CHALMERS

Procedural Generation of Natural Variations in Materials and Surfaces

JOHAN NILSSON

Master's Thesis
Information Engineering Programme
CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Division of Computer Engineering
Göteborg 2008

Abstract

One of the challenges when generating photorealistic images is to avoid a synthetic appearance of the material or surface. A critical step to overcome this obstacle is to introduce variations in the objects.

This thesis proposes how some of these variations can be introduced into some selected materials and surfaces with the use of procedural generation on the GPU. Numerous variations, chosen with rendering of cars as the main aim, in metallic paint, carpets, tempered glass and generic materials are presented along with suggested method to render them.

The graphical result is good and the artist is offered a good amount of control over the appearance.

Sammanfattning

En av utmaningarna med att generera fotorealistiska bilder är att få ett material att se levande ut. Man kan genom att introducera variationer i ett objekt få det att se mindre monotont ut och därmed hjälpa till att få ett objekt att se mer naturligt ut.

Denna tes presenterar hur vissa variationer kan introduceras i ett antal utvalda material och ytor. Utöver detta så kommer dessa variationer använda sig av procedurell generering, samt utföra alla beräkningar på GPU under rendering. De utvalda variationerna, som valts med rendering av bilar i åtanke, kan återfinnas i metallic lack, mattor, tempererat glas samt några generella material.

Det grafiska resultatet är bra och möjligheterna för användaren att anpassa utseendet är goda.

Acknowledgments

First and foremost I want to thank my supervisor at Spark Vision, Johnny Widerlund, for his guidance during the work of this thesis. Furthermore I want to extend my thanks to Spark Vision as a whole for letting me use their equipment and expertise.

I also want to thank Ulf Assarsson at Chalmers for establishing the contact between me and Spark Vision and for agreeing to be the examiner of this thesis.

Table of Contents

1 Introduction

Generation of photorealistic images continues to get better as new hardware, algorithms and techniques become available. Two of the main advantages with this type of image creation is the great amount of control the user can be offered and the elimination of prohibitive external factors that may ruin a photo taken in the real world. The problem, however, is to get the photos to look real.

Natural variations found within materials and surfaces are one of these factors that have to be taken into account when trying to create photo realistic images. Even though they might be hard to notice, the variations do contribute to the appearance of the object in question as a whole.

Spark Vision is a company that specializes in CGI (Computer Generated Imagery) of cars. These images should be photo realistic, which means that the same issue with natural variations is applicable here too. The materials and surfaces handled in this thesis have been chosen with rendering of cars in focus. This does not exclude, however, that some of these materials may be usable in other circumstances.

The reader should be familiar with real-time rendering in general and Simplex noise. Akenine-Möller and Haine's book *Real-Time Rendering* [1] is a very good source of information regarding real-time rendering. Simplex and Perlin noise is described very well by Stefan Gustavsson in *Simplex noise demystified* [2].

1.1 Problem Specification

Enhancing the realism in computer generated environments such as games, movies and images, is a process that has been going on for several years now. What was previously done by using physical models in movies can now be done entirely by computer generation. The problem though is to make these models look real, to make them look "alive" in a sense. One factor that that needs to be considered is the existence of natural variations. These variations can be introduced by e.g. letting the object have some texture and a bump map. Given that they are suited for the model in question they can greatly enhance the appearance of said object.

These textures, however, are often taken from external sources, such as photos. This procedure makes the artist dependent on that the sources are good to begin with. Even though the textures can be edited with external tools beforehand, the amount of work needed is still directly dependent on the quality of the original. Externally created textures also suffer from finite detail and tiling artefacts. Both of these deficiencies puts restrictions on how the artist can model and texture an object.

Objects created through procedural generation, in contrast, enjoy infinite detail and can avoid tiling artefacts. They can also offer the artist great amount of control by exposing several parameters.

Given that natural variations can make objects look more realistic, and the aim to give the artist a good amount of control, sets up the goal for this thesis. The thesis should present some suggestions on how to procedurally generate natural variations in certain materials. Furthermore, all calculations should be done on the GPU by the use of a high level shading language.

1.2 Limitations

One major aspect that is not covered in depth in this thesis is performance and therefore lacks measurements in render time for the different shaders. There is however some optimizations within some of the implementations and these will be discussed. The shaders have also been designed in such a way that the user can make trade-offs between render time and quality.

Even though variations in metallic paint are handled, metallic paint itself is not. A general method for rendering metallic paint will therefore not be presented in this thesis. The metallic paint shader used as a base during the work on this thesis was provided by Spark Vision. This also holds true for the carpet and the "general material" shader.

The simulations of natural variations in this thesis are in general not physically correct. There is some physical reasoning behind them, but in the end they try to approximate the visual appearance rather than a correct simulation of physical properties.

2 Background

2.1 Natural Variation

Take a look on a straw of grass. It may appear just to be some green object at first glance, but if it is observed closer an almost infinite amount of details emerge. There are visible veins, parts of it have holes in it and other areas may have a slightly yellow colour. Even if these variations are not noticed right away, they do contribute to the visual appearance of the object as a whole.

These occurrences are called natural variations and it is, as the name suggests, variations found in some object that occurs due to the nature of the material. It can be how things functions, like the veins in the grass, or wear and tear in the case of scratches on a CD or dust on a table. The things that make an object unique can be called variations.

2.2 Realistic images

There has been some research done on what makes a computer generated picture appear realistic or synthetic. Rademacher et al.[3] tried to find what components in a picture that makes humans believe that the contents in the image are real or computer generated. One of the main components that was found was that a rough surface on an object made the picture as a whole more prone to be interpreted as realistic.

The research done by Radermacher et al. was done by showing people different photos and then asking them if they thought the picture was computer generated or not. Among the aspects tested was roughness of objects in the picture. About 70% thought that the picture with the rough object was real in contrast to 38% for the smooth variant. Worthy of note is that the objects used, both the smooth and the rough, were all part of real photographs. The fact is that that even a real life setting can be interpreted as more or less real depending on the situation. When the test was done using only computer generated images, the impact of a rough surface was about the same. About 75% thought that the rough version was real compared to 25% for the smooth.

2.3 Procedural generation

Computer Graphics is all about visualising some type of content, such as models and textures. It is quite often the case that this content has already been created before the actual rendering. An example is when texturing some leather seat and a photo taken earlier of a leather material is used as a texture source. This is not the only way however, as content can be created during execution. The leather in this case can be calculated according to some mathematical expression that is evaluated during rendering. This type of content creation is called procedural generation.

The main advantages with procedural generation in computer graphics are infinite detail and flexibility. Procedural generation is just based on mathematical functions, and thus the rendering is not limited to some finite set of information. This means that it can get accurate information at any given point rather than resorting to interpolation between some existing finite data. The flexibility feature is also a direct consequence of these mathematical functions. These functions can namely be designed to allow a different number and variants of parameters. If done right, this can provide a very powerful way to change the appearance of the actual content.

3 Previous Work

3.1 Real-Time Visualization of Metallic Paint Glittering

Metallic flakes have previously been investigated in a Master's Thesis by Thomas Nilsson [4] at Spark Vision. This investigation first made tests with a two dimensional texture bump map, which got quite good results but with the drawback that the texture coordinates on the models was often quite bad. Therefore a three dimensional texture bump map was generated so that the model coordinates could be used instead of the texture coordinates. This yielded a much better result. It does, however, lack the ability to control the flux of the flake normals and how often sparkles should appear. Additionally, as it makes use of an external texture it is not making use of procedural generation.

3.2 Metallic paint by AMD/ATI

With the shader editor RenderMonkey, made by AMD/ATI, there is a shader sample provided for the metallic flake effect [5]. It handles both the slight variation caused by flakes and the strong glittering effect. The shader uses a pre-generated bump map with a seemingly uniform random distribution of values. It also provides means to alter the size and colour variation of the glitter and gives quite good result if not viewed to close up.

By using this method, a restriction is made on how close the surface can be viewed. If viewed to close, the flakes will be clearly visible as squares packed together, which is an undesired property. This can be countered though by setting the flake size to quite small. The squares will still be visible at really close range but the object should not be viewed at that range anyways. By using a two dimensional texture, it is dependent on well made texture coordinates. It also lacks control over how frequent the sparkles should be and the spread of flake normals.[5]

3.3 Efficient Acquisition and Realistic Rendering of Car Paint

This paper suggests a method for not just the rendering of flakes, but the metallic paint itself by measured BRDF. Just using measured BRDF is not enough when viewing the paint up close where the sparkle effect appears. So, the flakes are emulated by random perturbation of the surface normals.

One problem with this implementation has to do with flake size and viewing distance. When viewed up close, the highlighted flakes will risk looking too large. This can be avoided somewhat by setting a very small flake size. But this will, however, introduce the risk that the flakes will not appear at all if the viewing distances are not extremely close to the surface [6]. The model relies on multisampling to get rid of the aliasing effect, and it is also stated in paper that an interesting future work would be to have some multi-level model for the flakes to take care of the aliasing problem.

3.4 Cloth and carpets

The research that has gone into rendering and modelling of textiles are quite often about cloth rather than carpets. One difference between cloth and auto carpet is that the latter tends to be much more chaotic in its thread composition. While the cloth may be tied down to some quite rigid pattern, the auto carpet just has a couple of seams. The threads may also lean over those seams, entangle with neighbouring threads or stand out quite freely.

The different models for cloth are also quite focused on either very accurate rendering of cloth [7] by e.g. explicitly modelling the threads, which is very expensive. Other variants focuses on efficient models [8] that avoids to explicitly model the actual threads and instead concentrates on a good lighting model and repeating patterns for the cloth. The model in this thesis also incorporates repeating patterns but also different mapping techniques to emulate the chaotic threads to give the appearance of carpet.

4 Method

The shader language used in this thesis is CgFX with vp40 and fp40 as profiles. While both the vertex- and pixel shaders are used for these materials, the interesting code can be found within the pixel shader. The vertex shader only handles some space transformations.

The variations and materials that are handled here have been chosen from materials that are often found in a car. This has to do with the fact that the thesis has been done at Spark Vision, a company that creates photorealistic images for the automobile industry. By comparing photos of real cars with their computer generated counter parts, which can be found in e.g. commercials and games, a number of variations not covered well by the latter category were found. A number of these were then selected for further analysis based on how prominent they were. These selected variations can be found within this chapter.

4.1 Simplex noise

At an early stage it was realised that a large part of this thesis was going to be based on different ways to use noise. The first noise implementation in this thesis was that of standard Perlin noise but, because of a number of deficiencies, it was later changed to Improved Perlin noise [9] and shortly thereafter to Simplex noise.

Throughout this thesis the terms amplitude and frequency are used to describe the behaviour of noise. Amplitude determines the signal strength, and frequency decides how fast the noise values can fluctuate.

This section does not offer any description of Simplex noise. It is merely some notes about GPU implementations and how to analytically determine the noise gradient.

4.1.1 Notes on GPU implementation

There are two things that are important to know when doing a GPU implementation of Simplex noise: the lack of random generators and how to store gradient information.

The gradient information is just stored in a texture rather than a static array, so a texture must be provided as an external source to the shader. One way to implement a pseudo random number generator is described in the next section.

4.1.2 Pseudo random number generator (PRNG)

Even though a PRNG is needed for Simplex noise, it is not something that is supported natively as an intrinsic function in CgFX. Instead, this functionality has to be explicitly implemented into the shader. One way to do this is to use a texture to store a random permutation of some value [10].

If this permutation texture would be used directly, the texture would need to be very large in order to avoid repetitive patterns. But it is possible, by using another sampling technique, to use a texture as small as 256x1 with good results. Even smaller can be used, but then the risk is larger for repetitive patterns.

The sampling method used in this thesis doesn't just do one look up in the permutation texture for each sampling point; it does three, one for each axis. Each lookup only uses the value of one axis, rather than all the values at once. The result from the lookup is then used to offset the next input value. So, in CgFX syntax this would be:

```
#define TEXTURE_RES 256 // Resolution of the texture

float PRNG(float3 pos, float size, sampler2D perm_sampler)
{
    float permute_val = 0.0f;
    pos = pos /( size*TEXTURE_RES);
    for(int axis = 0; axis < 3; axis++)
        permute_val = tex2D(perm_sampler,float2(pos [axis]+ permute_val,0)).r;

    return permute_val;
}
```

Note that this method supports arbitrary dimensions depending on how the function is designed. This is a major advantage if one wants to use model space rather than texture coordinates as it doesn't need a three dimensional texture.

Also, note the scaling of the input position. Not only does it get divided by the size input but also by the texture resolution. That is because the texture size has very much to do with how frequent the changes in values are from the tex2D operation. With large textures comes frequent change in output as more data fits into the [0,1] range. By dividing the input by the resolution this behaviour is redeemed.

The sampling method described above has previously been used in e.g. Jönsson's [10] implementation. The only difference is the ability to resize the frequency of the output changes.

4.1.3 Noise gradient calculation

A noise function typically returns just one value, and that value represents the actual noise. There are times, however, when this information is not enough. One technique that puts additional requirements to the noise function is bump mapping. Bump mapping is not in need of the actual noise value, but rather the noise gradient. In Perlin noise a neighbourhood sampling method is often used, which is quite expensive as it needs to run through the noise function several times. This approach is fortunately not needed with Simplex noise as the gradient can easily be determined analytically.

When trying to determine the expression needed to calculate the gradient, it helps to look at the original expression for the noise calculation. For each node a specific interpolation expression is used that varies depending on the dimension. Throughout the thesis, the dimension used is three, so the calculation of the gradient would be like the one below.

$$n = 8 * (0.6 - x^2 - y^2 - z^2)^4 * \left(x * x_g - y * y_g - z * z_g\right)$$
$$t = (0.6 - x^2 - y^2 - z^2)$$
$$s = \left(x * x_g - y * y_g - z * z_g\right)$$
$$n = 8 * t^4 * s$$
$$n'_x = 8 * (-8 * t^3 * s + x_g * t^4)$$

So by simply calculating the resulting partial derivative for each axis in every neighbouring simplex node, the noise gradient can be obtained.

4.1.4 Fractional Brownian motion (fBm)

Fractional Brownian motion (fBm) is, in the world of Perlin and Simplex noise, all about self similarity. Self similarity means that different fractions of an object look similar to the object itself.

This property can be simulated by adding together several octaves of noise [11][12]. An octave is essentially a combination of frequency and amplitude that has some relation to the previous octave. The amplitude can for example be halved while the frequency is instead doubled for each consecutive octave. This means that if the starting octave has 2 in amplitude and 2 in frequency, the following octave would instead have 1 and 4 respectively. Throughout this thesis, the scaling of amplitude and frequencies between these octaves are referred to amplitude and frequency multipliers.

Lastly, the octave sum is divided by the amplitude sum in order to keep the result within the [-1,1] range.

4.1.5 Basic noise patterns

In this section, a number of basic and well known noise patterns are presented. The given names are not official ones but they help to distinguish the patterns from each other in this thesis. They all use fBm in some way to get their respective pattern.

Cloud

This pattern, with its very soft noise variations, bears similarities to how clouds look [11].

The pattern is achieved by using an amplitude multiplier of 0.5 and frequency multiplier of 2.

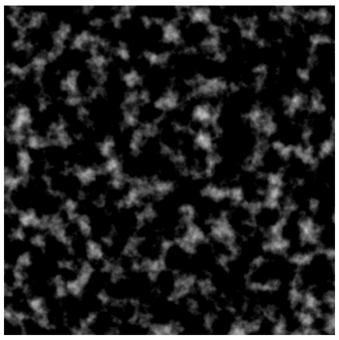


fig 4.1 Cloud pattern

Turbulence

A turbulence pattern [11] looks, like the name suggests, similar to turbulent flow found in e.g. smoke.

It is generated in a way that is similar to the cloud pattern. The difference lays in the summation of the octaves, where the Turbulence pattern uses absolute values rather than the usual signed ones. This means that no negative values will be generated, which is why there is a positive RGB value everywhere in *fig 4.2*. The areas where the values are close to zero can be observed as the veins that run through the picture.

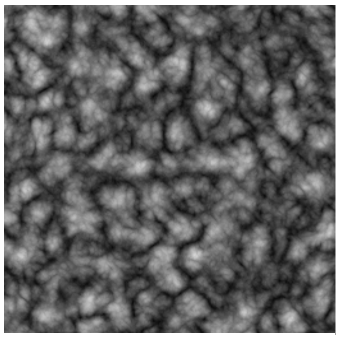


fig 4.2 Turbulence pattern

Plasma

This pattern looks like plasma or an electrical discharge.[11]

The plasma pattern is generated by subtracting the Turbulence pattern from 1.

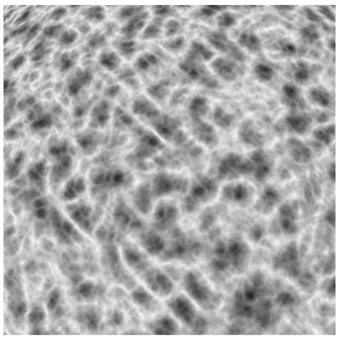


fig 4.3 Plasma pattern

4.1.6 Anti-aliasing of noise

Tatarchuck [13] described a way to implement anti-aliasing for noise in a way called frequency clamping. The idea is to find out how much the position used for sampling changes between pixels and then fadeout the signal if the change in position is too large. This thesis makes use of the very same implementation for anti-aliasing, which works quite well. A description of the implementation is given here as a summary.

A function is used as a kind of filter where the signal is provided as input along with some other necessary data. The output is then a faded variant of the input where the signal has become weakened to avoid aliasing.

```
float3  fadeout(float signal, float average_signal,
            float item_size, float pos_per_pix)
{
    float inter_point = smoothstep(0.2,0.6, pos_per_pix /item_size)

    return lerp(signal,average_signal,inter_point);
}
```

The signal parameter is the value of the actual noise while the *average_signal* is the average value of the signal type. The average signal value is 0 for Simplex noise if the range [-1,1] is used. The *item_size* is the physical size of the item that holds the noise value. In Simplex noise this would be the inverse of the frequency. The last parameter, *pos_per_pix*, is the ratio of how many position units can fit within one pixel. This only needs to be calculated once in the shader code rather than one for each fadeout call.

The ratio needed for the last parameter can be calculated by using the ddx and ddy methods in CgFX. These two methods can calculate what the difference would be for the given argument if the sampling pixel was moved one step along the x and y axis respectively.

So, if some position of the object is given as argument, the functions would calculate how much that position is changed in one pixel step. In practice this determines how wide a pixel is measured in position units, which is exactly what is needed as the last argument. The problem is that not only does ddx and ddy each output a value with the same dimension as the input, they also return two separate value vectors while only one value is needed for the last argument. Given that the aim is to anti-alias, the highest value should be used as it represents the largest possible change for any axis.

When using fBm, the input value of *item_size* needs to be scaled for each iteration step. It should be scaled by the inverse of the frequency to the power of the current iteration step. This makes the *item_size* adapt to the different sizes of the octaves in each step in fBm.

There is one more thing to keep in mind and that concerns effects that should only make a visual contribution at close range like sparkles in metallic paint. One of the parameters should be the mean value. So in the case of sparkles, where it is simply active or inactive (1 or 0), it might be tempting to set this value to 0.5. This is, however, wrong. It should be 0 as these effects are supposed to disappear entirely after some distance. If 0.5 is set it will fade towards 0.5, but never get to a smaller value than that.

4.2 Auto carpet

4.2.1 Material description

In the interior of a car there are several surfaces that are expected to withstand quite an amount of wear and tear. Additionally, these surfaces should often offer a good amount of friction in order to offer adequate grip. Examples of such surfaces are the floor of the trunk and in front of the seats. These surfaces can be made purely out of plastics but they can also make use of auto carpets. In the latter case, the construction is somewhat similar to those found on towels and doormats. This means that it is not a knitted carpet but rather several small threads extending from the base surface.

The auto carpet also displays quite dramatic colour fluctuations on a surface when the threads are leaning in different directions. This can be caused by simply dragging an object across the surface as the threads then will lean in the direction of the force that was applied.

The carpet is also based on regularly appearing seams that are often arranged as lines across the surface. These seams may not be considered natural variations, but they are certainly an important characteristic in the carpet. Furthermore, some threads may lean over the seams at certain places, which create variations on the surface. So the seams themselves are important to consider in order enable the generation of some natural variations.

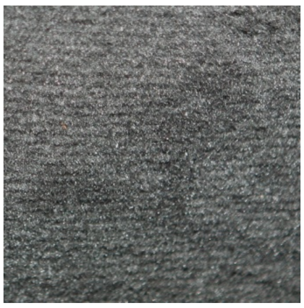


fig 4.4 Auto-carpet

Carpet, cloth and fabric in general are very interesting materials that are highly affected by natural variations. It's the complex formations of threads and how the light interacts with them that give the cloth its soft and diffuse appearance.

4.2.2 Rendering the threads

A carpet, as mentioned earlier, is built up by threads. These threads give the carpet its unique appearance while offering abundance in natural variations. This makes the rendering of threads paramount for the visual appearance of the carpet as a whole. It is extremely computational taxing to explicitly create every single thread and calculate the correct light interaction between them so this is not an option in this implementation. Just using a simple lighting model is not suitable either as the actual variations will be lost.

When they needed to render carpets previously at Spark Vision, they had done it mainly with a combination of diffuse and specular maps. By doing this, they avoided modelling the threads explicitly while still giving the impression of a surface filled with threads. Inspired by this, the thesis makes use of procedural generated diffuse and specular maps to represent the threads.

The way to use these maps in this implementation is to let them have very separate, but equally important, tasks. A carpet is, as mentioned earlier, a very rough object with plenty of light scatterings and diffuse reflections. Single threads are seldom seen on an auto carpet; it is instead the collective of threads that can be clearly seen. These collectives, however, can vary in shape and colour depending on how many threads that are in them, how tightly packed they are and so on. These variations can be distinguished even at range. The task the diffuse map has is to represent these collective threads.

There are, however, some threads that can be seen individually among the other threads quite clearly. The aim with the specular map in this case is to simulate these threads. It can be said that the diffuse map is used for the collective, and the specular map for the individual threads.

Diffuse map

Even though the diffuse map should concentrate on the collective of threads rather than individual ones, it should still emulate the rough surface caused by many threads. The natural thing is to turn to fBm for generation of this type of diffuse map, but the roughness issue needs to be kept in mind when determining on how to use said technique.

When looking at the regular fBm, with amplitude and frequency multiplier of 0.5 and 2 respectively, it might be tempting to use that directly. It has nice variations in noise values, and avoids too strong concentrations of high noise values given low enough frequency. It is, however, not rough enough. With an amplitude multiplier of 0.5, the amplitude for each octave is halved. This means that the contribution for each consecutive octave is halved while the earliest ones remain dominant. What is needed is more fluctuation in the noise.

This can be done by setting the amplitude multiplier to 1. With this setting, the amplitude is neither scaled up nor down for each step. This means that the value from each octave is valued just as high as any other one. This creates a rough surface as there is a larger chance that peaks may form. The number of octaves does, however, have an adverse effect after about three steps. After that, the averaging over the values is beginning to take its toll, and the values will smooth out.

The fact that the fBm output values ranges from -1 to 1 while the diffuse map should not contain any large black areas needs to be tended to. It may be tempting to simply rescale the value to [0, 1] as it will eliminate the aforementioned completely black areas. This will, however, not only halve the noise range but also the difference between noise signals. This leads to a softened signal, which is not desirable in this case. Using the absolute value of the octaves would also bring it into the desired range. It would, however, also get 0 values formed as clear veins, like in *fig 4.2*, throughout the diffuse map.

To preserve the range, while at the same time get rid of the black areas, a change can be done in how this diffuse map is used. Normally, the diffuse value is multiplied with the diffuse map to get the final value. This behaviour is kept, but the value is instead added to the diffuse value rather than replacing it. These yields diffuse values in the range of $[diffuse - strength, diffuse + strength]$. The $strength$ variable is a multiplier for the diffuse map so that the magnitude of the alteration can be controlled. This setup will, however, introduce a risk of getting diffuse value outside the colour range. The diffuse value and the diffuse map strength should thus be chosen with care.

This provides a quite nice, rough, surface that works well for the collectives of threads viewed from a distance. The rough surface also provides a sense of threads at close range as there are several peaks and ridges within the noise. Its Achilles heel is that it at loses the illusion of threads at really close viewing distance. This is quite natural as individual threads have not been generated.

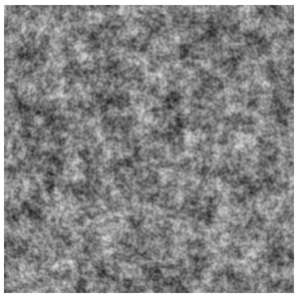


fig 4.5 Diffuse thread noise

To counter this, several attempts were made to introduce plasma patterns in different ways. Used alone the pattern lacked the fluctuations needed for the diffuse map, and when mixed with the usual thread it didn't have a significantly good impact. This was abandoned as it is only an issue at really close range, but it might be a good idea to look into some way to model the threads so that it provides a better representation at micro level.

Given that the noise gradient in Simplex noise can be calculated analytically, it becomes quite cheap to apply bump mapping. This is especially true if the noise signal is going to be used anyway, like in this case with the diffuse map noise. Thus, bump mapping can be applied with the values received from the generation of the diffuse map to give a more uneven feeling to the surface. It should, however, preferably be done with just a small amount. If too much is used it will look rocky.

Specular map

The focus of the specular map is to simulate individual threads. This means that the noise function that was used for the diffuse map is not suitable here as the two maps have different aims.

A natural place to start is with the quite well known plasma pattern, which can be seen at *fig 4.6* and described in *4.1.5 Basic noise patterns*. While it does resemble plasma, it also looks a bit like entangled threads.

Just like the noise for the diffuse map, it might be tempting to use the regular 0.5 amplitude multiplier. Looking at *fig 4.6* this might seem like a good idea as it has several threads entangled. The major problem is that it doesn't fluctuate much in signal strength so the actual veins are not that distinct. This creates something like a background noise where the noise is quite smooth and the veins are quite blurry. Applying a filter that sorts out the veins and background noise is hard with this noise pattern as the small threads run a high risk to disappear along with the blurred noise.

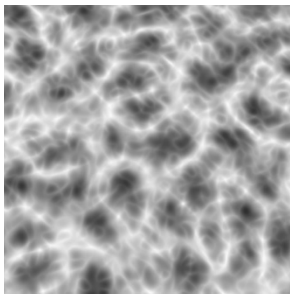


fig 4.6 Plasma noise

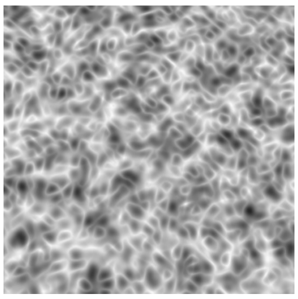


fig 4.7 Noise for threads

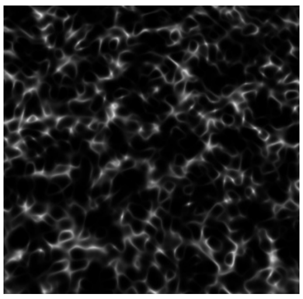


fig 4.8 Noise with filter applied

By setting the amplitude multiplier to 1, it becomes easier to filter the noise at the veins become more distinct. There is still a need though to filter the noise so that the background noise is eliminated. An easy way to do this is to use a simple power function. The power used in this implantation is 7, which means that all noise values are evaluated by a 7 as exponent. This will nearly eliminate the small values and only keep the larger ones relatively close to their original values.

4.2.3 Seams

Carpets found in cars often have a certain regular seam pattern, like lines or a grid. This is not a natural variation as such but is nevertheless important to consider in the carpet shader. Besides, there are natural variations that depend on these seams such as threads that lean over the seams.

The idea is to generate these seams by altering the diffuse and specular values and also apply bump mapping. Threads are represented in the diffuse and specular map, so it is natural to alter the diffuse and specular values if the amount of threads should be changed. By scaling these values with a value that fluctuates according to some seam pattern, the amount of threads will seem to vary periodically. The bump map can also help to make the seams more prominent by varying the surface normal.

To generate the seam patterns, the texture coordinates needs to be used rather than model space coordinates. This is provided by the texture coordinates to the shader. However, be aware that this will make the shader dependent on correctly generated texture coordinates

Given that one has access to the surface coordinates, there is just one simple expression that needs to be evaluated to get a pattern with parallel lines. The expression should have the property that the output ranges from 0 to 1, where 1 is the largest amount of threads and 0 the lowest. This value can then be multiplied with the specular and diffuse values to simulate varying amount of threads. A simple way would be to simply use linear interpolation between zero and one as a function of surface position as the example below shows.

```
float p = position.x/frequency;
float amount = abs(frac(2*p));
```

The *x* position is used here in this example, which means that the lines will go along the y-axis. This is just a simple example, any axis can be used to determine how the lines by altering the calculations a bit. Just replace *position.x* with a position that has been projected down to an axis that is perpendicular to the axis the lines should align with. By dividing the position with a frequency value, the user can control how frequent the change between seam and fabric is.

This simple calculation will give the *amount* variable a value in the range of [0,1] in intervals of 0.5 on *p*, or simply put the doubled distance that *p* has to the closest 0.5 multiple. If high frequencies are going to be used with little to no bump mapping, this will give a good enough result.

The calculation is simple, but the result will not look right; the slope is too small and gives a too smooth transition between seam and fabric. The fabric clusters, which are tightly packed, have more of a round shape than a prism. A more suitable shape can be found in the sinus curve since it has a more round shape and is simple to use.

The sine function should not be used directly as it will create too large seams. Natively, the sine curve yields negative values for half its period, and due to the fact that negative values will be interpreted as seams, the seams will become much larger than they should be. This is easily solved by the use of the absolute value of the sine function. That approach turns the negative values to positive, and thus seams to fabric.

```
float p = position.x/frequency;
float amount = abs(sin(p));
```

There is one additional desired parameter, and that is the amplitude, or rather the amplitude difference between peak and valley. With that parameter it becomes possible to control how prominent the seams should be. The current implementation will create very deep seams as it will yield values in the full [0,1] range. Note that the seam value should always range to 1 in order to not tone down the diffuse and specular values in general.

The amplitude difference is based on how far from the value 1 the seam value can get. To get this behaviour, and extra amount of control, the *amount* calculation is extended a bit:

```
float p = position.x/frequency;
float amount = 1-amplitude*( 1-abs( sin(p) ) );
```

With this it becomes possible to control the difference in amplitude of the seams. The problem is that threads that lean over the seams have not been accounted for, so the carpet still looks quite artificial. This raises the need for some way to fill some seams with threads, so that it looks like the threads are crossing over the gaps. Simplex noise is very suitable for this task as it can create coherent areas for where the threads cross.

One could choose fBm, but just usual Simplex noise is good enough. It is not that noticeable that fBm is applied or not. This can be explained by the fact that noise is already used to create the threads in the diffuse and specular map. So even if just the usual Simplex noise is used here, the noise in the diffuse and specular map will hide the monotone appearance of the crossover noise.

The Simplex noise default output range [-1,1] is not suitable for this task though because of the negative values. It should only be able to fill seams, not make them deeper. A simple way to solve the problem is to simply scale the noise values to the range of [0,1]. After that, the noise value is multiplied with the difference between the maximum, which is 1, and the current seam value. This will ensure that the seam value will not go over its capacity while at the same time enable it to be reach up to the said limit. Furthermore, a parameter should be provided to determine how much the disturbance should affect the amount of threads. Put into actual code it would look like this:

```
float disturb = (noise(p)+1)*0.5 * disturb_amount;
amount += (1-amount)*disturb;
```

The appearance of an uneven surface can be further enhanced by the use of bump mapping. In order to do this, a perturbation normal is needed, which can be calculated analytically.

One problem though is the use of absolute value within the function that calculates the periodic seam pattern. Absolute values are not differentiable so the seam function as whole isn't either. But the absolute value here is only used to make the negative sine signals turn positive. This can be achieved also by just assuring that that all the input points are within 0 and π. By using this approach, the functions derivate can be calculated without the absolute value. The function derivative then becomes:

```
float derivative = amplitude*cos(p-floor(p/PI)*PI );
float2 partial_deriv = derivative*align_vector;
```

The *align_vector* in this case where *p* is just *position.x* is [1,0]. Note that this derivative is in two, rather than three, dimensions. This is a direct consequence of using a two dimensional coordinate system. These two values in the derivative can be seen as the change in height when moving along the surface. The problem is that these values are not in model space, so there is a need for some transformation. In model space, the axis on the surface is the tangent and bitangent (also called binormal by some). By simply multiplying the tangent and bitangent with their respective partial derivative a transformation can be made.

$$N_{seam\_perturb} = Tangent * \mathrm{derivative}_u + \ Bitangent * \mathrm{derivative}_\mathrm{v}$$

That deals with the regular pattern gradient, but the final normal needs to accommodate for the leaning threads. Even though the analytical gradient is available for the Simplex noise, it doesn't need to be used in this case. What the leaning threads is used for is to fill the seams. This can be accomplished by using a simple linear interpolation between the seam normal and zero with the disturbance value as interpolation point. When the disturbance value increases, the seam normal perturbation is approaches [0,0,0]. This means in practice that the normal perturbation gets less influential with a higher disturbance value.

Below is an example of how the amount function as a whole can be implemented.

```
float3 amount_calc(float3 alignment, float2 pos,
            float amplitude, float frequency)
{
    float amount = 1.0f;
    float2 derivative = 0.0f;
    float length=0;
    pos *= frequency;
    alignment.xy  = normalize(alignment.xy);
    // Lines
    if(alignment.z == 1.0f || alignment.z ==2.0f)
    {
        length = dot(alignment.xy, pos);
        amount = 1-amplitude *(1-abs(sin(length));
    }
    // Grid
    if(alignment.z == 2.0f)
    {
        float2 p_pos = float2(pos.y,-pos.x);
        float p_length = dot(alignment.xy,p_pos);
        float alt_amount =1- amplitude *(1-abs(sin(p_length));
        if(amount>alt_amount)
        {
            amount=alt_amount;
            pos = p_pos:
            length = p_length;
        }
    }
    derivative = amplitude*cos(length -floor(length /PI)*PI );
    derivative = derivative*alignment;

    return float3(derivative,amount);
}
```

The pictures in *fig 4.9* show how the different seam patterns affect a carpet. All, except the one without seams, uses seam amplitude of 0.2.

fig 4.9 Different seam patterns applied, top left lacks seams.

It is also possible to get a carpet surface that looks like it has fuzz on it by the use of a grid seam pattern and seam bumps. It also helps to use the bump map generated by the diffuse threads as it gives the carpet a more uneven appearance. The result can be seen in *fig 4.10*.

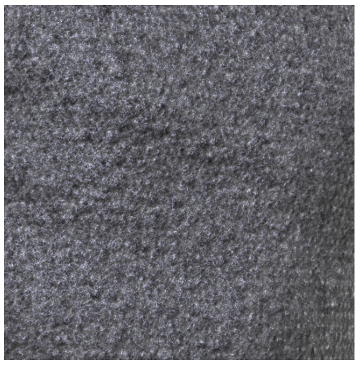


fig 4.10 Fuzz on a carpet

4.2.4 Dark areas

The threads in a carpet are not rigid and can therefore bend and entangle with each other quite freely. This has an impact on the appearance of the carpet as different thread orientations can give different colour tones, depending on how the light is reflected. As a result of this, arbitrarily large areas may look quite a bit darker than its surroundings when the threads in these areas lean in roughly the same way. To simulate this, the carpet colour can be altered depending on the output from Simplex noise. The output from the noise function can simply be multiplied with the specular and diffuse values to get colour variations. This requires that the noise output is within [0,1] range, which means there are some alterations needed.

The dark areas should be quite discrete, or there should at least be quite a dramatic change in colour between these dark areas and the surroundings. These means a rescaling of the noise output range to [0,1] should not be done. Instead of rescaling, the noise output can be clamped so that no values below zero are possible. This makes the fluctuations more dramatic than with a rescaling approach and offers the correct output range.

It is not necessary to use fBm for this task with the same reason it didn't need to be used for the thread crossover noise in *4.2.3 Seams*; there is simply already plenty of noise used in the threads. So, even if the diffuse colour is scaled in some large area with the same factor it, will still appear non-monotonous.

4.2.5 Roughness

A carpet is a very rough material; the surface has very frequent variations and it is hard to find any part of a carpet that is blank. This does, in fact, have an impact on how the light is reflected that is not covered in the standard Lambertian model.

The Lambertian model assumes that the diffuse reflections is all about randomly reflecting incoming light rays out from the surface. However, in a very rough surface there are lots of reflections between the bumps and backscattering taking place. This means that, for a very rough surface, the light reflected toward the observer is almost the same from any point of the object. An object with the aforementioned properties will therefore look quite flat. Carpets can certainly be considered rough, so this effect needs to be accounted for in the auto carpet model. There are lighting models that captures this behaviour, and one of them is Oren-Nayar [14].

Oren-Nayar is a generalization of the Lambartian diffuse model. This lighting model provides a parameter that can control the roughness of the surface. The higher this parameter is set, the more evenly will the diffuse light be spread in the diffuse lobe.

For more details about Oren-Nayar, read the paper written by Oren and Nayar [14], or the GamaSutra article [15] for a great explanation and implementation of this lighting model.

4.2.6 Fake rim light

One important aspect of cloth and carpet is the fact that light is reflected between the threads. This gives areas light that wouldn't be receiving it at all in other circumstances. One of the most noticeable effects of this is the lighting across the rim of an object when a light source is positioned behind it.

This rim light effect makes the object look softer, which is a good attribute when dealing with carpets. Real rim light can, of course, be used but this thesis uses a fake rim light. This means that an existing light source is not used. Instead, light is just added along the rim regardless if there actually is any light behind the object or not.

The idea is to provide a gradually changing lighting on the object where it is strongest in the rim and decreases towards the centre. One simple way to do this is to calculate the dot product between the view vector and the surface normal. That value will go towards zero the closer the sampling point gets to the rim. By taking that product and decrementing it from 1, the desired behaviour will be reached. The result can be seen in *fig 4.11* where the result has been rendered directly to a sphere.

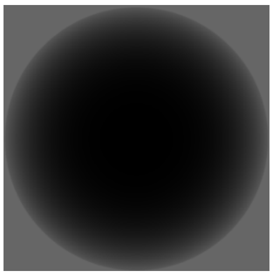


fig 4.11: Rimlight on a sphere

It is desirable to be able to control how wide this rim light should be. One way to do it is by using the *smoothstep* function provided by CgFX. This function offers a Hermite interpolation between 0 and 1 and a parameter for the actual interpolation point. The function also offers means to set the minimum and maximum range of the interpolation point. If the interpolation point would be lower or higher than that range, the output would become 0 and 1 respectively.

The raw rim light value is used as the actual interpolation point. Also, by using $1 - width$ as the minimum value, the actual rim light width can be controlled. By setting a smaller width, the rim light value will decrease more rapidly and also reach 0 earlier than before.

This implementation that follows has been directly taken from Kesson's webpage [16], no enhancements has been made on that implementation in the making of this thesis.

```
float calc_rim_light(float3 normal, float3 view_vector, float rim_light_width)
{
    float rim_light = (1-abs(dot(normal, view_vector)));
    rim_light = smoothstep(1.0 - rim_light_width, 1.0, rim_light);
   return rim_light;
 }
```

One important thing to be aware of is that the original surface normal should be used rather than a normal that has been altered through bump mapping. The reason is that the bumps will otherwise receive rim light regardless of where they are located on the surface.

These picture show the effect of the rim light applied to a sphere using the auto carpet shader.

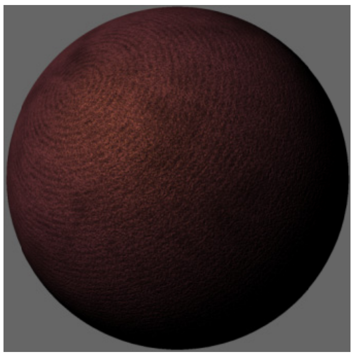


fig 4.13: Rimlight disabled

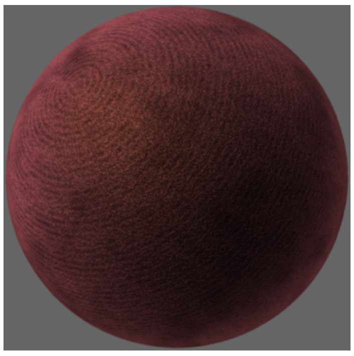


fig 4.12: Rimlight enabled

4.3 Metallic paint

This section will not give a suggestion for a general metallic paint model; it is focused entirely on the variations found within metallic paint.

4.3.1 Description of the material

Metallic paint has some important characteristics that makes it visual appealing and usable on cars. To understand how these properties came to be one need to understand roughly how the paint is composed.

Metallic paint can be seen as a composition of layers: clearcoat, basecoat, primer, electrocoat and lastly the metal body itself. The clearcoat layer is situated at the very top of the paint, where it provides protection for the other layers and gives a glossy appearance to the paint. Below the clearcoat resides the base coat, which adds colour to the paint. The primer is used to even out the surface on the metal body, which gives a surface with fewer bumps. Just above the actual metal body is the electrocoat which protects against corrosion.

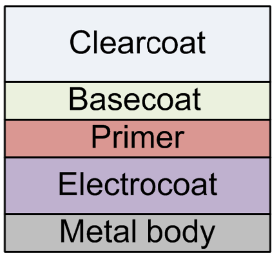


fig 4.14: Schematic view over the paint layers

The colour base coat in metallic paint is what makes it really special. Not only does it have colour pigments in that layer but also randomly scattered metallic flakes. This makes the colour vary a bit in appearance, and introduces the characteristic glittering effect.

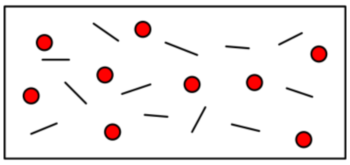


fig 4.15: Pigments and flakes in the colour base coat

4.3.2 Orange Peel

When applying metallic paint to a surface it is often desirable to get a smooth paint surface for aesthetic appeal. The resulting surface may however get slightly bumpy after the paint has dried. The surface texture bears a resemblance to that of an orange, so the defect is called orange peel.

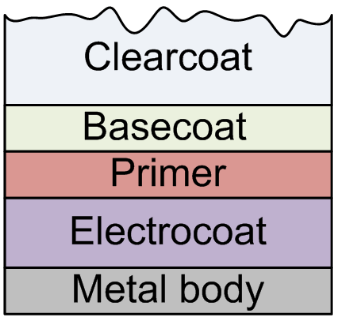


fig 4.16: Notice the bumps on the clearcoat layer

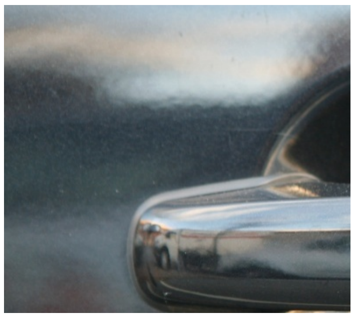


fig 4.17: Orange peel

There are a number of effects of orange peel that is quite prominent. Among these effects are:

- Ridges and valleys are visible up close.
- It affects the reflection on the surface so that it gets more of a diffuse characteristic.
- Self shadowing between the bumps is negligible, and the height difference cannot be registered by the human eye in grazing angles.

Based on these observations, it is quite clear that a simple bump map would fit nicely.

It is important to note how the texture of orange peel looks like. Like the name suggests, the texture bares a similarity to the rough surface of the peels of an orange. This appearance appears quite naturally by just simply taking the output from a Simplex noise function with medium to small frequency. The fBm function does not need to be used because of this and the orange peel should also just perturb the normal quite little so there is no real need to soften the signal with fBm.

Even though orange peel in paint will give it a rougher appearance due to the scattered reflection, it is important to note that the reflection also should be notably distorted. This means that the bumps should be so large that separate valleys and peaks may be distinguishable but also small enough to appear natural.

The main problem with this effect is to set the amplitude low enough so that the surface doesn't look too rough; it should be just enough to distort the reflections but low enough to avoid a rocky appearance.

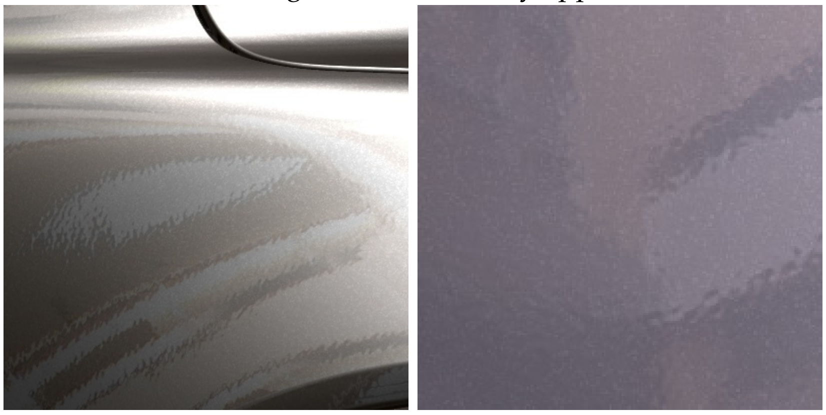


fig 4.18 Resulting orange peel effect

4.3.3 Flakes

As mentioned earlier in this thesis, there is a component called flakes within metallic paints. These flakes reside within the same layer as the colour pigments and are essentially tiny metallic mirrors.

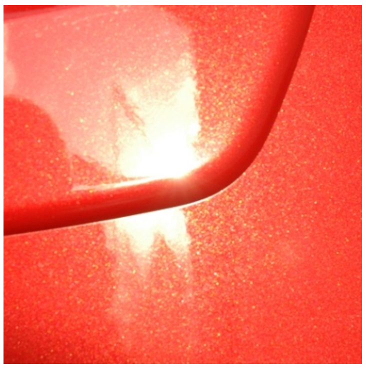


fig 4.19 Metallic flakes

Rather than being perfectly aligned with the surface, the metallic flakes have a slightly random orientation. By arranging the flakes in this way, incident light that hits the flakes will not necessarily be reflected perpendicular to the surface of the coat. Furthermore, due to the fact that flakes are like mirrors, there isn't much diffuse reflection from them but rather a pure specular reflection. This, combined with the small size of the flakes, makes it practically impossible to observe a single flake except when looking at one from an angle within the specular lobe. When viewed in this way, the flake will be seen as a sparkle.

Even though single flakes are seldom distinguishable, the collective do have a combined impact on the appearance of the paint as a whole. It gives variations to the paint colour as different areas reflect light in a different way. The very first implementation done within the scope of this thesis was to simulate the flakes and the sparkles in one common model. Given that the main problems with the RenderMonkey implementation [5] was its dependency on texture coordinates and Günther's et al.[6] with viewing distance, the focus in this thesis was first set to see if flakes could be generated with fBm.

One nice property with fBm, when used for flakes, is its self similarity property. This feature provides level of detail, which means that one could theoretically get flake contribution far away while still providing good detail at close range. It is furthermore not dependent of texture coordinates.

Both of these properties were the deciding factor when this model for flake simulation was chosen for further investigation.

A number of issues became clear very early. One of these was that the flakes should not be able to affect the reflections from the environment map. By having a complete flake cover and letting it affect the reflections, the surface will end up looking diffuse. The frequency can be lowered to make the surface less diffuse, but that will make the surface look like it has orange peel rather than flakes. Therefore, the altered normals should only be used during light calculations and omitted during environment lookup.

Another problem was to get a good high frequency effect while still providing contribution at a quite large viewing distance. The smooth noise value transition, which is one of the main properties in Simplex noise, becomes prohibitive in this case as it is easy to wind up with a rocky surface. This can be countered by using a higher frequency, but then the flake will filter out very early. Tests were done where the noise was generated with quite high frequency and low amplitude and the peaks were enhanced with an artificial specular colour boost. The hope was that by doing this, the rocky texture would be less prominent with the lower amplitude while still offering strong, high frequency signals. It was hard however to get single, strong points this way. Often an entire ridge would get a colour boost making relatively large, coherent areas light up.

Another variant that was more successful was to cut off noise values below some threshold. Just not boosting the peaks as before, but actually keeping just that noise. This got rid of the rocky appearance while at the same time providing quite small flakes. Ridges could still be seen but they were smaller now. The problem now was that flakes that made the surface reflect less light had a significantly more adverse effect then before. Previously they didn’t stand out as much as they were surrounded by other noise, but now these “negative” flakes could be seen clearly as dark spots.

To get rid of these dark spots, some kind of filter needs to be applied. Before an implementation attempt was made, a major shift occurred in the development of the flake effect. Even though the implementation with a cut off on the noise was crippled, the method did show promise as it gave a much better result than before. However, by applying noise cut off and filtering, the flakes that vary the colour, but not sparkling strongly, will be lost. The choice was thus made to separate the sparkle effect from the actual flake effect; one part simulates the sparkles while the other concentrated on the colour variations caused by the flakes.

This approach is not physically correct due to the fact that flakes and the sparkles are very much dependent on each other, but it gives the artist more freedom to alter the shader. The different ways to do glitter are handled in *4.3.4 Sparkling flakes*. Flakes in general will be handled in *4.3.5 Passive flakes*.

4.3.4 Sparkling flakes

This section handles the strong sparkling effect caused by flakes that are viewed within the specular lobe at close range. The sparkling effect is highly sensitive to viewing direction and gives small points a strong signal compared to its surroundings.

As stated in *4.3.3 Flakes*, the first implementation was with Simplex noise, but that version was omitted for various reasons. A part of the implementation that followed, more precisely the sparkling effect, will be discussed here. Three different types were considered in this implementation; among them was the old Simplex noise with a cut off.

The Simplex noise version was, however, omitted very early as it isn't suitable for generating isolated, high frequency, discrete points. The other two will be described below with their respective advantages and disadvantages along with a motivation for the final choice between them. After that a number of techniques needed for the development of the sparkling effect are described.

Flake grid using texture coordinates
The glitter can be generated as circles or squares using texture coordinates. This can be done in a grid that draws the shape at each distance multiple of x and y. The PRNG could then be used to generate the flake normals. It is quite straightforward and easily implemented. It does, however, suffer from one major drawback: its dependency on texture coordinates.

The reason for this dependency on texture coordinates is that one way or another one needs to generate squares, circles or other geometrical figures on the surface to represent the flakes. In order to do this, a two dimensional coordinate system, representing the surface, is needed. The model coordinates are in three dimensions and represents the entire model, while the texture coordinates represent the surface and resides in two dimensions. Therefore, the texture coordinate should be used for the aforementioned task. This dependency is a drawback because it is not unusual that three dimensional models tend to have stretched, or generally faulty, texture coordinates in some vertexes, which has a serious impact on the visual appearance.

It might be tempting to simply generate cubes or spheres and use the model coordinates instead, but this will not at all guarantee that every surface gets flakes of about the same size and distribution or any at all. The limitations with texture coordinates ruled out this one pretty early.

Direct use of the PRNG
The PRNG used for the Simplex noise takes, as stated earlier, three dimensional coordinates and outputs a pseudo random value based on the input values. Given that the PRNG has already been implemented earlier, the flake generation is simply a matter of one function call.

One major advantage of this method is that it can use model coordinates rather than texture coordinates. The flakes will, however, vary in shape depending on where they are located on the surface. This means that it will not yield forms independent on the shape of the model. It will however provide sufficiently similar looking flakes for the intended flake size. This way of creating the flakes also has the drawback that the flakes need to be very small indeed to avoid getting to large flakes at close range. Additionally, it also lacks native support for level of detail. There are means to get around these issues, which will be discussed later.

The method of using PRNG directly was the method among these three that was found to be the most beneficent for this application.

Flake normal variation

One important factor for the sparkles in metallic paint is how much the orientation tends to vary. A glittering flake is at its core just a perturbation of the surface normal, so it is simply a matter of how much the surface normal should be altered. The method proposed here requires that one has previously got hold of one value representing the flake. In this implementation, this value would be the one received from the PRNG function used for generation the actual flakes.

Given that there exists some value x representing the flake, three new random values can be calculated by using that value x as a seed for the PRNG. Now, using the same seed three times would get three identical outputs so some alterations are needed.

One variant is to use some offset for each PRNG call to get different values. Another approach is to change the input value to the output value from the previous PRNG call. Yet another variant is to use a larger texture for the PRNG that has three rows instead of one, and make it possible to select which row to use in the PRNG.

One potential problem with all of these approaches is that the number of combinations is quite limited. This didn’t introduce any noticeable problem in the implementation done in this thesis, but if more variation is needed a dependency on position can be introduced. It was never tested but it should work well to use the position of the centre of the flake as input instead of the other seed variants. How it can be retrieved is described in *Elimination of glitter flakes* as a way to uniquely identify a flake.

After the three values have been generated, the actual flake normal can be created. A parameter is introduced that decides how much the flake normal may affect the surface normal. The flake normal is merged with the surface normal later on by a simple addition followed by normalization, so if the flake normal is set to 0,0,0 it will not affect the surface normal at all.

```
float3  flake_perturb(float flake_value, float strength)
{
    float3 flake_values =  PRNG(flake_value);
    flake_values.y = PRNG(flake_values);
    flake_valyes.z = PRNG(flake_values);

    flake_values = (flake_values*2.0f) – 1.0f;
    return flake_values*strength;
}
```

Filter glitter flakes

The sparkles, as mentioned earlier, are represented through a bump map. In order to get sparkles from this bump map, a filter must be applied. This filter should eliminate the perturbation normals that do not make a positive contribution to the final colour. If the sparkles would have had a separate lighting calculation, it would just be a matter of making the calculations required and then add the result to the total specular value. But in this implementation, the sparkles are integrated in the regular lighting calculation by simply perturbing the face normal. Therefore, the perturbation normals that will have a positive impact on the specular value needs to be identified before doing said lighting calculations. There are different ways to do this depending on what type of lighting model that is used.

If the Phong specular model is used, it all boils down to finding out if the perturbed surface normal dot product with the half vector is larger than the corresponding product for the surface normal. So, in order to find out if a flake is actually making a positive contribution to the specular lighting, the following check should be applied:

```
float3 strongestNormalPointLight
(
float3 normal1, float3 normal2, float3 view_vector, float3 position,
 float3 light_pos)
{
    float3 light_vec =normalize(light_pos-position);
    float3 half_vec = normalize(view_vector+light_vec);
    float N1_dot_H = dot(normal1, half_vec);
    float N2_dot_H = dot(normal2, half_vec);

    if(N1_dot_H > N2_dot_H)
        return  normal1;
    else
        return normal2;
}
```

In the Image Based Lighting (IBL) case, the calculations found in Phong aren't used. IBL is namely little more than finding the reflected vector and use that to look up the lighting information. This means that the lighting information is localized in the cube map, making it infeasible to retrieve how the lighting changes as a function of the surface normal. The filter used in this case needs to reflect two rays, one for the original surface and one with the flake perturbation, and compare them. If the normal that has been altered by the flakes get a higher value then it is sparkling and should be kept.

There is one problem though, and that is that IBL is not restricted to just one tone of colour. One point may have RGB values of [0,0,255] and another [128,127,0]. So how should they be compared? It should be the one that has the largest value for any channel. The reason is that even if some point has a large sum of RGB values it does not guarantee a strong signal. These could, for example, be spread evenly across the channels, creating some gray tone. Compared to some other RGB value with the same sum but with only non-zero at one of the bands, the former would appear darker.

By using these methods mentioned above, the perturbation normals that have a negative impact can be filtered out.

Elimination of glitter flakes

The sparkle effect of metallic paint is a very important one indeed, and also an effect that the user should be able to control quite well. Even though the number of visible sparkles can be controlled indirectly by altering how much the perturbation normal can affect the surface normal, one might still end up with a sparkle cover that is too dense for a given scene. Another issue has to do with the fact that the bump map is generated through the direct use of PRNG. This function does namely suffer from gridding patterns when used directly for rendering.

One way to counter both of the earlier mentioned problems is to eliminate sparkling flakes according to some probability, which can be controlled by the user of the shader. This is similar to the noise cut off that was used for the Simplex noise in the earlier implementation as both aim to isolate single flakes. The aim is that the user can be offered a parameter ranging from 0 to 1 that determines the probability that a glittering flake is visible. Provided that Simplex noise has been implemented, there already exists a PRNG that can be used in this task.

The choice of input coordinates and the interpretation of the result from PRNG are the key issues in this implementation. PRNG gives a result ranging from 0 to 1, so the flake can be set as visible if it gives a result from 0 to some threshold and invisible otherwise. The input values must be the same for points within the same flake. This means that the current sampling position cannot be used. The reason is that this position does not uniquely identify the flake but rather the current position.

One naive approach is to use the flake normal as input to the PRNG. The flake normal is the same throughout the flake, so it would certainly work as an identifier. It is, however, not a unique identifier for the flake but for the flake type. If any flake that has some normal x gets discarded, all flakes with flake normal x would also be eliminated. This has a very bad impact on the distribution of the flakes, especially at high elimination rates. The glitter would be in favour for certain angles from the observer and maybe not exist in other angles.

If instead the centre, corner, or any other fixed spot on the flake can be found, that position can be used as a unique identifier for the flake. This can easily be calculated when creating the physical representation of the flake. Instead of just keeping track on the width, a vector containing the distance for each axis to the centre of the flake can be returned. This vector can then used to get the centre of the flake.

Rendering figures with absolute pixel size
One of the problems with the flake sparkles is the small size. It must be small as single sparkles should only be visible at close range. However, it should not appear to get much bigger if viewed somewhat closer. This creates a problem, either the flakes gets too big and cause visual problems at close range, or they get to small and cause aliasing problems at relatively close viewing distance. The solution proposed for this problem is to set the size of the sparkles in pixels rather than model coordinates.

Firstly, the flakes are created with a starting size given as argument. This starting size corresponds to the size in model space. It also corresponds to the maximal size the flakes can take before overlapping each other.

Secondly, information about how much a pixel on the screen corresponds to in model space needs to be retrieved. By having this information, the desired size in pixels can be multiplied with the ratio to get the necessary size in model space. If the information about field of view or aspect ratio setup is available, then a simple geometrical calculation can achieve this information. This is however not necessary if the ddx and ddy functions found in CgFX are used. These functions provide the derivative of the input variable, which in practice is the difference of the given argument if the pixel position was moved one step along the x or y axis respectively. Using these functions, a new size can be calculated for the flake as demonstrated in the code snippet below.

```
float3 new_sparkle_size(float pix_size, float3 position)
{
    float3  pos_change_x = abs(ddx(position));
    float3  pos_change_y = abs(ddy(position));
    float3  max_change = max(pos_change_x,pos_change_y);

    return max_change * pix_size;
}
```

Regardless of how the flakes are modelled, a function that can change the width of any given flake is needed. It is trivial to do this for texture coordinates, but a little more thought is necessary when doing it for PRNG. This issue is handled in *Width of a PRNG-patch.*

The new width in model space, as mentioned before, is calculated by simply multiplying the desired pixel width with the position units per pixel ratio. There is one major drawback with this method: even if the size of flakes changes dynamically with the height, the distance does not.

This means that the closer the viewer gets, the more isolated the sparkles will seem. The user needs to be aware that the starting size in model space will have a great impact at far and close range. Not only does it affect how far away the sparkles can be seen, it also directly affects the isolation of the sparkles. Ideally, the user would only have to specify some height that the glitter should begin to disappear. It would also be good if the sparkles increased in number at close range, filling in the gaps between the existing sparkles. This is, however, not supported in this model.

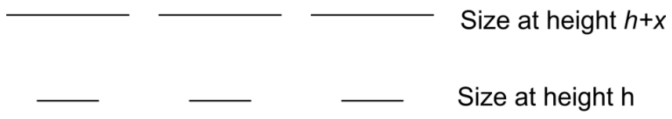


fig 4.20 Size of and distance between sparkles

The patches will have to fade out at some point, and it will be based on the specified maximum model space size. For anti-aliasing, the fadeout function mentioned in *4.1.6 Anti-aliasing of noise* can be used. The parameters to the fadeout function would in this case be:
$fadeout(1{,}0, startingSize, posPerPix * flakeSize)$
This result can then be multiplied with whatever the content of the patch is. In the case of the sparkling flake, the content is a perturbation normal.

A sample on how this behaves can be seen in *fig 4.21*. Note that the glitter remains about the same size up to the third picture. After that, the glitter has reached its maximum size and begins to fade away due to anti aliasing procedures.

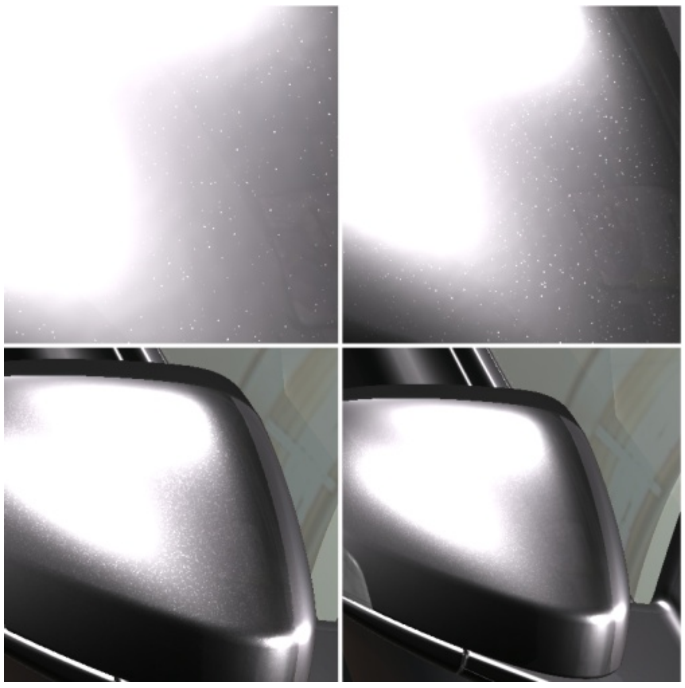


fig 4.21 Sparkling flakes at different distances

Width of a PRNG-patch

By using PRNG directly, one will get a patchwork of jagged squares and triangles that covers the surface of the object. In the section *Rendering figures with absolute pixel size* it is stated that there is a need to have access to the width of a flake in order to be able to adjust its size. This section describes how this can be done for PRNG.

The aim is that the width should be able to control with just one parameter. One way to do this is to represent the patch in some, preferably simple, geometrical form like a square or a circle. The desired width can then be used to find the points in the PRNG patch that fits within that shape. In order to make this comparison, the relative position of a point in a patch needs be found. This can be done by determining how close the sampling point is to result in another texture output value, which is shown in the function below.

```
#define TEXTURE_RES 256 // Resolution of the texture

float  PRNG_square(float3 pos, float size, sampler2D tex_sampler)
{
    float PRNG_value = 0.0f;
    float3 dists =0.0f;
    pos = pos/(size*TEXTURE_RES);

    for(int axis = 0; axis < 3; axis++)
    {
        // The distance to the next texture value for each axis
        dists [axis]=  frac(PRNG_value + pos [axis]) * (size*TEXTURER_RES);
        PRNG_value = tex2D(tex_sampler,float2(pos [axis]+ PRNG_value,0)).r;
    }
    // Is the sample point within the flake borders?
    if (abs(dists.x)> size || abs(dists.y)> size || abs(dists.z)> size)
        PRNG_value =-1.0f;

    return PRNG_value;
}
```

The output from the function above is either the actual random value or -1. If -1 is returned, it means that the sampling point was outside the flake square.

It is also mentioned in *Elimination of glitter flakes* that in order to get a unique identifier for a flake, the actual distance per axis is necessary. To get this, the above code can be modified so that the *dists* array is returned along with the random value. This also means that the output vector has to be enlarged to a float4.

Glitter strength

The strength of a sparkling flake, when compared to its immediate surroundings, is highly dependent on the variation of the face normal and the strength of the light source. With a high normal variation, the sparkling flakes may be spotted along a greater area, and with a strong light source, the difference in lighting from a glittering to non glittering spot is large. But there are times when a strong light source isn't desired but strong sparkles are. Also, the user may want to have white glitter when the glitter is strong rather than the materials specular colour tone.

For these occasions, a glitter boost can be introduced that artificially enhances the specular strength at the glittering flakes. The strength of the flake is enhanced by a simple linear interpolation between the specular value and some target value, where the interpolation point controls the amount of boost. The target value in the interpolation function represents the strongest signal the sparkles can take. White is a good choice for this value as it can emulate really strong sparkles, but RGB values where the original difference between the bands is kept but with a larger band sum is also a good alternative. The best variant is to let the user set the target colour as it allows for a good amount of flexibility.

Often the variation of normals is enough and the boost is not needed. The boost should be used with care as it is very artificial and can introduce unnatural looking material if used recklessly.

Adapting glitter
In the *Rendering figures with absolute pixel size* section, it was mentioned that one drawback with the resizing of sparkles is that the sparkles get more and more isolated as the viewing distance decreases. Another way to view this behaviour is that the flakes come closer together with increasing viewing distance. If flakes are not being eliminated, the sparkles will at some point become so close to each other that it looks more like a complete flake cover than just sparkles. This can actually be a good thing; it means that the glitter model can go from sparkles to then help in the general colour variations caused by the flakes at long range.

There is one problem though, namely the glitter elimination. While it is good at close range to get rid of superfluous sparkles and gridding patterns, it is prohibitive for simulating colour variation when viewed from a distance as it makes the flake cover less dense. It is therefore desirable to have some way to dynamically change the elimination rate as a function of viewing distance. By offering this function, the user can choose to let the glitter help in the general flake appearance by eliminating less flakes at long range and thus get better cover.

The fadeout function described in *4.1.6 Anti-aliasing of noise* is primarily made to avoid aliasing problems. It can, however, be used for other purposes, including this particular one. By setting the average value to 0, the signal to the probability of elimination and the feature size to 1, the elimination rate will increase when the viewing distance decreases and vice-versa. Expressed as a function call, it would look like this:

$$fadeout(elimProb, 0{,}1, posPerPix * flakeSize)$$

One advantage of this method is that the interpolation will accommodate for different sizes of flakes.

4.3.5 Passive flakes

Passive flakes are those flakes that don’t emit a strong sparkle but still gives the surface a variation in colour. Unlike the sparkling flakes, it is not at all a given that using perturbation of surface normal is the correct approach. The colour variations can namely also be achieved by altering the specular value directly.

The case going for the perturbation of surface normals is that it more accurately emulates the nature of the flakes. It also is gives more life to the surface as it will change appearance depending on viewing direction. Provided that methods similar to the ones proposed in *4.3.4 Sparkling flakes* is to be used, the lack of native support of level of detail is a severe drawback in this particular application. Also, the sparkling flakes already accommodates for the noticeable viewing dependent variations caused by the flakes.

In the case of specular map alteration, one of the major advantages is, depending if fBm is used, native support for level of detail. It is also potentially very easy to control as the alteration of the colour is predictable with its isotropic behaviour. This approach is also complemented well with the sparkling flakes as the latter will take care of the viewing direction dependent effects.

Both of these models have been tested and the model with the specular map alteration proved to be the better for its flexibility and level of detail. But, this result relies on that the sparkling flakes are used. A model with flake perturbation functions better as a standalone model when compared to the specular map alteration approach.

Another important aspect to consider is how the generation of the flakes physical shape is done. Three types were considered and are handled below.

PRNG
The PRNG method can be used directly when generating these passive flakes. By using the current model position as input and the output to alter

the specular map, one can get high frequent variations of the colour across the surface.

One big advantage of this method is that it is relatively cheap when compared to the alternatives. It looks slightly worse in close range when compared to e.g. Simplex noise as the jaggedness and gridding tend to give it away at a relative early stage. The major problem is that PRNG lacks the flexibility of fBm as the latter can both do high frequent variations and accommodate for level of detail by adding more octaves. So this model was omitted due to lack of flexibility.

Simplex noise

If the level of detail or flake contribution from quite some range is not a priority, usual Simplex noise is an alternative to consider. A quite high frequency is needed to simulate the flakes, which it also does quite well. This approach does suffer from too large flakes at close range, but this is not such a terrible drawback as the shader probably shouldn't be used at such close range anyway.

Due to the fact that fBm can act as usual Simplex noise by just setting the number of octaves makes this model unnecessary restrictive.

fBm

Fractional Brownian motion is an interesting choice because of its built-in level of detail. By having several octaves, the flake contribution can be observed quite some distance away. It should, however, have much more impact at close range than at a distance where the contribution should be quite small. This suggests that each fractal step should not have the same impact and favouring those with higher frequency.

One setting that fulfils these requirements is amplitude and frequency multipliers of 2.7 and 2.4 respectively. This setting not only makes the higher frequencies more prominent, they are also reached quite fast with a slightly higher frequency multiplier than normal.

Given that the main advantage with fBm is level of detail, this model can become somewhat redundant if the method described in *Adapting glitter* under *4.3.4 Sparkling flakes* is used. It will, however, make a good contribution to the visual appearance when the glitter gets more isolated as the viewing distance shrinks. Furthermore, it is a very good complement if the user chooses to not use the adapting glitter function. In the end, the fBm method was used to emulate the passive flakes due to aforementioned positive properties.

The passive flakes generated by fBm can be seen in *fig 4.23*. Note how much the colour fluctuates compared to *fig 4.22*. By combining the sparkles in *fig 4.24* and the passive flakes, an even more varied appearance is achieved, which can be seen in *fig 4.25*

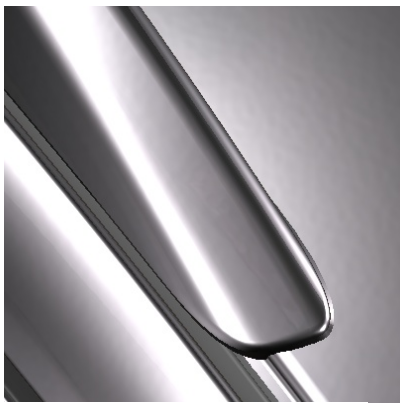


fig 4.22 No flakes

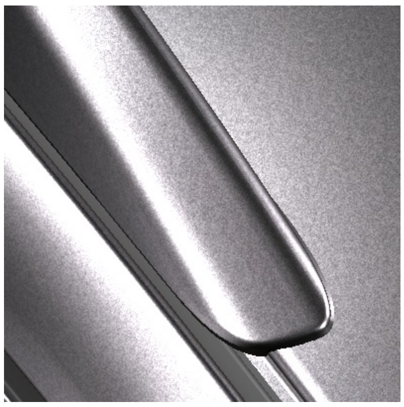


fig 4.23 Passive flakes

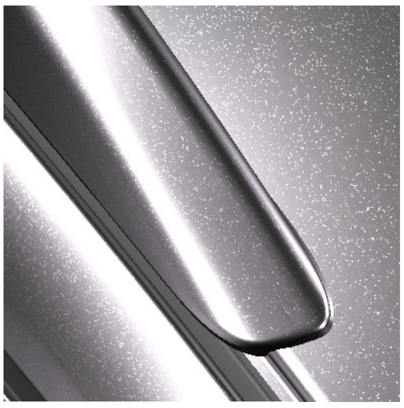


fig 4.25 Sparkling flakes

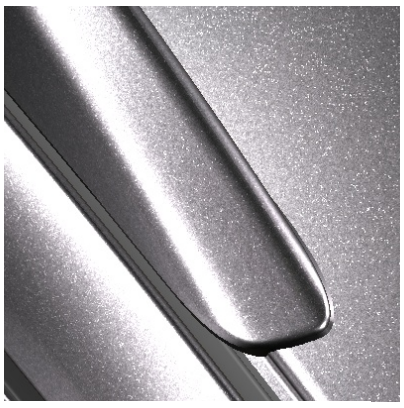


fig 4.24 Passive and sparkling flakes

4.4 Tempered glass

4.4.1 Material Description

Tempered glass is stronger than normal glass, but it also has the property that it breaks into tiny fragments when shattered. This is why there must be tempered glass in car windows (except the windshield, which uses laminated glass) rather than regular glass. If normal glass would be used, the glass would, in a collision, shatter into large chunks that become a serious threat to the passengers of the colliding car.

Depending on how the tempering is done, the glass may get slightly bent in certain ways. During the tempering process, it may namely be transported around by rollers. These rollers will bend the glass, creating a periodically wavy surface [17]. This results in a wavy reflection when faraway objects are reflected in the glass. The reason that it needs to reflect some object far away, is because the waves have such low amplitude that the difference in reflection will only be noticeable if the light has to travel a far distance. [18]

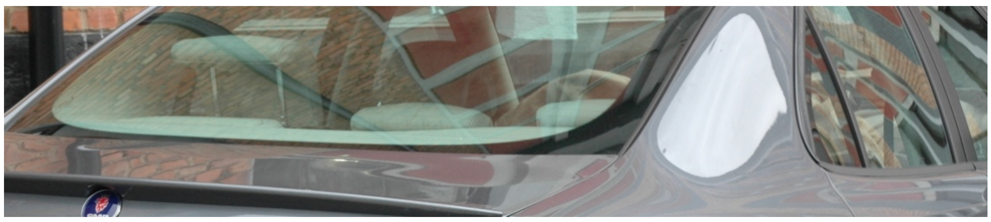


fig 4.26 Note the distortion in the reflection of the glass, especially at the windows on the right

4.4.2 Implementation

One important thing to note is that these waves in the glass appear periodically [17]. This means that a simple periodic function, like cosine, might do without having to use Simplex noise to alter the surface normals. Simplex noise should not just be omitted by default however, so a variant using noise is handled below along with a variant that only uses a simple cosine function.

Usual Simplex noise should be used instead of fBm due to the fact that the bumps should be quite distinct and not smoothed out. By using the noise gradient to determine how the face normal should be altered, the Simplex noise can be used directly for bump mapping. Simplex noise offers ways to let the user decide the frequency and amplitude, both very important in this particular application. Furthermore, as the simplex noise is used directly here, the frequency and amplitude parameters are the only ones that the user can alter.

This variant, however, does not yield a convincing result. The glass looks bumpy, which is not desirable in this case as wave pattern is more suitable. By lowering the frequency the bumps can be avoided, but it will also not vary the normals frequently enough to create the desired effect.

The other variant, using a simple sine function, works better. It is based on the assumption that the waves can be modelled as a sine function. It offers control of frequency, direction and amplitude of the waves, giving quite good control over it. The function that calculates the new position on the wave, given some surface normal, looks like this:

```
position = sin(dot(position,direction) * frequency)) * amplitude;
```

As it should be used in bump mapping, the derivate is needed of the former function, which is the following:

```
float3 wave_gradient = normal * direction * frequency * amplitude *
cos(dot(position,direction) * frequency);
```

This model is almost the simplest possible for a wave shape that uses a sine curve while still offering quite a bit of control. Even though this gives quite good results, there hasn't been that much time spent studying this variation. So there are certainly room for improvements.

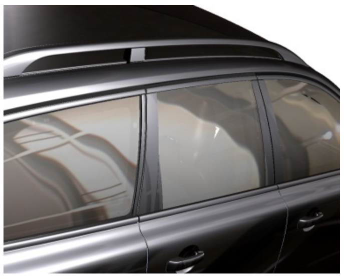


fig 4.27 Bent glass

4.5 General variations

One separate shader was created to be applicable for general materials, providing quite general methods to introduce variations in the material. Radermacher et al. [3] research on what people perceive as realistic images revealed that a rough surface on objects can add more realism to the picture as a whole. With this in mind, some simple methods that introduce disturbances in surfaces and materials in general were implemented. These are not ground breaking in any way, but highlight some different ways to give a material more life.

4.5.1 Roughness

The very same implementation of the Oren-Nayar diffuse model that the auto carpet shader uses is also used in this one. This diffuse model provides an easy way to make an object look rougher. For more information on this parameter, consult *4.2.5 Roughness*.

4.5.2 Small variations

Several materials, including some plastics, may appear quite monotone in their colour at first glance. But by looking closer on the material in question, some small variations can be seen. There might be small spots where the colour seems darker or brighter, there might be tiny bumps, or both.

With the ability to calculate the partial derivate analytically, the bumps and colour variation can be received for the price of just one Simplex noise function call. This makes it quite suitable to use for colour variation and bump mapping.

The colour variation uses the actual noise value to determine how much the colour deviate at a given point. At any given point, the colour can take any value within the range of $\left[col - \frac{col}{variation}, col + \frac{1-col}{variation}\right]$. This ensures that, as long the *variation* variable is set to a value in between zero and one, the result should not go outside [0,1].

In the implementation done in this thesis, the colour variation only affects the specular value. The reason was at the beginning to make the bumps look more highlighted so that they would seem to receive more light than the valleys. However, an error in the implementation had flipped the sign of the noise, which resulted in a higher specular value for the valleys.

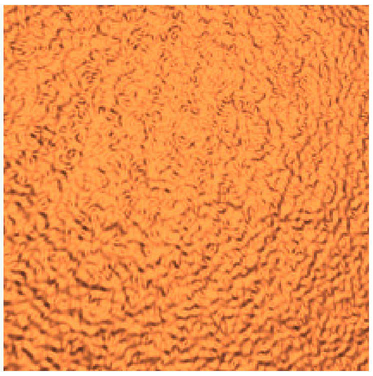


fig 4.28 Specular variation off

It turned out that it actually helped to produce an effect of bump embossing and a bit of fake subsurface scattering. The higher specular value makes the valleys look thinner than the bumps. So, with these combined, it looks like the surface actually is a thin skin where it is thicker at the bumps.

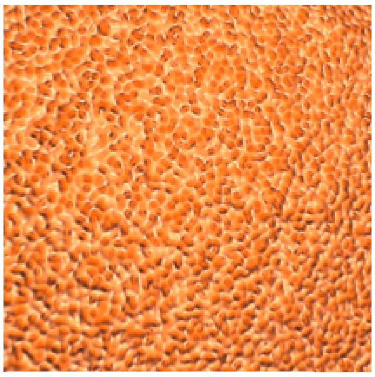


fig 4.29 Specular variation on

4.5.3 Dents

Some materials and surfaces, especially those with a soft core like pillows and cushioned seats, tend to have low frequency height differences along the surface. Take e.g. the seat of a car, where one might find quite large bumps after have being used as a place to sit. Another example is a sheet of some soft material that has been stretched over a surface but where the sheet is slightly larger than the surface, meaning that there will appear some bumps along the way due to the excess of material. On other materials, large dents may appear after have received some beating.

To handle these situations, a general fBm function is introduced. It is based on fBm where the quality can be specified by setting the number of octaves. The amplitude and frequencies multipliers are also exposed so that the appearance of the bumps can be controlled more thoroughly.

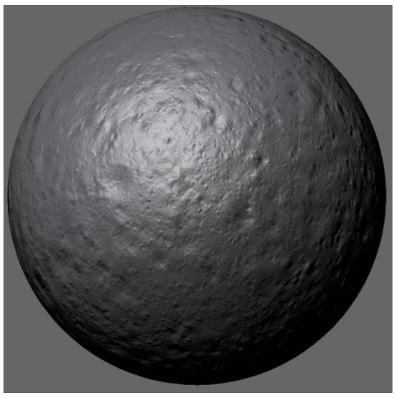


fig 4.30 Dents with 70% drop

Another parameter determines how often there should be bumps. This is done by eliminating some gradients in the Simplex noise function internally according to some given probability. Dropped gradients will no longer add to the noise values. Therefore, given high enough probability of a drop, areas with little to no bumps will appear. It looks quite good up to about 0.7 in drop rate; after that the bumps indeed get even more isolated but also unnatural looking.

5 Result

5.1 Visual realism

This is an area that is hard to be objective in as realism lies very much in the eye of the beholder. There are several pictures scattered throughout this thesis and also in *Appendix A* so that the visual result can be seen, but there are a number of findings that is of particular interest.

The introduction of flakes in the metallic paint does indeed give more life to the paint shader. This is not only true for the flake model in this thesis, but also for those presented in *3. Previous Work*. This model, however, handles strong sparkles as well as colour variations in a way that avoids tiling issues and provides means to let sparkles appear from a further distance while still maintaining a small physical size.

The carpet shader performs very well at close/medium and upward ranges, giving a rugged and lively look. By simulating some threading in the carpet through generation of separate diffuse and specular maps, the appearance of an actual carpet is enhanced. It does, however, lack realism in very close range where the illusion of a carpet is almost completely gone.

General application of simple noise functions on different material does indeed, if used correctly, make the material appear more realistic. Often, a very small noise contribution does the trick. There was nothing new in the general noise shader really; it was just means to show how much just some noise can do for a material.

5.2 Artist control

All of the shaders are highly configurable from the artist point of view. There are several parameters for each material and, in several cases, quite a few for each variation, which makes it possible to get quite different appearance with respect to the variations on each material. These parameters, however, are based on quite technical aspects, such as amplitude and frequencies of curves, rather than real material properties. The reason for this is that it enables high amount of control for the artist by letting them change quite low level parameters. It would, however, be nice to have another variant with less number of parameters that has a more to do with actual material properties.

One variation that is particular tricky to deal with is the metallic flakes with automatic resize support. It requires two parameters to determine the flake size and they function quite differently. While one controls the maximum pixel size of the glitter, the other determines the maximum size in model space. One could choose to view the latter as a distance of some sort; the larger it is the further the flakes will be apart at each specific viewing distance. This is somewhat cumbersome, just using one parameter for the size would be desirable.

The auto carpet and the general material shader are probably the easiest to handle while still being very configurable. It is quite clear what the parameters do and the outcome is predictable as they don't have parameters that are interdependent in the same way as the flake size is in the paint shader.

5.3 Performance

As mentioned in *1.2 Limitations*, there were no robust tests done on how the variations affected the performance. But the variations have been made so that quality can be traded for faster rendering by the use of different parameters. Examples of this are that the number of octaves in fBm can be set and that any effect can be turned on or off.

It is easy to see that the most demanding function to run is that of Simplex noise. This can be seen in just the number of calculations and texture lookups needed. So there is much to be gained in rendering speed by using an optimized Simplex noise function.

A simple modification can also do away the GPU calculation of the noise and replace it with a pre-generated three dimensional noise texture. This would, however, mean that the benefits of procedural generation will be lost. Another way to optimize is to try to find better models that e.g. make use of fBm with fewer steps than required in some variations in this thesis.

6 Future Work

6.1 Procedural generated 2D representation of threads

Even though the proposed model for threads in a carpet does give very good visual results at close/medium range and upwards, it is not good at all at very close range. So, a model that can handle carpets at micro level reasonably well could become more flexible than the current one.

6.2 Procedural generated scratches on a surface

After a while, surfaces and materials may get some scratches on them through wear and tear. This variation was considered to be covered in this thesis, but got omitted because the cars, for which these shaders was aimed at, should appear very new without any wear and tear.

It would, however, by quite interesting to see how some random scratch patterns can be generated on the GPU. There is some research done on the rendering of scratches, but less so on the generation of scratch patterns.

6.3 Better multilevel model of flake glitter

The flake sparkles in this thesis adjusts itself depending on the viewing distance by resizing themselves. The model doesn't, however, adjust the distance between the glitters. This makes the flakes get more isolated when the viewing distance shrinks. Therefore, it would be good to have a model that can increase the number of sparkles as the viewing distance shrinks to fill in the gaps.

7 References

[1] T. Akanine-Möller and E. Haines. *Real-Time Rendering.* 2nd ed. Natick : A K Peters Ltd, 2002.

[2] S. Gustavson. "Simplex noise demystified," 22 March 2005. [Online] Available: http://www.itn.liu.se/~stegu/simplexnoise/.[Cited: 06 March 2008.]

[3] P. Rademacher, J. Lengyel, E. Cutrell and T. Whitted. "Measuring the Perception of Visual Realism in Images," in *Proceedings of the 12th Eurographics Workshop on Rendering.* 2001. pp. 235-248.

[4] T. Nilsson. "Real-Time Visualization of Metallic Paint Glittering," Master's Thesis, Chalmers and University of Gothenburg, Gothenburg, VG, Sweden, 2004.

[5] ATI Technologies. *Metal.rfx.* [Source code HLSL] Markham : RenderMonkey 1.71, 2007.

[6] J. Günther, T. Chen, M. Goesele, I. Wald and H. Seidel. "Efficient Acquisition and Realistic Rendering of Car Paint," in *Proceedings of 10th International Fall Workshop - Vision, Modeling, and Visualization (VMV) 2005.* 2005. pp. 487-494.

[7] V. L. Volveich, E. A. Kopylov, A. B. Khodulev and O. A. Karpenko. "An Approach to Cloth Synthesis and Visualization," in *Proceedings of The Seventh International Conference on Computer Graphics and Scientific Visualization Graphicon-97.* 1997. pp. 45-49.

[8] N. Adabala, N. Magnenant-Thalmann and G. Fei. "Real-Time Rendering of Woven Clothes," in *Proceedings of the ACM symposium on Virtual reality software and technology.* 2003. pp. 41-47.

[9] K. Perlin. "Improving Noise," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* . 2002. pp. 681 - 682 .

[10] A. Jönsson. "Generating Perlin Noise," *AngelCode.* February 2002. [Online] Available: http://www.angelcode.com/dev/perlin/perlin.asp.[Cited: 28 01 2008.]

[11] K. Perlin. "Making Noise," *Noisemachine.* 19 Februray 2006. [Online] Available: http://www.noisemachine.com/talk1/index.html.[Cited: 29 01 2008.]

[12] H. Elias. "Perlin Noise," *Good Looking Textured Light Sourced Bouncy Fun Smart and Stretchy Page.* 7 December 1998. [Online] Available: http://freespace.virgin.net/hugo.elias/models/m_perlin.htm.[Cited: 03 February 2008.]

[13] N. Tatarchuk. "The Importance of Being Noisy: Fast, High Quality Noise," Conference session in Game Developers Conference 2007. [Online] Available: http://developer.amd.com/assets/Tatarchuk-Noise(GDC07-D3D_Day).pdf.[Cited: 14 03 2008.]

[14] M. Oren and S. K. Nayar. "Generalization of Lambert's Reflectance Model," in *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques.* 1994. pp. 239-246.

[15] R. Fosner. "Implementing Modular HLSL with RenderMonkey," *Gamasutra.* 14 May 2003. [Online] Available: http://www.gamasutra.com/features/20030514/fosner_03.shtml.[Cited: 26 02 2008.]

[16] M. Kesson. "Surface Shading: Fake Rim-Lighting (& other related effects)," *CG References & Tutorials.* 28 February 2005. [Online] Available: http://www.fundza.com/rman_shaders/surface/fake_rim/fake_rim1.html.[Cited: 26 01 2008.]

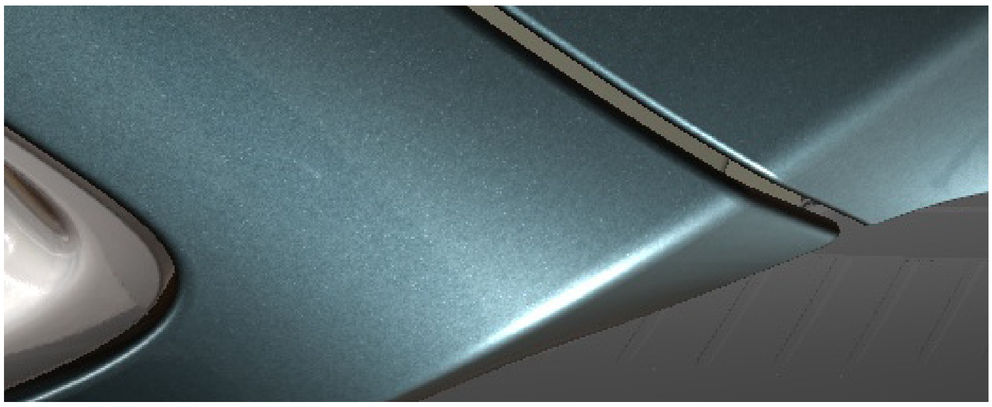
[17] M. Abbott and J. Madocks. "Roller Wave Distortion - Definition, Causes and a Novel Approach to Accurate, On-line Measurement,"

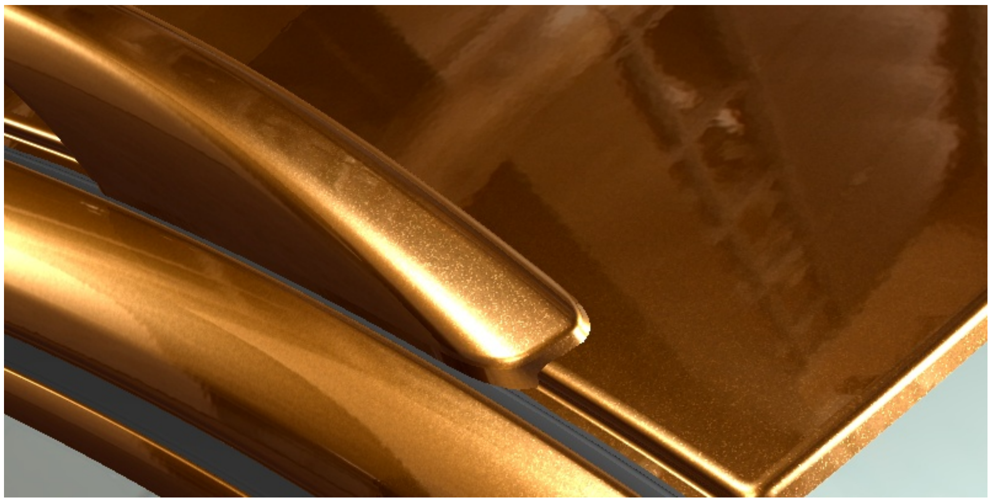
Glassolutions. June 2001. [Online] Available: http://www.aisglass.com/distortion-tepering/AIS-303.pdf.[Cited: 6 March 2008.]

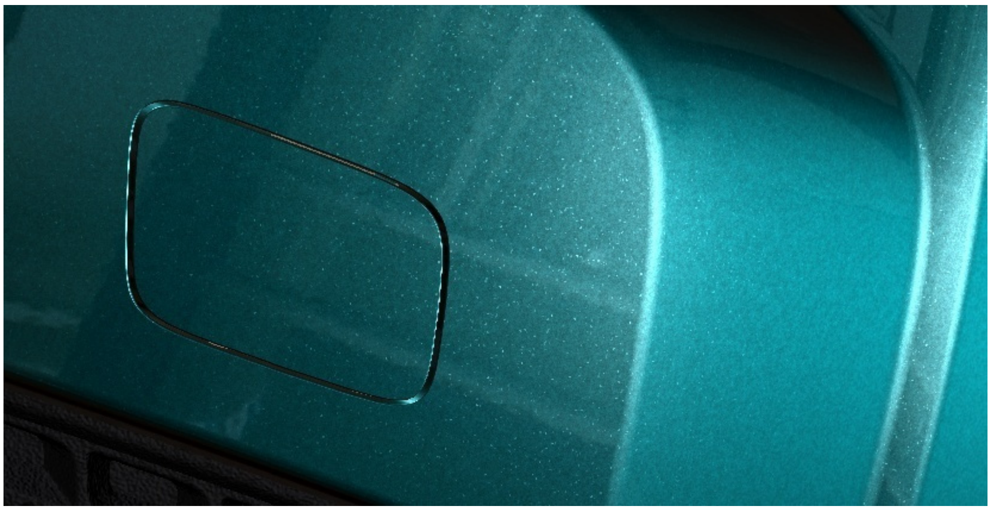
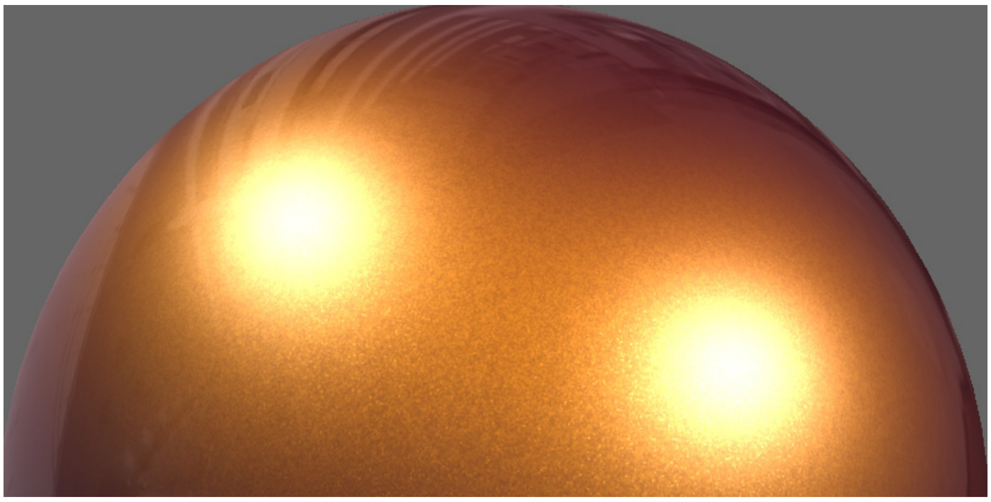
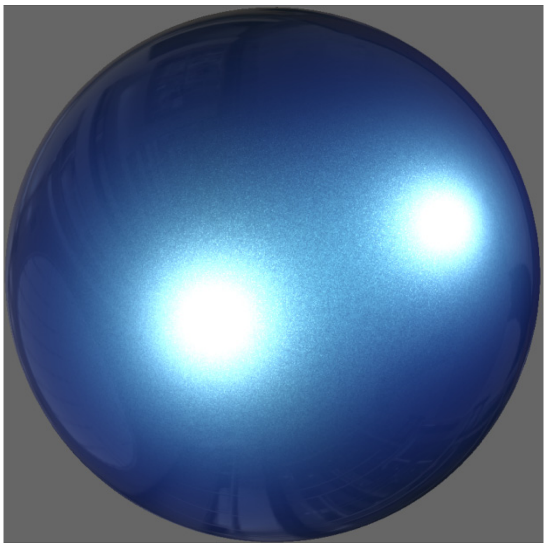
[18] AIS. "Tempered Glass," *Glassolutions.* [Online] Available: http://www.aisglass.com/flat_tempered.asp.[Cited: 24 01 2008.]

Appendix A. Sample Images

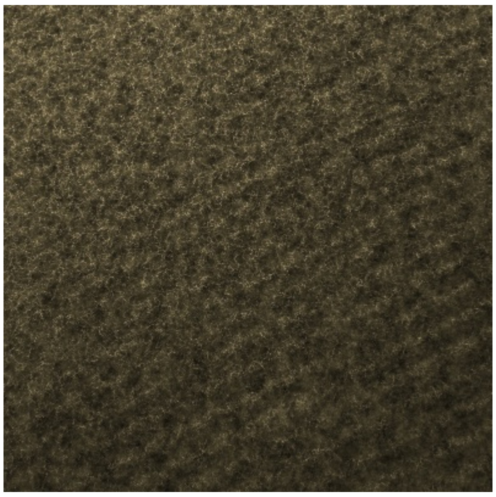
Metallic paint





Auto-Carpet



General

Appendix B. Variation parameters

Here is a list over the different parameters that can be used for the different variations.

Metallic Paint

Orange Peel Strength

Sets how much the orange peel can affect the surface normal. Goes from 0 to 1. Appears in *4.3.2 Orange Peel*.

Orange Peel Size

The size of the orange peel. Appears in *4.3.2 Orange Peel*.

Flake sparkle pixel size

Sets the maximum pixel size of the sparkle. Appears in *4.3.4 Sparkling flakes*.

Flake sparkle physical size

Sets the maximum model space size of the sparkle. Appears in *4.3.4 Sparkling flakes*.

Flake sparkle variation

Determines how much the flake normal can affect the surface normal. Appears in *4.3.4 Sparkling flakes*.

Flake Sparkle Boost

Gives the sparkle a specular colour boost. Appears in *4.3.4 Sparkling flakes.*

Flake Sparkle Elimination

Eliminates sparkling flakes according to a probability set between 0 and 1. Appears in *4.3.4 Sparkling flakes*.

Smart Flake Sparkle

Automatically adjust the elimination rate of sparkling flakes as a function of viewing distance. Binary value, on or off. Appears in *4.3.4 Sparkling flakes*.

Passive Flake Size

Size of flakes that don't emit strong sparkles. Appears in *4.3.5 Passive flakes*.

Passive Flake Strength

How much the passive flakes are allowed to alter the specular colour. Appears in *4.3.5 Passive flakes*.

Passive Flake Detail

Sets the number of octaves that should be used for the passive flakes. Appears in *4.3.5 Passive flakes*.

Auto-carpet

Diffuse thread size

Size of the threads that should build up the diffuse map. Appears in *4.2.2 Rendering the threads*.

Diffuse thread strength

Sets how much the diffuse colour can be altered by the diffuse threads. Appears in *4.2.2 Rendering the threads*.

Diffuse thread bump strength

Parameter to determine how much the surface normal should be affected by the diffuse thread normal. Value ranges from 0 to 1. Appears in *4.2.2 Rendering the threads*.

Specular thread size

Size of the threads that should build up the specular map. Appears in *4.2.2 Rendering the threads*.

Specular thread strength

Sets how much the specular colour can be altered by the specular threads. Appears in *4.2.2 Rendering the threads*.

Dark area size

Sets the size of the dark areas of the carpet. Appears in *4.2.4 Dark areas*.

Dark area strength

Sets how much the dark areas can affect the diffuse and specular values. Appears in *4.2.4 Dark areas*.

Seams propagation

Determines the direction of the seams. It is a vector with three elements; the first two is the x and y of the propagation vector. The third is used as a flag to determine the pattern shape: 0: no seams, 1: lines, 2: grid. Appears in *4.2.3 Seams*.

Seam distance

Sets the distance between neighbouring seams. Appears in *4.2.3 Seams*.

Seams amplitude

Sets the amplitude difference between seam and non-seam. Affects the diffuse and specular map but not the surface normal. Value ranges from 0 to 1. Appears in *4.2.3 Seams*.

Seams bump strength

Sets how much the seam normal may affect the surface normal. Value ranges from 0 to 1. Appears in *4.2.3 Seams*.

Seam crossover size

The size of the areas where threads leans over the seams. Appears in *4.2.3 Seams*.

Seam crossover strength

Determines how much of an effect these crossings will have on the specular, diffuse and surface normal. Value ranges from 0 to 1. Appears in *4.2.3 Seams*.

Roughness

The micro facet roughness of the surface. Appears in *4.2.5 Roughness*.

Fake rim light

Sets how much rim light the surface should get. Appears in *4.2.6 Fake rim light*.

Glass

Wave frequency

The frequency of the waves on the glass. Appears in *4.4.2 Implementation*.

Wave amplitude

The amplitude of the waves on the glass. Appears in *4.4.2 Implementation*.

Wave propagation vector

Sets the direction of propagation of the waves. Appears in *4.4.2 Implementation*.

General

Roughness

The micro facet roughness of the surface. Appears in *4.5.1 Roughness.*

Small blemish size

The size of the blemish. Appears in *4.5.2 Small variations*.

Small blemish bump strength

How much the blemishes can affect the surface normal. Value ranges from 0 to 1. Appears in *4.5.2 Small variations*.

Small blemish colour strength

How much the blemishes can affect the specular map. Value ranges from 0 to 1. Appears in *4.5.2 Small variations*.

Dent strength

Sets the how much the dents can affect the surface normal. Value ranges from 0 to 1. Appears in *4.5.3 Dents*.

Dent size

The size of the dents. Appears in *4.5.3 Dents*.

Dent cover

How much the surface should be covered with dents. In practice this is done by dropping noise gradients. Value ranges from 0 to 1. Appears in *4.5.3 Dents*.

Dent detail

Determines how many octaves that should be used for the dents. Discrete value. Appears in *4.5.3 Dents*.