IT 10 003

Examensarbete 30 hp
Februari 2010

An Illumination Model for Realistic Rendering of Snow Surfaces

Fei Liu

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

An Illumination Model for Realistic Rendering of Snow Surfaces

Fei Liu

In this thesis work, several prominent visual effects of a snow surface under a strong light source, for instance, the sun, have been identified and modeled, which includes glittering, micro shadows and regional translucency due to subsurface light scattering. The glittering effect was implemented using blooming technique which makes candidate pixels' colours bleed into their neighbours. Normal mapping and horizon map were experimented in attempt to obtain small surface details and micro shadows. In the end, a curvature-based local illumination model was devised instead to soften such a requirement as modeling micro shadows. From this local model, we gained regional light enhancement or suppression as well as a regional translucency effect. The light suppression effect can be considered as a kind of local shadowing effect. A surface point's mean curvature was employed in this model. All these effects were modeled in such a way that the rendering of them is capable of being carried out on an average machine at an interactive frame rate.

Handledare: Stefan Seipel
Ämnesgranskare: Stefan Seipel
Examinator: Anders Jansson
IT 10 003
Tryckt av: Reprocentralen ITC

Contents

1. Introduction 1

2. Related work 3

3. Implementation procedures 5

3.1 Experiment platform establishment. 5

3.1.1 Essentials of the class Vertex. 5

3.1.2 The surface mesh. 6

3.2 The illumination model 7

3.2.1 The glittering effect. 7

3.2.2 Fine surface details. 9

3.2.3 Curvature-based local illumination model. 15

4. Results 21

4.1 The in-scattering effect 21

4.2 The subsurface scattering effect. 22

4.3 Putting it all together. 23

5. Conclusions 26

6. References 27

1. Introduction

The nature possesses a lot of wonderful phenomena. When winter comes, the first one of such phenomena occurs to most people may be snow. This is very natural because during the winter months almost half of the landmass is covered by snow in the Northern Hemisphere [Fearing 2000]. Computer graphics, one task of which is to imitate the real world, inevitably needs to deal with snowy scenes.

In terms of modeling fallen snow as well as interaction between it and objects, quite a little work has been done with impressive results. However, modeling the physical shape is merely one aspect of rendering fallen snow. Given the fact that snow is formed by crystalline water ice, these crystals react pretty actively with a light source and thus take on several distinct visual effects. Without realizing these effects, computer generated fallen snow appears as hard white plastic or simply a white texture. To my knowledge, less attention, though, has been given to the illumination aspect of fallen snow. Therefore, in this thesis work we emphasized on realizing those visual effects we think contributing the most to tell snow apart from other similar matter.

During the early study phase, we have worked out a summary of the prospective illumination model. This summary contains three parts: the first two parts specify both the incoming and the outgoing light components and their own characteristics; the last part outlines all the possible effects we could think of caused by the interaction between the light and a snow surface.

1. Incoming light components on a bright sunny day
    a) Ambient blue light component from the sky gathered in the hemisphere over a surface point
    b) High intensity white-yellow component from the directional sunlight, most likely diffusely reflected
    c) In-scattering light from surrounding areas, if the region in question is concave
    d) Subsurface scattering light, if the region in question is convex
2. Outgoing light component
    a) Modulated by the angle formed between the eye vector and the surface normal at the point to achieve the micro-occlusion effect in a rough neighbourhood
    b) The micro-occlusion aforementioned may be varied by some subdued noise
3. Effects
    a) Light attenuation from self-shadowing because of roughness in the neibourhood, characterized by
        i. Angle between the light direction and the surface normal at the point
        ii. Roughness of the snow surface
        iii. Local height of the current point(crystals that stick out from a dark region still appears brighter)

iv. Irregularly/pseudo-randomly shaped
v. This effect attenuates the incoming light component in 1. b)

b) Glittering
   i. Fade in and out over very narrowly defined viewing angle
   ii. Sometimes shift in colour(Fresnel effect)
   iii. Cause blooming effect
   iv. Occur randomly
   v. Occurrence is independent on light-viewer relationship
   vi. Even occur in slightly shadowed regions of a snow surface

c) Colours
   i. Generally, bluish white colours dominate
   ii. White with a touch of yellow in highlighted regions
   iii. Bluish gray in partly self-shadowed and darker regions
   iv. Subsurface scattering component is possibly blue
   v. In-scattering component is probably bluish white

From the summary above, one can see that glittering, roughness, in-scattering and subsurface scattering of light are the visual effects that we have identified. During the implementation, I used this summary as a guideline rather than a specification, which means I did not implement everything listed in it. Meanwhile, we were also interested in having a real-time illumination model. There are surely plenty of offline rendering techniques that can generate very realistic lighting effects, but we think real-time applications of computer graphics play an important part nowadays, too. For example, video games, which are one of the main momentums pushing graphics technology forward, and professional simulation software both fall into this category. What's more, new technologies and faster graphics cards keep emerging while researchers keep presenting amazing approaches or new algorithms for real-time rendering. All these help open up a new opportunity to make real-time rendering more realistic. Therefore, we believe that this is a good time to investigate a real-time illumination model for fallen snow.

The objective of this thesis project is to model those visual effects, namely, glittering, roughness, in-scattering and subsurface scattering of light and implement them using rendering techniques, preferably real-time ones. Also be noted that the realistic geometric modeling of fallen snow, i.e., its physical form, is not what this project aims at.

Section 2 discusses some previous work which inspires this one or on which this one is built. The implementation details are unfolded in Section 3 with Section 3.1 describing roughly the experimental platform and Section 3.2 those effects themselves. Section 4 shows the final results of this work followed by conclusive discussions in Section 5.

2. Related work

There are some pieces of research work regarding generating fallen snow correctly over objects. Nishita et al. [Nishita et al. 1997] employed metaballs to model the shape and density distribution of a snow layer over a surface which was defined by Bezier patches. Fearing introduced an interesting way to model accumulated fallen snow [Fearing 2000]. The model comprises of two parts: one is calculating the occlusion factor of a surface by emitting a series of particles aimed upwards towards a sky bounding plane. The other one is after snow is accumulated on a surface, a stability test is performed to move snow away from physically unstable areas in a series of small, simultaneous avalanches. A much more real-time way to model the accumulation of snow was presented by Ohlsson and Seipel [Ohlsson and Seipel 2004]. They used a snow accumulation prediction function which contained an exposure component and an inclination component to compute how much snow a specific point on a surface would receive. This method created fallen snow as a per-pixel effect.

Since this thesis project focuses on a convincible illumination model for a snow surface, how to approximate the interaction between snow and light is what the interest lies in. However, the research about this is quite scarce in the field of computer graphics. Some work has been done about the optical properties of snow, such as [Bohren and Barkstrom 1974]. Chrisman gave a parameterized model of the optical characteristics of snow and implemented it using Monte-Carlo based technique[Chrisman no date]. In [Nishita et al. 1997], they generated a lot of small primitives using metaballs within the snow layer to represent the sub-surface snow particles and when the light was shot into the snow, sub-surface scattering and reflection could be calculated by assessing the interaction between the light and those small sub-surface primitives. According to the photos attached with the paper, the shading did a good job of displaying fine illumination details on a snow surface. Although Ohlsson and Seipel used the normal Phong illumination model in [Ohlsson and Seipel 2004], after the snow accumulation of a certain surface was determined, a noise function was applied to distort the normals of the surface in order to get an approximation of a realistic appearance of snow.

The following research results do not relate themselves to shading a snow surface but they have helped inspire ideas during the progress of the project for coming up with an illumination model. [James and O'Rorke 2004] describes very comprehensively the purpose of using glowing effect and how to implement it on modern graphics cards. The effect was employed to create glittering spots on the snow surface in this project.

As stated in the introduction part, a snow surface looks rough. This is because it has lots of fine details. These fine details are unlikely to be represented by real geometry objects. To add such surface details, there are several existing techniques which can be explored. As early as 1978, Blinn[Blinn 1978] simulated the presence of surface details by perturbing surface normals. This technique is commonly known as bump mapping. Since the real surface of an object remains unchanged, this technique is not capable of shadows cast by those details and silhouettes. Max's horizon map[Max 1988] addresses the shadow problem. By utilizing the power of modern graphics hardware, especially its texture mapping mechanism, Sloan and Cohen implemented the algorithm at an interactive speed [Sloan and Cohen 2000]. Parallax mapping in [Kaneko et al. 2001] produced an interesting result of adding surface details and realizing motion parallax by shifting the texture coordinate of a pixel in question according to the view direction and its depth value. An alternative to parallax mapping is relief mapping. Besides the representation of 3D surface detail and view-motion parallax, relief mapping also produces self-occlusion, self-shadowing and silhouettes. In the paper [Policarpo et al. 2005], Policarpo et al. harnessed the programmability of modern graphics hardware to implement an algorithm that can apply relief mapping to arbitrary polygonal surfaces in real-time. Similar to [Kaneko et al. 2001], this algorithm relies also on the interaction between a view direction and a depth field. Instead of directly using the newly-found(shifted) texture coordinate to sample the texture, however, the texture coordinate is used to access a series of other attributes that are needed for realistic rendering, such as normal and depth. Additionally, self-shadowing can also be assessed by determining if the light direction intersects the depth field before reaching the point in the depth field whose coordinate is represented by the newly-found texture coordinate aforementioned.

That snow surfaces don't appear like hard white plastics in the real world is because when light enters a snow surface, it scatters around inside and then exits the surface, potentially at a different point from where it enters. This phenomenon is called subsurface scattering. One of the visual features of an object with subsurface scattering is translucency. Hao et al. presented an empirical illumination model that changed the local illumination process into a run-time two-pass one [Hao et al. 2003]. The first pass is nothing but the normal light calculation. In the second pass, they used a simplified bidirectional surface scattering distribution function (BSSRDF) model that only focused on multiple scattering. This simplified model defines a neighbourhood for a vertex P which consists of all the vertices that lie with an effective range from P. And then the multiple scattering contribution from each neighbouring vertex to vertex P is computed. Their model gave a visually appealing result of simulating subsurface effect while maintained the frame rate at an interactive level.

3. Implementation procedures

3.1 Experiment platform establishment

To begin with, I need an experimental platform which provides a surface lit by the traditional phong model and other likely-to-use functionalities for this project. Such functionalities comprise, for instance, texture object managing, shader object managing, frame buffer object managing and camera control. In view of future addition of more functionalities, most of them were implemented as C++ classes so future needs can easily be incorporated into this project as additional classes. The surface was modeled by a mesh, of which structure will be discussed in greater details later this section but before that I feel a need to introduce the most fundamental and important class in this project, the Vertex class.

3.1.1 Essentials of the class Vertex

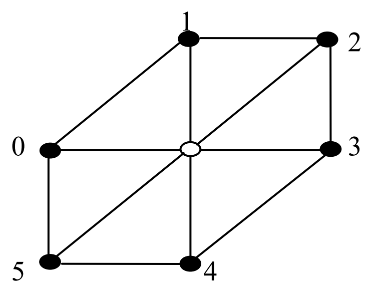


Figure 1: A vertex and its 6 neighbours(in black) forming a triangle ring on the mash

The main class of this project is called Vertex. Each of its objects contains necessary information to be used by this project and by a vertex in OpenGL. The major attributes of this class include: position, normal, neighbour, tangent and curvature. Position is a 3D vector containing x, y, z values of a vertex in the world coordinate system. The normal of a vertex is computed from averaging the normals of triangles that share this vertex. The neighbour attribute is actually an array structure of Vertex pointers. This array holds the six neighbouring vertices that share edges with the current vertex.(see Figure 1) This neighbour array is very important to this class because it contains all the connectivity information that I need for the mesh. The normal, tangent and curvature are all computed based on this neighbour array. When finding the tangent vector of a vertex, I did an approximation: because the overall shape of this mesh is rather flat, we can assume that the tangent vector is formed by subtracting the position of current vertex from the position of its immediate right neighbour, which is the neighbour number 3 in Figure 1. The curvature of a vertex contains the mean curvature. In this thesis work, in order to implement some local illumination effects, we surveyed and used two methods of computing discrete mean curvature for each vertex and stored one of them at a time in the curvature attribute of the Vertex class. For a more detailed description of discrete mean curvature we employed, please refer to the curvature-based local illumination model part of this thesis.

3.1.2 The surface mesh

I modeled a 3D mesh to act as the role of a ground surface that is covered by snow. The basic idea of creating a mesh is I have a bunch of vertices that are evenly scattered on the XZ coordinate plane.(the coordinate system is showed in Figure 2) By evenly I mean the difference between the X coordinate values of any pair of neighbouring vertices is identical and the same applies to the Z coordinate values. Then the rest thing I need to do is assign a height(Y) value to each vertex to make the position information of a vertex complete.

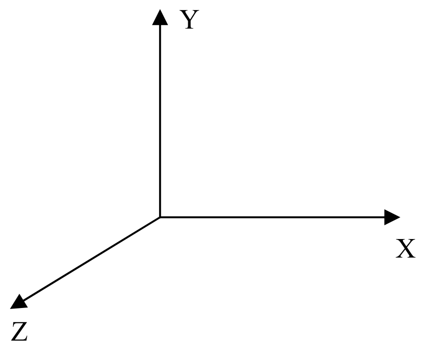


Figure 2: The world coordinate system used in this project

In practice, the mesh locates within -4 to 4 in both X axis and Z axis. And then I use a variable called INTERVAL to control the tessellation of the mesh. So by adjusting this INTERVAL value, I can have the control over the density of the mesh. The height values are acquired by a function called generateHeights. It employs the random function(rand()) in C library to generate a series of values for vertices and I have these random values divided by the maximal possible random value to scale them down to a small range between 0 and 1 so as to get a relatively flat surface. However, merely restricting the heights of vertices to a small value range won't create a naturally flat surface because there can still be a lot of small spikes on the surface. Hence, I wrote a box filtering function and applied it to the surface several times. The mesh is formed by rows of triangle strips. Each pair of neibouring strips has a row of vertices in common. But because of the formation of a triangle strip in OpenGL, I need to duplicate such row of vertices in order to form two neibouring strips. Since this duplicated row of vertices have exactly the same attributes of the original ones, they will not be spotted in the real rendering. For an illustration of this duplication idea, please refer to Figure 3.

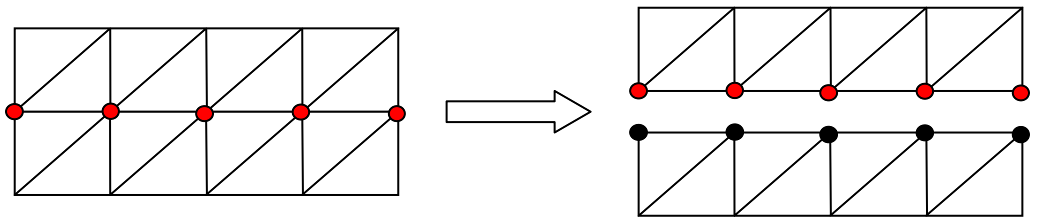


Figure 3: Common vertices(in red on the left) of two neighbouring rows of triangle strips are duplicated(in black on the right)

After having the mesh structure, I applied the phong model to it. The whole scene has only one light source acting as the sun. Both the global ambient term and the ambient term of the light source I use have a small blue flavor to make the shading of the mesh appear a little blue thus better resembling a real snowy ground. I set this light source as a directional light because the sunlight is considered to be directional.

3.2 The illumination model

3.2.1 The glittering effect

The first apparent feature we found on a real snow surface on a sunny day is that there are a lot of shining spots on the surface because of the reflection of sunlight caused by snow crystals. And these spots are dynamically changing when the viewer changes his position(actually, so long as his head budges a little, some of the spots will change responsively.) I further observed that if the snow surface lay between the viewer and the sun, some crystals would act like prisms, which means some spots would appear other colours than bright white. In order to simulate such effect as some random spots become shiny, my first idea was randomly picking out some pixels in my pixel shader that generates the final scene and making them white. To do this, I generated a texture, each of which's texels contains 4 random values. Then I loaded this texture into the scene pixel shader and treated each texel as a facing vector for a corresponding pixel on the surface. To put it in another way, I assumed that each pixel represented a one-facet crystal and it had a random facing angle. Note that here the facing angle has nothing to do with the surface normal at this pixel. It was added only for the purpose of introducing randomness into shining spots. The selection criteria of a pixel to be set to white would then be if the dot product of this facing vector and the eye vector(both vectors were normalized first) was within a small value range(in the program, such range is between cosine 0° and cosine 2°) and that pixel was facing towards the sun(I still used the interpolated surface normal at this pixel to determine this). I also made the camera move very slowly at each key pressing so that this shining effect could come and go more realistically. However, in action, it was nothing like the real phenomenon. My supervisor pointed out that I needed to have more than one pixel to represent this effect. Such an effect can be achieved by a pixel effect called blooming which bleeds the colour of a pixel out into its neighbouring pixels. To demonstrate the effect, see Figure 4.

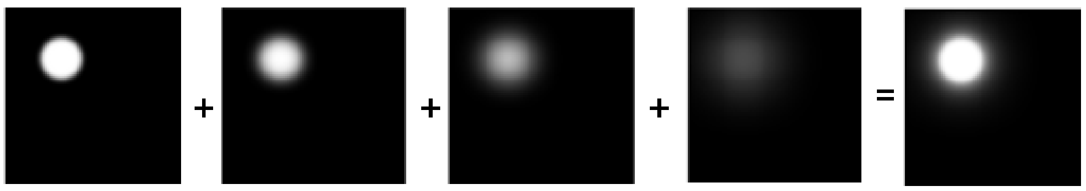


Figure 4: The four images on the left side of the equal sign are images blurred with Gaussian filters whose sizes are 5×5, 11×11,21×21 and 41×41 respectively. When we add them together, the image on the right most side really glows.

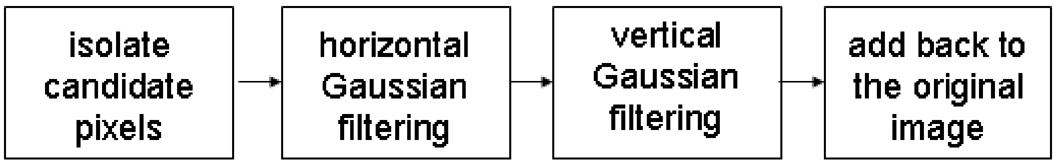

isolate candidate pixels
horizontal Gaussian filtering
vertical Gaussian filtering
add back to the original image

Figure 5: The flow chart of performing blooming

In this paragraph I will describe how I used the blooming technique in the project. Figure 5 gives the major steps of this process. First, continuing from the last attempt, I set the alpha channel of those randomly selected pixels to a nonzero value in the scene pixel shader so that these pixels would be told apart from other pixels whose alpha channels are zero. And from here I introduced render-to-texture technique using frame buffer objects(FBO) into my project. The scene was rendered to a texture first. Then when I rendered a full screen quad with the scene texture mapped onto it, a pixel shader called brightstrip was used to pick out the selected pixels by testing the alpha values of each pixel (the first box in Figure 5). To have that prism effect aforementioned, in the brightstrip shader, I used the eye vector together with the light direction to see if the current pixel needed to yield out another random colour instead of just white. That is, if the angle between these two vectors is between 120° and 170°, a non-white colour should be output for this pixel. After running this shader, the FBO contained a texture of only those selected pixels(white or other colours for the prism effect) and the rest were suppressed to black. The remaining tasks are regular blooming steps: I created three FBO's with each one half of the size of the previous one. By rendering the texture of selected pixels into these three FBO's sequentially, I got a good blooming result without making a bigger and bigger blurring kernel, which saved some time during running. Since I used 5×5 Gaussian kernel, during the blurring process, I separated it into horizontal and vertical kernels to gain some speedup again(this process is displayed in Figure 6 and corresponds to the second and the third box in Figure 5).

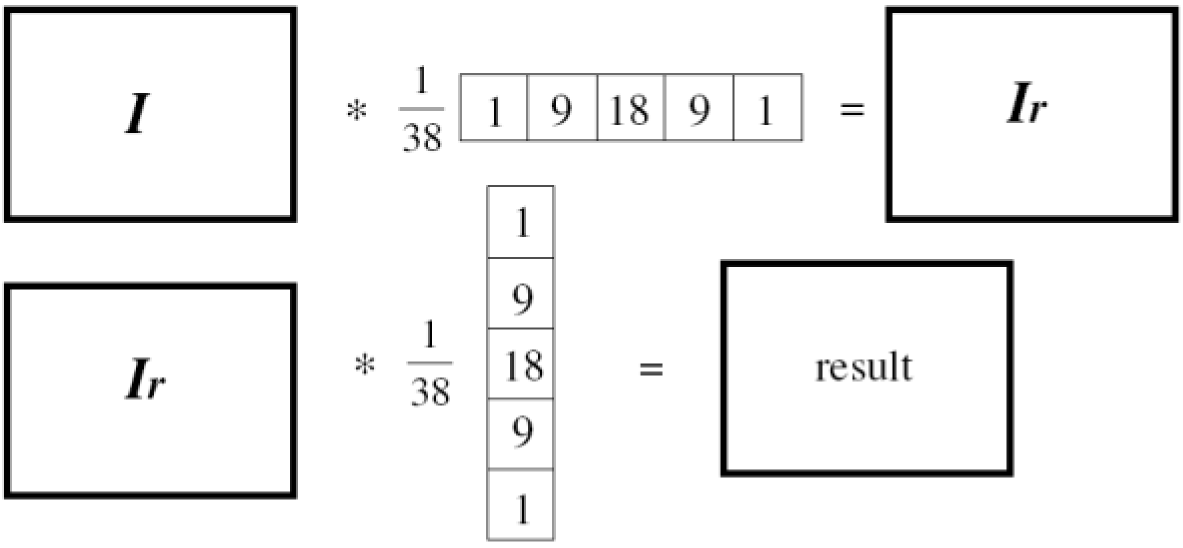


Figure 6: The separated 5×5 Gaussian filter used for blooming

At last, I combined the serial blurred results with the scene image to have the blooming effect in place (the last box in Figure 5) . Worth noticing, I also tried to use high dynamic range rendering (HDRR) to enhance the shining effect because I sensed that the shining spots were not just white in the real world but were actually much brighter. HDRR is a rendering technique using lighting calculations done in a larger range. The tradition graphics libraries and hardware only support colours whose individual channels ranging from 0 to 1. This possibly makes, for example, a light bulb have the same white colour intensity as the sun has in a scene. As we know in reality, this is hardly the case. With the support of HDDR, a very bright light source can exceed the limit of 1 to simulate its actual intensity value. At the end of the rendering, we just need to apply a mechanism to scale the range back to between 0 and 1, since our display devices still only have a 0 to 1 range. Nonetheless, since my graphics card is not fast enough (Nvidia Geforce go 6100), though it supports half float texture, the resulting rendering was simply too slow to even interact with. Therefore, I decided not to use it and the rendering with only blooming is already good enough to realize the glittering effect from the real snow surface.

3.2.2 Fine surface details

Snow surfaces don’t look like uniformly white shells covering grounds because they contain a lot of crystal bodies, which makes the surfaces appear rough, full of small details. Therefore, another visual effect observed from a snow surface we are interested in is roughness. We spent most of the time striving to get a convincing result of simulating this effect. These attempts are to be described in details within paragraphs coming after.

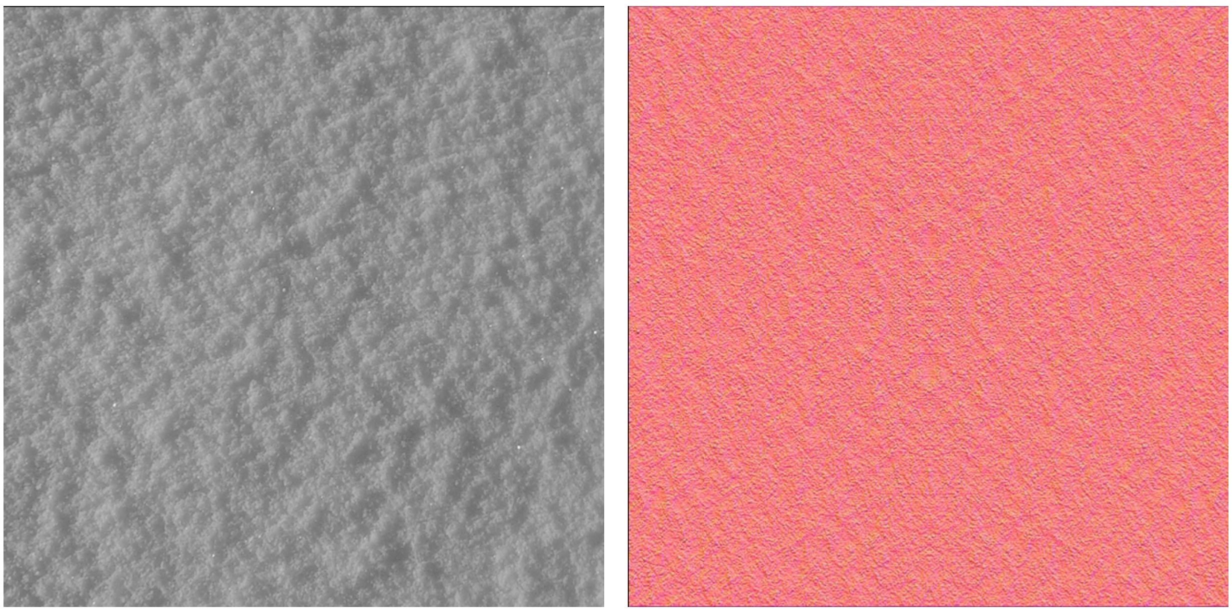


Figure 7: A close-look snow surface(left) and its normal map(right)

The first approach that we thought of is naturally normal mapping, which is commonly used for adding surface details without needing more polygons. We generated a normal map from a close-look snow picture (Figure 7). Since the generated normals are defined in each vertex's own tangent space, in order to have a correct lighting calculation later, we need to perform a vector transformation. Either these normals are transformed into the eye space where the light vector is represented or the light vector is transformed into the tangent space. In the implementation, I chose the first option and performed the transformation in a fragment shader called scene. With the interpolated normal (the geometry normal) and the interpolated tangent vector from the corresponding vertex, it is very easy to compute the third basis vector, which is called binormal, to form a matrix to transform generated normals into the eye space. After that, I substituted the geometry normals with generated ones to manipulate the light's interaction with the surface. Figure 8 shows the rendering result of combining both the glittering effect and the normal mapping. Interestingly, though I did not intentionally implement any mechanism to control the size of shining spots, as you can see in the figure, the shining spots do vary in size. I speculate it is due to the addition of blurred candidate pixel images to the final scene image. Suppose there is a candidate pixel, whose colour is close to the one at the place in the final scene image where this candidate pixel locates. After adding them together, the blurry periphery of the pixel actually blends in with the surroundings, thus becoming a smaller shining spot. The result demonstrates some promise, but it is still far from being convincible. Besides, we would like to quest some new ways to realize this effect. Generating micro shadows became our special interest because we inclined to believe that it was these shadows that contributed the most to such a visual effect. Inspired by [Max 1988] and [Sloan and Cohen 2000], we decided to give horizon map a try to see how we could exploit it for the purpose of this project.

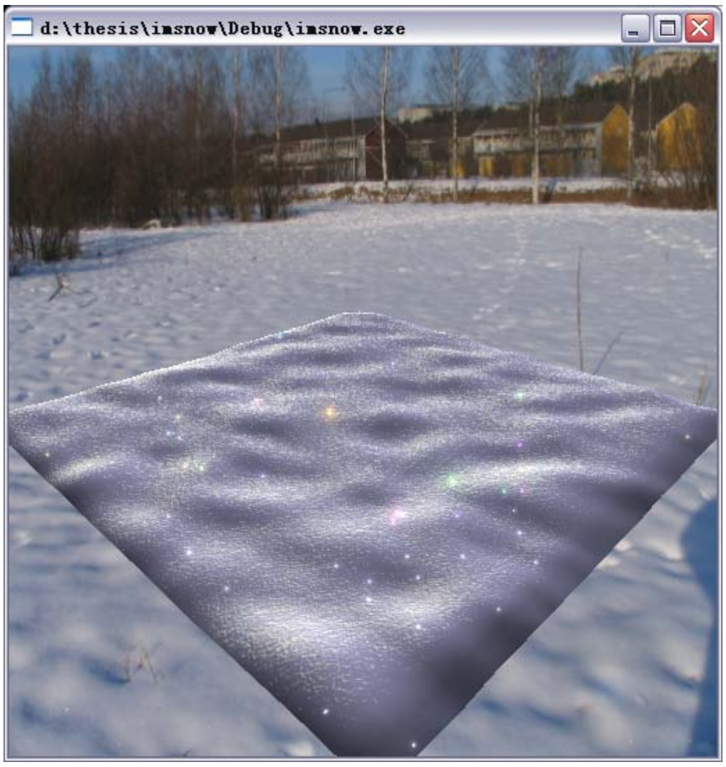


Figure 8: Combination of both glittering and normal mapping effects. The snowy background here is just for providing a context to make the surface more realistic.

Given a specific light position, a point P on a surface is in the shadow of another point Q if the vertical angle from P to Q is greater than the one from P to the light source so the idea of a horizon map is to precompute, for each point on a surface, the greatest vertical angle between it and other points in a given direction. Usually, only a set of discrete directions are sampled and an arbitrary angle can then be interpolated. Since we are simulating a snowy scene and the light source we have is the sun, this light source is not supposed to change. Based on this assumption, in order to get a faster precomputing process, I only computed the horizon angle in the light direction instead of 8 or more described in those papers. The implementation of horizon mapping can be broken down into two parts: the first part, which is the precomputation of horizon angle for each vertex, is done in the main program. Horizon angles are stored as vertices' attributes in the Vertex class. The second part is applying the horizon angles to the rendering to determine shadow parts. This is done in a fragment shader which generates the scene. I implemented a member function called computeHorizon in Vertex class. It takes the x and the z coordinate of the light source. By calling this function, each vertex will get its horizon angles in the light direction.

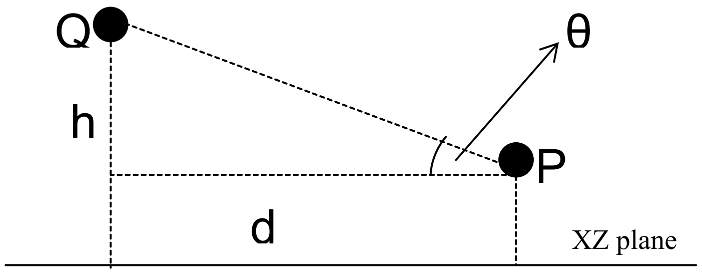


Figure 9: Illustration for the idea of horizon map

As you can see in Figure 9, the tangent of angle θ can be computed by d/h, where d is the distance from vertex Q to vertex P in plane XZ (Q is one of the vertices that lie in the light direction); h is the difference of height between P and Q. Note that if h is less than or equal to 0, that specific Q will be ignored for the calculation. After the function has enumerated all Q's, the angle θ which has the largest tangent value is the horizon angle for vertex P. Since θ is between 0° and 90° , the tangent is a monotonically increasing function. Thus, I simply stored the tangent value in place of the actual angle. During the run-time, such tangent value will be computed again but between the light source and a specific fragment that belongs to the surface and then the newly computed tangent value is compared with the fragment's horizon value (interpolated from a corresponding vertex's horizon angle attribute). If the fragment's value is greater, this fragment is occluded from the light and thus is in shadow. The result of this rendering is shown in Figure 10, which is definitely unsatisfactory.

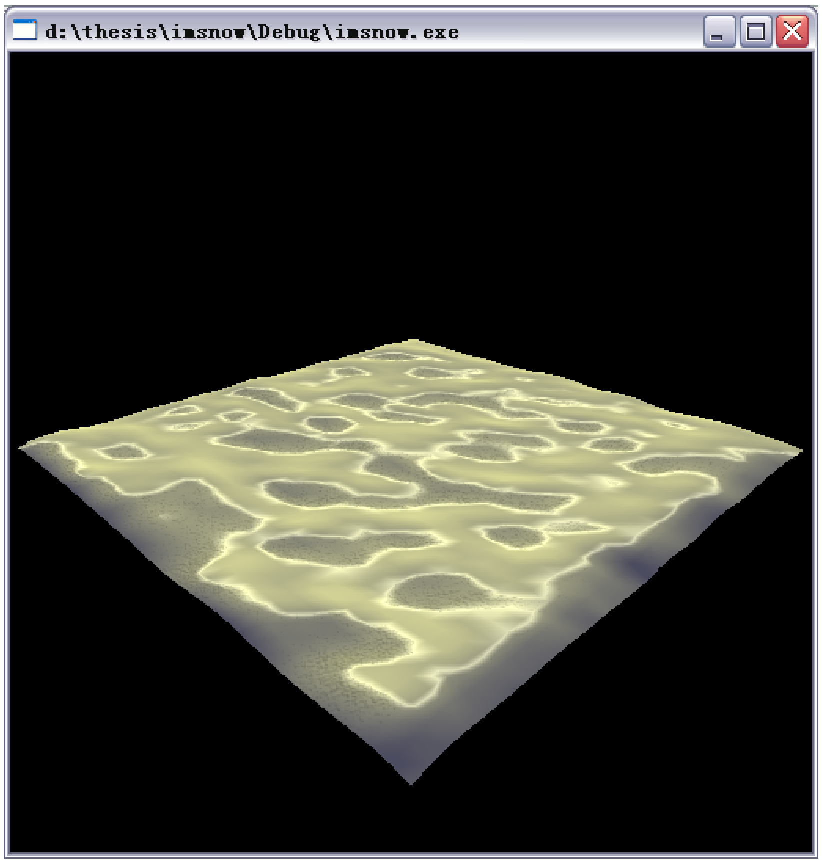
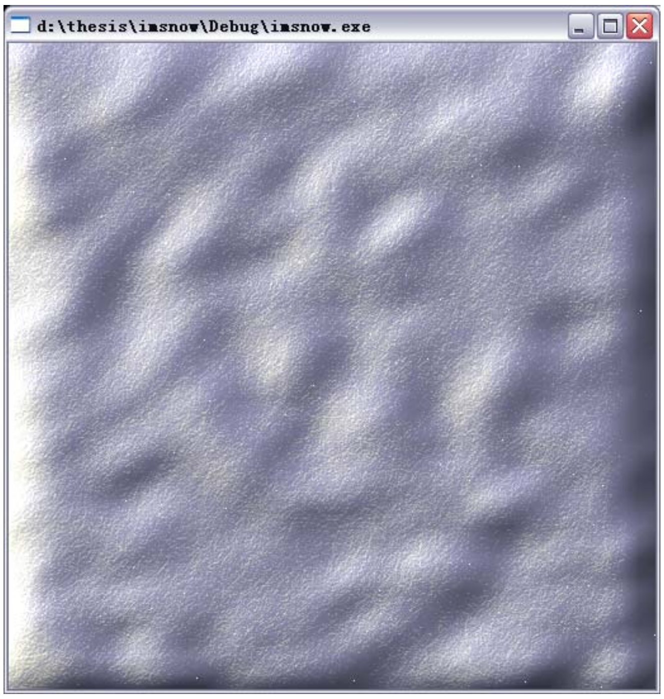


Figure 10: Shadow computation using per-vertex horizon map

As for the reason why it turned out like this, I reckon it is because the granularity of per-vertex horizon is too large. After interpolating these values, a region of pixels receive very close horizon values. The consequence of this is either their horizon values are all greater than the angle from the light source(in this case, all are occluded) or all are smaller (in this case, all are lit).

Now that per-vertex was too coarse, naturally we turned to per-pixel scale. To perform per-pixel horizon angle calculation on the surface, for a specific pixel P on the surface, I should be able to traverse all the pixels (and these pixels only) that comprise the surface along the direction of the light starting with P and get the position information for each pixel that I traverse. The steps of achieving this are described as follows. First, I placed the camera at the centre top of the surface and then I set the projection matrix to an orthographic matrix which defined a parallel viewing volume. This volume was defined in such a way that its left, right, top and bottom clipping planes were exactly aligned with the borders of the surface so that it was certain that no part of the surface would be clipped nor would the background pixels be included in the final scene (See Figure 11 left). Second, like the blooming effect, the scene was first rendered to a colour attachment of an FBO. In order to maintain the position information for each pixel after the scene was rendered, I used multi-target rendering. The basic idea of it is attaching more than one colour attachments to an FBO and then drawing them within one pass by using gl_FragData array in a fragment shader. In practice, I encoded the world coordinate of each pixel in the second colour attachment in the scene FBO so that when I accessed a texel in that attachment, I would get the x, the y and the z values of the corresponding surface pixel from the texel's r, g, and b channel respectively(See Figure 11 right). At last, I wrote a fragment shader called pixelHorizon. When it is executed, it will take the position texture (Figure 11 right) as input and will traverse all the texels in the light direction starting from each texel to compute the tangent values similarly to the per-vertex version. Candidate texels along the light direction can be located by calculating their texture coordinates in the pixelHorizon shader. However, the result is still not usable (Figure 12). The light is coming from the top left corner and the darker a pixel is in the image, the greater horizon angle it has. We can be sure that this approach itself is correct since the darker parts comply with the valleys seen in Figure 11 left. Note that the right and the bottom borders are especially dark, which is because the real surface at these parts are much lower than the other parts. What is more, across the surface, one can observe that the darker areas do have soft borders and smooth levels of transition. The unsatisfactory part of this result, however, is that it is also accompanied by a lot of obvious artifacts, which mean not all the horizon angle values were obtained correctly. I spent a considerable amount of time trying to overcome them but failed. I reckon that they were introduced by the discrete structure of a texture image. Theoretically, points on a surface in the light direction form a straight line but in practice these points are actually scattered closely around this line. Plus, when we calculated the texture coordinate for the next candidate texel, the specific coordinate may not hit a texel so the hardware will use filtering mechanism to find one. At last, the distance (d in

Figure 9) between a pair of neighbouring pixels was assumed(0.02 in my implementation). And there is no real data we could rely on to refine this assumption. All these factors can result in unwanted position information and thus anomaly occurs during calculating horizon angles.



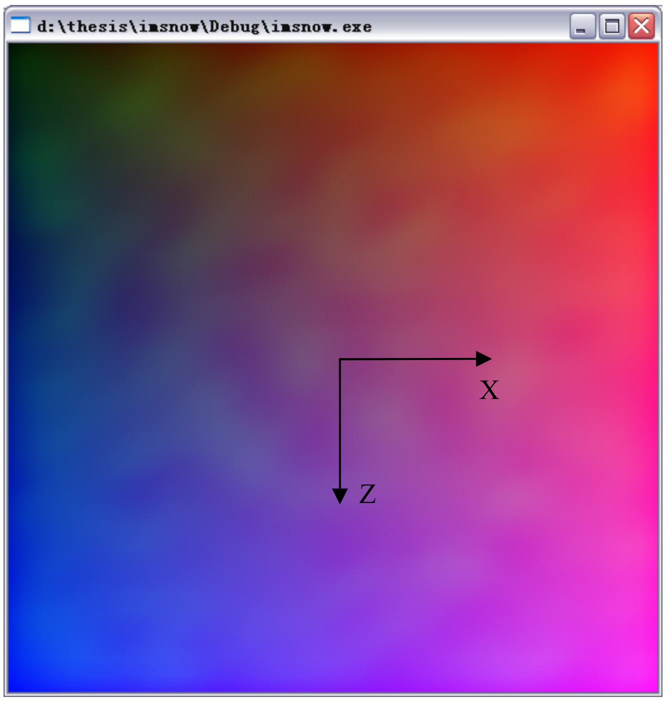


Figure 11: Bird-eye view of the snow surface(left) and colour-coded world coordinates of the surface, with the Y axis facing outward the paper (right)

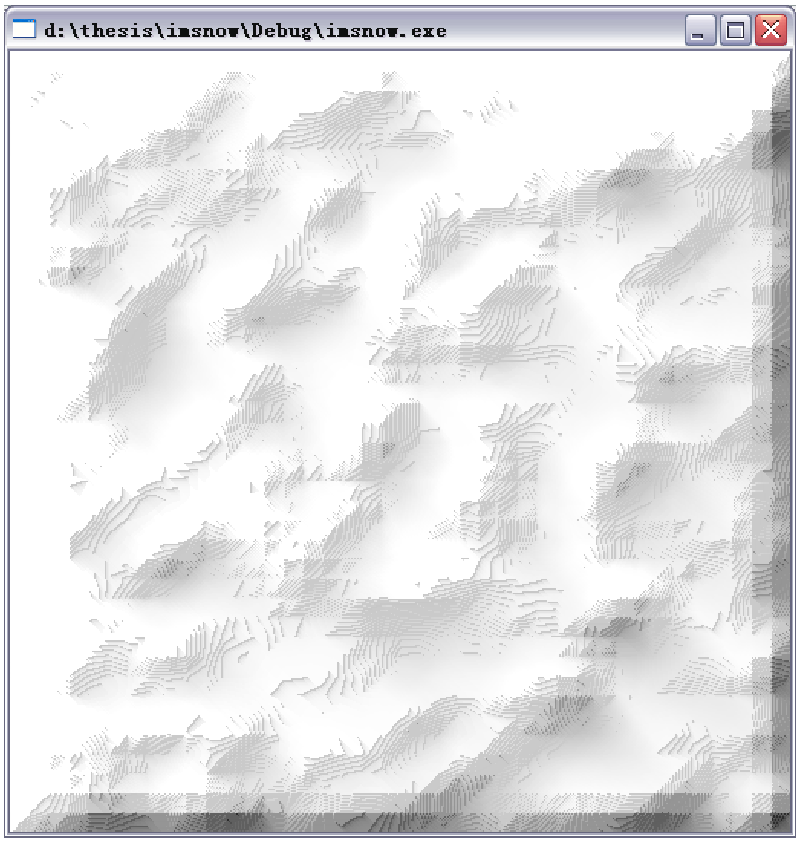


Figure 12: Per-pixel horizon angles in the light direction

3.2.3 Curvature-based local illumination model

In view of that the endeavours to model fine details on a snow surface were not very fruitful, at the last phase of my thesis work, we tried to soften the very requirement of modeling those micro shadows by devising a local illumination model for in-scattering and subsurface scattering effects. So what are the effects of in-scattering and subsurface scattering that we tried to model here and how do they differ from the original micro shadow effect? Before diving into the explanation, I would like to clarify that these two effects were come up with by common senses we have about snow. Therefore, it is an empirical model rather than a physically correct one.

The in-scattering effect tries to mimic the light enhancement in the concave areas on a snow surface. A snow surface consists of a lot of snow crystals which act like mirrors reflecting incoming light and in those concave areas the incoming light can be reflected multiple times, thus making those areas brighter than usual. However, if they are too concave, then the light is actually prevented from reaching these areas, which in turn makes these areas darker than usual. All in all, the in-scattering model has such a character that if an area is concave to some extent, the area will be brighter but if its concavity exceeds some extent, it will receive less light. Figure 13 illustrates the effect.

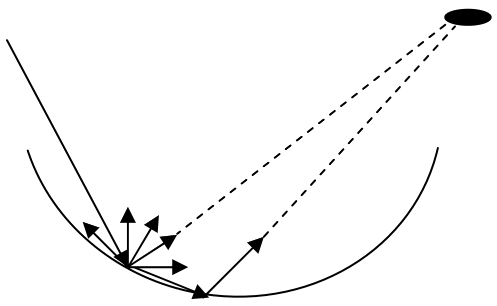


Figure 13: In-scattering effect, only showing the light enhancement case

An object with subsurface scattering effect tends to be translucent. That is because, although light can penetrate the surface of the object, the farther it travels within the object, the more it is attenuated and diffused. We assume this translucent visual effect is most prominent when the object, the light source and the viewer are on the same line while the object is between the light source and the viewer. This model tries to simulate this effect by adding a soft blue colour at the convex parts of a snow surface. The amount of the blue colour added depends on the angle between the viewing line and the light direct so, as stated above, a convex part of the surface will take on the most blue colour when the translucent effect there is most apparent. Figure 14 helps to understand the concept of subsurface scattering we tried to model better.

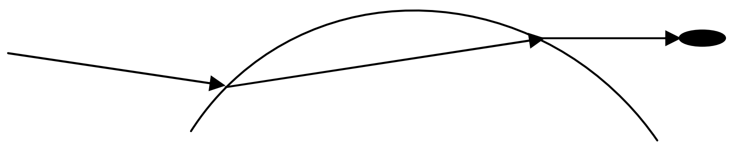


Figure 14: Subsurface scattering effect

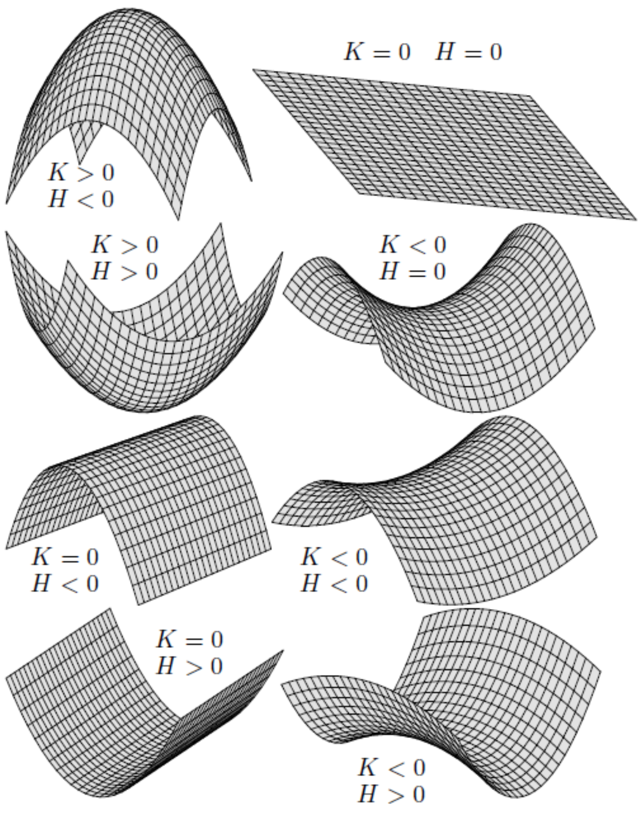


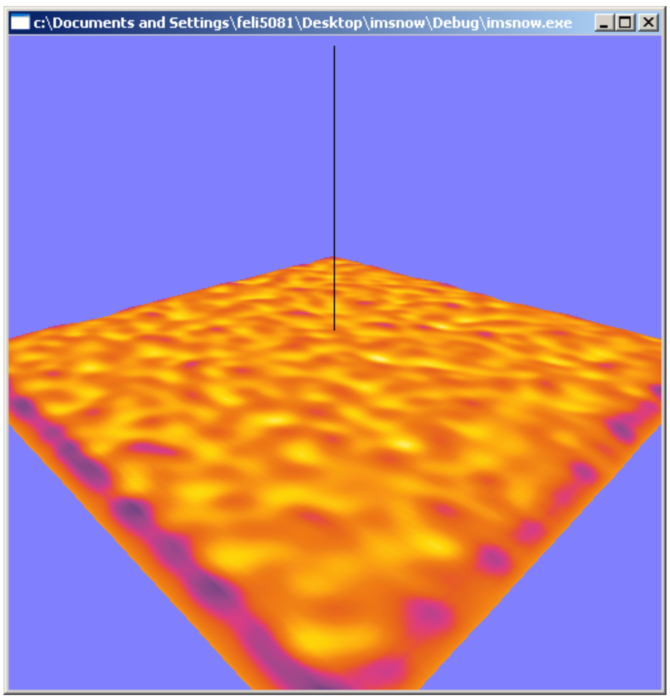
Figure 15: Relation between sign of surface curvatures and surface shapes

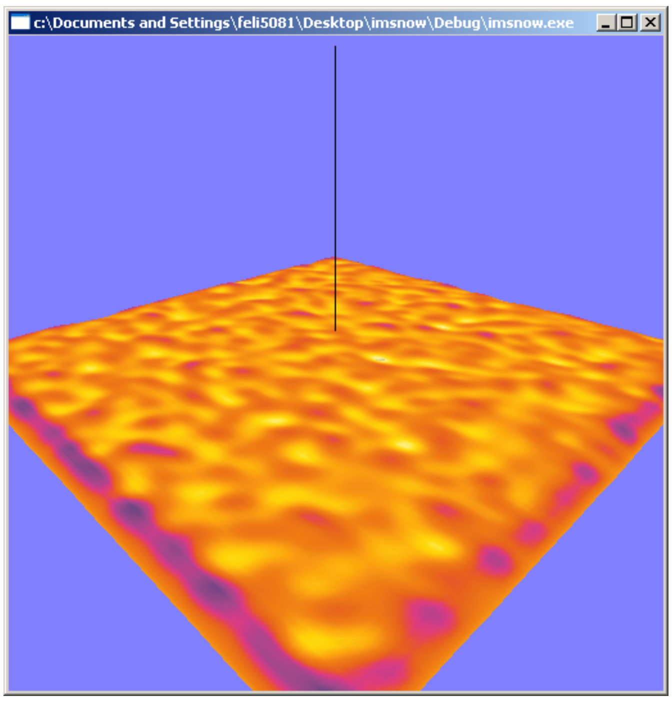
After briefly describing the effects, a question inevitably pops up: how to tell that a given part of the surface is concave or convex? To answer this question, let us have a look at Figure 15. The letter H and K stand for mean curvature and Gaussian curvature respectively. Those shapes cover the possible cases of surface concavity and convexity quite comprehensively and casting away the meaning of mean and Gaussian curvature for the time being, we can observe that by only using the mean curvature we are able to delimit the convex cases and the concave cases fairly well.

And now what we need to do is compute the mean curvature, denoted k_H, for a given point on the surface to continue with modeling those effects. Now let us go back to pick up those related important concepts needed to explain what a mean curvature k_H is before moving on to the acquisition of it. Let us start by how the curvature concept is applied to a point on a surface. As we know, for a plane curve, the curvature at a point is a measure of the changing rate of the tangent angle with respect to an arc length. For the point on a surface, we can find an infinite number of curves that pass through this point. Each curve from this set, which can be considered as a plane curve, thus, has a curvature at this point. The maximum and the minimum of such curvatures are called principal curvatures, denoted k_1 and k_2. Additionally, for each curve that passes the point, we can find a tangent vector at the point and the directions of the tangent vectors which belong to curves having k_1 and k_2 are called principal directions. At last, the mean curvature for a point on a surface is defined as $(k_1 + k_2)/2$. Again, to make it clear, since the mean curvature is merely a local, namely, vertex-based, attribute of a surface, the light enhancement, the light suppression and the translucency are thus regional accordingly. Besides, the light suppression from the in-scattering effect can be deemed as enlarged shadows comparing with micro shadows. So it is interesting to see how the result would turn out to be. If it is good, we actually have a cheaper way to generate the shadow effect.

Two methods were exploited, when it comes to calculating k_H. One is stated in [Meyer et al. 2002]. They presented an algorithm using a voronoi region established around each vertex on the surface to compute a K, which exists $K = 2*k_H*$ n, where n is the normalized normal at that vertex. Therefore, the value of k_H is actually half of the magnitude of K and the sign of it is determined by if K and n point to the same direction or the opposite one. The other way can be found in most differential geometry related books, for example [Kreyszig 1991]. It can be summarized as follows: for each vertex P in the surface, a polynomial function of degree two is fitted to the N (in this case N equals to 9) closest vertices using the least squares method and from that the curvature is calculated analytically. Figure 16 visualizes the results of these two mean curvature computation methods. Each pixel's curvature value was normalized to the range of 0 and 1 and then 0 was mapped to the left most colour in the colour map in Figure 16 while 1 was mapped to the right most colour. Comparing Figure 16 left with right, we can see that the differences are minor. Hence, we can use either result in our rendering.

Colour map used for this visualization


c:\Documents and Settings\feli5081\Desktop\imsnow\Debug\imsnow.exe


c:\Documents and Settings\feli5081\Desktop\imsnow\Debug\imsnow.exe

Meyer et al.'s method Least squares method

Figure 16: Visualization of surface mean curvatures

With the knowledge of local surface information, we can finally come back to model the two local illumination effects. We modeled the in-scattering as

$$I_{Is}(k_H) = 1 + \mu \bullet e^{\left[-\frac{(k_H - 0.5)^2}{\sigma}\right]} + \varepsilon \bullet k_H{}^2$$

because it gives us the profile depicted in Figure 17, which complies with the empirical result we desire about the in-scattering effect. The parameters μ,σ are brought in by the bell curve and ε is from the quadratic curve, which both control the characteristics of the final curve. Likewise, we have the subsurface scattering as

$$I_{SS} = K_{SS} \bullet \left[0.5 - 0.5 \bullet (\vec{V} \bullet \vec{L})\right]^{\beta} \bullet k_H{}^{\gamma} \bullet (1 - \vec{V} \bullet \vec{N}) \bullet (\vec{V} \bullet \vec{N}),$$

where K_{SS} is the material value for the subsurface scattering colour; $\vec{V}$ is the viewing vector pointing to the viewer; $\vec{L}$ is the light direction vector pointing to the light source and $\vec{N}$ is the normal vector at the surface point in question. The locus of the model is plotted in Figure 18.

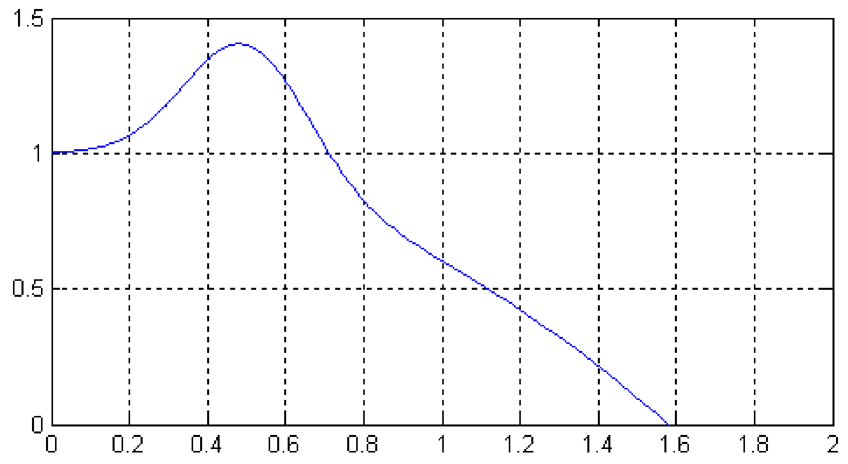


Figure 17: Relation between k_H (abscissa) and Iis (ordinate),where μ=0.5, σ=0.05 and ε=-0.4

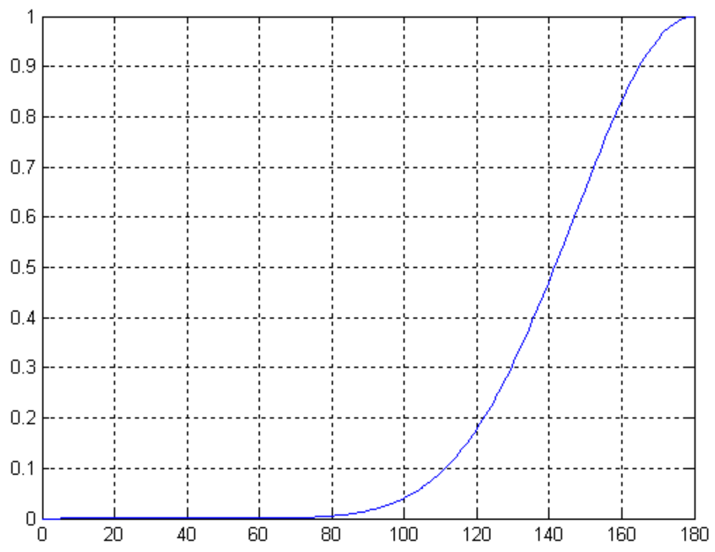


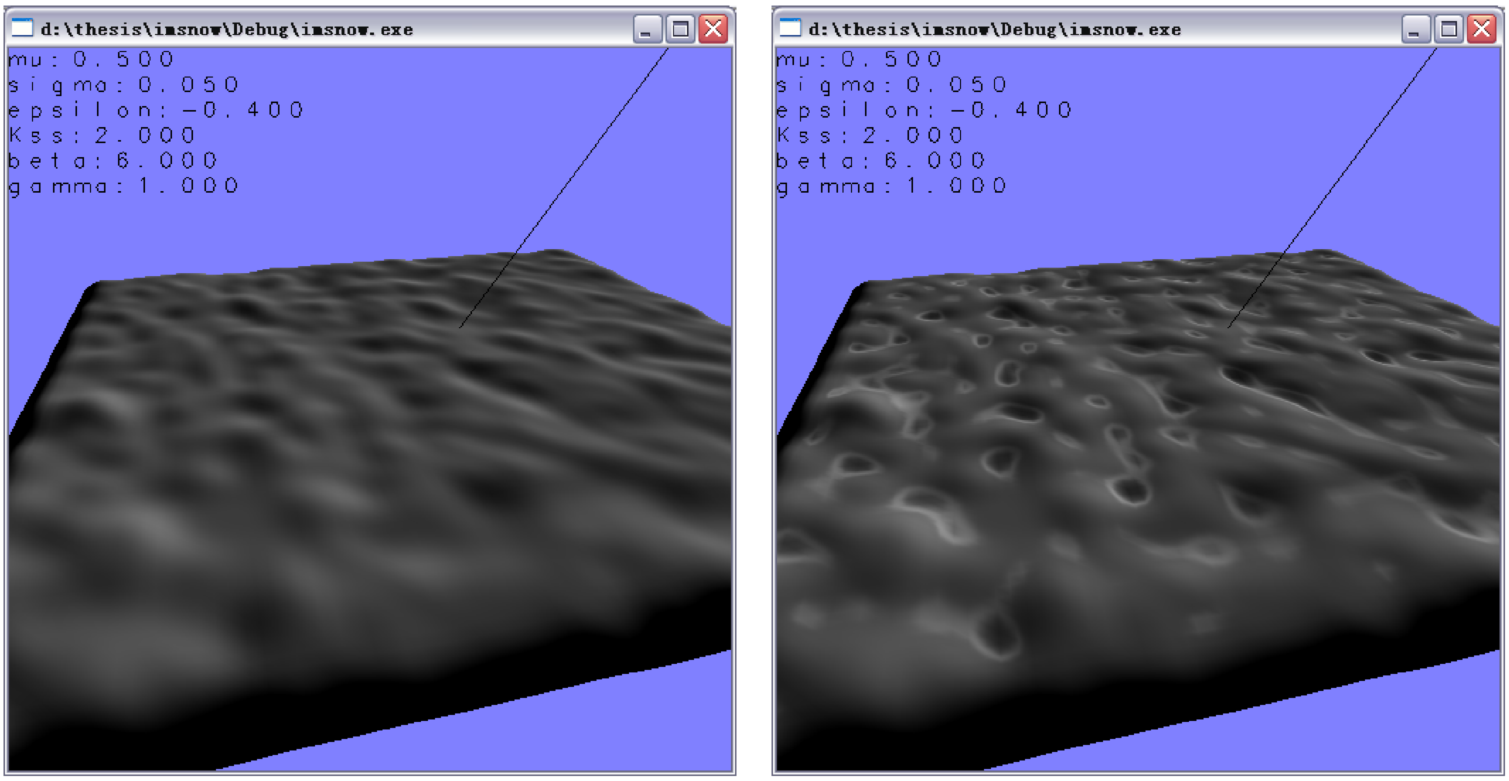
Figure 18: Relation between Iss (ordinate) and the angle formed by $\vec{V}$ and $\vec{L}$ (abscissa), where Kss=2, β =6 and γ =1

The last step is branching the logic of the scene fragment shader based on the mean curvature value the fragment receives. If k_H is greater than 0, before the final colour is output, the diffuse component of it is multiplied by the in-scattering factor. Otherwise, the subsurface scattering model is used. Originally, we thought that the subsurface scattering effect could be applied to the surface by adding an additional colour, in this case, blue to the corresponding parts on the surface. However, in practice, since the colour of the surface is already close to white, namely, close to 1 for each colour channel, it is impossible to make the surface colour blue by adding any colour. In fact, on the contrary, I subtracted values from some colour channels to achieve this goal. I defined a subsurface scattering colour and after computing the output colour for a fragment, the subsurface scattering colour is subtracted from the output one to make it appear blue. Finally, the program was designed such that those parameters in the two formulae can be interacted during the run-time for fine tuning in order to get a result as convincible as possible.

4. Results

In this section, I will show the rendering results of the local illumination model and a final combination of it with the glittering effect. In the end, some thoughts about improving the in-scattering effect are added as discussions.

4.1 The in-scattering effect


d:\thesis\imsnow\Debug\imsnow.exe
d:\thesis\imsnow\Debug\imsnow.exe

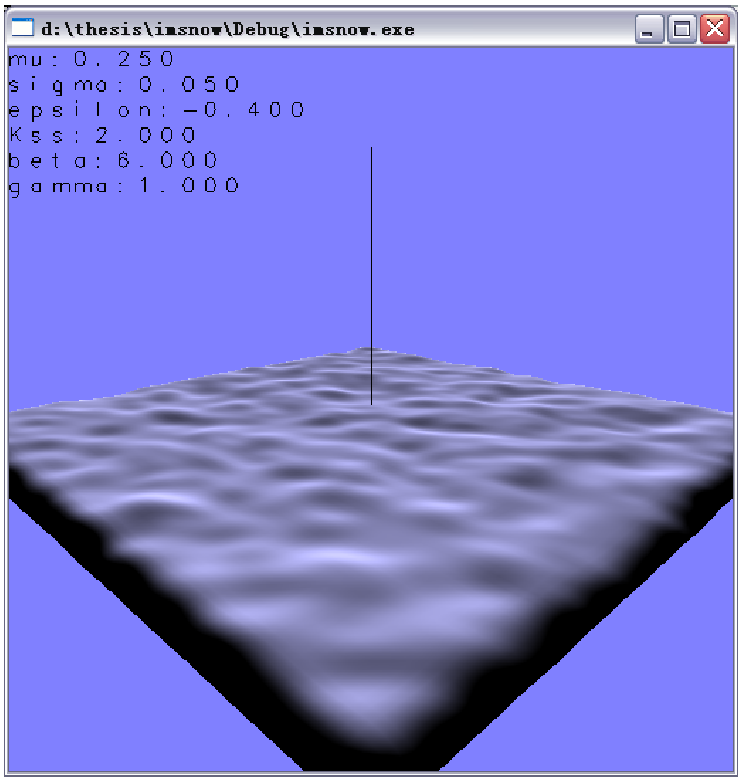
```
mu: 0.500
sigma: 0.050
epsilon: -0.400
Kss: 2.000
beta: 6.000
gamma: 1.000
```

```
mu: 0.500
sigma: 0.050
epsilon: -0.400
Kss: 2.000
beta: 6.000
gamma: 1.000
```

Figure 19: Rendering result of the in-scattering effect

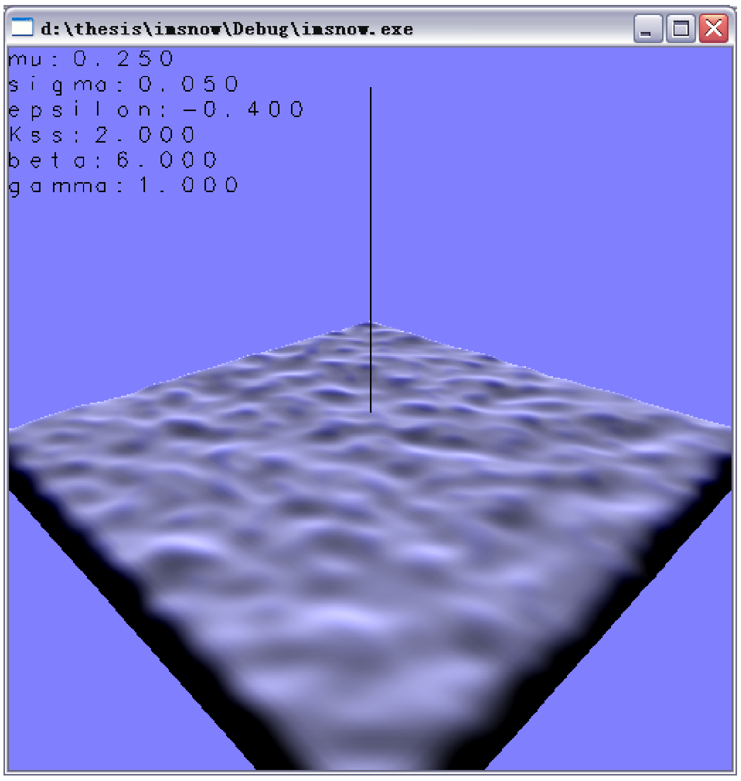
Both images in Figure 19 contain only the diffuse term of the phong illumination model, since the in-scattering effect only affects the diffuse term according to my implementation. The left image in Figure 19 has only the diffuse component, serving as a reference. The right one adds the in-scattering effect computed by the formula using the mean curvature (the value for each parameter in the formula is also displayed). The first thing one can notice is that those areas taking on such effect are concave areas, which complies with our initial intention. Secondly, some concave areas receive an enhanced lighting and some receive a suppressed one. However, those regions which receive suppressed lighting are all accompanied by brighter rings along the rims.

4.2 The subsurface scattering effect



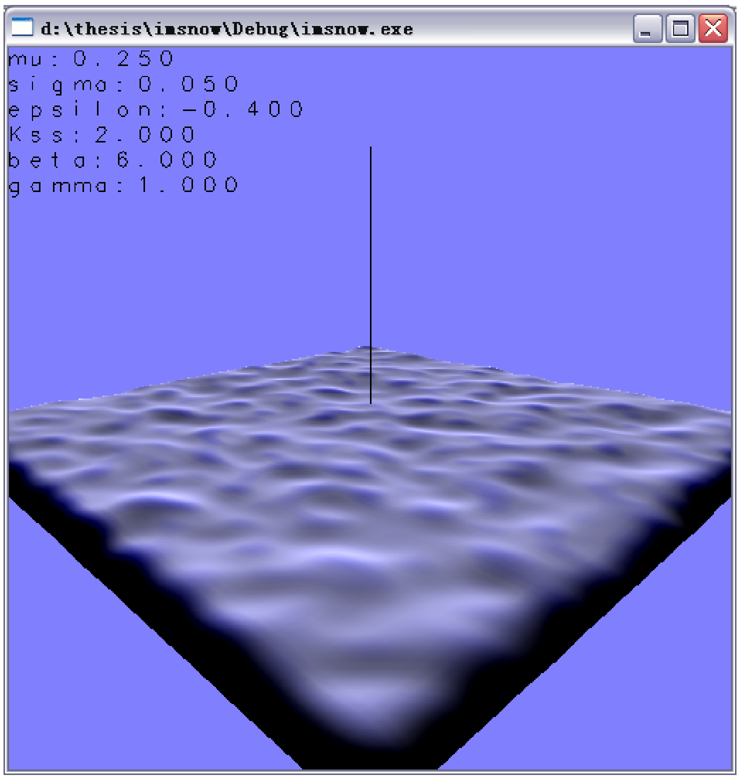
```
mu: 0.250
sigma: 0.050
epsilon: -0.400
Kss: 2.000
beta: 6.000
gamma: 1.000
```

a



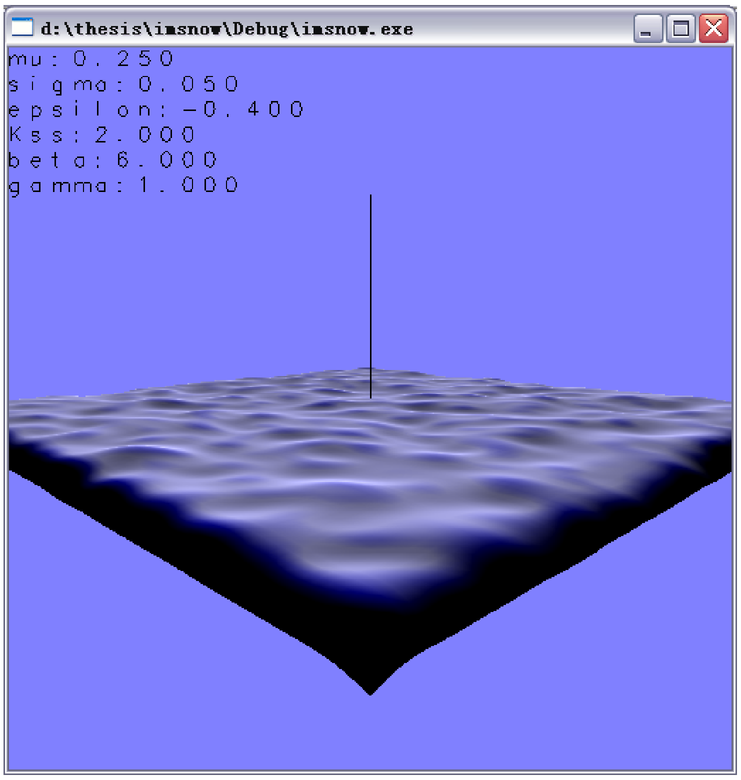
```
mu: 0.250
sigma: 0.050
epsilon: -0.400
Kss: 2.000
beta: 6.000
gamma: 1.000
```

b



```
mu: 0.250
sigma: 0.050
epsilon: -0.400
Kss: 2.000
beta: 6.000
gamma: 1.000
```

c



```
mu: 0.250
sigma: 0.050
epsilon: -0.400
Kss: 2.000
beta: 6.000
gamma: 1.000
```

d

Figure 20 : Rendering result of the subsurface scattering effect

Figure 20 demonstrates the subsurface scattering effect. Figure 20 a is the phong illumination model with only ambient and diffuse terms. Likewise, we can see that the subsurface scattering effect only exists in the convex regions of the surface. From Figure 20 b to Figure 20 d, the angle between the view vector and the light direction vector increases and one can really observe that the effect relates itself to this angle, as we desired.

4.3 Putting it all together

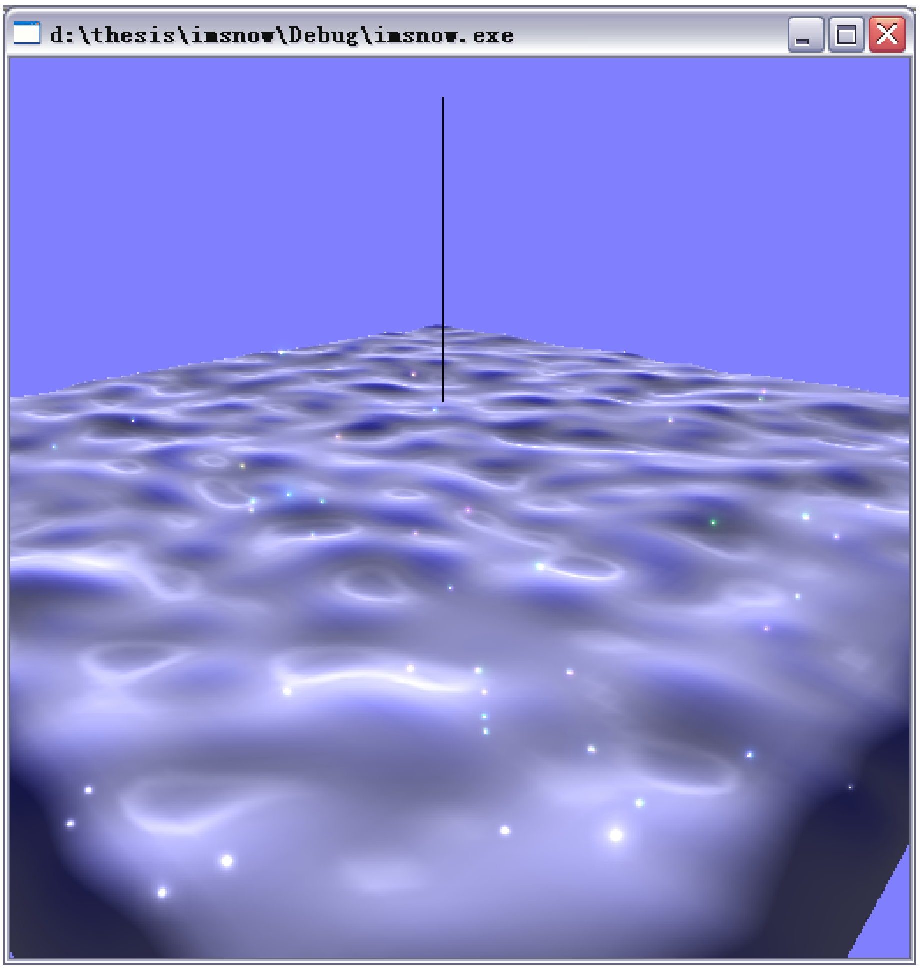


Figure 21: Combination of the local illumination and the glittering effect

The result of the in-scattering effect has one major issue, which, having already been pointed out in section 4.1, is each light-suppression regions comes with a bight ring along its edge. I infer this phenomenon has something to do with the enhancement-suppression transition brought in by the model. In order to verify this assumption, we need Figure 17 and a histogram of the mean curvature values of all the vertices, which can be seen in Figure 22. From Figure 17, we find that after around 0.7 the curve begins to decline below 1. Now let us have a look at the positive mean curvatures in Figure 22. There are still roughly 487 vertices whose mean curvatures are equal to or greater than 0.7. Given a surface with a relatively dense tessellation, a concave region can consist of several such kind of vertices. Due to linear interpolation, pixels in this region thus have mean curvatures that indicates light suppression. This is true for most of the pixels in the valley of the concave region. When it comes to the rim region, the interpolated curvatures are influenced by surrounding convex regions, whose curvatures are negative. The result is making the curvatures along rims less, which may change them to light enhancement (in this case, for example, make curvatures less than 0.7). Hence, there are brighter rings along concave areas' borders. Such phenomenon can not be changed by merely tuning ε in the formula, which modifies the steepness of the suppression slope. In Figure 23, ε is changed from –0.4 to -0.1 thus a less steep curve slope(corresponding curves are below respective images). We can observe that the suppression has been alleviated but the ring effect is still as prominent as before. One possible solution, though is not tried here, could be coming up with a new in-scattering formula, where the suppression part never goes down to below 1. That is, suppression is in relation to enhancement instead of an absolute value (1 here).

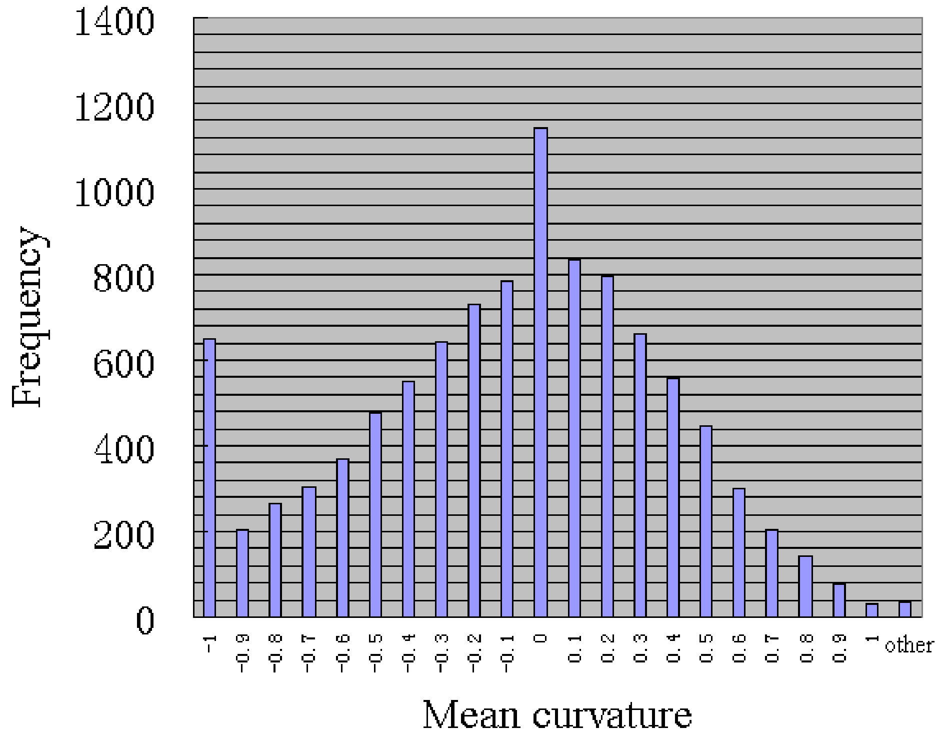
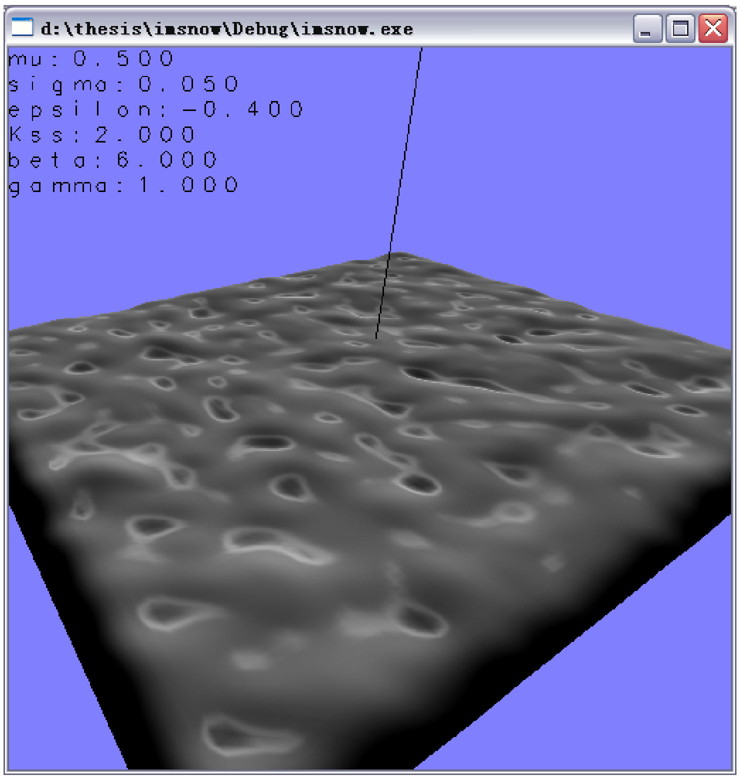
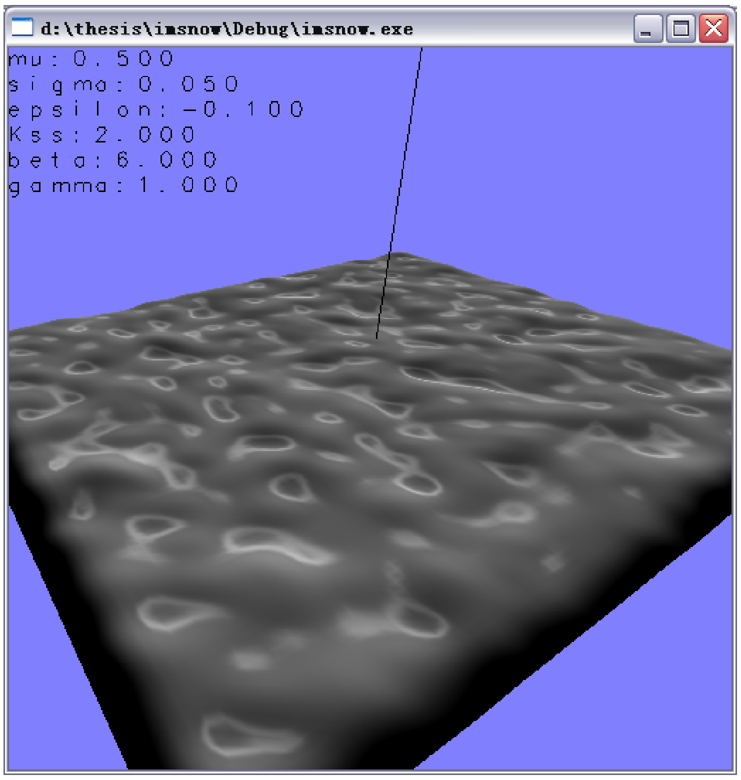


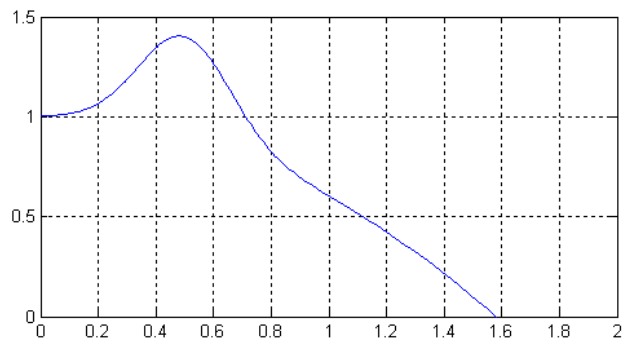
Figure 22: Histogram of vertex mean curvatures


d:\thesis\imsnow\Debug\imsnow.exe

```
mu: 0.500
sigma: 0.050
epsilon: -0.400
Kss: 2.000
beta: 6.000
gamma: 1.000
```


d:\thesis\imsnow\Debug\imsnow.exe

```
mu: 0.500
sigma: 0.050
epsilon: -0.100
Kss: 2.000
beta: 6.000
gamma: 1.000
```



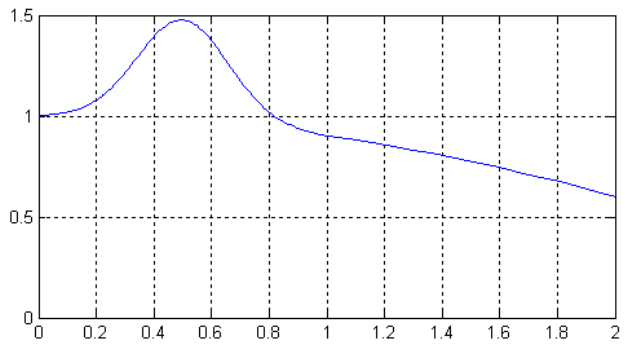


Figure 23: In-scattering effect results with different ε values and their corresponding curves

5. Conclusions

During this thesis, we have picked out several key visual characteristics of a snow surface exposed to a strong light source and implemented them with the OpenGL graphics library. The glittering effect is realized by pixel blooming which uses a multi-pass render-to-texture technique. As for the surface roughness which we accredited to micro shadows, we encountered difficulty due to the failure of finding a way to model this fine level of details. Consequently, we enlarged the granularity and came up with a model that, based on surface local convexity or concavity, can enhance (or suppress) surface light intensity locally and display local translucency because of subsurface scattering. This model is computed at a per-pixel level: if a pixel belongs to convex portion of the surface, subsurface scattering calculation will be applied to yield the translucent effect. Otherwise, in-scattering calculation will be used to adjust the original light intensity at a pixel. The per-pixel concavity or convexity information is acquired from interpolated vertex mean curvature which is computed before the rendering starts.

This thesis work has made the first step into the area of using a surface's local information to model the local illumination effects. As we can see, this approach does add a lot of surface details without too much cost. Therefore, I believe it has plenty of potential thus deserving further investigations. Given more time, it would be interesting to quest a more elaborated way of depicting the surface topology so that a better local effect can be acquired to more realistically simulate this challenging visual effect of fine details found on a snow surface. On the other hand, although this model is an empirical one based on our assumption about the interaction between light and fallen snow with some known common properties of snow crystals, it does add nice local illumination effects to a surface to make it appear to be covered by snow. Through this work, I actually feel that establishing an empirical model which shows a convincible result is a way to go if real-time rendering is the requirement because an empirical model, by making some assumptions, frees itself from a complex calculation featured by a physical model, which makes a per-pixel computation possible to be carried out in real-time. Albeit graphics hardware becomes more and more powerful, an empirical model will still have an advantage over its physical counterpart.

In the end, it has been a great excitement to work with a programmable graphics pipeline. Based on the traditional fixed pipeline, the new pipeline entitles graphics programmers to control how each vertex and fragment should behave and in order to do so a whole lot of pipeline's attributes have thus been exposed to the programmers, which simply brings in untold flexibility. With faster and faster graphics hardware, programmable pipelines are certainly paving the way to create more and more real-time per-pixel effects and make the virtual world closer and closer to the real one.

6. References

Blinn, J. F.(1978) Simulation of wrinkled surfaces. In Proceedings of the 5th annual conference on Computer graphics and interactive techniques, ACM Press, pp.286-292.

Bohren, C. F. and Barkstrom, B. R.(1974) Theory of the optical properties of snow. *Journal of Geophysical Research*, *79*, 30 (October), pp. 4527–4534.

Chrisman, C, L.(no date) Rendering Realistic Snow
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.125&rep=rep1&type=pdf [viewed 12/25/2009]

Fearing, P.(2000) Computer modelling of fallen snow. In Proceedings of the 27th annual conference on computer graphics and interactive techniques, pp.37-46.

Hao, X., Baby, T. and Varshney, A. (2003) Interactive subsurface scattering for translucent meshes. In Proceedings of the 2003 symposium on Interactive 3D graphics.

James, G. and O'Rorke, J. Chapter 21 Real-Time Glow.
in: Fernando, R.(Ed.)(2004) GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics. Addison-Wesley Professional.

Kaneko, T., Takahei, T., Inami, M., Kawakami, N., Yanagida, Y., Maeda, T. and Tachi, S.(2001) Detailed shape representation with parallax mapping. In Proceedings of the ICAT 2001 (The11th International Conference on Artificial Reality and Telexistence), pp.205–208.

Kreyszig, E. (1991) Differential Geometry. 1st Ed. Dover Publications. 0486667219

Max, N. L.(1988) Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer 4*, 2 (July 1988), pp.109–117.

Meyer, M., Desbrun, M., Schröder, P. and Barr A.H. (2002) Discrete Differential-Geometry Operators for Triangulated 2-Manifolds.
http://www.cs.caltech.edu/~mmeyer/Publications/diffGeomOps.pdf [viewed 11/13/2009]

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, pp.C357-C364.

Ohlsson, P. and Seipel, S. (2004) Real-time rendering of accumulated snow. In

SIGRAD 2004. The Annual SIGRAD Conference. Special Theme – Environmental Visualization, pp. 25–32

Policarpo, F., Oliveira, M. M. and Comba, J. L. D.(2005) Real-time relief mapping on arbitrary polygonal surfaces. ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games, pp. 155-162.

Sloan, P.-P. J. and Cohen, M. F.(2000) Interactive horizon mapping. In Proceedings of the Eurographics Workshop on Rendering Techniques 2000, Springer-Verlag, pp.281-286.

Xu, Y.-Q. , Chen, Y., Lin, S., Zhong, H., Wu, E., Guo B. and Shum, H.-Y.(2001) Photorealistic rendering of knitwear using the lumislice. In Proceedings of the 28th annual conference on Computer graphics and interactive techniques, pp.391-398.