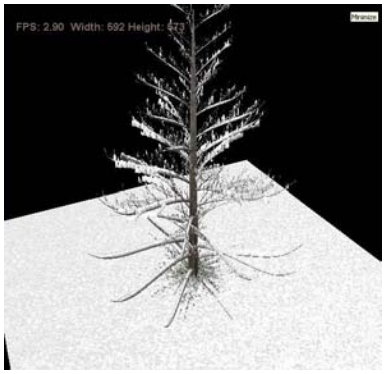
Real-time Rendering of Accumulated Snow

Per Ohlsson
Uppsala University

Stefan Seipel
Uppsala University



Abstract

This paper presents a method of computing snow accumulation as a per pixel effect while rendering the scene. The method is similar to the shadow mapping method for shadow calculations. A depth buffer is used to find out how much snow a particular surface should receive. The amount of snow is then modified depending on the slope of the surface. To render the snow in a convincing way 3D noise is utilized for the lighting of the snow surface.

Keywords: snow, computer graphics

1. Introduction

Snow is a common phenomenon in nature. It has the ability to completely transform the mood of a scene, turning a rocky landscape into a gentle sear of white tranquillity. But even though snow is a common occurrence in many places of the world, snow accumulation in real-time seems to be an area that has still to be explored. Few attempts have been made, and the methods demand lots of work do be done on the scene modelling before they can be used.
Usually when snow occurs in interactive simulations it is either modelled by an artist, or just plain textures drawn upon the original scene.
The ability to automate the snow accumulation process in real-time would give us the ability to add snow to any scene, and by doing this produce beautiful settings for any kind of interactive environment. This would enable artists to just create non snow scenes and then let the algorithm create a snow cover, effectively creating two scenes. It would greatly decrease the amount of work needed to model a snow covered scene.

2. Related Work

The subject of offline snow rendering has been looked into quiet throughout.
Paul Fearing [2000] presented a method to create beautiful snow scenes in 'Computer Modelling Of Fallen Snow'. Unfortunately each frame took very long time to render making the method not viable for adaptation to any kind of real-time situation. It worked by first tracing snow paths from the ground upwards towards the sky, and by that accumulating the snow. In a second stage the stability of the snow was calculated, moving snow from unstable areas to stable,
In the paper "Animating Sand, Mud and Snow" Summer et al. [1998] describe a method for handling deformations of surfaces due to external pressures. In their algorithm the surface is divided into voxels containing different height values indicating the height of the current surface. This grid is used to calculate the deformation of the surface when interacting with objects.
Nishita et al. [1997] use metabals and volume rendering for their snow accumulation and rendering, taking into account the lights path and scattering through the snow. All this makes it less suitable for real-time modification.

Snow rendering in real-time has received less attention.
Haglund et al. [2002] propose a method where each surface is covered with a matrix containing the snow height at that position. The snow accumulation is handled by dropping snow flakes, represented by particles, from the sky. In the place where a snow flake hits the ground the height value gets increased. Triangulations describing the snow are then created from the matrices containing the height values.
This method demands lots of work to be made by the modeller, by creating the matrices by hand, before a scene can be used.

Although not about snow Hsu and Wong [1995] presented a method for dust accumulation in their "Visual Simulation of Dust Accumulation". In this paper they use an exposure function to tell whether the surface should be covered in dust or not. The exposure function is calculated by sampling the surrounding area with rays to find any occluders.

A variant of this method turned out to be a viable way to handle the process of snow accumulation. The method we will present uses [Hsu and Wong 1995] as a basic foundation.

3. Snow Accumulation

The process of rendering accumulated snow can be split into two parts. The first part is to decide what regions should receive snow. The second is to actually render the snow in a convincing way at the places decided by the first part of the algorithm.
To accomplish the first step a function called the Snow Accumulation Prediction Function is introduced. This function should take a point in space and calculate how much snow that point has received. Factors that should be taken into account are surface inclination and exposure to the sky.

3.1. Snow Accumulation Prediction Function

Due to gravity a surface facing upwards should receive more snow than a vertical surface. However, even a horizontal surface does not accumulate any snow if it is occluded from the perspective of the sky. It seems like the task of calculating the prediction function can also be separated into two mutually independent parts. One part should calculate the snow contribution due to the inclination of the surface and another part should calculate the effect of occlusion.
Let us formulate a Prediction function in terms of these two parts:

$$f_p(p) = f_e(p) \cdot f_{inc}(p)$$

where p is the point of interest
f_p is the prediction function
f_e is the exposure component
f_{inc} is the component giving the snow contribution due to inclination

The exposure part (f_e) should vary between 0 and 1 indicating the amount of occlusion that would prevent snow from falling on the surface. This value should vary gracefully to achieve a smooth transition from an area with snow to an occluded area. Hsu and Wong [1995] calculates this value by sampling the surrounding area with rays uniformly distributed over the upper hemisphere in order to search for potential occluders. Unfortunately this is not a viable way of doing it if we want to calculate this in real-time without lots of pre-processing for all surfaces. An alternative way of implementing this function based on the shadow mapping technique will be examined later in this paper.

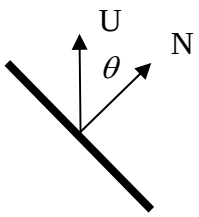


Figure 1: Inclined surface

The f_{inc} should work in a similar way to the Dust Amount Prediction Function in [Hsu and Wong 1995]. A surface facing towards the snow direction should receive more snow than a surface facing away from it. The amount should depend on the angle, with a rather step falloff when the angle between the normal and a vector pointing upwards grows (Figure 1). Surfaces facing away from the sky should not receive any snow. This implies that the function should decrease from 1, when the angle is 0, to 0 when the angle is 90 degrees. If the angle is greater than 90 f_{inc} should be 0.
This gives a function like this:

$$f_{inc} = \begin{cases} \cos\theta + n, & 0 \le \theta \le 90 \\ 0, & \theta \ge 90 \end{cases}$$

where θ is the angle between vector N and U
n is a small positive noise value

A small noise value [Perlin 1985], typically between 0 and 0.4, is used to get a more natural look, and to account for the phenomena called flake dusting [Fearing 2000], where snow dust clings to a steep uneven surface.

3.2. Snow Colour Function

After the snow accumulation of a certain surface is determined it's time to calculate the correct shading of the snow. The light model that is used is the normal Phong illumination model.
To get the typical look of snow a noise function is used to distort the normal of the surface. This way a realistic approximation of the look of a snow cover is achieved. To get the glittering effect of snow the normal is distorted slightly more for the specular part of the light calculation than for the diffuse part. The derivative of the exposure function is also used to transform the normal to get an illusion of actual snow depth around any occlusion boundaries that may exist in the scene. In the implementation the normals are calculated in this way:

$$N_\alpha = N + \alpha n - dE$$

where N is the original normal
α is a scalar value indicating how much distortion should be applied
n is a normalized vector containing three noise values
dE is a vector containing a scaled value of the exposure derivatives in respective direction of screen space

A distorted value α of 0.4 was used in the images presented in this paper. This was chosen through testing different values and deciding on what looked best.
The renormalized resulting normal is then used in the diffuse part of the lighting equation. For the specular part another distortion term was added with the α value 0.8 to get a more glittering effect on the snow.
To calculate dE the derivative of the exposure function in both x and y direction of the screen is needed. In this implementation the derivative operations of Cg is used. The derivative with respect to x is placed as the x-component and the derivative with respect to y in the z-component, assuming that the y-axis points upwards. The resulting vector is then scaled to obtain a suitable impression of decreasing height when the exposure function decreases.
When calculating the snow colour as above a white colour should be used in the Phong equation. The blue part of the colour should

be slightly higher than the rest to give the snow a more glittering effect.

3.3. Full Snow Equation

The full equation to calculate the colour of accumulated snow then becomes

$$C = f_p \cdot C_s + (1 - f_p) \cdot C_n$$

where C_s is the snow colour calculated with the distorted normal, as explained above
C_n is the surface colour without snow

To obtain an impression of thickness to the snow each vertex should be displaced depending on the f_p value. The amount to displace the vertices depends on the scene and needs to be tested to achieve the best result. The displacement introduces the restraint on the geometry that it should be closed to give the impression of actual height of the snow.

4. Implementation

The above algorithm where implemented in a pair of vertex/pixel shaders. Most of the equations can be implemented in a straight forward way when implementing the standard shading equations, as described by [Everitt et al. 2002].

4.1. Implementing the exposure function

The exposure function did not lend itself to the same easy implementation as the other components of the equation. To calculate the exposure of a surface, knowledge about the surrounding world is needed. This does not suit well with the limitations inherent when working with the pixel shader on a modern GPU. To solve this, a solution is taken from the way shadows are implemented with the aid of a depth buffer. This image space algorithm is very suitable for what we are trying to do here.
As a pre-processing stage the whole scene is rendered with respect to the sky, using a parallel projection. The depth buffer is then saved for later usage.

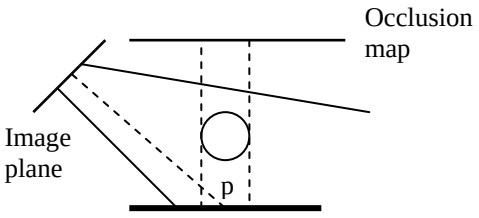


Figure 2: Point p is not the closest point to the sky, it will not be covered b snow

When rendering the scene each fragment is projected into the sky view frustum, as described in [Everitt et al. 2002], and compared with the stored depth value in the occlusion map. If the fragment is further away from the sky than the value indicated in the occlusion map, as in Figure 2, no snow should be drawn, otherwise snow should be drawn. This method works but is still unsatisfactory because it produces very sharp and sudden occlusion boundaries, as can be seen in the picture below.

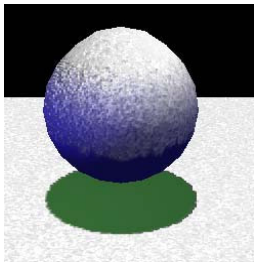


Figure 3: One sampling from the occlusion map produces very sharp transitions.

To address this issue the depth map is sampled a number of times, with the first sample on the original fragment position and the others in a circle around it in the occlusion map. The number of snow covered fragment is then divided with the total number of samples to get a fractional value for the occlusion. This creates a better result, but it still leaves us with plateaus in the snow and possible artefacts on objects as seen in Figure 4. The length of the offset used for the different samples determines how big the occlusion boundary becomes. A larger offset produce a smoother occlusion boundary. The larger the occlusion boundary should be the more samples need to be done to get enough overlapping in the samplings. How big the offset should be is entirely dependent on the size of the scene the occlusion map covers, and the desired area of the occlusion boundary.
To sample the depth buffer in the area surrounding the current fragment the change in z direction due to offsets in the depth map must be known. To get this value the derivative functions in cg are used to calculate the proper position of neighbouring fragment.
All of the above mentioned offsets are performed in the projected space of the occlusion map.

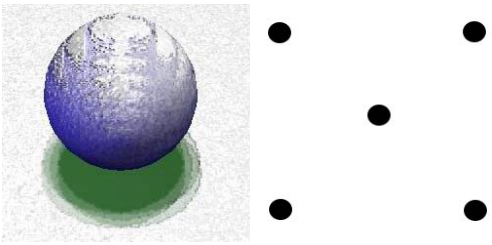


Figure 4: (a) 5 Samplings produces better transition but introduces artefacts. (b) Sampling pattern

As can be seen this produces quite a bit of artefacts in the exposure value. To fix this the final result is combined with a noise value in the range of [0, 0.5], if it is bigger than 0, to produce a more natural looking boundary, and to conceal surface artefacts due to discontinuities in the surface.

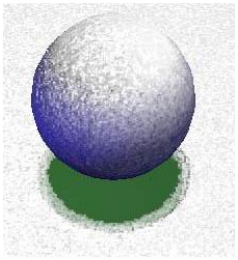


Figure 5: Same scene as figure 3 but with noise included.

The adding of the noise value effectively removes all of the artefacts shown in figure 3. It also makes the occlusion boundary look much more natural.

4.1.1. Vertex Displacement Method 1

One problem with this method is that the vertex program can't read from the occlusion texture. This means that we have to provide the snow values to the vertex program in some other way. One way is to introduce another pre-processing stage where the exposure map is first read back from the GPU. The position of each vertex in the exposure map must then be calculated. This is done by scaling the x and z coordinate according to the world size so that they can be used to index into the occlusion map, ignoring the y coordinate. This value should then be streamed to the vertex shader in the same way as position and normals.

4.1.2. Vertex Displacement Method 2

Another way of solving the problem with the vertex shaders inability to access the occlusion map is to split the rendering into 2 passes. The first pass should draw the scene without snow and the second pass with the snow. When displacing the vertices for the snow pass no consideration should be taken to if the vertex is snow covered or not, only the inclination should be considered. The exposure value should instead be assigned as an alpha value to each fragment in the fragment shader. The second pass should then be blended onto the first, using alpha as blending factor for the second pass and one minus alpha for the first. This method is easier to implement than the first, and places less demands on the format of the scene. It does unfortunately create some artefacts where the snow floats above the scene that can be annoying in certain scenes where the user can view the scene from any perspective. Figure 6 show this artefact.

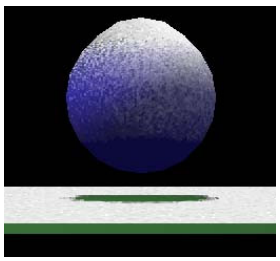


Figure 6: Floating snow artefact seen on the ground. The snow layer floating some distance above the ground

The artefact is extremely visible in the above picture because of the fact that the ground plane isn't a closed geometry. But even when working with closed geometries artefacts as the above can still happen, especially with very sharp corners.
Method one does completely eliminate this artefact but introduces the need for more pre-processing and the need to store the values which might be a problem when using already existing scenes and scene formats.

4.2. Implementing noise

All of the noise values used in this paper can be implemented in the fragment shader by using a 3D texture filled with random 3 component values in the range [0, 1]. This texture is then used to draw random numbers that will always be the same for the same input.
To get the correct noise appearance the random texture can be sampled a number of times, called octaves [Perlin 1985], each octave having double frequency of the last, and half the amplitude. The sum of these octaves can then be used for a noise value.
The noise used in the pictures shown in this paper where created by reading 3 octaves from the 3D texture. The single noise value used in f_{inc} was calculated by adding the 3 components together and the dividing by 3 to bring the range back to [0, 1]. The noise vector read from the 3D texture was then expanded to the range [-1, 1] and normalized to create a suitable vector for distortion of the normals used in the light calculation.

4.3. Performance Issues

The implementation of the algorithm includes a lot of normalizations that affects the performance quiet thoroughly. To avoid this a cube map is used for normalization purposes.
In a cube map only the direction of a vector is used for lookup, not the magnitude. This can be used for normalizations by storing a normalized version of the direction vector in each component of the cube map. The components of the normalized vectors must first be transformed to the range [0, 1] before they are stored in the cube map. This is done by multiplying the vector by 0.5, taking the elements into the range [-0.5, 0.5]. By then adding 0.5 the range [0, 1] is achieved. The components are then encoded in the RGB components of the cube map texture. To use the cube map for normalization a texture lookup should be done as usual, with a vector as texture coordinates. The resulting vector must then be unpacked from [0, 1], the range in which colours are normally stored, to [-1, 1] by multiplying by 2 and subtracting 1.
By using linear interpolation on the texture lookup the cube map turns into a smooth normalization function without needing very high resolution. A cube map of 64x64 or 128x128 is usually enough.

5. Performance

The implementation is tested on a machine with a Geforce FX 5600 Ultra, using CG to compile the shaders to the NV_vertex_program2 and NV_fragment_program.
The performance is directly dependent on the resolution and the amount of the screen covered with potential snow covered surfaces.
In the demo presenting two instances of the Stanford bunny, Figure 7, each consisting of 16000 triangles an average frame rate of 13 frames per second were achieved. This was with a screen resolution of 600 x 600. When increasing the screen size to 900 x 900 the frame rate dropped 11 frames per second.

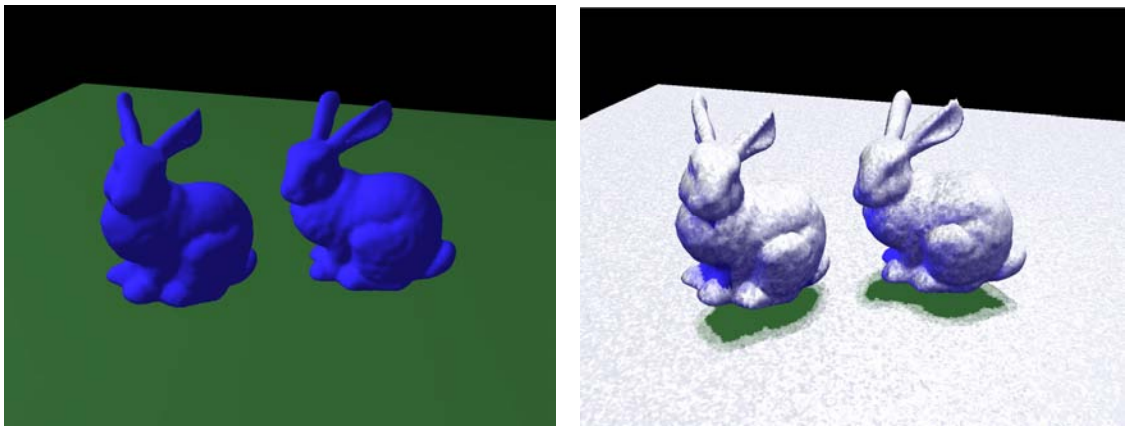


Figure 7: Stanford bunny without and with snow

6. Discussion

One problem with the snow accumulated in the way proposed in this paper is that it does not take into account the area of the region being covered in snow. This means that small parts, as the small twigs in the tree pictures in Figure 8 and Figure 9, can be loaded with an unnatural amount of snow. There isn't really much to do about this, because it would demand additional knowledge about the scene that does not exist at the fragment level.
The tessellation of the snow cover will be that of the underlying geometry. This leaves the scene modeller free do add polygons in places where a smoother curve is desired.
There are several possible improvements that could make the proposed method faster. One is to save the four surrounding sample values of the occlusion map in the RGBA parts of a new texture to reduce the numbers of samples needed to two, non dependent lookups. This would be done as a pre-processing step when the occlusion map is created.
Something that speaks for this method in the future is that it is independent of the number of objects in the scene, enabling the usage of complex and cluttered scenes without any extra work. And as the speed of fragment processing increases with the advantage of GPU, speed would cease to be an issue.
Another problem with the method is if you want to have dynamic objects acquire snow as they move out in the scene. That would mean that when they move in to shelter the snow accumulated would be lost on them.

7. Conclusion

In this paper, a new method for calculating snow on a fragment level was presented. The method uses a depth map to find out what parts of the scene should be covered with snow. The snow is then calculated per fragment in the scene without needing any more pre-processing of the scene data. Although the method isn't very fast as of today, increases in the computational power of today's graphic hardware should make this method a good candidate for snow accumulation and rendering in the future. The advantage of not needing to modify or store any extra data about the scene except for the occlusion map means that it should be easy to implement and combine with other existing techniques for rendering.

8. Future Work

Something that would be an interesting extension to the above presented method is the usage of a control map where things as footsteps could be drawn. This could be done by using another texture map stretched in the same way as the occlusion map. Each fragment would then multiply its exposure value with the value in the control map. Footsteps could then be drawn in the control map as half occluded fragments.
Another area that could prove interesting is the possibility to take into account the influence of wind on the falling snow. Perhaps this could be modelled by tilting the projection when calculating the occlusion map.

References

Everitt, C., Rege, A., and Cebenoyan, C., 2002. Hardware shadow mapping, ACM SIGGRAPH 2002 Tutorial Course #31: Interactive Geometric Computations using graphics hardware, ACM, F38-F51

Fearing, Paul 2000. Computer Modelling Of Fallen Snow. Proceedings of the 27th annual conference on Computer graphics and interactive techniques, 37-46

Haglund, H., Anderson, M., and Hast, A., 2002. Snow Accumulation in Real-Time, Proceedings of SIGRAD 2002, 11-15

Hsu, S.C, and Wong, T.T. 1995. Visual Simulation of Dust Accumulation, IEEE Computer Graphics and Applications 15, 1, 18-22

Nishita T., Iwasaki, H., Dobashi, Y., and Nakamae, F. 1997. A Modeling and Rendering Method for Snow by Using Metaballs. Computer Graphics Forum, Vol 16, No. 3, C357

Perlin, Ken, 1985. An Image Synthesizer, Computer Graphics (Proceedings of ACM SIGGRAPH 85), 19, 3, 287-296

Summers, Robert W., O'Brien, James F., and Hodgins, Jessica K. 1998. Animating Sand, Mud, and Snow. The Proceedings of Graphics Interface'98, 125-132

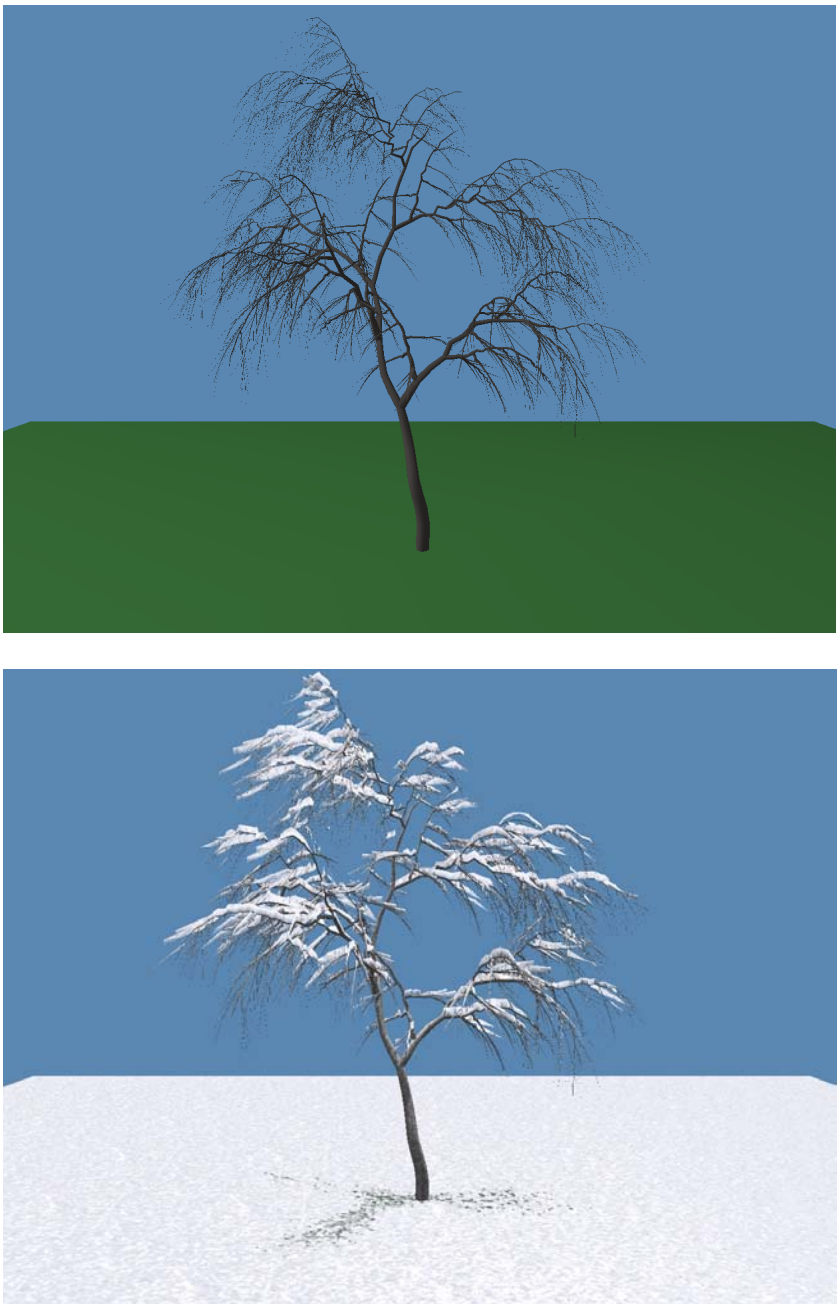


Figure 9: The algorithm tested on a more complex scene containing lots of small structures. As can bee seen in the picture the algorithm does not take into account the area of the surface being displaced. This results in the small twigs getting more snow than would look natural.

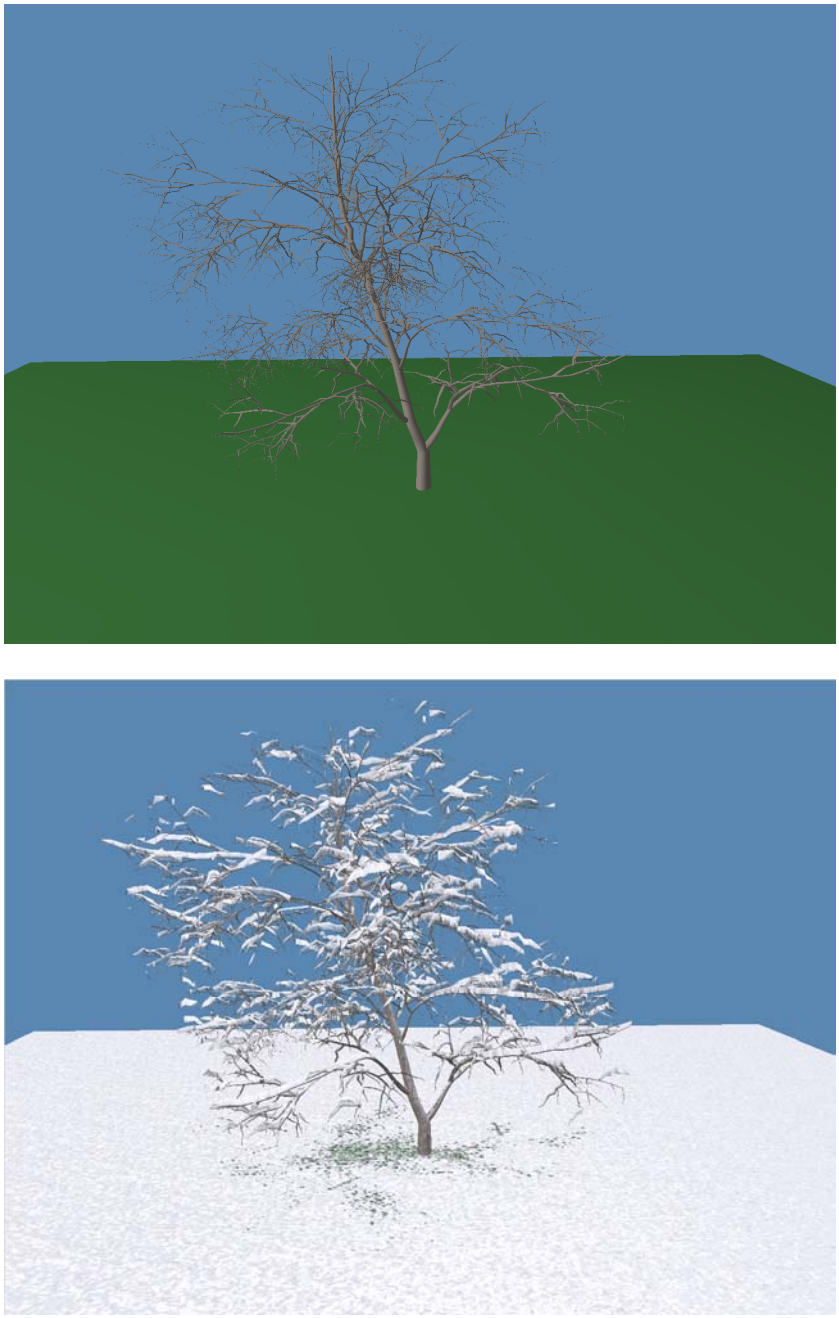


Figure 10: Another tree used for testing of the snow algorithm. As can be seen the area beneath the tree is partially covered depending on amount of branches above it.