Technical Report

Snow Accumulation

Abstract

Games these days are requiring more and more art time. Schedules are blowing out, an game developers are struggling to meet tight schedules. Any method that can produce good looking results, and save art time can be a schedule win for a game title. This sample shows a technique to procedurally render snow accumulation. It also uses SM3.0 to enhance the effect by offsetting vertices based on their snow accumulation value.

This sample is based on the paper entitled "Real-time rendering of accumulated snow", by Per Ohlsson and Stefan Seipel from Uppsala University.

Bryan Dudash
bdudash@nvidia.com

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

2/9/2005

Table of Contents

How Does It Work?

At it's heart, this method calculates the snow coverage of the scene , and then use this information to procedurally modify the mesh and lighting to reflect snow accumulation. Most important to this technique is the orthographic rendering of the snow occluders from the point of the sky. During this rendering we store the distance to the closest point in the scene(similar to a shadow map rendering). Then during the normal render, we transform the point into the ortho rendering. After this, we read from the previous ortho render target and compare the values. If the distances are around the same, then the point is exposed to snow, if it is further away, then it is occluded. After a point is determined to be exposed to snow, we project the normal onto the up vector to determine a level of accumulation.

The basic equation for level of accumulation is as follows:

$$F = F_e + F_a$$

$$F_e = \frac{\left(\sum_{\#samples} texturedepth - actualdepth >= 0 \right)}{\#samples}$$

$$F_a = \overline{N} \bullet (0,1,0)$$

Equation 1: Exposure equation.

Once we have the full calculated exposure, we render the scene, offsetting vertices based on how much snow the would receive, and coloring pixels based on how much snow they would receive. Both the normal, and the Fa value are jittered using Perlin noise.

Exposure Coefficient (Fe)

The Exposure coefficient is calculated by determining if the point being considered is the closest to the snow plane. To do this, we must first calculate which point is closest to the sky. Specifically, we calculate the distance to the closest point and store it in a render target. This is done by first rendering all objects that may occlude the snow using an orthographic projection facing in the direction of snow fall(usually straight down). Then when rendering the real scene, we again transform the point into the orthographic space, and can determine the distance from the sky. This distance value can be interpolated across the pixels and the comparison is done to determine per pixel coverage.

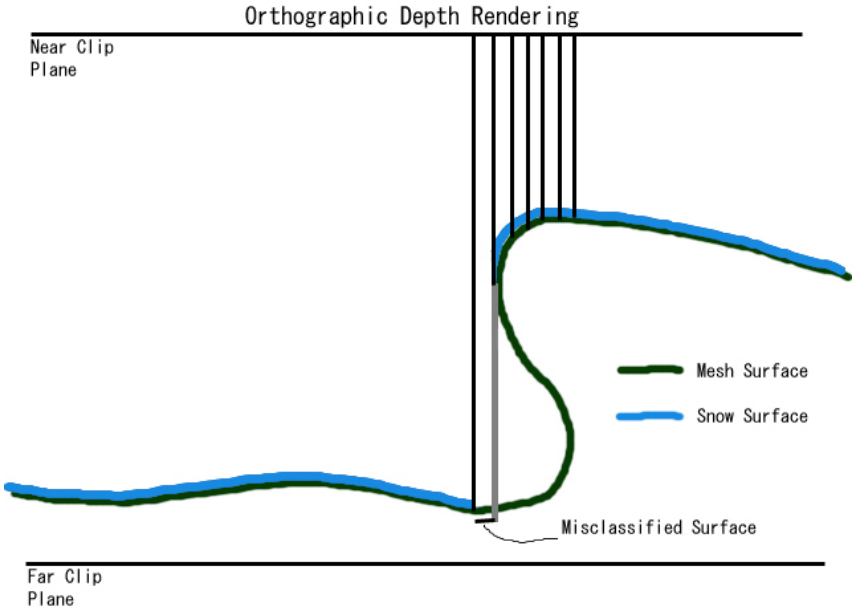

Figure 1: Orthographic Depth Rendering Diagram. Green is the original mesh, and he blue area is the part of the mesh that is calculated to be accumulating snow.

Incline Coefficient (Fa)

The incline coefficient represents the fact that steep slopes will accumulate less snow than flat ground. To accomplish this we simply dot the normal of the surface with the up normal. In actual implementation, this just translates directly to the Y component of the normal, so the operation is just a copy operation. In real life, snow however, follows a much more complicated rule for accumulation, and on slopes, there is an effect called "clustering"($$$4 this isn't the name what is it?) We can accomplish a good approximation of this effect by applying a little perlin noise to the Fa value.

Implementation Details

The above sections covered general discussions of the technique. This section will cover some details on the sample implementation, including potential artifacts, and how Shader Model 3.0 increases the effect quality.

Depth precision

This technique and shadow mapping are very similar in that they use a render target from a different point of view to calculate information about a scene. And as such, they share some of the same artifacts. However, one artifact that shadow mapping has is depth precision. This is error induced because you are using a float render target (Z buffer) that doesn't have enough precision to accurately represent the number ranges. This snow technique has the potential to sufer from this issue, but the solution is the same as shadow mapping. Offset your sample to avoid value that may be too close. In general, if two the sampled depth value and the calculated depth value are close to each other, then the surface is exposed, so you can offset by a substantial amount and still have the scene look correct..

Aliasing artifacts

Edge Aliasing

As you can see in figure 1(previous section), there is some aliasing that occurs because the depth of objects in the scene s quantized into a render target. Depending on the resolution of your render target, these effects may be more pronounced. Figure 2 below shows a low resolution map and single texture sampling, notice the harsh pixilated edges. However, the multisampling adds texture accesses to the pixel shader. This is the same issue that shadow maps run into when the light is far away from the surface and camera positions. Without any type of multisampling, the aliasing become much more visible.

Figure 2: aliasing from using no multi-sample and a very small render target

The aliasing issue has two tradeoffs that you can make to improve performance or quality.

1. Increase/Reduce resolution of orthographic depth map.

You can change the resolution of your depth map to either increase performance or increase quality. Higher resolution is better quality, but the texture won't be as cache friendly. If you aren't texture bandwidth bound, ten feel free to increase this for no performance penalty.

2. Add extra texture samples

You can also add extra texture samples and blend the values together. The more samples you add, the smoother the edges will look. You can also vary the sample distance to control the level of blurriness in the resulting edges. However, more samples increase the texture bandwidth of the shader, and add instructions. If you are shader bound, then adding samples with hurt performance, and removing some will help.

Depth Aliasing

Even though depth precision is less of an issue for this technique, depth aliasing is still an issue for very steep slopes. Similar to the regular edge aliasing issue, depth aliasing can be fixed by increasing the resolution of the depth map, and/or increasing the number of samples from the depth map.

SM3.0 and VTF

With the addition of SM3.0, the snow accumulation becomes more compelling. With SM3.0 we gain the ability to read the exposure depth map in the vertex shader and then offset vertices based on their exposure value (in the direction of the vertex normal). In this sample, I load 4 samples from the depth texture. This is necessary because my mesh is not that heavily tessellated. For a more tessellated mesh, it is possible that a single sample may be sufficient.

Also, there is an issue to be aware of when using VTF and offsetting the vertices in the shader. This is the fact that the normals will be incorrect. Depending on how much you offset the vertex, the incorrect normal may be obvious to the viewer. This sample does not address this issue, however, a potential solution would be to generate a new normal in the vertex shader and use that in place of the passed in vertex normal. You could generate the normal, by taking a guess at the snow accum of neighboring points and then crossing the vectors to those guessed points.

Perlin Noise

Both the per-pixel normal and the result of the normal dot up vector are jittered using Perlin noise. This is accomplished using a 64x64x64 3D noise texture. Samples from this noise texture are pulled using procedural texture coordinates. Multiple samples are drawn using different frequencies, and amplitudes (according to the standard method of 2x frequency and ½ amplitude each pass) for a number of passes. In writing this demo, we found that 3 passes was enough to give a good effect of random normals.

Pseudo code follows:

```
int freq = 1;
int amplitude = 1;
int noise = 0;
for o in octaves
      noise += amplitude*texture(freq*uv);
      amplitude /= 2;
      freq *= 2;
endfor
```

This noise value will generally be between [0..1].. So we stretch it into the range of [-1..1] and the it can be used to modify things to give the effect of a extensible random system.