Material aging for game environments

Victor Antón Domínguez

Thesis supervisor: Marc Comino Trinidad
Tutor: Carlos Andújar Gran

Master in Research and Innovation
Facultat d'Informàtica de Barcelona

June 2019

Table of Contents

Abstract

In most video games, we want the player to be completely immersed in our world. For this purpose, we usually need very realistic scenes. And at the same time we need to keep models as simple as possible to be able to run the game at real-time frame rates, so that players do not lose the illusion of immersion.

One common way to improve the level of realism of an environment is to model imperfections such as dirt, scratches and peeling. Nowadays, many programs allow us to create such effects. However, it can be challenging to add them without increasing the complexity of our models.

In video games we can not afford very complex models, because it would make it difficult to achieve real-time frame rates. This is why it would be useful to have a tool that lets us create these effects with minimal overhead.

We propose a software and work-flow to create these imperfections while adding very little overhead to the rendering time of the models. Specifically, the tool we have implemented integrates all these effects into the physically based rendering (PBR) textures of the input model. Thanks to this, we will be able to create good quality effects at a very cheap cost.

Keywords: aging, physically based, textures, imperfections, real-time.

1. Introduction

1.1 Motivation

Nowadays, computer graphics has become a subject present in many scientific, social and entertainment areas. In many of these fields, it is extremely important that people feel very immersed inside artificially created 3D environments. But achieving this immersion is not a trivial task.

On the one hand, one of the key factors to make users believe they are inside this environment is the realism and detail of the models. For this, we do not only need to model the shape of the environment objects as meshes. We also need to consider the temporal dimension of it: we must also take into account that the environment ages and degrades with time. In addition, we must reflect the fact that most objects and scenarios have many tiny imperfections that our eye can rapidly identify, becoming crucial for the realism of our scene.

On the other hand, this task becomes more difficult if we want to immerse users inside an interactive experience. One must be able to provide users with real time frame rates, so that they do not notice they are inside a simulation. A minor stall in the continuous flow of images will break the immersion experience. Furthermore, with new incoming VR technologies, the frame rate must be even greater than it used to be. For this reason, one can not arbitrarily increase the complexity of the models in the scenario, as it would take a considerable amount of time to render them. Instead, one must carefully choose which are the features that the eye of the user will be able to see, and try to generate those chosen effects as simply and efficiently as possible.

Since this is a very time consuming and difficult process, it would be very useful to have a tool that provides these two key features. Furthermore, it would also be beneficial that this tool could be seamlessly integrated inside an asset or video game creation pipeline. At the end of the day, it would save human, economic and time resources to many small studios that may not be able to afford them. With such software, artists would be able to save time taken by this tedious task, and could instead focus on other more complex and creative processes.

This thesis aims to explore the best paradigm to create such a tool. As you will see, during the implementation of the tool, we have also created several techniques and algorithms that could prove useful for other fields.

1.2 Contributions

The work presented in this thesis explores a new tool paradigm on imperfections generation and aging. We present a layer-based architecture based on masks and the modification of the commonly used PBR textures (albedo, height, normal, roughness and metalness). This provides a versatile framework for artists so they can easily author new effects and tweak them.

Furthermore, we provide a set of imperfection brushes to draw the different PBR textures. on top of the model. When combined with the auto-generated masks, this lets the artist select the zones in which an effect should be applied. This interaction of procedural masks and brushes to paint over the model, lets users achieve the desired results very fast.

Finally, while implementing different types of masks, we have developed several GPU algorithms that could be useful for other programs. For instance, we provide a method to rapidly blur textures in 3D surface space. Another example is an algorithm to detect global edges that is independent of the model topology.

1.3 Thesis overview

This thesis will attempt to provide a modular tool specifically created to add imperfection effects to 3D models. We could approach the aging and imperfections problem in many ways, but after thoroughly revising all related and previous work, and taking into account the time that we had to develop this thesis and all the goals we wanted to reach, we have determined that a good way of tackling this problem would be to implement the graphical tool seen in Figure 1.

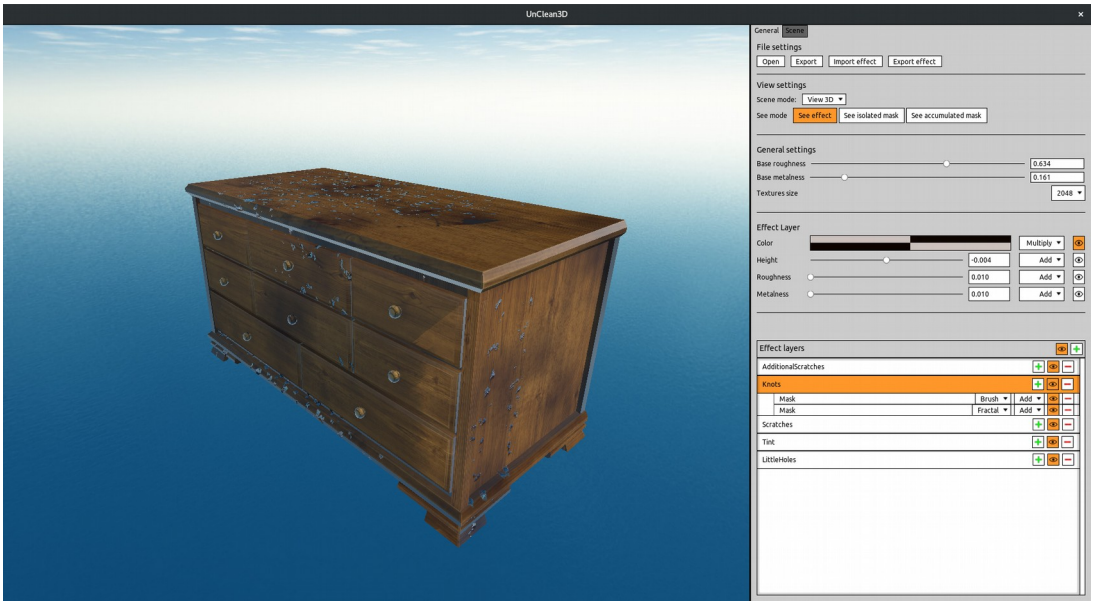

Figure 1: Picture of the tool that has been developed for this thesis.

This thesis has the following structure:

- 2. Previous work: We will go over some of the aging and imperfections work that has been previously done. This includes physical, artistic and heuristic approaches, and we will also analyze strong and weak points of each approach.
- 3. Physical Based Rendering: After showing the previous work, we will give some theoretical and practical background on physical based rendering. This part is very important because it sets up the physical rendering model in which all the work has been based, and we will refer to it very often.
- 4. UnClean3D development: Once we have set the needed background, this next section will explain the development process of the tool implemented for this thesis. This section will be subdivided in the following subsections:
    - 4.1 Objectives: In this section we explain the objectives of the tool and thesis, from which we will derive many of the decisions taken during the development process.
    - 4.2 Effects to achieve: The final goal is to implement a software to semi-automatically create a series of imperfection effects for a single 3D model. In this section we state those different effects that the tool will attempt to reproduce.
    - 4.3 Texture-only effects: Here we will explain what are texture-only effects, which will be a key component of our tool, since all the imperfection effects created by this tool will be added to the model by just modifying its PBR textures of albedo color, roughness, metalness, height and normals.
    - 4.4 Why texture-only effects: Choosing texture-only effects was not an arbitrary decision. Here we explain why we chose texture-only effects for this thesis, based on the stated objectives.
    - 4.5 Layer-based architecture implementation : Due to the fact that we will only be modifying textures, we must find some modular workflow that can let us modify them intuitively and effectively at the same time. In this section we explain how we implemented the layer based architecture, and the different iterations until we arrived to the final version.
    - 4.6 Types of mask: Our layer architecture will have mask textures as its key component. This section is very interesting, since here we show the different types of masks that have been developed, why, and how they can be useful for creating effects.

    - 4.7 User interface: We decided to create a graphical tool because we want authors to be able to instantly see the result of the effects being applied. Here we will quickly go through all the elements of the user interface, to give an overview of what the user can do with the tool.
- 5. Results: After explaining how the tool was developed, we will evaluate how the tool works. This section is subdivided into two smaller sections:
    - 5.1 Effects results: This is one of the most interesting section of this document. Here, we go through several pictures we have taken these past months, and try to reproduce the imperfection or aging effects seen in them using our tool. It is very important because it will let us see the capabilities and limitations of the software.
    - 5.2 Performance results: Finally, we go through some performance results. Here we will be able to see the amount of overhead that is added when using the tool.
- 6. Conclusions: Conclusions of this thesis, in which we see whether we have fulfilled the initial objectives.
- 7. Future work: And finally, in this last section, we will give a list of features that could be either improved or added in a future. This section can be useful for people who aims to extend this work, or is simply trying to implement a similar tool. By reading it, you can get some guidelines and avoid trying things that did not work for us.

2. Previous work

We can classify previous work based on whether they are physically based, artistic oriented or they use a mixed heuristic approach.

2.1 Physically based approaches

Physically based approaches use physical models to generate the aging or imperfection effects. Given a set of input variables that specify physical properties of the environment, they simulate or compute the process described by the chosen physical model, and later add it to the model.

Due to relying on well-tested physical models, they can provide very accurate results. In addition, once we set the physical attributes of a model or environment, the tool can process the models all alone by itself. Thanks to this, it is really easy to automate the aging process, and it can be really efficient to use in some development environments.

Nevertheless, physically based approaches sometimes do not allow users to see the results in real time, since the simulation time can take very long. Furthermore, tools based in physical models usually make it very difficult to predict or to control the final result.

This is why these tools are often aimed to create very realistic scenarios. But even if the physical model is very accurate, we also need to precisely define the physical properties of the environment. Gathering all this physical data can be very time consuming and it may require a good amount of research.

Dust accumulation

Dust accumulation is a very common effect. It is a typical aging effect that we can perceive in our everyday life. When dust is accumulated over a surface, it modifies the way light is reflected, and we can see how the surface color changes according to the dust appearance.

Hsu and Wong [HW95] proposed a dust accumulation approach which considers the dust source as if it was a light source. Then, it takes into account the angle between the dust source and the normal of the surface, taking into account the gravity. They also describe how to accumulate dust of different sources. In addition, they also consider the removal of dust because of external agents: they calculate which areas are more exposed (less occluded) by tracing rays in an hemisphere. See Figure 2 to see an example of how this removal effects might change the expected dust accumulation results.

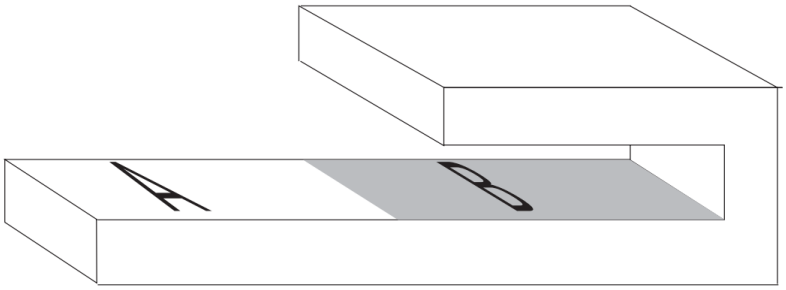

Figure 2: Example of [HW95] dust removal. Here, A is more exposed to dust, but B is safer from wind and dust removal entities. Due to this, given enough time, B will have more dust than A, despite A was more exposed to dust.

More recently, Guo and Pan [GP14] addressed it with a more accurate physical model than the one presented in [HW95]. First, they propose a fast evaluation approach to compute dust thickness distribution based on surface inclination and stickiness properties, as well as surface exposure and local stability. Then, in order to ensure high physical fidelity of the method, they render the scene by taking into account external and internal surface reflection (which [HW95] was not taking into account), and then they compare it with experimental results in order to demonstrate that results are very accurate. As we can see in Figure 3, instead of a single layer of dust from multiple dust sources as in [HW95], they can also handle several layers of dust accumulated one after the other and with different dust colors.

Figure 3: Results of dust accumulation using the technique presented by [GP14].

Impacts and scratches

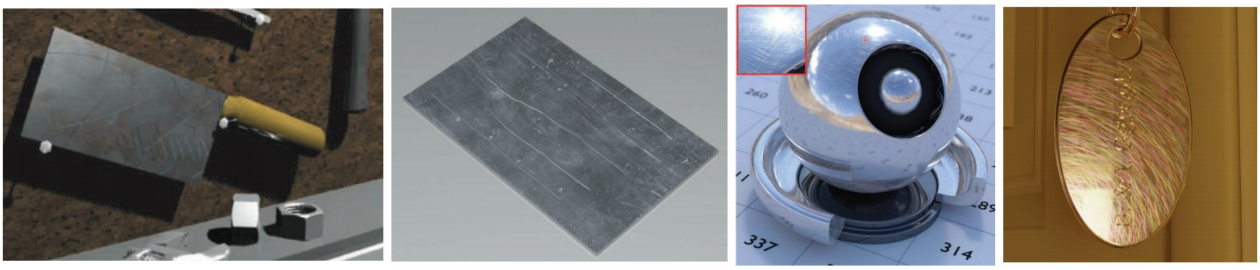

Figure 4: Different scratch results. From left to right: Merillou et al. [MDGS01], Bosch et al. [BOS04], Raymond et al. [RGB16] and Werner et al. [WER17].

Impacts and scratches are caused due to external bodies hitting the object over time, or because of tools or sharp objects scraping the object surface.

Mérillou et al. [MDGS01] approached it by using their own BRDF model. They use physical measurements on real objects to derive an accurate geometric model of scratches at small scale range (roughness scale). In addition, they also introduce a new geometric level between bump mapping and BRDFs. They represent where the scratches are located in the surface by using 2D textures.

Bosch et al. [BOS04] also approached it by using their BRDF model as [MDGS01]. They applied a different BRDF in zones that had individual scratches. However, in this case they derived the scratches data from a set of physical inputs, instead of from real word data as in [MDGS01]. For example, they let the user specify which scratching tool has been used, the penetration forces, or the material properties of the surface being scratched. In addition, instead of using simple 2D textures as in [MDGS01], they model each individual scratch with curves defining paths. Thanks to this way of defining scratches, they can offer “infinite” scratch resolution and they can determine with precision what is the scratch direction at each point of the scratch path.

Raymond et al. [RGB16] proposed a new spatially varying BRDF model that improves the rendering of scratched materials such as finished woods, plastic or metals. In order to get a very performant implementation without losing visual fidelity, they take advantage of the regularity of the structure of scratch distributions. In addition, they provide users with controls over the micro-BRDF, orientation and density of scratches. The BRDF for a single scratch is simulated using an optimized 2D ray-tracer and stored in a three-component 2D texture. In contrast to [MDGS01] and [BOS04], their approach takes into account all inter-reflections inside each scratch, including Fresnel effects.

And recently, Werner et al. [WER17] proposed a wave-optical shading model based on scalar diffraction theory to render the iridescence effects seen in scratched metals. Their model expresses surface roughness as a collection of line segments, very similarly to the individual scratch representations of [BOS04]. And in contrast to [MDGS01], [BOS04] and [RGB16], this technique is

taking into account complex diffraction effects. Finally, they compare it with real-world examples to demonstrate their physical plausibility.

Peeling

Peeling is a process in which the layers of a surface peel and fall off. This leaves the lower layer of the material visible at some zones of the surface.

Paquette et al. [PPD02] introduced a physically based technique to compute peeling. They use a 2D grid in which they simulate the peeling process by propagating starting cracks on the surface. They take into account physical properties such as paint strength and tensile stress to determine where cracks appear on the surface. In this case, they use explicit geometry to represent the peeling cracks. See Figure 5.

Figure 5: [PPD02] peeling approach.

Gobron et al. [GC01] describe a method for modeling the natural peeling over any type of triangulated 3D object. In contrast to [PPD02], they use crack input data, which is precomputed with a physically based procedure. In addition, they present how to define groups of peeling, how to determine a realistic and natural peeling order, and how to simulate the rendering of two types of special peelings. As in [PPD02], they use explicit geometry to represent the peeling cracks as well. See Figure 6.

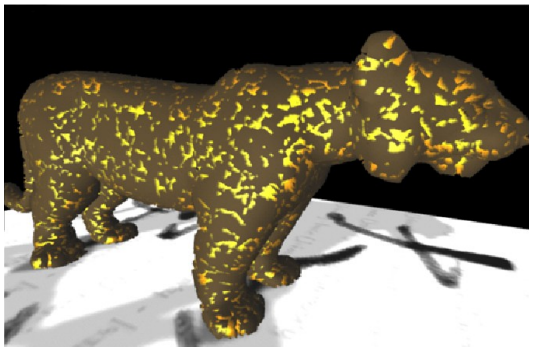

Figure 6: [GC01] peeling approach applied on a tiger with a paint layer.

Corrosion

Corrosion is a chemical process seen in nature that gradually destroys materials (usually metals), and gives them the well known brownish and deteriorated appearance.

Mérillou et al. [MDGC01] first presented a model that simulated corrosion guided by physical models. They take into account real world time and different atmospheric condition categories according to experimental data. The corrosion reaction is simulated using a random walk technique adapted to their generic model. They propose a BRDF and texture models for realistic rendering, that affects color, reflectance and geometry. Finally, they introduce a set of rules to automatically predict the preferential starting locations of corrosion.

Later on, Chang and Shih [CS03] proposed a technique to simulate the generation of rust in metals under sea water. The proposed model consists of two modules – a seawater module and a rusting module. The seawater module adopts tendency distributions to reflect the potential of rusting in seawater. The rusting module simulates the development of the diffusion of the rust. To generate the rust, instead of using random walk as [MDGC01], they use an L-system approach similar to that used for tree generation. They finally render it by using ray tracing.

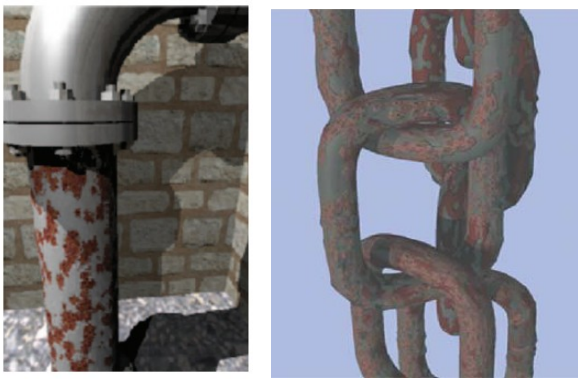

Figure 7: Results of corrosion effects. From left to right, [MDGC01] and [CS03].

Others

There are many other effects that have been tackled using physically based approaches. These include erosion, pollution or skin aging. There exists previous work for all of them that is also based in an underlying physical model or simulation. Nonetheless, they are not so relevant for this thesis. In [MBP18] you can find a detailed survey of these extra topics.

2.2 Artistic approaches

Artistic approaches are not usually presented as aging or imperfection software. We could say that a tool that gives an artistic approach to this problem is any program which allows the user to freely edit the model PBR textures or the mesh itself. This way, the artist is the only responsible for manufacturing plausible imperfection effects on the model, by using the generic tools provided by the software.

The main advantage of artistic approaches is the freedom that the user has to create effects. This lets the artist to stylize the aging in such a way that fits the target environment.

For example, imagine the case in which an artist wants to manufacture models for a cartoon video game. On the one hand, physically based software would be useless in this case because it will create effects that are too realistic. But on the other hand, with an artistic based software, the artist has much more control over the effects, and consequently can better tweak and adapt the style of the imperfections to match that of the cartoon environment.

Another advantage is that these approaches are often much faster to compute. Consequently, the tools using this approach can be implemented in such a way that they provide real time feedback to the artist. This is really important, because this way users can instantly see the final result in their screen and get an idea of how it will merge with the final environment.

Despite all this, one of the obvious weak points of artistic based approaches is that they usually lack of strict physical realism. Since these tools do not guide the user on the process, the physical plausibility of the effects completely relies on the skills of the user.

In addition, the aging process can not be automated. With these tools, a manufacturing user is always needed in order to use the tools to create the imperfections for each model. This means that they demand more human and time resources, and consequently they may not be efficiently suited for some production pipelines.

As said before, artistic approaches provide the artist the tools to freely modify the model PBR textures and/or the model mesh. There are many programs that can perform this task, such as Blender, Maya, ZBrush or Houdini.

Although all these pieces of software are extremely complex, they do not provide a specific easy and straightforward framework to add imperfections and aging effects to models. Consequently, at first sight, they can overwhelm users who seek more simple programs to create aging effects. The same job with a more generic tool becomes more difficult than needed and less efficient.

Nevertheless, it is still interesting to mention some artistic features that could be useful to create imperfections on models. For example, all Maya, Blender and Houdini provide physics simulation tools that can help a little bit when trying to create physically plausible imperfection effects. For instance,

one could let very tiny particles fall from several points, and see how matter would be deposited in the model surfaces and corners. After this, one could add extra geometry or texture effects to create the matter deposited in the zones in which the physical particles have deposited before. And of course, all these programs excel when it comes to providing artists the tools to author all types of 3D content.

While Maya and Blender are more focused on 3D mesh creation and animation, Houdini is a more interesting example because it is much more oriented to the authoring of very realistic visual effects, and you can certainly create very convincing and realistic imperfection effects with this tool if you know how to use it. However, it is more aimed for very high quality effects, which may not be well suited for very high frame rate applications such as video games. See Figure 8 for a very realistic and physically plausible effect made in Houdini.

In addition, there is a totally artistic program for PBR painting called ArmorPaint, and it is open source. It modifies only PBR textures, and the results are very realistic. However, it is not focused in aging or imperfections, it is meant to be used for general PBR painting. In addition, it is still in a very early development stage.

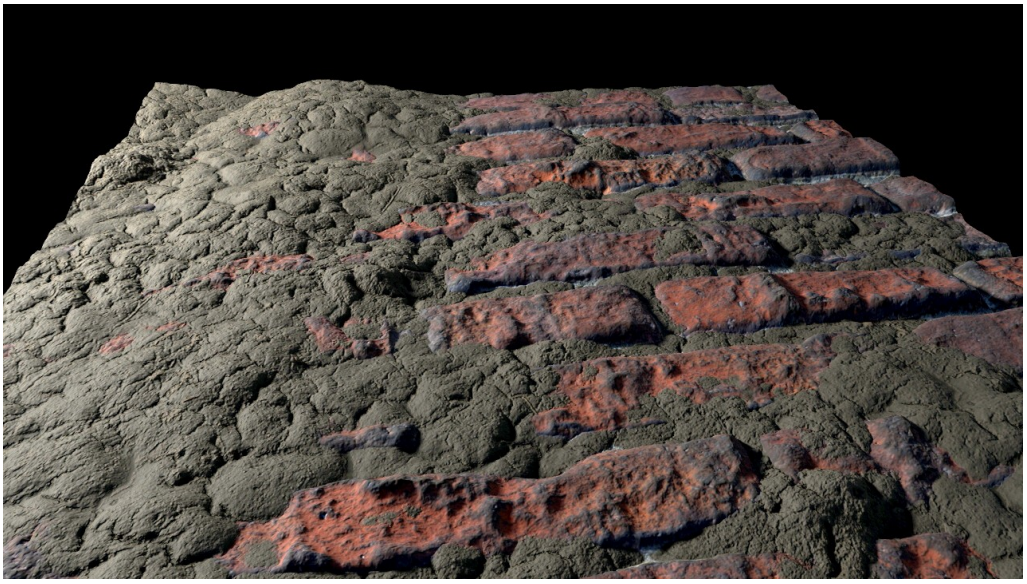

Figure 8: Houdini effect. As you can see it is of very high quality, but it certainly adds a heavy performance footprint to the model, which would not be appropriate for real time rendering.

2.3 Heuristic approaches

Heuristic approaches provide tools to rapidly create effects that resemble the ones we see in the real world, but they are not really physically based. These techniques use heuristics to approximate the physical process that we see, but they do not simulate any process from begin to end, nor they strictly follow any physical model.

Since they do not need to be physically accurate, the generation time of the process can easily achieve real time frame rates. Moreover, if they are specifically aimed towards aging or adding imperfections, they can provide a significant boost to quality and speed of the manufacturing process. This is because with very few clicks they can provide a physically plausible base over which the artist can work. After this, the user can tweak the effects and stylize them as needed. In contrast with physically based approaches, these tools provide you knobs to directly and instantly alter the final result, instead of modifying the input variables and unpredictably running the process all over again. Needless to say, this speed up in the creation process means that the pipeline will need less economic and human resources to do the same task.

As we can see, they combine both the physically based advantage of the plausibility of the results (to some extent), and the tweaking potential of the artistic approaches.

However, again, heuristic based approaches are not following any physical underlying models. This is why, under detailed inspection, we can find cases in which they differ from what would happen in reality.

In the already analyzed cases of corrosion previous works ([MDGC01] and [CS03]), we have seen that although they are using physical models, there is a part of generation that is heuristic. In [MDGC01] they use a random walk to simulate the propagation of corrosion, and they also use a set of rules to automatically decide the starting points of the corrosion. And in the case of [CS03], they use an L-system approach similar to that used for tree generation, which is heuristic as well. In both cases, these are generation processes easy to guide or control and have little physical base, but they still look very realistic.

Another example that we have already seen is [HW95], which uses an heuristic process when considering the zones less exposed to dust removal agents such as wind. They use as heuristic the exposure of each zone by tracing rays, but they do not actually simulate in any physical manner the removal agents. Consequently, this part of the process is also heuristic.

There is a program called Substance, that uses heuristic processes such as ambient occlusion or particle simulations, and lets the user modify the final results. It is very mature, and the effects that you can get with it are very realistic. Unfortunately, it is not free and it is not open source.

3. Physical Based Rendering

Since one of the objectives of the tool is that the effects are physically plausible, we need a realistic way of representing the objects in our screen. In other words, a way that let us specify materials using physical parameters, and that renders them accordingly. For this purpose, we will use Physical Based Rendering (PBR).

This section will explain the theoretical background of PBR, and how it has been implemented for the tool.

3.1 PBR theory

PBR is a set of rendering techniques and approaches that are built on top of a physical theoretical foundation. Because of this adherence to how the world works physically, it is capable of creating much more realistic images, compared to the typical old ad-hoc rendering techniques.

Specifically, it aims to solve the rendering equation in real-time. The rendering equation describes how light physically interacts with the real world, and consequently let us calculate with precision and physical plausibility the color at each pixel. This is the rendering equation for surfaces:

$$L_\mathrm{o}(\mathbf{x}, \omega_\mathrm{o}, \lambda, t) = L_e(\mathbf{x}, \omega_\mathrm{o}, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_\mathrm{i}, \omega_\mathrm{o}, \lambda, t)\, L_\mathrm{i}(\mathbf{x}, \omega_\mathrm{i}, \lambda, t)\, (\omega_\mathrm{i} \cdot \mathbf{n})\, \mathrm{d}\,\omega_\mathrm{i}$$

Where:

- $L_\mathrm{o}(\mathbf{x}, \omega_\mathrm{o}, \lambda, t)$ is the color of the pixel we want to compute. Specifically, it is the total spectral radiance directed along ω_o with wavelength λ, at position $\mathbf{x}$ and at time t.
- $\mathbf{x}$ is the position in the 3D space.
- ω_o is the direction towards which light is going out.
- λ is the wavelength of light.
- t is time.
- $L_e(\mathbf{x}, \omega_\mathrm{o}, \lambda, t)$ is the emitted radiance in the specified position, direction, wavelength and time.
- $\int_\Omega \ldots \mathrm{d}\,\omega_\mathrm{i}$ is an integral over Ω.
- Ω is the unit hemisphere centered around and facing $\mathbf{n}$.

- $\mathbf{n}$ is the normal of the surface we want to render at position $\mathbf{x}$.
- $f_r(\mathbf{x}, \omega_\mathrm{i}, \omega_\mathrm{o}, \lambda, t)$ is the bidirectional reflectance distribution function (BRDF), which will be explained below.
- ω_i is the opposite direction of the incoming light.
- $L_\mathrm{i}(\mathbf{x}, \omega_\mathrm{i}, \lambda, t)$ is the total spectral radiance directed along ω_i with wavelength λ, at position $\mathbf{x}$ and at time t.

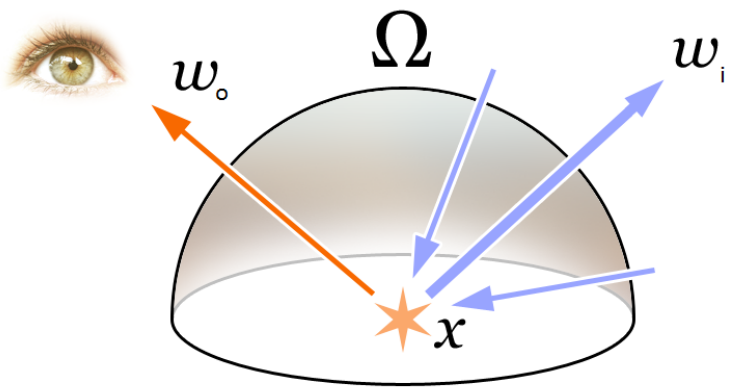

Figure 9: Rendering equation illustration.

To obtain the color for a pixel, we want to know what is the value of left hand side of the equation: $L_\mathrm{o}(\mathbf{x}, \omega_\mathrm{o}, \lambda, t)$, which is the outgoing radiance at that pixel position and in the direction of our eye or camera. Consequently, we need to compute the right hand side of the equation.

In order to get the outgoing radiance, we first find the emissive component $L_e(\mathbf{x}, \omega_\mathrm{o}, \lambda, t)$ which is zero in our case, because our surfaces will not emit light by themselves.

Later, we find an integral $\int_\Omega \dots \mathrm{d}\,\omega_\mathrm{i}$ over the normal oriented hemisphere Ω. This simply means that we want to gather all the light that is coming from all possible directions into our point. Each one of these possible incoming light directions is called ω_i, and we will be adding up all of these incoming light rays.

For each of these ω_i, we want to get the incoming radiance $L_\mathrm{i}(\mathbf{x}, \omega_\mathrm{i}, \lambda, t)$, which would be calculated by recursively evaluating again our rendering equation. Because of this recursive nature, it is very expensive to accurately calculate very realistic images. The factor $(\omega_\mathrm{i} \cdot \mathbf{n})$ simply weights the

radiance sum based on how aligned ω_i is to the normal $\mathbf{n}$, which is what happens physically (it varies with the cosine of the angle between incoming light and the surface normal, which is that dot product).

And finally we have another weighting factor $f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t)$ in front of the incoming radiance. This factor is the bidirectional reflectance distribution function, or BRDF. This function is a weighting function that outputs a number between 0.0 and 1.0. Specifically, given a position, wavelength and time, and an incoming and outgoing light direction, it tells us the fraction of radiance that will be reflected by the surface given those parameters. In our case, our BRDF will be constant in position and time, and the wavelength will be represented by RGB. This leaves us with a BRDF with two parameters: the incoming and outgoing radiance. In addition, in this BRDF we will not be taking into account subsurface scattering or other more complex interactions of light.

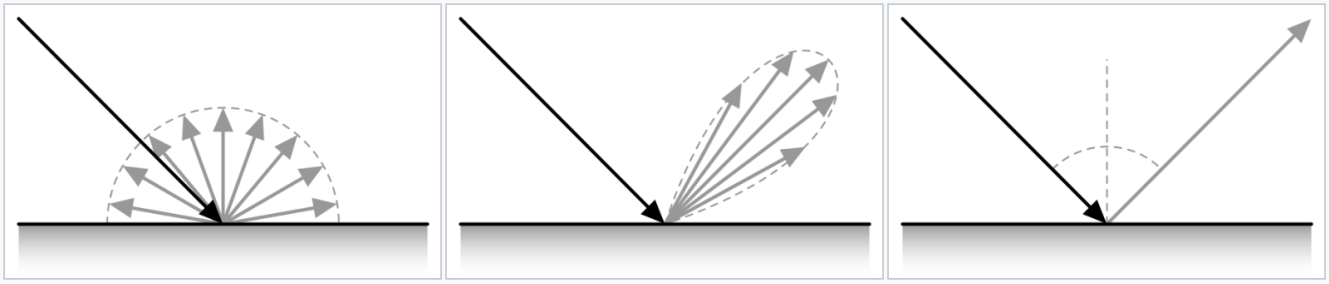

Figure 10: Illustration of 3 different BRDFs. In the left, a completely diffuse/lambertian surface. In the middle, a glossy surface. In the right, a perfect mirror.

Now, for each ω_i, we can use our BRDF to know how much light bounces off the surface into the eye direction ω_o.

We can split this BRDF into the diffuse and specular part (see Figure 11). The diffuse part represents the light that bounces off a surface in all directions, because it enters the material, scatters when colliding with its particles, and comes out again in a random direction. And the specular part represents the light that is being reflected without penetrating the surface.

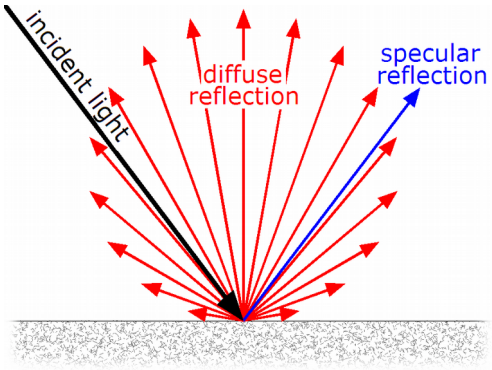

Figure 11: Diffuse and specular parts of the BRDF. [JME]

Our BRDF is based in microfacet theory, which models the surface micro geometry as many little perfect mirrors (Figure 12). All these mirrors are much smaller than one single pixel (at least in most real time applications).

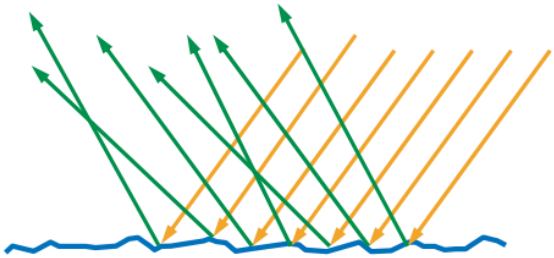

Figure 12: Illustration of a surface represented with microfacets. [SGPBR]

If these mirrors are laid out very randomly, our surface will be very diffuse, and consequently will have very blurry reflections. In the case that these micro mirrors are perfectly aligned with the normal, then we would have a perfect mirror. See Figure 13 to see a real life comparison of a smooth and a rougher surface.

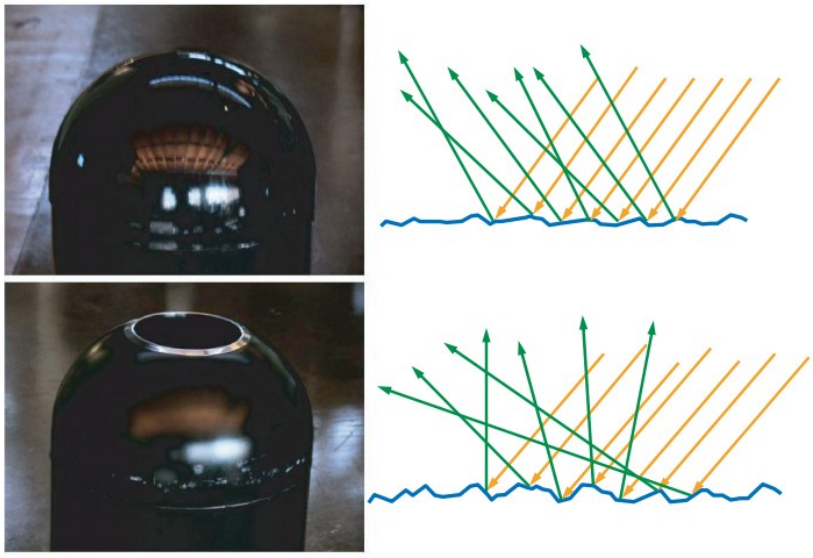

Figure 13: Microfacet view of a smooth surface vs. a rougher one. [SGPBR]

Consequently, we can represent the BRDF as a sum of these diffuse and specular parts, weighted by how diffuse or specular a surface is:

$$f_r = k_d \cdot f_{diff} + k_s \cdot f_{spec}$$

With this, we can split our integral in a diffuse and a specular part, which will help us with the implementation of PBR.

The diffuse part of our BRDF can be computed by using the lambert BRDF, which weights each incoming light direction evenly. This is very easy to calculate, we just have to do an average of all the incoming light in the hemisphere.

The specular part of our BRDF is more difficult to calculate. There are several models to compute it, but one of the most used is the Cook-Torrance BRDF specular model. This model tries to represent how light reflection would interact with all our microfacets. Since these are much smaller than a pixel, we have to use statistical models to determine how light interacts with all the microfacets that are contained in each single pixel.

Since we want to know what proportion of the incoming light is being reflected to our eye, then we are interested in knowing how many of the microfacets are oriented in such a way that rays of light bounce off to our eye. The microfacets that will reflect light rays into our eye are those with a normal facing the half vector (H), which is the vector bisecting the incoming and outgoing directions. We can see it in Figure 14.

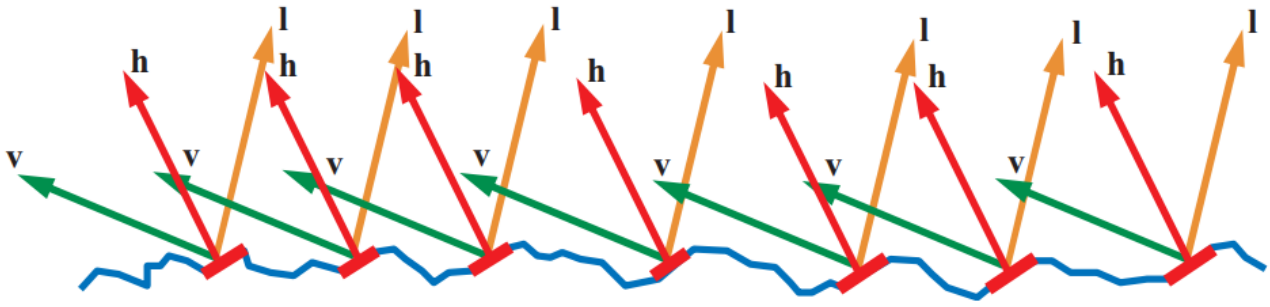

Figure 14: Illustration of microfacets facing H, and how incoming light reflects directly into the eye V when hitting them. [SGPBR]

The Cook-Torrance BRDF model will not take into account more complex interactions between microfacets such as multiple light bounces (see Figure 15) or diffraction. Since it would make the model more difficult, it is an assumption that is made in order to simplify it. Nonetheless, it still gives accurate results for most cases.

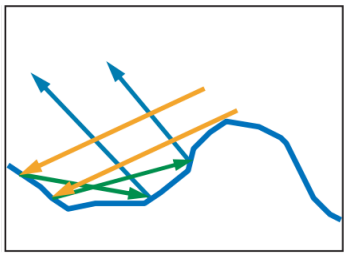

Figure 15: Multiple light bounces. [SGPBR]

The formula for the Cook-Torrance BRDF is the following:

$$f_{\mu\text{facet}}(\mathbf{l}, \mathbf{v}) = \frac{F(\mathbf{l}, \mathbf{h})G(\mathbf{l}, \mathbf{v}, \mathbf{h})D(\mathbf{h})}{4(\mathbf{n} \cdot \mathbf{l})(\mathbf{n} \cdot \mathbf{v})}$$

We can see that the numerator it is a combination of three different terms F, G and D. The denominator is just a collection of normalizing factors.

The factor $F(\mathbf{l}, \mathbf{h})$ is the factor representing the Fresnel Reflectance. It computes the fraction of light that is reflected from an optically flat surface. It models how light reflects based on the reflection and refraction of different materials. For example, from it we can deduce that the fraction of reflected light increases as the view angle becomes more shallow, as we can see in Figure 16.

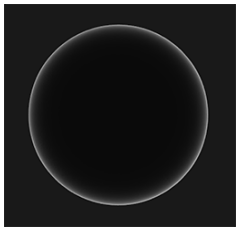

Figure 16: Visualization of the Fresnel reflectance on a sphere. [LOGL]

Instead of computing the full formula for it, we can come up with a good approximation that is cheaper to compute. This is the Schlick approximation, and is widely adopted:

$$F_{\text{Schlick}}(F_0, \mathbf{l}, \mathbf{h}) = F_0 + (1 - F_0)(1 - (\mathbf{l} \cdot \mathbf{h}))^5$$

Where F_0 is the base Fresnel reflectivity at angle 0º, which varies from material to material.

The next factor is $G(\mathbf{l}, \mathbf{v}, \mathbf{h})$ which is called the geometry function. It represents the probability that a microfacet is seen by both the view and the light vector. It takes into account masking and shadowing (see Figure 17). We expect a rougher surface to have less specular light bouncing into our eye because of these masking and shadowing effects.

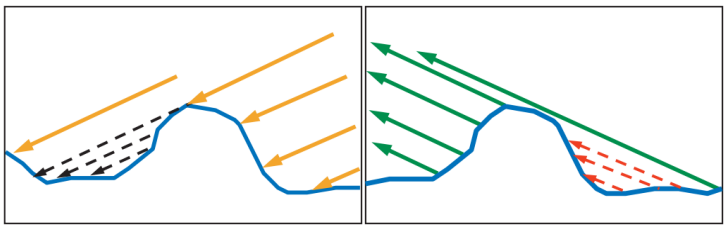

Figure 17: Microfacet shadowing and masking. [SGPBR]

This can be approximated by multiplying two times the G_{sub} function. One for the view vector, and another for the light vector:

$$G(n, v, l, k) = G_{sub}(n, v, k)G_{sub}(n, l, k)$$

And we can use the most typical G_{sub} function which is called GGX Schlick:

$$G_{SchlickGGX}(n, v, k) = \frac{n \cdot v}{(n \cdot v)(1 - k) + k}$$

Where *k* is a remapping of the roughness of the surface.

We can see the effect of the geometry function over increasing roughnesses in Figure 18.

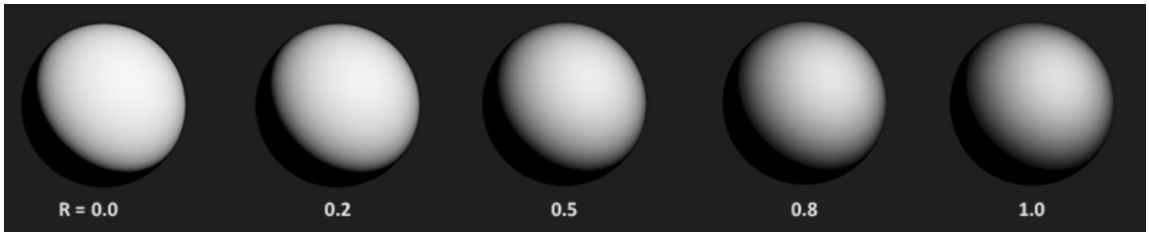

Figure 18: Geometry function over increasing roughnesses. [LOGL]

And the final factor is $D(\mathbf{h})$, which is the normal distribution function (NDF). The NDF represents a statistical approximation of the relative surface area of the microfacets facing *h*. One of the most used functions and the one that will be used for this tool is the Trowbridge-Reitz GGX function:

$$NDF_{GGXTR}(n, h, \alpha) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2}$$

Where α is the roughness of the surface.

We can see the effect of the NDF over varying roughness in Figure 19.

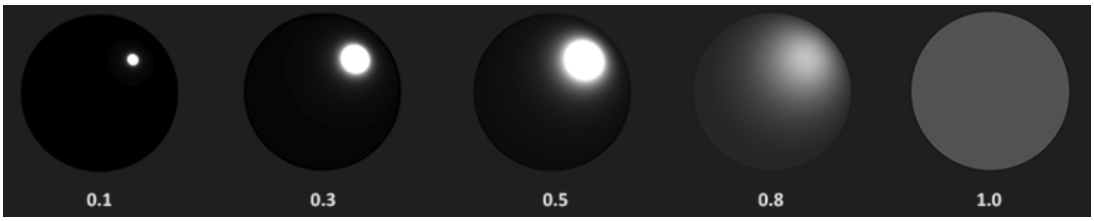

Figure 19: NDF over increasing roughness. [LOGL]

3.2 PBR implementation

Now that we have some theoretical background on how the physical based models work, we can implement it. The implementation will have to evaluate the rendering equation in real time, so that we know which color we have to put in each pixel.

For this, we will first use a technique called Image Based Lighting (IBL), which lets us represent the environment light as a set of images arranged in a cubemap, such as the one in Figure 20.

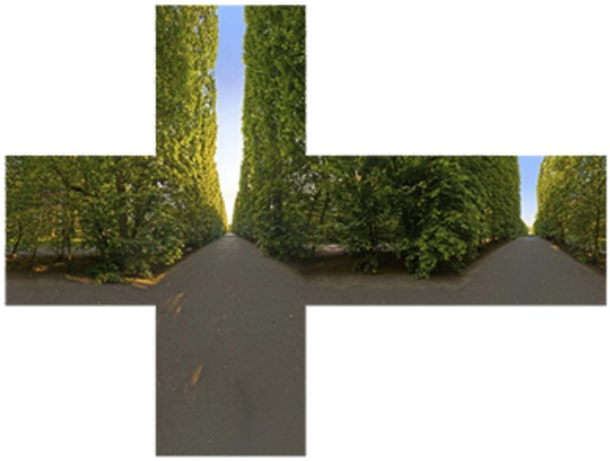

Figure 20: Cubemap representing the surrounding environment so that we can retrieve the incoming light from it. [LOGL]

This will let us retrieve the light around us with a single texture lookup. The environment light is very important for PBR, because it is a key component in the rendering equation integral. This will help very much with the implementation. However, this will assume that all points are being evaluated in the center of the cubemap. In addition, this PBR implementation will not take into account the

recursive bounces of light between the different objects composing the actual scene. This is usually worked around by using reflection probes, but this is not important for the scope of this thesis.

The expensive part of evaluating the rendering equation is integrating the light over the normal oriented hemisphere. However, we can precompute it if we make some assumptions. Remember that we will need to precompute two different integrals: the diffuse part, and the specular part.

The diffuse part of our integral can be integrated in a single cubemap called the irradiance map. With it, we can pick the diffuse integral value with a simple cubemap texture sample. The precomputation process discretizes the integration method into a Riemann sum over a series of points or sections around a sphere. The more points we use, the closer the precomputation will be to the actual continuous integration value.

Since it produces very low frequency images, we do not need many precision nor resolution for the irradiance map. If we apply this precomputation to the environment map in Figure 20, we will get the irradiance map shown in Figure 21.

Figure 21: Example of precomputed irradiance map. [LOGL]

And finally, we will have to compute the specular part of the integration. This is not as easy as the diffuse part, because we can not use a single image, because it depends on many input variables. However, we can still approximate it well by precomputing the specular integral into a cascade of cubemaps, as in Figure 22.

Figure 22: Cascade of precomputed specular integrals. [LOGL]

And with these sets of cubemaps, we have a good approximation of our integral in the rendering equation.

These previous steps were for the precomputation of the environment lighting. In addition, we will have to evaluate in real time the BRDF for each extra dynamic light in our scene (directional lights, omnidirectional lights, spot lights...), but this is much simpler because we will not need to integrate anything, and thus can be directly evaluated in run time. This is thanks to the fact that all the outcoming light for these types of lights is being generated from a single point. If we needed to add area lights, then things would get more difficult, but thankfully we will not use them for our tool's viewer.

With this, we are ready to provide a physically based rendering that will help our users visualize in a realistic way the effects they build with our tool.

4. UnClean3D development

For this thesis we have developed a tool called UnClean3D, which aims to solve the previously stated problems, while trying to feature as many advantages of both the physically and artistic based approaches.

It is an open-source project under MIT license, so that it can be used wherever needed, by whoever needs it and for whatever may be needed. It can be found on GitHub:

https://github.com/sephirot47/UnClean3D

4.1 Objectives

It is very important that we have in mind the objectives we want to reach with this thesis. The implemented tool will aim to achieve the following goals:

- **Tool specifically crafted to create different aging, dirt and imperfection effects:** We want to create a tool that is specifically made to fit the needs of rapidly adding imperfections to a model. It has to be a simple and effective software that anyone can use to rapidly add feasible and good looking imperfection effects to models. We want it to be able to create typical aging effects.
- **Real time feedback**: In the process of creation of the effects, it is really important to provide the user an interactive experience. Each tweak the user does, must result in an instant change in the viewport, so that the work-flow can be as fast-paced as possible. This means that we will have to build a tool containing a viewport in which users can instantly see how the model is being affected by the aging effects.
- **Physical plausibility of the effects**: Another important objective will be to produce effects that seem physically plausible to the human eye. Despite the fact that none of the used techniques will be based in physical models, they will still try to give a result that resembles the reality as close as possible, so that our eyes can not notice the difference. Obviously, under close inspection or comparison with a reference, you might be able notice that the effects are not physically based. Nonetheless, in most fields this is not needed, not to say the ones scoped by this thesis, and consequently we can consider this plausible approximation as a good target. This physical inaccuracy is a trade-off that must be made in order to fulfill the aforementioned real-time feedback objective. Otherwise, it would take very long to compute some effects at each change, and real time feedback would be very difficult to reach.
- **Minimal performance footprint on the modified model:** We want the generated effects to be very lightweight and add a minimal overhead/footprint to the complexity of the model. This is

very important because we want our tool to be useful for asset authoring of real time applications such as video games, in which it is very important that 3D models are optimized, lightweight and very fast to render. Because of this, we will have to make sure that our tool keeps things simple and makes use of techniques that will not increase the asset complexity, while at the same time producing convincing and realistic effects.

- **Modularity and flexibility:** We do not want to have a completely closed implementation of the tool. It would be beneficial to provide a modular system which can be easily extended, or whose pieces can be used and combined between them in a creative way to come up with new effects or usages.
- **Integration with well-known game engines and 3D software:** It is essential that the results of the software can be easily integrated and moved back and forth between the tool and external game engines or 3D software. This way, all users will be able to add it to their pipeline and work-flow with minimum effort.
- **Letting artists easily modify the final results, to match the desired style:** Artists frequently need to make some final touches to the results, so that they match their initial design or idea. For this, we want to provide easy and straightforward knobs to adjust each effect, and modify the final outcome.

  In addition, to constrain users as little as possible, we need to also set as an objective the creation of a brush-like tool, so that they can paint the model or modify the effects with it. This will provide a method to overcome the limitations of the provided heuristic procedural effects, in case users do not find one that suits their needs. Furthermore, we want brushes to be flexible enough so that artists can create their own brushes and add them to the software in an easy way.

4.2 Effects to achieve

Now that we have set the general objectives of this thesis, we can focus on deciding which effects we would like our tool to be able to create. There are many different types of aging or imperfection effects. However, some of them are not easily perceivable, and some other are not very used or typically seen in video games or real time applications scenes. This is why we will build our tool giving more importance to those more used and noticeable effects. Nonetheless, since we are building a modular and generic tool, artists can still come up with ways of creating new effects by combining the provided mechanisms.

Consequently, we will focus on providing the needed modules or mechanisms to easily build the following effects.

Rust

Rust is a very well known aging effect often seen in metals, that makes surfaces go brown and deteriorate. This is an example of a very used aging effect in old scenes that gives them a more realistic appearance. See Figure 23 and Figure 24.

Figure 23: Rust on a screw and piece of a chain.

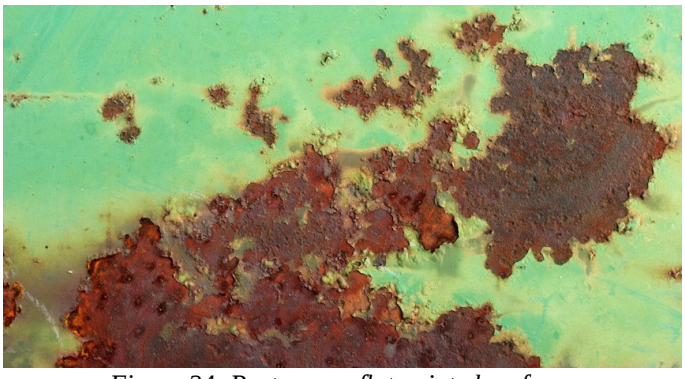

Figure 24: Rust over a flat painted surface.

Peeling

Peeling is an effect caused when an object is made of two layers, and one of them is lost, leaving the layer underneath visible. It can be combined with other effects such as rust to give them more realism. See Figure 25.

Figure 25: Peeling effect on a wall. See how the layer beneath is visible through the holes in the upper one.

Mold and moss

Mold is a fungus that grows specially in zones that are warm and have high humidity. And moss is a green plant that also grows in zones with high humidity. They appear in walls and other objects in environments with those conditions. In old scenarios it is typical to see them in corners of walls for instance. See Figure 26 and Figure 27.

Figure 26: Mold sticked on a wall.

Figure 27: Moss in a building made of rock.

Scratches

Scratches appear due to the interaction of the surface with tools or other objects over time. They are seen everywhere, and the object does not need to be extremely old to have them. Because of this, they are essential in order to increase the realism of any scene. See Figure 28 and Figure 29.

Figure 28: Scratches in a car.

Figure 29: Scratches in a table.

Surface irregularities and dents

Surfaces are usually not perfectly flat. This is very noticeable under specific light conditions or with specular highlights. Surface irregularities are present when surfaces are a little bit wavy. Dents are usually seen in cars or other thin surfaces that have received a hit with a blunt object (see Figure 30). Both surface irregularities and dents can be added to objects or surfaces so that the light interaction is more real and they do not seem perfectly smooth.

Figure 30: Dent in a car.

Humidities

Humidities come up in walls very often. They usually bump the surface up, and they may also change its color. This effect can help to give realism to walls in interiors for example. See Figure 31.

Figure 31: Humidity in ceiling.

Dirt and dust

Dirt usually come up in all kind of surfaces, and tends to gather in occluded zones. Dust can be present in surfaces that have not been cleaned over several days. See Figure 32 and Figure 33 respectively.

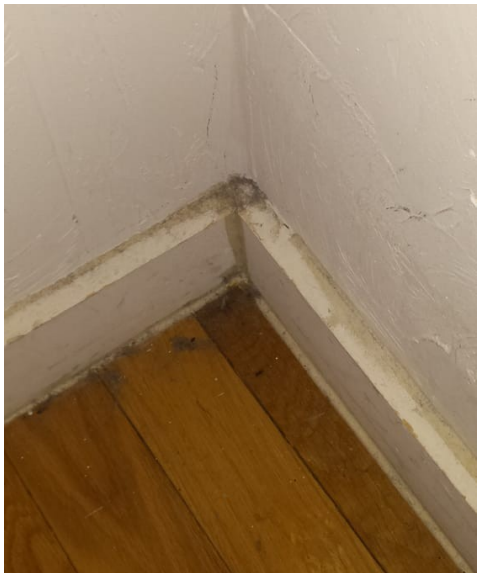

Figure 32: Dirt in corner of a room.

Figure 33: Dust over a furniture.

Color loss

Color loss can be due to chemical or physical processes. One typical example is an object that has lost part of its color due to a constant exposure to sun light over some time. Another example would be a book whose pages become yellowish. We can add this effect to some exterior objects if we know they will be exposed to sun light. See Figure 34.

Figure 34: Color loss of a newspaper.

Broken edges

Broken edges are very common in furniture, rock and objects made of hard materials that can be easily fractured. These can be added to all sort of objects, specially if we know they may have been moved around or manipulated (and consequently with high probability of being damaged). See Figure 35.

Figure 35: Broken edges in a rock table.

4.3 Texture-only effects

When using PBR, we usually use a set of textures to define the different physical properties of the surface at each point.

All the effects of UnClean3D will be exclusively applied to a model by modifying its PBR textures. This means that when exporting a modified model from the tool, the only thing that will differ from the original one will be its PBR textures. In other words, the mesh will not be modified at all.

There are the five PBR textures that will be modified:

- **Albedo color**: this texture represents the base color of the material at each point of the model surface. In this case, it represents the amount of reflected RGB light in diffuse reflections. It will change depending on the material light absorption properties. For instance, if the albedo color is red, we will expect to perceive as red the color of the surface under white light. In Figure 36 we can see an example of an albedo texture, and how it is applied to its 3D model.

Figure 36: On the left, an albedo texture. On the right, that albedo texture applied to its 3D model.

- **Roughness**: this texture represents the roughness of the material at each point of the model surface. From the point of view of micro-facet theory, the roughness tells us how smooth or rough is the microfacet distribution of the surface. The higher the roughness, the higher the disparity of orientations in adjacent microfacets. This has a direct impact on how light is reflected when it hits the surface. On one hand, very rough surfaces will tend to reflect environment in a blurry way. On the other hand, very smooth surfaces will be similar to a mirror, with almost perfect reflections, as in Figure 37. To see the roughness texture applied to a model with the rest of PBR textures, see Figure 42.

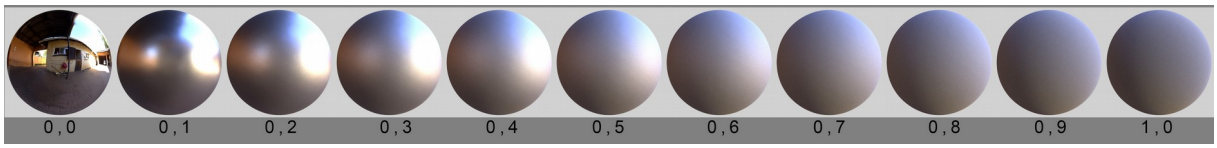

Figure 37: From left to right, perfectly smooth surface to extremely rough surface.

Figure 38: Surface microfacets. Above, a considerably smooth surface. Below, a quite rough surface.

- **Metalness**: this texture represents the metalness of the material at each point of the model surface. The metalness simply tells us how metallic the surface is. This, again, has a direct impact on how the surface reflects the light. For very metallic surfaces (conductors), due to their internal structure, they will absorb most of the light that penetrates the surface. Consequently, the diffuse reflection will not come out of the surface very much, and thus we will only perceive the specular reflection. To see the metalness texture applied to a model with the rest of PBR textures, see Figure 42.

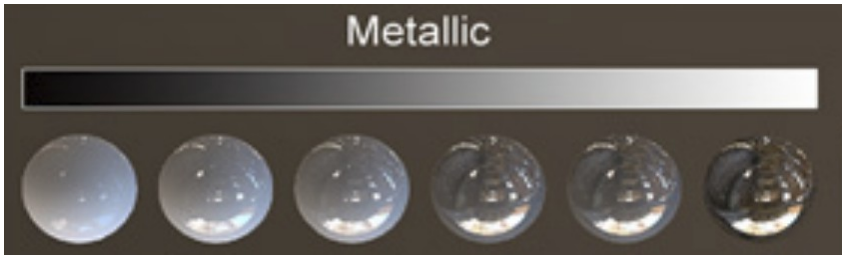

Figure 39: From left to right, not metallic material to "very metallic" material.

- **Normal texture**: this texture represents the normal at each point of the model surface, in tangent space. These are useful to fake extra geometry on the model, because the lighting changes thanks to this normal variation. Consequently, using them we can add extra detail at very cheap cost, without the need to modify the actual surface geometry. However, the illusion breaks when viewing the surface at very shallow angles, because you can notice the missing parallax and occlusion, and at boundaries you can see that there is no real geometry. Check Figure 40 to see the effect of a normal map on a flat plane.

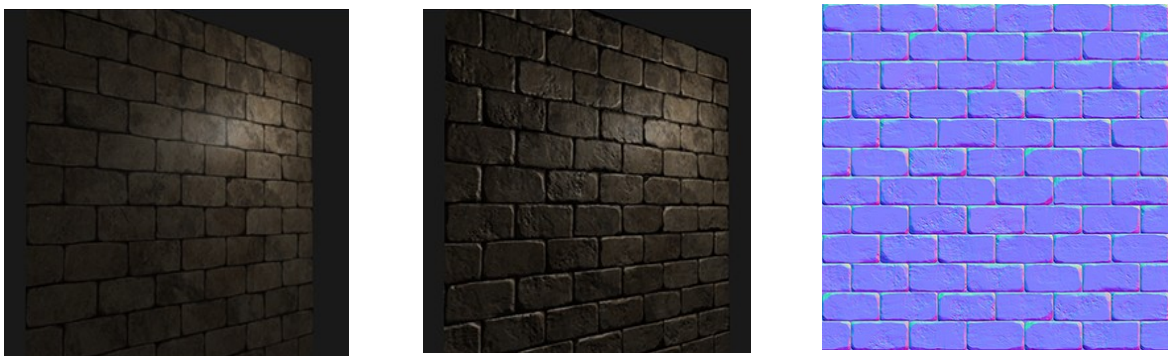

Figure 40: From left to right: without normal map, with normal map, and the applied normal texture.

- **Height texture**: this texture represents a displacement of height at each point of the model surface, normal to it. This is useful to implement displacement maps or more complex techniques than normal textures for adding extra geometry to a surface, such as relief mapping or displacement mapping using tessellation shaders. However, although we will generate and export height textures, the renderer will not apply relief nor displacements to the mesh to represent them. Nevertheless, they can still be useful for rendering if the game engine to which we export them supports any of these more complex techniques. Consequently, we will just use them as an intermediate step to generate the normal texture.

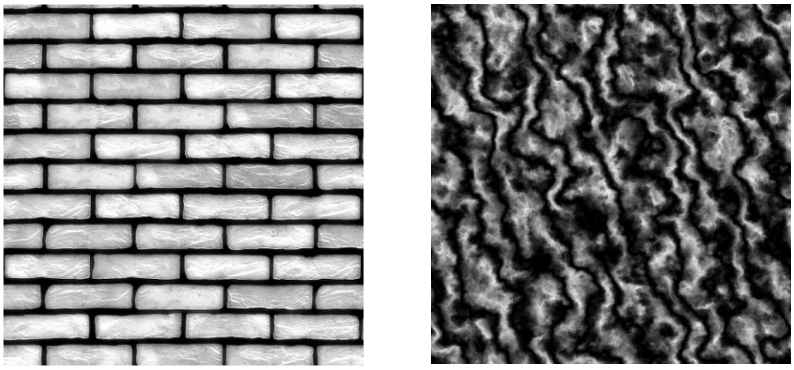

Figure 41: Examples of height textures.

With all these textures, and using our PBR implementation, we can get very realistic effects. In Figure 42, we can see how by combining 3 simple PBR textures and using the physical based rendering techniques, we can get a very convincing rust effect. We can also see how the light interaction and

reflections vary across the surface, depending on metalness and roughness. The highlights disappear in corroded zones, as we would expect in the real world.

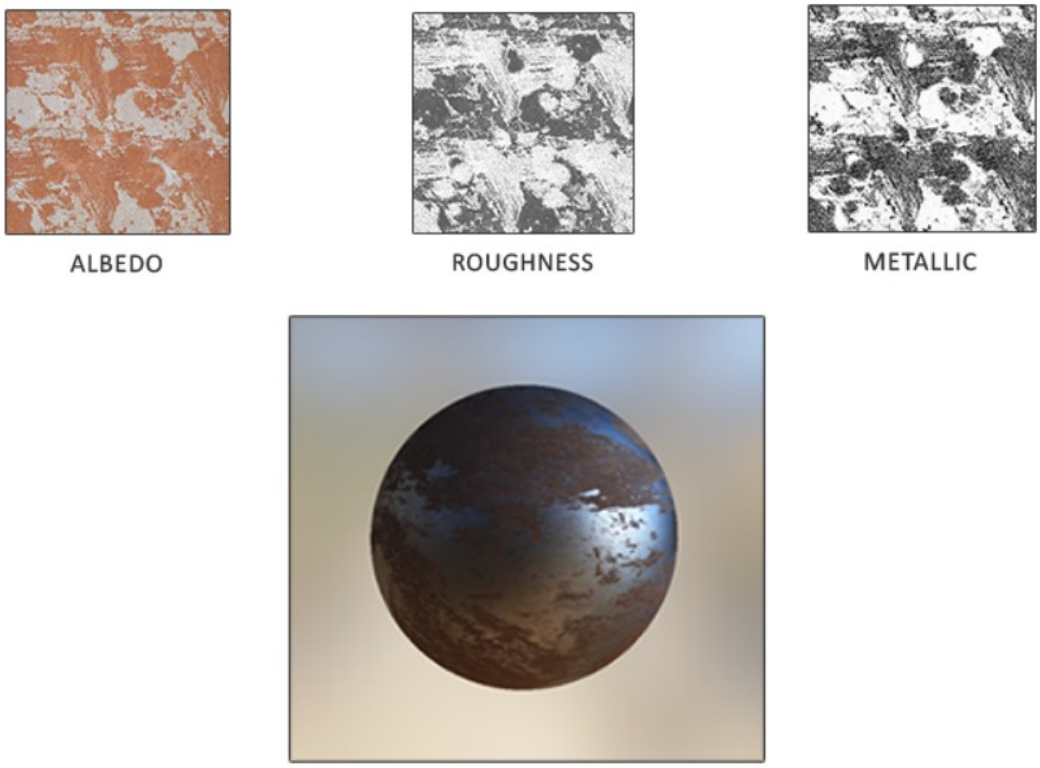

Figure 42: Example of PBR textures applied to a model. [LOGL]

4.4 Why texture-only effects

The decision of constraining the tool to modify only the aforementioned 5 PBR textures has its advantages and limitations. We decided to do it this way to achieve the established objectives of this thesis.

On the one hand, using effects only applied to the PBR textures has many advantages:

- Modifying only the PBR textures means that we will add very low or no performance hit when the model is rendered. In case the software was already using all the PBR textures and normal mapping when rendering the model, then the cost remains the same as before. However, if we add some PBR texture to a model that did not have one, then we will be increasing the rendering time of the model, but only slightly. In any case, we will never be adding new geometry to the model. This will help us reach the real time frame rate we are targeting. Consequently, it will be appropriate to adopt this strategy since we want to meet the objective of creating a tool that can be used for video games or other interactive applications.
- Using only the PBR textures makes it really easy to combine many effects for the same model. Specifically, we can create layers of effects by composing our modified PBR textures one above the other. Thanks to this, we can achieve the objective of creating a very modular and flexible tool.
- Since these textures are a de facto standard across the industry, they can be easily imported from other material edition software, or exported to finish some details after using it. In addition, most well-known game engines such as Godot, Unity or Unreal also use these PBR textures to represent the physical properties of materials. Thanks to this, exported models can be seamlessly integrated with them. There is no need to even restart the game engine or reimporting the asset, since they will detect that these textures have changed and automatically reload them. This way, the tool can be smoothly adapted to each pipeline work-flow, as we wanted in another of our objectives.
- By constraining ourselves to texture-only effects, we avoid tackling with all the difficulties that mesh problems pose. Mesh processing for some effects would be very troublesome and would add many geometry that the artist may not want.
- We can easily increase the quality and resolution of the effects just by increasing the texture size. However, we must be careful with this, because we can not arbitrarily increase the texture size, since it can have a negative impact in performance and all GPUs have a texture size limit.

But on the other hand, there are still several limitations to this approach:

- One of the weak points of using only textures, is that the normal mapping illusion can be easily broken when seeing the mesh at shallow angles (see Figure 43). However, since we are providing a height-map, a more advanced technique like relief mapping ([POC05]) or displacement mapping ([MNCL13], [KMMM02]) with tessellation can be applied to overcome this limitation (at a greater cost of course).

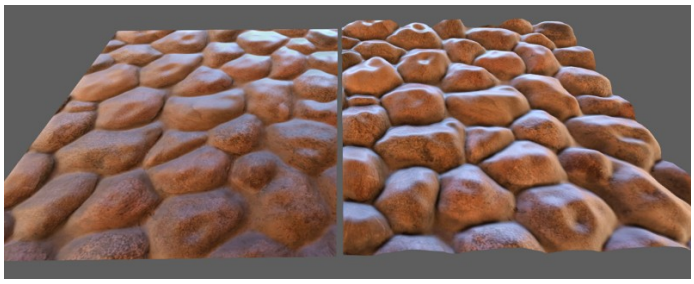

Figure 43: On the left we can see how normal mapping breaks the illusion at flat angles. On the right relief mapping solves it.

- Another problem that we have with traditional texture mapping for these PBR effects are the artifacts that appear near texture seams. These seams are almost always present because textures are mapping our 3D surface to a 2D surface, which can not be easily done for the great majority of surfaces. In most cases, this means that the artist has to mark some seams in the model in order to be able to “stick” the texture properly. Due to the texture sampling process when rendering, a sample of a texture may need to access several surrounding texels, or even access prefiltered smaller versions of that same texture (mip-maps). These sampling techniques create problems when retrieving a texture value near these seams, because such algorithms assume that the information around a texel is coherent with it, which does not happen with texels near the seams.

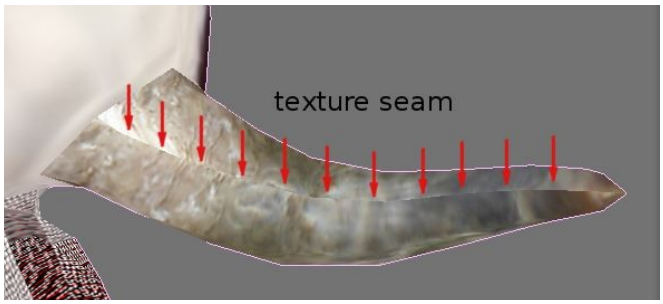

Figure 44: Problems around texture seams.

Due to this, we will have to apply some techniques to fix these artifacts, or at least alleviate them.

- Since this tool wants to be easily adaptable to any pipeline, it will use the texture mapping (uv layout) provided by the user. This however can be problematic.

  In some cases, artists map the same zone of a PBR texture to different zones of the 3D surface. For example, imagine we are texturing a character, and in its design we can see that the colors of the left and right side of the face are identical. We could take advantage of this symmetry and just paint into the texture the colors of one half of the face. Then, we would be able to specify the texture coordinates of the left and right sides of the face mesh in such a way that they use the same texels but symmetrically, so that the final result still looks the same as if we had painted into the texture the whole face (see Figure 45). Thanks to this technique, we can save a considerable amount of texture space.

Figure 45: Example of texture mapping that reuses texels. See the face on the left, only half of it is being painted.

  However, the effects that UnClean3D will apply to the model will mostly be in model space instead of texture space. This means that it will apply the effects in a 3D space, and we will be assuming that each triangle can be independently textured. If an input model happens to reuse the same texels for several triangles, then the effects will not be applied correctly, and we will end up with only one of these triangles looking good, while the others will have a texturing that does not make sense.

  This can be a serious inconvenience for some models. Artists can overcome this by not applying effects on those triangles, or by unfolding the symmetries in the PBR texture beforehand.

4.5 Layer-based architecture implementation

During the development process of UnClean3D, there have been several iterations on the architecture of the program. We will go through each version that has been implemented, and why each improvement had to be done. At the end, we will explain the final version in more detail, in order to better understand the tool work-flow and definite layer architecture.

Layer architecture iterations

Architecture iteration 1. Static CPU effects.

This was the first attempt of architecture. Here, the effects consisted of fixed CPU processes that modified the PBR textures without any possibility of parameterization. The final result was made of sequentially applying each of these effects one on top of the other. Each of these separate effects is called an effect layer.

Despite the fact that this first implementation was parallelized with the CPU, the performance was extremely far from interactive frame rates. Although the setup and programming was easier with CPU effects, we had to handle all the intermediate steps as rasterization or barycentric coordinates interpolations. With all this work on each frame it was impossible to provide real time frame rate, even when using acceleration structures such as quadtrees. This made the tool very difficult to use, and the user had to wait for every effect to be computed.

Due to these limitations, the CPU implementation of the effects did not achieve the stated objectives, and this is why using the CPU for the generation of the PBR texture effects was discarded.

Architecture iteration 2. Static GPU effects.

The next step was moving all the effects and pipeline to the GPU. This required much work, because we had to set up all the GPU communication, shader creation, and moved all algorithms to GLSL.

Moving everything to the GPU means that many texels can be processed in parallel. In addition, all the work of rasterization and interpolation is being done by the GPU, which is extremely fast and optimized to do such job.

This provided a huge boost in performance, and thanks to it the effects could be generated in real time without any problem.

However, we was not still providing any way of modifying the effects. They were always being generated in the same way, and with the same parameters. This led to the next iteration.

Architecture iteration 3. Modifiable GPU effects.

In this step we programmed all the needed UI and serialization in order to be able to add parameters to each effect. The ability of tweaking the effects from the UI, provided us a way of testing that the effects

were indeed being generated in real time. Thanks to this, it became clear that the decision of moving everything to the GPU was essential for the usability of the tool, since being able to instantly see the result of the effects applied to the model makes it much easier and comfortable to create them.

Architecture iteration 4. Modifiable GPU effects with a single mask.

Although being able to modify each effect separately was working well, it still lacked flexibility. We decided to split each effect layer into two elements:

- **A mask**: the mask defines where the effect layer is going to be applied. It is a grayscale texture that specifies the zones of the 3D surface in which the actual effect is going to be painted. If a texel in the mask is white, we will apply the effect at 100%, if it is black at 0%, and all gray values in between.
- **Effect parameters**: the effect parameters tell how each of the PBR textures is going to be modified by this effect layer at each texel specified by the mask. It is composed by 4 properties:
  - **Color:** determines the albedo color at each point where the effect is being applied.
  - **Height:** determines the height at each point where the effect is being applied.
  - **Roughness**: determines the roughness at each point where the effect is being applied.
  - **Metalness**: determines the metalness at each point where the effect is being applied.

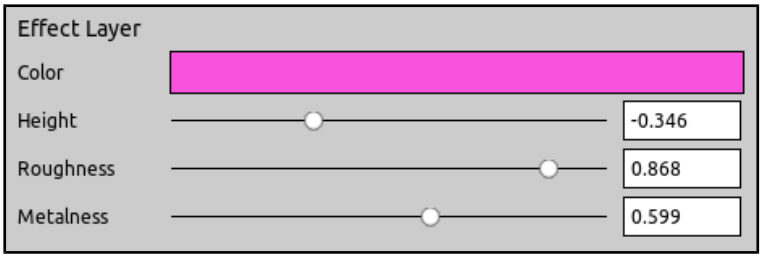

Figure 46: Different effect parameters of an effect layer: color, height, roughness and metalness.

Consequently, this means that, for each effect layer, we have one mask that tells where to apply the effect, and the effect parameters which tell how color, height, roughness and metalness must be modified where that mask specifies. In other words, with the mask we can specify the zone in which we want an effect to be applied. And we use the effect parameters to tell how we want this effect to modify the color, height, roughness and metalness of that masked model surface.

This helps making the architecture more modular and flexible, preparing it for the following final architecture.

Architecture iteration 5. Modifiable GPU effects with several masks.

Finally, this is the current architecture. This last iteration consisted in providing as many masks as needed for each effect layer, instead of having just one single mask per effect layer. In addition, we created blend modes (Add, Subtract and Multiply), which determine how masks should be blended with other masks. The same blend modes were also implemented for effect layers themselves, to tell them how they should blend with one another.

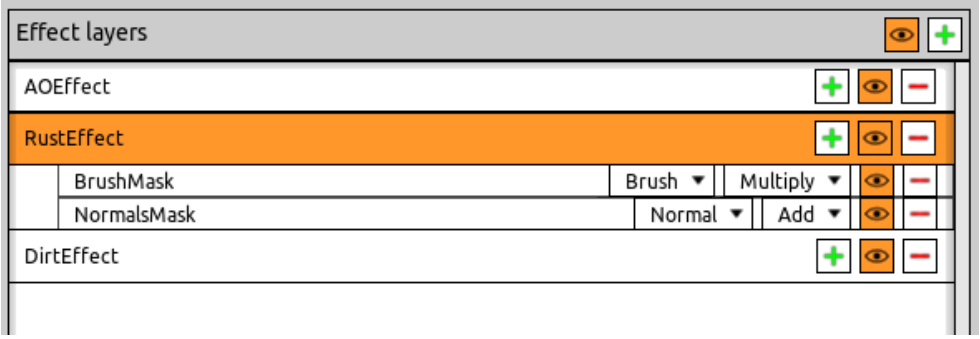

Figure 47: Final layer architecture of UnClean3D. Here you can see several effect layers. The RustEffect has 2 masks, which will be combined (with Multiply) to yield the final layer mask for that effect layer.

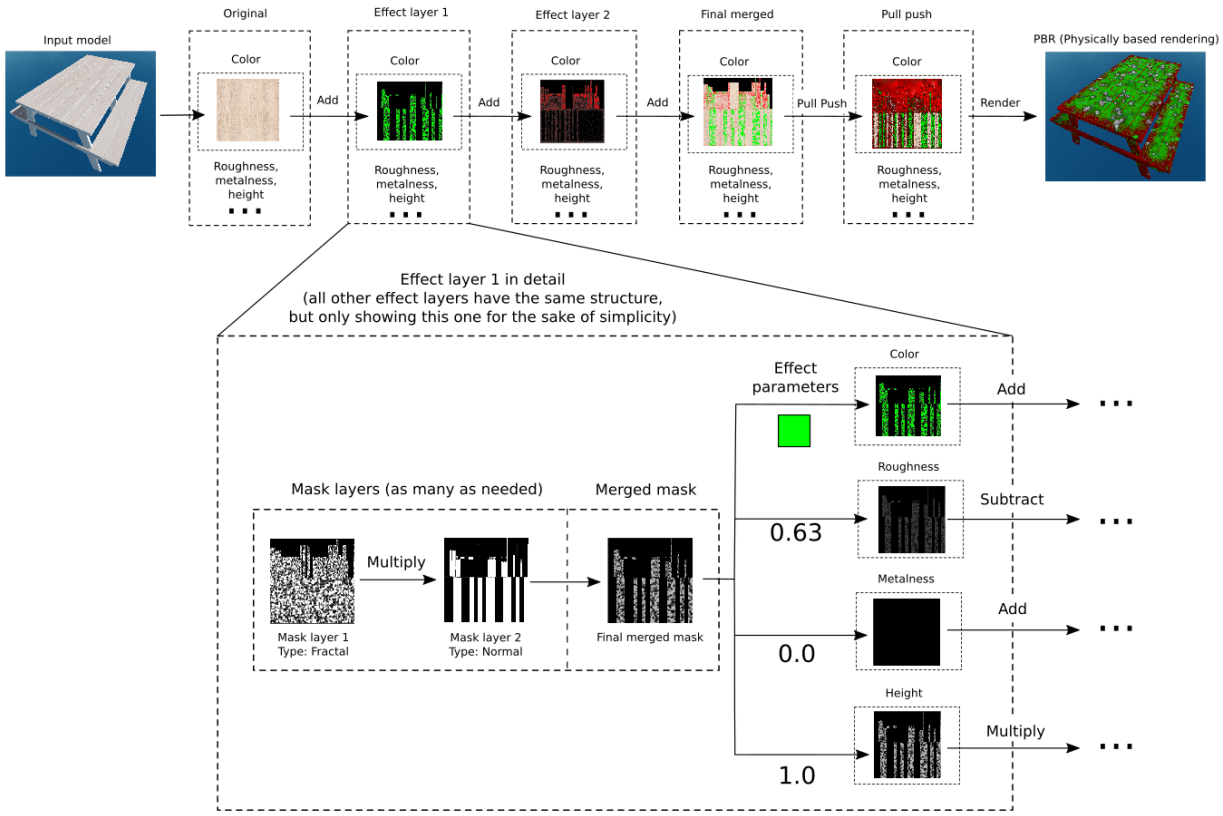

Figure 48: Final layers architecture scheme. In the upper row, we only see color, but the same pipeline is applied to all other PBR textures (roughness, metalness and height), as we can see in the detail of effect layer 1.

In Figure 47, we can see an example of several effects applied to a model. In this case, we have 3 effects: *AOEffect*, *RustEffect* and *DirtEffect*. If we select *RustEffect*, then we can see it contains 2 masks: *BrushMask* and *NormalsMask*. Each of these masks has a control to select the type of mask (in Figure 47 we see Brush and Normal mask types) and another to select the blend mode with the mask below (Multiply, Subtract or Add). In this case, to get the final mask for *RustEffect*, we will pick *NormalsMask* as the base mask, and then we will multiply *BrushMask* on top of it.

You can see an scheme of this final layer architecture in Figure 48, where you can see what is the complete tool architecture and process from beginning to end.

4.5.1 Masks creation base algorithm

Each mask has its own type. There are different types of mask and each type has its own parameters.

But despite all mask types are different, they all use the same base algorithm to be generated:

- First, we define a viewport that is as large as the PBR textures we want to modify.
- Resize the mask texture to match the PBR textures size, if needed.
- Bind a framebuffer to draw into the mask texture.
- Bind the GLSL shader of our effect. This is what changes between mask types, each one has its specific implementation that will determine the masking amount of each texel.
- Bind a mesh formed by each triangle with its texture coordinates. Each vertex can also have interesting data such as triangle position or normals in model space, which we can later use in our shader. The GPU will interpolate them for us to each texel.
- Render the bound mesh. As said before, the mesh is 2D and is made of all the triangles in their uv/texture layout (this means that we have 2D coordinates triangles). This will effectively paint each triangle into the mask texture. The shader will determine how to mask each texel.

After doing this for every mask of the effect layer, we will merge all of them using the specified blend mode. This final merged mask will be the one used to generate its corresponding effect layer result.

4.5.2 Effect generation algorithm

Once we have generated a single merged mask for the effect layer from all its masks, we can actually generate the effect. This means that now is when we will paint/modify our PBR textures of color, height, roughness and metalness. This process is easier than the mask generation. In this case, we will apply a texture pass painting over each PBR texture, while taking into account the final merged mask of each effect layer.

For example, in the case of the PBR texture of color, we will apply the specified effect color depending on how masked the texel is. If the mask texel is completely white and the blend mode is Add, we will overwrite the color on the PBR color texture with the effect color. But if we find a mask texel that is completely black, we will just ignore it and not modify the PBR color texture, since the mask is telling us that the user does not want to paint this texel. And for gray mask values, we would do a weighted blend accordingly to this gray value and blend mode.

Analogously, with height, roughness and metalness we will modify their grayscale PBR textures according to the mask value.

After all effect layers have been generated, the last step of the process is merging all of them in one PBR texture for each PBR property: one final PBR texture for color, one for height field, one for roughness and another one for metalness.

Then, after this step, we will have our 4 almost-final PBR textures. The final step will be to apply the pull push technique to them.

4.5.3 Pull push

After generating the aforementioned PBR textures, we will apply a technique to fix one of the issues caused by using texture-only effects: seams artifacts.

The idea is to process the texels near the seams in some way to reduce these artifacts. For this, we will apply a technique called pull push [KM09], that can fill the gaps of missing information of a texture. This technique proceeds by generating a cascade of downscaled images. At each downscaling step, we will just take into account those texels with information, ignoring the ones without any data. We do this until we arrive to the final single pixel image. Afterwards, we begin to upscale this pixel image level by level until we reach the original image size. With this upscaling process, we will be able to fill the problematic texels that had no information, since we have the corresponding filtered texel generated when downscaling in cascade.

This process works very well for color, making all seams disappear in a very clean way.

In the Figure 49, we can see a texture before applying pull push, and the same texture after applying pull push.

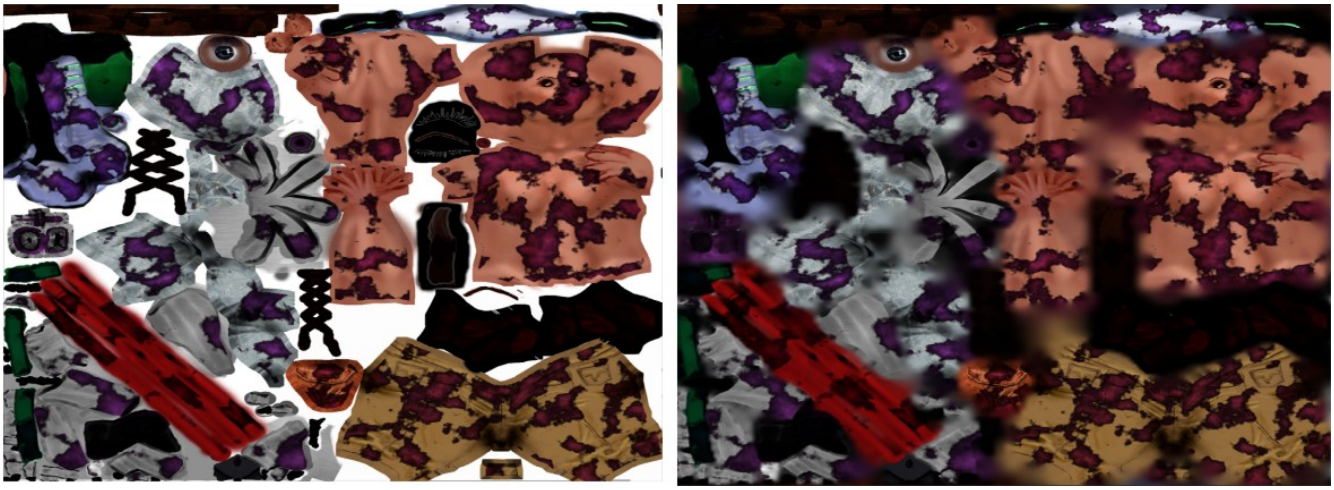

Figure 49: Example of pull push to albedo color texture, before (left) and after (right).

In this example, we have taken the albedo color texture of an existing model (left image), and added an effect with the tool (purple shapes). After this, we have applied a pull push and the result is the image on the right. If we take a look, what pull push does is filling all the empty space in the first texture (white space in this case), with information that is created by extending the seam texels in a blurry averaged way.

The improvements are very easy to appreciate. In the Figure 50 we can see the results of the texture in Figure 49 applied to its model, before and after pull push.

As we can see in Figure 50, the artifacts caused by the seams on the left image are solved after using our pull push (right image).

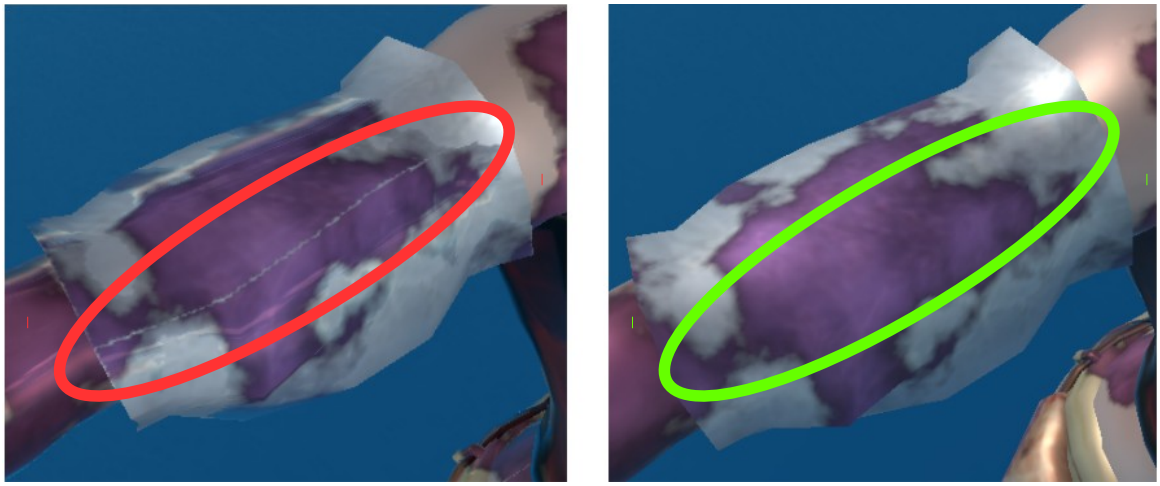

Figure 50: Example of using pull push. Left without pull push. Right with pull push.

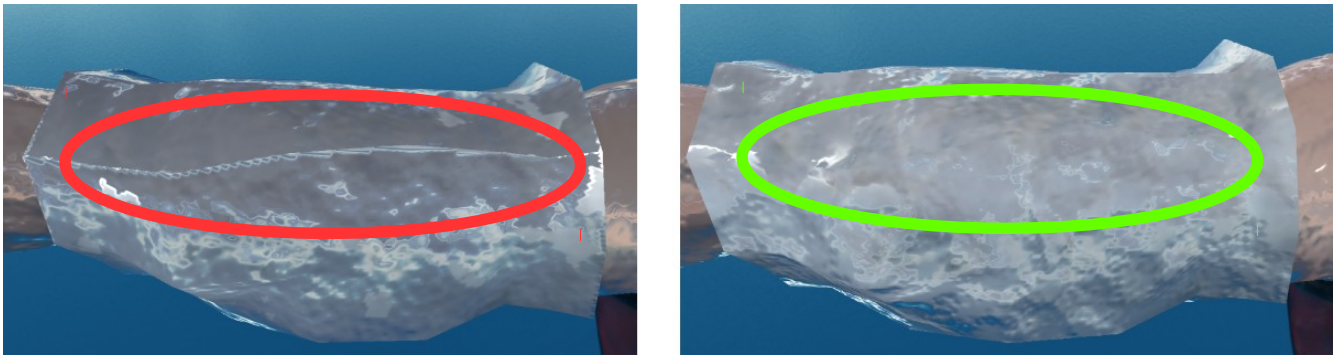

Figure 51: Example of seam artifact being fixed on height field and normals after pull push.

For the height field and normal texture, what we do is applying first a pull push to the height field, and then generating the normal texture. In Figure 51 we can see how it does a good job removing the artifacts.

However, for some high frequency height fields, it still fails in some places, such as the example in Figure 52.

Figure 52: Example of failing pull push for height fields and normal maps.

4.5.4 Normal texture from height field

After we have created, merged and pull-pushed all effects, we will have 4 PBR textures, including the final height field. From this final height field texture, we will create the normal texture, which is needed to use the normal mapping render technique.

In order to create this final normal texture, we will have to compute the normal at each texel in tangent space. We can do this since we have the height at each texel. We use the well-known sobel filter in order to compute the height difference in X and Y, and then use a height of 1 for Z and normalize. Since we are painting the normals into a texture, we remap them from [-1,1] to [0,1]. The result of this process looks like the one in Figure 53.

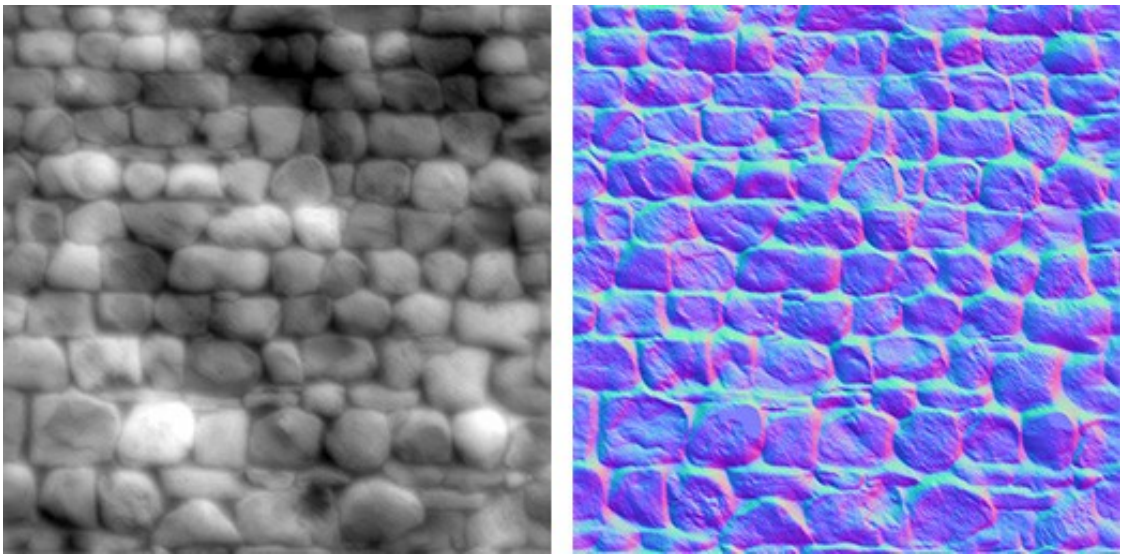

Figure 53: On the left, the original height field. On the right, the resulting normal texture.

4.6 Types of mask

Now we will explain every type of mask. Masks are the building blocks of our effects, so we will need to provide several useful types. In this section we will describe why each mask can be useful for the objectives of the tool and how it has been implemented.

4.6.1 Ambient occlusion

The ambient occlusion mask will select those zones which are occluded, such as corners. This is useful to create effects such as dirt, which tends to be accumulated on corners or little holes.

Its implementation consists in, for every texel, throwing several very short 3D rays in a random direction in the hemisphere facing the face normal. If we find that many of these rays hit the model, then we know that it must be quite occluded. On the other hand, if very few of these rays hit the model, then we can conclude that that texel must be somewhat exposed.

Unfortunately, this is an effect which can not be easily generated in real time. Instead, you have to tell the program when you want to regenerate it. This is because ray tracing is a complex task, and its implementation is not fast enough to work in real time. In addition, it also depends on the number of triangles of the model: for some models it works in real time, but if we try with a bigger model it slows down the application, making it impossible to use smoothly. However, we have still managed to make it work quite fast for our purposes.

The first approach when handling this problem was implementing ray tracing on the CPU. However, it turned out to be extremely slow, even after many optimizations. There were many rays to be traced against many triangles for each texel. This is why, after some time trying to optimize it with no good results, we decided to implement it on the GPU.

Ray tracing on the GPU

Implementing ray tracing on the GPU took much work until it gave proper and fast results. Indeed, its speed is orders of magnitude faster than the CPU version.

To do the ray tracing on the GPU, we do these steps:

- First of all, when loading the mesh, we compute a 3D uniform grid [RRHM]. For each cell in the grid, we save which triangles it contains. The process of creation of the grid must be done by recursively subdividing the space in octants, because otherwise it is very slow. To test whether a triangle is inside a cell or not, we use the well known Separating Axes Theorem [SAT01].
- Then, we serialize the uniform grid information inside a 2048x2048 texture [RRHM]. This serialization consists of a header telling which are the beginning and end indices in the texture

corresponding to each cell, and then a stream of all the cells data, one after the other, which just consists of a series of triangle indices. With this, if we want to retrieve the list of triangles contained inside a cell, we retrieve the begin and end indices from the texture for that cell, and then simply iterate one triangle index after the other, from begin to end. The position of each vertex of each triangle can be retrieved from another 2048x2048 texture that serializes them in a similar way.

- After this, we render the triangles in their uv layout, so that each texel is processed by the fragment shader. At each texel, we determine which is the starting cell inside the uniform grid. And from here, we walk across the uniform grid, following the ray direction. At each cell we traverse, we test whether the ray is intersecting any of its triangles. The search stops when we find an intersected triangle in any of these traversed cells. We obviously also stop the process if we reach the maximum travel distance, or when we arrive to the boundary of the model uniform grid.

In Figure 54 we can see an image of a 2D analogy of the process: first we locate the starting grid cell, then we traverse the grid, checking whether the ray hits the triangle in any of those cells.

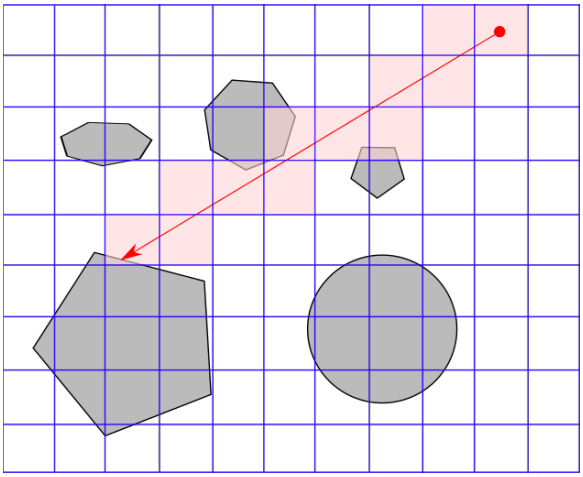

Figure 54: Example of the grid uniform traversal algorithm, but in 2D so that it is clearer.

In the following code snippet, you can see the GLSL pseudo code of a ray trace function based on the described algorithm (some simplifications have been made, the full code is in the file *RayCast.glsl*).

```
bool RayTrace(in vec3 rayOrigin, in vec3 rayDirection, in float maxDistance,
              out float hitDist, out int hitTriId, out vec3 hitBaryCoords)
{
  if (rayDirection == vec3(0) || maxDistance <= 0) { return false; }
  vec3 rayDirectionSign = sign(rayDirection);
  vec3[3] rayDirsXYZ = {vec3(1,0,0), vec3(0,1,0), vec3(0,0,1)};
```

```
  // While we are inside the grid and we have not traveled the maximum distance
  bool insideGrid = true;
  vec3 currentRayOrigin = rayOrigin;
  float currentMaxDistance = maxDistance;
  while (insideGrid && currentMaxDistance > 0) // Start at cell, and traverse the grid
  {
    vec3 gridCoordXYZ = floor((currentRayOrigin - GridMinPoint.xyz) / GridCellSize.xyz);

    // Loop through all the triangles in this cell. Get the number of triangles in it.
    int NC = NumGridCells;
    int cellIndex = int(dot(vec3(NC * NC, NC, 1), gridCoordXYZ.zyx));
    int numTriIdsInCell = GetUniformMeshGridArraySize(cellIndex);
    for (int i = 0; i < numTriIdsInCell; ++i) // For each triangle in this cell
    {
      int triId = int(GetUniformMeshGridElement(cellIndex, i).x); // Get the tri id
      vec3 triPoints[3] = GetTrianglePositions(triId); // Get tri positions from texture

      // Intersect the ray vs. this triangle
      IntersectRayTriangle(rayOrigin, rayDirection, triPoints, hitDist, hitBaryCoords);

      // If the ray distance is less than the max distance, then we have found a triangle
      if (hitDist < maxDistance) { hitTriId = triId; return true; }
    }

    // Advance in ray to the next grid cell. Look for the next closest grid separating
    // plane, and move there along the ray.
    vec3 cellMin = GridMinPoint + (gridCoordXYZ * GridCellSize);
    vec3 cellCenter = cellMin + GridCellSize * 0.5f;
    {
      float minIntDist = 1e12;
      for (int i = 0; i < 3; ++i) // For X, Y and Z plane.
      {
        vec3 planePoint = cellCenter + rayDirsXYZ[i] * GridCellSize[i] *
                          rayDirectionSign[i] * 0.5f;
        float intDistPlane = IntersectRayPlaneDist(currentRayOrigin, rayDirection,
                                                   planePoint, rayDirsXYZ[i]);
        if (intDistPlane >= 0) { minIntDist = min(minIntersectionDist, intDistPlane); }
        }
      }

      // Advance in grid: update the remaining max distance, the ray origin and whether
      // we are still inside the grid or not.
      currentMaxDistance -= minIntersectionDist;
      currentRayOrigin += rayDirection * (minIntersectionDist + 0.001f);
      insideGrid = (all(greaterThanEqual(gridCoordXYZ, vec3(0))) &&
                    all(lessThan(gridCoordXYZ, vec3(NumGridCells))));
    }
  }
  return false;
}
```

With this ray trace algorithm and function, now we just have to throw several short rays in a random normal-oriented-hemisphere direction for each texel, and count how many of them hit a triangle. The more rays that hit some triangle, the more occluded the texel will be.

Finally, it is recommended to apply a blur in order to eliminate some noise, which can be done with the blur mask type, that will be explained later. In Figure 55 we can see some examples of ray traced ambient occlusion, before and after the blur. As you can see, the brighter a zone is, the more occluded it is.

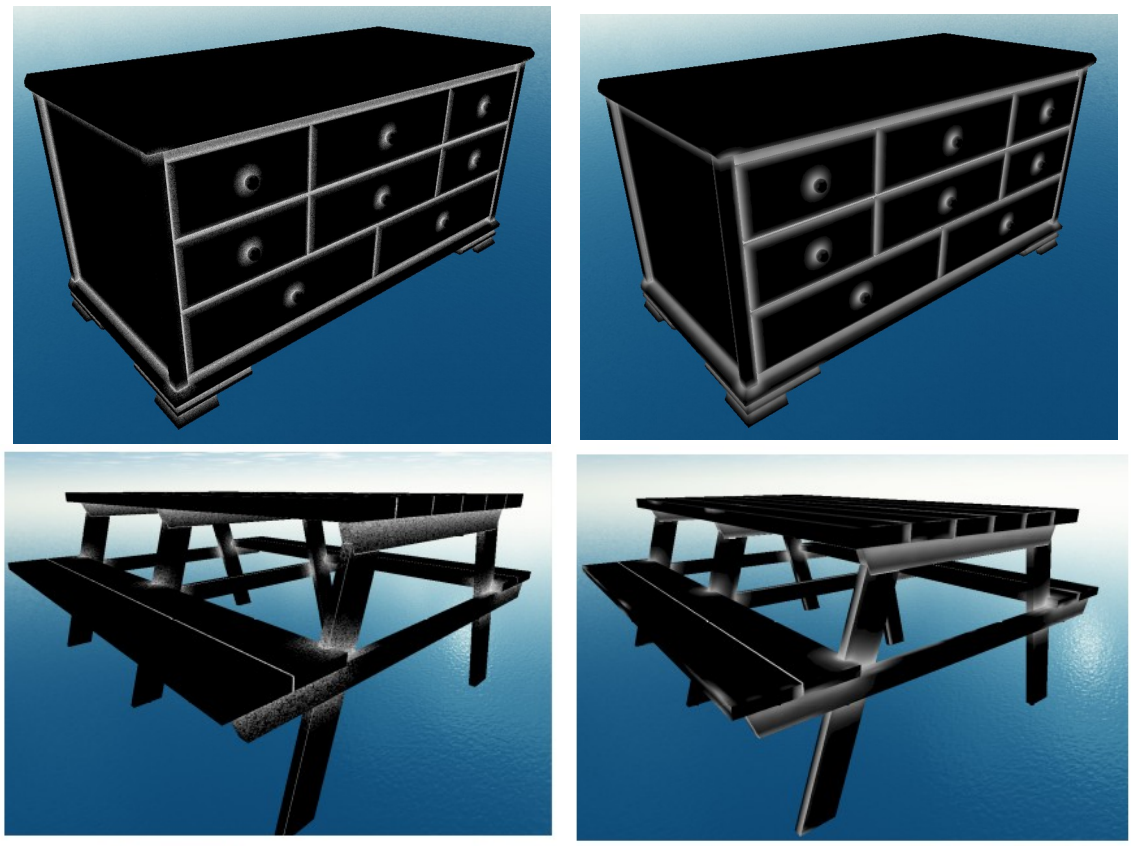

Figure 55: Examples of ambient occlusion mask before and after blur.

The parameters of this mask type are:

- **Number of rays**: determines the number of rays to be thrown. The higher the better, but slower.
- **Max distance:** determines the maximum distance that rays can travel. If you want the mask to be focused on very occluded zones, then max distance should be low.

We have made some experiments to see the performance of the generation of this ambient occlusion mask. These times have been taken in an Intel HD Graphics 4000 GPU with a texture size of 1024x1024. As seen in Figure 56 and Figure 57, the time grows linearly with the number of triangles and thrown rays, as we would expect.

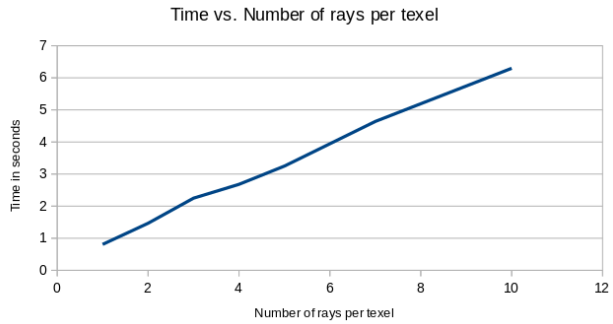

Figure 56: Time depending on the number of rays traced per texel, in a model of 30K triangles.

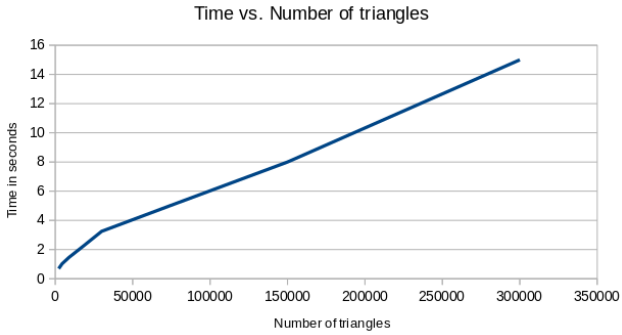

Figure 57: Time depending on the number of triangles of the model, with 5 rays per texel.

With 3 rays the quality per texel is quite good in most cases, and the great majority of models in video games have way less than 50K triangles. This means that we can generate a good ambient occlusion mask in less than 2~3 seconds in most cases.

In addition, in Figure 58, we can see how the time decreases as we increase the number of cells in the grid. However, grids with more than 50 cells per dimension (125K cells) start to have more overhead than time benefits. These times have been taken in an Intel HD Graphics 4000 GPU, in a model of 30K triangles and a texture size of 1024x1024.

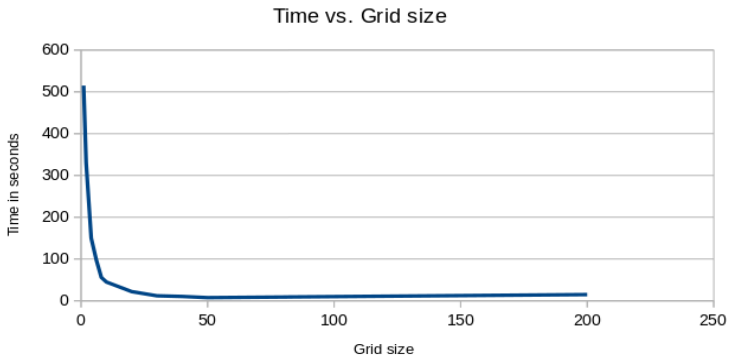

Figure 58: Time depending on the grid size.

Needless to say, the ray tracer could be faster and more cleverly implemented, but we thought this was more than enough for the scope of the thesis, and decided to move on to other more relevant features. Only implementing this ray tracer so that it was robust enough took a big fraction of the time of this thesis. In fact, implementing a fast ray tracer is a very wide and complex topic that would deserve a thesis on its own.

4.6.2 Blur

The blurring mask type will blur the merged mask coming from layers below. It is useful for post processing some other mask types, to soften them out. For example, it is essential for the ambient occlusion effect as we have already seen, so that it can smooth out its high frequency noise.

When blurring, you can choose between two different approaches.

4.6.2.1 Texture blur

This first implementation of the blur is the typical blur over a texture. It simply passes a gaussian blur kernel through the mask texture. An example of this mask type applied can be seen in Figure 59.

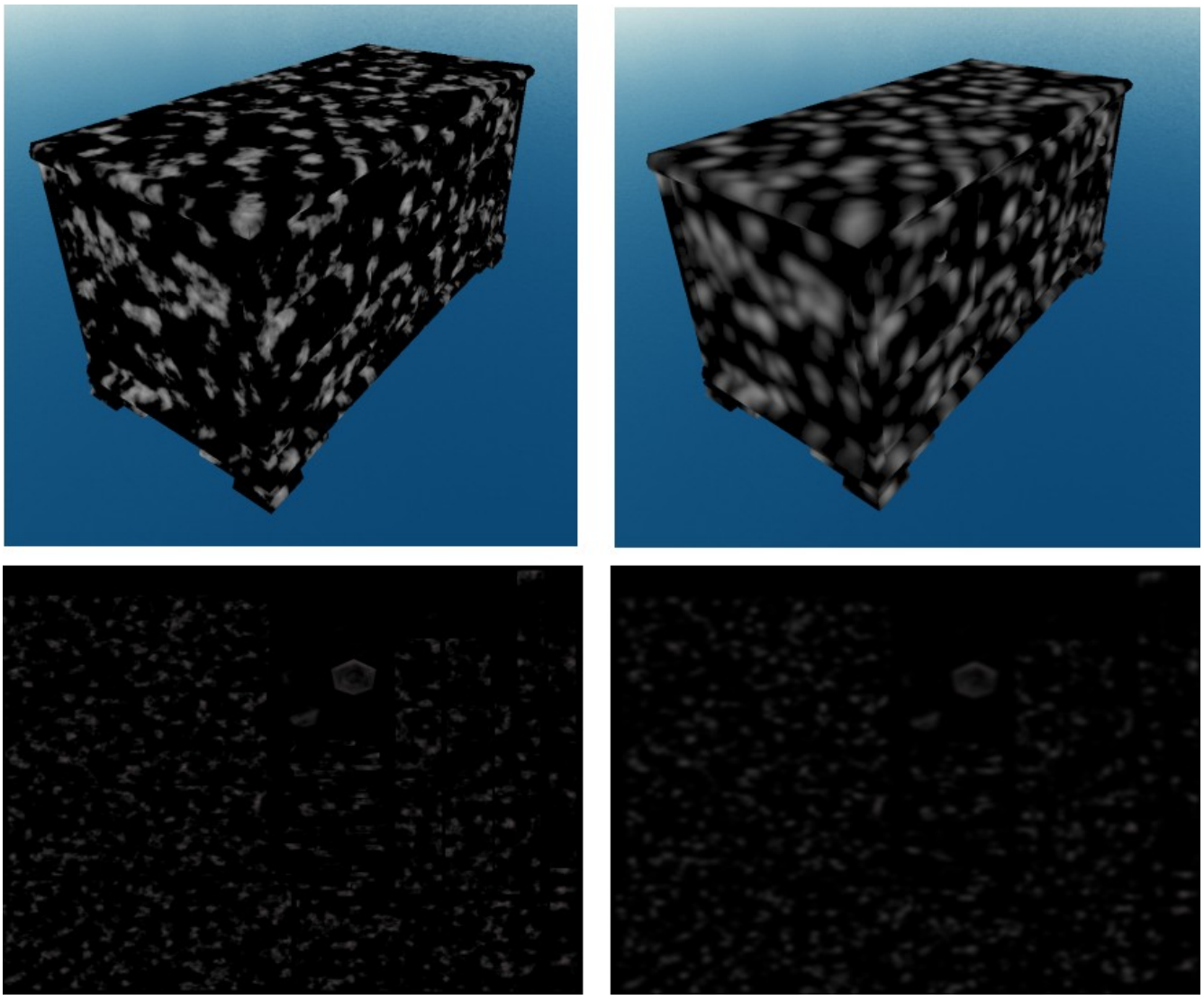

Figure 59: Example of blur. On the left, a mask before the blur. On the right, after the blur.

However, in a typical texturing layout we find many seams that split neighboring triangles. Due to this, applying a simple texture-space blur, will produce visible discontinuities between neighbor triangles split by seams, and this is usually an undesirable effect. Imagine we have to do a blur in a model of a cube that an artist textured so that triangles are separated between them, as in Figure 60.

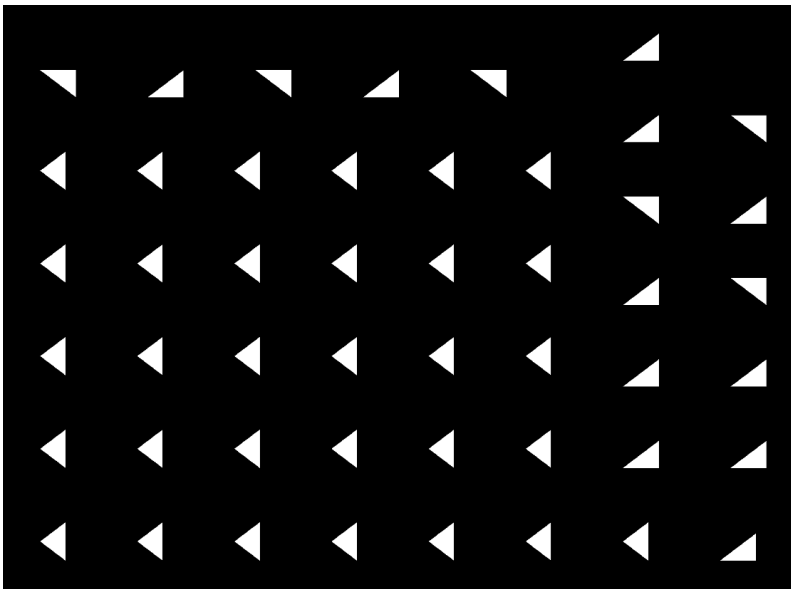

Figure 60: Example of a cube texture layout with many seams. Each triangle is separated from the other.

And now we apply a simple effect to the cube: two red stains. Imagine we now want to do a blur. If we do a simple blur in texture space, we would get the results in Figure 61.

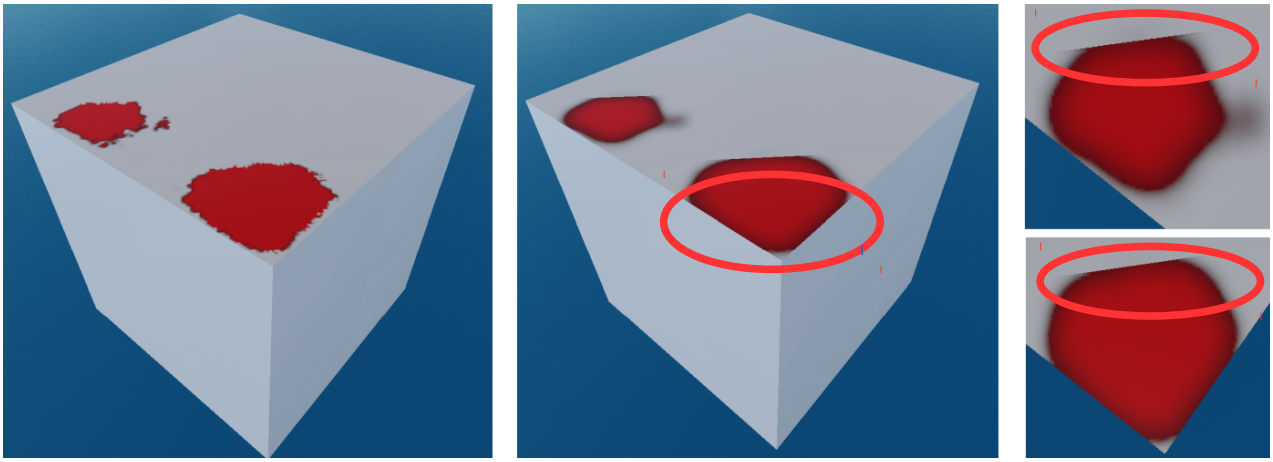

Figure 61: Example of texture blur over a cube with very split triangles in the texture layout.

As you can see, the blur is abruptly cut in the edges of the triangle and in the cube corners. This is because the red color information is not being retrieved by neighbor triangles, because we are simply

doing the blur in texture space. To fix this typical problem, we developed a technique to blur in 3D surface space, explained in the next section.

4.6.2.2 3D Surface blur

This other kind of blur is a 3D version of a typical blur, in which the sampling vectors “follow the surface”. Thanks to this, since we are not doing it in texture space, we will not find the seams problem.

However, its implementation is more complicated, and it takes more time to be computed. For each mask texel, we must do the following steps:

- Identify the triangle it belongs to. For this we can simply retrieve the vertex id in the vertex shader, and integer divide it by 3 . Then just forward it to the fragment shader, so that each pixel knows to which triangle it belongs to.
- Define two perpendicular 3D unit vectors tangent to the triangle, which will be used to pick our samples. These vectors are analogous to the X and Y axis when blurring a texture classically. To calculate these two 3D tangent basis, we can for example pick as the first basis vector as the edge from the first to the second vertex of the triangle. Then the second tangent basis vector is just the cross product of the first vector with the triangle normal. See Figure 62.

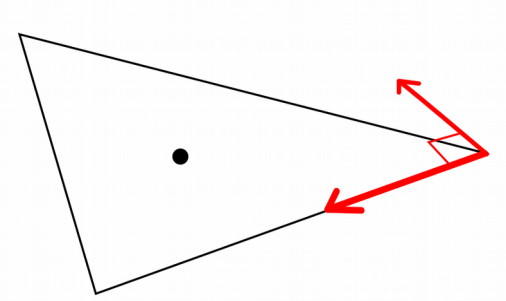

Figure 62: Definition of two 3D perpendicular basis vectors tangent to the triangle.

- Now we must sample points in a grid tangent to the triangle using our basis axes calculated in the step above. To do this, we will retrieve a series of 3D points aligned to this tangent grid, as we would do with a normal texture, but in this case in a tangent 3D grid. If we average all these 3D samples, then we get the final blur value. See Figure 63.

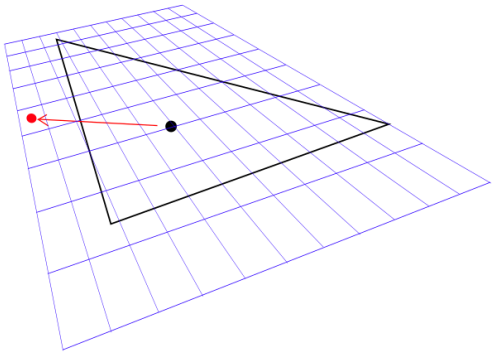

Figure 63: 3D sample grid for the triangle. See how the sample can fall outside the triangle...

But it is not that straightforward to determine what is the mask value of each of these 3D samples. What happens if the 3D sample point does not fall inside the original triangle as in Figure 63? This is what this algorithm can handle. To sample a mask value for a given 3D sample location we must:

- Project the 3D sample point to the triangle.
- Check whether this 3D sample point belongs to the original triangle.
- Only if the projected 3D sample point does not belong to the original triangle, we must find the triangle it belongs to and the correct the sample point. This is done as follows:
  - Look to which of the neighbor triangles the sample point is bleeding to. To do this, we can check the distance of the projected sample point to each of the neighbor triangles (by projecting the sample in each one of them). We will pick the triangle that has the smaller projected distance to the sample point, and such that the projected point is inside the triangle.
  - Then, we bend the sample vector so that it “follows” the neighbor triangle, and then pick the mask value in that new 3D location. To bend the sampling vector, we must first find the intersection point of the original sampling vector with one of the edges of our triangle. This is done by intersecting the original sampling vector with a plane representing each triangle edge.

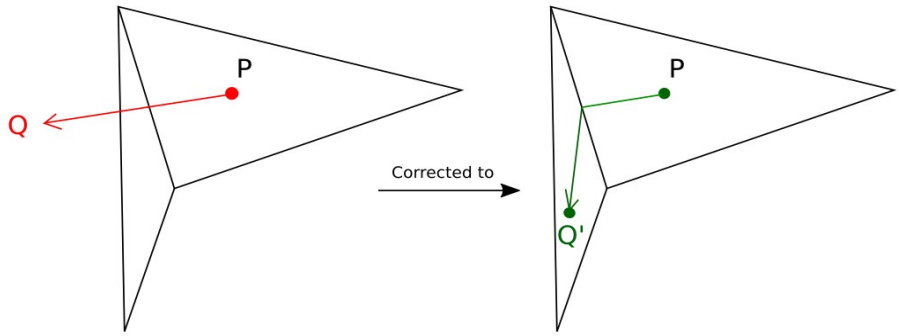

Figure 64: P is the texel we want to blur. Q is the sample point without bending/correction, falling outside the triangle. Q' is the final bent/corrected sample point, which now belongs to a triangle.

Then, get the normalized bending direction by normalizing the vector from the edge intersection point to the sample point projected into the neighbor triangle. Sometimes, this will not work because the projected point can happen to be the same as the intersection point. Because of this, before getting the bending direction, we must move the sample point a little bit. We perturb the sample point by moving it along the normal of the original triangle in the direction that will move it closer to the neighbor triangle. And now, we can effectively compute the bending direction as we mentioned before. Finally, get the new sample point that follows the surface by adding the edge intersection point with the bend direction with the proper magnitude. At the end of the process, we achieve something similar to Figure 64.

  - Now that we know the corrected (if needed) 3D sample point and the triangle that contains it, we get the barycentric coordinates of the sample point inside the triangle it belongs to.
  - Next, calculate the texture coordinates in that projected point by interpolating the triangle vertices texture coordinates with the barycentric coordinates.
  - Finally, just sample the mask texture in the calculated texture coordinates.

- And at the very end, we will have to average each of the sampled values, as we would do with a normal blur. This final average is the blurred mask value that we will assign to our texel. For better results, we can weigh each sample mask value based on how far they are to the original point (for instance, applying a gaussian).

With this type of blur, we can solve the problem we had with the simpler texture space blur (see Figure 65).

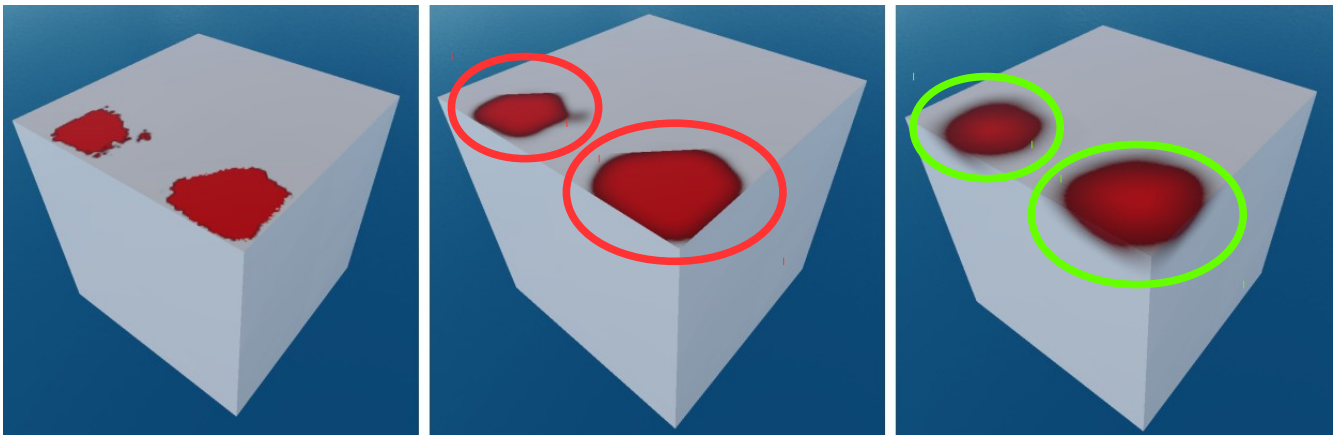

Figure 65: On the left, effect without any blur. On the middle, texture blur. On the right, 3D surface blur.

As you can see in Figure 65, on the rightmost picture, the 3D surface blur takes into account the information on neighboring triangles. With this, we can see how the blur can bleed across neighbor triangles, and even across the corners.

These are the parameters for the blur mask type:

- **Blur mode**: whether to use the first texture space technique or the second 3D surface blur approach.
- **Blur radius**: the radius of the blur. The bigger it is, the wider the blur will be.
- **Blur step resolution**: only for World 3D surface blur mode. It is used to specify the size of the surface pixel in world space. In other words, the size of the pixels in the grid (Figure 63) tangent to the triangle.

4.6.3 Normal

The normal mask type is a very simple mask type which lets you select all the zones in the model with a certain range of normals. It is very useful to define zones in which dust would accumulate, zones in which sun light would hit or zones exposed to stains or scratches. The implementation is as straightforward as looking how aligned is each texel normal with the defined normal. In addition, users can also specify how wide is the marked range and how abruptly it fades out.

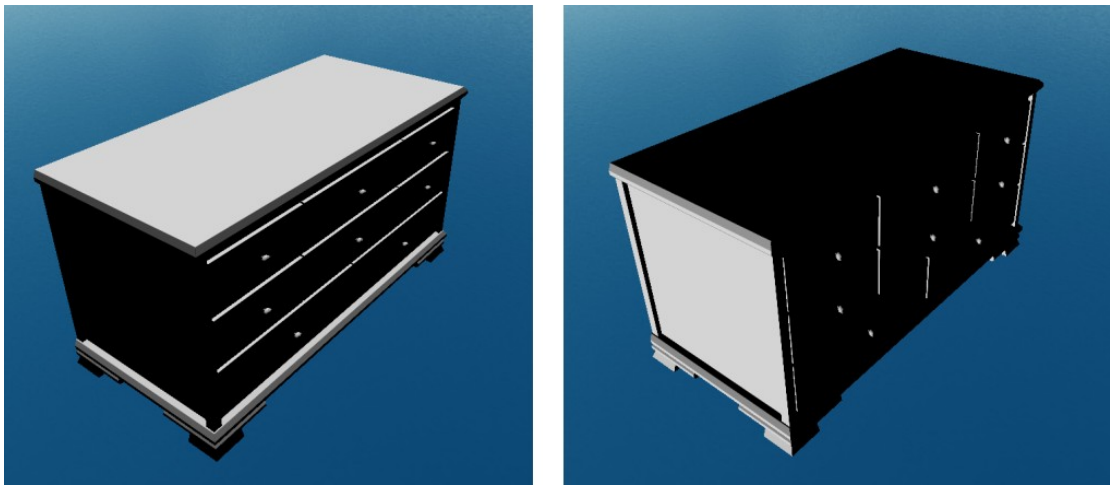

Figure 66: Example of the normal effect applied. On the left, with normal (0,1,0). On the right, with normal (-1,0,0).

You often want to use it with the multiply blend mode, to restrict where another masks must be applied, such in the case in Figure 67. In this case, the effect is only being applied to surfaces that are facing upwards.

Figure 67: Example of effect applied only to surfaces facing upwards.

The parameters for the normal mask are the following:

- **Normal**: a vector specifying the normal orientation of the mask.
- **Intensity**: the strength of the mask.
- **Fadeout**: the amount of fadeout beginning from the normal. Surfaces exactly facing the normal have a mask value of 1, and as you get far from it, it fades out faster or slower depending on this value. This way you can get harder or smoother edges.

4.6.4 Noises

There are several masks with different kind of noises. They can be used for many purposes, such as adding stains, cracks, dots or scratches.

White noise

The white noise mask is the simplest noise mask. It fills the texture with random texel grayscale values. Despite its simplicity, it is still very useful to add some randomness to other effects. It can also be used combined with a blur to add irregularity to surface. An example of this mask is shown in Figure 69.

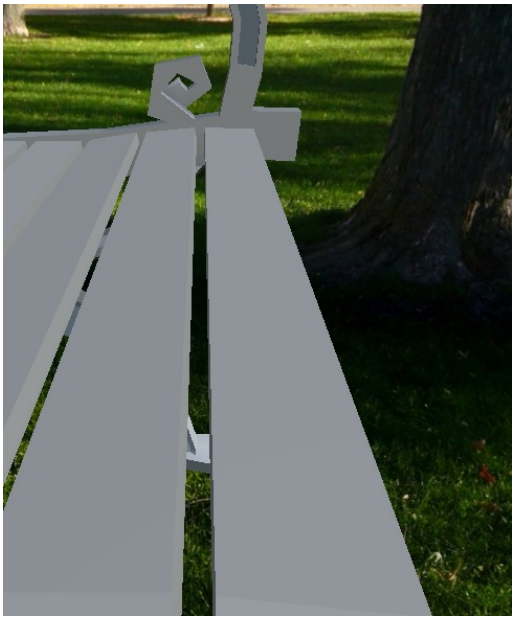

Figure 68: Model before white noise.

Figure 69: Model after white noise applied to height.

Simplex Noise

The simplex noise mask type defines a 3D simplex noise volume, which creates a random pattern that, for our purposes, resembles the shapes of dirt, mud or corrosion.

Its implementation consists in, for every texel, picking its 3D location and evaluating the 3D simplex noise function in that position. In the article [SGSN] you can see a very good and detailed explanation of how simplex noise works in 3D.

We have added some parameters so that the user can intuitively tweak the result: intensity, offset, stains size, grain and seed. Most of these parameters are a mapping of the typical simplex noise parameters (amplitude, amplitude multiply, frequency, frequency multiply).

Finally, it always uses 8 octaves, because it is the value that gives the highest quality and detail. Higher number of octaves just yield the same quality but at greater cost.

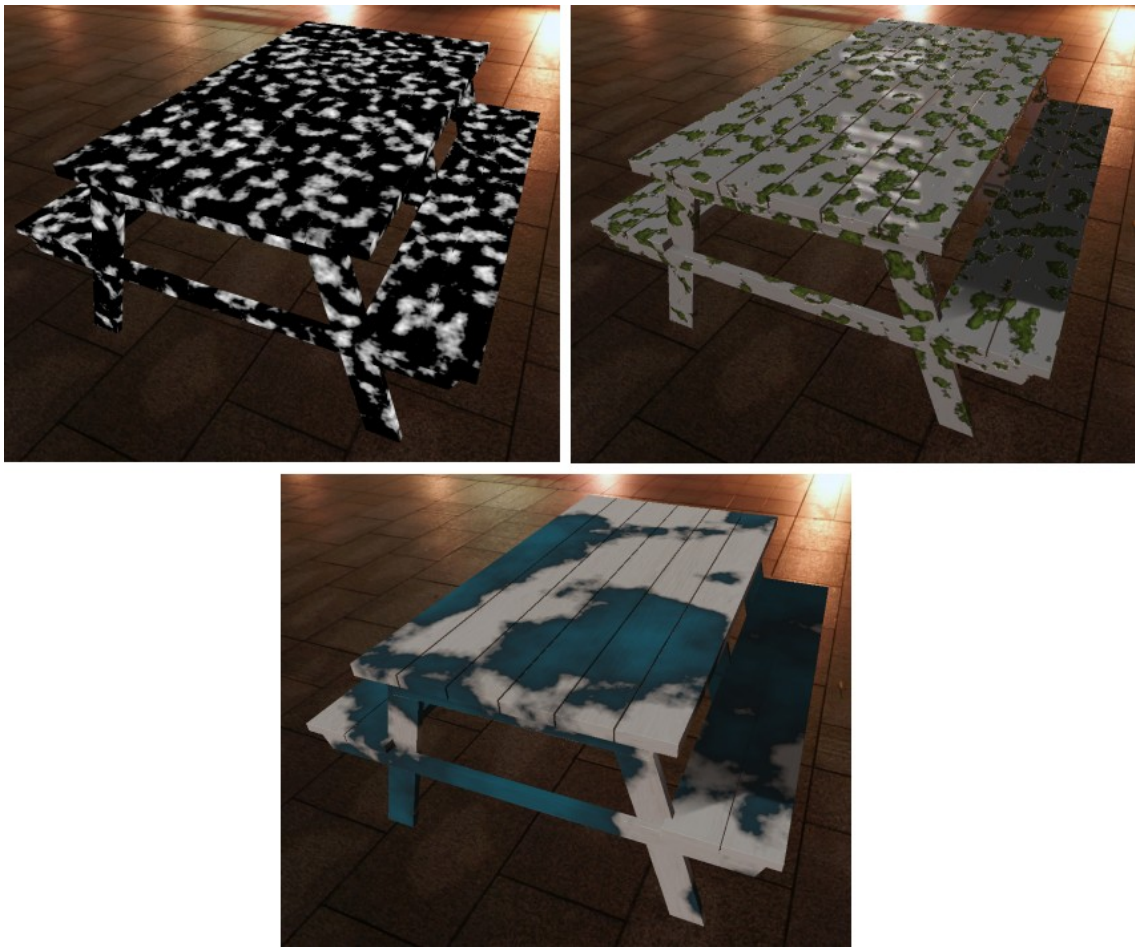

Figure 70: Examples of simplex noise mask applied to a model.

Dots Noise

The dots noise is a noise function that creates random dots. The implementation is as simple as implicitly defining the different points using a 2D hash function, and then computing the minimum distance to one of the dots. The 2D hash function is very important, because it let us generate the same set of points for each pixel. In the following code snippet you can see the code of the 2D hash function (*GetHashedPoint)* to generate our set of points for each pixel. It divides the space into a 2D grid, and for each cell define a random point inside it.

```
vec2 CellSize = vec2(Size * 0.001);
vec2 GetHashedPoint(vec2 inPoint) // “Hash” function to generate the set of points
{
    vec2 cellPos = floor(inPoint / CellSize) * CellSize;
    return cellPos + rand2(cellPos) * CellSize;
}
```

We have added parameters to control the amount of dots, their intensity, their size and their sharpness.

It is useful to add irregularities to a surface and to add detail to some types of plants, mold or moss.

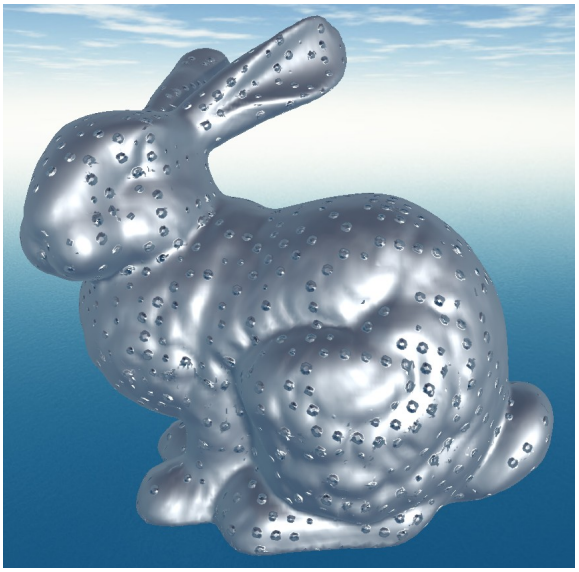

Figure 71: Dots noise to add bumps to a surface.

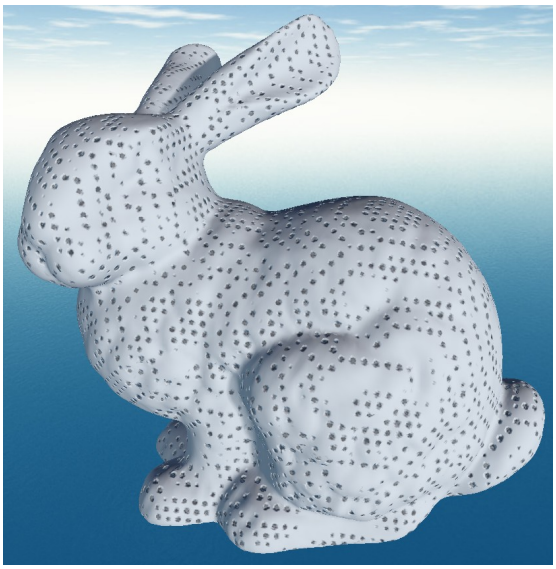

Figure 72: Dots noise to add holes to a surface.

Cells Noise

The cells noise is a noise that creates a pattern of cells. It is somewhat similar to a voronoi diagram, because it is created by defining again a set of points using a 2D hash function (the same as in the dots noise), and then computing the euclidean distance to the closest of these points. It can be useful to create cracks or bump patterns on the surface by modifying the height, as you can see in Figure 74. In addition, it can also be used to add some vein-like pattern imperfections to the model, which can be useful to add realism to plant or wood effects for example. It has parameters to control its intensity, its sharpness and its size.

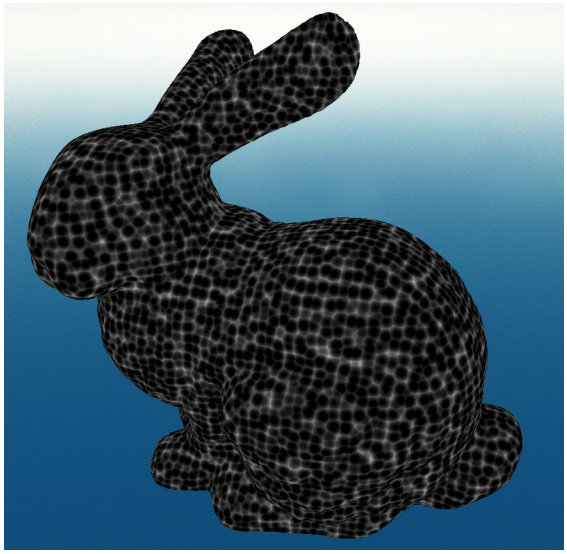

Figure 73: Cells noise mask.

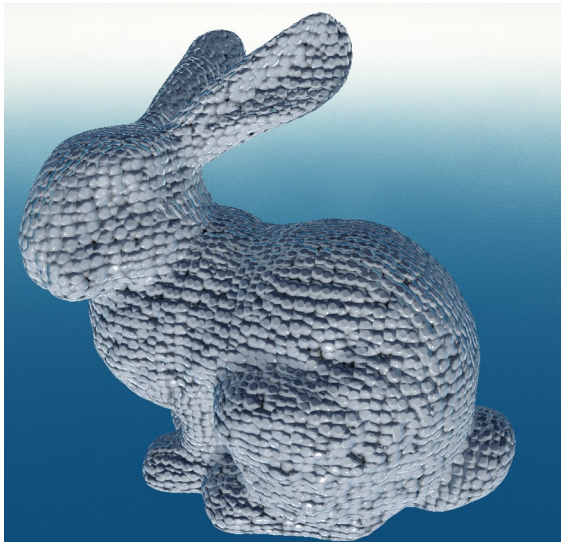

Figure 74: Cells noise mask used in a bunny to decrease heights, and create that bumps/cracks pattern.

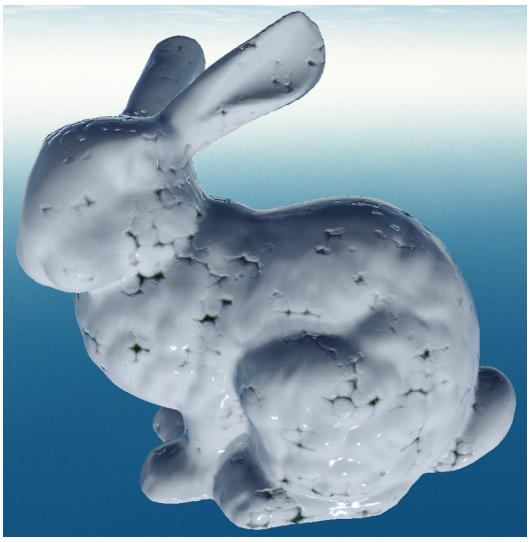

Figure 75: Cells multiplied with simplex noise.

Cracks Noise

The cracks noise is a more complex noise that creates a pattern of cracks. Indeed, this one is actually a voronoi diagram. To implement it, we first define a 2D hash function that implicitly creates points in 2D space (the same as in the dots noise). Then, for each pixel, we compute its two closest points. From these two closest points, we compute the perpendicular line bisecting them, and then the distance from the pixel to this line. If the distance to the bisecting line is low enough, then we paint it white. Otherwise, we paint the mask texel black.

We have created parameters to control the pattern size, intensity, lines width and its fade (how fast is the transition from white to black in the mask). Here we have the simplified and not optimized GLSL code. The cells noise and polka dots pattern GLSL code is similar to this one, but simpler. In fact, the hash function to generate the 2D points is the same (*GetHashedPoint*).

```
vec2 CellSize = vec2(Size * 0.001);
vec2 GetHashedPoint(vec2 inPoint) // “Hash” function to generate the set of points
{
    vec2 cellPos = floor(inPoint / CellSize) * CellSize;
    return cellPos + rand2(cellPos) * CellSize;
}

float CracksNoise()
{
    vec2 firstClosestPoint = vec2(0), secondClosestPoint = vec2(0);
    for (int i = -1; i <= 1; ++i) // Find 2 closests point of cells around
    {
        for (int j = -1; j <= 1; ++j)
        {
            vec2 ijPoint = GetHashedPoint(inUv + vec2(i,j) * CellSize);
            float ijDist = distance(inUv, ijPoint);
            if (ijDist < distance(firstClosestPoint)) {
                secondClosestPoint = firstClosestPoint;
                firstClosestPoint = ijPoint;
            }
            else if (ijDist < distance(secondClosestPoint)) {
                secondClosestPoint = ijPoint;
            }
        }
    }

    vec2 midPoint = (firstClosestPoint + secondClosestPoint) / 2;
    vec2 lineDir = vec2(firstClosestPoint - secondClosestPoint);
    vec2 perpLineDir = vec2(-lineDir.y, lineDir.x);
    float MinTh = 0.02 * Width * 0.01;
    float MaxTh = MinDistThresh * Fade;
    float distToLine = DistToLineWithDir(midPoint, perpLineDir, inUv);
    if (distToLine < MaxTh)
    {
        distToLine = (MaxTh - distToLine) / (MaxTh - MinTh));
        return clamp(distToLine, 0, 1) * Intensity; // We are in a voronoi cells junction
    }
    return 0; // We are inside a cell. Very far to the closest voronoi edge
}
```

It can be useful to create cracks or bump patterns on the surface by modifying the height.

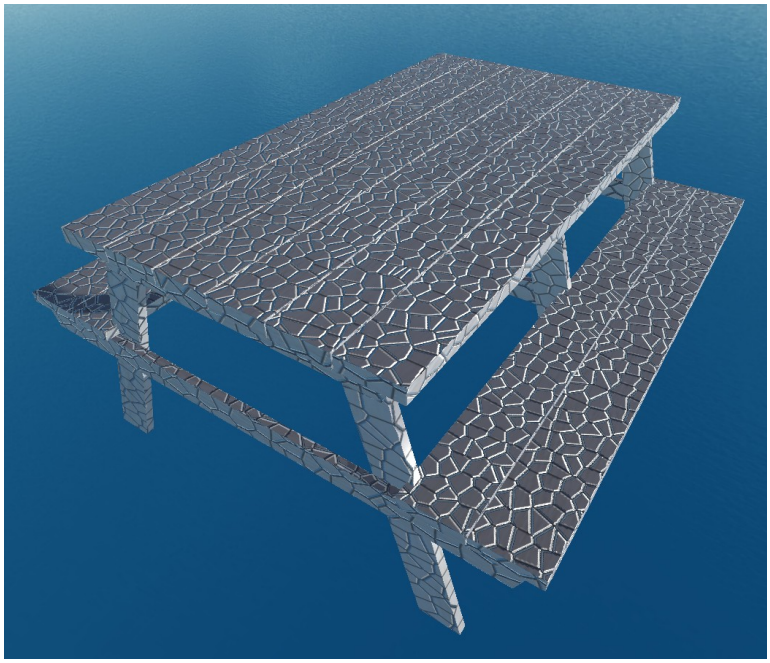

Figure 76: Cracks noise applied to a table.

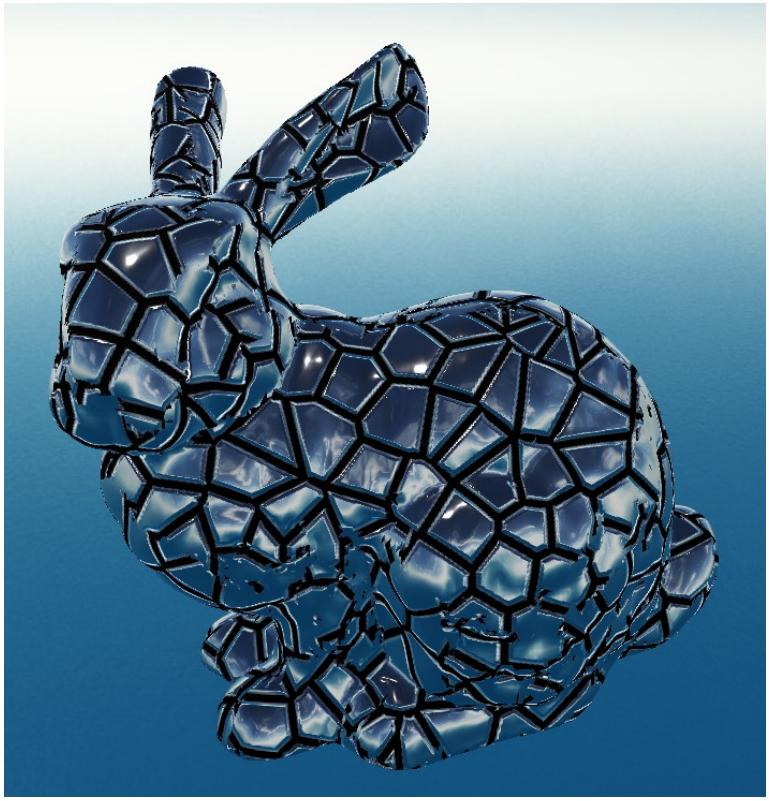

Figure 77: Cracks noise applied to a bunny.

Scratches noise

The implementation of this noise is based in an existing implementation that served a very different purpose [BSGLSL]. However, it has been modified so that it is suitable for our tool and to make it create shapes similar to scratches.

Specifically, it defines pairs of points and direction vectors, and the pixels in between interpolate them. The final result of it is a series of bent segments. All points close to the bent segments are masked.

This noise can be very useful to create scratches/cuts in our models, as we can see in Figure 78 and Figure 79.

Figure 78: Scratches noise in dwarf.

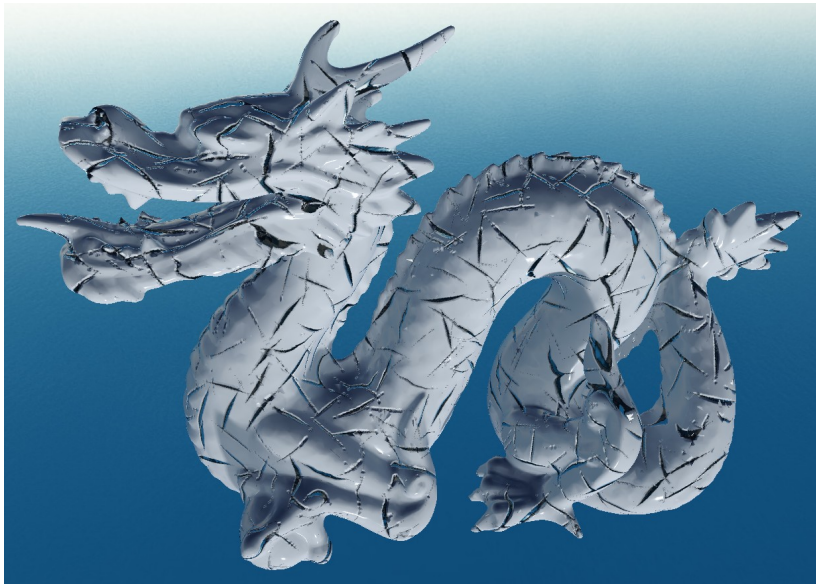

Figure 79: Scratches noise in dragon.

4.6.5 Brush

The brush mask lets the user paint the 3D surface with different preset brushes. These brushes have shapes similar to stains and scratches. We have implemented brushes in such a way so that users can easily add new types of brush by creating a grayscale image and importing it into the brushes directory.

The brush has been implemented in the following way:

- If the user is pressing the mouse button, then we know that we want to paint in this frame.
- As with all the effects, we render the mesh triangles in their 2D uv layout.
- For each texel, we pick its 3D position in model space. Next, we transform it by using the ProjectionViewModel matrix of the current viewport. After this transformation, we now have our 3D position projected into the camera viewport.
- Since we know the mouse position and the brush size, we then sample the brush texel corresponding to the 3D projected position of the texel in which we want to apply the brush.
- Finally, we paint the mask texel with the value of the sampled brush texel. In the cases in which the 3D projected position is outside the brush zone, we just ignore it.

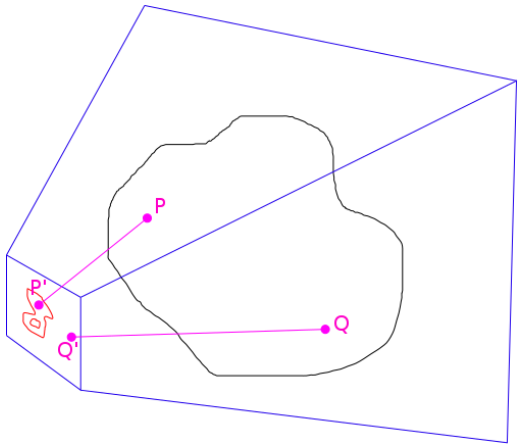

Figure 80: Brush algorithm illustration.

In Figure 80, you can see an example of it. In blue, we can see the camera frustum, which contains a black 3D object. In red, we can see the brush shape in the camera viewport (what the user would be seeing in the screen). Finally, in magenta, we can see the 3D locations P and Q of two texels, and the projected points P' and Q'. In this example, we would paint P, because P' is inside the shape of our brush. But we would ignore Q, because Q' is outside the brush.

The former approach will paint all texels without taking into account whether they are occluded from our point of view or not. This is shown in Figure 81.

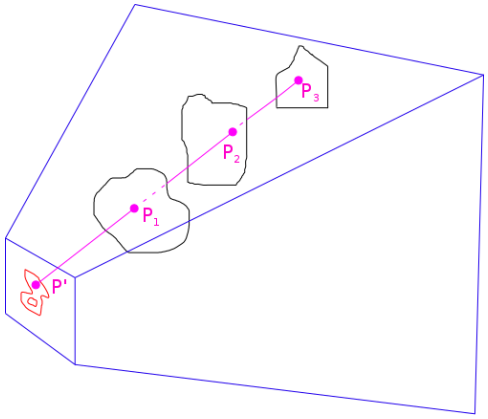

Figure 81: Example of problematic case in which several 3D points (P1, P2 and P3) project on the same viewport pixel P'.

Without any improvement, in the example of Figure 81, the algorithm would paint all points P1, P2 and P3, because they all project into the same viewport point P', which is inside the brush shape. But this is usually not what the user wants when painting a 3D surface. The user would expect only P1 to be painted. Due to this, we want to find a way to only paint texels that are not occluded, i.e. directly visible from the current user perspective (in this case, only P1). To do this, we compare the depth of the projected texel to the depth at that projected point of the viewport camera depth buffer. If the depth of the texel after applying the camera transformation and being projected is greater than the depth seen by the camera at that point, then this texel is definitely occluded, and consequently we will not paint it (as it would happen to P2 and P3). Otherwise, we paint it, as with P1.

You may have noticed this last technique is a very similar process to the one used in shadow mapping. Indeed, if we imagine the camera having a directional light facing its same view direction, we could say that all texels in shadow are the ones occluded (and consequently ignored). In other words, we would only paint the lit texels, which would be the texels that the camera is seeing.

In Figure 82 you can see an example of the brush mask. On the left, we see the brush in screen space before being applied (the red mark is what the brush will paint if the user clicks the mouse). On the right, we see the brush after being applied, in this case, we see the brush paints green color and also height.

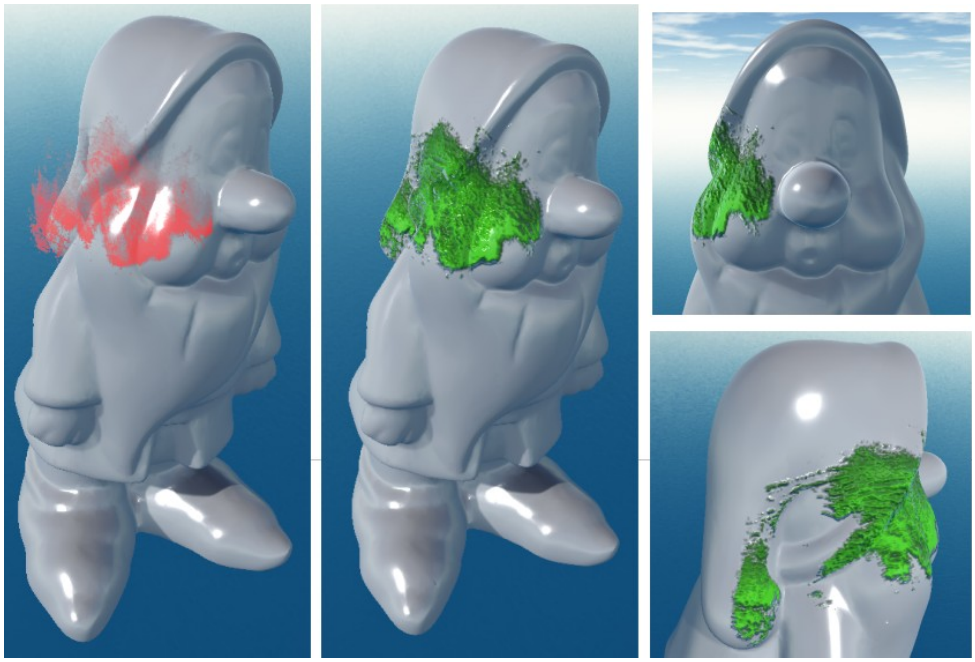

Figure 82: Example of brush applied a gnome model.

Finally, in Figure 83, we see how the brush paints the textures of Figure 82. From left to right, we see the mask gray scale texture, and two of the resulting PBR textures (color and normal map in this case). It is interesting to see how the brush painted in the texture has a completely different shape from what we see in the 3D viewport in Figure 82.

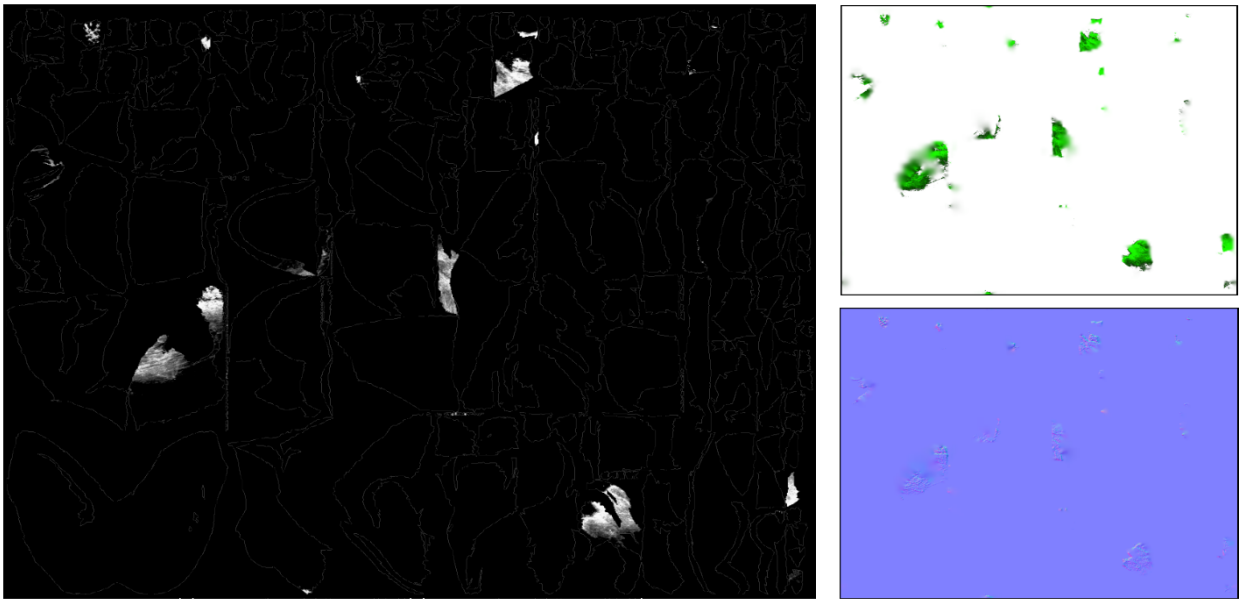

Figure 83: Painted textures by the brush in Figure 82. On the left the mask. On the right, color and normals.

And in Figure 84 there is another example of how the brush can be used to restrict an effect to an arbitrary zone of the surface. Here we use it to give a metallic appearance to a hand, as if it had a metallic glove.

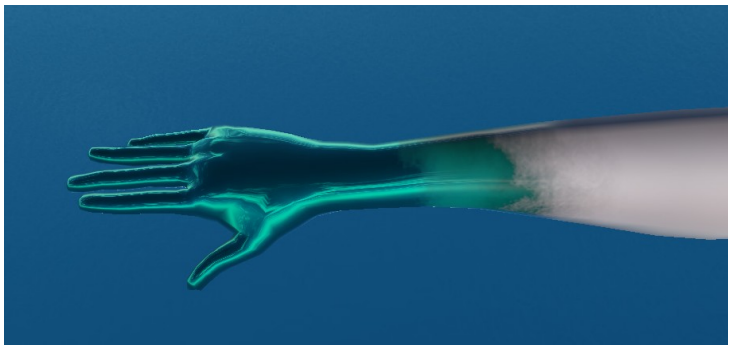

Figure 84: Example of brush used to restrict an effect on an arbitrary zone.

And the parameters of the brush are shown in Figure 85, which are the following:

- **Brush type**: the first parameters lets you choose the type/shape of brush. Users can add their own brushes here by just importing their brush grayscale image into the brushes directory.
- **Strength / size**: the strength / the size of the radius of the brush.
- **Depth aware**: whether the brush paints only the first found surface or it penetrates all the way through.
- **Erase**: whether we want to use the brush to erase or to paint.
- **Clear/fill mask**: lets you clear/fill the whole layer mask.

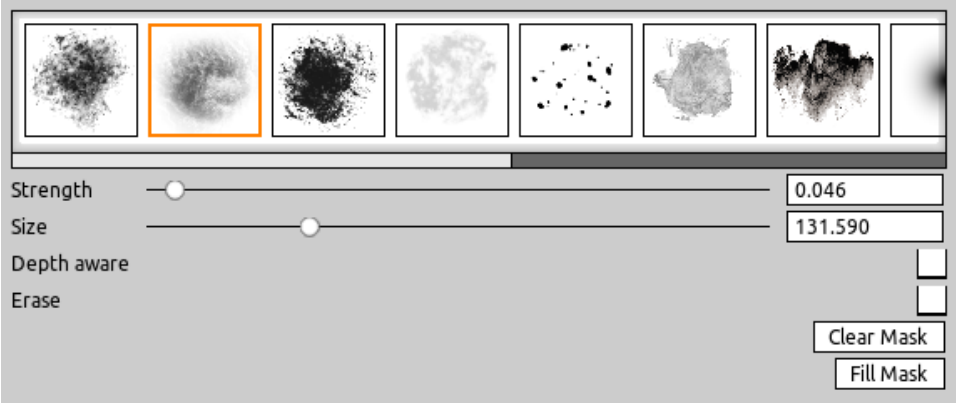

Figure 85. Brush parameters

4.6.6 Edges

This mask type selects the edges in the model. It uses ray tracing, and consequently, as the ambient occlusion mask type, it can not be generated in strict real time, but it still is very fast.

Selecting the edges can be very useful for many effects. For example, an object usually breaks much easier at edges Another use cases would be to define the zones in which an object would have edge wear, or the zones in which moss would accumulate.

The method we came up with to detect the edges in a model is the following:

- We want to determine the “edgeness” of each texel.
- For each texel, we first determine its world position. See Figure 86.
- After this, we define a plane parallel to the surface at that texel. See Figure 87.

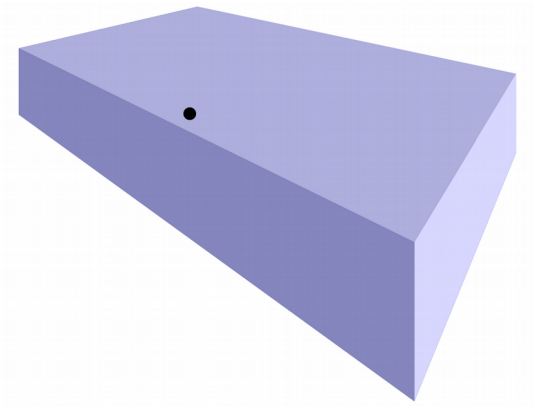

Figure 86: We want to determine the "edgeness" of the marked point in the image.

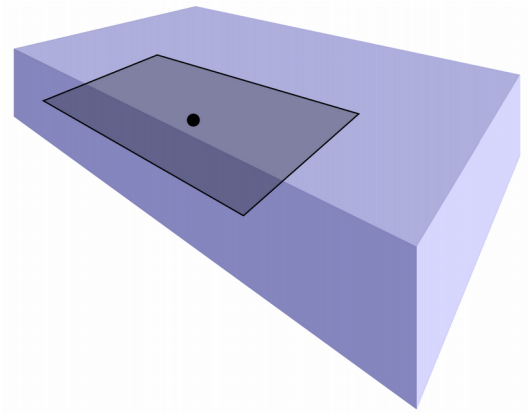

Figure 87: Parallel plane to surface at the point whose "edgeness" we want to determine.

- After we have defined the parallel plane, we sample random points in this plane. For each sampled point, we trace very short rays in the opposite direction to the normal of the surface. All the rays must have the same length. Out of all these short rays, we count the number of rays that hit the model. If many of these rays hit the surface, it means that we are over a surface that locally seems like a plane. But if very few rays hit the surface, then we know we are on a local edge. See Figure 88 and Figure 89.

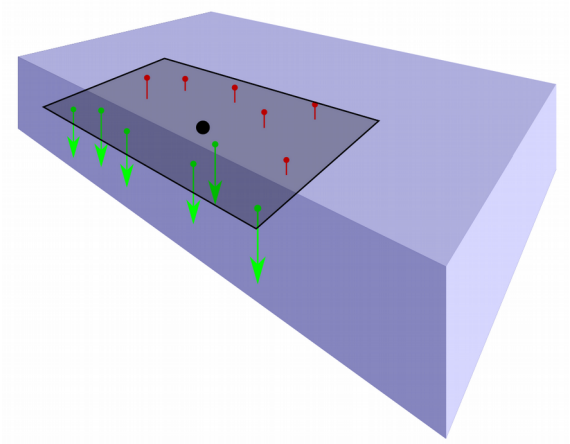

Figure 88: After we have defined the plane, we throw many short rays and count the hits. Here we are detecting an edge, because several rays (green) do not hit the surface.

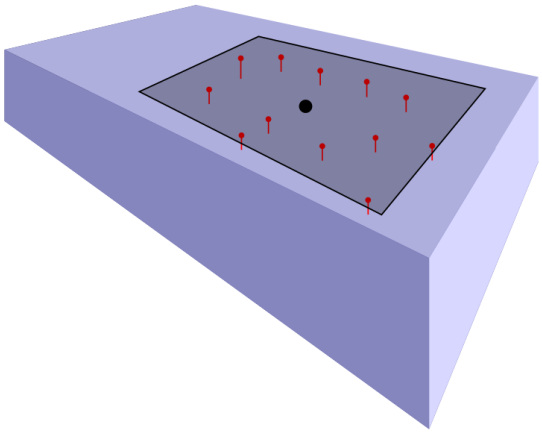

Figure 89: In this case, many rays (in this case all of them) hit the surface. This means we can consider to be over a locally plane surface.

In addition, to handle properly some special cases, we must consider also as hit all those rays whose origin is inside the surface. If the mesh is a two-manifold, we can know whether a point is inside the model if we throw a ray to infinity and we hit a surface from the inner side (the dot of the ray direction with the hit surface normal is greater than 0). Or by throwing a ray and counting the number of hits (if odd we are inside, if even we are outside). But in the current implementation, we use the first method because you can stop at the first hit and is consequently faster.

Before this ray tracing approach, we tried computing the curvature of each vertex. However, we saw that results depended very much on the size and shape of the triangulation of each model, giving bad, unexpected and varying results even across the same mesh. This is why we had to switch to the more generic ray tracing technique that we explained, which is much more portable between models and stable.

These are the parameters of the edge effect:

- **Num rays**: determines the number of rays to throw. The more rays, the more precision. However, for edge detection with 3~4 rays there is usually enough.
- **Edge threshold:** determines how much curvature an edge must have in order for it to be classified as an edge. This is achieved by moving further or closer the ray tracing plane. The further the plane is, the more sensitive the technique is to edges.

- **Edge amplitude**: determines the width of the marked edges. This is achieved by simply making the ray tracing plane larger or smaller.

You can see some examples of the edge mask type in Figure 90 and Figure 91. And in Figure 92 you can see a use case for this mask.

Figure 90: Example of edges mask with low edge amplitude.

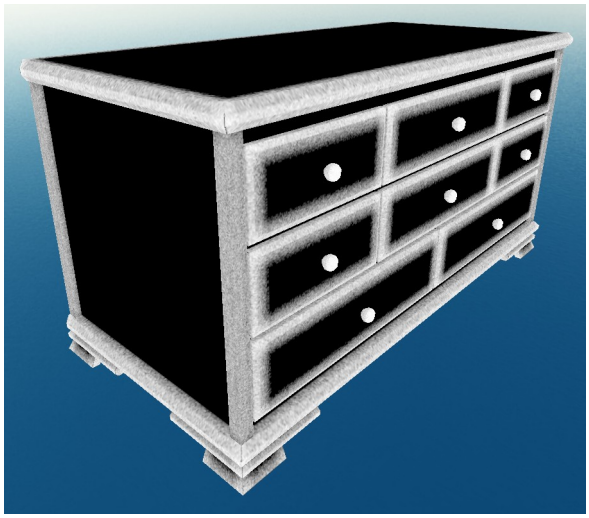

Figure 91: Example of edges mask with high amplitude.

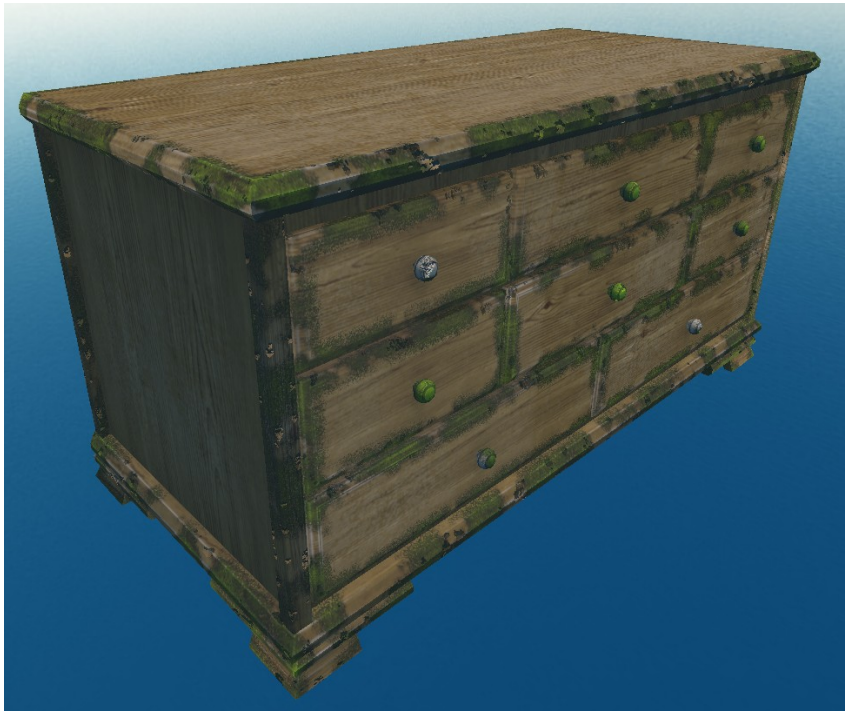

Figure 92: Use case example of edges mask. Here we multiply a simplex noise with the edges mask. Then we add scratches (height) and moss (green color).

4.7 User interface

In this section we will explain how the user interface of the tool is structured. This will let us explain some of the features it provides, as well as giving a better idea of what its work-flow would be.

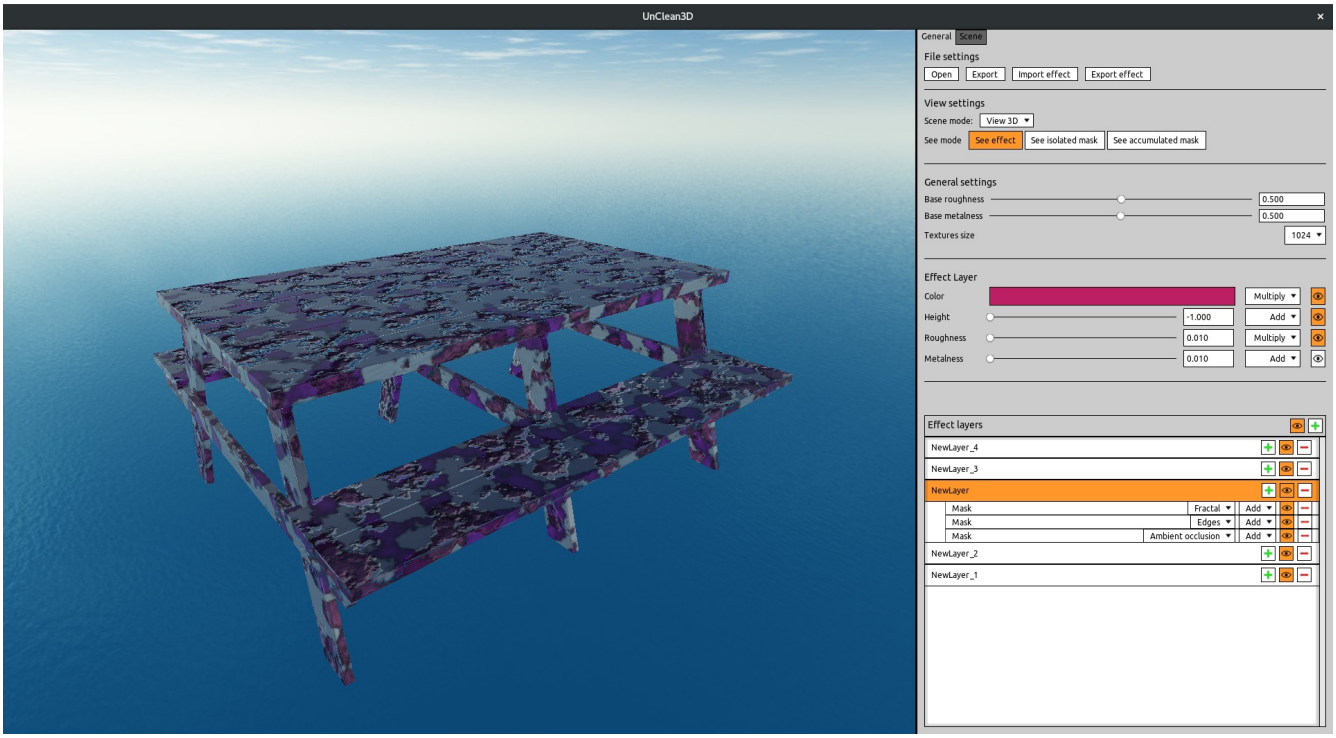

Figure 93: Overview of UnClean3D user interface.

In Figure 93, we can see the 3D viewport on the left, and on the right we can see the control panel.

The 3D viewport lets the user see the 3D model that is being modified. Here you can move the camera by zooming, panning or rotating it around the model. In addition, you can to move the orbit pivot to a point in the model, to be able to focus on more specific model features.

Furthermore, you also have a texture view (see Figure 94), where you can inspect each used texture in the compositing/blending process. It is very useful to debug some effects and see how they actually translate into textures. You can also hover each texture to see it in a larger size.

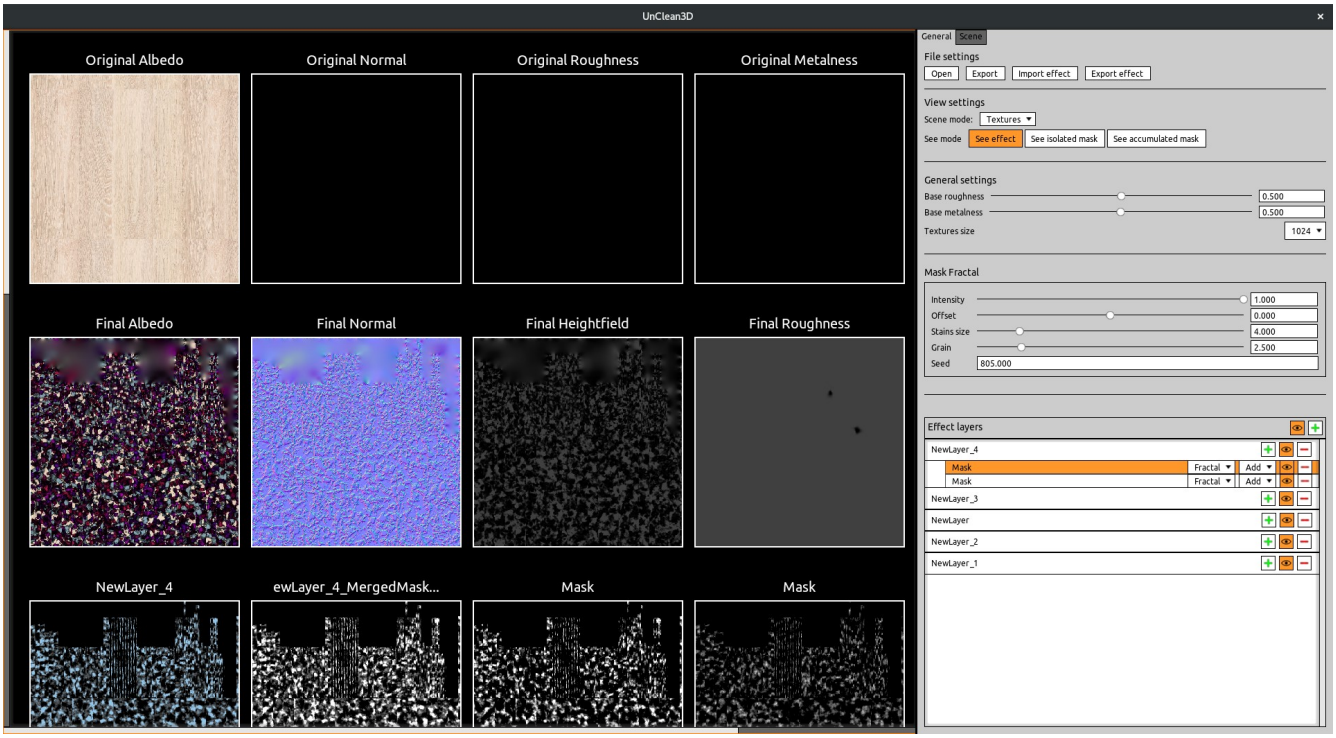

Figure 94: UnClean3D textures view.

Then we have the control panel. It provides you with many options to control everything that happens in the program. First we have the general control panel, which has the following components:

- **File settings:**
  - **Open:** lets you open a 3D model to start the edition.
  - **Export:** lets you export your model to be used in external software such as game engines.
  - **Import effect:** lets you import a previously exported UnClean3D effect file (*.fx).
  - **Export effect:** lets you export the current effect to an effect (*.fx) file, so that it can later be reused for other models.

Figure 95: File settings.

- **View settings:**
  - **Scene mode:** lets you switch between the 3D viewport and the textures view.
  - **See mode:** lets you see in the 3D viewport either the full effect, the selected mask, or the accumulated (merged) mask of the selected effect layer.

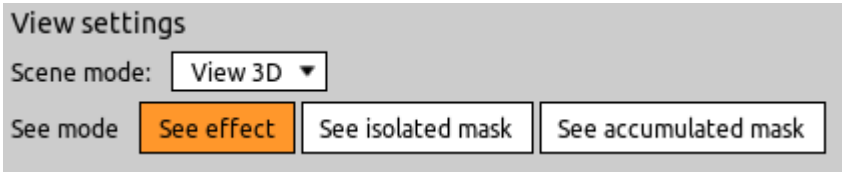

Figure 96: View settings.

- **General settings:**
  - **Base roughness:** lets you change the base roughness of the model.
  - **Base metalness:** lets you change the base metalness of the model.
  - **Textures size:** here you can change the textures sizes to improve the effects quality.

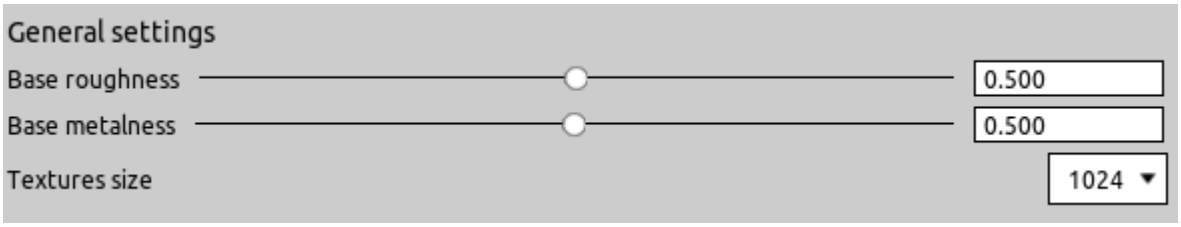

Figure 97: General settings

- **Parameters**: here you can see the parameters of either the selected effect or the selected mask.

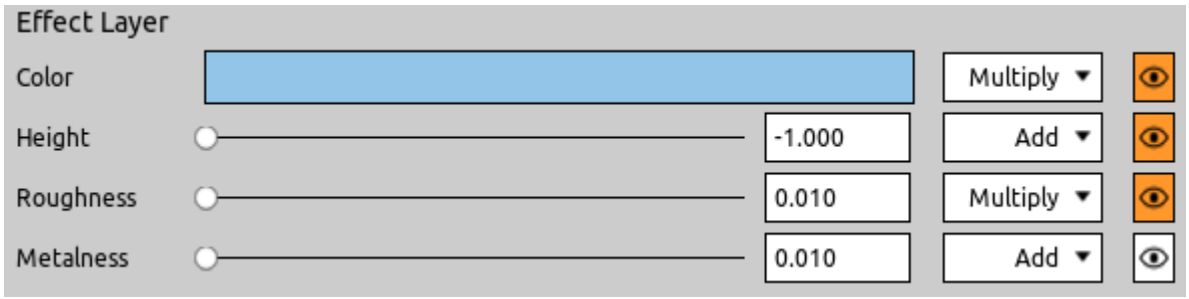

Figure 98: Example of effect parameters for a selected effect layer.

- **Effect layers:** here you can see all the effect layers. It lets you add, remove, toggle the visibility or change the name of layers and masks. Here you can also change the masks blend mode. Finally, you can drag and drop both the effect layers and the masks, so that you can easily change their blending order.

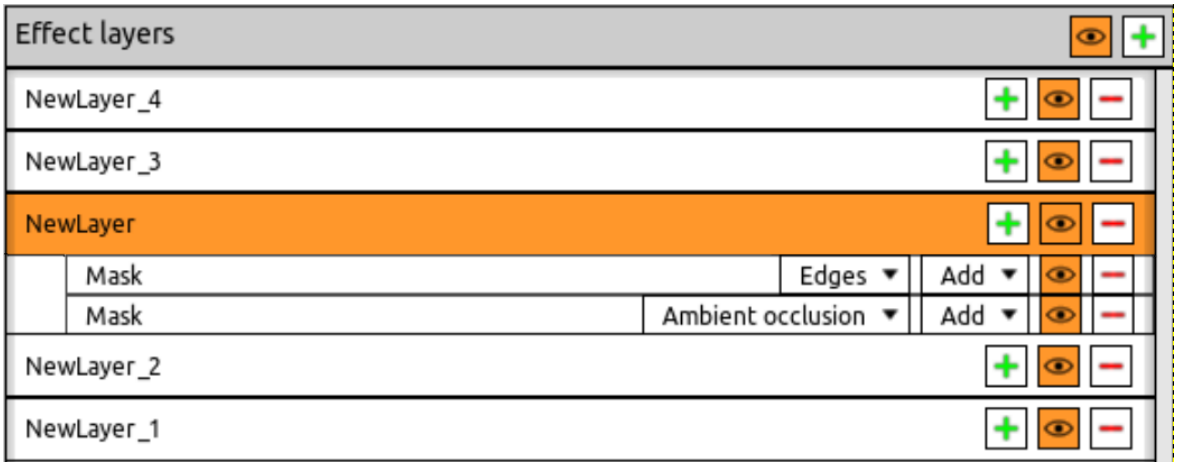

Figure 99: Effect and masks layers.

And finally we have an extra “Scene” tab in the control panel (see Figure 100), which lets you specify whether to enable lights, and if so, whether you want them to rotate automatically. And finally you can also change the environment cube map, to see the model in different scenarios.

Figure 100: Scene control panel.

5. Results

5.1 Effects results

In this section we will show different effects that can be created by combining the types of masks. In addition, for some of the effects, we will provide real life pictures of the same effect in the real world.

In each effect, we will also give an overview of which masks and effect layers have been combined to achieve it.

5.1.1 Light ambient occlusion

This first effect is not an imperfection, but can also be easily achieved with the tool and give more realism to models. This effect consists in painting/baking in the albedo color texture the ambient occlusion shadows of the model. This means, darkening the areas that light would not reach easily because they are very occluded.

The effect is made of a first ambient occlusion mask below so that all the occluded zones are marked. Then, to remove the high frequency noise, we apply a blur mask. Finally, the effect parameters simply applying a little bit of black color to the marked zones, so that they seem shadows. Due to its simplicity, this effect was done in only 2 minutes, and you can see that it improves the realism of the model substantially.

Figure 101: Baking the ambient occlusion into the model color.
Left without bake, right with bake. Notice that occluded zones are darker in the picture on the right.

5.1.2 Chair degradation

In this case, we want to replicate the degradation effects of the real picture of a chair, see Figure 102.

Figure 102. Real life picture of damaged chair.

The first effect layer consists of a simplex noise and dots mask to simulate the little holes in it.

After this, we used another layer with a scratches noise to create the line scratches. These two first effect layers are only modifying the height field and the roughness (scratches have lower height and roughness).

Then we added another simplex noise effect layers in order to create the knots of the wood, and to give it a bit of an orange tint. These two effects only use color.

Finally, we lowered the base roughness so that the light interacts in a similar way, as if it was varnished (notice the specular highlight).

It has been done in less than 5 minutes.

Figure 103: Model before using the tool.

Figure 104: Model after using the tool.

5.1.3 Column with scratches

In this case, we want to create an effect of scratches on a metallic column.

This one is achieved with a single effect layer to create the scratches. The scratches mask is made by combining brushes, scratches noise and simplex noise. Then, the effect layer simply adds gray color and increases the roughness and metalness. This effect was made in 5 minutes.

Figure 105: Real picture

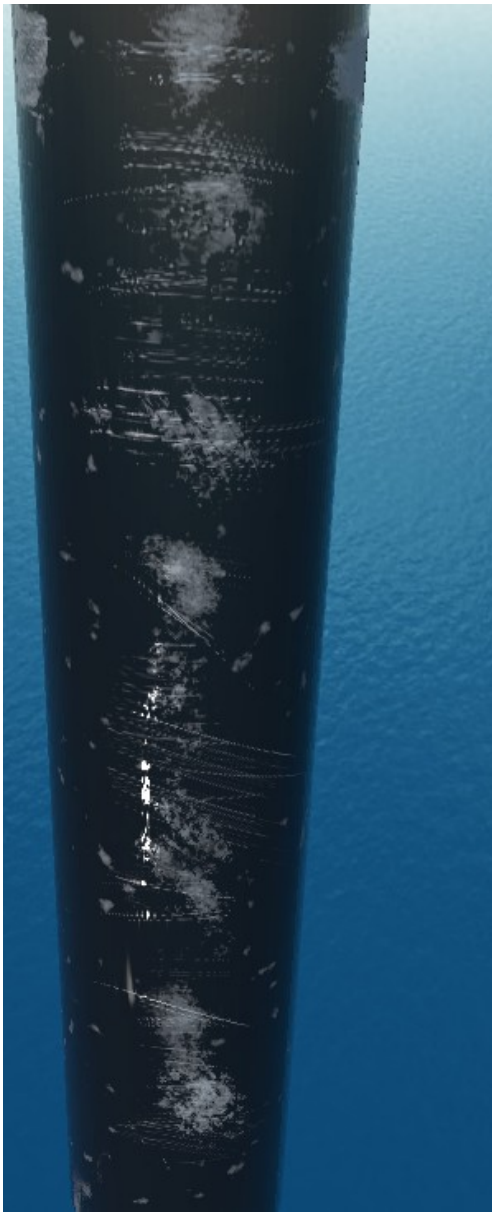

Figure 106: Model with effects

5.1.4 Wall humidities, moss, mold and dirt in corners

In this example, we add several effects to a wall. For wall humidities we mix simplex noise and brush so that they add height to the wall (see irregularities in right wall). Then, for dirt in corners we use the an ambient occlusion mask. And finally, for mold and moss we use brushes, dots noise and the edges mask type. It was made in 5 minutes.

Figure 107: Real picture of wall humidities

Figure 108: Real picture of wall moss and mold

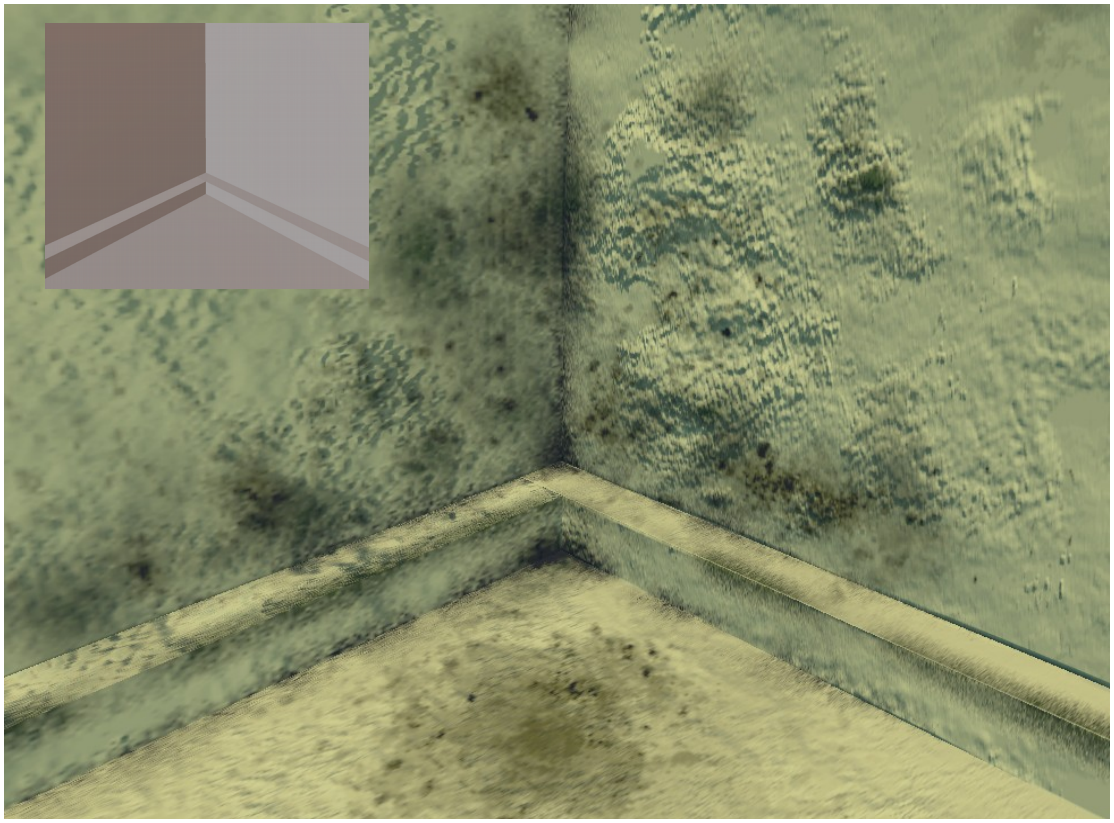

Figure 109: Model after effects. In top left corner you can see the model before the effects.

5.1.5 Peeling, problematic case

This example shows one of the limitations of texture-only effects. In this case, we want to create a peeling effect. We achieved the effect seen in Figure 110 and Figure 112 using simplex and cells noise.

But the problem with peeling is that, because of texture-only effects restriction, we can not create additional geometry. And this makes it difficult to create the illusion of two layers of height. Consequently, it is currently not possible to achieve the double-layered peeling as the one seen in the real picture of Figure 113.

Although we tried to fake the double-layered peeling using several layer combinations, none of them worked as desired. This is why it would be interesting to further research this problem in future work.

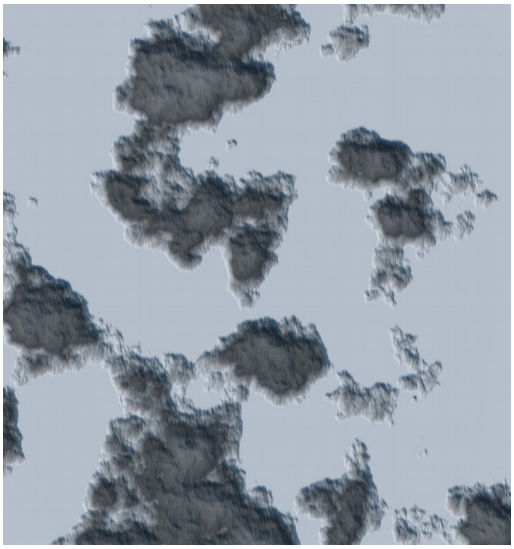

Figure 110: Peeling effect achieved with simplex noise.

Figure 111: Real picture of peeled wall.

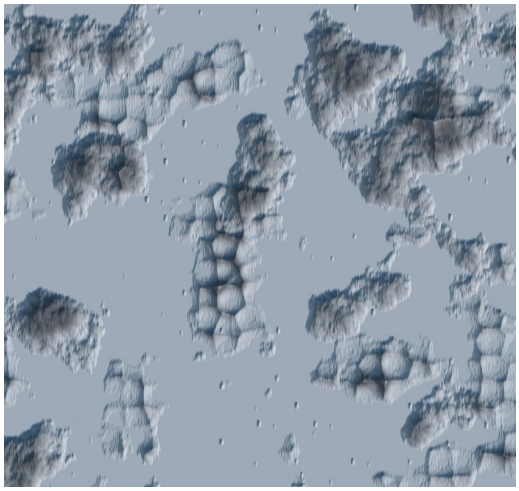

Figure 112: Peeling effect achieved by combining simplex noise and cells noise.

Figure 113: Real picture of peeled wall.

5.1.6 Chain corrosion

Here we create a corrosion effect for a chain. Indeed, we create 4 different types of corrosion, which can be seen in Figure 115. It is just a combination of simplex noise layers, each one adding color, removing metalness, adding roughness and altering height. In addition, we also use white noise to add more grain to the texture. Each of them took approximately 2 minutes to be created.

Figure 114: Real pictures of corrosion on chains.

Figure 115: On the left, the original chain without effects. Then, four different types of rust.

5.1.7 Stone bench degradation

In this example we add effects to age a stone bench. For the broken edges we use the edges mask multiplied with simplex noise, and the effect is subtracting height. For the moss also edges, scratches and white noise, and the effect simply adds color. Then we also added dirt in the corners by using the ambient occlusion mask. Finally, we added some mold with again with dots noise painted in white. This effect took 10 minutes to be created.

Figure 116: Real picture of old stone table and bench.

Figure 117: Bench with effects. On the bot right corner you can see the original model without effects.

5.1.8 Bunny sculpture

Here we show the same model in porcelain and metal. As you can see in Figure 120 and Figure 121, here we use edges masks to create the worn out borders (see for example the eyebrow and the ear in Figure 121). We also use ambient occlusion to add dirt in occluded zones (see the ears hole, legs and feets). As you can see in Figure 121, the roughness and metalness varies across the model depending on simplex noise, notice that the surface is not smooth anymore as in the original pictures. In addition, both have several holes done with simplex noise again. Each effect has been done in 6~7 minutes.

Figure 118: Original ceramic bunny.

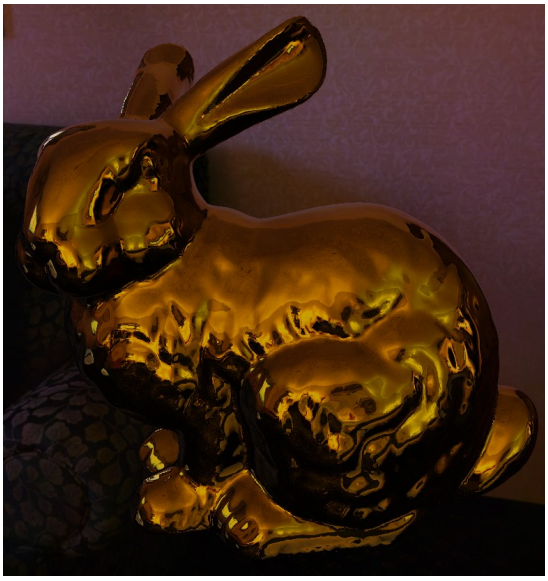

Figure 119: Original metal bunny.

Figure 120: Ceramic bunny after applying effects.

Figure 121: Metal bunny after applying effects.

5.1.9 Reusing stone effects

In this example, we can see how we can reuse effects by using the import / export options. In this case, we reuse the effects created for 5.1.7 Stone bench degradation and 5.1.8 Bunny sculpture.

It took only one minute to create the effect for each of these models. We just had to import it and adjust one edge threshold that did not fit well in these smoother models.

The fact that all masks except for the brush are completely procedural, makes it really easy to use the same effects in different models, as we see in these examples. Furthermore, since none of the masks depend on the triangulation, they adapt even better. For example, for the edges mask we could have used the curvature of the triangles, but it would make it difficult to tweak the effect for every model. But since we use ray tracing, we overcome these limitations.

Figure 122: Armadillo reusing effect.

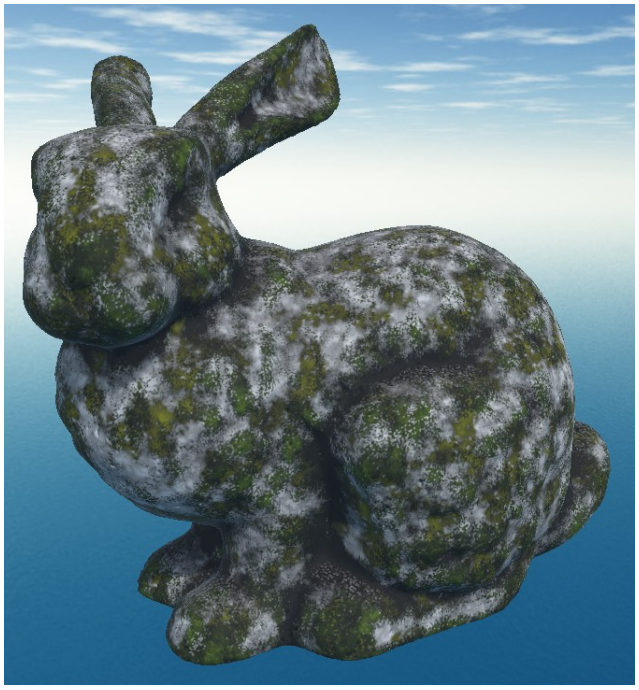

Figure 123: Bunny reusing effect.

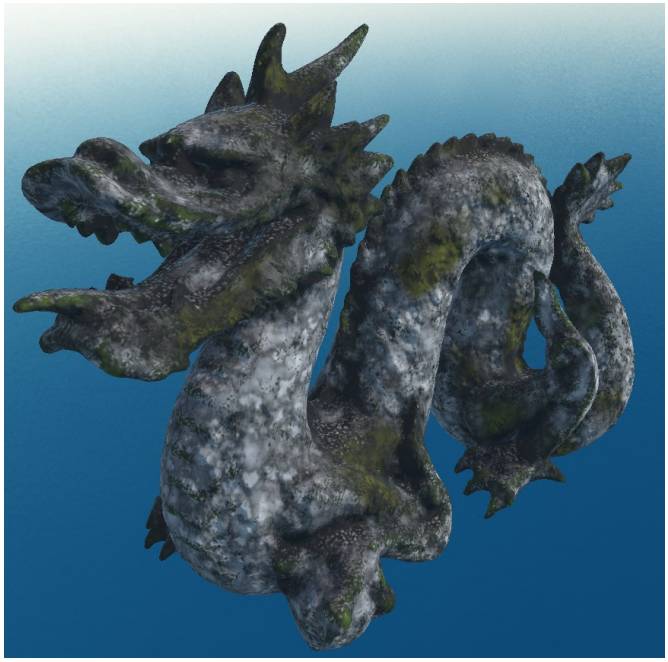

Figure 124: Dragon reusing effect.

5.1.10 Rusty edges effect

Here we can see the same rusty edges effect, applied to several models. The first effect for the gnome took 7~8 minutes to be created, and then for each other just one minute by reusing the effect.

Figure 125: Rust applied to gnome.

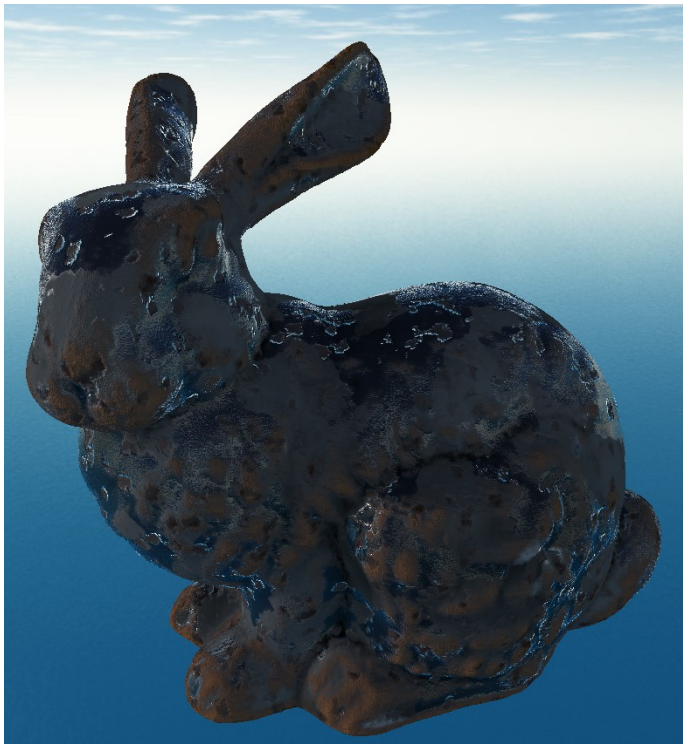

Figure 126: Rust effect applied to bunny.

Figure 127: Rust effect applied to table.

Figure 128: Rust effect applied to armadillo.

5.2 Performance results

Here we will show the results of the performance tests we have carried out. These will test the overhead of the effects on an exported model. All of them have been done with an *AMD Radeon R9 200 Series* GPU (2GB GDDR5), which is a very affordable and basic GPU. The experiments have been done with a very well known commercial engine called *Unity*. In addition, all tests are using all PBR textures (albedo, roughness, metalness, height and normal) with a size of 2048x2048, which is quite good quality.

The first thing we did was exporting several models to the *Unity* engine. In this step we could confirm that indeed it was straightforward to export models from the tool to other software. For instance, to use the exported model in *Unity* we just had to add the model to a game scene and create a Unity material with the exported PBR textures.

Then we created an empty scene with one single light. We wanted to test the results in the worst scenario, so we chose a screen size of 4096x4096, which is extremely high resolution. Furthermore, since we think that the bottle neck in this case will be in the fragment shader (the computations on each pixel) because of the PBR, we positioned the camera so that every pixel of the screen was rendering our model.

First, we did an experiment to see whether the effects have an overhead on the rendering time when used on a model. In case there is overhead, we want to see whether the overhead of the effects on a model increase with the number of triangles of the model or not. To test this, we will render a scene several times with one single model with and without effects, and with increasing number of triangles (from 1K up to 60K), and then measure the time spent to render each frame each time, so that we can measure the render time overhead. My hypothesis is that the effects will indeed cause an overhead, but we should not see an increase in this overhead as we increase the number of triangles of the model. We suspect this because most of the computation needed to use the PBR textures is done for each pixel, and consequently it should be independent of the number of triangles. It is true that when applying PBR some computations can be done per vertex, but we would say these per vertex operations will be negligible, because *Unity* will be doing much more work for each pixel, and there are many pixels (~16M in this case).

The results of this experiment are in Figure 129. As we expected, we can see that there is an overhead when using the effects. Moreover, as we also hypothesized, the overhead remains constant (1 ms), independently of the number of triangles of the model. A similar experiment was done for scenes from 1 object up to 10K objects, and the overhead remains constant as well. From these first experiments, our first hypotheses are confirmed. Consequently, we have seen that there is an overhead when using all the PBR textures vs. when not using any texture at all. However, this overhead is only of 1 ms per frame with one light, independently of the number of triangles or object in our scene. In addition, this 1

ms overhead is in the scenario of making that absolutely all the pixels of our screen are applying all PBR textures, and with a high resolution screen size of 4096x4096.

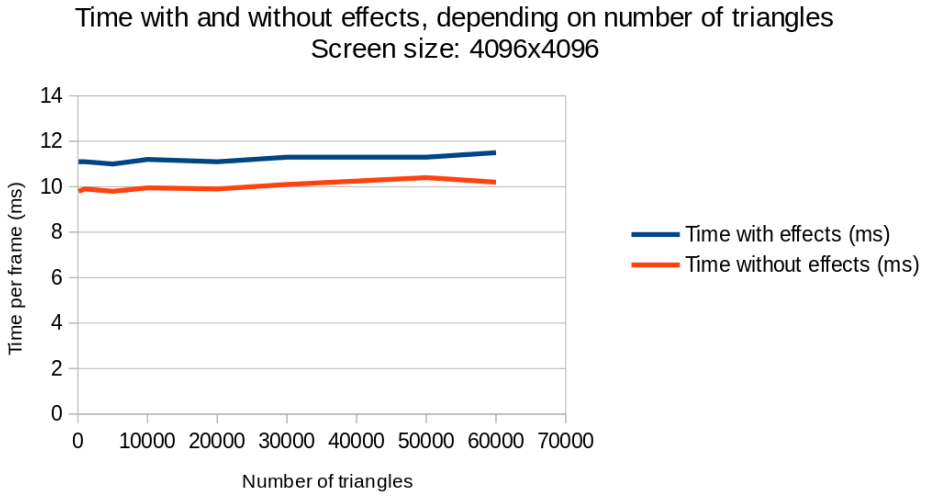

Figure 129: Time with and without effects depending on the number of triangles of our model.

We can now move on and carry out the rest of experiments with a similar camera setup, being sure that we will not lose generality if we use a single model with any number of triangles. In these next experiments, we will be setting up a scene with a single model (this way we can also avoid contamination from other *Unity* engine overheads). Specifically, we will use the bench model and effect from 5.1.7 Stone bench degradation, which has 360 triangles.

Since most of the real time computations for PBR are per pixel and must be done for each light, it will be interesting to see how the effects overhead varies with screen size and with the number of lights. With this next experiment, we want to see how the effects overhead grows as we increase the number of lights. And we will carry out this experiment with two different scenarios: one with a screen size of 4096x4096, and another one with a more realistic and common screen size of 1920x1080. To do these experiments, we created a scene with a single model, and increased the number of directional lights little by little.

Their results are in Figure 130 and Figure 131. As expected, in the plots we can see how the effects overhead increases with each light we add to the scene (the separation between the two curves increases with each additional light). With 1 light the overhead is of 1 ms (~10% of the frame time), while with 25 lights and a screen size of 4096x4096, we can see that the overhead is around 10~12 ms (~20% of the frame time). These results make sense, because each light adds a series of operations we need to do to the model with effects, that are not needed for the model without effects.

And in the more realistic case of screen size of 1920x1080, we can see that the overhead for 10 lights is 0, and for 25 lights is as low as 1ms, as we see in Figure 131. Consequently, we can conclude that for typical 1080p screen sizes, in scenes with less than 10 lights and with an arbitrary number of triangles and objects, the overhead is very close to zero. And for more lights, it still remains very very low.

Nonetheless, for next generation video games of bigger screen sizes of 4K, the overhead for many lights can not be ignored (for 10 lights we already have an overhead of 5ms), and consequently the PBR effects should be added carefully keeping this overhead in mind. One solution is to only compute per pixel the most important lights, and the other ones per vertex (this is what *Unity* does by default).

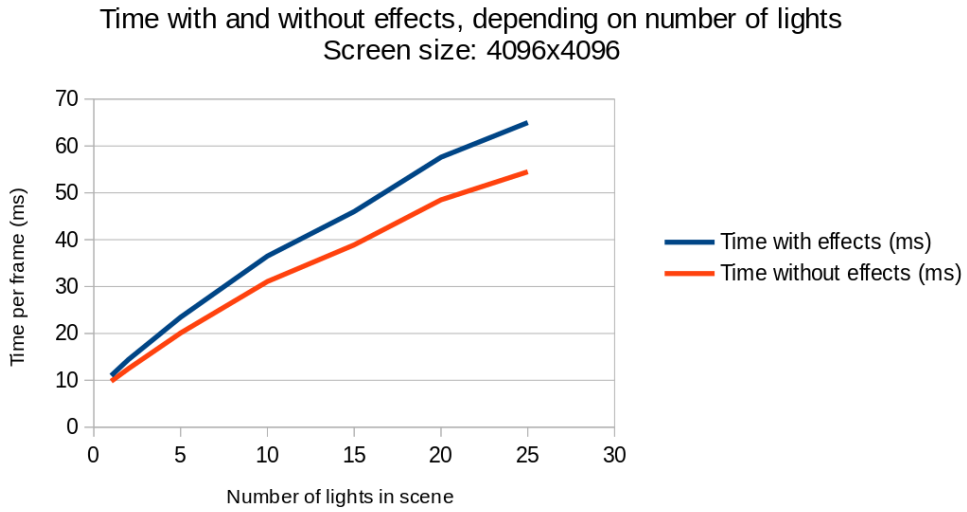

Figure 130: Time with and without effects depending on the number of lights of our scene, with a screen size of 4096x4096.

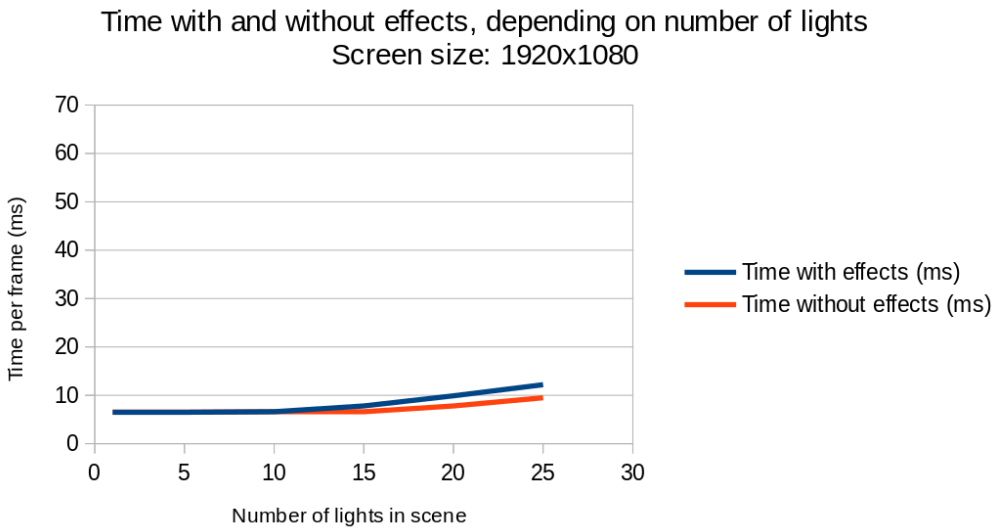

Figure 131: Time with and without effects depending on the number of lights of our scene, with a screen size of 1920x1080.

6. Conclusions

In this thesis we have presented a method to add aging effects and imperfections to models in a very fast and easy way. The implemented tool provides the user with a software specifically tailored to create such effects.

We have covered with success most of the objectives stated at the beginning of it.

First, we have shown that we can create very plausible and realistic effects using only the typical textures seen in a PBR system. And since PBR textures are widely used, we can easily export the effects to be immediately integrated to external software such as game engines.

Next, we have also seen that the choice of using only PBR textures has helped us to create a solution that can keep a low overhead. However, we also know that this choice imposes some constraints that limit the quality and the scope of the results. This is why the implemented tool should only be used for what the purpose it was designed for. For other tasks higher quality effects or offline-rendering, there are many other programs that will do a much better job.

In addition, we have also come up with a work-flow that suits very well the creation of these effects. The proposed layer architecture is modular and very flexible, letting the user author effects that were not even contemplated in this thesis.

Finally, we have implemented a tool that gives real time feedback for most of the effects. Furthermore, we have also designed several GPU oriented algorithms for the creation of masks, such as the edge detection algorithm or the surface blur. These have been useful for our tool, but we hope they prove useful for other fields or research works as well.

Nonetheless, there is still room for improvement, and we are convinced that, with more dedication and time, the results seen in this thesis can be even better under the same performance constraints.

7. Future work

Due to the lack of time and task prioritization, there were some features that were not developed or improved but would be important for the tool. These are:

- An extremely important feature that is missing from the tool is the undo/redo mechanism. This is a very important feature that provides users the ability of correcting their mistakes. It would pose some difficulties and challenges, specially for the brush layers. For these layers, it would be interesting to explore what would be a compact but still fast way of storing the differences between brush layers.
- The brush provided in the tool is a screen space version. However, it would be useful to add another type of brush that adapts to the normal of the surface which is being pointed by the mouse. In addition, the current brush sometimes produces little artifacts in the surfaces oriented with a normal perpendicular to the camera direction, so this would also be a field in which there is room to improve as well.
- As mentioned in section 4.5.3 Pull push, the current seam fix solution for normal maps does not work well in some cases. For this reason, there is still work to do in this area. One solution that would work would be to replicate neighboring triangles around the seam edges. However, since one of the self-imposed restrictions was to not modify the model uv layout, we may have overlapping problems. Maybe we could explore other run-time sampling solutions when filtering, although they might not be fast enough to render them at interactive frame rates. Another solution could use a similar 3D sampling method to the 3D surface blur, but with the normals. Or maybe even just dispose the uv layout restriction and recreate the uv layout as needed, although this would maybe generate more problems than benefits...As said, there is still room to think and improve in this regard.
- As we have seen in 5.1.5 Peeling, problematic case, trying to achieve some effects such as peeling becomes a problem because we can only specify and render one layer of heights. This is why it would be interesting to find a way to create a peeling without adding geometry, and a method to render it. Maybe it is even out of scope for our texture-only methods, which would mean that peeling should be created by relaxing the constraints. In any case, this is an interesting problem that could be improved with further research.
- It would be nice to explore a more visual way of editing or connecting layers between them. For example, a thing that we wanted to try but could not because of time, was a graphical node layer editor. See Figure 132 to know what such a graphical node editor would look like. This provides several advantages:

- It is easier to see what is happening. With a quick glance you can see how masks and effects are connected and how they interact between each other.
- It provides an opportunity to provide a preview of the mask or effect layer result in each node, making it very intuitive and straightforward to identify each node/layer.
- It can be even more modular than layers. With the implemented layer system for this thesis, you can not reuse a mask layer for several effects. However, with a node editor, you can easily reuse the result of a mask in as many inputs as you want.
- It would provide a way to create arbitrarily complex hierarchies of many layer depths instead of just 1 as with the current implementation.
- With a graphical node editor, the user can spatially organize the nodes as wanted. In contrast, with the current layer system, reorganizing them up or down changes the result.

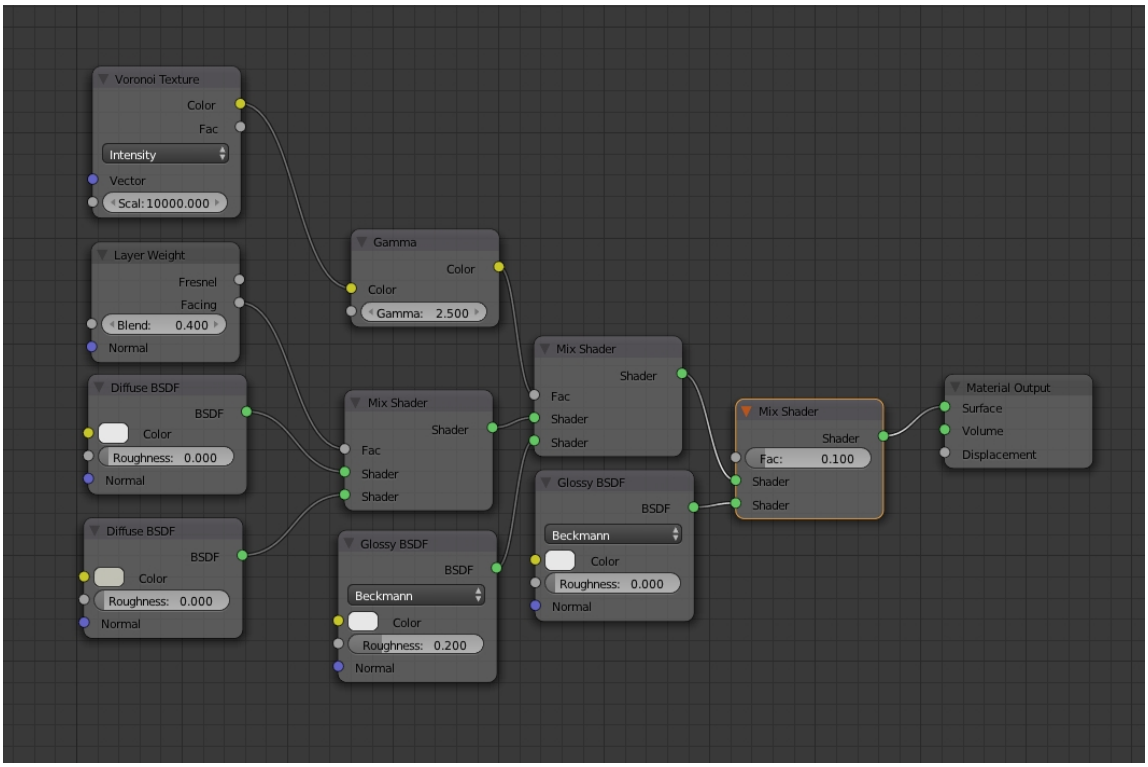

Figure 132: Example of graphical node editor. As you can see, it is much more flexible than the current one-level-nested list of layers approach.

8. Acknowledgements

I would like to thank all the people that helped me in some way with this thesis.

First of all, I would like to thank my thesis supervisor Marc Comino Trinidad and my tutor Carlos Andújar Gran for giving me ideas and advice. Also, thank them for trusting me, accepting the initial idea and letting me implement what I wanted to try.

Next, I would like to thank all my friends, family and other people who tested the program and gave me interesting ideas.

Finally, I would also want to thank Universitat Politècnica de Catalunya and all the people involved in helping me move my thesis presentation to several weeks earlier, specially Isabel Navazo. They were very kind to guide me through the whole process and giving me all the information I needed. And of course, thank also the thesis presentation panel for evaluating it several weeks earlier and outside of the official dates.

Bibliography

[BOS04] Carles Bosch, Xavier Pueyo, Stéphane Mérillou, and Djamchid Ghazanfarpour. "A Physically-Based Model for Rendering Realistic Scratches". In: Computer Graphics Forum 23.3, pages 361-370.

[BSGLSL] Brian Sharpe, GPU-Noise-Lib. URL: https://github.com/BrianSharpe/GPU-Noise-Lib.

[CS03] Yao-Xun Chang, Zen-Chung Shih. "The synthesis of rust in seawater". In: The Visual Computer 19.1 (2003), pages 50-66. URL: https://doi.org/10.1007/s00371-002-0172-0

[GC01] S. Gobron and N. Chiba. "Simulation of peeling using 3D-surface cellular automata". In: Proceedings Ninth Pacific Conference on Computer Graphics and Applications, pages 338-347.

[GP14] Jie Guo and Jin-Gui Pan. "Real-time simulating and rendering of layered dust". In: The Visual Computer 30.6 (2014), pages 797-807. URL: https://doi.org/10.1007/s00371- 014- 0967-9

[HW95] Siu-chi Hsu and Tien-tsin Wong. "Simulating Dust Accumulation". In: IEEE Computer Graphics Appl. (1995), pages 18-22.

[JME] JMonkeyEngine, “Physically Based Rendering: Part Two”. URL: https://wiki.jmonkeyengine.org/ jme3/advanced/pbr_part2.html

[KM09] Kraus, Martin. "The Pull-push Algorithm Revisited - Improvements, Computation of Point Densities and GPU Implementation". In: GRAPP (2009), pages 1-20.

[KMMM02] Kevin Moule, Michael D. McCool. “Efficient Bounded Adaptive Tessellation of Displacement Maps”. In: Graphics Interface, vol. 1, pp. 32-63, 2002. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.4.9115&rep=rep1&type=pdf

[LOGL] Joey de Vries, “Learn Open GL tutorials”. URL: https://learnopengl.com

[MBP18] Muñoz-Pandiella, C. Bosch, N. Mérillou, G. Patow, S. Mérillou and X. Pueyo. "Urban Weathering: Interactive Rendering of Polluted Cities". In: IEEE Transactions on Visualization and Computer Graphics (2018), pages 3239-3252.

[MDGC01] Stephane Merillou, Jean-Michel Dischler, and Djamchid Ghazanfarpour. "Corrosion: Simulating and Rendering". In: Proceedings of Graphics Interface (2001), pages 167-174.

[MDGS01] Stéphane Mérillou, Jean-Michel Dischler, and Djamchid Ghazanfarpour. "Surface scratches: measuring, modeling and rendering". In: The Visual Computer 17.1 (2001), pages 30-45.

[MNCL] Matthias Nießner, Charles Loop. “Analytic Displacement Mapping using Hardware Tessellation”. In: ACM Transactions on Graphics (TOG), 2013, vol. 32, num. 3, p. 26. URL: http://www.niessnerlab.org/papers/2013/3analytic/niessner2013analytic.pdf (2013)

[POC05] Fabio Policarpo, Manuel M. Oliveira, Joao L. D. Comba. “Real-Time Relief Mapping on Arbitrary Polygonal Surfaces”. In: ACM Transactions on Graphics, vol. 24, num. 3, pp. 935. July 2005. (ISSN 0730-0301). URL: http://www.inf.ufrgs.br/~oliveira/pubs_files/Policarpo_Oliveira_ Comba_RTRM_I3D_2005.pdf (205)

[PPD02] Eric Paquette, Pierre Poulin and George Drettakis. "The simulation of paint cracking and peeling". In: Proceedings of Graphics Interface (2002), pages 10-11.

[RGB16] Boris Raymond, Gaël Guennebaud, and Pascal Barla. "Multi-scale Rendering of Scratched Materials Using a Structured SV-BRDF Model". In: ACM Transactions on Graphics 35.4 (2016). URL: http://doi.acm.org/10.1145/2897824.2925945

[RRHM] Rachmadi, Reza. Hariadi, Mochamad, “GPU-Based Ray Tracing Algorithm Using Uniform Grid Structure”, In: Jurnal Ilmiah Ilmu Komputer UPH. 7 (2011), pages 111-116. URL: https://www.researchgate.net/publication/236329553_GPU-Based_Ray_Tracing_Algorithm_Using_Uniform_Grid_Structure/download

[SAT01] David Eberly. “Intersection of Convex Objects: The Method of Separating Axes”. URL: https://www.geometrictools.com/Documentation/MethodOfSeparatingAxes.pdf (2001)

[SGPBR13] Natty Hoffman, “Background: Physics and Math of Shading”. URL: https://blog.selfshadow.com/ publications/s2013-shading-course/hoffman/s2013_pbs_physics_math_notes.pdf

[SGSN05] Stefan Gustavson, Linköping University, Sweden. “Simplex noise demystified”. URL: http://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf (2005)

[WER17] Sebastian Werner, Zdravko Velinov, Wenzel Jakob, and Matthias Hullin. "Scratch Iridescence: Wave-Optical Rendering of Diffractive Surface Structure". In: ACM Transactions on Graphics (TOG), vol. 36, issue 6, article no. 207, pp. 1-20 (November 2017)