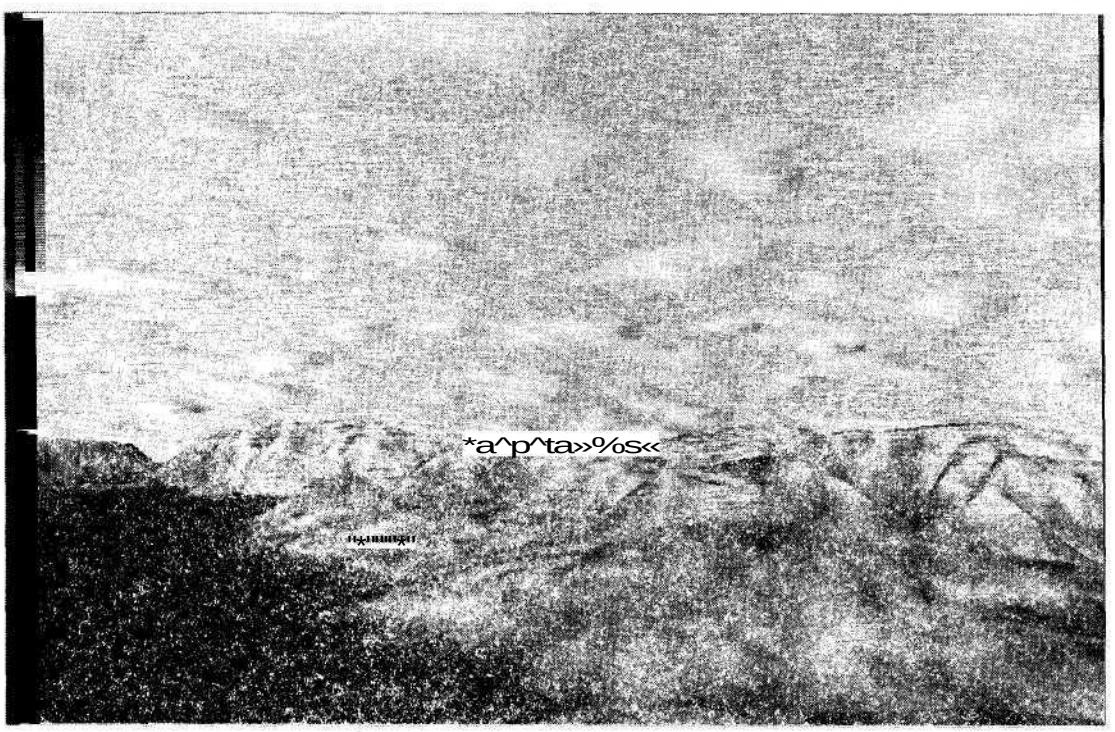

FIGURE **5.4.7** *Final result of procedural cloud example program.*

We hope this technique has provided you with some insight into procedural texture generation, using clouds as an example, and how to get graphics hardware to do much of the per-pixel work needed in generating dynamic procedural textures.

As hardware grows more capable, so will the techniques in which we can generate these textures in real time. We encourage you to try your own experiments and share them with the development community.

References

[Ebert et al, 98] Ebert, David S., et al, *Texturing and Modeling: A Procedural Approach*, AP Professional Inc., 1999. ISBN: 0-12-228730-4.

[Elias99] Elias, Hugo, "Cloud Cover," available online at http://freespace.virgin.net/hugo.elias/models/m_clouds.htm, offers a software-only technique similar to this one.

[Pallister99] Pallister, Kim, "Rendering to Texture Surfaces Using DirectX 7", available online at www.gamasutra.com/features/19991112/pallister_01.htm