UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GUILHERME DO NASCIMENTO OLIVEIRA

Procedural Textures Mapping using Geodesic Distances

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Prof. Dr. João L.D. Comba
Advisor

Prof. Dr. Marcelo Walter
Coadvisor

Porto Alegre, February 2011

CIP – CATALOGING-IN-PUBLICATION

Oliveira, Guilherme do Nascimento

Procedural Textures Mapping using Geodesic Distances / Guilherme do Nascimento Oliveira. – Porto Alegre: PPGC da UFRGS, 2011.

91 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2011. Advisor: João L.D. Comba; Coadvisor: Marcelo Walter.

1. Procedural texturing. 2. Geodesic distance. 3. Distance fields. 4. Texture mapping. 5. Hardware Tessellation. I. Comba, João L.D.. II. Walter, Marcelo. III. Title.

“Everybody got mixed feelings
About the function and the form
Everybody got to elevate
from the norm... ”
— Vital Signs by Rush

ACKNOWLEDGMENTS

A special thanks to professor João Luiz Dihl Comba for all the support with the research and study over the last two years, giving good directions yet with some degree of freedom for choices on research topics. And specially, for trusting some unknown guy from some distant city as a candidate student for a master degree, and for the patience.

Also a great thanks to Rafael Piccin Torchelsen, for the great help with related works specially regarding geodesic distances, and to Marcelo Walter for support with the submitted papers. Thanks to both for the brainstorms, directions, ideas and revisions.

Thanks to all the colleagues of the UFRGS Computer Graphics group for feedback and discussion on the ongoing research, and for all the off topic stuff and great work environment that helps a researcher to keep a part of his sanity, it was very important.

Thanks for all the UFRGS Inf. staff that creates and runs this great facility for study, research and development, providing all the resources necessary for a excellent master degree course. Also great thanks to CNPq for funds that helped me to live well in such a nice city, away from the family, but without regrets.

Of course, great thanks to my family for the trust, patience, funding, and for being always there for me. And great thanks to God for everyone.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Texture mapping is an important technique to add detail to geometric models. Image-based texture mapping is the preferred approach but employs pre-computed images, which are better suited for static patterns. On the other hand, procedural-based texture mapping offers an alternative that rely on functions to describe texturing patterns. This allows more flexibility to define patterns in dynamic scenes, while also having a more compact representation and more control for parametric adjustments on the texture visual appearance. When mapped with 3D coordinates, the procedural textures do not consider the model surface, and with 2D mapping the coordinates must be defined in a coherent way, which for complex models is not an easy task. In this work we give a introduction to procedural texturing and texture mapping, and introduce *GeoTextures*, an original approach that uses geodesic distance defined from multiple sources at different locations over the surface of the model. The geodesic distance is passed as a parameter that allows the shape of the model to be considered in the definition of the procedural texture. We validate the proposal using procedural textures that are applied in real-time to complex surfaces, and show examples that change both the shading of the models, as well as their shape using hardware-based tessellation.

Keywords: Procedural texturing, Geodesic distance, Distance fields, Texture mapping, Hardware Tessellation.

Mapeamento de Texturas Procedurais usando Distâncias Geodésicas

RESUMO

O mapeamento de texturas é uma técnica bastante importante para adicionar detalhamento a modelos geométricos. O mapeamento de texturas baseadas em imagens costuma ser a abordagem preferida, mas faz uso de imagens pré-computadas que são mais adequadas à representação de padrões estáticos. Por outro lado, texturas procedurais oferecem uma alternativa que depende de funções para descrever os padrões das texturas. Elas garantem mais flexibilidade na definição dos padrões em cenas dinâmicas, tendo ainda uma representação mais compacta e dando um maior controle da aparência da textura através do ajuste de parâmetros. Quando mapeadas por coordenadas 3D, as texturas procedurais não consideram a forma da superfície do modelo, e com coordenadas 2D torna-se necessária a definição dessas coordenadas de forma coerente, que, em modelos complexos ,não é uma tarefa simples. Neste trabalho nós introduzimos o leitor às texturas procedurais e ao mapeamento de texturas, então apresentamos *GeoTextures*, uma nova abordagem que faz uso de distâncias geodésicas definidas com base em múltiplos pontos de origem sobre a superfície do modelo. As distâncias geodésicas são passadas como parâmetros que permitem que a textura procedural se adeqüe ao relevo do modelo texturizado. Nós validamos a proposta ao usar alguns exemplos de texturas procedurais aplicadas em tempo real na texturização de superfícies complexas, mudando tanto a textura do modelo como a forma, através do uso de tesselagem em hardware.

Palavras-chave: Texturas Procedurais, Distâncias Geodésicas, Campos de Distâncias, Mapeamento de Texturas, Tesselagem em Hardware.

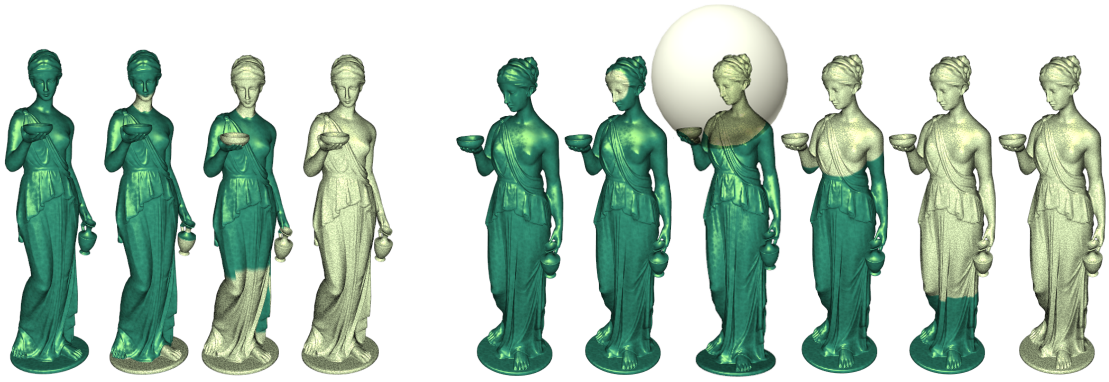
1 INTRODUCTION

For a long time the common approach for rendering objects of some visual complexity has been separating the object overall shape from the fine scale variations on the surface. While the first one represents the geometry and topology of the object with geometric primitives as vertexes, edges and faces, representing the minor details of the surface in the same way would require a large number of geometric primitives to render and to store. To add such minor details to the rendered geometry the most common solution is the use of textures. By texture mapping, colors from an image are mapped to the fragments of the surface to be rendered. In this sense, textures have been the main solution to the lack of detail in geometric models. They are widely used in several applications in the computer graphics area both as a mean of enrich geometric objects with a higher level of detail and in the simulation of a variety of visual effects like reflectivity and shadowing, as well as being essential in dynamic scenes, like movies, animations and games. Given such importance of textures in computer graphics, texturing has been a quite explored area where a lot of effort has been put to reach even greater complexity and fidelity levels in the final scenes. The works in this area seek improvements in the appearance of the textured objects through different methods for the creation of the textures, its placement in the target surface and the animation.

1.1 Motivation

Texture mapping is a viable solution since it allows adding detail to often simplified geometric models. Image-based texture mapping is the most used approach, since many applications only require static textures, and is implemented in every graphics hardware available nowadays. For the creation of texture patterns that can be described by functions, procedural texturing surfaced as an alternative. In comparison to the image-based approach, procedural textures have a smaller memory footprint, but requires the definition of a function to describe texture appearance (which often is non-trivial). This comparison favors the image-based approach since most current systems have plenty of memory available. However, for describing dynamic texture patterns the use of procedural textures become more appealing, since image-based approaches require increased memory usage to store textures for each time-step, not to mention the tedious creation process of several individual textures. Following the saying that an image is worth a thousand words, a model is then worth a thousand images, and particularly for creating dynamic texture patterns, the model offered by procedural textures is richer.

Despite its expressive power, procedural texturing still has some limitations. For instance, the space in which the texture mapping is defined directly impacts the final result. In 2D, the interpolation along the surface of a model depends on the texture coordinates



(a) (b)

Figure 1.1: (a) Texturing using the 2D texture mapping space. A sand-like feature start propagating from a single point, but soon appears on different spots of the surface due to duplicated texture coordinates. (b) Texturing using the 3D object space. The sand-like texture propagates vertically down from the head along the surface. However, it jumps to the bowl in the right hand before passing by the right elbow. We highlight the spherical nature of the propagation in the third image of the sequence.

assigned to each mesh vertex. For the resulting texture to show the desired appearance and avoid discontinuities in the appearance, texture coordinates must be consistently assigned. In Figure 1.1(a) we illustrate what might happen when texture coordinates are not properly assigned. In this example, a sand texture is defined over a mesh as a function of the distance to a given point – in this case the point with texture coordinates $(0, 0)$. A sequence of four images from left to right shows the resulting texture when the distance threshold is set to $0.0, 0.12, 0.5$ and 1.0 – assuming the distance is normalized between 0 (closest) to 1 (farthest). In the second image sand starts to spread from four distinct points: the neck, the floor, and two distinct places in the vase. This happens because texture coordinates are duplicated across vertices of the surface. In the third image features did not spread uniformly in the dress, which reveals that texture coordinates were not consistently defined. An even worst problem happens when texture coordinates are not defined. A consistent texture mapping would be a global mapping over the entire surface, however such approach results in high distortion for complex surfaces. To avoid distortion, the mesh can be broken into charts and use separate mappings(TORCHELSEN et al., 2009).

Procedural texturing using the 3D vertex coordinates as texture coordinates is called solid texturing (EBERT et al., 2002a; PEACHEY, 1985). This approach is suitable for cases where the surface has to be textured as if it was carved from some solid material, such as a chair carved from a wood trunk. Features defined in 3D are independent from the 2D texture mapping coordinates and have no correlation with the mesh surface. Even if the function is continuous, texture patterns do not follow the model surface and for irregular shapes may show incoherent behavior. In figure 1.1(b) we illustrate the solid texturing approach. A given point is chosen in 3D as the source of the propagation. In this example corresponds to a point over the forehead of the model, and the sand feature is propagate to the area within a threshold distance to the source. As the threshold increases, the feature seems to spread uniformly along the surface, but in the third image the feature appears at the bowl before even reaching the right elbow, revealing the spherical nature of the propagation front.

1.2 Objective

Simple Euclidian distance will not be enough for what we want, the solution is geodesic distances. Geodesic is a generalization of the idea of straight line to curved spaces. With a metric, the geodesic is then the shortest path between two points in a surface, and the distance we need is the length of such path. We developed *GeoTextures*, as an approach that extends procedural texturing to take into account the geodesic distance from multiple sources, and applies it to define features on time-varying procedural textures over complex surfaces. Using the geodesic distance as parameter, we are able to define procedural texturing patterns that follow the model surface while introducing no discontinuities. We demonstrate results of our technique using several constructions, including propagation from multiple sources, noise textures and displacement mapping effects using the tessellation feature of recent graphics hardware.

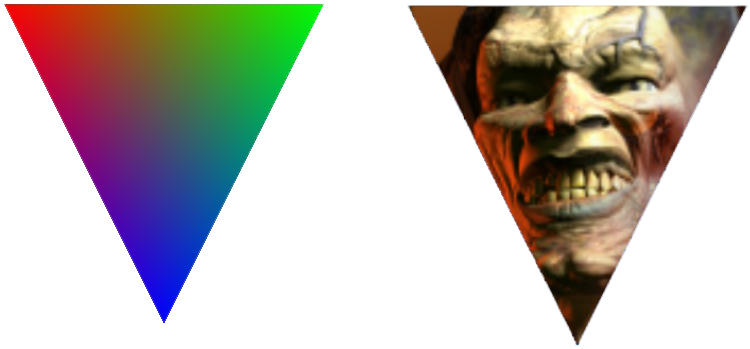
1.3 Structure of Thesis

In this work we present a different method to map procedural textures on the surface of models that have a considerable complexity (closed surfaces with holes). The main application of this new mapping scheme is to allow the diffusion of the procedural texture over such surface in a coherent way. To provide a better understanding of our technique, as well as our motivation and make clear its relevance as a mapping scheme, we organized this work as follows: in Chapter 2 we present the basic texturing method, we address some issues inherent to the texturing process like filtering and uv maps, and also present the different classes of textures. In Chapter 3 we introduce procedural textures, the class of texture our method is designed for. In Chapter 4 we show the importance of mesh parameterization in the texturing schemes, we also present the most relevant parameterization methods available. In Chapter 5 we present our new mapping scheme, *Geotextures*, and the results. Finally, in Chapter 6 we conclude our work.

2 TEXTURING

Surface texturing is a process somewhat similar to the mesh coloring one, except that the colors to be used are defined in color maps. To color a geometric model without textures, one can define a color at each vertex of the mesh. The color of a given fragment will be estimated, based on the colors of each vertex of the face were the fragment is, through linear interpolation. The number of fragments changes depending on the camera position, and is usually very high, making the assignment of a color for each fragment impractical, so interpolation is essential.

Texturing follows this same principle but instead of defining colors to the vertex and using the interpolated values in the fragments, each vertex will have a second coordinate attribute (texture coordinates), this one not related to the geometric position but to the position of a color value in a color map. A color map, also known as *texture*, is usually a 2D set of color values (i.e., an image), but 1D and 3D textures can also be used in some cases. Such small modification in the process of coloring meshes represent a great improvement in the final result, the surface can have a amount of detail that would require a very complex geometry if using only vertex colors. However, problems that arise specially related to the difference between the resolution of the surface and the texture, and to the assignment of texture coordinates, turns the texturing scheme a lot more complicated.

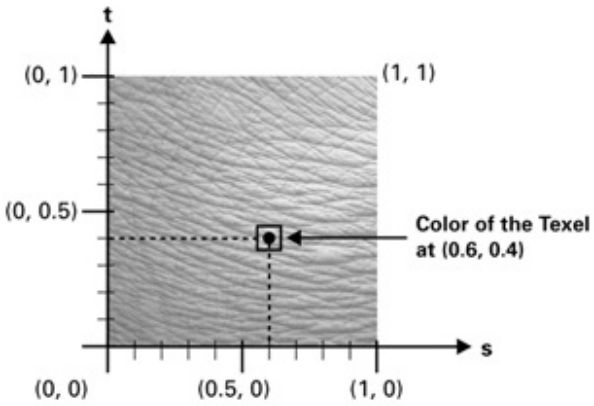


(a) (b)

Figure 2.1: (a) A triangle rendered with vertex colors (Red for the left vertex, green for the right, and blue for the bottom one). Colors inside the triangle are interpolated from the colors defined in the vertices. (b) A triangle with a texture mapped. (FERNANDO; KILGARD, 2003)

In figure 2.2, the diagram shows a query for the color to be applied in a fragment whose texture coordinates are 0.6 and 0.4. The most common notation for texture coor-

dinates is *u* and *v* (for this reason the process of assigning colors from an image to the surface of a polygon is called *uv mapping*), but the letters *s* and *t* are used as well. The *uv* coordinates are real numbers, and since the size of the dimensions and number of *texels* (color units in the texture) changes from texture to texture, they are usually normalized to range from 0.0 to 1.0, but the interval can be different depending on how the graphic API was configured.



(a)

Figure 2.2: Sampling a color from the texture for the fragment whose texture coordinates are (0.6, 0.4). (FERNANDO; KILGARD, 2003)

The process is the same for 1D and 3D textures, the only difference is the number of texture coordinates for each case, 1 and 3 coordinates respectively. 1D textures are used to define color gradients along the surface. *Cel Shading* is a good example of application of 1D textures. In cel shading the 1D texture is used to set the color of surface fragment based on how much light it receives. Due to its cartoonish look it became very popular within the game industry. 3D texture are also very useful, they are perfect for volumetric objects that can be carved and sliced, like fruits, as shown in (TAKAYAMA et al., 2008).

(a) (b)

Figure 2.3: (a) A 1D texture mapped over a surface to represent different illumination levels. (b) A solid object textured with a 3D texture to resemble a fruit. (TAKAYAMA et al., 2008)

As said before, the difference between the area of the surface and the size of the texture represents a new problem. The number of fragments is variable and so are the color values to be fetched from the texture, however the number of texels in the texture

is constant, it will make the image mapped into the surface look blurry, with contrast loss, when we have more fragments than texels. In the inverse situation the result will be an image with loss of detail, a distorted version of the original one. This distortion can appear in the form of jaggy, blocky or shimmering feel in the image. This effect is called *aliasing*. In signal processing and the related fields, the term *aliasing* refers to the effect that makes a signal reconstructed from samples to be different from the original one due to improper sampling. An aliased signal have distortion and artifacts not present in the original one. The techniques to solve aliasing problems are called *anti-aliasing*. Since aliasing can ruin the appearance of a material, anti-aliasing is a very important step of the texturing process.

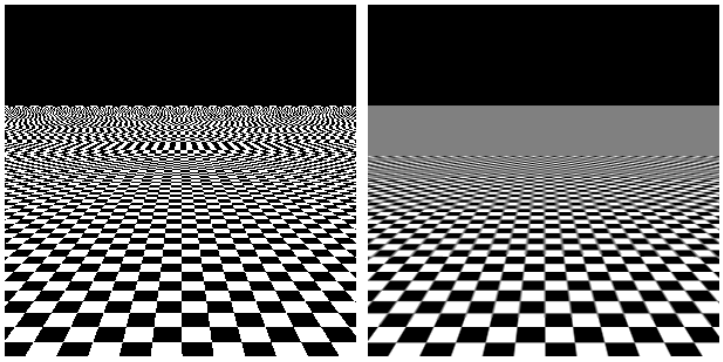


Figure 2.4: (Left) Aliasing on a plane due to improper sampling. (Right) The same scene now with an anti-aliasing filter.

To handle problems of texture filtering, graphics API and cards offer several filtering options, to be applied on each of two cases: magnification and minification. The filter chosen for the magnification case will be applied when the texture is smaller than the area to be textured (i.e., close surfaces), it will have to be scaled up and can became blurry. Minification filter will be used for sampling when the texture is larger than the target surface (i.e., far away surfaces), it is when aliasing can occur.

Before understanding the filtering options, it is important to understand the *mipmapping* technique. Mipmapping is an anti-aliasing scheme introduced in (WILLIAMS, 1983), and is now a standard technique for filtering during texture minification. Different from the texture magnification case, where few samples are need to determinate the color for a fragment, in the texture minification several texels might have to be sampled for a accurate result, with the number increasing even more as the surface moves farther away from the camera, not discarding the possibility of the whole texture falling into a single rendered fragment when the all the texels would have to be fetched by the filter leading to a very expensive operation. The solution, in mipmapping, is given by the creation and storage of a set of pre-filtered versions of the original texture with decreasing sizes. At rendering stage, as the surface moves away from the camera, the smaller pre-filtered versions of the texture are used for sampling instead of the original one, eliminating aliasing effects from the result. The smaller images are related to mipmapping levels, being the original and largest one level 0, with level increasing as the size decreases. The level to be used for sampling is then defined based on the distance from the fragment to the camera.

The provided filtering options are:

- **Nearest-neighbor filter**: It is the most simple texture filter. The color chosen is the one of the closest texel to the uv coordinate of the given fragment. While being the

Figure 2.5: Example of mipmap levels.

least expensive method it results in several artifacts, being blockness in magnification, and jaggy and shimmering look in minification.

- **Nearest-neighbor filter**: This method adds mipmapping functionality to the previous one, reducing the jaggy and shimmering look in minification, but magnification still is blocky since mipmapping affects only texture minification.

- **Bilinear filtering**: This method is a more clever version of the nearest-neighbor filter. The four closest texels are sampled, and the final color is the combination of the colors of the sampled texels weight averaged according to the distance from each texel to the center of the fragment. The interpolation of the colors creates a smooth gradient eliminating the blockness in texture magnification. This method can also be used with mipmapping to solve the problems of texture minification.

- **Trilinear filtering**: The purpose of trilinear filtering is to solve a problem of the mipmapping technique when used with bilinear filtering. When the renderer changes the mipmap level to be used for sampling, it often causes a visible change in the quality of the result. The solution given is finding one color through bilinear filtering in each of the two closest mipmap levels and then making a linear interpolation of the two results. It then creates a smooth degradation in the quality of the result as the surface moves farther away, instead of the previous abrupt and noticeable quality loss when the mipmap level changes.

- **Anisotropic filtering**: The previous filters take the same amount of samples in each direction in the neighborhood of the uv coordinate provided, they are isotropic filters ('iso' for *same* and 'tropic' relates to direction). The problem appears when the surface is at a very oblique viewing angle. In such case the image resolution varies at a different ratio at each axis, and now using the same amount of samples in each direction would make one to be down-sampled resulting in a blurred appearance. The solution given by anisotropic filtering is sampling a trapezoid according to the viewing angle. The major problem of the technique is that it is extremely bandwidth-intensive as each sample for the anisotropic filtering is a mipmapped sample, so with Trilinear filtering is used, 16 anisotropic filtering samples would

take 128 texture fetches (4 samples twice 2 mipmap levels for each anisotropic filtering sample).

Figure 2.6: Comparison of a game scene without anisotropic filtering and with anisotropic filtering with 16 samples. Notice how the white stripes in the road become blurry as they get distant from the viewer when anisotropic filtering is off.

Another problem of texture mapping is the definition of the uv coordinates for the vertices of the model. If its geometry has few vertices it might not be a such a great problem, but for models with dozens, hundreds and thousands primitives, choosing texture coordinates for each one is impractical. The common solution is to have an algorithm assign these coordinates for us. The process maps the geometry of the object from the 3D space to the plane, and then it is up to designer to paint the image over the 2D geometry. The process of mapping a 3D geometry to the plane is called *parameterization*. The coordinates of each vertex in the plane will be its texture coordinates in the original 3D geometry, and the final image created, painted over 2D geometry, will be the texture applied, also known as *uv map*. Mesh parameterization is a very technique for texturing and is a quite complex problem. Due to its importance in texturing and in our mapping method, we reserved a whole chapter related to parameterization (Chapter 4).

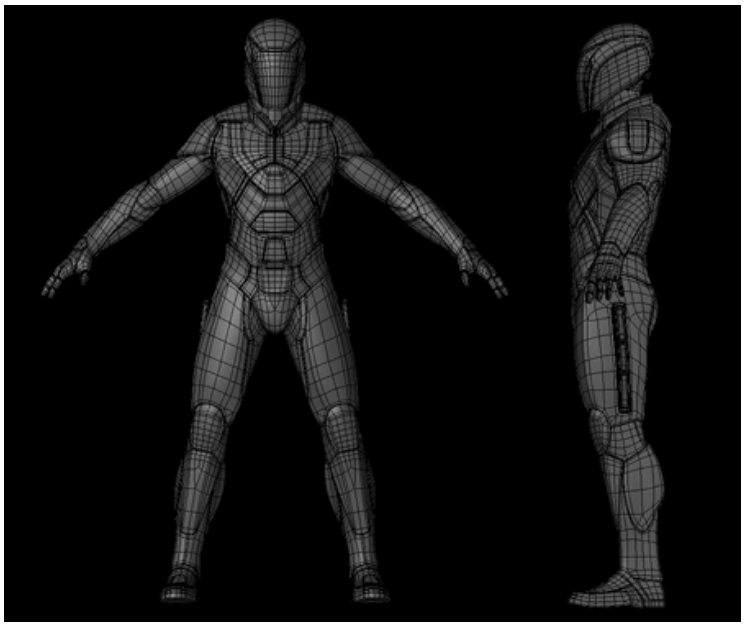


Figure 2.7: A mesh created to represent a character of the TRON serie. (image from (TEN24, 2011))

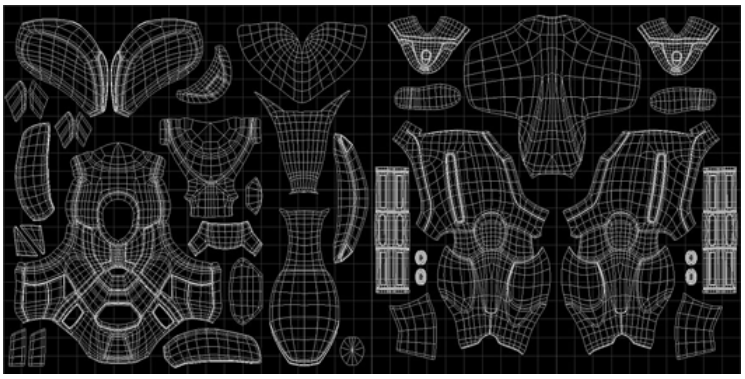
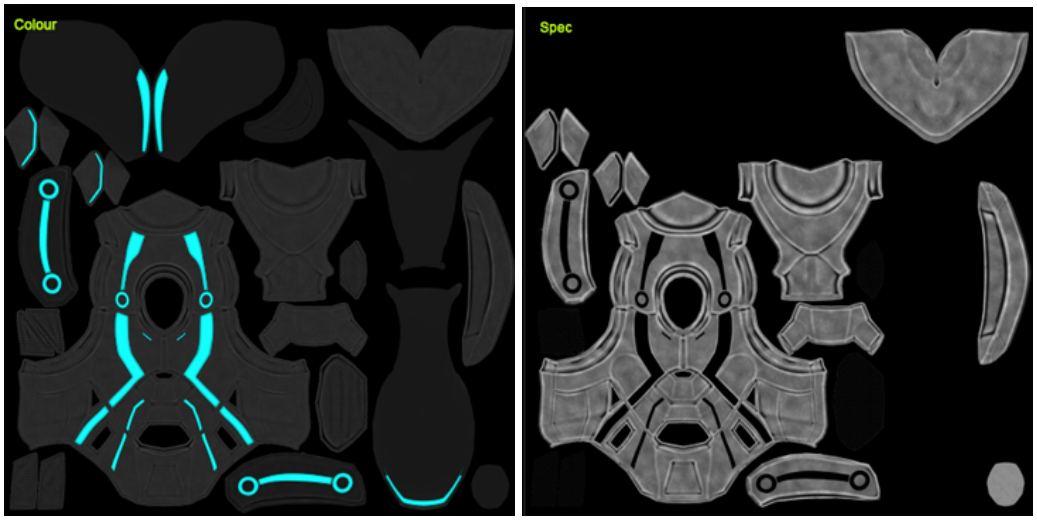


Figure 2.8: UV Map created by parameterization for the mesh in figure 2.7. (image from (TEN24, 2011))



(a) (b)

(c) (d)

Figure 2.9: Four textures created with the UV Map of figure 2.8 (only the left half of the uv map). Each texture represents a different property of the final materia: (a) Color, (b) Specular, (c) Gloss and (d) Luminosity. (images from (TEN24, 2011))

Figure 2.10: Final rendering of the model of figure 2.7 textured. (image from (TEN24, 2011))

The last area of research related to texturing of relevance to our work is texture synthesis. Texture synthesis focus on the creation of more complex, interesting and realistic looking textures and ways of automate such task. According to the creation method, we can classify textures as:

1. **Image-based textures**: Textures created by digitalized images taken from the real world or created by an artist with a graphics design tool.

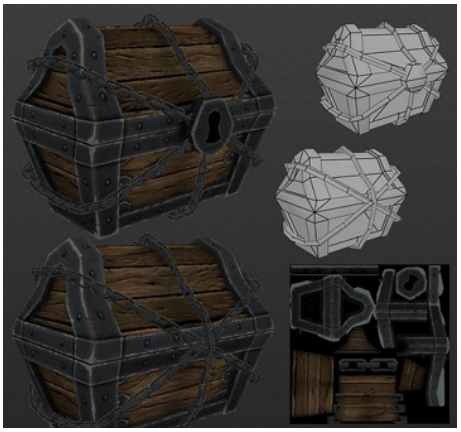


Figure 2.11: A chest model textured with a image-based texture painted by a digital artist.

2. **Procedural textures**: In contrast with the traditional texture creation, procedural textures are created by algorithms that try to replicate the appearance of a given material. By exploiting the crescent computational power of CPUs and GPUs, this kind of texture can created images with a level of detail that would be hard to achieve

with the traditional painting approach, being widely used also due to other advantages as the compact representation and independency of resolution. The textures can be generated in a preprocessing stage and stored as images to be used when rendering, like the image-based method, or the function that determinates the color of the fragments can be evaluated in real-time while rendering, sparing memory storage resources. Our mapping solution is designed for procedural textures, so the next chapter further explore the field of procedural texture synthesis.

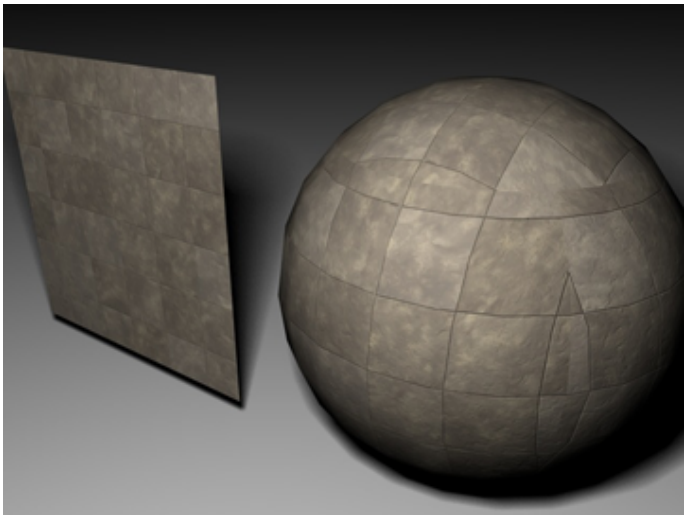


Figure 2.12: A plane and a sphere textured with procedurally generated stone-like texture.

3. **Textures synthesized based on samples**: This is a quite new method where textures are created based on several images. Methods like the ones presented in (LEFEBVRE; NEYRET, 2003), (TURK, 2001), (BROOKS; DODGSON, 2002) and (MATUSIK; ZWICKER; DURAND, 2005) try to create a new pattern by extracting the desired characteristics from other images. In (BROOKS; DODGSON, 2005) such methodology is applied with procedural textures, and there are also works where evolutionary genetic models are explored to synthesize textures (HEWGILL; ROSS, 2003), (WIENS; ROSS, 2002) .

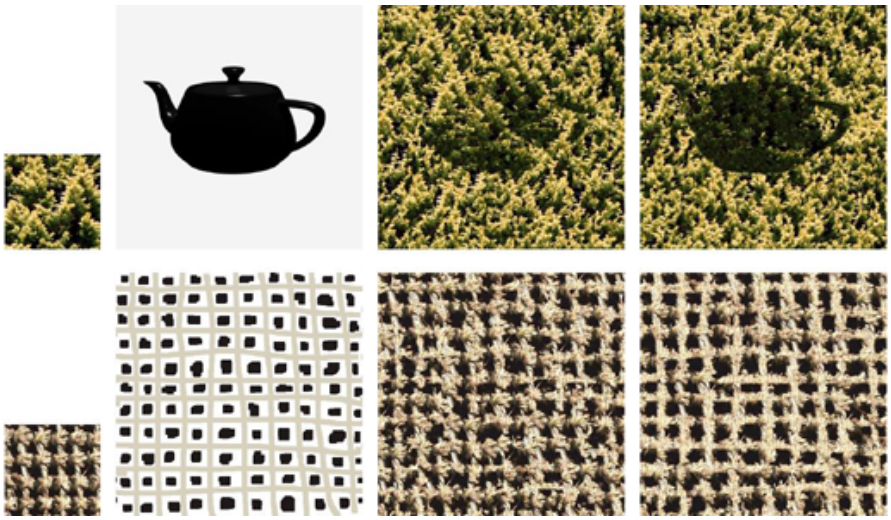


Figure 2.13: Textures created by the combination of a image and a pattern.

3 PROCEDURAL TEXTURES

Procedural techniques have been present through the whole evolution process of computer graphics. Since the early stages, methods to create geometry and color surfaces already included procedural techniques in their definition. The use of such techniques became even more frequent at late 80's when started to be applied in the creation of realistic images that tried to simulate several natural materials like wood, stones and marble, being later extended to modeling, making possible the simulation of water, steam, fire, smoke and even terrain.

The use of procedural methods became of great importance in the creation of realistic images and animations, and lately, the ever growing processing power of the CPUs and GPUs, as well as the programmable stages of the rendering pipeline represented a great leap forward in such context, allowing the execution and parameter adjustment of this techniques in real-time and consequently the creation of even more complex and realistic models and effects.

But what is a procedural technique exactly? In (EBERT et al., 2002b) a procedural technique is defined as an algorithm that specifies a given characteristic to a model or computer generated effect. The term 'procedural' has been used in computer science to make a distinction between entities described by program codes and those defined by data structures. As a example, there is a distinction in the artificial intelligence area between the procedural representations of knowledge and the declarative ones. Obviously every task performed by a computer has a procedural nature at some stage, the same way that there is also always a declarative part, like the parameters of a procedure. Taking texture mapping as the context, the image to be mapped represents the declarative part, as for the mapping algorithm, it is the procedural one.

As seen, the classification itself becomes a quite subjective task. The usual method is to classify as procedural, the methods whose results are affected mostly by modifications on the algorithm instead of changes in the used data, however a procedural texture provided in a encapsulated way, such that the user cannot make changes in the source code, is still a procedural texture. Related to textures, the most coherent manner to classify them as procedural and not procedural, is to know that procedural textures are synthesized by algorithms instead of being only digitalized images or created manually by an artist. As an example we can take a procedural texture created to resemble a marble surface; in contrast to the image-based method where the color of each point in the surface would be given by a sample of the digitalized image, our procedural marble is defined by an algorithm that through a set of mathematical functions evaluates the color to be applied in each point of the surface based on its position.

As will be shown with major detail in the procedural texture context along the next sections of this chapter, the procedural creation methods have very interesting character-

istics. As one of the most important we can take the abstraction. Instead of explicitly specifying and storing information related to each detail of the model expected, everything is represented by a function that will lead to the intended feature in the final result and returns the desired information in the moment it is required. Such implicit form of the model represents memory saving, reduces the time to specify a high number of features in the image as the task is now up to the machine and allows an easy adaptation of the result to any desired resolution.

Another interesting feature of the procedural methods is that the result can be controlled by the functions parameters. The input of the algorithm can be changed to give different outputs, and can be done in execution stage, thanks to the performance of current hardware available, ensuring a level of dynamicity to the final result. Parametric control also has the advantage of improving flexibility, making it easier to achieve different results but with similar features. The programmer needs only to represent the 'look' of the expected result instead of restrict it to an exact image, and by changing the parameters of the functions create new different images but similar features with no need to redefine low level details.

Although the characteristics presented before seems to justify an intensive use of procedural textures, there are some problems that motivate the use of the other class of texture in different cases. The difficulty to create a procedural texture is one of these problems. Usually procedural patterns are not so trivial to create and they can become quite hard to program as the desired pattern gets more complex. Parameters adjustment is also very important and sometimes finding the right value or range of values to result in a given look can be hard. The difficult creation and control of procedural textures can be enough to discard such texturing method and use the image-based form instead, as it is easier to predict the final textured model by using painted or digitalized images. The tradeoff between processing and storage is also an aspect to be considered, because even though the processing power of today hardware keep increasing, the GPUs are optimized for texture fetches giving advantage to the image-based scheme, and the complexity of some patterns can require algorithms that are very expensive, in processing terms, that would slow down the rendering stage. Lastly, problems that are usually solved in the image-based approach with help from the hardware itself like aliasing, need to be addressed by the programmer himself when developing the procedural texture.

In procedural textures, we have two classes of procedural methods: explicit methods and implicit methods. The explicit methods gives us explicitly a point that make a shape, as for the implicit methods, they answer some query related to a point given its position. A texture pattern is represented by a function defined in the space where the texture is to be applied. In this sense, we can understand the explicit methods as a question like 'what is the y coordinate of the point P that belongs to the function F where the x coordinate is the given value?'. The explicit methods generate the patterns in a fixed order, that usually is not fit to the rendering process. In most rendering processes the use of a procedural texture defined by a explicit method would imply in running the texturing algorithm in a prepass stage, that would store the result in a image buffer to be queried later for the true texturing, like in the image-based approach, losing all of the advantages related to procedural textures.

With the implicit methods, however, the answer we seek has a question like: 'given a function F and a point P, whats is the value evaluated by F in P?'. The evaluated value will be then used as a property of the surface in the given point, like the color. We see that implicit method have no ordering restriction, so they are fit for real-time

texturing, once that in both ray tracers and depth buffers renderers, the color must be evaluated following an order defined by the rendering algorithm and not by the texture creation process. This parallel, order independent, position related classification paradigm turns the today shaders into a proper environment for the synthesis of implicit procedural textures.

3.1 Creating patterns

Create procedural textures can be a very difficult task. In this section we give some direction to make the process easier and also show an example of simple pattern procedurally created. What is usually done is look for an existing procedural texture that somehow resemble the appearance expected to achieve with the new one and make adjustments towards the desired result, but sometimes no such model exists or is available to be used. In such case, to elaborate a model from the beginning the common approach is to take as starting point the observation of existing images where the pattern can be found, and even its occurrence in the real world. Most models can be divided in two stages: the generation of the pattern and the shading model. The generation of the pattern defines the pattern of the texture and its properties along the surface, information to be used by the shading model. The shading model deals with illumination aspects, it purpose is to simulate the final appearance of the material based in lights in the scene affects the surface and the texture.

Bellow is the source code for a procedural texture, defined in the fragment shader and written in the Cg shading language, to texture a plane with a simple pattern resembling bricks, one more complex form of the quads patterns widely used in texture creation. Concepts related to shaders use and programming as well as the used programming languages are out of the scope of this work, being used only as a mean to exemplify the implementation of the textures, we assume that the reader has some prior knowledge of this concepts, if not, we refer the interested reader to (FERNANDO; KILGARD, 2003) and (ROST et al., 2009) for introduction on the subject.

```
#define BRICKWIDTH 0.25
#define BRICKHEIGHT 0.08
#define MOTARTHICKNESS 0.01

#define BMWIDTH (BRICKWIDTH + MOTARTHICKNESS)
#define BMHEIGHT (BRICKHEIGHT + MOTARTHICKNESS)

#define MWF (MOTARTHICKNESS*0.5/BMWIDTH)
#define MHF (MOTARTHICKNESS*0.5/BMHEIGHT)

struct brick_f_Output {
   float3 color : COLOR;
};

brick_f_Output brick_f(uniform float3 Cbrick = {0.5, 0.15, 0.14},
             uniform float3 Cmortar = {0.5, 0.5, 0.5},
             float3 position : TEXCOORD0)
{
  float4 Ct;
```

```
  float ss, tt, sbrick, tbrick, w, h;
  float scoord = position.x;
  float tcoord = position.y;

  ss = scoord / BMWIDTH;
  tt = tcoord / BMHEIGHT;

  if (frac(tt*0.5) > 0.5)
    ss += 0.5;

  sbrick = floor(ss);
  tbrick = floor(tt);
  ss -= sbrick;
  tt -= tbrick;

  w = step(MWF,ss) - step(1-MWF,ss);
  h = step(MHF,tt) - step(1-MHF,tt);

  brick_f_Output OUT;
  OUT.color = lerp(Cmortar, Cbrick, w*h);
  return OUT;
}
```

The prior code is a clear example of an implicit procedural method. The function defines a grid that will represent the mortar layers that tie the bricks together, then the color of each fragment is defined based on the grid function.

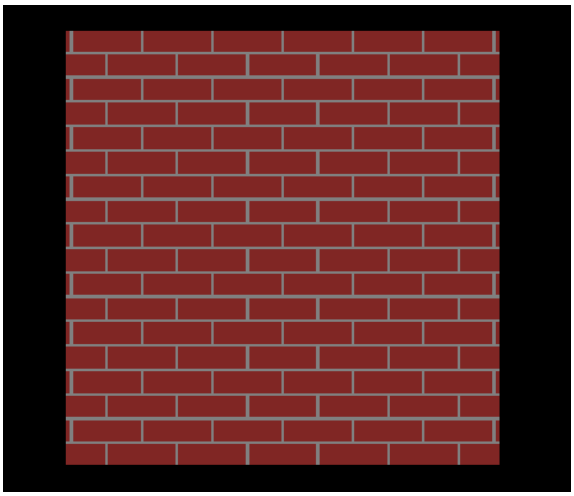


Figure 3.1: A plane procedurally textured with a brick pattern.

3.2 Adding Chaos

Even though the resulting texture in the prior example does resemble a brick wall, it is still far from the real thing. One of the reasons is that the pattern is very regular. It is important to understand that realism is not attached to perfectness, not in the sense of

regularity. We notice irregularities present in almost every real texture that we may want to simulate with a procedural texture. Taking a brick wall as example, there is always some variation in the mortar layers between the bricks and even in the bricks themselves. This irregularity can be seem in the dimensions, color of the materials, in discontinuities in the shapes like a brick with a broken edge, and even in minor relief variations along the surface.

The tool to give such irregular aspect to the procedural textures is a kind of function known as *noise*. The noise function is a function that seems random in the results and can be used to add the desired level of chaos to the procedural textures, breaking the artificial regular appearance of the patterns in the results. By classifying something in this work as random we will be talking about something that only seems random, a pseudo-randomness, i.e., there is a pattern, however it is not so clear. It is unusual the application of true randomness in computer science and in the procedural texturing is an undesired behavior once that compromises the controllability of the patterns generated.

The kind of noise that is truly random is called *whitenoise*, it is a set of values without any correlation. A common example of white noise is the image created by a television display when a channel is selected while the carrier is off. Every image displayed is different and unrelated.

White noise is not an appropriated solution to remove the regularity of the procedural patterns, because it would always create different results even when the same parameters were used, resulting in a different image every time a frame of the scene is rendered. Applying *white noise* in the brick shader for example, would give us a different brick wall every frame. This kind of behavior is undesirable, since we expect that the texture remains the same independently of the time and the position of the camera, and even when we want an animated texture, that changes its appearance over time, we want it to change in a coherent way between consecutive frames, and not an abrupt change with images completely different.

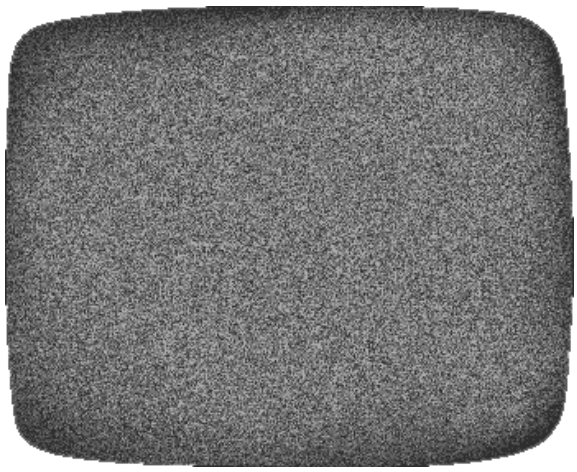


Figure 3.2: White noise in a television screen. The values have no spatial or temporal coherence.

Pseudo-random numbers generators are the perfect solution to this kind of application. They can represent a good approximation of *white noise*, but providing a level of control by using a set of parameters to create its results and ensuring that different evaluations of the procedure with the same parameters will lead to the same results. In procedural textures, these parameters are usually related to the texture coordinates of the surface

fragment. Ahead we present some *noise* functions widely used with procedural texturing.

- **Lattice Noise**

  Lattice noise is a simple and efficient implementation of *noise* serving as the base for many of the other used methods to create *noise*. First a regular grid is defined where every vertex will associated a pseudo-random integer number to a point in the space to be textured. Later, real numbers will be assigned to the points inside each cell of the grid through the interpolation of the values of the four vertices that define the given cell. The quality of the noise function will then depend on the interpolation method used.

- **Value Noise**

  Value noise is a variation of *lattice noise* where the pseudo-random numbers in the grid vertices range from -1 to 1 only, limiting the results of the function to this interval of real numbers. Due to the regular grid used to place the pseudo-random values, using simple lineal interpolation to evaluate the values inside the cells leads to a grid-like appearance in the resulting image, with visible horizontal and vertical artifacts and sharp borders. To reduce this grid-like look, interpolation methods of higher order are used, also ensuring the continuity of the function derivatives.

Figure 3.3: Image created by a *value noise* function.

- **Gradient Noise**

  Gradient Noise is also based in *lattice noise*, however, instead of assign only one pseudo-random number to each vertex in the grid, a vector is assigned. This vector is usually transformed into a unit vector and represents the gradient of the *noise* function in that point. Different from *lattice noise* and *value noise* functions, in *gradient noise* the value of the function on each vertex of the grid is zero. The value of the function for points inside the cells is found by means of trilinear interpolation of the vectors of each vertex that defines the cell. This regularity of zeros also turns to be a problem as it can also give a grid-like appearance to the final image. The popular *Perlin noise*, the noise function described by Ken Perlin in (PERLIN, 1985) was the first implementation of *gradient noise*.

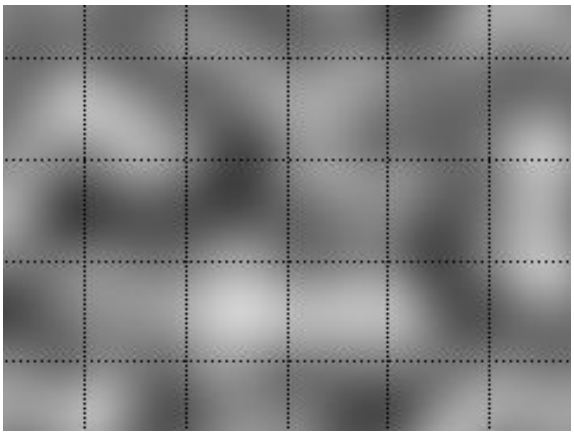


Figure 3.4: I
mage created by the *gradient noise* function. The grid for the placement of the vectors is depicted in dotted lines.

- **Value-Gradient Noise**

  Value-Gradient noise is an implementation that tries to reduce de grid-like look that can be noticed with the prior methods. In *Value-Gradient noise*, *value noise* is combined with *gradient noise* and the final function becomes a weighted sum of the values evaluated by each of the two functions in the given point.

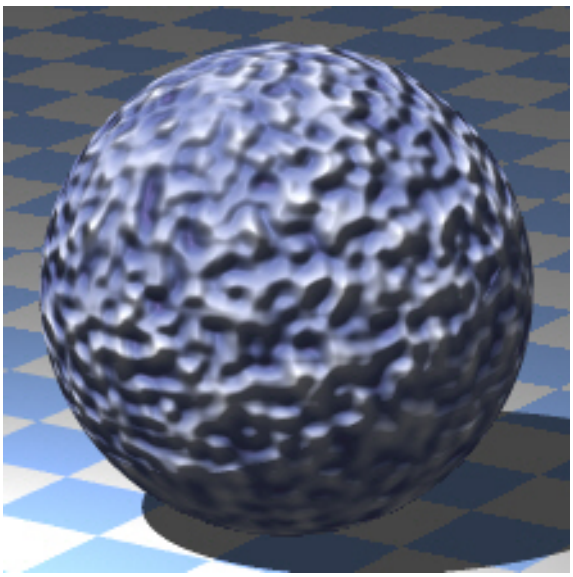


Figure 3.5: Sphere textured with *value-gradient noise*.

- **Sparse Convolution Noise**

  Sparse convolution noise is variant that uses a function with a random distribution of random pulses, applied in the grid of integers, resulting in a texture with less apparent regularity than the previous approaches.

Recently a extension to *Sparse convolution noise* was presented in (LAGAE et al., 2009). The presented function has very nice properties like good controllability, can be isotropic or anisotropic, can be used as 2D surface conforming noise or 3D solid noise,

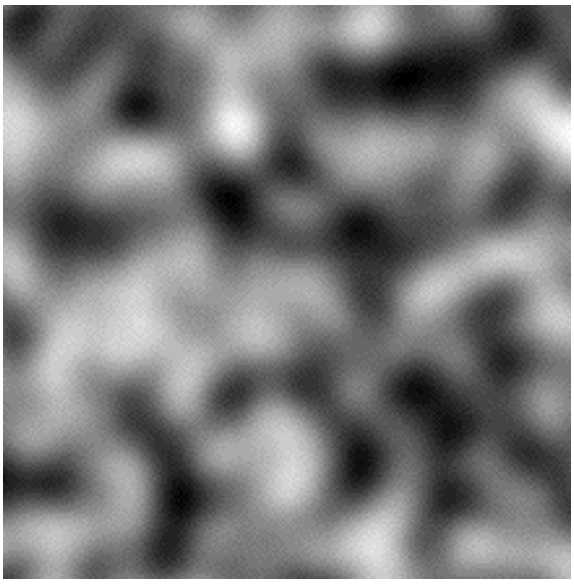


Figure 3.6: Image created by the *sparse convolution noise* function.

and do not use discretely sampled data like the integer grid, however, even being able to run at interactive rates the function is much more expensive in processing terms then the other noise functions, in such way that a texture with multiple noise evocations would be drastically slow down the rendering.

Figure 3.7: Different textures created with the noise solution from (LAGAE et al., 2009).

The noise function we used to the creation of the procedural textures of Chapter 5 was *simplex noise*, a extension of *Perlin noise* proposed in *Perlin2001*. The main difference is that *simplex noise* replaces the axis-aligned-grid for simplicial grid, this way instead of interpolating the values on the vertices of a quad, it is done with the vertices of a triangle (the 2D simplex), as for the 3D case, instead of interpolating the 8 vertices of a cube, it uses only 4 vertices of a tetrahedron (the 3D simplex). The result is function of faster evaluation and also the axis-aligned grid feature is replaced by a by a far less visually noticeable one. Further details on the implementation of *simplex noise* for real-time procedural texturing with shaders is given in Appendix 6.3.

There are other methods used to create *noise* with different properties, however the method presented are the most used ones and their implicit form is suitable for evaluation

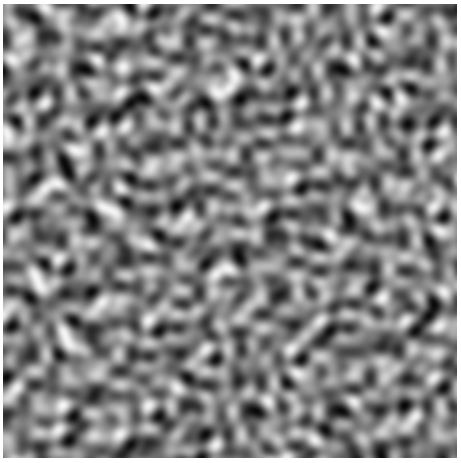


Figure 3.8: Image created by the *simplex noise* function.

in real-time, without the need to be generated previously and stored in a image to be queried during rendering stage.

Starting from these functions we can create more complex forms of *noise* by combining the results obtained from different methods, and with the same method but evaluated with different parameters, including spectral synthesis based on theses base functions. An example of such spectral synthesis of *noise* is the fractal sum where the amplitude is inversely proportional to the frequency.

```
float fractalsum(point Q) {
   float value = 0;
   for (f = MINFREQ; f < MAXFREQ; f *=2)
         value += noise(Q * f)/f;
   return value;
}
```

Another well known and used example is the *turbulence* function, that is similar to the fractal sum but only uses the absolute value of the noise function used.

```
float turbulence(point Q) {
   float value = 0;
   for (f = MINFREQ; f < MAXFREQ; f *=2)
         value += noise(Q * f)/f;
   return value;
}
```

The applications of *noise* in the creation of procedural textures are countless. A simple one is using the *noise* values to perturb the texture fetches, distorting the final result. Another very common application is the creation of perturbations in the regularity of the patterns. For example, in the brick texture shown before, an evocation of the *noise* function can be added to the code snippet that determinates the position of the fragment in the pattern, as shown in the following code snippet.

```
tbrick = floor(tt);
ss += 0.1 * noise(tbrick+0.5);
```

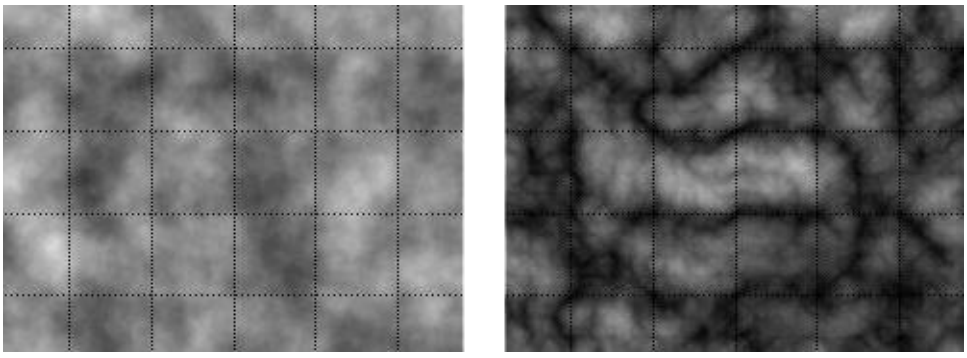


Figure 3.9: T
he *fractal sum of noises* (left) has a kind of cloud-like appearance. *Turbulence* (right) appearance is more well -defined, with more noticeable boundaries.

```
sbrick = floor(ss);
ss -= sbrick;
tt -= tbrick;
```

The values from the *noise* functions can be mapped straight to fragment color, represent perturbations in the surface with *bump mapping* or combine multiple scales to create a fractal version of *noise*. This simple nature and the wide applicability of the *noise* function makes of it what is called a texturing base function. For more information on noise we refer the interested reader to (LAGAE et al., 2010).

3.3 Cellular Texturing

In this section we present a new basis function for the creation of procedural textures known as *cellular texturing*. Although it leads to a quite different behavior and result from the ones of the *noise* functions, by being basis functions, they both share some properties. The purpose of *cellular texturing* is to distribute discrete elements randomly along the textured surface, usually subdividing this space into well limited regions giving the texture the appearance of sponge, scales, stones and similar materials.

Cellular texturing was first presented as a basis in by Steven Worley in (WORLEY, 1996), and is also known as *Worley noise*. The basic idea behind *cellular texturing* is quite simple: some points are randomly scattered in the texturing space, we refer to this point as *feature points*, then we want to create a function that relates a scalar value to a given fragment according to the distribution of *feature points* close to it. For a given point P (defined by the texture coordinates of a given fragment of the surface), we define *f1(P)* as the distance from P to the closest *feature point* to it. As the position of P changes, *f1(P)* changes in a continuous way even when the closest *feature point* to P changes. However, the derivatives of *f1(P)* will shown discontinuities where the closest *feature point* to P becomes another one. The points where discontinuities in the derivative appear will define the equidistant planes between the two closest *feature points* to P in 3D space. The set of such planes of each pair of *feature points* will create the Voronoi diagram of the set of *feature points*.

In the same way we can create a function *f2(P)* as the distance of P to the second *feature point* closest to it. The same properties of *f1(P)* will hold for *f2(P)* as well: it will be continuous along the space and its derivative will shown discontinuities were

the second *feature point* closest to *P* changes to be the one that was previously the first closest one or the third closest one. So we can follow the same idea and create *fn(P)* as the distance from *P* to it nth closest *feature point*.

These functions can be used straightly to map colors or normals. Depending on the function used the appearance changes. Using *f1(P)* for example results in a polka-dots look, circles, holes, round shapes, as for *f2(P)* and *f3(P)* the features are less smooth. The result can also be further improved by combining the different functions, and also by adding *noise* and using spectral synthesis like with fractal sum as well. *Worley noise* was used in some of the procedural texture shown the Results section of Chapter 5. Resources related to the implementation of real-time *Worley noise* functions used for the textures are presented in further detail on Appendix 6.3.

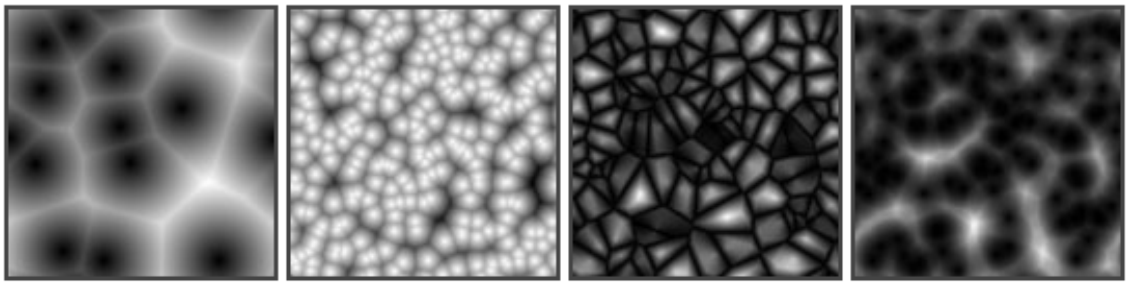


Figure 3.10: Images created by different cellular texturing functions.

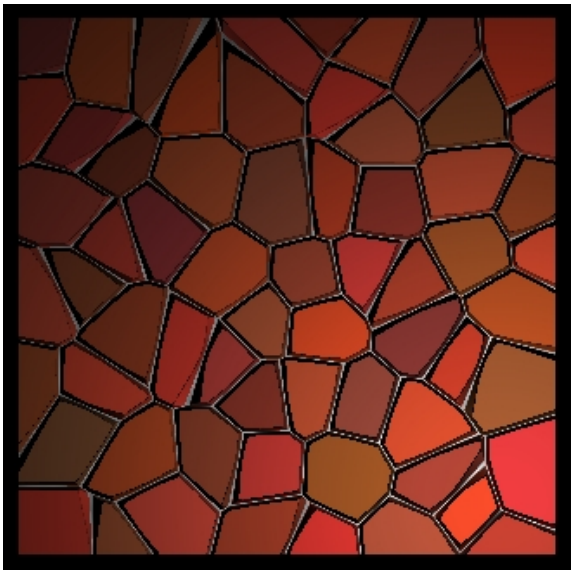


Figure 3.11: A stone-mosaic pattern created by using *cellular texturing* with *bump mapping*.

3.4 Solid Texturing

Most applications in computer graphics tend to used geometric models represented by their boundaries, the surface, the visible part only. So using textures defined only in two dimensions seems to be enough. However the 2D nature of such textures is usually unable to represent complex spatial characteristics of some materials found in the real

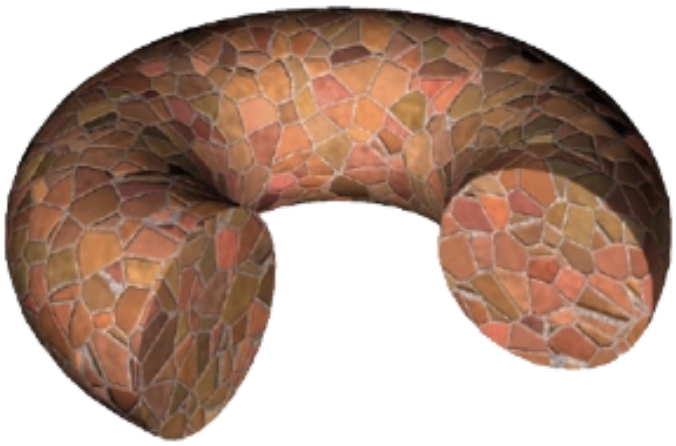


Figure 3.12: A torus with a stone-mosaic pattern created with *cellular texturing*.

world like gases, wood and marble. In these cases, a simple superficial representation of these materials may be not enough, for instance, a object carved out from a single wood trunk can be easily distinguished from a similar one that was assembled with pieces carved from different wood boards, because in the second one the texture along the visible surface will show noticeable discontinuities in the junctions of the different parts. Another good reason to using solid textures are the distortion created when mapping 2D image-based textures to surfaces of some higher complexity.

In our procedural textures context, the solid textures, or volumetric textures, become functions defined in the 3D space. Its value is then evaluated on each point of the object surface, but can also be evaluated anywhere in the 3D space being it inside or even outside the object. This way the texture starts to represent a block of some material from which the object is being carved from. Starting from the solid texture concept, Ken Perlin developed the concept of *hypertexture* (EBERT et al., 2002a). The idea is to represent the model shape itself as a function of in the 3D space as well, in such manner that every point located outside the object is associated to value 0 by the function, the inner points are given the value 1, and the points on the surface have values ranging between these 0 and 1 giving the surface a width. Then the function that defines the object is combined with the texturing function to create the final result, a textured shape.

There are two main methods to combine the functions of object shape and texture in *hypertextures*. The first one uses the texturing function to distort the space and then created the object with the object shape function taking the already distorted space as domain. The second method is limited to sum the texture function to the shape one and apply a sharpening filter in the result.

Despite the utility of solid textures, its use has been more popular with offline rendering of high quality scenes. This is due to the complexity of the algorithms to render this kind of texture. Only with the creation of the programmable graphic hardwares, solid textures started to be used in the real-time rendering of games and virtual environments and simulations that required real-time interactivity.

Figure 3.13: A vase textured with a solid texture that resembles marble.

3.5 Fractals

The task of reaching realism in the area of computer graphics is a matter of trying to reproduce the visual complexity of our world in the images synthesized. The fractal geometry represents a powerful way to create models with the complexity that we find in several natural phenomena. With the use of fractals we manage to replicated all this apparently chaotic nature to the deterministic one of mathematics. Also, fractals represent such complexity by means of simple functions that are fit to the capability of computers and that, through computer graphics, can be translated from mathematical abstractions to images, the most appropriated vehicle for human cognition.

But what is a fractal exactly? In (EBERT et al., 2002b), Musgrave defines a fractal as an object with a geometric complexity resulting from the repetition of a shape along a range of scale. From this definition, fractals can be seen as a new kind of symmetry called *dilatation symmetry* and that have *self-similarity*. Theses concepts relates to the fact that fractals will look the same independently of the scale, or zoom level that is used. Its shape is invariant between the different scales. This invariance is not necessarily exact in details, but can be the preservation of the overall appearance. For instance, a lesser portion of a cloud has still the same overall look of a greater portion or the whole cloud itself, even though the details are different, it is a case of *dilatation symmetry* and *self-similarity*. The same happens with branches of a tree, river networks, thunderbolts and other natural phenomena. All of them are examples of fractals.

The study of fractals introduced the new concept of *fractal dimension*. Different from the euclidian dimensions that me are used to work (zero dimensions corresponding to a point, one to a line, two to a plane and three to the space), the fractal dimension can be a real number like 2.3 for instance. The integer part of the fractal dimension indicates the base euclidian dimension of the fractal, in our example, 2.3, it would be a plane. The fractional part we call *fractal increment* and as it changes from .0 to .99..., the fractal modifies itself in such way that it begins to occupy only the base dimension and starts

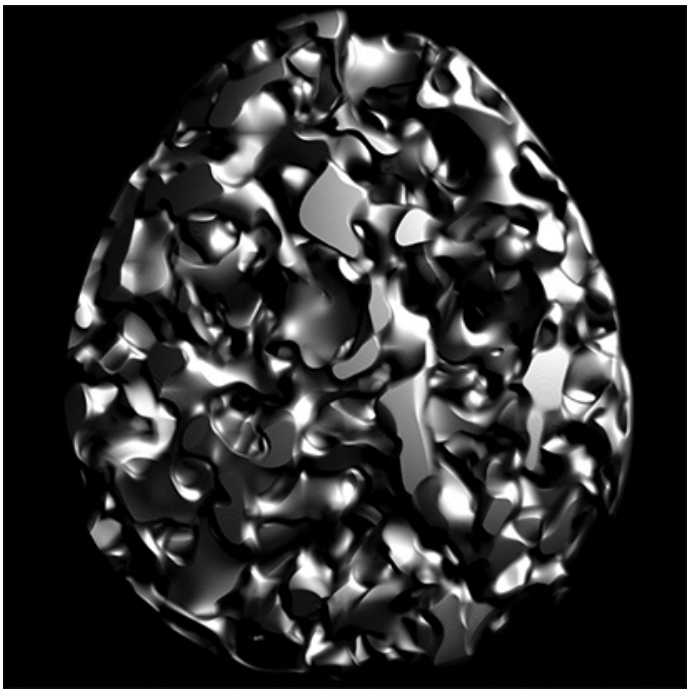


Figure 3.14: A volumetric texture limited by a function that defines the shape of an egg.

to move to the next one, in our example, it jumps from the plane to the space. The qualification *locally* is used because even though a sphere is a tridimensional object, as we zoom-in infinitely to its surface, it tends to become flat, locally. This way a shape do not need to be perfectly flat to have fractal dimension of 2.0 nor occupy the whole space to have fractal dimension of 3.0.

The cause of the complexity of the shape as well as the intermediate dimensionality is the repetition of the fundamental shape that defines the fractal in its different scales. Let us take a function that scatter bumps of a given size over a surface, a fractal can be created by the repetition of this function, but reducing the size and the height of the bumps at each iteration and stacking the result from the new iteration to the result with of the previous one. The size of the bumps is the frequency along the surface and the height is its amplitude. The variation of the frequency between each iteration is called the *lacunarity* of the fractal. A common value used for lacunarity is 2.0, and as with music, doubling the frequency means raising the tone by an octave, the term *octaves* is also used in the context of fractal geometry. The *noise* created by means of *fractal sum* in section 3.2 is a clear example of the use of fractals to obtain complex visual.

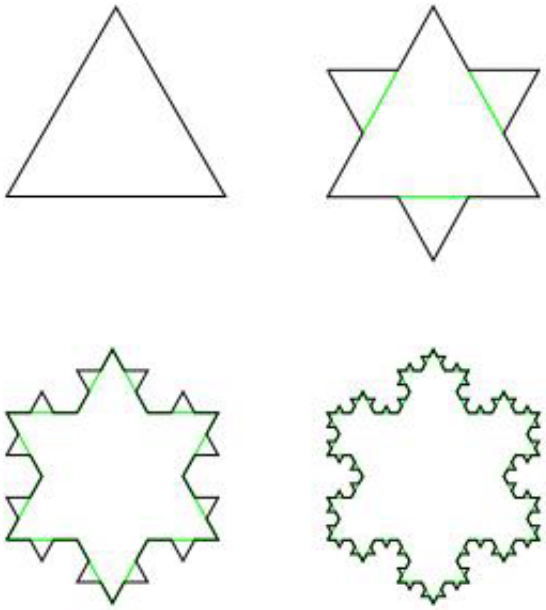


Figure 3.15: Four iterations of the von Koch snowflake. The fractal is created by replication of the fundamental form (in this case a triangle) on each edge of the shape resulting from the previous iteration.

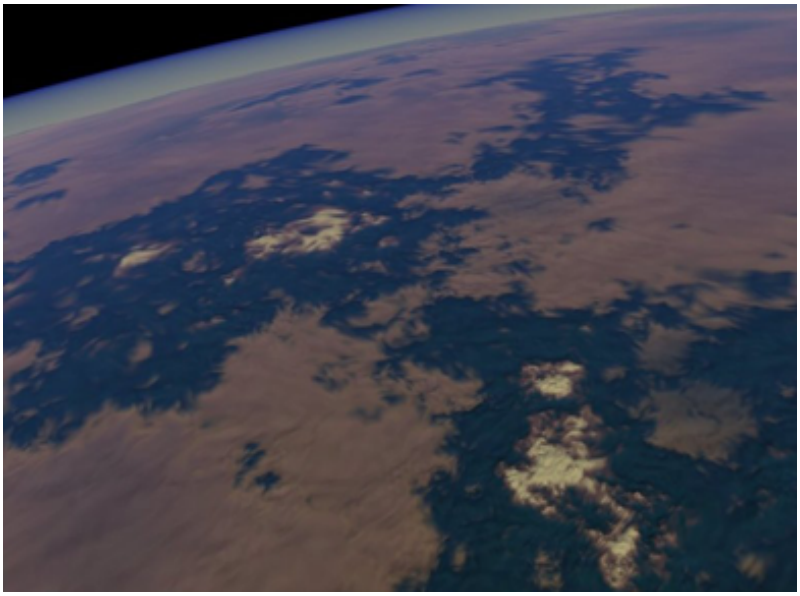


Figure 3.16: Texture created with fractals to simulate the landscape of a planet seen from the space.

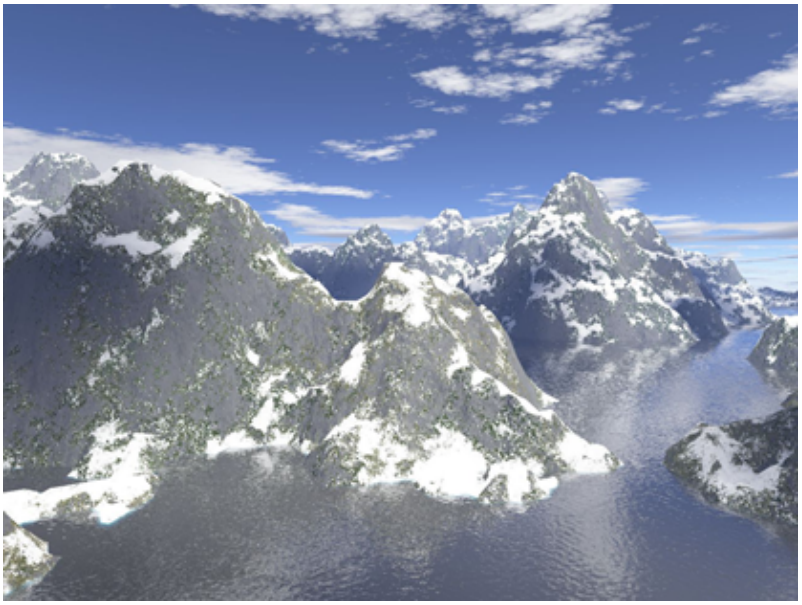


Figure 3.17: Fractals applied to the procedural creation of geometry and textures to model mountains.

4 PARAMETERIZATION

Parameterization is a very important, well-known and studied problem in computer graphics. It is a problem of great complexity and require knowledge of differential geometry concepts. A complete introduction to the parameterization area would represent a long deviation from the focus of this work, so in this chapter we present only some concepts on the subject so that the reader can understand our choices on parameterization for our *Geotextures* mapping technique, for procedural textures, presented in Chapter 5.

We saw on Chapter 2 that parameterization is used to create a uv map starting from a given 3D model being then very important to surface texturing. Parameterization is also an area of great importance in several fields of science and engineering, and still in computer graphics it is applied in the solution to a great variety of problems. Its introduction in the computer graphics area was in (BENNIS; VéZIEN; IGLéSIAS, 1991) and (MAILLOT; YAHIA; VERROUST, 1993) as a texture mapping scheme to improve the visual quality of polygonal models. Texture mapping was the major driving force for the evolution of parameterization methods in computer graphics until the quick development of 3D scanning technology that resulted in a great demand for efficient compression solutions for 3D meshes of ever increasing complexity. Now parameterization is common on several fields of computer graphics like detail mapping, detail synthesis, detail morphing, mesh completion, mesh editing, models databases, remeshing, mesh compression, surface fitting and others.

A parameterization is a transformation that takes a function f defined in a domain D to a function f^* defined in a domain D^*. In texture mapping we are concerned to parameterizations where f is a 3D mesh of triangles and f^* is a 2D mesh of triangles. So function is a mapping of the 3D mesh to the plane, such map is piecewise linear as it maps each triangle of the 3D mesh to a triangle in the planar mesh. The inverse of such map is known as parameterization. Also an usually important goal is to have a bijective function so there is a invertible map, and each point in the plane corresponds to only a single point in the 3D mesh, this way eliminating the overlapping of triangles in the *uv map*.

4.1 Map types

The different parameterization methods tries to create mappings of one of the three following categories:

- **Conformal mapping**: a mapping between two surfaces S and S^* is said to be conformal or angle-preserving when the angle defined by the intersection of every intersecting pair of arcs of S^* is the same of the intersection of the two corresponding arcs in S. An example is the projection of the Earth created by the cartographer

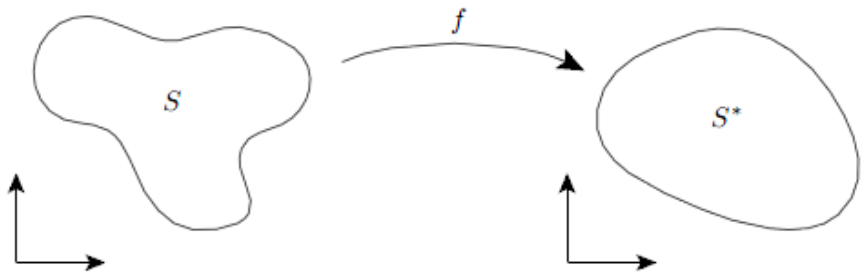


Figure 4.1: A simple diagram of the mapping of a 2D surface to another 2D surface.

Gerardus Mercator in 1569, shown in figure 4.2.

Figure 4.2: Mercator famous conformal projection of the Earth.

- **Equiareal mapping**: a mapping is said to be equiareal when every region of S^* has the same area of the corresponding region of S. The first equiareal projection was created by Johann Heinrich Lambert in 1772 by discarding the preservation of angles, shown in figure 4.3.

- **Isometric mapping**: a mapping from a surface S to another surface S^* is said to be isometric if the length of every arc of S^* is the same of the respective one in S. Such property is called *isometry*. Two given surfaces are isometric if there is *isometry* between them. The *isometry* exists if the surfaces have the same Gaussian curvature on each pair of corresponding points, for instance, the mapping from a cylinder to a plane by transforming cylindrical coordinates into Cartesian ones.

Isometric maps are proven to be both conformal and equiareal (KREYSZIG, 1991). So isometric mapping is the ideal type of mapping once that it preserves everything we

Figure 4.3: Lambert equiareal projection of the Earth.

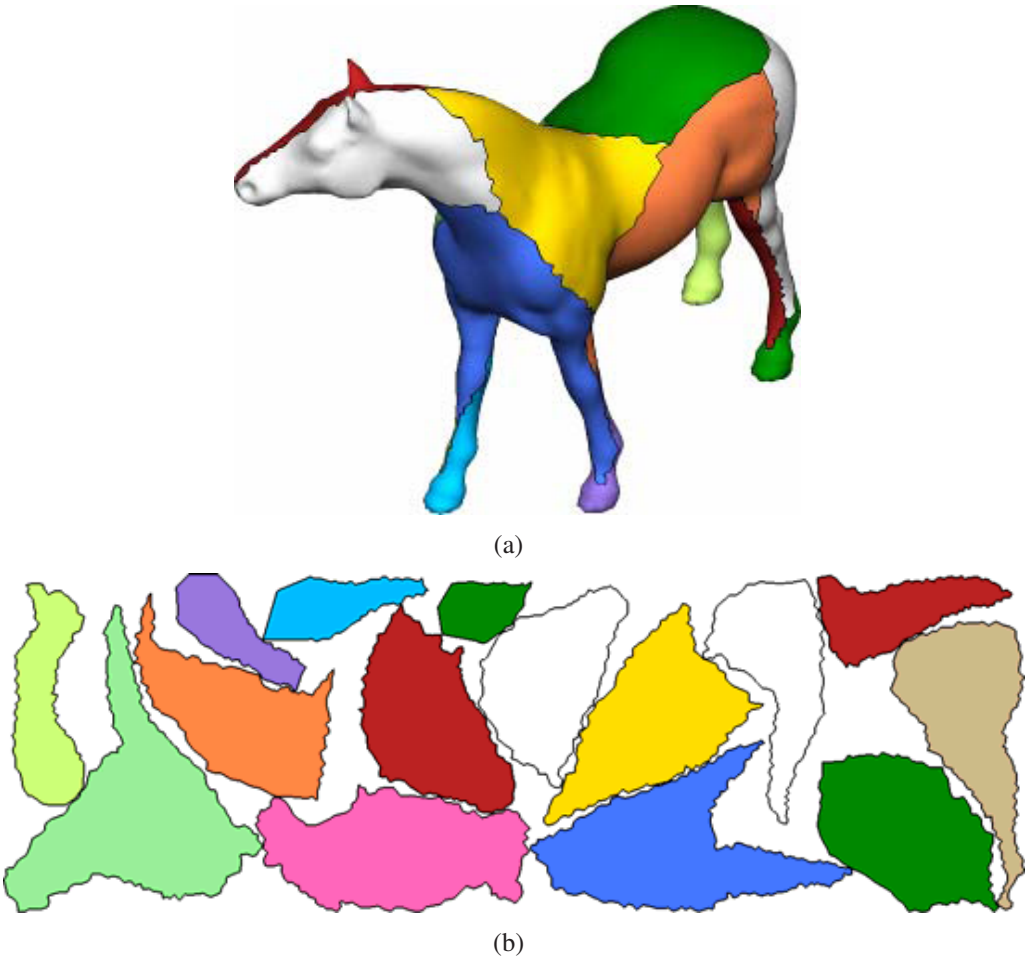
may need: angles, areas and lengths. The problem is that a isometric mapping is only possible in some rare cases, because the surface to be mapped to the plane must be developable, like the cylinder. Due to this limitation the methods of parameterization create maps that are conformal, i.e. preserve angles but has area distortion, equiareal, preserving areas but distorting angles, or tries to reduce some combination of angle and area distortion.

4.2 Mesh Cutting

The choice between preserving areas or angles is not the only problem. To be able map a 3D surface to the plane, the surface needs to have a boundary, in this sense the parameterization of closed 3D surfaces, like a sphere, represents another problem. The solution is to cut the surface open to give it a disk-like topology. Several works address the cutting of surfaces to give it disk-like topology, and they can be classified as segmentation techniques and seam generation techniques.

- **Mesh segmentation**: segmentation techniques use several different criteria to partition the original mesh in several disconnected parts, known as charts, creating an atlas of charts. Another good reason for these approach, besides the need of a disk-like topology, is that the parameterization of smaller and simpler charts leads to lesser distortion. Many methods are designed for mesh segmentation like (MAILLOT; YAHIA; VERROUST, 1993), (GARLAND; WILLMOTT; HECKBERT, 2001), (SANDER et al., 2001), (SANDER et al., 2003) and (COHEN-STEINER; ALLIEZ; DESBRUN, 2004a) to cite a few. These approached also motivated the methods of chart packing, that looks for solutions to put the generated charts together in the same parameter domain while trying to minimize the gaps between them for efficient storage (SANDER et al., 2003).

- **Seam generation**: seam generation techniques tries to give the mesh a disk-like topology by creating cuts but without create disconnected parts, the result is still a unique chart. They also use the creation of cuts to locally reduce the distortion in the parameterization. As example of solutions that follow this idea we can cite (SORKINE et al., 2002), (GU; GORTLER; HOPPE, 2002), (SHEFFER, 2002) and (SHEFFER; HART, 2002).



(a)

(b)

Figure 4.4: (a) Mesh segmented with the method in (SANDER et al., 2003). (b) Atlas of charts created and packed with the same method.

The concepts on parameterization presented are enough for the scope of this work. The amount of related work on parameterization is quite large, so for a more detailed survey on the subject we refer the interested reader to (SHEFFER; PRAUN; ROSE, 2006) and (HORMANN; LéVY; SHEFFER, 2007).

Figure 4.5: Seams created by the method of (SORKINE et al., 2002) to cut a closed mesh.

5 GEOTEXTURES

First we review some works that motivated the creation of our mapping scheme. In section 5.2 the method implemented for distance computation is presented, justifying relevant decisions and exposing others possibilities and tradeoffs, and our scheme for application of the geodesic distances on procedural texturing is then explained. The results are shown in section 5.3 with some procedural textures examples.

5.1 Related Work

The material of a graphical model is essential for the appearance of the model. In several cases, there is a close relationship between material and the model surface. In (WALTER; FOURNIER; MENEVAUX, 2001), Walter et al. modeled the creation of different patterns found on mammalian coats and its behavior as the body grows. Xu et al. (XU et al., 2009) applied salient feature curves to texture synthesis resulting in shape-revealing textures that stress the importance of shape characteristics. For dynamic texturing, the way features are propagated is also important, and distance is often used for this purpose. For example, in (YU et al., 2009) the distance field is used to define texture features that simulate the flow of water in a river. The Euclidian distance often suffices in $2D$ and $2\frac{1}{2}D$ applications, but for defining textures over 3D surfaces, inconsistencies might show up if the surface is not taken into account. Hegeman at al. (HEGEMAN et al., 2006) presented an approach for modeling relief conforming fluid flow over surfaces limited to genus zero. Stam (STAM, 2003) solved the fluid flow over surfaces of arbitrary topology, but restricted to Catmull-Clark subdivision surfaces (CATMULL; CLARK, 1998).

Surface parameterization (SHEFFER; PRAUN; ROSE, 2006; HORMANN; LéVY; SHEFFER, 2007) is a well-known problem strongly related to texture mapping. Different techniques focus on minimizing the parameterization angle and area distortion, but this process becomes harder when the complexity of the surface topology increases. The partitioning of the surface into charts allows to reduce distortion, but creates seams that may become visible when the parameterization is used for texture mapping. For example, in (ZHANG; MISCHAIKOW; TURK, 2005) surfaces composed of several genuses are partitioned into charts of disk-like topology. This allows each chart to be parameterized independently, but creates seams across adjacent neighboring charts. An approach to solve this problem using a continuity mapping is discussed in (GONZáLEZ; PATOW, 2009).

The discussion in the previous section (Figure 1.1) motivated the need for a global and consistent mapping defined over the surface of the model. Recent work (GU; YAU, 2003; KHODAKOVSKY; LITKE; SCHRöDER, 2003; BOMMES; ZIMMER; KOBBELT, 2009; RAY et al., 2006) address the creation of a global parameterization. However, they do not

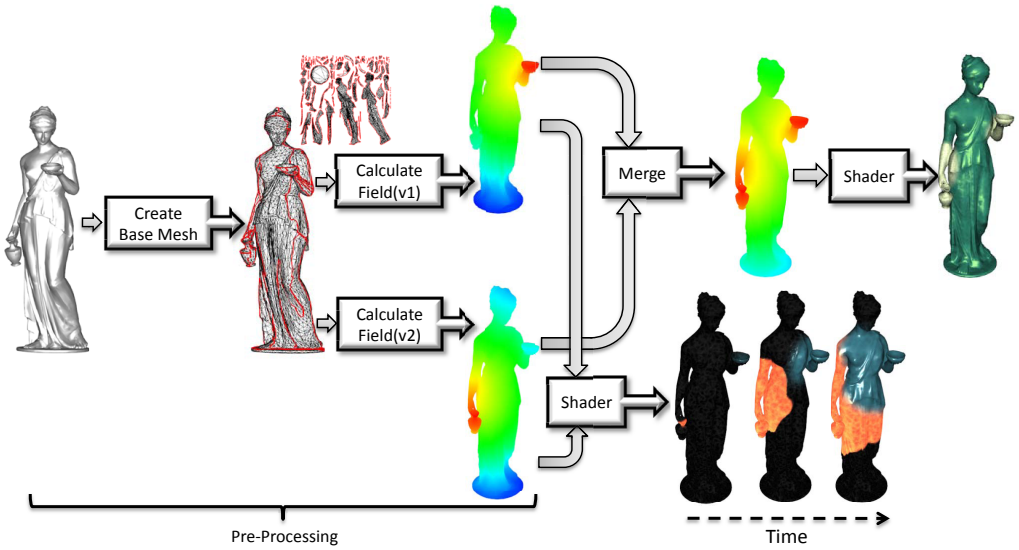
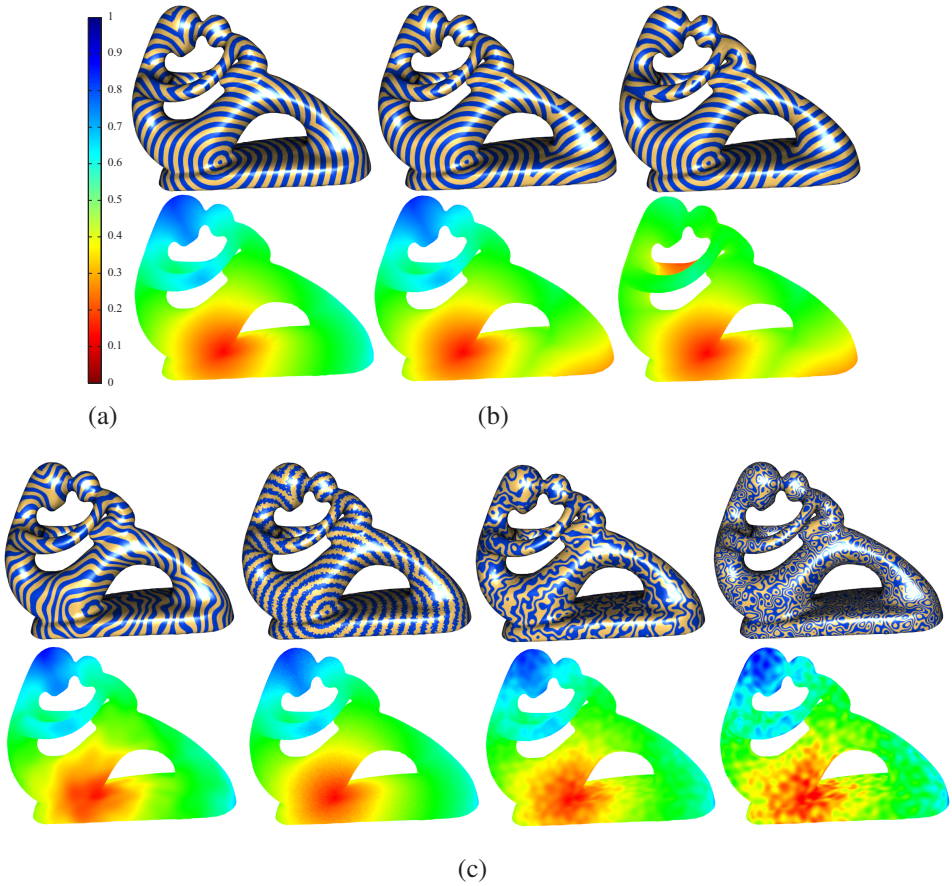


Figure 5.1: The input mesh is processed by the algorithm described in (TORCHELSEN et al., 2009) to partition into charts, which are assigned a 2D parameterization that guides a simplification of each chart. Distance field computation is performed over the base mesh. In this example, two distance fields are computed. For single material, both fields are combined before procedural texturing is applied (sand example). For multi-materials (lava and water) both distance fields are input to the rendering shader. Although only two distance fields are used in the diagram, the method imposes no restriction in the number of distance fields and materials used.

provide a global UV mapping in closed surfaces with holes that allows distances between points to be easily calculated. To overcome the need of such UV mapping, (YUKSEL; KEYSER; HOUSE, 2010) proposes the storage of colors for edges and faces of a surface mesh, while in (BURLEY; LACEWELL, 2008) textures are mapped to each individual face. Torchelsen et al. (TORCHELSEN et al., 2009) introduced an approximate but faster computation of geodesic distance fields over complex surfaces exploiting the simplification of charts and calculations on the parametric domain. Later, the same approach was successfully applied to calculate geodesic distance fields to guide the navigation of agents over complex surfaces (TORCHELSEN et al., 2010). Geodesic computation used in their work was the method of (SURAZHSKY et al., 2005), but any other method could have been used. Bommes and Kobbelt (BOMMES; KOBBELT, 2007) extended (SURAZHSKY et al., 2005) to the computation of geodesic distance fields taking polygonal curves as sources instead of only points. Weber et al. presented a parallel method in (WEBER et al., 2008) that also uses the parametric domain and reaches an even faster computation of geodesic distance fields, however it depends on a regular connectivity of the surface mesh and leads to less accurate results.

The proposal introduced in this work, Geotextures, relates to the above work as follows. We use the geodesic distance computation described in (TORCHELSEN et al., 2009) because it allows triangular meshes with irregular connectivity, instead of (WEBER et al., 2008) that would restrict us to regular connectivity meshes. Since we want to hadle complex meshes, we partition the mesh into charts using Cohen et al. (COHEN-STEINER; ALLIEZ; DESBRUN, 2004b), followed by the method of Wang (WANG, 2008) that reduces the number of charts by merging charts with less than a given threshold of distortion. This allows us to control the distortion by defining the amount of charts to



(a) (b)

(c)

Figure 5.2: (a) Distance field color scale. (b) Different geodesic distance fields color mapped from red to blue on a surface and isolines of the respective distance fields. (c) Distance field distortion with different noise scales.

be generated using the technique of (COHEN-STEINER; ALLIEZ; DESBRUN, 2004b). Therefore, we can trade accuracy for speedup since fewer charts result in faster geodesic field computation. The plane parameterization method was the one described in (SHEFFER et al., 2005) with an angle-based flattening approach. Since the final charts are quasi-developable surfaces, the distance variation inside each chart is constant in any direction and each chart can be re-triangulated to reduce the number of primitives, while preserving the borders and connectivity between adjacent charts. Geodesic computation uses the method of (SURAZHSKY et al., 2005) applied to the atlas of simplified charts, which allows faster wave front propagation.

5.2 Geotextures: Geodesic Texturing

The idea behind Geotextures is the construction of procedural textures over a surface according to a given distance field. The method is described by its architecture (Figure 5.1), with each step detailed in this section. We use the same approach proposed in (TORCHELSEN et al., 2009) that first creates a simplified mesh, which is used to compute the distance field. The main ideas behind Geotextures are related to how this information is used to create different variations of procedural textures.

5.2.1 Creation of the Base Mesh

The input to the system is given as a 3D model represented by a 2-manifold unstructured triangular mesh. To perform geodesic computations efficiently, we create a simplified version of the input mesh (called *base* mesh), which corresponds to the initial mesh broken into charts, each represented by fewer triangles.

This is accomplished in several steps. First, we partition the input mesh into charts using the methods of (COHEN-STEINER; ALLIEZ; DESBRUN, 2004b) and (WANG, 2008). Each chart is parameterized into quasi-developable regions using the ABF++ parameterization (SHEFFER et al., 2005). The triangles internal to each chart are discarded and its interior is re-triangulated, resulting in a simplified triangulation for each chart. The simplified charts representing the input mesh define what we call the base mesh, and serve as input to the procedural texturing stage.

5.2.2 Calculation of the Distance Field

The distance field computation stage receives as input a source vertex in the mesh (a *feature source*), and generates a geodesic distance field as output. The distance field is normalized into a 0-1 interval using the farthest point and serve as input to the procedural texturing stage. We computed separated distance fields, even though Surazhsky et al. (SURAZHSKY et al., 2005) algorithm allows to create a single distance field for more than one source point. We used separate fields since we can exploit CPU parallelism to compute multiple distance fields, while giving more flexibility to the procedural texture by providing independent distance fields. In Figure 5.1 we illustrate two distance fields calculated using different vertices (one vertex at the bowl on the left hand, and the other in the jar on the right hand) as feature sources. Distances are mapped to a red-to-blue rainbow colormap over the surface according to the scale in figure 5.2(a).

For multi-source computations, we also considered using the method of (TORCHELSEN et al., 2010), that creates an hierarchy of distance fields to allow distance field computation for several sources in real-time. Such solution could be easily adapted to procedural texturing, since the main idea is the use of a hierarchy of versions of the same mesh with different simplification levels. The difference in the accuracy of distance fields is not so noticeable in the simulation of agents navigation, however it would lead to perceptible changes in the texture appearance when the applied distance field is replaced by a more accurate version. Therefore, we chose to define source points and the respective distance fields in the preprocessing stage and do not use the hierarchy scheme.

5.2.3 Using Distance Fields in Procedural Texturing

Several distance fields with different feature sources can be used for procedural texturing with no extra cost in the evaluation of the functions used for the texture generation. They can either be used in separate to define features with different sources, or can be easily merged into a single distance fields (in preprocessing or real-time). In preprocessing, we can consider for each vertex only the minimum value from all distance fields provided. An alternative is to apply the same concept in real-time using either the vertex shader, or in the fragment shader (where distances values have already been interpolated for each fragment), which would result in a merge distance field with higher resolution.

Figure 5.2(b) shows isolines of different geodesic distance fields over the surface of the fertility model. In the leftmost column, we use a single source. In the middle and right columns, we combine the field in the leftmost column with an additional source (placed

at the bottom-right of the base and the elbow of the back, respectively). Isolines show that the distance field is defined in a consistent manner. One variation is to add noise to the distance field, which allows us to modify the propagation front to achieve more realistic effects. This is illustrated in Figure 5.2(c), with different noise scales and amplitudes.

5.2.4 Implementation using GLSL shaders

Procedural texturing is implemented in hardware using shaders. The input consists of the distance field, the distance threshold, and the material to be propagated. The code below shows how this mapping is implemented in a GLSL fragment shader:

```
float Threshold=smoothstep(distanceValue-thresholdWidth*0.5,
 distanceValue+thresholdWidth*0.5, distanceThreshold + thresholdWidth);
vec3 bump=mix(bumpMaterial1,bumpMaterial2,Threshold);
vec3 color=mix(colorMaterial1,colorMaterial2,Threshold);
```

In this shader, *distanceValue* is a fragment attribute defined by the interpolation of the geodesic distances values defined at the vertices of the face. The uniform variable *distanceThreshold* is a parameter to define the percentage of the surface covered by material, ranging from 0.0 to 1.0. The variables *distanceValue* and *distanceThreshold* are used to define a function *Threshold* that determines where one material ends and the other starts. This function is used to combine the materials property, such as the color and the bump function (that perturbs the fragment normal for shading calculations).

In order to map more materials, there must be defined more *Threshold* functions to be applied in a chain of nested combinations. This is illustrated in the code below:

```
float Threshold1=smoothstep(distanceValue1-thresholdWidth1*0.5,
 distanceValue1+thresholdWidth1*0.5,
 distanceThreshold1+thresholdWidth1);
float Threshold2=smoothstep(distanceValue2-thresholdWidth2*0.5,
 distanceValue2 + thresholdWidth2*0.5,
 distanceThreshold2 + thresholdWidth2);
vec3 bump=mix(mix(bump3,bump1,Threshold1), bump2, Threshold2);
vec3 color=mix(mix(color3,color1,Threshold1), color2, Threshold2);
```

Figure 5.3 illustrates the results using two materials. In this case each material has a different distance field and threshold. In Figure 5.4 we show the same distance field can used with different materials and distance threshold values. If we used the same threshold values, materials would overlap. A combined distance field with multiple sources can also be used (Figure 5.5), mapping different materials from multiple sources.

5.3 Results

All the preprocessing stage, from the creation of the base mesh to the computation of the distance fields, was implemented in C++, results rendered with OpenGL and procedural textures defined as GLSL shaders. For most examples we used a Intel Core2 CPU 6420 2.13GHz, and a NVIDIA GeForce 8600 GT. In the tesselation example we used a Fermi-based NVIDIA GeForce GTS 450. We show examples with up to three feature sources to provide better understanding of the method, but the techniques has no limitation on the number of feature sources to be used. Check the video provided as Online Resource 1 for a better visualization of the results presented.

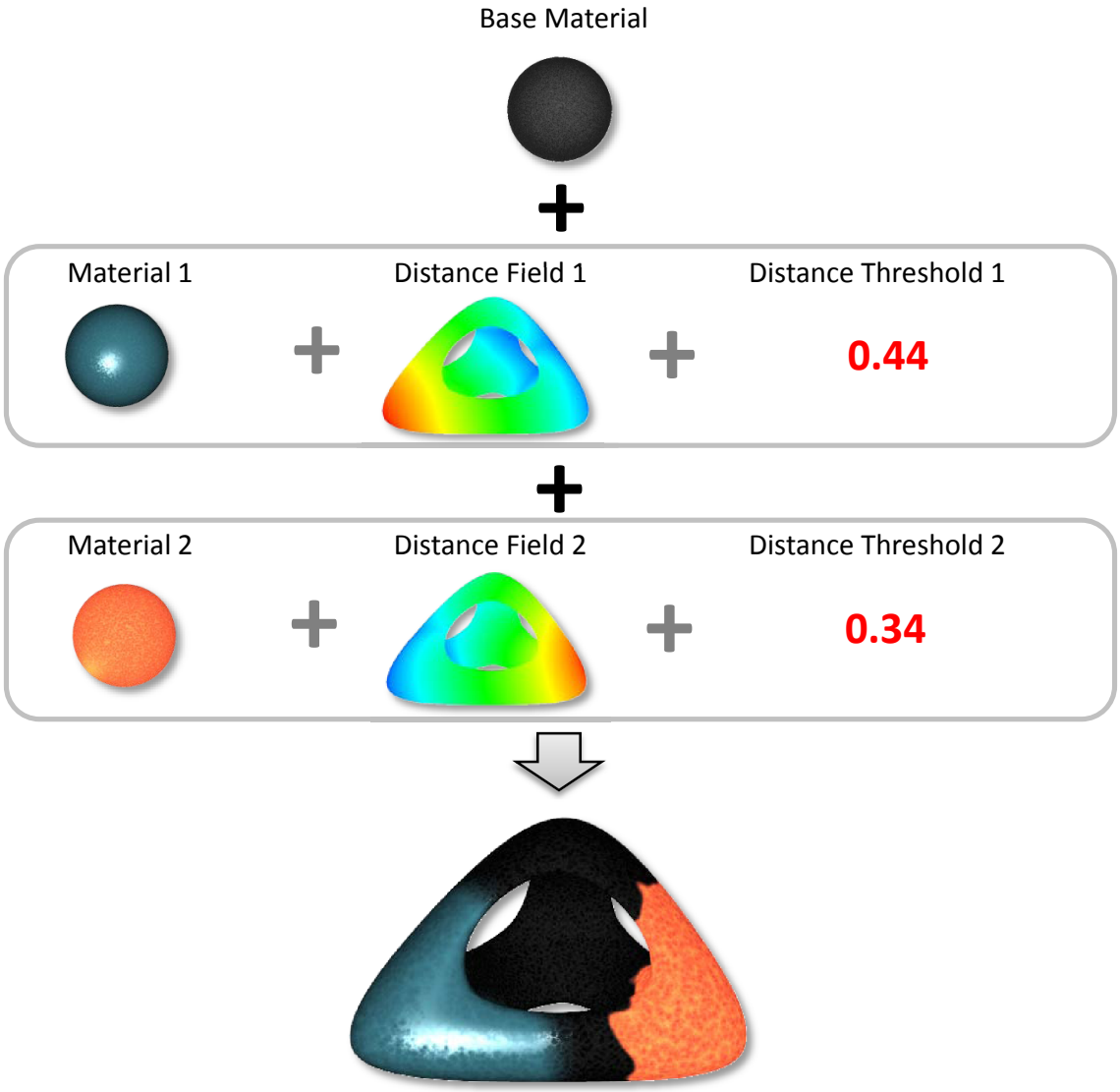


Figure 5.3: Different distance fields and distance threshold used to map different materials from distinct sources.

In Figure 5.6 we used the distance field to propagate a sand-like material over a model textured with a water-like green material. The threshold defines the transition between the two materials like an interpolation parameter for the color and bump function. In this example, the sand-like material is applied when the geodesic distance value is less than the provided threshold. The texture front advances uniformly from a single point in every direction, following the gradient of the distance field and along the surface shape. It shows no undesired behavior such as the ones formerly shown on figure 1.1. The sand-like appearance was achieved using a high frequency simplex noise as bump function, and the water-like one resulted from a fractal sum of Worley noise also as bump function. We refer the interested reader to (PERLIN, 2001) and (WORLEY, 1996) for more about simplex noise and Worley noise respectively and (EBERT et al., 2002a) for procedural texture synthesis.

Figure 5.7 uses the geodesic distance threshold to alter the surface opacity. Transparency is useful to explore complex surfaces as well as its interior. As shown in the

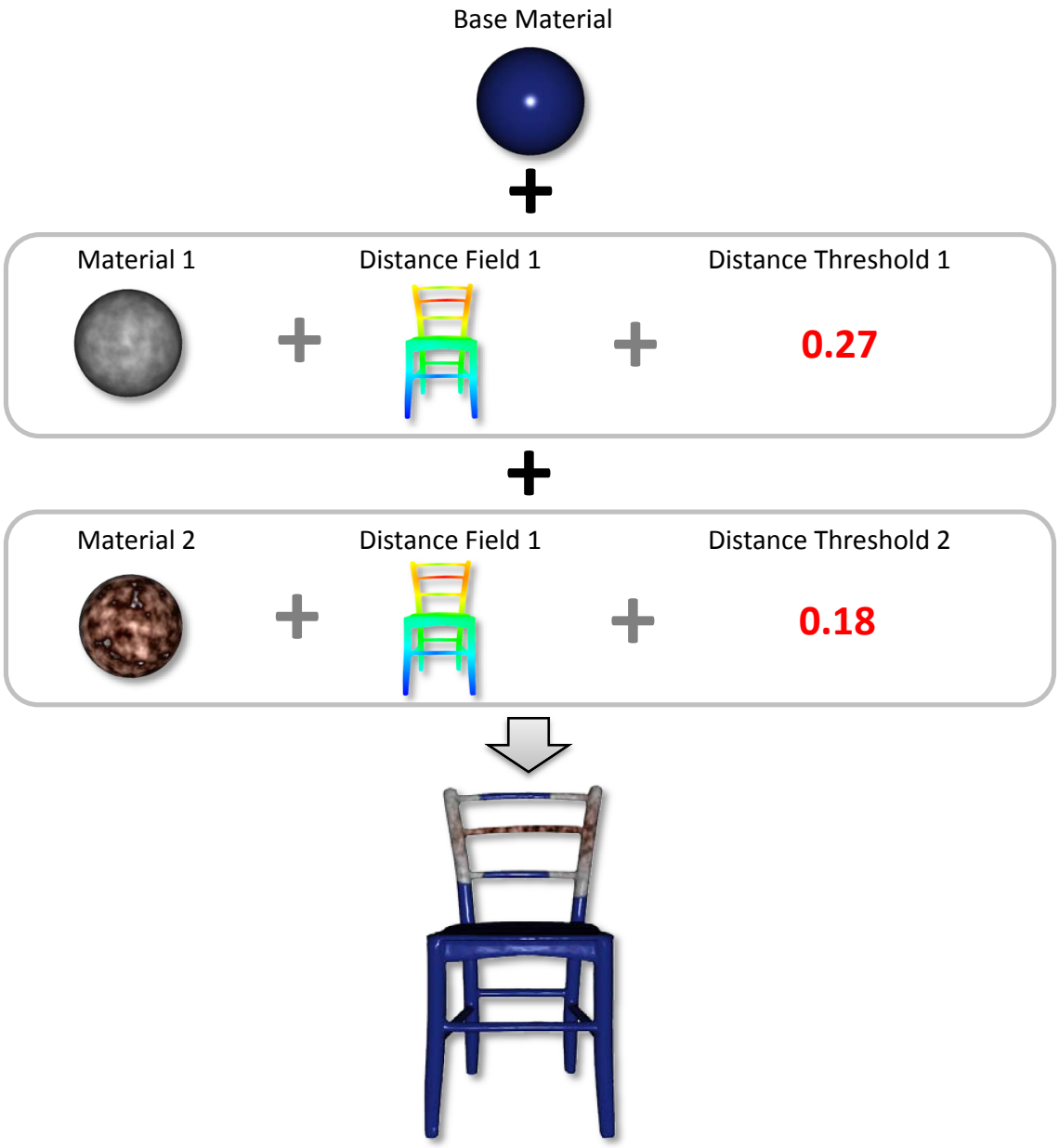


Figure 5.4: Distance field with different distance threshold values to map different materials with different propagation rates from the same source. Notice that each material propagate from every source in layers.

sequence, the continuous variation of the threshold reveals aspects of the complex surface with no change on the point of view.

In Figure 5.8 we simulated the dispersion of two fluids to exemplify independent propagation of materials with different properties from different sources. In this example, one distance field is required for each material. We used only a single source point for each material, but multiple sources can be defined as well. The water and lava appearance was created as described above. Noise was added to the threshold value associated to the lava material front to introduce a slight irregularity that simulates higher viscosity.

To propagate the same material from several sources at the same rate, a single distance field can be used. The spreading of moss in Figure 5.9(a) uses a single distance field with

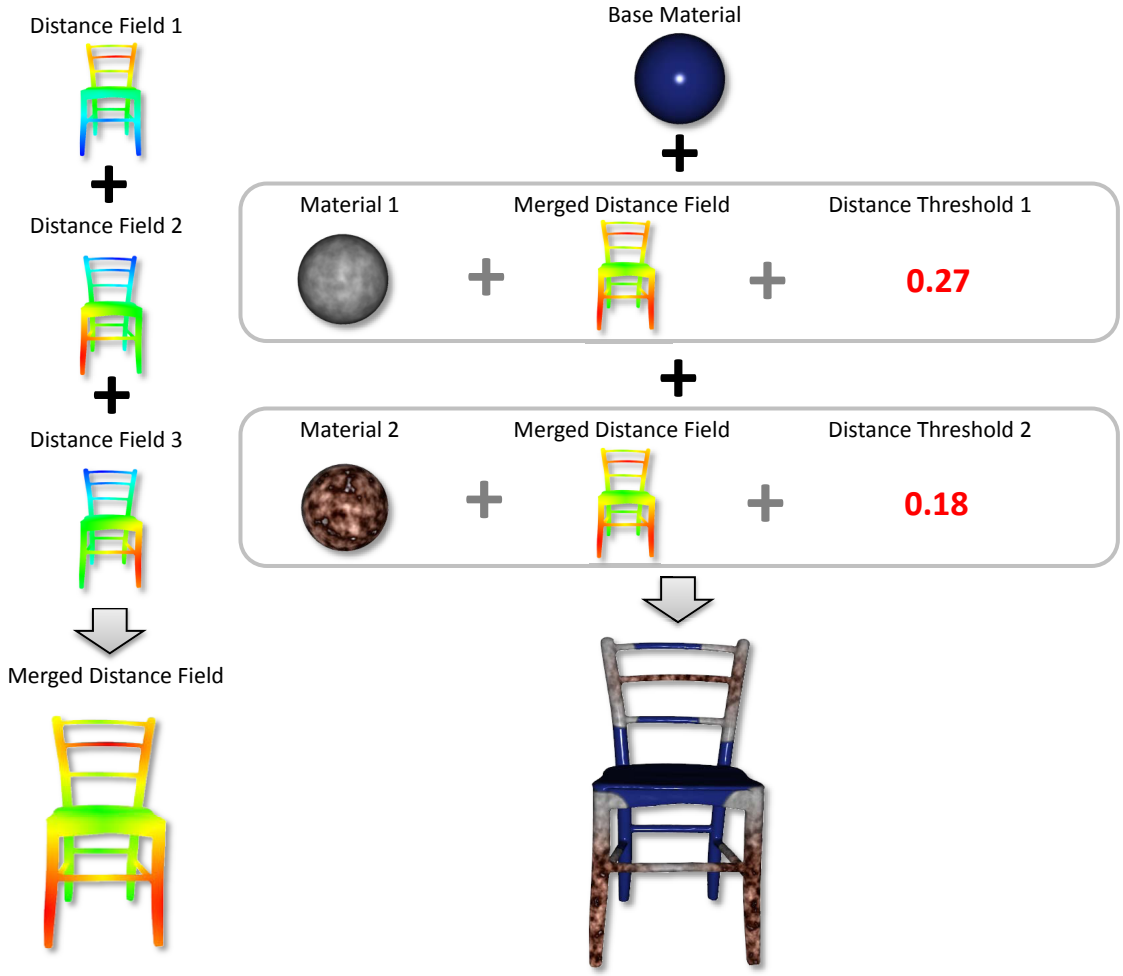


Figure 5.5: Variation of Figure 5.4 using a single distance field with multiple sources to propagate the same materials from multiple sources.

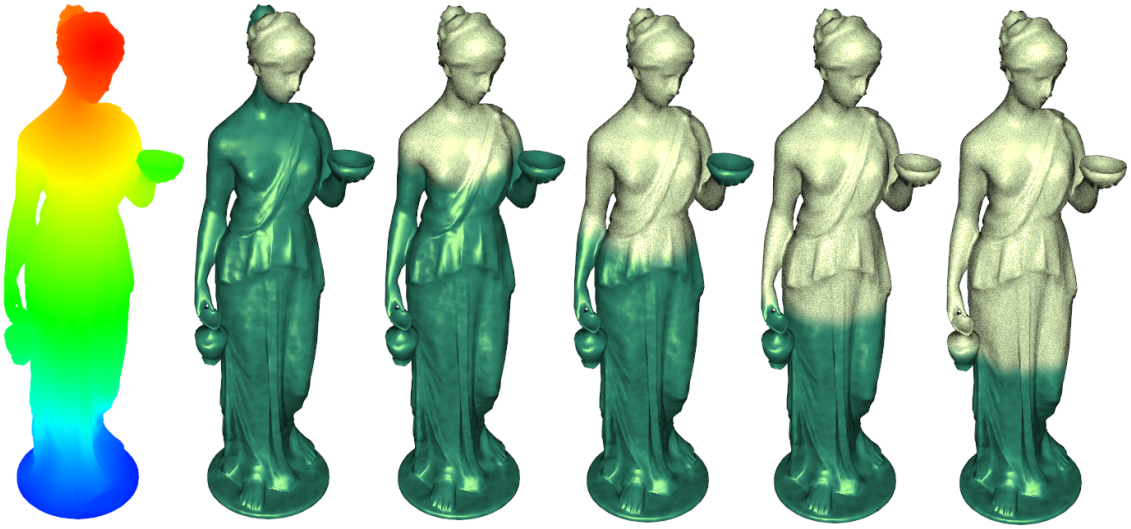


Figure 5.6: Geodesic distance field applied to define sand isle with the center on the model forehead. Notice the difference from the result in Figure 1.1.

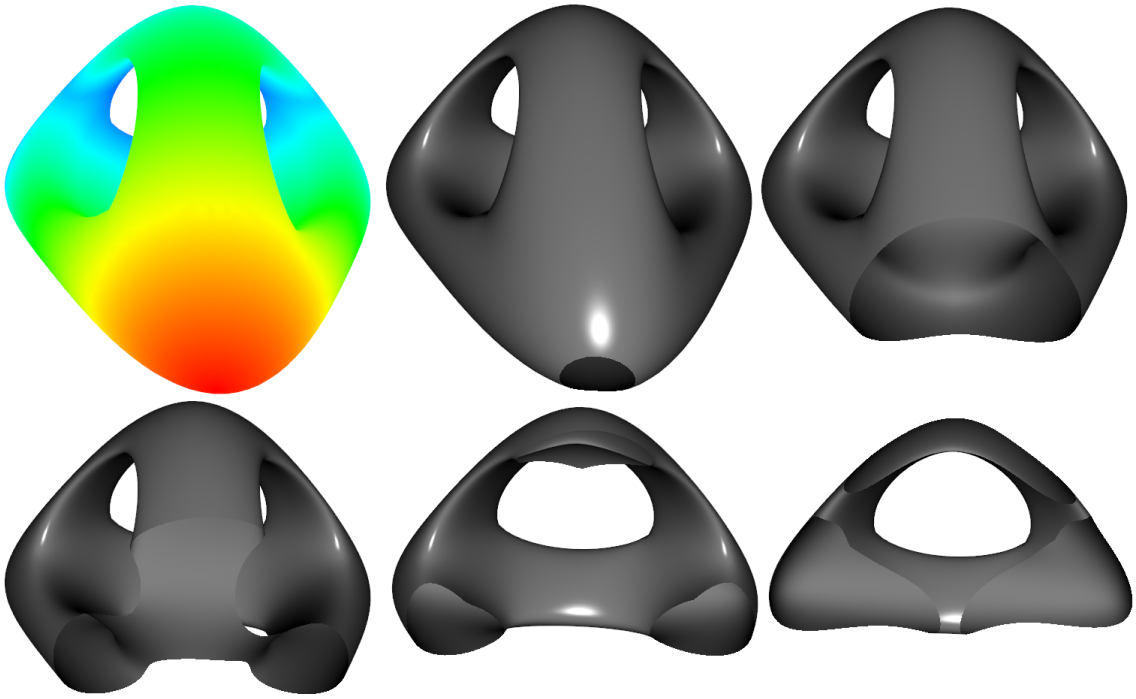


Figure 5.7: Application of geodesics to define the opacity along the mesh's surface.

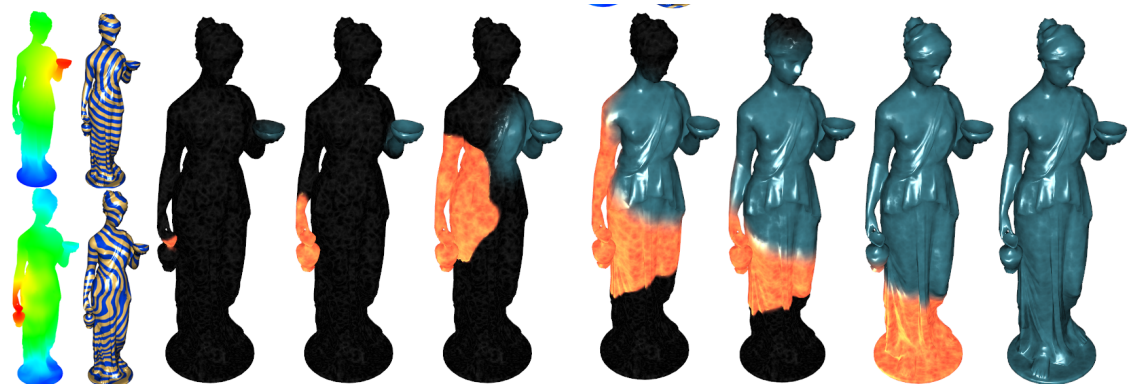
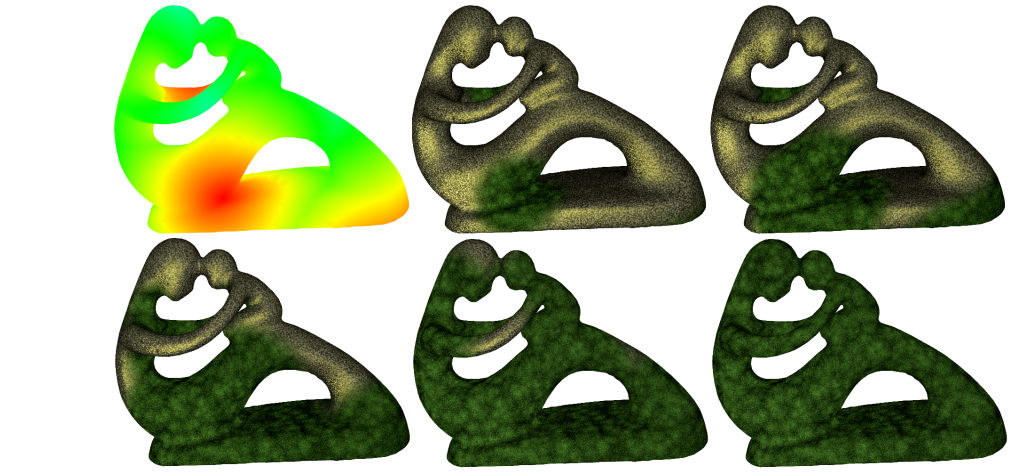


Figure 5.8: Propagation of different features from distinct sources. Noise is added to distance field associated to the lava material introducing irregularities in the propagation front. Water propagation has no noise addition and overlaps the lava flow. Both propagations are independent from each other and continue to evolve unaffected by the other front.

three distinct sources: two at the base of the sculpture and one at the elbow of the farther arm. A sum of simplex noise turbulence and fractal Worley noise was employed to model the bumps of the moss texture, while simplex noise turbulence defined the bumps for the grainy stone. We also used a single distance field to model rust evolution on a metal chair (Figure 5.9(b)). Three materials are defined according to the distance to three different points. In the chair model it is more evident how the propagation follows the surface due to several holes of the chair shape. The propagating front of each material follows strictly the silhouette of the model. The paint layer is just a blue color with some specular and a bump variation on the threshold with the metal material. The last one is the result of a fractal sum of simplex noise and high frequency simplex noise as bump with low specular



(a)

(b)

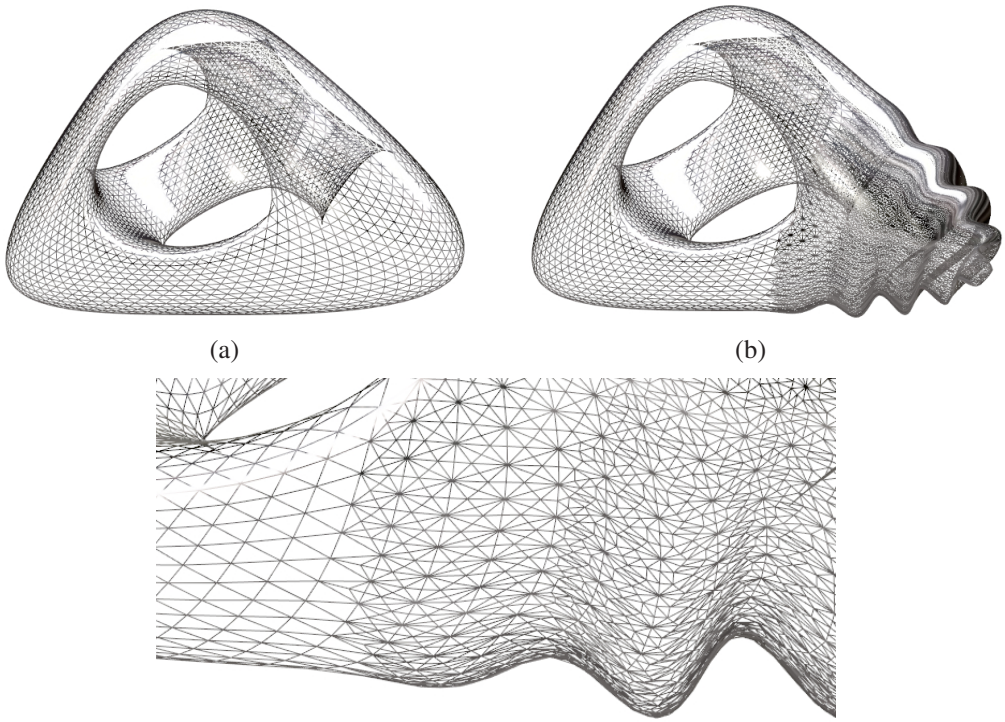
Figure 5.9: (a) Moss spreading from multiple sources over granite. (b) Different materials propagating from the same sources simulating rust evolution on a metal chair.

influence. The rusty appearance is also created with fractal sum, and a fourth threshold is defined, with some high frequency noise added to its threshold in order to change the opacity of the surface simulating the degeneration of the chair due to the rust action. By only adding noise to the threshold that defines the front of this last transparent material, we distort only this propagation front instead of modifying the whole field. Therefore, we can use the same field to propagate fronts with different perturbation levels.

As a last application we modeled the propagation of waves over a surface. A sine wave defined over the surface with frequency and amplitude varying accordingly to the distance to the source of the propagation. We implemented this example using a simple bump mapping technique, where only the surface normals are modified to give the illusion that the surface is deformed, and with displacement mapping computed by adaptive on-hardware tessellation, where the surface mesh is refined in real-time and the position of the vertexes are truly modified. Figure 5.10 compares the results of both implementations using a single propagation source. The clear advantage of displacement mapping is the deformation of the model silhouette, giving a more interesting appearance. The inherent problem is that the mesh must undergo a high tessellation to provide smooth silhouettes when displaced. The hardware tessellation allows us to refine the mesh in real-time accordingly to the level of displacement that will be applied in the region, and not refining the non-perturbed part of the mesh. Figure 5.11 compares the original tessellation of the model with the mesh produced by the hardware tessellator shader in our example. Figure 5.12 shows a similar comparison but with the propagation of one wave from each one of two different sources. We used a NVIDIA GeForce GTS 450 to run both wave propagation implementations. Bump mapping implementation ran on an average of 390

(a) (b) (c) (d)

Figure 5.10: (a) Distance field used for the wave propagation. (b) Small waves propagated from the source of the distance field. Simulated with displacement mapping [Top] and bump mapping [Bottom]. (c - d) Bigger waves simulated with displacement mapping [Top] and bump mapping [Bottom]. Notice the smooth silhouette of the waves in the displacement mapping.



(a) (b)

(c)

Figure 5.11: (a) Original tessellation of the surface. (b) Resulting mesh after hardware tessellation stage, where the mesh is refined adaptively according to the frequency of the perturbations added to the surface, which correspond to wave frequency in this example. (c) Detail of the adaptive tessellation. As the frequency of the waves increases the mesh is further refined.

FPS while the displacement mapping with adaptive on-hardware mesh tessellation ran on 360 FPS, however performance depends on the number of faces refined and the level of refinement used. In our tests, 0 to 2 levels sufficed for smooth displacements.

Table 5.1 lists the number of faces in each model, number of charts used and the av-

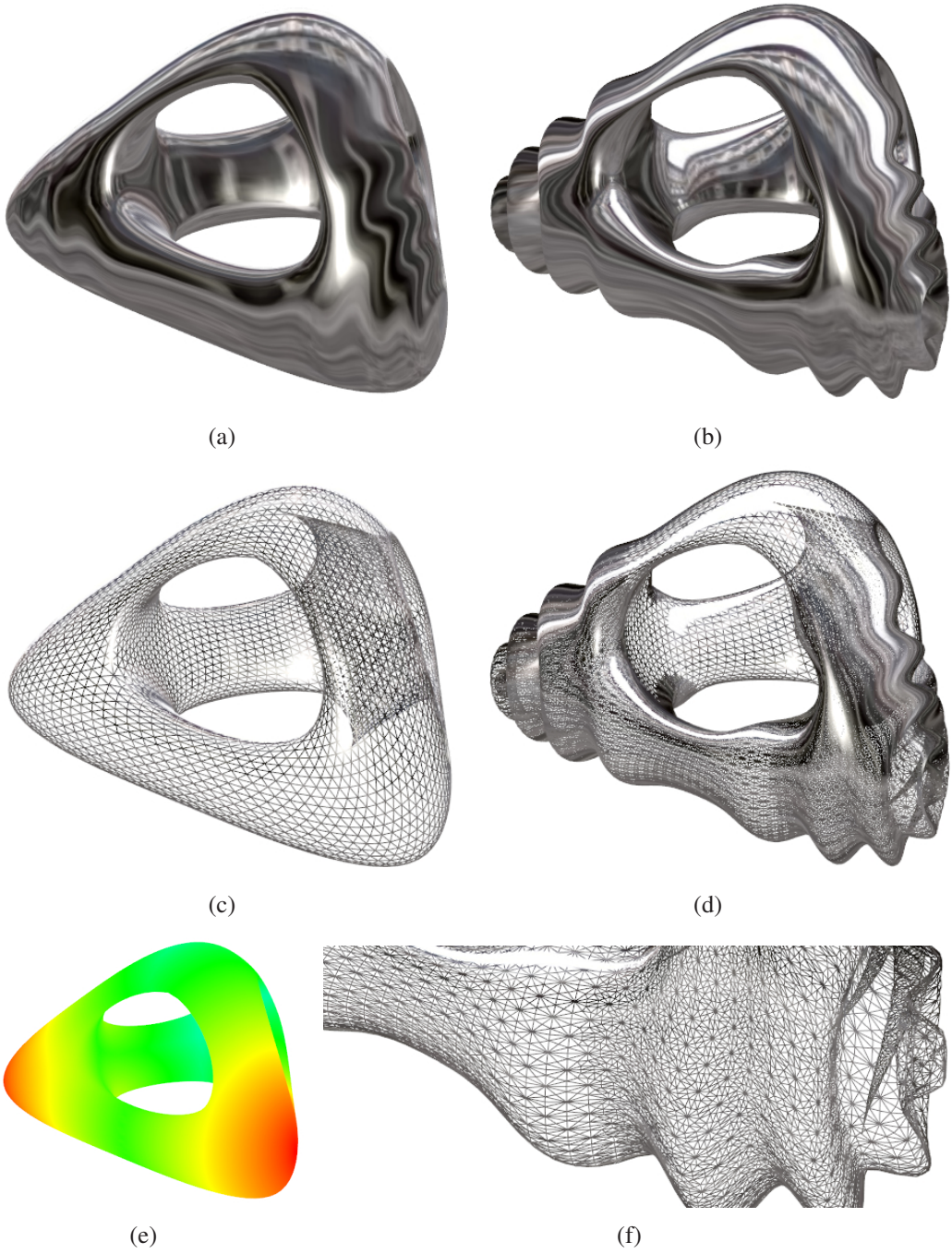


Figure 5.12: (a) Waves propagating from two different sources, simulated with bump mapping. (b) The same waves propagating from the same sources but with mesh displacement. (c) Original mesh, the same used in the bump mapping example. (d) Resulting mesh after the hardware tessellation stage in the displacement example. (e) Distance field with two sources, used to propagate the waves. (f) Detail of the adaptive tessellation.

erage time in seconds to calculate the geodesic distance field in the preprocessing stage, and its standard deviation. In Table 5.2 and 5.3 the frame rates for rendering the previous examples are given. The isolines frame rate was added for comparison purposes as an example of rendering time for a simple texture. Rendering is affected only by the complexity of the texture function and the number of rendered fragments, so for every texture example on table 5.2 and 5.3 the frame rate correspond are given when the model

Model	Triangles	Charts	Average(s)	Deviation
Fertility	10000	14	2,97	0,12
Genus3	13312	19	5,28	0,37
Chair	25524	127	9,36	0,27
Hebe	7989	101	1,78	0,15

Table 5.1: Model complexity and distance field calculation times

	Water/Sand		Moss		Rust	
Model	Near(fps)	Far(fps)	Near(fps)	Far(fps)	Near(fps)	Far(fps)
Fertility	9	75	7	75	9	97
Genus3	8	52	6	53	9	83
Chair	10	46	7	39	10	55
Hebe	15	77	13	78	18	117

Table 5.2: Rendering Times 1: average frame rates for the different models and texture examples.

	Water/Lava		Isolines	
Model	Near(fps)	Far(fps)	Near(fps)	Far(fps)
Fertility	3	48	71	232
Genus3	4	45	100	239
Chair	6	32	124	219
Hebe	10	64	148	247

Table 5.3: Rendering Times 2: average frame rates for the different models and texture examples.

is rendered close to the camera (Near), with more fragments, and farther (Far), with less fragments. The number of distance fields has no influence on the rendering performance.

6 CONCLUSIONS AND FUTURE WORK

6.1 Summary

In this work we present *Geotextures*, a geodesic-based mapping technique for procedural textures. The main contribution is an original technique that allows real-time procedural textures to be defined based on the distance to a set of source points. While conventional mapping approaches like 2D texture mapping and solid texturing shows unexpected results when using such distances with models of some complexity, the presented method leads to a coherent propagation of the textures along the surface relief even for surfaces of higher genus.

The technique is based on the use of geodesic distance fields as a parameter for the functions that define the procedural textures. First the original mesh is subdivided, parameterized and simplified to speed up the geodesic distance calculations. A distance calculation method is applied on the new mesh to calculate the distance from every vertex to a given source point in the surface. The resulting distance fields are then used as a parameter for the shaders that implement the procedural textures.

First we gave a brief introduction to the texturing of surfaces and some important concepts of procedural texture synthesis used to create some procedural textures to test the mapping scheme. Then a small study on parameterization was presented to highlight it importance on the mapping of textures and in our solution. The proposed mapping technique was explained and validated through the application to map several real-time animated procedural textures over different surfaces with holes, represented by meshes of triangles.

6.2 Conclusions

Our mapping technique proved to be fit for the mapping of real-time procedural textures over surfaces with no genus restriction, and represented by the most common form: meshes of triangles. The technique imposes no restriction to the number of textures and distance sources used. Also, its application for animation showed to be very straightforward, allowing the textures to propagate from the distance sources in a continuous and relief-conforming way, and can be used for other purposes like mesh deformation, exemplified with the waves displacement.

The main drawback of the technique lies in the computational cost for the creation of the base mesh for the geodesic distance field calculation, and also for the calculation of a new distance field when the base mesh is ready. The time spent to calculate a new distance field is long enough to create a noticeable stall between two frames, so we chose to calculate the distance fields in preprocessing stage. The same holds for the creation

of the base mesh, this way we cannot work with dynamic meshes in real-time rendering, as every time the mesh is modified, a new base mesh would have to be created and the distance fields recalculated based in this new base mesh.

6.3 Future Work

We believe this work has still room for several improvements, specially with regard to the geodesic distance field creation stage. An even faster computation of geodesic distances would allow the modification of feature sources in real-time, opening way for new applications. The use of the geodesic distance field can also be further explored to define other behaviors to materials, for example to induce flow direction.

Up to our knowledge it is an original manner of mapping procedural created features to simulate its propagation over surfaces. Though we show examples of real-time rendering, the technique can be used in the same way with offline renderers. In such case, the stall from the calculation of new distance fields will have low impact on the rendering time, since time spent to render each frame of a complex animated scene can take several minutes. Another advantage that offline rendering may offer is that by automating the whole scheme, including the creation of the base mesh for geodesic distance calculation, we can work with animated meshes by recalculating the distance fields periodically. It would lead to an increase in the time spent to render the frames where the distance is updated, but yet it may be an acceptable increase related to the overall time necessary for expensive features like ambient occlusion, transparency and smooth shadows.

REFERENCES

BENNIS, C.; VéZIEN, J.-M.; IGLéSIAS, G. Piecewise surface flattening for non-distorted texture mapping. In: COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 1991, New York, NY, USA. **...** ACM, 1991. (SIGGRAPH '91).

BOMMES, D.; KOBBELT, L. Accurate Computation of Geodesic Distance Fields for Polygonal Curves on Triangle Meshes. In: VISION MODELING AND VISUALIZATION, 2007. **...** [S.l.: s.n.], 2007.

BOMMES, D.; ZIMMER, H.; KOBBELT, L. Mixed-integer quadrangulation. In: ACM TRANSACTIONS ON GRAPHICS, 2009, New York, NY, USA. **...** ACM, 2009. v.28, n.3.

BROOKS, S.; DODGSON, N. A. Integrating procedural textures with replicated image editing. In: GRAPHITE '05: PROCEEDINGS OF THE 3RD INTERNATIONAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES IN AUSTRALASIA AND SOUTH EAST ASIA, 2005, New York, NY, USA. **...** ACM, 2005.

BROOKS, S.; DODGSON, N. Self-similarity based texture editing. In: SIGGRAPH '02: PROCEEDINGS OF THE 29TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 2002, New York, NY, USA. **...** ACM, 2002.

BURLEY, B.; LACEWELL, D. Ptex: per-face texture mapping for production rendering. In: EUROGRAPHICS SYMPOSIUM ON RENDERING 2008, 2008. **...** [S.l.: s.n.], 2008. p.1155–1164.

CATMULL, E.; CLARK, J. Recursively generated B-spline surfaces on arbitrary topological meshes. In: 1998, New York, NY, USA. **...** ACM, 1998.

COHEN-STEINER, D.; ALLIEZ, P.; DESBRUN, M. Variational shape approximation. In: ACM SIGGRAPH 2004 PAPERS, 2004, New York, NY, USA. **...** ACM, 2004. (SIGGRAPH '04).

COHEN-STEINER, D.; ALLIEZ, P.; DESBRUN, M. Variational shape approximation. In: SIGGRAPH '04: ACM SIGGRAPH 2004 PAPERS, 2004, New York, NY, USA. **...** ACM, 2004.

EBERT, D. S. et al. Texturing and Modeling: a procedural approach. In: ACM, 2002, San Francisco, CA, USA. **...** Morgan Kaufmann Publishers Inc., 2002.

EBERT, D. S. et al. Texturing and Modeling: a procedural approach. In: MORGAN KAUFMANN PUBLISHERS INC., 2002, San Francisco, CA, USA. **...** Morgan Kaufmann Publishers Inc., 2002.

FERNANDO, R.; KILGARD, M. J. The Cg Tutorial: the definitive guide to programmable real-time graphics. In: MORGAN KAUFMANN PUBLISHERS INC., 2003, Boston, MA, USA. **...** Addison-Wesley Longman Publishing Co.: Inc., 2003.

GARLAND, M.; WILLMOTT, A.; HECKBERT, P. S. Hierarchical face clustering on polygonal surfaces. In: SYMPOSIUM ON INTERACTIVE 3D GRAPHICS, 2001, New York, NY, USA. **...** ACM, 2001.

GONZáLEZ, F.; PATOW, G. Continuity mapping for multi-chart textures. In: ACM TRANSACTIONS ON GRAPHICS, 2009, New York, NY, USA. **...** ACM, 2009. v.28, n.5.

GU, X.; GORTLER, S. J.; HOPPE, H. Geometry images. In: COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 2002, New York, NY, USA. **...** ACM, 2002. (SIGGRAPH '02).

GU, X.; YAU, S.-T. Global conformal surface parameterization. In: SGP '03: PROCEEDINGS OF THE 2003 EUROGRAPHICS/ACM SIGGRAPH SYMPOSIUM ON GEOMETRY PROCESSING, 2003, Aire-la-Ville, Switzerland, Switzerland. **...** Eurographics Association, 2003.

HEGEMAN, K. et al. GPU-based Conformal Flow on Surfaces. In: EUROGRAPHICS ASSOCIATION, 2006. **...** [S.l.: s.n.], 2006.

HEWGILL, A.; ROSS, B. J. Procedural 3D Texture Synthesis Using Genetic Programming. In: COMPUTERS AND GRAPHICS, 2003. **...** [S.l.: s.n.], 2003. v.28.

HORMANN, K.; LéVY, B.; SHEFFER, A. Siggraph Course Notes Mesh Parameterization: theory and practice. In: 2007. **...** [S.l.: s.n.], 2007.

KHODAKOVSKY, A.; LITKE, N.; SCHRöDER, P. Globally smooth parameterizations with low distortion. In: SIGGRAPH '03: ACM SIGGRAPH 2003 PAPERS, 2003, New York, NY, USA. **...** ACM, 2003.

KREYSZIG, E. Differential Geometry. Dover, New York, 1991. In: ACM, 1991, New York. **...** Dover, 1991.

LAGAE, A. et al. Procedural noise using sparse Gabor convolution. In: ACM SIGGRAPH 2009 PAPERS, 2009, New York, NY, USA. **...** ACM, 2009. (SIGGRAPH '09).

LAGAE, A. et al. State of the Art in Procedural Noise Functions. In: EG 2010 - STATE OF THE ART REPORTS, 2010. **...** [S.l.: s.n.], 2010.

LEFEBVRE, S.; NEYRET, F. Pattern based procedural textures. In: I3D '03: PROCEEDINGS OF THE 2003 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS, 2003, New York, NY, USA. **...** ACM, 2003.

MAILLOT, J.; YAHIA, H.; VERROUST, A. Interactive texture mapping. In: COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 20., 1993, New York, NY, USA. **Proceedings...** ACM, 1993. (SIGGRAPH '93).

MATUSIK, W.; ZWICKER, M.; DURAND, F. Texture design using a simplicial complex of morphable textures. In: ACM TRANSACTIONS ON GRAPHICS, 2005, New York, NY, USA. ... ACM, 2005. v.24, n.3.

PEACHEY, D. R. Solid texturing of complex surfaces. In: SIGGRAPH '85: PROCEEDINGS OF THE 12TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 1985, New York, NY, USA. ... ACM, 1985.

PERLIN, K. An image synthesizer. In: COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 1985, New York, NY, USA. ... ACM, 1985. (SIGGRAPH '85).

PERLIN, K. Noise Hardware. In: ACM, 2001. ... [S.l.: s.n.], 2001.

RAY, N. et al. Periodic global parameterization. In: ACM TRANSACTIONS ON GRAPHICS, 2006, New York, NY, USA. ... ACM, 2006. v.25, n.4.

ROST, R. J. et al. OpenGL Shading Language. In: ACM, 2009. ... Addison-Wesley Professional, 2009.

SANDER, P. V. et al. Texture mapping progressive meshes. In: COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 2001, New York, NY, USA. ... ACM, 2001. (SIGGRAPH '01).

SANDER, P. V. et al. Multi-chart geometry images. In: EUROGRAPHICS/ACM SIGGRAPH SYMPOSIUM ON GEOMETRY PROCESSING, 2003, Aire-la-Ville, Switzerland, Switzerland. ... Eurographics Association, 2003.

SHEFFER, A. Spanning Tree Seams for Reducing Parameterization Distortion of Triangulated Surfaces. In: SHAPE MODELING INTERNATIONAL, 2002, Washington, DC, USA. ... IEEE Computer Society, 2002.

SHEFFER, A. et al. ABF++: fast and robust angle based flattening. In: ACM TRANSACTIONS ON GRAPHICS, 2005, New York, NY, USA. ... ACM, 2005. v.24, n.2.

SHEFFER, A.; HART, J. C. Seamster: inconspicuous low-distortion texture seam layout. In: VISUALIZATION, 2002, Washington, DC, USA. ... IEEE Computer Society, 2002.

SHEFFER, A.; PRAUN, E.; ROSE, K. Mesh parameterization methods and their applications. In: FOUNDATIONS AND TRENDS IN COMPUTER GRAPHICS AND VISION, 2006, Hanover, MA, USA. ... Now Publishers Inc., 2006. v.2, n.2.

SORKINE, O. et al. Bounded-distortion piecewise mesh parameterization. In: VISUALIZATION, 2002, Washington, DC, USA. ... IEEE Computer Society, 2002.

STAM, J. Flows on surfaces of arbitrary topology. In: SIGGRAPH '03: ACM SIGGRAPH 2003 PAPERS, 2003, New York, NY, USA. ... ACM, 2003.

SURAZHSKY, V. et al. Fast exact and approximate geodesics on meshes. In: SIGGRAPH '05: ACM SIGGRAPH 2005 PAPERS, 2005, New York, NY, USA. ... ACM, 2005.

TAKAYAMA, K. et al. Lapped solid textures: filling a model with anisotropic textures. In: ACM TRANSACTIONS ON GRAPHICS, 2008, New York, NY, USA. ... ACM, 2008. v.27.

TEN24. Making of Tron Evolution. In: ACM, 2011. ... [S.l.: s.n.], 2011. Visited on: Feb. 06, 2011.

TORCHELSEN, R. P. et al. Approximate on-Surface Distance Computation using Quasi-Developable Charts. In: COMPUT. GRAPH. FORUM, 2009. ... [S.l.: s.n.], 2009. v.28, n.7.

TORCHELSEN, R. P. et al. Real-time multi-agent path planning on arbitrary surfaces. In: I3D '10: PROCEEDINGS OF THE 2010 ACM SIGGRAPH SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES, 2010, New York, NY, USA. ... ACM, 2010.

TURK, G. Texture synthesis on surfaces. In: SIGGRAPH '01: PROCEEDINGS OF THE 28TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 2001, New York, NY, USA. ... ACM, 2001.

WALTER, M.; FOURNIER, A.; MENEVAUX, D. Integrating shape and pattern in mammalian models. In: SIGGRAPH '01: PROCEEDINGS OF THE 28TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 2001, New York, NY, USA. ... ACM, 2001.

WANG, C. Computing Length-Preserved Free Boundary for Quasi-Developable Mesh Segmentation. In: IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS, 2008, Piscataway, NJ, USA. ... IEEE Educational Activities Department, 2008. v.14, n.1.

WEBER, O. et al. Parallel algorithms for approximation of distance maps on parametric surfaces. In: ACM TRANS. GRAPH., 2008, New York, NY, USA. ... ACM, 2008. v.27, n.4.

WIENS, A. L.; ROSS, B. J. Gentropy: evolving 2d textures. In: ACM, 2002. ... [S.l.: s.n.], 2002.

WILLIAMS, L. Pyramidal parametrics. In: SIGGRAPH COMPUT. GRAPH., 1983, New York, NY, USA. ... ACM, 1983. v.17.

WORLEY, S. A cellular texture basis function. In: SIGGRAPH '96: PROCEEDINGS OF THE 23RD ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 1996, New York, NY, USA. ... ACM, 1996.

XU, K. et al. Feature-Aligned Shape Texturing. In: ACM TRANSACTIONS ON GRAPHICS, 2009. ... [S.l.: s.n.], 2009. v.28, n.5.

YU, Q. et al. Scalable Real-Time Animation of Rivers. In: COMPUTER GRAPHICS FORUM (PROCEEDINGS OF EUROGRAPHICS 2009), 2009. ... [S.l.: s.n.], 2009. v.28, n.2.

YUKSEL, C.; KEYSER, J.; HOUSE, D. H. Mesh colors. In: ACM TRANSACTIONS ON GRAPHICS, 2010, New York, NY, USA. ... ACM, 2010. v.29, n.2.

ZHANG, E.; MISCHAIKOW, K.; TURK, G. Feature-based surface parameterization and texture mapping. In: ACM TRANSACTIONS ON GRAPHICS, 2005, New York, NY, USA. ... ACM, 2005. v.24, n.1.

APPENDIX A SIMPLEX NOISE IMPLEMENTATION RESOURCES

We used the implementation of *simplex noise* provided by Stefan Gustavson. The code is provided by the author himself at *http://staffwww.itn.liu.se/ stegu/simplexnoise/MacOSX-port/*. Evocation of *simplex noise* in the procedural texture shaders of Appendix C will appear in the form of his *snoise()* function. His implementation uses two textures to store predefined values for gradient in the vertices of the simplices one of which is only used in the evaluation of the 4D noise function.

Below are the used implementations of *fractal sum* and *turbulence* with *simplex noise*.

```
float fractalsum(vec3 pos, float minFreq, float maxFreq, float multiFreq)
{
   float value = 0.0;
   float f;
   float ns;

   for (f=minFreq;f<maxFreq;f*=multiFreq)
   {
      value += snoise(vec3(pos * f)) / f;
   }

   return value;
}

float turbulence(vec3 pos, float minFreq, float maxFreq, float multiFreq)
{
   float value = 0.0;
   float f;
   float ns;

   for (f=minFreq;f<maxFreq;f*=multiFreq)
   {
      value += abs(snoise(vec3(pos * f))) / f;
   }

   return value;
}
```

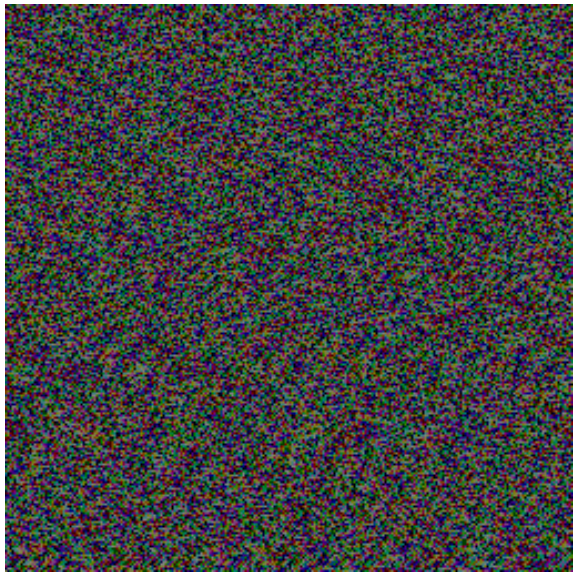


Figure A.1: Texture for gradient lookup in the *simplex noise* implementation.

APPENDIX B WORLEY NOISE IMPLEMENTATION RESOURCES

Our implementation of *Worley noise* is a simplified version that is fast to evaluate so we can use *fractal sum* of *Worley noise* in the procedural texture and still be able to render the scene at interactive frame rates. It is quite similar to the noise approaches, we define a grid and take each vertex of this grid as a *feature point* of the cellular texture. Such scheme would result in a very regular look, so we use noise to perturb the position of each vertex of the grid.

This method allow us to limit the calculation of distances only to the *feature points* of the grid cells of the immediate neighborhood of the evaluated fragment. A more complete implementation of cellular texturing would require to calculate the distance of the fragment to every *feature point* defined making the function too slow for real-time evaluation. However, the price is a limitation on how much the *feature point* position is perturbed. If we allow the *feature point* to be moved too far from its original position in the grid we cannot ensure that the closest *feature point* to a given fragment is one of those of the immediate neighbor cells, as now one from a faraway cell may be the closest one. We control this perturbation intensity by the value of the *noiseScale*variable in the *gpuCellNoise3D* function.

Our *Worley noise* is defined in the *gpuCellNoise3D*. Given a position, it returns the distance from it to each one of the three closest *feature points*.

```
vec3 gpuCellNoise3D(const in vec3 xyz)
{
   int xi = int(floor(xyz.x));
   int yi = int(floor(xyz.y));
   int zi = int(floor(xyz.z));

   float xf = xyz.x - float(xi);
   float yf = xyz.y - float(yi);
   float zf = xyz.z - float(zi);

   float dist1 = 9999999.0;
   float dist2 = 9999999.0;
   float dist3 = 9999999.0;
   vec3 cell;

   for (int z = -1; z <= 1; z++)
   {
```

```
      for (int y = -1; y <= 1; y++)
      {
         for (int x = -1; x <= 1; x++)
         {
            cell = snoise(xi + x, yi + y, zi + z) * noiseScale;
            cell.x += (float(x) - xf);
            cell.y += (float(y) - yf);
            cell.z += (float(z) - zf);

            float dist = dot(cell, cell);

            if (dist < dist1)
            {
               dist3 = dist2;
               dist2 = dist1;
               dist1 = dist;
            }
            else if (dist < dist2)
            {
               dist3 = dist2;
               dist2 = dist;
            }
            else if (dist < dist3)
            {
               dist3 = dist;
            }
         }
      }
   }

   return vec3(sqrt(dist1), sqrt(dist2), sqrt(dist3));
}
```

A *fractal sum* function of *Worley noise* is created the same way as with *simplex noise* in Appendix 6.3:

```
vec3 fractalcell(vec3 pos, float minFreq, float maxFreq, float multiFreq)
{
   vec3 value = vec3(0.0);
   float f;
   float ns;

   for (f=minFreq;f<maxFreq;f*=multiFreq)
   {
      value += gpuCellNoise3D(vec3(pos * f)) / f;
   }

   return value;
}
```

APPENDIX C PROCEDURAL TEXTURES SHADERS

All the procedural textures used in Chapter 5 were implemented in the fragment shader using GLSL (OpenGL Shading Language). Vertex shader is basically the same for all the textures, it only purpose is calculate light related values and vertices properties, including the values from the geodesic distance fields, to be used in the fragment shader. The only exception is the displacement example, that also uses the two new shaders of the rendering pipeline: *tessellation control shader* and *tessellation evaluation shader*.

C.1 Basic Vertex Shader

In vertex shader we calculate illumination related data and also send vertex attributes down the pipeline to be interpolated for the fragments to the next rendering stage. The attributes are position, normal, and geodesic distance value as texture coordinate. When using multiple distance fields, the value of each one is assigned to a different texture coordinate.

```
varying vec3 position;
varying vec2 Texcoord;
varying vec3 ViewDirection;
varying vec3 LightDirection;
varying vec3 Normal;

void main(void)
{
   vec3 fvLightPosition = gl_LightSource[0].position.xyz;
   vec3 fvEyePosition = vec3(0.0,0.0,0.0);
   vec4 fvObjectPosition = gl_ModelViewMatrix * gl_Vertex;

   ViewDirection  = fvEyePosition - fvObjectPosition.xyz;
   LightDirection = fvLightPosition - fvObjectPosition.xyz;
   Normal         = gl_NormalMatrix * gl_Normal;

   gl_TexCoord[0]  = gl_MultiTexCoord0;
   gl_TexCoord[1]  = gl_MultiTexCoord1;
   gl_TexCoord[2]  = gl_MultiTexCoord2;
   position     = gl_Vertex.xyz;

   gl_Position = ftransform();
}
```

C.2 Sand/Water texture fragment shader

This example uses two textures/materials, one tries to simulate the look of sand grains with high frequency noise to perturb normals, the other tries to resemble sea water flow using a fractal sum of Worley noise. We removed the functions related to simplex noise and Worley noise, since they are in Appendix A and B respectively.

Tha following parameters come from the application:

- **maxDistance** is the longest geodesic distance measured in the distance field. It is used to normalize the distance value of each vertex.
- **time** is a value that increases at every frame rendered. It is used to animate the water texture so it appears to flow.
- **distanceThreshold** is a value between 0.0 and 1.0, to define where the threshold of the two textures is. The region of the surface where the normalized geodesic distances are bellow the value of *distanceThreshold* will be textured with one of the textures, and the region where the distance field is above this value will be textured by the other texture.
- **noiseScale** is used by the Worley noise function (see Appendix 6.3). It controls how much the *feature points* of the noise grid will be displaced.

First we define the colors and some parameters. The parameters can be changed at will to change the appearance of the resulting texture. They are basically related to scale, and might be changed depending on the size of the surface. Some are hardcoded in the shader to give the appearance we wanted to surfaces of the size of the models we used in the examples. They are:

- **bumpScale**: we created the appearance we want by creating a pattern with simplex and Worley noise and use such pattern to perturb the normals of the fragments. The *bumpScale* variable controls the strength of this perturbation.
- **sandNoiseScale**: this variable controls the frequency of the noise for the sand texture. Higher values results in smaller grains of sand.
- **cellScale**: *cellScale* is quite similar to *sandNoiseScale*, but controlling the appearance of the water texture. It modifies the frequency of the Worley noise, so higher values lead to more water ripples.
- **shoreWidth**: this value defines how smooth will be the transition between the two textures in the threshold area. Higher values mean a more smooth mix.

We start by defining the threshold of the two textures with a *smoothstep()* function based on the *shoreWidth* and *distanceThreshold* parameters and the normalized distance value assigned to the fragment. Then the perturbation for both materials are evaluated using the fractal form of Worley noise for water and a single call of scaled simplex noise for sand.

This was the pattern creation part. Now we use the created pattern in the lighting calculations by mixing both patterns based on the threshold (*shore variable*) and using it to modify the normal. The color of the two texture are also mixed using the threshold value for a smooth and coherent color transition. The process is quite similar for the other texture examples with minor changes.

```
uniform float maxDistance;
uniform float time;
uniform float distanceThreshold;
uniform float noiseScale;

uniform sampler2D gradTexture; //used by simplex noise function
uniform sampler2D permTexture; //used by simplex noise function

varying vec3 position;
varying vec2 Texcoord;
varying vec3 ViewDirection;
varying vec3 LightDirection;
varying vec3 Normal;
void main(void)
{

   vec4 fvAmbient = vec4(0.0,0.0,0.0,1.0);
   vec4 fvSpecular = vec4(0.549,0.528,0.164,1.0);
   vec4 fvDiffuseWater = vec4(0.177,0.503,0.413,1.0);
   vec4 fvDiffuseLand = vec4(0.894,0.969,0.755,1.0);
   float fSpecularPower = 12.0;

   float bumpScale = 0.15;
   float sandNoiseScale = 1000.0;
   float cellScale = 7.0;
   float shoreWidth = 0.0485;

   float distanceValue = gl_TexCoord[0].x / maxDistance;

   float shore = 1.0 - smoothstep(distanceValue - shoreWidth * 0.5,
    distanceValue + shoreWidth * 0.5,
    distanceThreshold + shoreWidth);

   vec3 fractalWorleyNoise = fractalcell(position * cellScale
     + time * 0.5, 1.0, 5.0, 2.0);

   float waterBump = fractalWorleyNoise.x;
   float sandBump = snoise(position * sandNoiseScale);

   //
   //light stuff
   //
   vec3  fvLightDirection = normalize( LightDirection );
   vec3  fvNormal         = normalize( Normal )
    + (mix(sandBump, waterBump, shore)) * bumpScale;
   float fNDotL           = dot( fvNormal, fvLightDirection );

   vec3  fvReflection     = normalize( ( ( 2.0 * fvNormal ) * fNDotL )
```

```
    - fvLightDirection );
   vec3  fvViewDirection  = normalize( ViewDirection );
   float fRDotV           = max( 0.0,
    dot( fvReflection, fvViewDirection ) );

   vec4  fvTotalAmbient   = fvAmbient * 1.0;
   vec4  fvTotalDiffuse   = mix(fvDiffuseLand, fvDiffuseWater, shore)
    * fNDotL * 1.0;
   vec4  fvTotalSpecular  = fvSpecular *
    ( pow( fRDotV, fSpecularPower ) );

   gl_FragColor = ( fvTotalAmbient + fvTotalDiffuse
    + fvTotalSpecular );

}
```

C.3 Moss/Granite texture fragment shader

The idea is the same as the Water/Sand shader. The changes are in the colors, scale parameters and functions to create the pattern for each texture. The moss texture is created by a combination of Worley noise and turbulence of simplex noise. As for the granite texture, it created by turbulence of simplex noise alone.

```
uniform float maxDistance;
uniform float time;
uniform float distanceThreshold;
uniform float noiseScale;

uniform sampler2D gradTexture; //used by simplex noise function
uniform sampler2D permTexture; //used by simplex noise function

varying vec3 position;
varying vec2 Texcoord;
varying vec3 ViewDirection;
varying vec3 LightDirection;
varying vec3 Normal;

void main(void)
{

   vec4 fvAmbient = vec4(0.0,0.0,0.0,1.0);
   vec4 fvSpecular = vec4(0.549,0.528,0.164,1.0);
   vec4 fvDiffuseGranite = vec4(0.222,0.219,0.174,1.0);
   vec4 fvDiffuseMoss = vec4(0.094,0.142,0.061,1.0);
   float fSpecularPower = 12.0;
   float bumpScale = 1.0;

   float shoreNoiseScale = 153.0;
   float graniteNoiseScale = 115.0;
```

```
    float mossNoiseScale = 163.0;
    float cellScale = 7.0;
    float shoreWidth = 0.1;

    float distanceValue = gl_TexCoord[0].x / maxDistance;


    float shore = 1.0 - smoothstep(distanceValue - shoreWidth * 0.5,
     distanceValue + shoreWidth * 0.5, distanceThreshold + shoreWidth);

    vec3 worley = fractalcell(position * (cellScale)
       + time * 0.5, 1.0, 5.0, 2.0);


    float mossCells = 1.0 - worley.x;

    //
    //light stuff
    //
    vec3  fvLightDirection = normalize( LightDirection );
    vec3  fvNormal         = normalize( Normal ) +
     (mix(mossCells +
     turbulence(position * mossNoiseScale, 1.0, 5.0, 2.0),
     turbulence(position * graniteNoiseScale, 1.0, 5.0, 2.0)
      * 0.5, shore)) * bumpScale;
    float fNDotL           = dot( fvNormal, fvLightDirection );

    vec3  fvReflection     = normalize( ( ( 2.0 * fvNormal ) * fNDotL )
       - fvLightDirection );
    vec3  fvViewDirection  = normalize( ViewDirection );
    float fRDotV           = max( 0.0, dot( fvReflection,
     fvViewDirection ) );


    vec4  fvTotalAmbient   = fvAmbient * 1.0;
    vec4  fvTotalDiffuse   = mix(fvDiffuseMoss, fvDiffuseGranite, shore)
     * fNDotL * 1.0;
    vec4  fvTotalSpecular  = mix(vec4(0.0), fvSpecular *
     ( pow( fRDotV, fSpecularPower ) ), shore);

    gl_FragColor = ( fvTotalAmbient + fvTotalDiffuse
     + fvTotalSpecular * 0.5);
}
```

C.4 Paint/Metal/Rust texture fragment shader

Although this example is more complex than the previous ones, it still follows the same idea. Once again there are different colors, parameters and functions, but the main

different is that now we are mixing three different textures (four if we consider the transparent one). Now we need to replace the simple mix by a nesting of mixes that will represent a priority order for the textures. For so, we also need to define the threshold of each pair of textures in the priority order, i.e. the threshold between the paint and the metal textures, another one between the metal and rust texture and a final one between the rust texture and the fully transparent one. We can use different threshold values for each threshold (*thresholdLayer1*, *thresholdLayer2*, *thresholdLayer3*).

We start defining the color of each texture and some scale parameters for the noise functions and variations. Then we define each of the 3 thresholds based on the same distance value for each threshold (we use only one distance field, so each fragment has only one distance value assigned) and values from *thresholdLayer* 1, 2 and 3. The color of the stained metal is created by a fractal sum of simplex noise. The rust texture color is also created by fractal sum but we also modify the alpha value, based on the fractal sum, to create holes with transparency. The final color is defined by the nested mix of the color of each texture. Now we set the bump values of each texture, to perturb the normal latter. The final perturbation is a nested mix as well. Another interesting difference is that we add some noise to *thresholdLayer3* when evaluating the threshold between the rust texture and the transparent one, in order to create a irregular propagation front.

```
uniform float thresholdLayer1;
uniform float thresholdLayer2;
uniform float thresholdLayer3;

uniform float bumpScale;
uniform float time;
uniform float maxDistance;

varying vec2 Texcoord;
varying vec3 ViewDirection;
varying vec3 LightDirection;
varying vec3 Normal;
varying vec3 position;

uniform sampler2D permTexture;  //for simplex noise
uniform sampler2D gradTexture;  //for simplex noise
void main( void )
{
   vec4 fvAmbient1 = vec4(0.0,0.0,0.0,1.0);
   vec4 fvSpecular1 = vec4(1.0,1.0,1.0,1.0);
   vec4 fvDiffuse1 = vec4(0.11,0.16,0.48,1.0);
   float bump1 = 1.0;

   vec4 fvAmbient2 = vec4(0.0,0.0,0.0,1.0);
   vec4 fvSpecular2 = vec4(0.18,0.18,0.18,1.0);
   vec4 fvDiffuse2 = vec4(0.64,0.64,0.64,1.0);
   float bump2 = 0.8;

   vec4 fvAmbient3 = vec4(0.0,0.0,0.0,1.0);
   vec4 fvSpecular3 = vec4(0.0,0.0,0.0,1.0);
```

```
vec4 fvDiffuse3 = vec4(0.29,0.17,0.13,1.0);
float bump3 = 1.0;

float fSpecularPower = 60.0;

float metalNoiseScale = 5.0;
float rustNoiseScale = 4.62;
float rustBorderNoiseScale = 0.0;
float rustBorderNoiseScale2 = 10.0;
float paintBorderNoiseScale = 0.0;

vec4 fvAmbient;
vec4 fvSpecular;
vec4 fvDiffuse;
float bump;

float distanceValue = gl_TexCoord[0].x / maxDistance;
float thresholdWidth = 0.005;

//defines the end of each layer
float threshold1 = smoothstep(distanceValue - thresholdWidth,
  distanceValue + thresholdWidth, thresholdLayer1
   + snoise(position * paintBorderNoiseScale));
float threshold2 = smoothstep(distanceValue - 0.08,
  distanceValue + 0.08, thresholdLayer2
   + snoise(position * rustBorderNoiseScale));
float threshold3 = smoothstep(distanceValue - 0.0,
  distanceValue + 0.0, thresholdLayer3
   + snoise(position * rustBorderNoiseScale2) * 0.01);

fvDiffuse2 -= fractalsum(position
 * metalNoiseScale, 1.0, 5.0, 2.0) * 0.1;
fvDiffuse3 -= fractalsum(position
 * rustNoiseScale, 1.0, 5.0, 2.0)
 * (distanceValue - clamp(thresholdLayer2,0.0,1.5) - 0.5)
   * 0.3;
float fvDiffuse3AlphaMod = clamp(fractalsum(position
 * rustNoiseScale, 1.0, 5.0, 2.0)
 * (distanceValue - thresholdLayer2 - 0.5)
 * 16.0, 0.0, 1000.0);
fvDiffuse3 -= vec4(vec3(0.0), 0.1) * fvDiffuse3AlphaMod;

fvDiffuse = mix(mix(mix(fvDiffuse1,fvDiffuse2,vec4(threshold1)),
  fvDiffuse3, vec4(threshold2)), vec4(0.0), vec4(threshold3));
fvSpecular = mix(mix(mix(fvSpecular1,fvSpecular2,vec4(threshold1)),
  fvSpecular3, vec4(threshold2)), vec4(0.0), vec4(threshold3));
fvAmbient = mix(mix(mix(fvAmbient1,fvAmbient2,vec4(threshold1)),
  fvAmbient3, vec4(threshold2)), vec4(0.0), vec4(threshold3));
```

```
    bump1 = -threshold1 * bump1;
    bump2 = snoise(position * 400.0) * 0.025 * bump2;
    bump3 = bump2 + length(fvDiffuse3) * 0.5 * bump3;

    bump = mix(mix(mix(bump1,bump2,threshold1), bump3, threshold2),
     0.0, threshold3);

    //
    //light stuff
    //
    vec3  fvLightDirection = normalize( LightDirection );
    vec3  fvNormal         = normalize( Normal ) + bump
     * bumpScale;
    float fNDotL           = abs(dot( fvNormal, fvLightDirection ));

    vec3  fvReflection     = normalize( ( ( 2.0 * fvNormal )
      * fNDotL ) - fvLightDirection );
    vec3  fvViewDirection  = normalize( ViewDirection );
    float fRDotV           = max( 0.0, dot( fvReflection,
      fvViewDirection ) );

    vec4  fvTotalAmbient   = fvAmbient * 1.0;
    vec4  fvTotalDiffuse   = (fvDiffuse) * fNDotL * 1.0;
    vec4  fvTotalSpecular  = fvSpecular * ( pow( fRDotV,
      fSpecularPower ) );

    //alpha value is define based on threshold value
    gl_FragColor = ( fvTotalAmbient + fvTotalDiffuse
      + fvTotalSpecular );

}
```

C.5 Water/Lava texture fragment shader

This shader is quite similar to the one for the Paint/Metal/Rust texture. The main difference here is that we use two distinct distance fields. One for the propagation of the water texture and the other for the lava texture. The water and lava texture are created the same way as the water from the Sand/Water texture: with fractal sum of Worley noise. The other material tries to create the look of some burnt surface by using turbulence with simplex noise. We also add some low frequency noise to *thresholdLayer1* when evaluating the threshold between the lava and the burnt texture to make the lava propagation a little irregular.

```
uniform sampler2D permTexture; //for simplex noise
uniform sampler2D gradTexture; //for simplex noise

uniform float thresholdLayer1;
```

```
uniform float thresholdLayer2;

uniform float bumpScale;
uniform float time;
uniform float maxDistance;

varying vec3 position;
varying vec3 ViewDirection;
varying vec3 LightDirection;
varying vec3 Normal;

#define noiseScale 2.2
#define cellScale 65.16720
#define rockNoiseScale 47.0
#define lavaBorderNoiseRange 0.03

void main( void )
{

   vec4 fvAmbient1 = vec4(0.39,0.09,0.09,1.0);
   vec4 fvSpecular1 = vec4(0.79,0.76,0.24,1.0);
   vec4 fvDiffuse1 = vec4(0.27,0.08,0.06,1.0);
   float bump1 = -0.04;

   vec4 fvAmbient2 = vec4(0.0,0.0,0.0,1.0);
   vec4 fvSpecular2 = vec4(0.78,0.78,0.78,1.0);
   vec4 fvDiffuse2 = vec4(0.21,0.43,0.48,1.0);
   float bump2 = -0.16;

   vec4 fvAmbient3 = vec4(0.0,0.0,0.0,1.0);
   vec4 fvSpecular3 = vec4(0.07,0.07,0.07,1.0);
   vec4 fvDiffuse3 = vec4(0.14,0.14,0.14,1.0);
   float bump3 = 0.66;

   float fSpecularPower = 26.0;

   float distanceValue1 = gl_TexCoord[0].x / maxDistance;
   float distanceValue2 = gl_TexCoord[1].x / maxDistance;


   float thresholdWidth1 = 0.01;
   float thresholdWidth2 = 0.05;

   //defines the end of each layer
   float threshold1 = smoothstep(distanceValue1 - thresholdWidth1,
     distanceValue1 + thresholdWidth1, thresholdLayer1 +
      snoise(position * 10.0) * lavaBorderNoiseRange);
   float threshold2 = smoothstep(distanceValue2 - thresholdWidth2,
```

```
    distanceValue2 + thresholdWidth2, thresholdLayer2);

  vec3 worley = fractalcell(position * cellScale + time * 0.5,
    1.0, 5.0, 2.0);

  bump1 *= worley.x;
  bump1 -= 1.0;
  fvDiffuse1 += vec4(vec3(0.20,0.15,0.05), 1.0) * worley.x;
  bump2 *= worley.x;
  bump3 = turbulence(position * rockNoiseScale, 1.0, 5.0, 2.0)
    * bump3;

  vec4 fvDiffuse = mix(mix(fvDiffuse3,fvDiffuse1,
   vec4(threshold1)), fvDiffuse2, vec4(threshold2));
  vec4 fvSpecular = mix(mix(fvSpecular3,fvSpecular1,
   vec4(threshold1)), fvSpecular2, vec4(threshold2));
  vec4 fvAmbient = mix(mix(fvAmbient3,fvAmbient1,
   vec4(threshold1)), fvAmbient2, vec4(threshold2));

  float bump = mix(mix(bump3,bump1,threshold1), bump2,
   threshold2);

  //
  //light stuff
  //
  vec3  fvLightDirection = normalize( LightDirection );
  vec3  fvNormal         = normalize( Normal ) + bump * bumpScale;
  float fNDotL           = abs(dot( fvNormal, fvLightDirection ));

  vec3  fvReflection     = normalize( ( ( 2.0 * fvNormal ) * fNDotL)
   - fvLightDirection );
  vec3  fvViewDirection  = normalize( ViewDirection );
  float fRDotV      = max( 0.0, dot( fvReflection, fvViewDirection));

  vec4  fvTotalAmbient   = fvAmbient * 1.0;
  vec4  fvTotalDiffuse   = (fvDiffuse) * fNDotL * 1.0;
  vec4  fvTotalSpecular = fvSpecular * ( pow( fRDotV, fSpecularPower));

  //alpha value is define based on threshold value
  gl_FragColor = ( fvTotalAmbient + fvTotalDiffuse + fvTotalSpecular);

}
```

C.6 Adaptive Displacement

This example is the most complex one, and make use of the new functionality in the rendering pipeline: the hardware tessellator.

C.6.1 Vertex Shader

The vertex shader is very different from the ones for the procedural textures, and much more complex. The reason is because we want to refine our original mesh adaptively, based on our wave pattern. We want the mesh to be more refined where the frequency of the waves is high and where there will be more displacement so we can have a smooth silhouette. Both the frequency and amplitude of the waves are inversely proportional to the distance from the source of the wave, i.e. the distance value of the fragment. This way, our function must be calculated in the vertex shader, so the result can be used in the next stage, the *tessellation control shader*, to set how much the mesh must be refined. To create the series of waves that flow in the direction of one distance field, first we use two *smoothstep()* to create a smooth bump line, the width of this bump line will represent how far the waves will reach, then we multiply this single large bump with sine function to transform it into a series of shorter waves. Then the waves function of each distance field are added. But to define the tessellation level, the sum of the bump lines is enough.

```
uniform mat4 Modelview;

varying vec3 vPosition;
varying vec3 vNormal;
varying float vDisp;
varying float vDistanceValue;
varying float vDistanceValue2;

uniform float animation2;
uniform float stripeWidth;
uniform float bumpScale;
uniform float frequency;
uniform float maxDistance;

void main(void)
{

    vec4 Position = gl_Vertex;
    vec3 Normal = gl_Normal;
    float texCoord0 = gl_MultiTexCoord0.x;
    float texCoord1 = gl_MultiTexCoord1.x;

    mat3 NormalMatrix = mat3(Modelview);

    //convert normal and tangent (from main program) into eye space
    vNormal = NormalMatrix * Normal.xyz;
    vPosition = vec3(Modelview * Position);

    vDistanceValue = texCoord0 / maxDistance;
    vDistanceValue2 = texCoord1 / maxDistance;

    float distanceValue = vDistanceValue;
```

```
    float distanceValue2 = vDistanceValue2;

    float animation = 0.0;

//first wave
    float north = smoothstep(distanceValue - stripeWidth * 0.5,
      distanceValue + stripeWidth * 0.5, animation);
    float south = 1.0 - smoothstep(distanceValue - stripeWidth * 0.5,
      distanceValue + stripeWidth * 0.5, animation + stripeWidth);
    float bump = north + south;

    vDisp = (1.0 - bump) * (frequency * (1.0 - distanceValue));

    //second wave
    float north2 = smoothstep(distanceValue2 - stripeWidth * 0.5,
      distanceValue2 + stripeWidth * 0.5, animation);
    float south2 = 1.0 - smoothstep(distanceValue2 - stripeWidth
      * 0.5, distanceValue2 + stripeWidth * 0.5, animation + stripeWidth);
    float bump2 = north2 + south2;
    vDisp += (1.0 - bump2) * (frequency * (1.0 - distanceValue2));

    gl_Position = ftransform();
}
```

C.6.2 Tessellation Control Shader

The purpose of the *tessellation control shader* is to determinate how much the surface must be further tessellated by the fixed functionality tessellation stage. We do this by defining how many times each edge of the triangle must be subdivided based on the value of our wave function (*vDisp*), from that comes from the vertex shader, on the vertices of the given edge.

```
#version 400 compatibility
#extension GL_ARB_tessellation_shader : enable

layout(vertices = 3) out;
in vec3 vPosition[];
in vec3 vNormal[];
in float vDisp[];
in float vDistanceValue[];
in float vDistanceValue2[];

out vec3 tcNormal[];
out vec3 tcPosition[];
out float tcDistanceValue[];
out float tcDistanceValue2[];

uniform float TessFactor;
```

```
#define ID gl_InvocationID

void main()
{
    tcPosition[ID] = vPosition[ID];
    tcNormal[ID] = vNormal[ID];
    tcDistanceValue[ID] = vDistanceValue[ID];
    tcDistanceValue2[ID] = vDistanceValue2[ID];


    if (ID == 0) {

        float TessLevelInner = floor((vDisp[0] + vDisp[1] + vDisp[2])
          / 3.0) * TessFactor;
        float TessLevelOuter0 = floor((vDisp[1] + vDisp[2]) / 2.0)
          * TessFactor;
        float TessLevelOuter1 = floor((vDisp[0] + vDisp[2]) / 2.0)
          * TessFactor;
        float TessLevelOuter2 = floor((vDisp[0] + vDisp[1]) / 2.0)
          * TessFactor;

        if (TessLevelInner < 1.0) {
TessLevelInner = 1.0;
        }

        if (TessLevelOuter0 < 1.0) {
TessLevelOuter0 = 1.0;
        }

        if (TessLevelOuter1 < 1.0) {
TessLevelOuter1 = 1.0;
        }

        if (TessLevelOuter2 < 1.0) {
TessLevelOuter2 = 1.0;
        }

        gl_TessLevelInner[0] = floor(TessLevelInner);
        gl_TessLevelOuter[0] = floor(TessLevelOuter0);
        gl_TessLevelOuter[1] = floor(TessLevelOuter1);
        gl_TessLevelOuter[2] = floor(TessLevelOuter2);
    }
}
```

C.6.3 Tessellation Evaluation Shader

Now we need to set the position of the vertices (including the ones created by the tessellator) based on our wave function, so in this shader we need to evaluate our function again and this time including the sine part to create the series of shorter waves. We also

need to set the properties of the vertices created by the tessellator by the interpolation of the properties of the original vertices, and modify the normals.

```
#version 400 compatibility
#extension GL_ARB_tessellation_shader : enable

layout(triangles, equal_spacing, ccw) in;
in vec3 tcPosition[];
in vec3 tcNormal[];
in float tcDistanceValue[];
in float tcDistanceValue2[];

out vec3 teNormal;
out vec3 teLightDir;
out vec3 teEyeDir;

uniform mat4 Projection;
uniform float animation2;
uniform float stripeWidth;
uniform float bumpScale;
uniform float frequency;
uniform float maxDistance;

float bias(float b, float x) {
    return pow(x, log2(b)/log2(0.5));
}

void main()
{

    vec3 n0 = gl_TessCoord.x * tcNormal[0];
    vec3 n1 = gl_TessCoord.y * tcNormal[1];
    vec3 n2 = gl_TessCoord.z * tcNormal[2];
    teNormal = (n0 + n1 + n2);

    vec3 p0 = gl_TessCoord.x * tcPosition[0];
    vec3 p1 = gl_TessCoord.y * tcPosition[1];
    vec3 p2 = gl_TessCoord.z * tcPosition[2];
    vec3 tePosition = (p0 + p1 + p2);

    float d10 = gl_TessCoord.x * tcDistanceValue[0];
    float d11 = gl_TessCoord.y * tcDistanceValue[1];
    float d12 = gl_TessCoord.z * tcDistanceValue[2];
    float teDistanceValue = (d10 + d11 + d12);

    float d20 = gl_TessCoord.x * tcDistanceValue2[0];
    float d21 = gl_TessCoord.y * tcDistanceValue2[1];
    float d22 = gl_TessCoord.z * tcDistanceValue2[2];
    float teDistanceValue2 = (d20 + d21 + d22);
```

```
    vec3 fvLightPosition = vec3(0.0,0.0,0.0);
    vec3 fvEyePosition = vec3(0.0,0.0,0.0);
    vec4 fvObjectPosition = vec4(tePosition,1.0);
    vec4 LightPos = vec4(fvLightPosition,1.0);
    vec4 EyePos = vec4(fvEyePosition,1.0);
    vec3 LightDirection = LightPos.xyz - fvObjectPosition.xyz;

    teEyeDir = normalize(EyePos.xyz - fvObjectPosition.xyz);
    teLightDir = normalize(LightPos.xyz - fvObjectPosition.xyz);

    float distanceValue = teDistanceValue;
    float distanceValue2 = teDistanceValue2;

    float animation = 0.0;

//first wave
    float north = smoothstep(distanceValue - stripeWidth * 0.5,
      distanceValue + stripeWidth * 0.5, animation);
    float south = 1.0 - smoothstep(distanceValue - stripeWidth
      * 0.5, distanceValue + stripeWidth * 0.5, animation + stripeWidth);
    float bump = north + south;

    float isolines = sin(animation2 + distanceValue
      * (frequency * (1.0 - distanceValue))) * (1.0 - distanceValue);

    bump = (1.0 - bump) * isolines;

    //second wave

    float north2 = smoothstep(distanceValue2 - stripeWidth * 0.5,
      distanceValue2 + stripeWidth * 0.5, animation);
    float south2 = 1.0 - smoothstep(distanceValue2 - stripeWidth
      * 0.5, distanceValue2 + stripeWidth * 0.5, animation + stripeWidth);
    float bump2 = north2 + south2;

    float isolines2 = sin(animation2 + distanceValue2
      * (frequency * (1.0 - distanceValue2))) * (1.0 - distanceValue2);

    bump2 = (1.0 - bump2) * isolines2;

    if (distanceValue2 > 0.0) {
bump = bump + bump2;
    }

    bump *= bumpScale;
    vec2 p = vec2(bump); // angle of vNormal changing over the bump area
```

```
    float d, f;
    d = p.x * p.x + p.y * p.y;
    f = 1.0 / sqrt(d + 1.0);

    tePosition += bump * teNormal;

    teNormal = normalize(vec3(p.x, p.y, 1.0) + teNormal);

    gl_Position = Projection * vec4(tePosition, 1.0);

}
```

C.6.4 Fragment Shader

Finally, in the fragment shader, we just need to do lighting calculations.

```
#version 400 core

out vec4 FragColor;
in vec3 teLightDir;
in vec3 teEyeDir;
in vec3 teNormal;

uniform sampler2D EnvMap;

void main()
{

    bool useEnvmap = true;

    vec4 fvAmbient = vec4(0.0,0.0,0.0,1.0);
    vec4 fvSpecular = vec4(1.0,1.0,1.0,1.0);
    vec4 fvDiffuse = vec4(0.5,0.5,0.5,1.0);
    float fSpecularPower = 60.0;
    float att = 1.0;

    vec3 Normal = teNormal;

    float fNDotL           = dot( Normal, teLightDir );
    vec3  fvReflection     = normalize( ( ( 2.0 * Normal )
      * fNDotL ) - teLightDir );
    float fRDotV           = max( 0.0, dot( fvReflection,
      teEyeDir ) );
    vec4  fvTotalAmbient   = fvAmbient * 1.0;
    vec4  fvTotalDiffuse   = vec4(fvDiffuse.xyz, 1.0)
      * fNDotL * 1.0;
    vec4  fvTotalSpecular  = fvSpecular
      * ( pow( fRDotV, fSpecularPower ) );
```

```
    vec4  materialColor    = ( fvTotalAmbient +
     att*fvTotalDiffuse + att*fvTotalSpecular );

    vec4 finalColor = materialColor;


    if (useEnvmap) {
       fvReflection.z += 1.0;
       float inv_m = 0.5/sqrt(dot(fvReflection,fvReflection));
       vec2 index = fvReflection.xy * inv_m + 0.5;


       vec4 envColor = vec4 (texture2D(EnvMap, index));
       finalColor = envColor +  fvTotalSpecular;
    }

    FragColor = finalColor;
}
```