Ecole Doctorale IAEM Lorraine
Département de formation doctorale en informatique

Exploration et rendu de textures synthétisées
Exploring and rendering synthesized textures

Thèse
présentée pour l'obtention du titre de
Docteur de l'Université de Lorraine
(spécialité informatique)
Par
Anass Lasram

Thèse soutenue publiquement le 10 décembre 2012

Membres du jury

Directeurs de thèse:	M. Bruno LEVY	Directeur de recherche INRIA
	M. Sylvain LEFEBVRE	Chargé de recherche INRIA
Rapporteurs:	M. Eric GALIN	Professeur à l'Université Lyon 2
	M. Li-Yi WEI	Professeur à l'Université de Hong Kong
Président:	M. Jean-Michel DISCHLER	Professeur à l'Université de Strasbourg
Examinateurs:	M. Carsten DACHSBACHER	Professeur à Karlsruhe Institute of Technology
	M. Dominique MERY	Professeur à l'Université de Lorraine

Remerciements

Je voudrais tout d'abord exprimer ma profonde gratitude à mon encadrant, Sylvain Lefebvre, qui a été et restera un mentor exceptionnel. Je le remercie pour son suivi rigoureux, son amitié, les projets passionnants dans lesquels il m'a impliqué et pour tout ce que j'ai pu apprendre de lui.

Je souhaite aussi remercier mon directeur de thèse, Bruno Lévy, pour m'avoir inlassablement soutenu et encouragé.

J'adresse tous mes remerciements à messieurs Eric Galin et Li-Yi Wei de l'honneur qu'ils m'ont fait en acceptant d'être rapporteurs de cette thèse. Je suis aussi particulièrement honoré par la présence de messieurs Carsten Dachsbacher, Jean-Michel Dischler et Dominique Méry dans mon jury.

Je remercie également notre collaborateur industriel Allegorithmic pour nous avoir fourni les superbes textures procédurales utilisées dans ce document. Je remercie particulièrement Cyrille Damez pour son support et sa forte implication dans tous mes projets de thèse.

Je tiens aussi à remercier tous ceux avec qui j'ai eu la chance et le plaisir de travailler. Merci en particulier à Ismael Garcia, Samuel Hornus, Gurpreet Singh et Shizhe Zhou.

Un grand merci à tous les collègues de la famille ALICE que j'ai eu la chance de côtoyer. Merci à Laurent Alonso, Dobrina Boltcheva, Nicolas Bonneel, Xavier Cavin, Alejandro Galindo, Isabelle Herlich, Thi-Phuong Ho, Bruno Jobard, Thomas Joste, Kun Liu, David Lopez, Chongyang Ma, Romain Merland, Vincent Nivoliers, Gilles-Philippe Paillé, Jeanne Pellerin, Alexandra Petitjean, Nicolas Ray, Dmitry Sokolov, Nicolas Saugnier, Kai Wang, Dong-Ming Yan et Rhaleb Zayer.

Je remercie aussi les chercheurs, thésards et ingénieurs du LORIA ayant contribué aux études utilisateur réalisées dans cette thèse.

Je remercie également Bernard Péroche, Jean-Claude Iehl et Imed Riadh Farah pour m'avoir initié à la recherche.

Je tiens enfin à remercier mes proches pour leur infaillible soutien, patience et amour. Merci à mes parents, Amel et Slaheddine, à mon grand-papa Ridhaya ainsi qu'à Khaoula, Souhaib et Lila.

Je dédie ce travail à la mémoire de
Zayneb Zmerli Baccouche
décédée le 8 octobre 2010

Contents

List of Figures

Chapter 1

Introduction

1.1 Context of research

Virtual environments are typically represented by a mixture of polygons capturing the scene geometry and images capturing the essential details of the various surfaces (Figure 1.1). These images, called *textures*, store fine scale geometric and material details such as the albedo or the surface normals and describe how the light interacts with the textured surfaces. For example, multiple texture layers storing a variety of material information have been used to render the realistic human skin of Figure 1.2.

Textures in interactive applications In interactive graphics applications virtual scenes have to be rendered at high frame rates. This is especially the case in applications containing fast motion animations like first person shooter games or racing simulations. Often, these applications need a frame rate at least as high as 60 images per second for a good experience.

To achieve such high frame rates, the geometry is often simplified and textures are used to retain the fine scale details. Figure 1.3 shows an example of a video game character where textures are responsible for complex details despite a relatively simple geometry. The character is first built with hundreds of thousands of polygons. Its geometry is then reduced to a few thousands of polygons. The textures however contain material information about the original high resolution model. When applying these textures to the low resolution model, complex details and fine scale lighting interactions can still be perceived.

Limitation of textures Textures are fundamental in any computer graphics application. However, their quality and diversity directly conflict with the available bandwidth and the available amount of video memory.

To face the memory limitation of textures, early real time three dimensional games had to repeat small sets of low resolution textures on simple planar geometric objects (Figure 1.4, Left). With the introduction of graphics hardware, the quantity of used textures increased

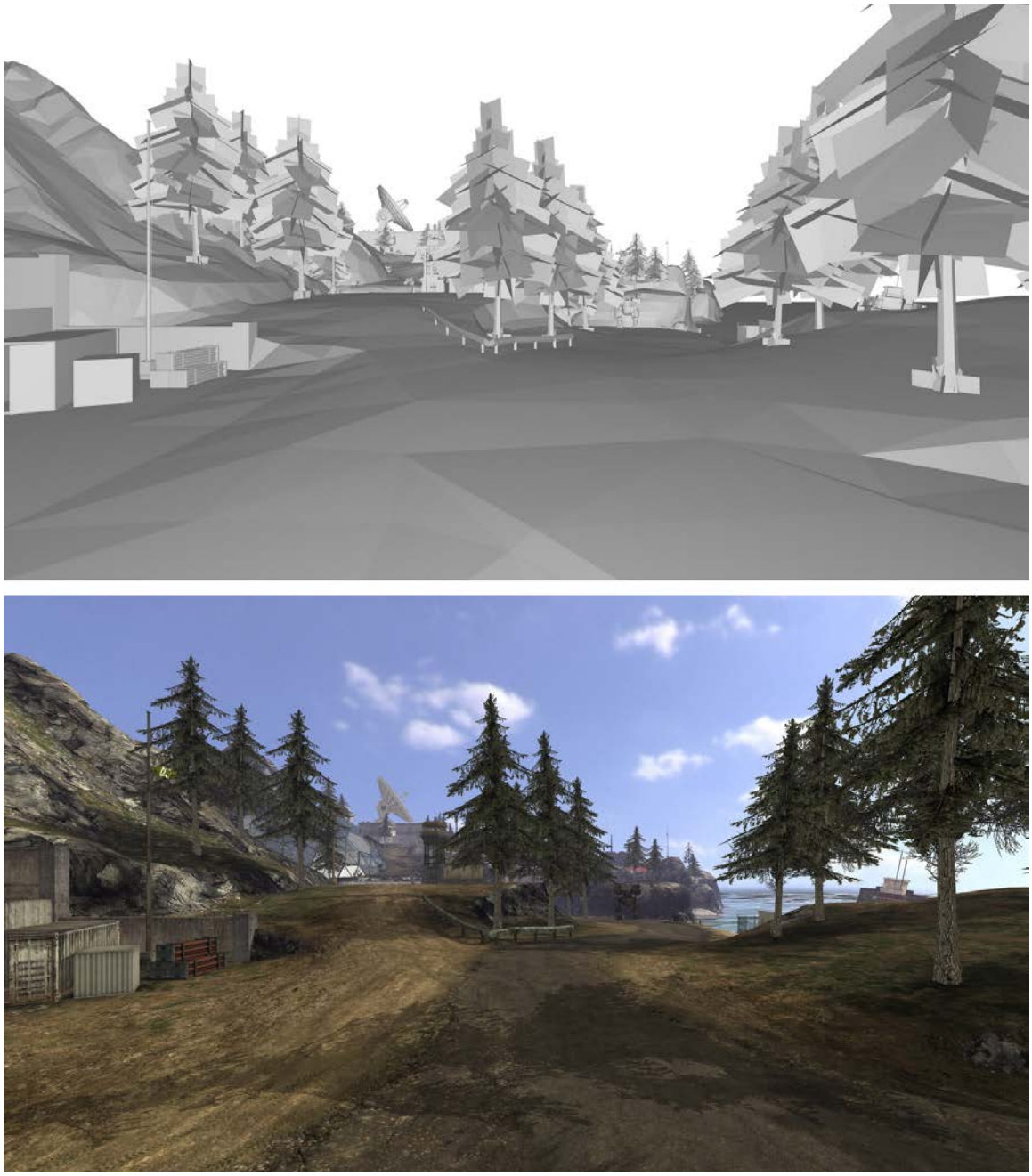

Figure 1.1: A video game scene with and without textures. *Top:* The scene is rendered without textures. *Bottom:* Applying textures adds lots of details to the scene. *Source:* Quake wars, Splash Damage/Activision.

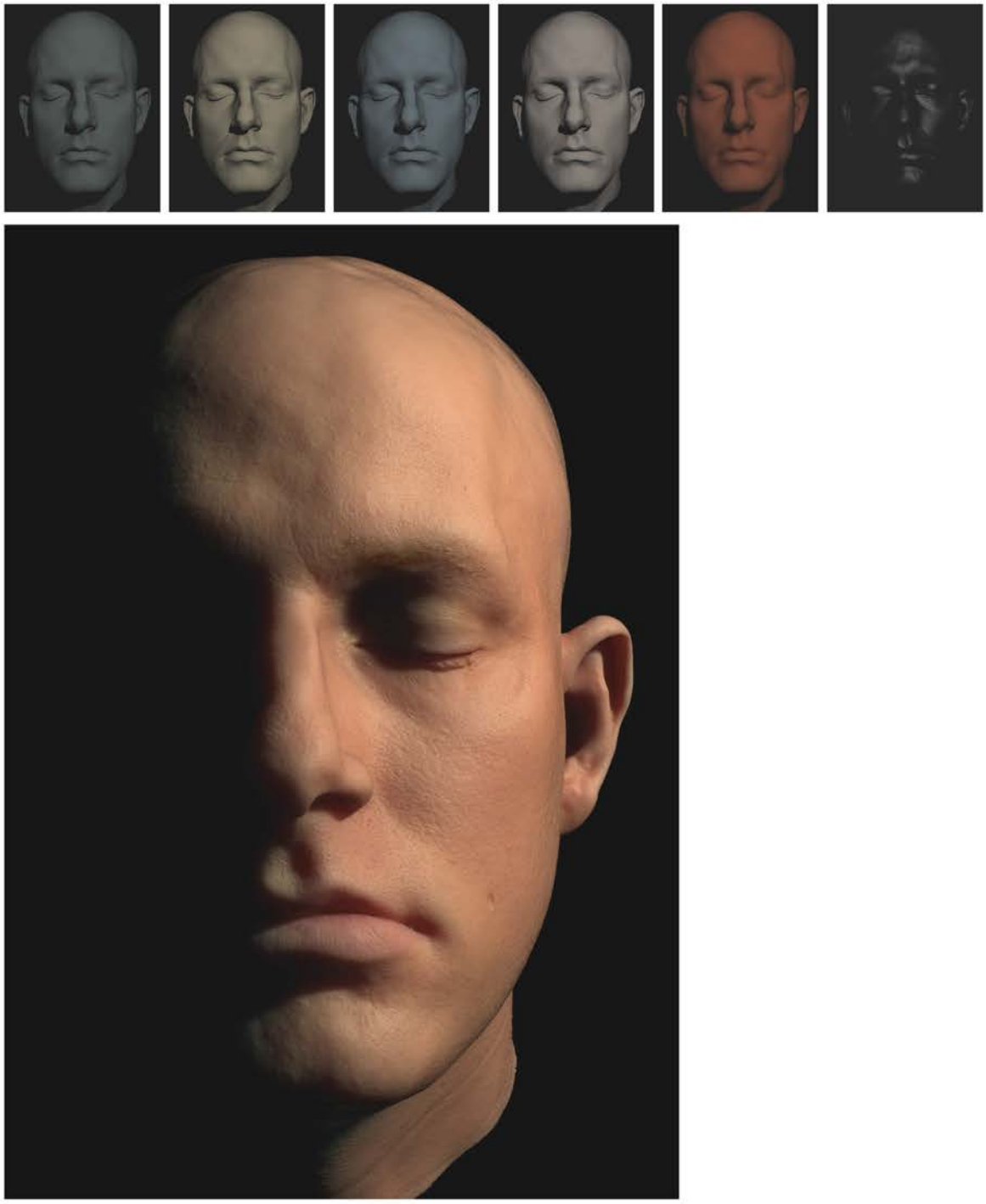

Figure 1.2: A rendering of a human face using multiple layers of skin materials stored as textures. *Top:* Many texture layers describing different material properties. *Bottom:* Rendering result. *Source:* Donner and Jensen [DJ05].

Figure 1.3: Textures mapped on a simple geometric model. *Left:* A low resolution mesh composed of a small number of polygons shown in wire–frame. This low resolution mesh is obtained by simplifying a high resolution version of the same model. *Middle:* Geometric and material details of the high resolution model are stored in 2D texture maps. The top texture represents the albedo, the middle texture represents surface normals and the bottom texture represents specular intensities. *Right:* Rendering of the low resolution mesh with textures. Note how complex details can be seen within a same flat polygon. *Source:* Guild Wars, ArenaNet/NCsoft.

Figure 1.4: Textures in old video games. *Left:* Quake, one of the early real time three dimensional games, makes use of very low resolution textures. These textures only range from 16×16 to 256×256 resolution. *Source:* Quake, id Software/GT Interactive. *Right:* The Quake 3 game exclusively targets machines having a graphics card with dedicated texture memory. This game uses a relatively large amount of textures with resolutions up to 512×512. *Source:* Quake 3 Arena, id Software/Activision.

(Figure 1.4, Right). These textures were not only applied to surfaces but were also used for special effects.

Most today's graphics hardware give full control over the graphics pipeline and game developers have taken advantage of this flexibility to exploit a tremendous amount of high quality textures. For instance, the game RAGE (Figure 1.5), the latest game from *id Software*, is able to give a unique look and feel to every surface of the vast environment that composes the game. To achieve this richness, the game makes use of a single extremely high resolution texture that in total size exceeds the maximum resolution supported by the hardware. This texture is transferred progressively to video memory in several parts depending on the viewer state. (The texturing technique used in RAGE is described in section 2.5).

Consequences in the video game industry The use of an enormous amount of textures to define the complexity, richness and realism of video game scenes not only makes textures the most memory expensive data but also implies huge production endeavors: Artists have to manually author each and every texture of the game and engineers have to design and implement new game engines to support the hundreds of gigabytes of raw texture data. This often leads to years of production time and considerable costs that can exceed a hundred million dollars budget [WB10].

Figure 1.5: A video game scene composed of a large amount of texture data. Note how each object has its unique high resolution texture and this even for fine scale objects like the grass blades. *Source:* RAGE, id Software/Bethesda Softworks.

Textures in new media The convergence between the Internet and computer graphics has led to the emergence of new media ranging from on–line role–playing games to geographical information programs like Google Earth. Such applications open the possibilities to wider and richer environments but suffer from even more pressure on data storage and bandwidth. Google Earth for instance allows users to navigate across the entire earth. The textures that compose the world are obtained from satellite imagery and their total size exceeds 70 terabytes of data [CDG+08]. Because of their important size, textures have to be transmitted progressively over the slow and congested Internet network.

Another rapidly growing industry is the mobile gaming industry [ST08, LLP12]. Mobile games are accessible to a broad audience with low costs. Today's mobile phones and tablet computers are capable of displaying realistic three dimensional content in real time and with a quality that approaches those of current generation video game consoles (Figure 1.6). Game engines such as *Unreal Engine 3* from *Epic* or *CryENGINE 3* from *Crytek* which are used for PC and console games are also available for mobile game development.

When developing games for mobile devices, the amount of texture data becomes critical: Storage, bandwidth and networking are further limited in addition to power consumption concerns [CPAM08, JGDAM12].

Existing approaches Often, to face the continuous stress textures apply on memory and artists authoring time, interactive applications had to sacrifice visual quality by reusing a

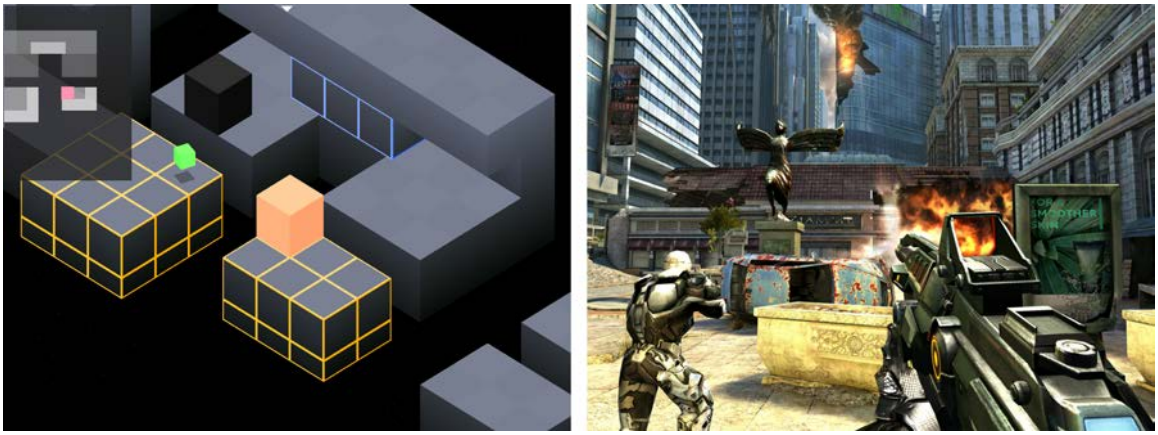

Figure 1.6: Growth of mobile video games. *Left:* Earlier mobile 3D video games used simple geometric models with few or no textures. *Source:* Edge, Mobigame. *Right:* Recent mobile video games make use of a an important amount of textures to display realistic scenes. *Source:* N.O.V.A. 3, Gameloft.

Figure 1.7: To texture a large terrain without stressing video memory, a texture is repeated over the terrain surface. However, repetition artifacts occur and this especially at distant views. *Source:* The Elder Scrolls 5: Skyrim, Bethesda Game Studios/Bethesda Softworks.

same texture on different surfaces, reducing the resolution of textures or applying aggressive lossy compression:

Reusing textures often results in a lack of variety and may cause repetition artifacts at distant views as seen in Figure 1.7.

Resolution reduction and lossy compression lead to a very blurry result and lose fine

Figure 1.8: Virtual depiction of Las Vegas in Google Earth. Due to storage and bandwidth limitations, the textures have a low resolution. *Source:* Google Earth.

scale details. For instance, in order to display its 3D content over the Internet Google Earth applies very aggressive resolution reduction and lossy compression. This results in a low quality rendering as seen in Figure 1.8.

Texture synthesis Rather than reducing the quality or repeating the texture content researchers have been looking for alternatives to reduce the size of textures and to reduce the time required to author them. One of the most promising research development is the field of *texture synthesis*.

Texture synthesis represents techniques that computerize the generation of textures. The aim of these techniques is to replace conventional textures by algorithms which are typically thousands of times smaller than the images they produce. When applicable, these algorithms produce an infinite amount of texture data and have no limit to the resolution of the produced textures.
Figure 1.9 shows a scene that exclusively uses synthesized textures. In this scene, the used algorithms and their parameters fit in few Kilobytes of memory instead of several Megabytes of conventional bitmap textures.

Since synthesized textures are represented by algorithms they can be controlled through

Figure 1.9: A scene entirely composed of synthesized textures authored with *Allegorithmic Substance Designer*.

Figure 1.10: Changing the texture parameters of Figure 1.9 changes the look of the scene.

parameters. This enables endless variety of appearances and offers a simple way to customize and make each texture unique. For instance, the parameters corresponding to the appearances of Figure 1.9 can be changed to obtain the appearances of Figure 1.10. Using classical bitmap textures to achieve such dynamic effects would require a substantial amount of memory to store all the textures and their transitions. This would also require a considerable time for the artist to author all the textures and their transitions.

With all the benefits brought by texture synthesis and with the emergence of on–line applications the industry started to get increasingly interested in texture synthesis. This

led to the development of new texture synthesis frameworks such as *Substance* from *Allegorithmic*, *Genetica* from *Spiral Graphics* or *Filter Forge*. These frameworks are based on a graph representation that let technical artists carefully design complex procedures generating large varieties of realistic textures. This type of texture is described in section 2.3.1.

1.2 Difficulties

Despite vast research efforts and the emergence of industrial products, synthesized textures are not yet widely used in modern games and interactive applications. For instance, their use is limited to few games such as *Spore* by *Maxis, Electronic Arts*, *RoboBlitz* by *Naked Sky*, *Aaaaa!* by *Dejobaan Games* or the demo–scene game *.kkrieger* by *.theprodukkt*.
We believe this is due, in part, to the two following difficulties:

- **Difficulty 1:** While solutions exist to define complex parametrized textures, it can be complex and tedious to select the parameters of the synthesized textures. Not only parameters can be non–obvious and unintuitive for the user but the proliferation and the inter–dependency between parameters also make the selection task difficult and time consuming. The space of parameters can quickly become very large and the user has to spend a significant amount of time to visualize and explore the variety of appearances produced by synthesized textures. This is especially the case when one considers multiple synthesized textures at once such as in a market–place of textures like the *TurboSquid Substance Library*.

  Overcoming this difficulty would require a more efficient way to visualize and explore synthesized textures.

- **Difficulty 2:** During display, only parts of the textures are accessed by rendering algorithms and this, in a random order. Some synthesizers, called *point–wise* evaluated synthesizers, allow the synthesis of only sub–parts of the textures in any order. However, some other synthesizers do not allow for point–wise evaluation meaning that the whole texture has to be generated at once. Not only generating all the texture is time consuming but this requires storing the whole result in memory. This becomes problematic if many textures have to be generated and stored in video memory.

  Overcoming this difficulty would require the synthesized result to be stored compactly. Furthermore, a texture should only be generated and stored in video memory if its is needed for the current displayed frame

1.3 Objective and contributions

The goal of this thesis is to promote the use of texture synthesis to enable rapid authoring of large and rich virtual environments without strong memory and bandwidth constraints. For these purposes, the aim is to simplify the use of synthesized textures, to compactly encode synthesized data in video memory and to decode the result on–demand at rendering time. Precisely, we propose to tackle the two difficulties described in the previous section through the following two main contributions:

- To overcome **Difficulty 1** and ease the visualization, exploration and parameter selection of synthesized textures we introduce two novel ideas: the *Procedural texture preview* and the *Scented sliders for procedural textures.*

  A procedural texture preview consists of a single continuous image akin to a map that summarizes in a limited pixel space the appearances produced by a given synthesis procedure. This lets end users quickly understand the content of a texture, and conveniently select appearances in the preview.

  The scented sliders augment classical sliders controlling parameters with visual previews revealing the changes that will be introduced upon manipulation. These previews are constantly refreshed to reflect changes with respect to the current settings.

  In both the procedural texture preview and the scented sliders, the main challenge is to ensure that most appearances are visible, are allotted a similar pixel area, and are ordered in a smooth manner throughout the preview. In addition, the scented sliders have to reveal as much as possible the visual variations induced by the slider.

  Procedural texture preview and scented sliders are described in chapter 3.

- To overcome **Difficulty 2** we propose a new fast by–example texture synthesis algorithm achieving high quality and tuned to take advantage of massive GPU parallelism.

  Rather than generating the result as a bitmap texture, the synthesizer can directly generate an encoded compact representation of the result. To display this compact result we propose a scheme that rapidly decodes texture data at rendering time. For maximum quality and efficiency the decoding scheme relies on an approximate filtering approach that takes advantage of the hardware to efficiently enable tri–linear and anisotropic filtering without noticeable artifacts.

  Details on how to perform fast by–example texture synthesis are described in chapter 4. The encoding and decoding schemes are described in chapter 5.

  In addition to compactly encode synthesis results, we discuss texture streaming which aims at minimizing the number of textures loaded in memory by progressively synthesizing textures during rendering. The streaming strategy must start the synthesis

process of a texture early enough so that the texture is already synthesized and loaded in memory when it becomes visible. Streaming is discussed in chapter 5.

The contributions described above appear in the following publications:

- **By-example Synthesis of Architectural Textures**
  Sylvain Lefebvre, Samuel Hornus and Anass Lasram
  ACM Transactions on Graphics (SIGGRAPH conference proceedings), 2010
  patent - FR 10/02902, 2010

- **Coherent Parallel Hashing**
  Ismael Garcia, Sylvain Lefebvre, Samuel Hornus and Anass Lasram
  ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia), 2011

- **Procedural Texture Preview**
  Anass Lasram, Sylvain Lefebvre and Cyrille Damez
  Computer Graphics Forum (Eurographics conf. proc.), 2012

- **Scented Sliders for Procedural Textures**
  Anass Lasram, Sylvain Lefebvre and Cyrille Damez
  Computer Graphics Forum (Eurographics conf. proc.) - short paper, 2012

- **Parallel Patch-based Texture Synthesis**
  Anass Lasram and Sylvain Lefebvre
  High Performance Graphics, 2012

1.4 Thesis overview

In the remainder of this thesis we start by describing in Chapter 2 the prior work that serves as the fundamental for exploring and rendering synthesized textures. Chapter 3 describes how to ease the visualization, exploration and parameter selection of procedural textures through the use of the procedural texture preview and the scented sliders. Chapter 4 introduces a fast patch–based texture synthesis algorithm which serves as a basis for exploring and rendering synthesized textures. Chapter 5 discusses techniques to encode, decode and stream synthesized textures in order to render large and rich virtual environment. Finally, we conclude in Chapter 6 by giving remarks and prospect of future research.

Chapter 2

Previous work

2.1 Introduction

In this chapter we introduce the fundamentals of texture synthesis and describe the work related to our research.

Section 2.2 introduces the fundamental notion of texture mapping. In particular, we describe bitmap textures and their use in practice, covering topics such as parametrization, filtering and texture compression. Although general, these notions are particularly related to chapter 5 where we aim to represent textures compactly and still efficiently render from them with high quality.

Section 2.3 gives an overview of texture synthesis methods. We use texture synthesis throughout all this thesis.

Section 2.4 describes previous work related to exploring and visualizing a space of textures. We describe previous work related to navigating a space of appearances as well as sampling techniques in high dimensional spaces. We further describe how to organize and visualize appearances in a global view. These notions will be used in chapter 3 where we build previews summarizing the appearances that can be generated with a synthesis algorithm.

Section 2.5 describes work on texture streaming, a technique where instead of loading all textures in memory at startup, textures are progressively loaded in memory during the user exploration. We discuss the streaming of synthesized textures in chapter 5.

2.2 Fundamentals of texture mapping

In the early days of computer graphics researchers have been particularly interested in how light interacts with surfaces. Reflectance model functions [Pho75, Bli77, CT82, War92] called bidirectional reflectance distribution functions or *BRDF* have been developed to describe light reflectance depending on surface properties. Although realistic, these models were applied *uniformly* to a surface (shift–invariant BRDF). This resulted in a lack of realism and the materials looked as if they were made of homogeneous metals or plastics.

Texture mapping [Cat74] is a powerful technique that increases realism by varying material properties over the surface without the need to alter the geometry.

In its most generic form, a texture is defined as a function $T(x, y, z)$ that returns the BRDF properties of the point that is located at (x, y, z) on the surface. BRDF properties may consist of parameters like the albedo, the curvature of the surface, the intrinsic emission or the specular intensity.

In most interactive applications the geometry is composed of thin surfaces embedded in the three dimensional space. Rather than relying on 3D functions, it seems natural to describe an embedded surface in a 2D parametric space. More specifically, we consider an embedded surface S with a two variable parametrization: An invertible function $P: \ ]0,1[^2 \rightarrow \mathbb{R}^3$ such as:

$$S = P(]0,1[^2) = \{P(u,v) \ : \ (u,v) \in]0,1[^2\}$$

The space $]0,1[^2$ of valid parameters (u, v) is often called texture space or uv–space. We will later describe how a parametrization P is constructed for arbitrary triangle meshes while some surfaces admit a natural parametrization.

Using P, the texture function $T(x, y, z)$ can be reduced to a parametric version $T(u, v)$ such as:

$$\forall (x,y,z) \in S, (u,v) = P^{-1}(x,y,z), T(u,v) = T(x,y,z)$$

The resulting 2D texture $T(u, v)$ wraps the geometric surface S in a similar way wallpapers decorate walls and furniture (Figure 2.1). Note that we use the symbol T to describe both $T(x, y, z)$ and $T(u, v)$ for clarity reasons. $T(x, y, z)$ and $T(u, v)$ differ from each other in the type of the input.

2.2.1 Bitmap textures

Bitmap textures are widely used in interactive applications. A bitmap is a discretized version of the continuous texture function T. Precisely, a 2D bitmap is a discretized version of $T(u, v)$ and a 3D bitmap is a discretized version of $T(x, y, z)$. A BRDF sample in the bitmap is referred to as a *texel*. Usually, the texels are specified manually by texture artists

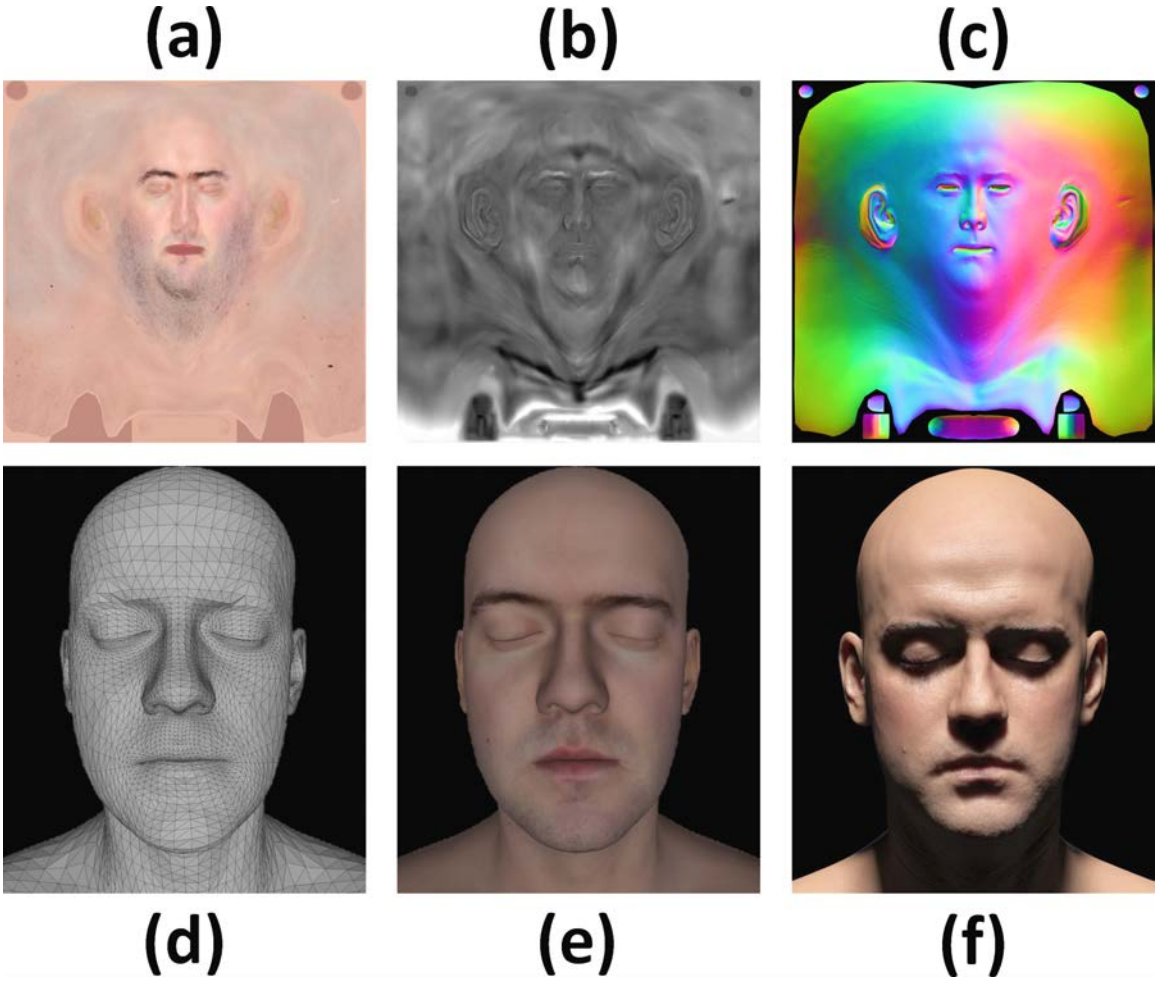

Figure 2.1: Mapping a 2D bitmap texture on a surface. The diffuse bitmap texture **(a)** is mapped on the head model **(d)** to produce the image **(e)**. Adding displacement information **(b)** and normals **(c)** then rendering with enabled light produces the image **(f)**. *Source:* 3D Head Scan by Lee Perry-Smith. *http://www.ir-ltd.net/infinite-3d-head-scan-released*

through direct painting over the surface or using an image editing tool. Texels may also be acquired from real–world objects using cameras or other special acquisition devices such as spectrometers or gonioreflectometers [MMS+05]. For simplicity, we consider in the following that a texel only stores an RGB color triple representing the albedo of the textured surface. All described methods generalize to any BRDF property.

2.2.2 2D bitmap textures

A 2D bitmap texture consists of an image I of size $W \times H$ where each pixel (or texel) stores the color of a point on the surface S. I is constructed by sampling the function $T(u, v)$ at a constant interval $\delta = (\frac{1}{W}, \frac{1}{H})$ starting from $(\frac{0.5}{W}, \frac{0.5}{H})$ to $(\frac{W-0.5}{W}, \frac{H-0.5}{H})$. We note $I[x, y]$ the stored color at pixel (x, y). The sample stored at $I[x, y]$ corresponds to $T(\frac{x+0.5}{W}, \frac{y+0.5}{H})$.

Interpolation

For a point $(x, y, z) \in S$ with parameters (u, v), we aim to access its corresponding texel $T(u, v)$ using the samples stored in I. To recover T from the samples in I a reconstruction filter is needed [Hec89]. If the signal T is band limited (it has a finite minimum and maximum frequencies making a range $[-B, B]$ called bandwidth) and if the sampling interval δ is such as $\delta < \frac{1}{2B}$ ($\frac{1}{2B}$ is known as the Nyquist limit [Nyq24]) then T can be reconstructed from I as follows:

$$T(u, v) = \sum_{(x,y)\in I} I[x, y] \times sinc(u \times W - x - 0.5) \times sinc(v \times H - y - 0.5)$$

Exactly reconstructing T is impractical. Applying the reconstruction in frequency domain is faster but is not convenient for real time texturing [Hec89] and hence, an approximation is required. The two most used approximations are *nearest point* filtering and *bi–linear* filtering.

Nearest point filtering approximates $T(u, v)$ by finding the nearest texel sample in I as follows:

$$T(u, v) \approx I[\lfloor u \times W \rfloor, \lfloor v \times H \rfloor]$$

Nearest point filtering is fast but results in jaggies and blockiness as shown in Figure 2.2.

Bi–linear filtering approximates $T(u, v)$ by interpolating the four texels in I that surround the coordinates $(u \times W, v \times H)$. If we note $x = \lfloor u \times W - 0.5 \rfloor$, $y = \lfloor v \times H - 0.5 \rfloor$, $\alpha = u \times W - 0.5 - x$ and $\beta = v \times H - 0.5 - y$ then $T(u, v)$ is approximated as follows:

$$T(u, v) \approx (1-\alpha)(1-\beta)I[x, y] + \alpha(1-\beta)I[x+1, y] + (1-\alpha)\beta I[x, y+1] + \alpha\beta I[x+1, y+1]$$

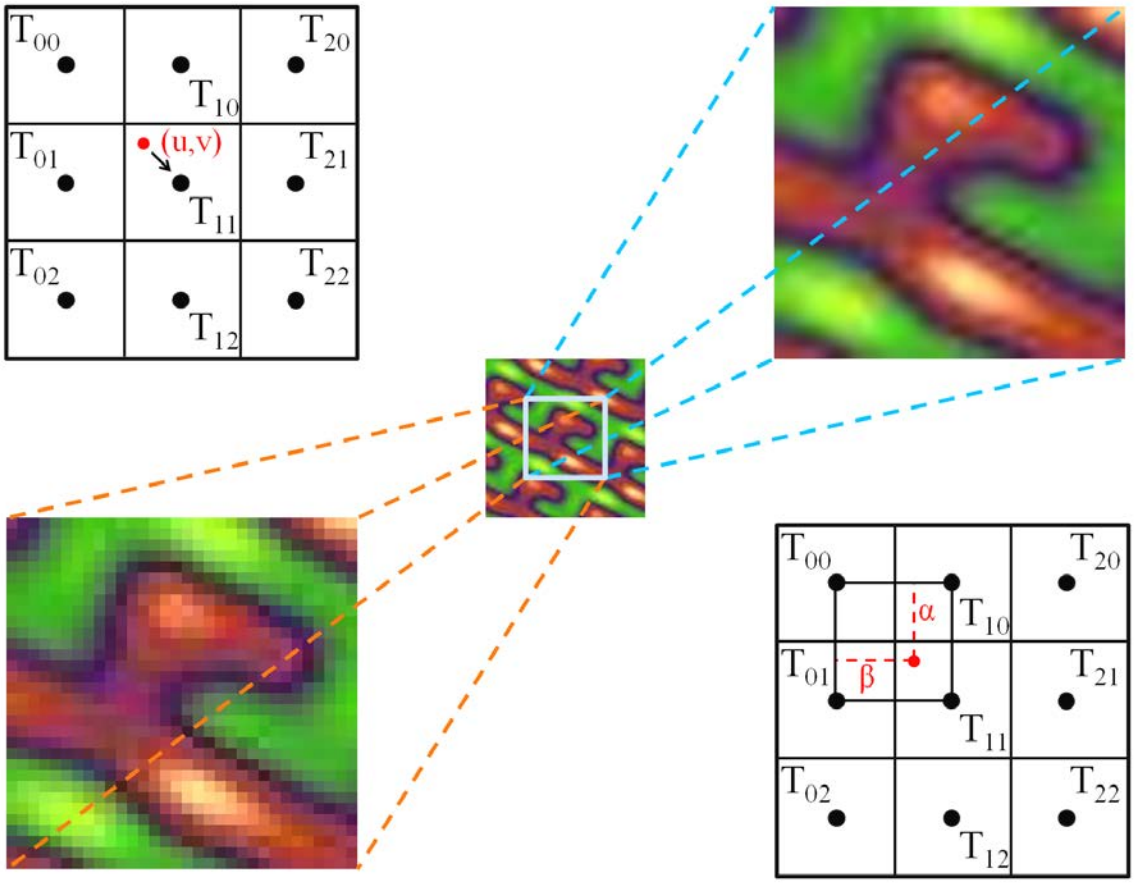

Figure 2.2: Texture interpolation in case of magnification. The centered light blue square in the middle texture is magnified and nearest point filtering is used to have the image on the bottom–left. For instance, this is done by returning for the red texel at (u, v) its nearest sample T_{11} in I. The same square is magnified and bi–linear filtering is used to have the image on the top–right. This is done by interpolating the four texels T_{00}, T_{10}, T_{01} and T_{11} that surround (u, v) in I.

Bi–linear filtering offers a good trade–off between quality and memory bandwidth. Figure 2.2 shows a reasonably good result with bi–linear filtering. Of course, a higher order filtering is possible if quality is more important than performance.

Most GPUs dedicate special units called *texture mapping units* to accelerate nearest point filtering and bi–linear filtering among other texturing operations.

Minification filtering

Figure 2.2 showed a case where a texture is magnified. This means than the sampling rate of the texture as seen on screen is lower than the sampling rate of the screen pixels. This often happens when the viewer zooms toward a textured surface.

It is also common that the sampling rate of the texture as seen on screen is higher than the sampling rate of the screen pixels. This usually happens when the textured surface is distant from the viewer and many texture samples are projected within the same screen pixel. Figure 2.3 shows a case where many texels are projected in every pixel. If the texels in each pixel are not filtered, severe *aliasing* artifacts appear.

For simplicity, we define here a screen pixel as a unit square and we aim to average all texels projected within this square (Figure 2.3). Please note that other pixel representations exist [Hec89].

Aliasing can be removed by eliminating all frequencies above the screen pixels bandwidth. This can be done through *filtering*: in our case averaging all texels projected in a same screen pixel. This can be done as follows: First, the square that corresponds to the pixel is projected in texture space (through the inverse transformation that maps the texture to the screen [Hec89]). The result is a quadrilateral portion Q called the *footprint* of the pixel in the texture (Figure 2.4). Second, all texels within the footprint Q are averaged to obtain the final pixel color.

The main issue is that summing–up all texels in Q can be very expensive. For instance, if a polygon with a 1024^2 texture is far away from the camera such as it is projected in one pixel then all 1024^2 texels need to be averaged and this would be too expensive for a real time rendering.

Heckbert [Hec89] describes many practical filtering techniques like mipmapping [Wil83], summed area tables [Cro84] or ripmapping. We limit the description to mipmapping: the most used technique in practice which is also natively supported by most graphics hardware.

Mipmapping Mipmapping consists in pre–filtering the texture recursively to form a pyramid representation (Figure 2.5). The pyramid contains a list of N images I_0, I_1, ..., I_{N-1}. The pyramid image I_i is called the i'th *level of detail* of the texture. If I is a power of two square image then the resolution of I_i is equal to 2^{N-1-i}. Using a box–filter the pyramid

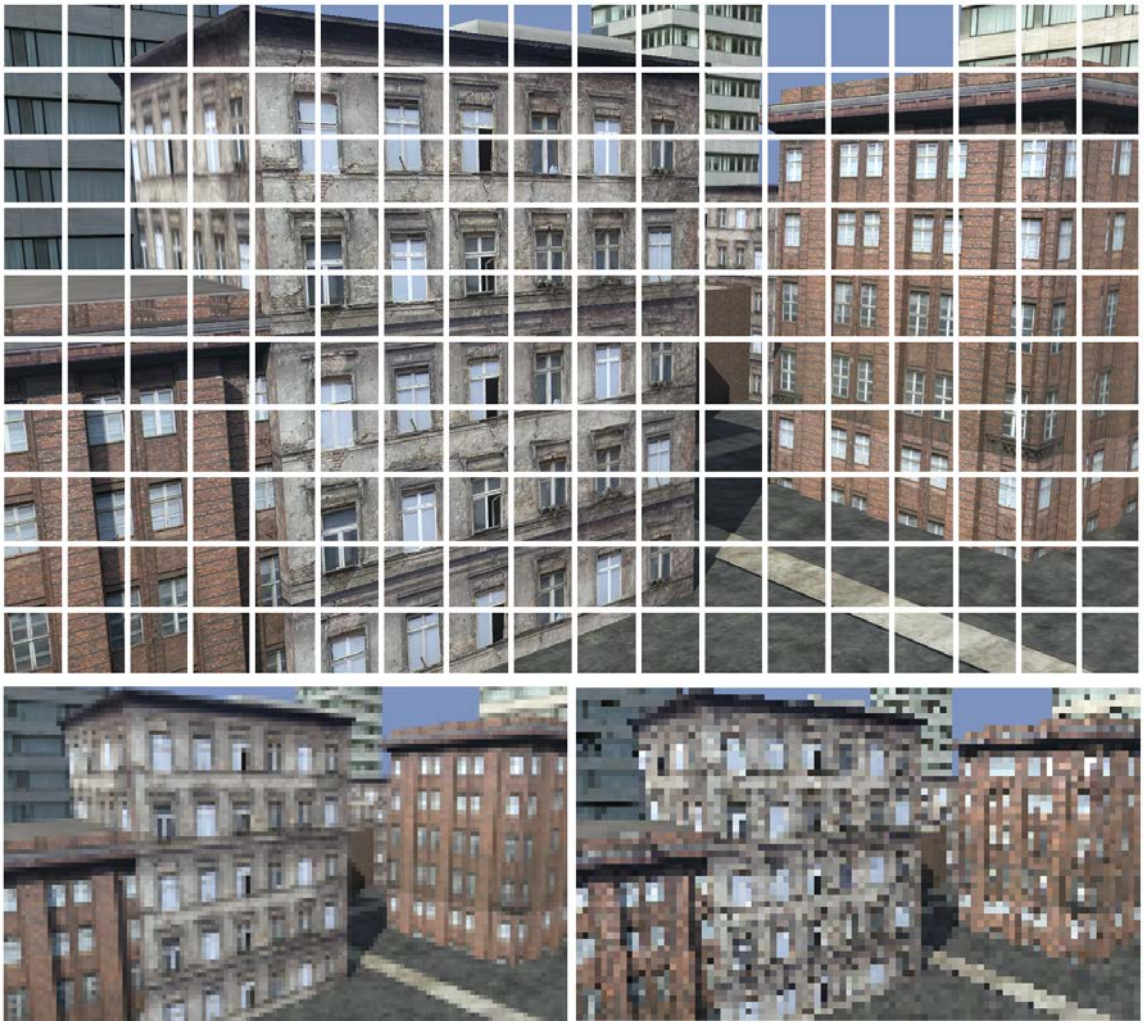

Figure 2.3: *Top:* Many texels are project in each pixel of the screen frame buffer. This frame buffer has a size of 18×10 pixels where each pixel is represented with small squares. *Bottom–left:* All texels in each pixel are averaged to produce a final filtered result. *Bottom–right:* Among all texels projected in each pixel, a single one is selected and this produces aliasing artifacts.

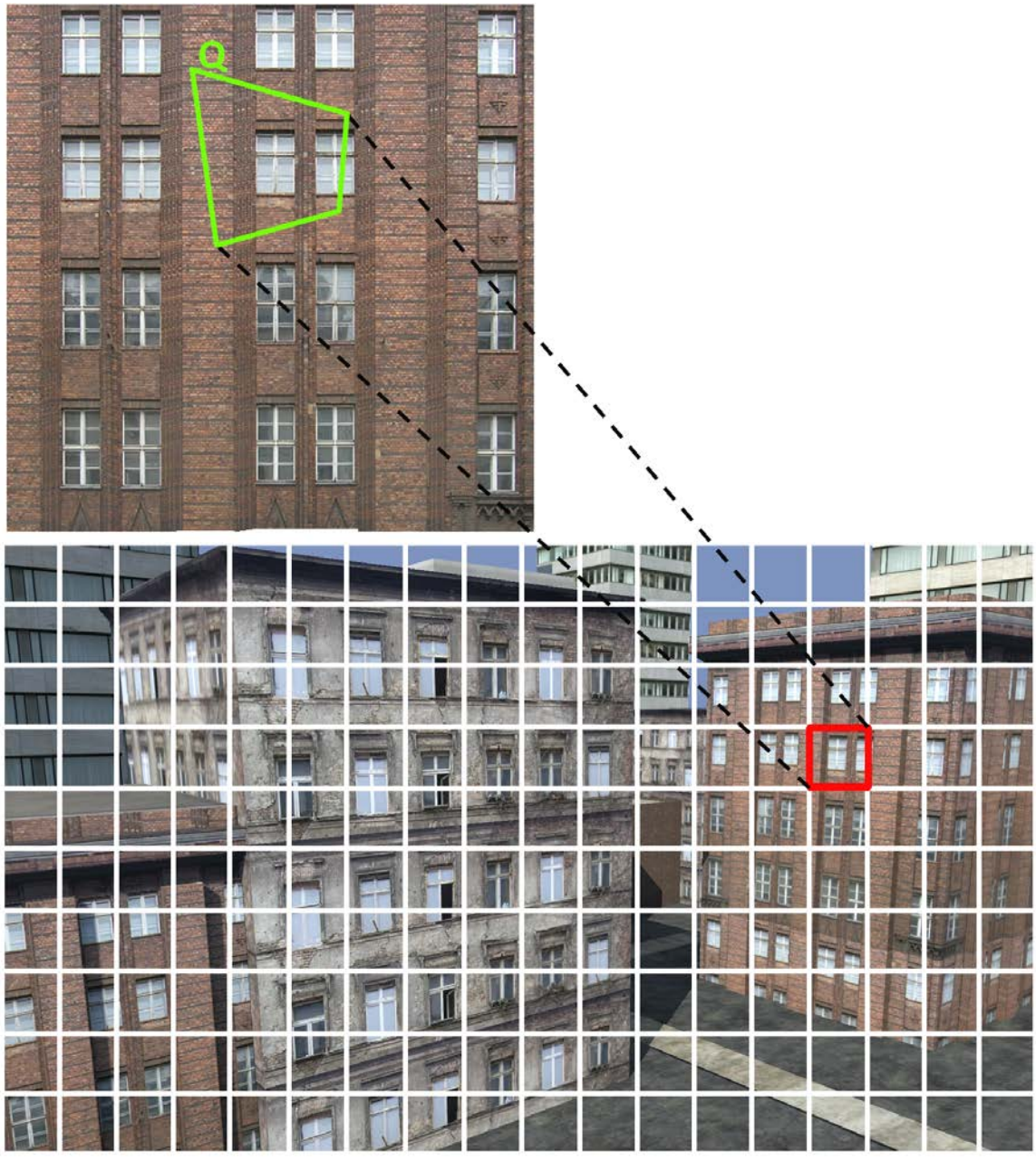

Figure 2.4: Filter footprint. the red pixel square is projected in texture space and the result is the green quadrilateral Q.

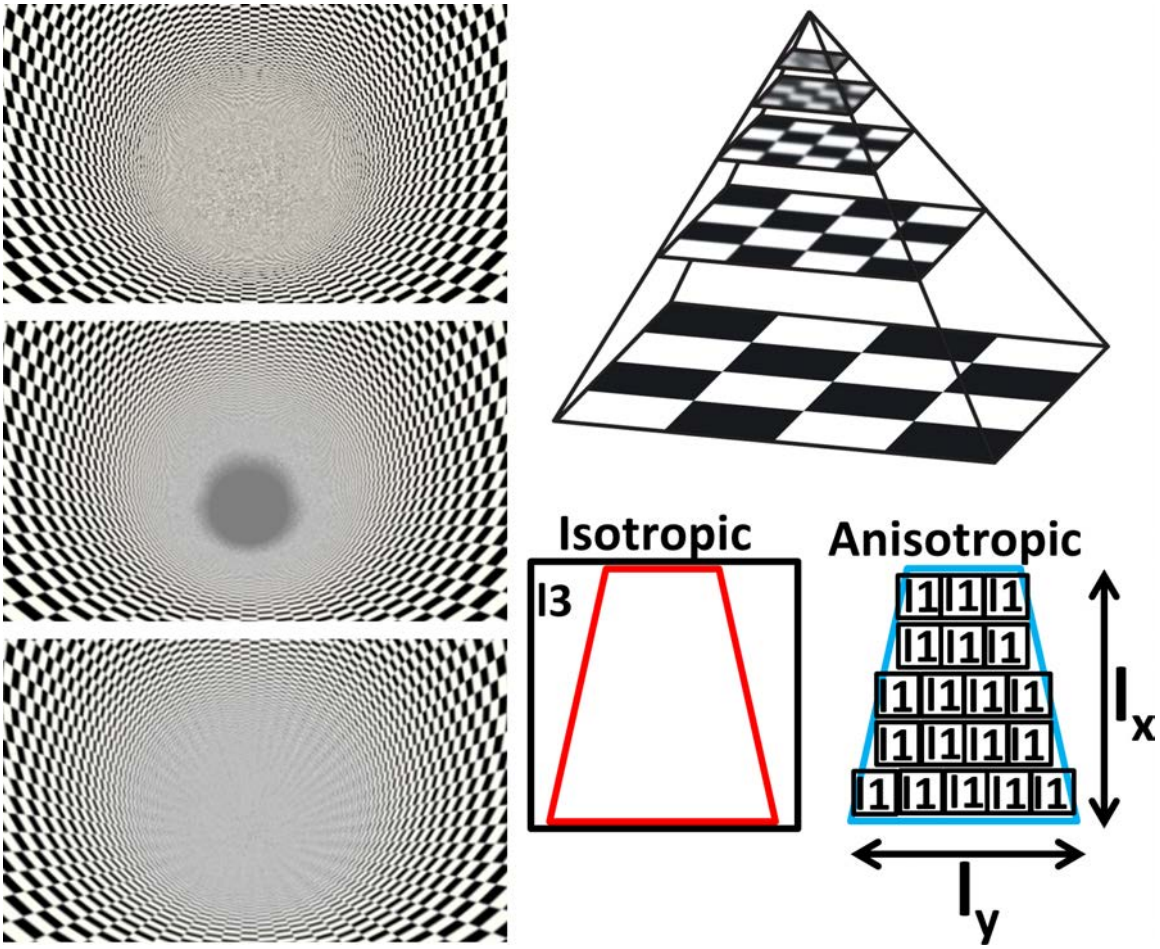

Figure 2.5: *Top–left:* Unfiltered texture. *Middle–left:* Tri–linear filtering. *Bottom–left:* Anisotropic filtering. *Top–right:* The used mipmap pyramid. *Bottom–right:* For the red trapezoidal filter footprint, isotropic filtering always selects the level of detail that corresponds to the square encompassing the footprint: level **l3** in the illustration. For the blue trapezoidal filter footprint, anisotropic filtering depends on both X and Y to determine the minimum combination of mipmap samples that best fit the footprint.

is constructed as follows: $I_0 = I$ and $I_i[x, y] = \frac{1}{4} \sum_{i,j \in \{0,1\}^2} I_{i-1}[2x + i, 2y + j]$.

At rendering time a level of detail is selected so that the sampled texel fits as much as possible the pixel filter footprint. When shading pixel (x, y) using texture coordinates (u, v), the level of detail is computed as follows:

$$l_x = \sqrt{(\frac{\partial u}{\partial x})^2 + (\frac{\partial v}{\partial x})^2}$$

$$l_y = \sqrt{(\frac{\partial u}{\partial y})^2 + (\frac{\partial v}{\partial y})^2}$$

$$l = log_2(\max(l_x, l_y))$$

The computed level of detail l is a real number while the mipmap pyramid is discrete

and follows a logarithmic to the base 2 scale.

Tri–linear filtering estimates the filtered texture sample at (u, v) in level l by linearly interpolating between texels in $I_{\lfloor l \rfloor}$ and $I_{\lceil l \rceil}$ using the interpolation parameter $\alpha = l - \lfloor l \rfloor$.

Anisotropic filtering Previously, the level of detail l has been computed as the maximum of l_x and l_y (Figure 2.5, Bottom–right). This produces good results if $l_x = l_y$ meaning that the filter footprint is a square (Isotropic). However, when l_x is very different from l_y, assuming a square filter footprint by taking the max of l_x and l_y leads to over–blurred results. This is especially the case on surfaces angled obliquely away from the viewer as shown in Figure 2.6.

Practical techniques exist [Hec89] to filter anisotropic footprints. Most GPUs however average N samples along a line in a rectangle within a finer level of detail to better approximate the footprint (Figure 2.5, Bottom–right). N is computed as follows:

$$N = \max(\lceil \frac{\max(l_x, l_y)}{\min(l_x, l_y)} \rceil, A_{max})$$

Where A_{max} is a constant set by the user to limit the maximum number of samples N. The finer level of detail is selected as follows:

$$l = log_2(\frac{\max(l_x, l_y)}{N})$$

Parametrization

So far we did not discuss how to construct a parametrization P. For many surfaces like parabolas, sphere, torus or a flat surface, P can be obtained trivially. For example, the unit cylinder can be represented with two variables u and v using the parametrization:

$$P(u, v) = (cos(2\pi u), sin(2\pi u), v)$$

The inverse of $P(u, v)$ is:

$$P^{-1}(x, y, z) = (\frac{atan2(y, x)}{2\pi}, z)$$

Catmull [Cat74] is the first to use bitmap textures. He applied textures on bi-cubic surfaces. These type of surfaces are naturally parametrized with a pair (u, v).

In practice, however, surfaces are commonly represented with triangle meshes. Finding a good parameterization P for an arbitrary mesh is usually difficult. Automatic methods exist [HLS+07] but require expensive nonlinear solvers in order to get a good parameterization. Very often, a complex and time consuming manual intervention done by a technical texture artist is required. For a triangle mesh, the result is usually stored as texture coordinates (u, v) for each vertex. Finding the texture coordinates of an arbitrary point on

Figure 2.6: *Top:* Isotropic filtering. *Bottom:* Anisotropic filtering. Notice how the bricks on the wall, the ground and the train railroad are better preserved in the bottom image. *Source: Half–Life 2, Valve Corporation.*

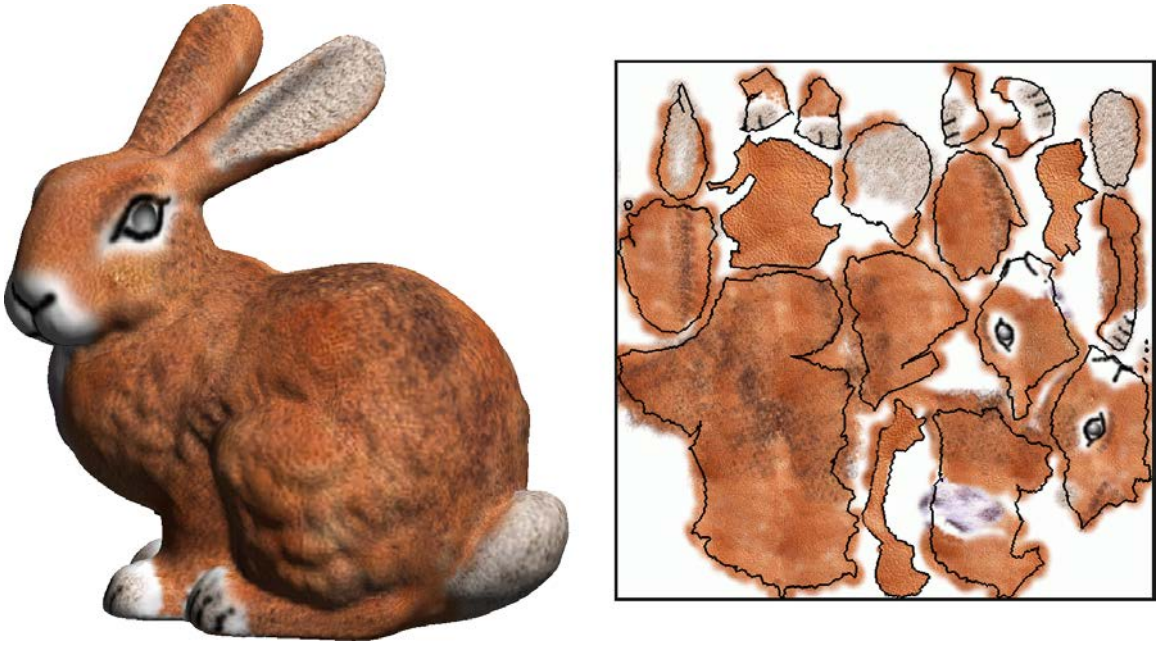

Figure 2.7: *Left:* Textured Stanford bunny model. *Right:* Texture Atlas. *Source:* Lévy et al. [LPRM02].

the mesh is done by interpolating, using barycentric coordinates, between the three stored coordinates of the triangle that contains the point.

In order to map a surface in texture space, the parameterization requires flattening the curved 3D surface. However, as the bitmap sampling rate is constant and independent from the surface curvature, the flattening may cause distortions when the bitmap is displayed along the surface.

The only way to reduce distortions is to cut the geometry into pieces called *charts* [MYV93]. The charts have to be chosen so that distortions in each one of them are reduced.

Once constructed and parametrized, the individual charts are packed together in the 2D bitmap (Figure 2.7). A 2D bitmap containing charts is called an *atlas*.

The limitation of an atlas is that the packing leads to some amount of wasted void space. Also, interpolation and filtering are no longer correct since neighboring texels on the surface may not be neighbors in the atlas. In particular, automatic hardware filtering is not directly applicable on the charts boundaries.

To reduce the interpolation problem the border of a chart is usually padded with some neighboring texels. Also a border is chosen at a location that is less likely to be visible [AS02] or at a location that coincides with a transition area from one part of the object to another. This process is often done manually by technical texture artists.

In chapter 5 we encounter a similar problem. We synthesize a new texture by combining patches from a texture sample and store the result through texture coordinate in the sample. Filtering problems therefore occur at the patch boundaries. To overcome this, we describe

an approximate filtering method that takes advantage of hardware filtering.

Limitation of bitmap textures

Apart from distortions caused by the parametrization, the main limitations of bitmap textures are mostly related to memory.

First, bitmap textures require too much memory bandwidth. Surface shading is often stalled by texture reads despite low latency video memory. This problem is expected to be increasingly evident with the continuous increase in processing speed, parallelism and power consumption.

Second and more importantly, bitmap textures require a large amount of storage. During rendering, it is common that bitmap textures do not fit in video memory or even in system memory.

Volume bitmap textures, described next, further suffer from these limitations.

2.2.3 Volume bitmap textures

Volume bitmap textures (or *solid* textures) are useful to represent volumetric texture data. The bitmap is a 3D image I consisting of a regular discretization of the function $T(x, y, z)$. Each texel $I[x, y, z]$ is called a *voxel*.

One advantage of a volumetric texture is that it usually has a natural parametrization: the identity. Thanks to this, volume bitmap textures are sometimes used to texture complex surfaces. In this case however, small number of voxels is actually used to texture the surface. Also, interpolation and filtering are more difficult: Mixing empty and non–empty cells or mixing disconnected regions (see Figure 2.8) must be avoided otherwise, color bleeding artifacts may occur though the object interior.

Interpolation and filtering can be improved by avoiding the mixing of texels where it should not happen. This can be achieved by separating conflicting texels in different *octrees* (see next section) or adding geometry information within the texels [BD02].

Efficient storage of texels along surfaces

A 3D bitmap texture applied to surfaces is mostly composed of empty cells. This results in an important waste of memory. For instance, a complex surface would require a 3D texture as large as 1024^3 to achieve good quality and little aliasing artifacts. A single 32bits RGBA color texture of size 1024^3 requires 4 gigabytes of memory. This is simply impractical on most current hardware.

To cope with memory limitations, axis aligned sparse voxel octrees [DGPR02, BD02] partition the space to form an irregular grid as shown in Figure 2.9. This way an axis aligned connected empty region will be merged in a single cell and this dramatically reduces storage.

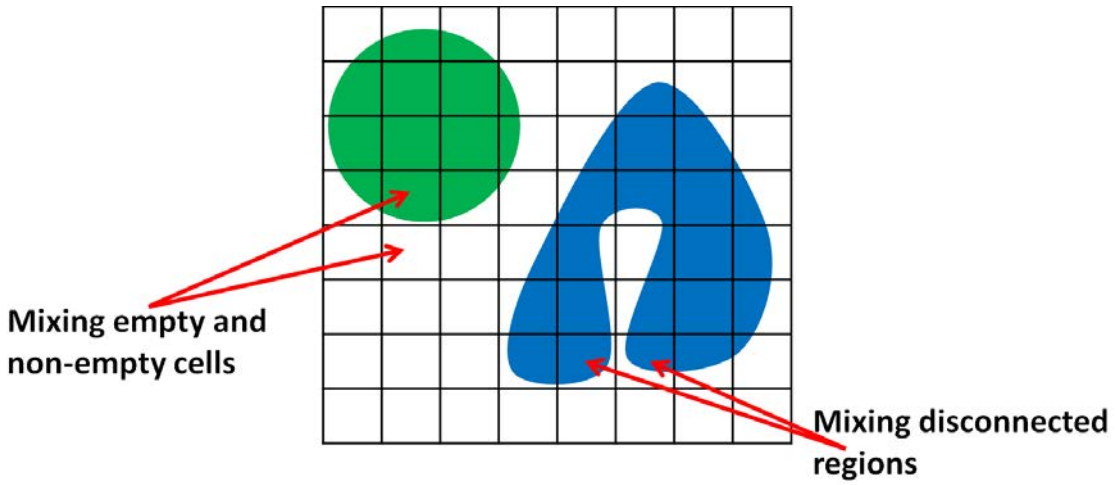

Figure 2.8: A correct interpolation between neighboring texels in a 3D bitmap texture must avoid mixing empty and non–empty cells. Also, it must avoid mixing disconnected regions.

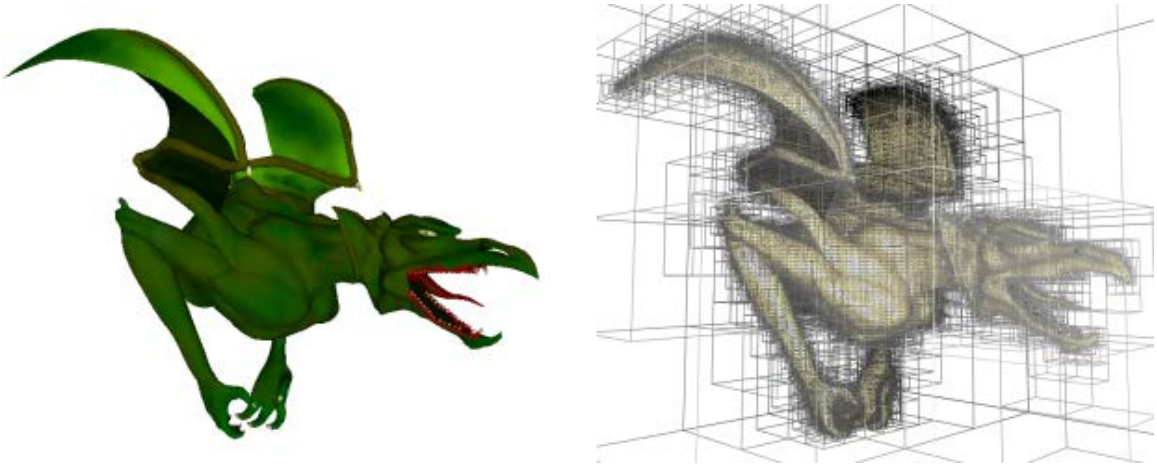

Figure 2.9: Octree texture. *Source:* Lefebvre et al. [LHN05].

The limitation of a sparse voxel octree is that rather than a constant time access to a texel, the octree has to be traversed in an $O(log(N))$ time where N is the number of texels in I.

For very large octree structures, out of core techniques are often used to progressively load volumetric data [CB04, CNLE09].

Hashing structures represent an alternative to space partitioning structures. In particular, *spatial hashing* schemes are intended for graphics applications to store sparse data such as 3D textures. Perfect spatial hashing [LH06b] is a static hash construction that resolves all collisions then stores the resulting hash table in video memory prior to rendering. During rendering, data are accessed in constant time. Alcantara et al. [ASA+09, Alc12] propose a fast hash construction on the GPU based on Cuckoo hashing. Their fast construction enables hashing of dynamic data. However, in order to successfully construct the hash table the method needs to allocate a table about 20% to 30% larger than the data that need to

be hashed and more fetches are required to access the data. In section 5.2.3 we will use a similar dynamic hashing technique that offers fast access and has a minimal waste of space.

2.2.4 Texture compression

Either 2D or 3D bitmaps, textures often represent the most memory expensive data in graphics application. Compression is the most common approach that is used to alleviate memory requirements. There are, however, specific requirements when compressing texture data. We have to distinguish between compression *prior* to rendering, for instance, for storage on disk and, compression *during* rendering where compressed data are directly used for display.

Compression prior to rendering Image compression [Wal91, TMR02] methods are commonly used to compactly store texture images prior to rendering. The images are usually stored on a disk device or on an external machine in a compressed format such as PNG or JPEG. The compressed images are then loaded from disk or through a network, decompressed in main memory and finally compressed in video memory in a compressed texture format (described in the next paragraph). These operations are done before rendering, for instance, during the loading of a game level, or streamed on the fly during the game [VW06, VW08] (streaming is detailed in section 2.5).

Most of these methods rely on frequency analysis on the images [VW06]. For instance, JPEG relies on a *discrete cosine transform* DTC [Wal91, VW06] and JPEG2000 relies on *wavelet transform* [TMR02, VW06]. After the frequency analysis, the compression relies on global entropy encoding schemes like Huffman coding [Huf52] or arithmetic coding [WNC87]. These methods considerably reduce storage if it is acceptable to allow some loss of details (lossy compression [VW06]).

The decompression of the images is fast enough for streaming operations but not suitable for direct rendering of textures [VW06, VW08]: First, the DTC or wavelet transforms are retentively expensive for real time rendering. Second, the entropy encoding schemes are sequential and are not suited to access texels randomly during rendering as required by rendering engines.

Compression during rendering Texture compression aims to reduce storage and bandwidth and still ensures random access to texels. The decoding must be fast, should result in little loss of quality and should use relatively simple algorithms that can be implemented in hardware, thus, frequency analysis schemes and entropy encoders are excluded.

The most successful texture compression methods subdivide the texture into small blocks that can be processed independently. Block Truncation Coding [DM79] and Color

Cell Compression [CDF+86] are two early examples of such block–based compression methods. Later, these two methods have been adapted to make texture compression schemes that are widely used in the industry. This includes S3TC [INH99], ETC [SAM05], BPTC [ARB] or ASTC [NLP+12]. Other texture compression techniques use vector quantization [BAC96], variable–rate coding [IM06] or the modulation of low resolution textures [Fen03].

Texture compression considerably reduces storage and memory bandwidth requirements. However, it only pushes the problem away: Texture compression allows storing four to eight times more textures. This is very significant in practice but due to the growing needs for highly detailed textures, memory remains an important bottleneck in most interactive graphics applications.

We next describe techniques to even further reduce the memory required to describe texture data.

2.3 Texture synthesis

In the previous section, we have seen that bitmap textures present a number of challenges: authoring time, storage and transfer. Texture synthesis aims to avoid these drawbacks by generating a texture automatically using an algorithm. The synthesis algorithm can generate a texture of any size without any periodicity. This saves authoring time and potentially reduces storage because textures can be computed instead of stored.

Texture synthesis is an active area of research and two main classes of synthesis approaches have been developed. The first class of approaches, known as *procedural* approaches, completely define a texture with an algorithm and a set of few parameters. Procedural algorithms are discussed in section 2.3.1.

The second class of approaches, called *data–driven* approaches, synthesize an output texture by copying pixels directly from an input texture sample, called an *exemplar*. Since a data–driven synthesis algorithm requires an input exemplar, it is also a *by–example* synthesis algorithm. The exemplar is always needed by the data–driven algorithm to guide the synthesis process. An illustration of a data–driven synthesis result is shown in Figure 2.10, Bottom. We discuss data–driven synthesis approaches in section 2.3.2.

Note that a data–driven synthesis algorithm is always a by–example synthesis algorithm. A procedural synthesis algorithm may also be a by–example one if its parameters are determined from an input image. This consists in two steps: First, the exemplar is analyzed to extract a statistical parametric description of the texture. Second, the extracted parameters are used as input to a procedural synthesizer which goal is to synthesize a texture similar to the exemplar. The two steps are shown in Figure 2.10, Top. Contrary to data–driven approaches, the exemplar is used to extract parameter and is not used during the synthesis process.

Another important property we consider in this thesis is the *granularity* of the output: If the output texel can be generated independently from all other texels, the algorithm is said to support *point–wise* evaluation otherwise, the whole texture must be synthesized in order to obtain a sub–region from it. This means that either the output must be permanently stored in video memory to access sub–regions in it or the output must be completely re–synthesized every time sub–regions need to be accessed. We discuss the granularity of the synthesis output in section 2.3.3.

Note that often in the literature, for an algorithm to be called procedural, it has to be non–data–driven and has to allow for point–wise evaluation. Our definition of procedural differs in that the algorithm does not have to necessarily support point–wise evaluation to be considered procedural.

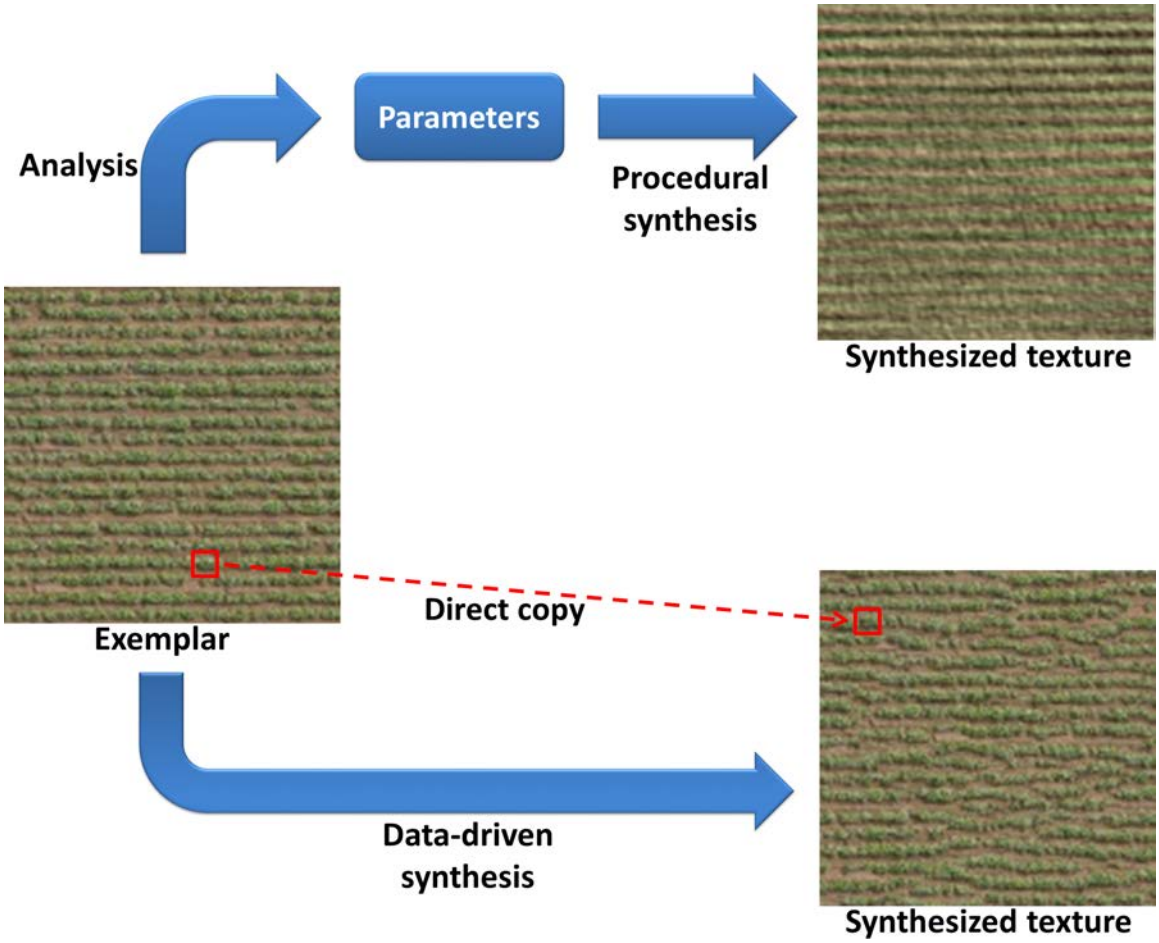

Figure 2.10: *Top:* A by–example procedural synthesis algorithm first extracts from the exemplar a parametric representation describing the appearance of the exemplar. Second, a procedural algorithm uses the extracted parameters to synthesize a texture similar to the exemplar. The result is generated using the by–example procedural synthesis scheme of Galerne et al. [GLLD12]. *Bottom:* A data–driven synthesis algorithm synthesizes a texture that is similar to the exemplar by copying pixels directly from the exemplar. The result is generated using the by–example synthesis scheme of Lefebvre and Hoppe [LH05].

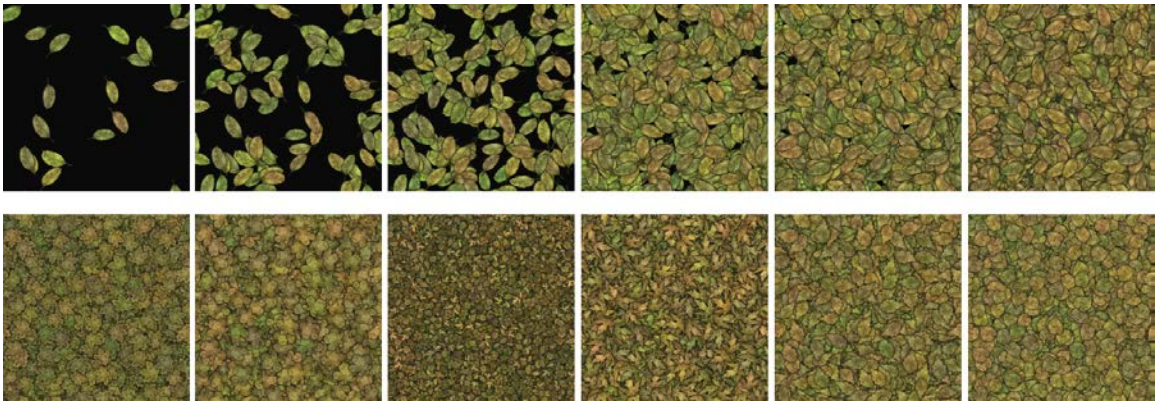

Figure 2.11: Different settings of the leaves *Substance* texture. *Top:* A parameter varies the density of the leaves. When the parameter is increased the density of the leaves increases. *Bottom:* A parameter varies the type of leaves.

2.3.1 Procedural approaches

As described above procedural textures are completely defined by the algorithm generating them in addition to a *small* set of parameters controlling them. By small, we imply that the number of parameters is much smaller than the description of the algorithm. For example, Figure 2.11 shows a procedural leaves texture with a real number parameter controlling the density of the leaves and an integer integer parameter that gives 7 different type of leaves.

In this section we discuss four categories of procedural textures: noise–based approaches, graph–based approaches, simulation–based approaches and by–example procedural approaches.

Noise–based approaches

Many textures in nature exhibit irregular patterns that lack structure and appear to be random. Noise–based approaches aim to model these phenomena. Such approaches are often used by artists in practice to model complex visual details.

A texture modeled with a noise–based approach consists in a noise pattern generated by a noise function $n : \mathbb{R}^3 \to [0, 1]$. For each point (x, y, z) in space, a noise value $n(x, y, z)$ is computed. It is then used as input to a color function $C : [0, 1] \to [0, 1]^3$ which gives the final RGB texture color at (x, y, z). The resulting texture color c at position (x, y, z) is therefore computed as $c = C(n(x, y, z))$. Figure 2.12 shows a sphere textured with a noise pattern and a color function.

In order for artists to achieve a desired appearance the noise function has to be controllable. In his pioneering work, Perlin [Per85] allows spectral control over the noise pattern through a weighted sum of band–limited noise images. Each band–limited noise image, which gives a grainy aspect of different scales to the noise pattern, is called an *octave*. The

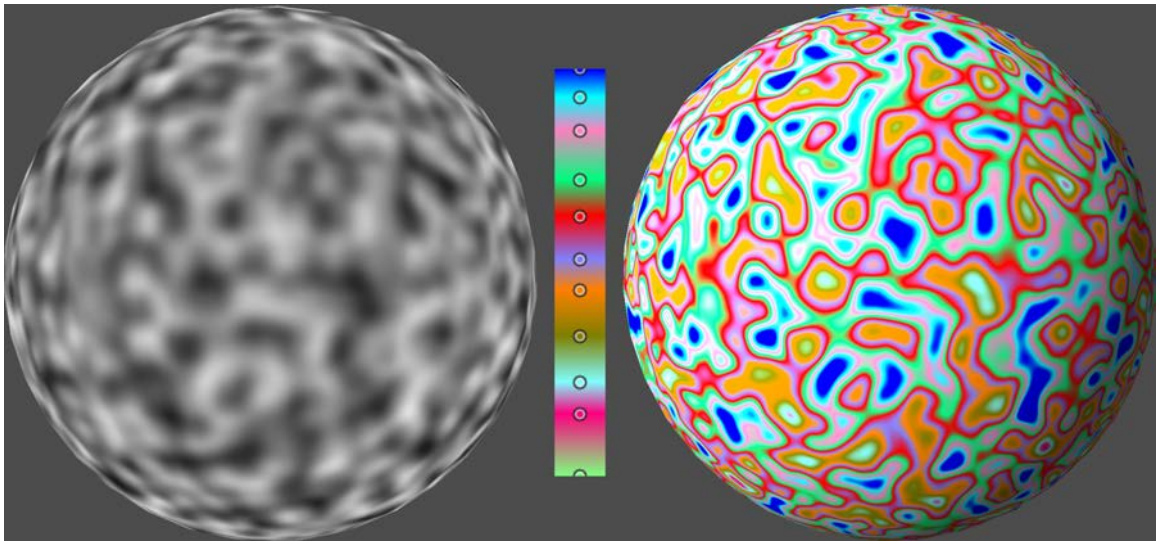

Figure 2.12: *Left:* One octave Perlin noise applied on a sphere. *Middle:* A color map. *Right:* The noise and the look–up color map produce a colored sphere.

sum of weighted octaves $N(x, y, z)$ is computed as follows:

$$N(x, y, z) = \sum_i \omega_i \times n(x2^i, y2^i, z2^i)$$

Where ω_i is the weight of the i-th octave. Artists can control the grains of each noise octave by manually specifying its weight. Figure 2.13 shows a sphere textured with a sum of weighted noise octaves.

Perlin also presented a variant of the noise function called turbulence and computed as follows:

$$N_T(x, y, z) = \sum_i 2\omega_i \times |n(x2^i, y2^i, z2^i) - 0.5|$$

Perlin integrated his noise function in different mathematical expressions to synthesize varieties of artistic and realistic materials like marble, wood, stones, grass and paintings. For example, Figure 2.14 shows a texture where the color c at point (x, y, z) is computed using the following expression:

$$c = C(sin(x + N_T(x, y, z)) \times 0.5 + 0.5)$$

Note that since its introduction, the speed and quality of Perlin noise have been improved [Per02, OHH+02].

We have seen in section 2.2.2 that during rendering, texture frequencies above the screen sampling bandwidth should be discarded to ensure an aliasing free result. For Perlin noise to be correctly filtered, this means that in addition to the manual control of the weights, an automatic control must set the weights to zero for all octaves corresponding to

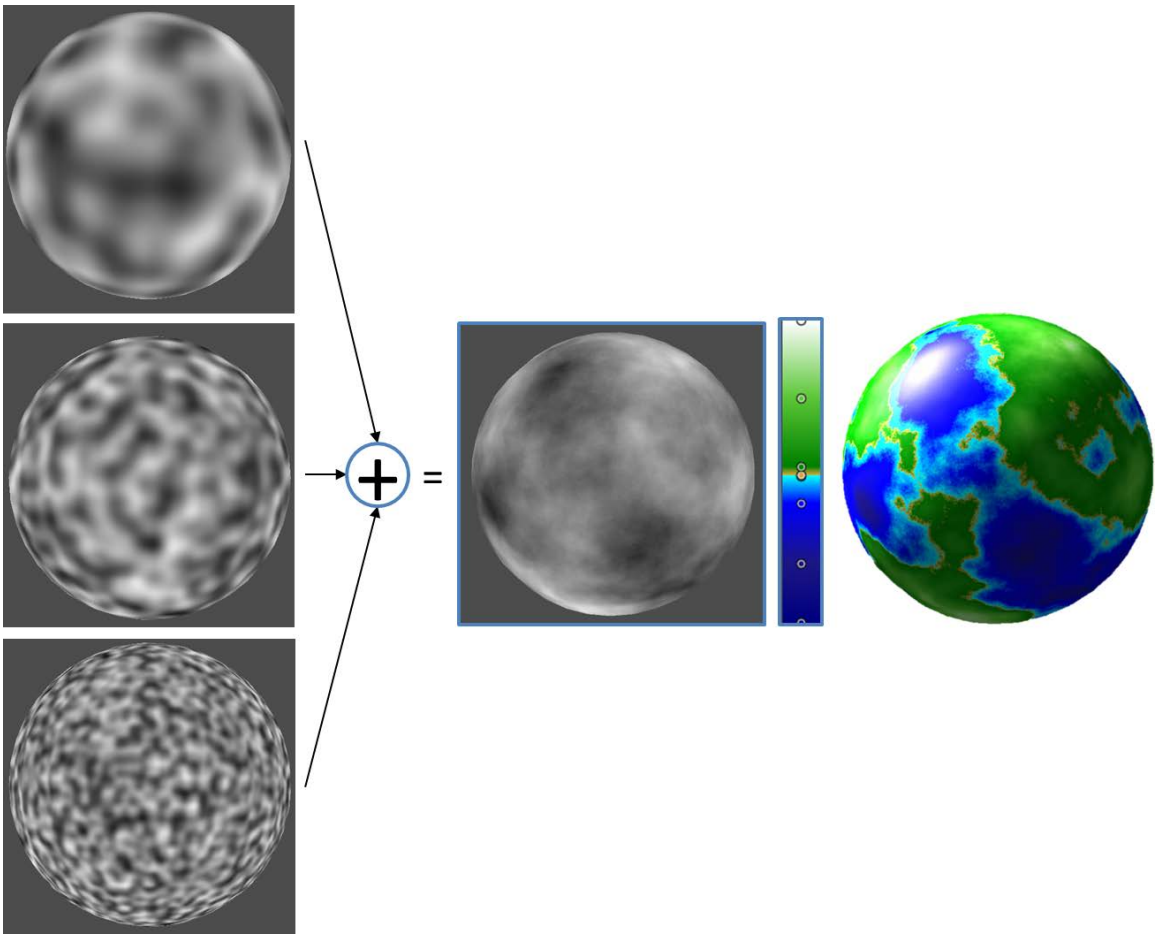

Figure 2.13: Multiple Perlin noise octaves are summed–up to produce a complex pattern.

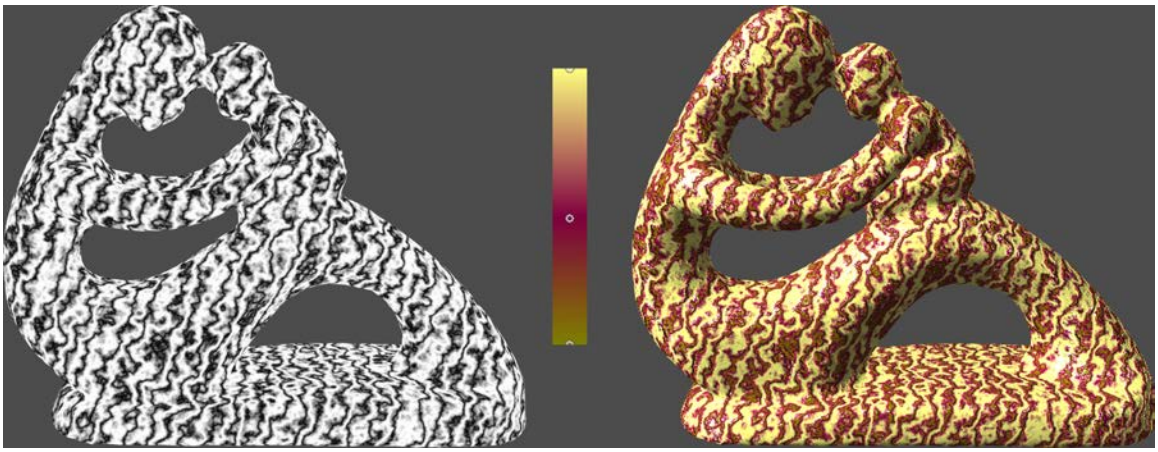

Figure 2.14: A procedural mathematical expression using Perlin noise.

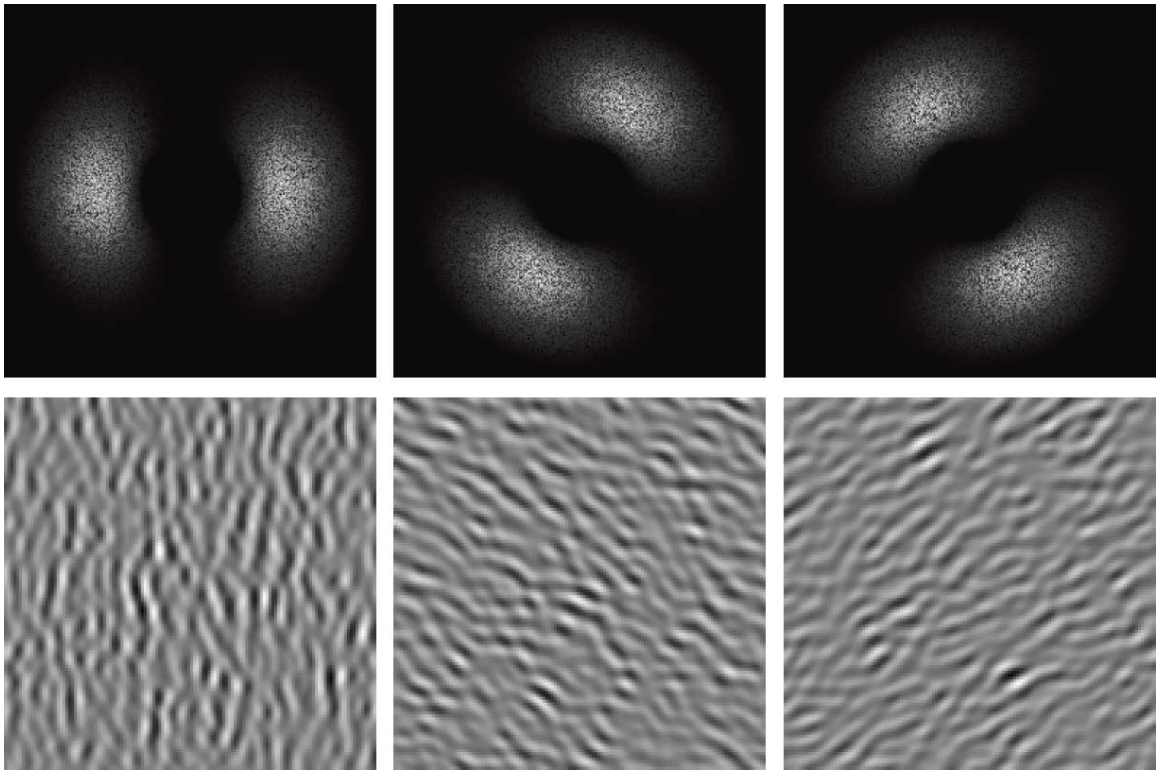

Figure 2.15: Anisotropic noise. *Top:* Three power spectrum corresponding to different oriented sub–bands of noise. *Bottom:* The three noise patterns corresponding to the power spectrum above.

frequencies above the screen sampling bandwidth. However, Lewis [Lew89] then Cook and DeRose [CD05] showed that Perlin noise is only approximately band limited meaning that a same band can have frequencies below and above the bandwidth of the screen. If such a band is not discarded the high frequencies will cause aliasing. Conversely, if the band is discarded, visible details will be lost and the filtered result will look over–blurred.

To have better control over the octaves, Cook and DeRose [CD05] represents the octaves as a Laplacian pyramid [BA83]. They use wavelet analysis for the filtering of the down–sampled and up–sampled octaves.

Goldberg et al. [GZD08] give further spectral control over the octaves by decomposing an octave band into oriented sub–bands. The artist can give weights to the different orientations to control the anisotropy of the noise pattern in addition to its grain. Figure 2.15 shows examples of anisotropic noise patterns. Note that during rendering, the method supports anisotropic filtering: Rather than discarding a whole band, only oriented sub–bands with frequencies above the screen bandwidth are discarded. Lagae et al. [LLDD09, LD11] rely on sparse convolution of Gabor noise kernels [Gab46] to allow further spectral control over isotropic and anisotropic noise patterns with accurate isotropic and anisotropic filtering.

Other techniques rely on a random distribution of points. To generate procedural textures, the distribution often follows a blue noise distribution [KCODL06, Ost07] which is convenient to associate primitives with the random points without overlapping them. For instance, texture bombing [SA79], pattern based procedural textures [LN03] or procedural generation of rock piles [PGGM09b] associate primitives with the randomly distributed points. Worley [Wor96] generates cellular patterns by constructing a basis functions based on the the distance between a point in space and the set of randomly distributed points. Zhou and al. [ZHWW12] recently proposed a random distribution generation process allowing for an accurate spectral control.

A random distribution of points can also serve to generate continuous noise patterns. Lewis [Lew89] proposed sparse convolution noise. He associates a kernel to every randomly distributed point. A weighted sum of the kernels gives a noise value $n(x, y, z)$ as follows:

$$n(x, y, z) = \sum_i \omega_i \times K(x - x_i, y - y_i, z - z_i)$$

Where K is the kernel function, ω_i is the weight of the i-th kernel and (x_i, y_i, z_i) is the position of the i-th kernel. For the kernel function K, Lewis uses the radially symmetric smooth cosine kernel [Lew89]. Lagae et al. [LLDD09] rely on a Gabor kernel and this allows a very accurate spectral control over the noise pattern as previously mentioned.

For a detailed survey on noise based procedural approaches, the reader is referred to Lagae et al. [LLC+10].

Graph–based procedural approaches

We have seen that noise based approaches allow artists to control the final appearance of the resulting texture. The offered control however is a spectral control. This means that in order to use noise based approaches, artists require technical skills, for instance, an understanding of Fourier analysis and shader programming skills to make complex procedural expressions.

To make procedural texture creation accessible to non–technical artists, Sims [Sim91] introduced genetic textures. An iterative texture simulation process inspired by evolutionary techniques. The artist is presented with a list of textures generated with random procedural math expressions. From this list, the artist selects one or many textures. The expressions of the selected textures are then mixed together and randomly altered to produce a new list of textures. The selection process is repeated until the artist reaches the desired texture.

To give more flexibility to artists to express themselves, procedural texture authoring frameworks such as *Substance* from *Allegorithmic*, *Genetica* from *Spiral Graphics*, *DarkTree* from *Darkling Simulations* or *Filter Forge* use a texture graph editor allowing artists

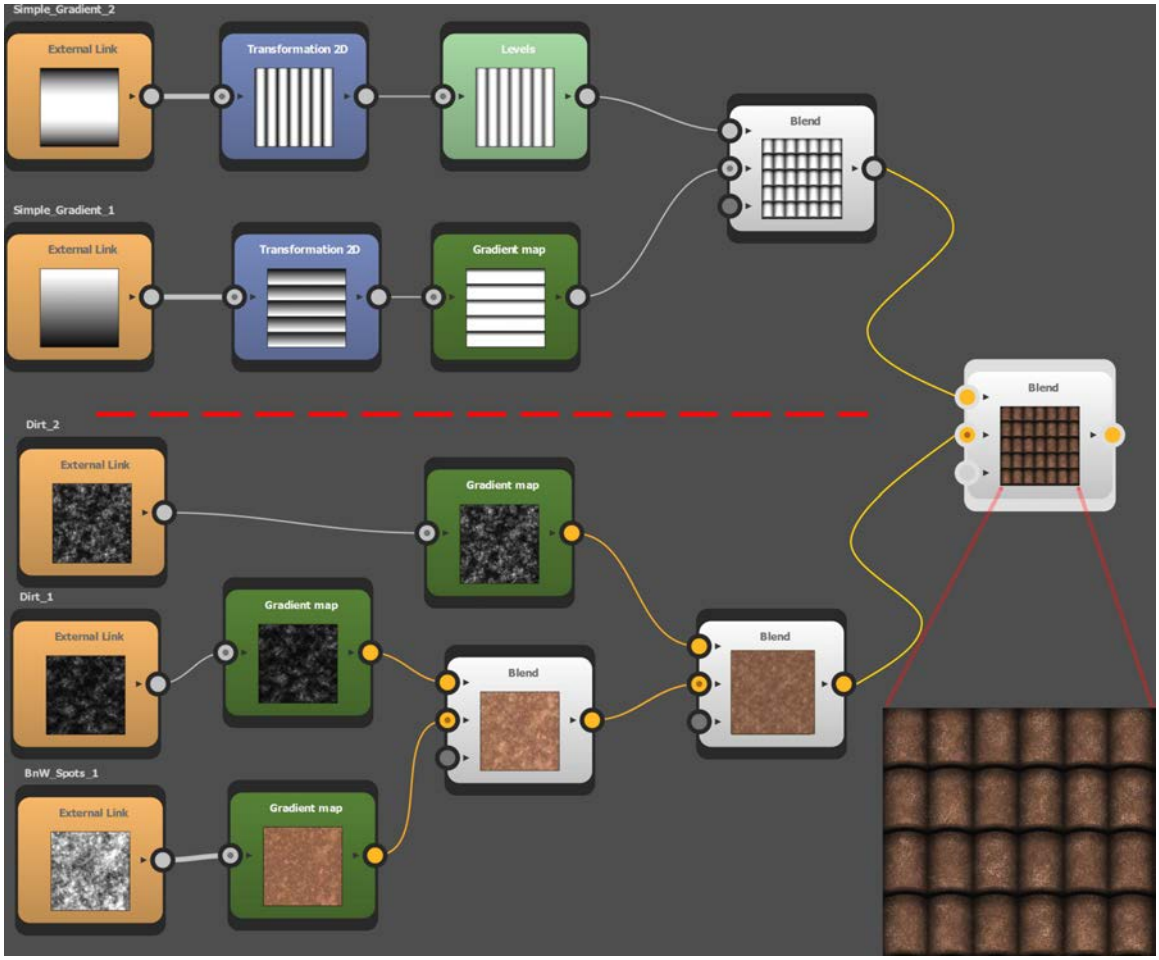

Figure 2.16: Simple Allegorithmic Substance Designer graph. *Top:* The sub–graph above the red dashed line is responsible for the texture structure. *Bottom:* The sub–graph below the red dashed line is responsible for the texture material. *Bottom–right:* A zoom on the final result.

to author texture materials by simply connecting operators in a graph representing the procedural expression. In such a graph, each and every node produces a texture appearance. The result of the node can be used as input for other nodes. A node can take multiple inputs, for example, it may take two appearances as input and produce a blending of the appearances as output.

Figure 2.16 shows a ceramic roof tiles texture specified by a simple graph in *Substance Designer*. In this figure, the sub–graph above the red dashed line is responsible for the texture structure. The structure is built starting from two simple patterns that are rotated then merged together. The sub–graph below the red dashed line is responsible for the ceramic material. The material is built by combining noise patterns and a color map.

With this graph concept, artists can author textures without deep knowledge about complex technical concepts like frequency analysis. In addition, they can naturally specify a texture in a layer by layer fashion or mix different layers to achieve very complex effects

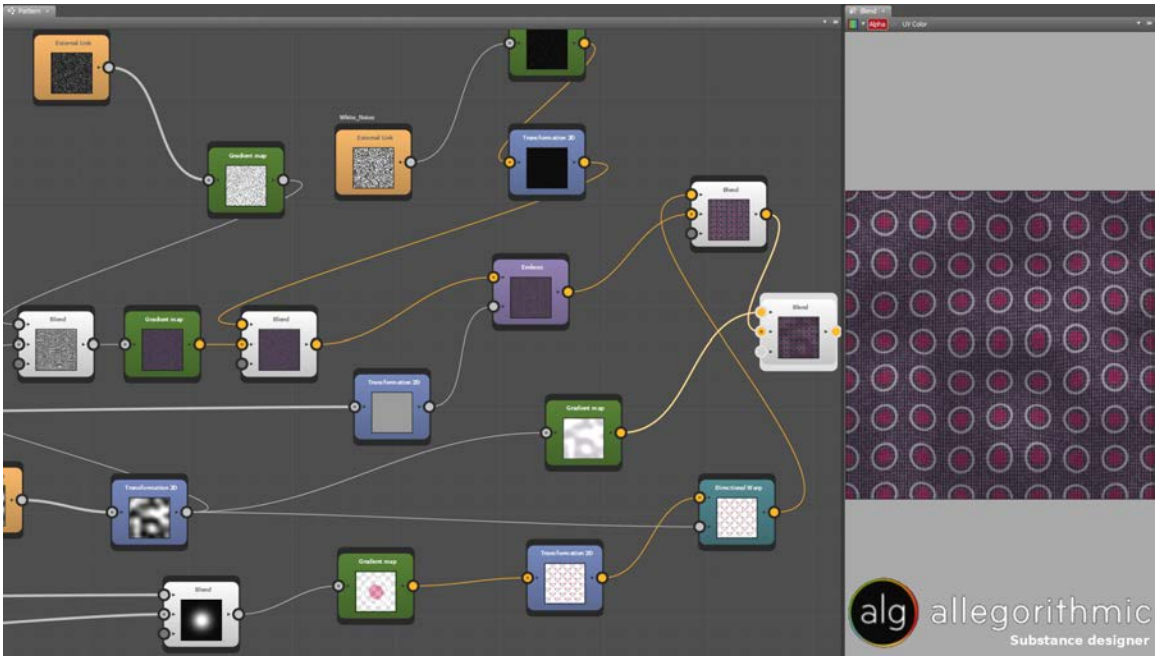

Figure 2.17: The artist intuitively assembles node in the graph to produce a complex texture.

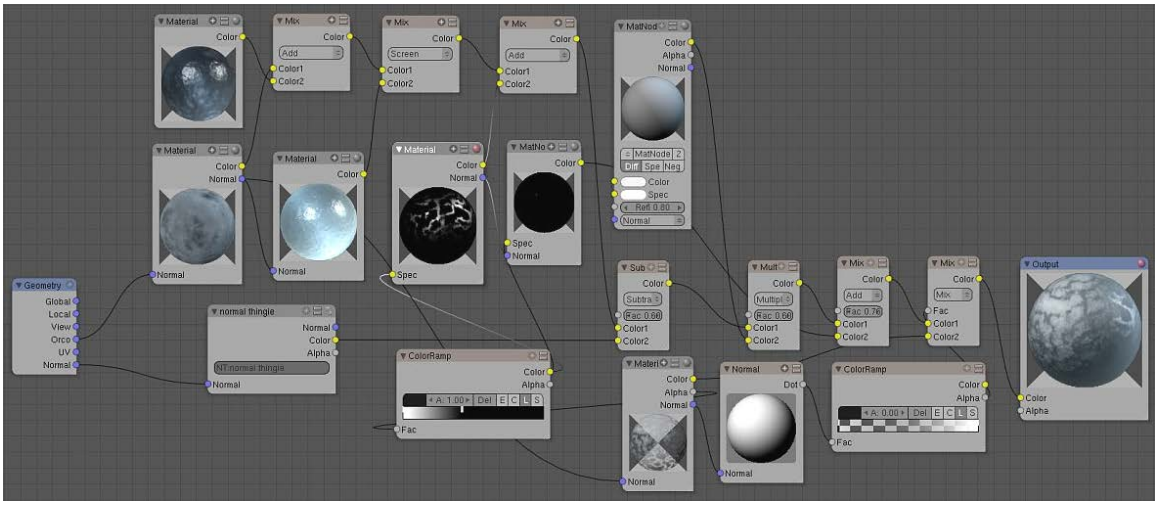

Figure 2.18: A shader graph editor used in blender for material specification. *Source:* orange.blender.org/blog/noodles-and-cinepainting

and patterns. An example of a complex texture composed in Substance Designer is shown in Figure 2.17.

Note that many 3D computer graphics software such as *Blender* or *Autodesk Maya* have a shader graph editor similar to the procedural texture graph editor. A shader graph can be seen in Figure 2.18. A shader graph is useful for general shading and material specification. However, its use for authoring textures remains limited compared to Substance Designer or Filter Forge which specifically target procedural texturing tasks.

The work in this thesis is in collaboration with *Allegorithmic* who develops the procedural texture framework Substance Designer. Textures authored with Substance Designer are called *Substances*. A substance takes a small number of parameters and generates a texture in few milliseconds. Figure 2.11, Figure 2.16 and Figure 2.17 show examples of substances. Most results in this thesis use substances. Our work however, is independent from substances and any parametric texture representation should be supported.

Simulation–based approaches

The process of procedurally synthesizing a texture can also be done using physically based simulations to achieve realistic weathering effects: Dorsey et al. [DPH96] simulate rainwater flowing over a surface; Dorsey and Hanrahan [DH96] simulate metallic patina; Paquette et al. [PPD02] simulate paint cracking and peeling; Merillou et al. [MDG01] simulate corrosion; Bosch et al. [BPMG04] simulate scratches; Desbenoit et al. [BDA04] simulate lichen growth.

Realistic weathering effects can be simulated in a non–physical ways, for instance, using a general framework such as γ–ton tracing [CXW$^+$05] or targeting a specific effects like tree barks [LN02].

Turk [Tur91] and Witkin and Kass [WK91] propose reaction–diffusion textures, a biologically motivated method that synthesizes textures by forming complex patterns resembling patterns found in nature.

Generally, simulation–based approaches give limited controls over the result while artists usually prefer interactive controllable techniques. Because of this, simulation–based approaches are usually used in post–production to add complex effects to the final scene.

By–example procedural approaches

Recall that a by–example procedural algorithm is a procedural algorithm for which the input parameters are automatically determined from an exemplar. For instance, we have seen that a Perlin noise is a weighted sum of octaves and that the weights of the different octaves represent the parameters of the Perlin noise. A by–example Perlin noise therefore consists in automatically finding the weights of the octaves so that the resulting noise texture is as similar as possible to the input exemplar.

Ghazanfarpour and Dischler [GD95] use Fourier analysis to determine from the exemplar a set of sine waves and a set of filters. The sine waves correspond to the high amplitude sine waves of the exemplar and give a crude approximation of the exemplar. The filters are applied to a white noise image so that the power spectrum of the result approximates the power spectrum of the exemplar. The sum of the learned sin waves modulated with a filtered white noise image produce a texture similar to the exemplar. Ghazanfarpour and Dischler demonstrate many synthesis results for 2D or 3D (solid) textures. Furthermore,

they use multiple exemplars [GD96] to obtain realistic texture variation along the three axis in the case of the solid texture.

Lagae et al. [LVLD10] learn the weights of a multi–resolution isotropic noise function then apply histogram matching to match colors of the exemplar. The same method can be applied with anisotropic noise [GZD08] to match anisotropic patterns in the exemplar. Gilet et al. [GDS10] also handle anisotropic noise textures by decomposing the power spectrum into a set of ellipses then fit Gabor kernels with the ellipses. Galerne et al. [GLLD12] learn the parameters of Gabor kernels by decomposing the power spectrum into a set of symmetric Gaussians then solve for the parameters of the Gaussians using a *non–negative basis pursuit denoising* problem [KKL$^+$07]. The sparseness of the solution guarantees a small description of the exemplar.

Apart from by–example procedural noise, other methods build general statistical models learned from natural uniform texture exemplars then synthesize new textures having the same statistical model. Gagalowicz and Ma [GdM85] used different second order statistics to model textures with just 40 to 2000 coefficients; Heeger and Bergen [HB95] use multi–scale color histogram matching to transform a noise image in a one similar to the input exemplar; Portilla and Simoncelli [PS99] further improve the method by matching first and second order wavelet coefficients; Lefebvre and Poulin [LP00] match structural statistics; Jagnow et al. [JDR04] synthesize solid textures from 2D exemplars using stereology to learn 3D statistics from 2D observations;

Rather than statistics, GenShade [Ibr98] builds on genetic textures [Sim91] to evolve a procedural texture expression that matches the exemplar. Bourque and Dudek [BD04] use an input exemplar as a query to automatically select a procedural texture from a database. A global search in the database is performed to find the procedure that is most similar to the exemplar. A local optimization is then performed to fine tune the parameters of the selected procedure. The method fails if the exemplar is not similar to any of the procedures in the database.

The advantage of a by–example procedural algorithm is that artists are relieved from manually setting the procedure parameters. However, the state of the art by–example procedural approaches are limited to stochastic textures as shown in Figure 2.19. Data–driven synthesis which we discuss hereafter handle a larger variety of textures.

2.3.2 Data–driven approaches

Data–driven synthesis algorithms are by–example synthesis algorithms that generate a texture by directly copying texels from the exemplar. By directly using data from the exemplar, these algorithms better preserve high order statistics and can handle non–stochastic textures as opposed to existing by–example procedural techniques. For instance, Figure 2.20

Figure 2.19: *From left to right:* The exemplar, result of Heeger and Bergen [HB95], result of Lagae et al. [LVLD10], result of Gilet et al. [GDS10] and result of Galerne et al. [GLLD12].

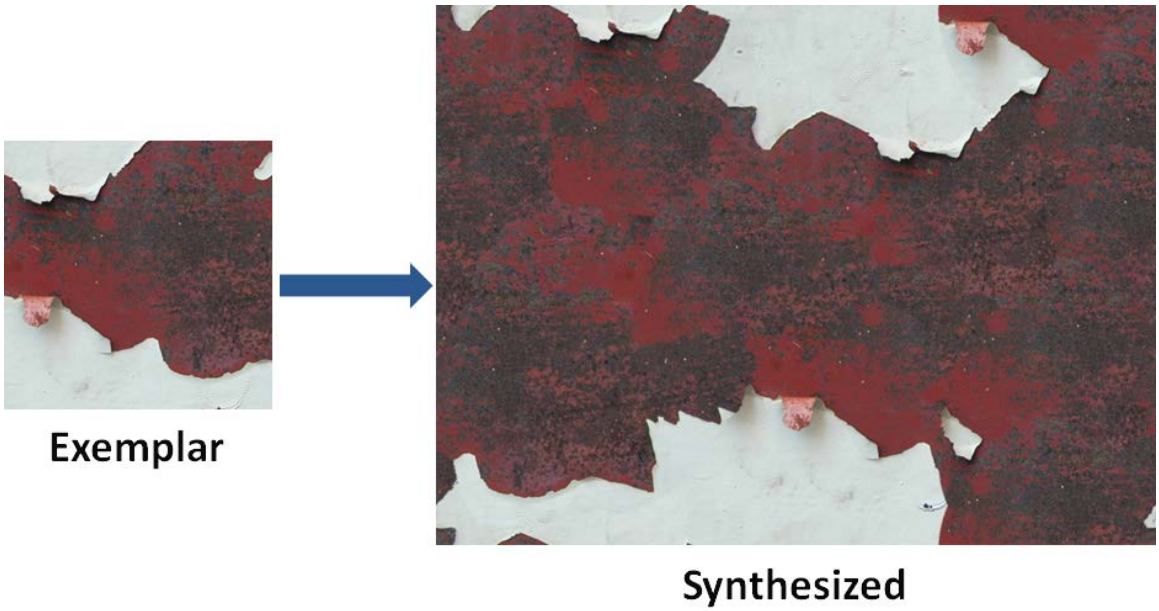

Figure 2.20: Data–driven texture synthesis generates a texture similar to an input exemplar by directly copying data from the exemplar. *Exemplar source: Mayang's Textures.*

shows a synthesis result that uses a data–driven synthesis algorithm that we describe in chapter 4.

The obvious downside of data–driven texture synthesis is that the exemplar must be

in memory during the synthesis process. To make sure that the exemplar uses the least possible amount of memory, inverse texture synthesis [WHZ+08] summarizes a texture into the smallest possible exemplar that retains all the features of the texture.

Data–driven texture synthesis can be roughly categorized into *pixel–based* synthesis and *patch–based* synthesis. Pixel–based synthesis approaches copy one pixel at a time from the exemplar to the output. Patch–based synthesis approaches in the other hand copy a larger sub–region (a patch) at a time from the exemplar to the output. Since large regions are copied from the exemplar, patch–based synthesis approaches are generally better at preserving the structures of the exemplar.

In chapter 4 we introduce a *fast* patch–based synthesizer. Because of this, we give particular attention to the speed of the different algorithms. We refer the reader to Wei et al. survey [WLKT09] for further details on data–driven texture synthesis.

Pixel–based synthesis

Early pixel–based synthesis methods [Gar81, PP93, EL99] synthesize a texture pixel by pixel in a defined order using a *neighborhood matching* process. In these work, the texture is modeled as a *Markov random field* and the output is an image which maximizes the likelihood of the field of colors to belong to the model. For instance, Efros and Leung [EL99] grow the output pixel by pixel. For an output pixel (x, y), its value is set by first searching in the exemplar for a set of neighborhoods similar to the neighborhood that is centered at (x, y) in the output. Among the similar neighborhoods, one is selected randomly and the centering pixel is copied at (x, y) as illustrated in Figure 2.21. The quality of the result is highly dependent of the chosen size N of the neighborhood. In particular, when N is large the performance are greatly degraded.

Many researches have been done to accelerate pixel–based synthesis: Wei and Levoy [WL00] synthesize the texture in a multi–resolution fashion as described in Algorithm 1. This eliminates the dependency of the neighborhood size N. They also use tree structured vector quantization to accelerate the neighborhood search.

To further accelerate neighborhoods comparisons, Lefebvre and Hoppe [LH06a] compress the neighborhoods using PCA. This lets them represent large patches with a small feature vector. This way, small neighborhoods of feature vectors are enough to achieve high synthesis quality.

Ashikhmin introduced the notion of coherence [Ash01] that further improves performance. Although designed to improve performance, Ashikhmin coherence method also improves quality. The notion of coherence has been extended to k–coherence [TZL+02, ZG02]

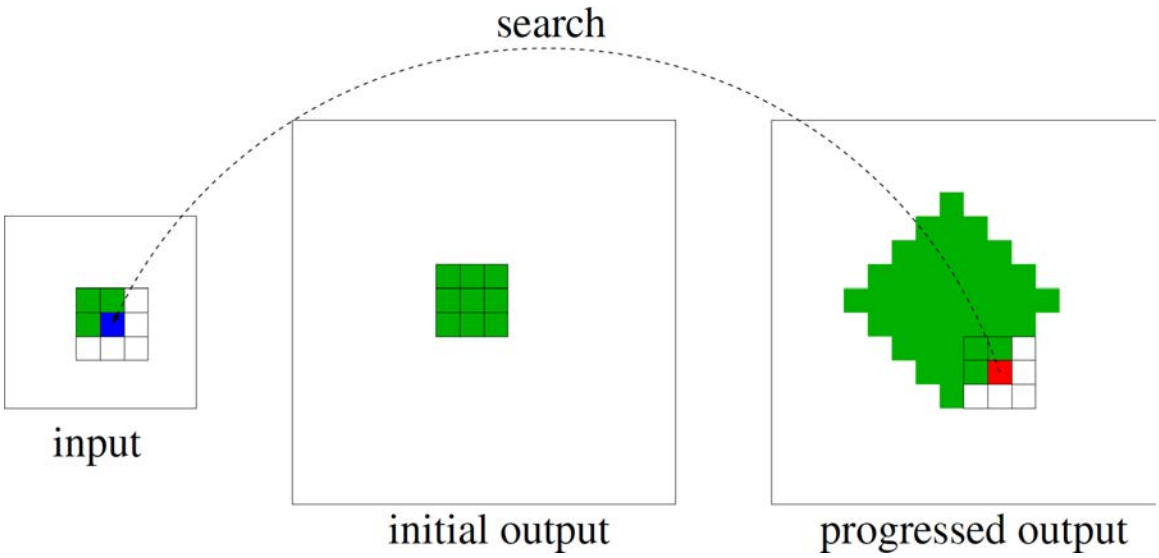

Figure 2.21: The output is initialized with a random neighborhood copied from the exemplar. During synthesis, a neighborhood in the exemplar similar to the neighborhood around the red pixel is matched. Then, the blue pixel centering the matched neighborhood is copied to the red pixel location. *Source:* Wei et al. [WLKT09].

Algorithm 1 Multi–resolution pixel–based synthesis

Input: exemplar E, a fixed neighborhood size N.
Output: synthesized texture S.
Set size of S to $\frac{S_{size}}{E_{size}}$.
for $l \leftarrow$ coarse to fine mipmap level in E **do**
  for $x, y \in S$ in scan line order **do**
    $N_S \leftarrow$ neighborhood around x, y in S
    $N_E \leftarrow$ neighborhood in E_l most similar to N_S
    $S[x, y] \leftarrow center(N_E)$
    if $l \neq$ finest mipmap level **then**
      $S \leftarrow UpSample(S)$
    end if
  end for
end for

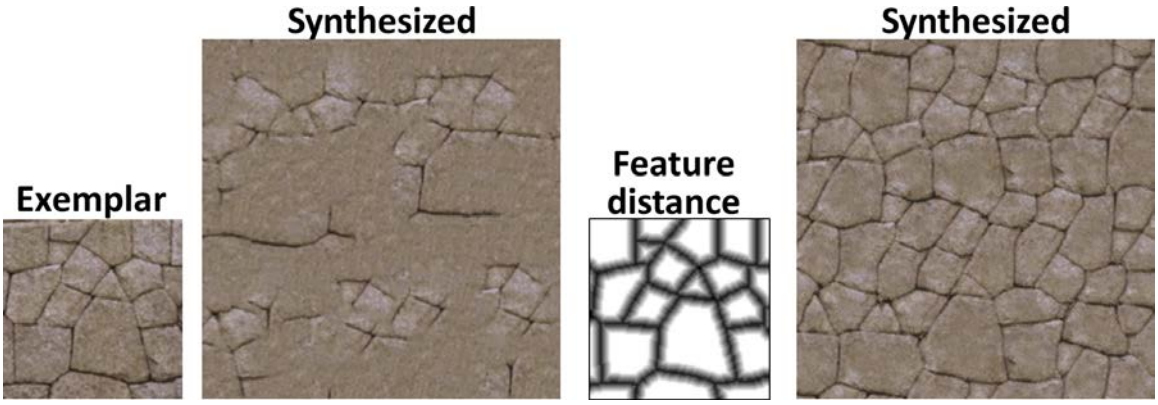

Figure 2.22: *Left:* Using pixel–based synthesis, the result does not preserve the structural features of the exemplar. *Right:* Considering a feature distance map during synthesis produces a result that well preserves features.

where a set of multiple coherent candidates is pre–computed for each neighborhood. PatchMatch [DSC03, BSFG09] removes the pre–computation required by previous coherent techniques. Busto et al. [PELS10] implement an efficient matching scheme that mixes PatchMatch with a random walk search.

In addition to algorithms accelerating the search and taking advantage of the coherent content of textures, attention has been paid to make parallel synthesis implementations. Wei and Levoy [WL03] proposed an order independent neighborhood matching scheme that enables the synthesis of pixels in parallel. Lefebvre and Hoppe [LH05] proposed a successful parallel synthesis algorithm entirely running on the GPU.

The high interactivity and the intuitive input of pixel–based synthesis algorithms make them ideal to rapidly author diverse high resolution textures content [ETB+10]. However, despite high interactivity, pixel–based approaches sometimes fail to capture structural patterns as shown in Figure 2.22. Lefebvre and Hoppe [LH06a] use a feature distance map to guide the synthesis and preserve structures (Figure 2.22). However, the creation of a feature distance map is usually complex and often requires manual intervention. We next discuss patch–based texture synthesis which naturally preserves structural patterns without needing a feature distance map.

Patch–based synthesis

Rather than operating at the pixel level patch–based synthesis takes large patches from the exemplar and merges them with an output texture being synthesized. The problem is similar to a jigsaw puzzle where the puzzle pieces are extracted from the exemplar and then reassembled to form the output texture. This process is shown in Figure 2.23.

A good patch–based synthesizer has to decide:

- How to select patches from the exemplar and where to place them on the output.

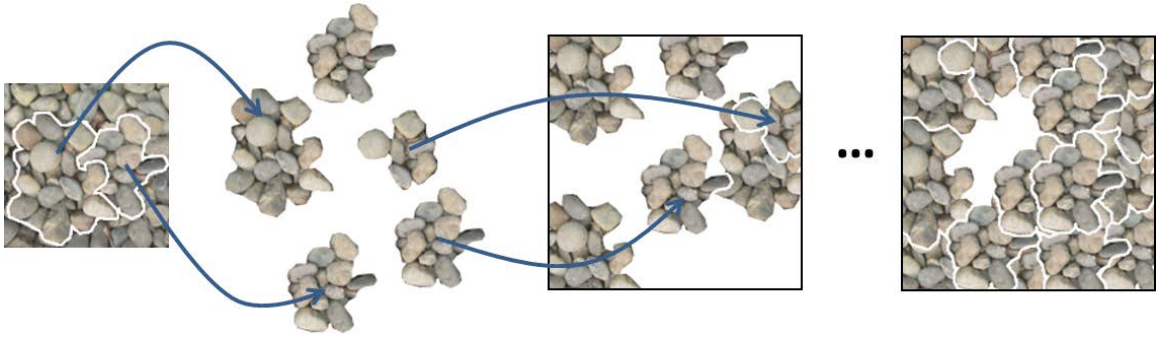

Figure 2.23: In patch–based texture synthesis, patches are extracted from the exemplar then reassembled to form the output texture. The process is similar to a jigsaw puzzle.

- How to stitch a patch with the output so that discontinuities do not appear at the patch border.
- How to ensure feature alignment in the output.
- How to iteratively refine the result.

We next discuss each of these steps:

Patch selection and placement Lapped textures [PFH00] and texture particles [DMLG02] pre–compute a set of patch patterns then randomly arrange them on the output texture. Similarly, texture bombing [SA79] fills the output by randomly arranging patch patterns that do not require special stitching.

Ma et al. [MWT11] manually extract discrete element patches from the exemplar then synthesize a new texture that match the properties and distribution of the discrete elements.

Instead of a random placement Kwatra et al. [KSE+03] use an exhaustive search accelerated by an FFT–based block matching algorithm to find a patch in the exemplar that best matches the synthesis region.

Patch stitching Early schemes [GSX, PFH00, LLX+01] overlap the patches and use feathering to reduce discontinuities.

In image quilting [EF01] patches are added in scan–line order to a grid. The new patch is stitched to its left and top neighbors in the grid. The frontier between the new patch and a previous patch is optimized, finding a path of minimal color difference with *dynamic programming* (DP). Figure 2.24 illustrates the stitching algorithm of image quilting.

The dynamic programming used in image quilting has the advantage of being very fast. It optimizes the boundaries of the the patches in $O(n)$ time. In chapter 4 we take advantage of a similar fast DP to develop a novel patch–based synthesizer. The DP is described in Algorithm 2. In this algorithm, a color difference map M is input and the goal is to find a

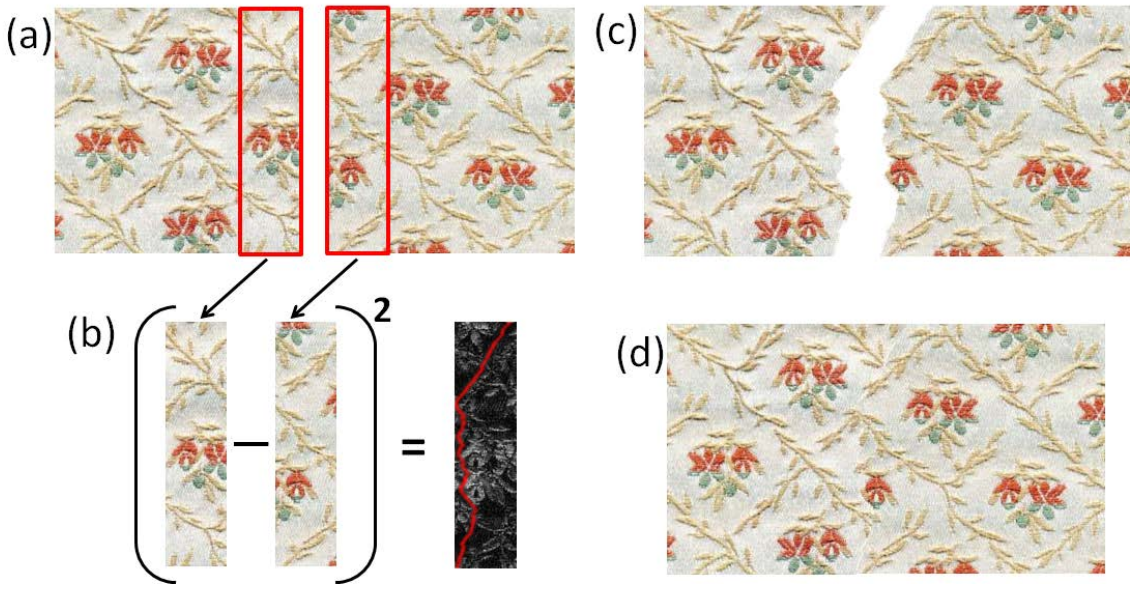

Figure 2.24: Stitching two patches using Image Quilting. *(a):* the two texture regions inside the red boxes are overlapped. *(b):* the overlapped regions are compared to obtain an error map (the gray scale image). Dynamic programming is then used to find a path of minimal color difference (the red path in the error map). *(c):* the found path defines an optimized boundary for the two patches. *(d):* the final stitching.

path P that starts from the top of M and ends at the bottom of M so that color differences along P are minimized.

Graph–cut textures [KSE+03] uses graph–cut [BVZ99] to find an arbitrary shaped cut around the patch. This method produces better results than image quilting. However, the used graph–cut algorithm has an $O(n^2)$ complexity while dynamic programming has $O(n)$ complexity. The push–relabel graph–cut algorithm [GT88] is suitable for a parallelized implementation but its $O(n^3)$ complexity makes it slower than DP even if implemented on a GPU.

Rather than optimizing the boundaries of the patches, gradient domain Poisson image editing [PGB03] alters the colors of the pasted patch so that the gradients are consistent with the target texture.

Drag–and–drop pasting [JSTS06] optimizes the boundary of the patch so that Poisson image editing produces the best possible result. This is done by computing a cyclic boundary around the patch so as to reduce gradient mismatch during Poisson optimization. Their method relies on dynamic programming to find the boundary. However, the cyclic requirement of the boundary prevents the use of a single pass DP as in image quilting. The optimal solution is found by repeating the boundary optimization for all pairs of starting and ending positions and this increases the complexity to $O(n^2)$.

In practice, the combination of graph–cut and gradient domain Poisson image editing

Algorithm 2 Image quilting DP

Input: color difference map M of size $w \times h$.
Output: minimum color difference path P.
// compute the cost of all paths that start from the top of M
for $y \in [2..h]$ **do**
  for $x \in [1..w]$ **do**
    $M[x, y] \leftarrow M[x, y] + min(M[x-1, y-1], M[x, y-1], M[x+1, y-1])$
  end for
end for
// identify the best path that starts from the top and ends at the bottom of M
$P[h] \leftarrow \arg\min_{\forall x}(M[x, h])$
// backtrack the best path
for $y \in [h-1..1]$ **do**
  $P[y] \leftarrow min(M[P[y+1]-1, y], M[P[y+1], y], M[P[y+1]+1, y])$
end for

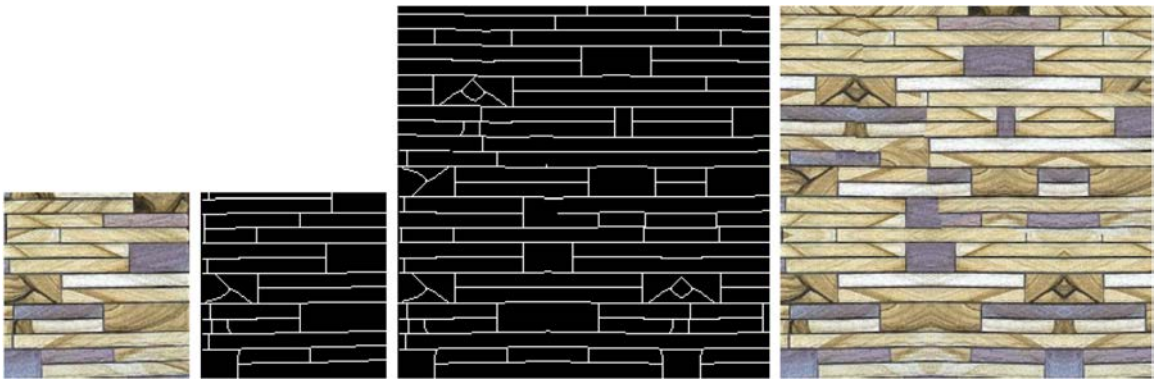

Figure 2.25: Synthesis with Wu et al. [WY04] approach. *From left to right:* The exemplar, a feature map extracted from the exemplar, a synthesized feature map and the final synthesis result guided by the synthesized feature map. Note how features are aligned in the synthesized feature map. *Source:* Wu et al. [WY04].

constitutes the most used stitching technique for textures and images [ADA+04, LZPW04].

In chapter 4, we present a patch stitching method based on a dynamic programming optimization that has the same complexity as image quilting and yet produces better results.

Feature alignment Most texture synthesizers have difficulties capturing and aligning contours and edges. For example, it is difficult to randomly position patches in the output such as the wood planks in Figure 2.25 are perfectly aligned.

Wu et al. [WY04] extract a sparse set of curvilinear features capturing contours (Figure 2.25). A feature map that ensure contour alignment is then synthesized and used to guide the synthesis of the final colored texture.

In appearance space texture synthesis [LH06a] the user provides a feature distance map as input. This map guides the synthesis similarly to Wu et al. method.

Figure 2.26: *Left:* A periodic tiling showing repetition artifacts. *Right:* An aperiodic tiling. *Source:* Peytavie et al. [PGGM09b].

To smoothly morph between two textures, Matusik et al. [MZD05] first locate features in the two textures to produce two feature maps. The two feature maps are overlaid and a global optimization is done to warp the first map so that it matches the second map. To improve the initial state and accelerate the optimization, the warping is done in a multi–resolution fashion.

Jia and Tang [JT05, JT08] stitch two images by detecting and aligning a sparse set of features along the boundary that separates the two images. The alignment optimization is done in a greedy way where large structured features are aligned first. To smoothly propagate deformation after alignment, an optimization similar to gradient domain Poisson image editing is applied.

Stereo–pair matching techniques [BB81] typically use dynamic programming to densely align features along epipolar lines. In chapter 4 we also use dynamic programming to align features along patch boundaries. The dynamic programming optimization has the same complexity as image quilting and is implemented on the GPU.

Szeliski [Sze06] presents a thorough description of feature alignment for image stitching.

Synthesis refinement If the synthesis result contains discontinuities due to misplaced patches, then new patches should be placed on top of the discontinuities to refine the result. Kwatra et al. [KSE+03] achieve this by changing their graph representation so that the graph–cut optimization not only tries to find a seamless cut but also tries to hide as much discontinuities as possible. In chapter 4, we also rely on an iteratively refined synthesis. The refinement process is directly integrated to our dynamic programming formulation while Kwatra et al. integrate the refinement to the graph–cut algorithm. Also, our iterative refinement enables a moronically decreasing global error similarly to the later texture optimization scheme of Kwatra et al. [KEBK05].

Tile–based methods To make patch–based synthesis fast, many techniques focus on pre–computing a set of texture tiles that are then rearranged to synthesize large textures.

The simplest method tiles the whole exemplar to synthesize a texture. Such method does not work if the exemplar is not tillable and even when it does the result would present severe repetition artifacts as shown in Figure 2.26. Many solutions exist to address the repetition artifacts. Cohen et al. [CSHD03] synthesize a set of *Wang tiles* from the exemplar. An aperiodic tiling of any size can be obtaining by assembling these Wang tiles. The assembling is fast and can be done on the fly during rendering [Wei04]. However, unlike general patch–based synthesis methods the final quality of tile–based methods is affected by the number of tiles. For instance, the number of Wang tiles increases exponentially meaning that a large number of tiles is required to synthesize an aperiodic and artifact–free large texture.

2.3.3 Output granularity

We have initially distinguished between data–driven synthesis and procedural synthesis: If an exemplar is used during the synthesis process, the algorithm is data–driven otherwise it is procedural.

We now classify synthesis algorithms depending on the granularity of the *output*:

For a texture containing N texels, some algorithms are able to output a single texel and this, independently from any other texel in the texture. Precisely, these algorithms generate in $O(1)$ memory cost a single texel at a given location. We call these algorithms: *point–wise* evaluated algorithms. Most noise–based approaches like Perlin noise [Per85, Per02] or Gabor noise [LLDD09] are point–wise evaluated. With these approaches, a single texel can be quickly computed with little access to the memory. This way, memory and bandwidth are almost entirely replaced with computation. Furthermore, only visible texels within the currently rendered frame have to be computed.

Other algorithms do not support point–wise evaluation and need to generate the whole output texture at once. This means that even a single texel takes $O(N)$ memory space to generate. This is the case of most non–noise–based methods including graph–based procedural methods like Substances. Substances for instance may contain point–wise evaluated nodes based on a point–wise method like Perlin noise, but also nodes applying non–point–wise global operations such as *warping* or *histogram correction*.

The obvious limitation of non–point–wise algorithms is that the whole output texture has to be generated and stored in video memory. Also synthesizing $O(N)$ texels may be computationally expensive requiring the result to be stored on a hard-drive or a DVD prior to rendering. This becomes problematic if the number of such textures is considerable. In chapter 4, we describe a patch–based synthesizer that does not support point–wise evaluation. In order to use this synthesizer to render large environment with many synthesized textures, we describe in chapter 5 a method to encode the synthesis result in a compact structure from which texels can be accessed in a point–wise fashion.

Generating a texel in $O(1)$ or $O(N)$ memory space represents two extremes. Some other algorithms like the order independent scheme of Wei and Levoy [WL03], the scheme of Lefebvre and Hoppe [LH05] or the scheme of Dong et al. [DLTD08] are able to compute a texel by only knowing a small neighborhood of texels around it at all levels of detail. Since the neighborhood size is a small constant and since the levels of detail follow a logarithm to the base 2 scale, the number of generated texels required to compute the final value of a single texel is proportional to $O(log(N))$. Wei and Levoy [WL03] then Lefebvre and Hoppe [LH05] also show that it is possible to render an infinite texture by synthesizing a spatially deterministic texture portion around the viewer.

In order to turn a non–point–wise evaluated algorithm into a point–wise evaluated one, some algorithms sacrifice a certain amount of memory by pre–computing an intermediary result that is stored in memory and from which texels can be accessed directly in a point–wise fashion. For example, Cook and DeRose [CD05] noise approach and Goldberg et al. [GZD08] anisotropic noise approach generate an intermediary result consisting in a small set of octave images. The set is reduced by exploiting the fact that octaves with different scales and orientations can be obtained by rotating and scaling a same octave image stored in memory. A weighted sum of texels in the stored octaves gives the final result of a texel.

In the case of data–driven synthesis methods, some algorithms pre–compute from the exemplar a set of tillable textures that are combined together during rendering. For instance, Wei [Wei04] pre–computes and stores a set of Wang tiles then, during rendering, texels are accessed from the Wang tile set in a point–wise fashion and without any additional intermediary result.

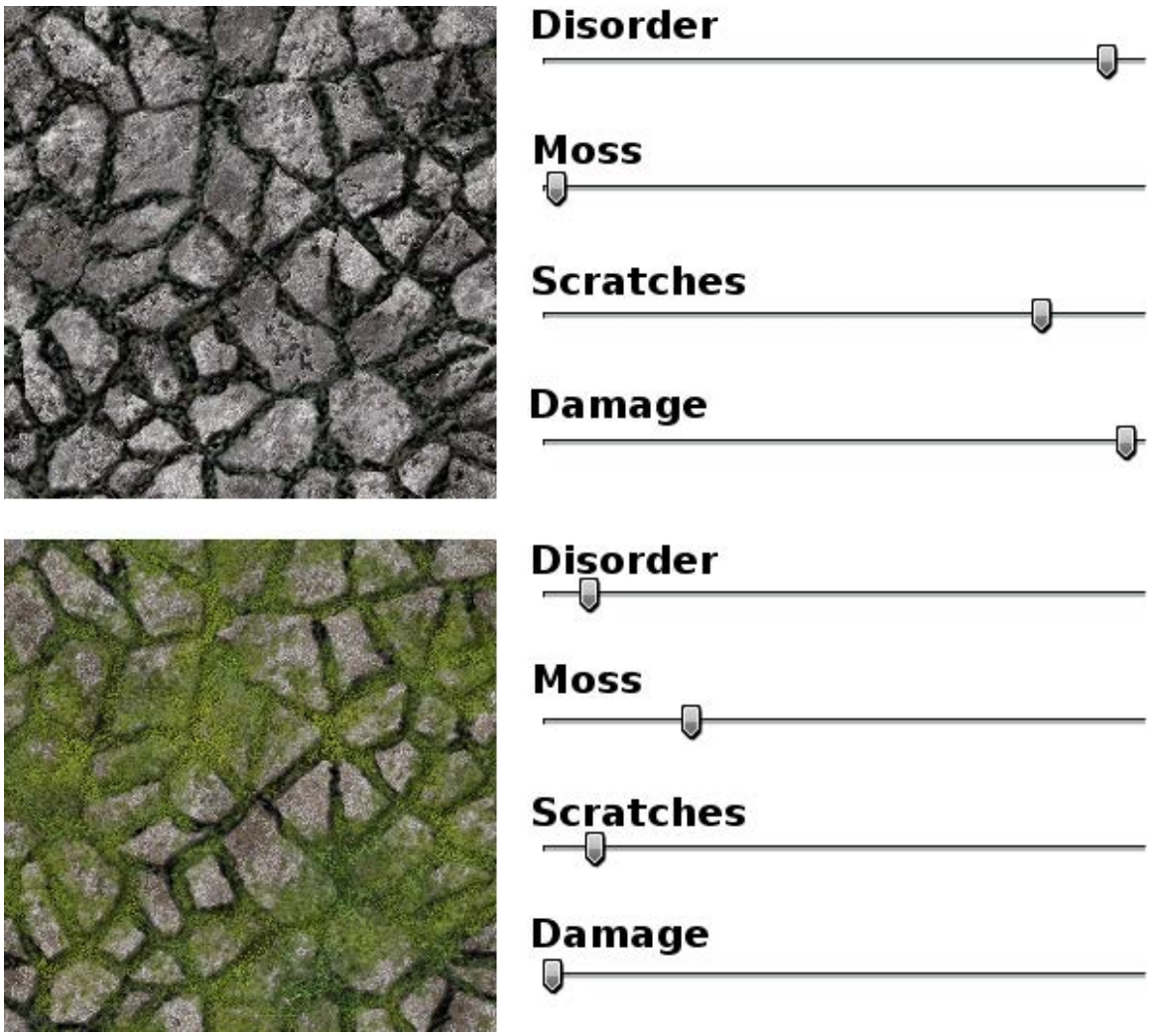

Figure 2.27: *Left:* Two different variation of a Substance texture. *Right:* The sliders controlling the appearance of the texture.

2.4 Procedural textures visualization and navigation

Procedural textures have the ability to create a space of different textures rather than a single image. This space is typically explored by the user through sliders mapped to the parameters (Figure 2.27). Since parameters have a complex link to the final result and may also depend on each other, it is difficult to realize the richness of the generated textures without spending time trying a variety of settings.

Not only parameter selection and manipulation are rather tedious but the complexity of the space of texture appearances makes complex the interpretation of the possibilities that a procedure offers. The complexity is even higher if one considers multiple procedural textures, for instance when selecting a procedural texture from a pre–existing database.

We present in chapter 3 approaches to efficiently visualize and select parameters for procedural textures. In this section we discuss related work to these approaches.

Figure 2.28: *Top:* A market–place of substances. *Bottom:* A single texture in this market–place can produce a variety of different appearances.

2.4.1 Down–sampling and Cropping

A well–known example of a texture database is a web–based texture market place showing a gallery of textures from which the user can make a selection. Conventional texture galleries typically present the user with static grids of thumbnails. The user can select a thumbnail to see a high resolution version of the texture.

In practice the thumbnail concept is also used for procedural textures. Figure 2.28, Top, shows an actual procedural texture market–place. Since a procedural texture can have a diverse content a single thumbnail is not enough to display what the procedure is capable of (Figure 2.28, Bottom). Because of that, the user has to first select a texture thumbnail in the gallery then use sliders to navigate through the parameters of the selected texture. Such process can be very tedious especially if the user wants to explore many different textures in the database.

One solution to avoid tweaking the parameters manually would be to increase the number of displayed thumbnails per texture so that most appearances are displayed at once. However, the display limited pixel space would require reducing the size of the thumbnails so that all of them fit in the display. Reducing the size of the thumbnails can be done by down–sampling them. This has the drawback of altering the quality of the texture and even losing fine details as shown in Figure 2.29. Another way is to crop the textures. Cropping however loses texture patterns unless the texture is fully stochastic. Cropping effect is illustrated in Figure 2.29.

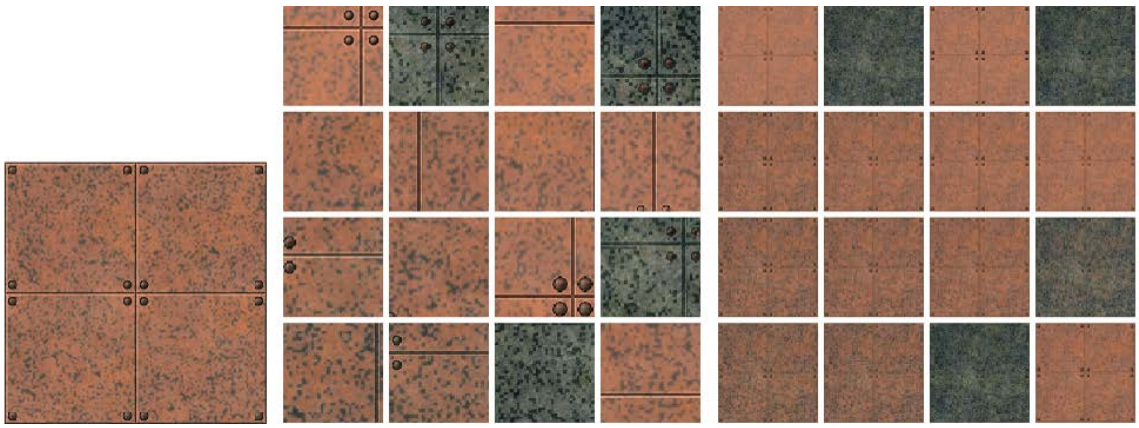

Figure 2.29: *Left:* A texture sample generated by a procedure. *Middle:* Cropped thumbnails. *Right:* Down–sampled thumbnails.

Cropping is still a common technique employed in the web for images. Many researches have been done to make cropping intelligent and automatic [VTP+10]. Intelligent methods try to detect salient objects by analyzing the gradients or by applying face detectors in case of images containing persons. Intelligent cropping assumes that the image contains salient objects and this is usually not the case of textures.

2.4.2 Texture summarization

An alternative to down–sampling and cropping is texture summarization.

Summarization as described by Simakov et al. [SCSI08] aims to create a small summary of an image by optimizing a bi–directional similarity function ensuring the following two constraints:

- **Completeness:** a patch in the input image has to be as similar as possible to at least one patch in the summary.
- **Coherence:** a patch in the summary has to be as similar as possible to at least one patch in the input image.

Image factorization [JFK03, WWOH08] solves the completeness problem by finding the best smallest set of representative patches. This takes advantage of the coherence between neighboring patches to grow larger patches that represent more data in the original image. Image factorization only solves the completeness and ignores the coherence. The factorized image only serves as a compact representation for compression purposes rather than visualization purposes.

Wei et al. [WHZ+08] and Simakov et al. [SCSI08] described solvers that optimize for both completeness and coherence. The scheme of Simakov et al. focuses on images while the scheme of Wei et al. called *Inverse texture synthesis* targets textures.

Seam carving [AS07] is a summarization technique that reduces the size of an image by progressively removing paths of pixels from the image. For example, to reduce the resolution of the image one pixel along the horizontal direction, a path starting from the top of the image and ending at its bottom is removed. The path is computed using dynamic programming and contains as little gradient energy as possible. In contrast to Inverse texture synthesis, Seam carving produces strong distortions if a large number of paths are carved and this especially when gradient exist everywhere in the image.

For many different texture appearances, summarization might be applied to each texture separately to produce small thumbnails. In chapter 3 we present the procedural texture preview, a technique that summarizes the separate appearances into a single image.

2.4.3 Sampling appearances

Procedural textures generate large number of appearances. Their parameters are usually defined in a continuous space and in theory the space of generated appearances contains an infinite set of different appearances. We call Ω the space of generated appearances. To visualize a procedural texture on a limited display space, it is important to first sample a limited number of appearances in Ω, called *representative appearances*, that best describe the procedural texture.

Sampling a continuous space

Sampling a continuous space is often done by minimizing Lloyd energy [Llo82]. Let $\mathcal{X} = (x_i)_{i=1}^n$ be the set of the n representative samples we are seeking from Ω. Lloyd energy is defined as:

$$F(\mathcal{X}) = \sum_{i=1}^{n} f(x_i) \quad ; \quad f(x_i) = \int_{Vor(x_i)} ||x - x_i||^2 dx$$

Where

$$Vor(x_i) = \{x \in \Omega \mid ||x - x_i|| \leq ||x - x_j||, \forall j \neq i\}$$

Lloyd energy can be minimized in different ways, for instance using Lloyd relaxation [Llo82] or quasi-Newton optimization [LWL+09].

Lloyd energy optimization does not result in samples reaching the boundaries of the appearance space Ω. In chapter 3 we will see that reaching the boundaries of Ω is desirable for sampling appearances. If the space Ω is unbound, e.g. the surface of a torus or a sphere, there will be no concern about the boundaries of Ω and Lloyd optimization should converge to a good local minimum.

Sampling a discrete space

We now consider the case where Ω is a point cloud containing N_Ω samples. To select the n best representative samples, Ω has to be partitioned into n clusters $\{C_1, C_2, \ldots, C_n\}$ where each cluster contains as similar samples as possible. For each cluster $C_i \subset \Omega$, a representative sample has to be selected so that it is as similar as possible to all other samples in the cluster. The sample that is as similar as possible to all other samples in the cluster corresponds to the geometric median of the cluster. The geometric median x of a cluster C_i is defined as:

$$x = \underset{x \in C_i}{\arg\min}\left(\sum_{y \in C_i} ||x - y||\right)$$

Within a cluster C_i, we call *inter–cluster–bottleneck* the distance d_i between the representative sample and the furthest sample from it. Among all clusters, we call a *bottleneck* the distance d such as: $d = \max_{\forall i}(d_i)$.

The problem of selecting representative samples so that the bottleneck d is minimized is called the *p–center* problem. The p–center ensures that any sample is as similar as possible to its representative one.

When selecting representative samples, it is interesting that all of them are as different as possible from each so as to enhance the global variety in the set of representative samples. The *p–dispersion* problem aims to enhance variety by selecting representative samples so that the two closest ones are as far as possible from each other. We refer to the distance d between the pair of the two closest representative samples as a *bottleneck*. The p–dispersion aims to maximize this bottleneck distance d.

Note that the p–center optimization only ensures that a representative sample is as similar as possible to all samples in the cluster. Nothing prevents a representative sample from being different from other representative samples.

The p–center and the p–dispersion problems are two examples of what is called a bottleneck problem. They are well studied in network location theory and known to be NP–Complete [HS86].

Many heuristics exist to efficiently optimize the p–center and the p–dispersion problems. Gonzalez [Gon85] uses a successful greedy heuristic to optimize the p–center problem. The heuristic is a 2–approximation (the minimized energy is at most twice larger than the optimal energy) and this approximation is best possible (a polynomial algorithm with a better approximation does not exist). This heuristic is described in Algorithm 3. It has a $O(n \times N_\Omega)$ complexity.

Hochbaum and Shmoys [HS85] presented another best possible heuristic with the very same approximation.

Algorithm 3 Gonzalez heuristic

```
1: output ← ∅ // start with an empty output
2: output[1] ← rand(Ω) // add a random sample from Ω
3: for j ∈ {1..N_Ω} do
4:    // closest distances between output and Ω
5:    bottleneck[j] ← distance(output[1], Ω[j])
6: end for
7: for i ∈ {2..n} do
8:    j ← arg max(bottleneck) // find the bottleneck
9:    output[i] ← Ω[j] // add the bottleneck to output
10:   for j ∈ {1..N_Ω} do
11:      // update closest distances between output and Ω
12:      bottleneck[j] ← min(bottleneck[j], distance(output[i], Ω[j]))
13:   end for
14: end for
```

To solve the p–dispersion problem, Ravi et al. [RRT91] rely on Gonzalez heuristic that solves the p–center problem. They showed that Gonzalez heuristic is also a 2–approximation and best possible heuristic for solving the p–dispersion problem. In fact, Gonzalez heuristic and Hochbaum and Shmoys heuristic are best possible heuristics for both the p–center and the p–dispersion problems. Although different, the two problems are related by duality [TFL83].

2.4.4 Graph labeling

In chapter 3, we encounter the problem of ordering appearances in a two dimensional grid. This translates to a 4–adjacency graph labeling problem where nodes correspond to appearances (the labels) and the edges correspond to dissimilarity between appearances. The goal is to choose the labels so that the cumulative cost of the edges is minimized.

Successful graph labeling heuristics are based on a Markov random field energy minimization in which a same label can be assigned to many different nodes. A survey by Szeliski et al. [SZS+06] describes these heuristics.

If a label has to appear exactly once and all labels have to be used, the labeling problem becomes harder: Little work has been done in relation with this problem. Chartrand et al. [CEVZ05] introduced the term γ–labeling. They consider an ordered list $\{0, 1, \ldots, n\}$ of integers representing the graph nodes. The cost of an edge is equal to the absolute value of the difference between its two nodes. They gave many theoretical properties regarding the value of the global optimal energy but they did not give an algorithm to solve the labeling.

If the graph is a unary tree (a path) the problem is equivalent to finding a Hamiltonian path. Finding a Hamiltonian path is NP–Complete. The most accurate heuristic achieves a $\frac{33}{25}$–approximation [HR00].

Hamiltonian path heuristics cannot be easily extended to the case of a grid layout. To prevent repeating labels in a grid, Cho et al. [CBAF08] add a special exclusion heuristic to the loopy belief propagation algorithm [Pea88]. They were able to solve a jigsaw puzzle of up to 400 labels if the initial state has a dozen of labels fixed in the right position [CAF10]. If priors do no exist, the exclusion heuristic fails from preventing labels to repeat.

2.4.5 Dimensionality reduction

Ordering high dimensional appearances on a two dimensional plane is also related to dimensionality reduction. We describe here techniques that project high dimensional points on the 2D plane. We consider projecting on the 2D plane; however, all the described techniques are able to project on any other dimension.

Principal component analysis (PCA) is a common technique to reduce dimensionality. It finds the best 2D plane that minimizes the projection error. Non–linear dimensionality reduction techniques like Isomaps [TdSL00] or Locally Linear Embedding (LLE) [TdSL00] project data on a non–planar 2D manifold that is then flattened on a 2D plane. These techniques require connecting each high dimensional point to its neighbors using k nearest neighbors(k-nn) for instance. The result is however very sensitive to the chosen value of k. These methods may also fail if there is not a dominant 2D manifold embedded in the high dimensional space.

Multidimensional scaling (MDS) [Kru64] is a projection technique that is independent from the dimensionality of the input points. It takes the points affinity matrix (pair–wise distance matrix) as input and finds an ordering of the points on the 2D plane so that points close to each other in the multidimensional space are also close to each other in the 2D space. Note that Isomaps can be seen as an extension of MDS that can handle non–linear structures.

Richards and Koenderink [RK95] propose *trajectory mapping*, a technique that replaces the metric function used in MDS by trajectories. Trajectory mapping locates smooth paths of appearances (trajectories) in the multidimensional space. It then tries to find an ordering of the points in the 2D space so that a path that connects appearances in the multidimensional space connects the same appearances in the 2D space. Richards and Koenderink argue that their method could be more appropriate for complex multidimensional spaces such as the space of textures. However, the final layout obtained after projection would not be very different from the one produced by MDS. Furthermore, in the general case, Heaps and Handel [HH99] claim that it is difficult to model the multidimensional space of textures with linear or non–linear manifolds.

2.4.6 Parameters selection, exploration and visualization

We previously described the fundamentals of sampling, graph–labeling and dimensionality reduction. We now review previous work related to exploring appearances of parametrized graphics objects.

Direct parameter selection

In a human–computer–interaction context techniques have been developed to improve the efficiency of sliders by augmenting them with visual clues such as histograms or color bars [Eic94, WHA07]. Video tapestry [BGSF10] replaces the slider used to navigate in a video by a single image strip that summarizes the content of the video. Sakamoto et al. [SKK04] generate a map of character poses – icons of postures – from which the user can select a pose parameters. The pose icons are ordered so that similar poses are near each other. This allows the selection of a path of icons describing an animation.

Content based image retrieval (CBIR) [DJLW08] is an important object of research in computer vision. The goal of CBIR is to seek, in a large database, images that resemble a query image. Using a query texture rather than manipulating parameters would greatly improve the selection. The procedural matching technique of Bourque and Dudek [BD04] can be viewed as a CBIR scheme applied to procedural textures. GenShade [Ibr98] uses genetic textures [Sim91] to tweak a procedural texture and its parameters so that its result matches an input texture.

Appearance space exploration

Rather than direct parameter selection, suggestive interfaces allow the user to interactively explore the space of appearances. In these interfaces, appearances are often suggested to the user in the form of a grid of thumbnails. By selecting one of the appearances the suggestions are updated with new appearances that are similar to the selected appearance. This allows the user to progressively refine his selection.

Design Galleries [MAB+97] is a generic approach to select parameters for Computer Graphics. Thumbnails generated from random parameters are spatially arranged for easy selection, either by recursive partitioning or using multidimensional scaling (MDS).

Sims [Sim91] uses evolutionary techniques to let the user design procedural textures proposing new variations at each selection step. Talton et al. [TGY+09] generates an interactive map showing variations of parameterized 3D models. The map is sampled from a distribution that is learned from users interactions.

Chaudhuri et al. [CK10, CKGK11] allow modeling 3D shapes by assembling primitives suggested from a database of 3D shapes.

Shapira et al. [SSCO09] let the user explore the possible color variations of an image. The user can select one or many images and the suggested color variations are updated to match the selection.

Ngan et al. [NDM06] let the user explore BRDF parameters by showing images that are similar to the image produced by the current BRDF settings. The user then steps to one of these images to refine the selection.

Brochu et al. [BBdF10] extend suggestive interfaces of static objects to animations. A grid of video clips showing different animations is suggested and the user rates the different animations depending on his preferences. The grid is updated with new animations that better match the user preferences.

Bourque and Dudek [BD04] propose a method to interpolate in between two procedural textures, thus defining a continuous space of appearances.

Matusik et al. [MZD05] define a space of textures from a sparse set of input textures. Images are compared to each other using a warp-based metric. Sub–spaces of images are obtained by morphing between compatible images. An interface lets the user to continuously navigate from one sub–space to another.

Layout of variants

In a suggestive interface it is important to order the set of suggestions to improve the exploration of appearances. Ordered maps are proved to be much more efficient for visual data exposition [TAH+07].

The Design Galleries interface [MAB+97] have been applied mainly to animation and rendering. While the technique could be applied to procedural textures, the low resolution of the thumbnails combined with the sparse layout is a major obstacle. Indeed, dimensionality reduction including MDS does not provide any control on the layout of the reduced space: The thumbnails are mapped to small points spread arbitrarily on the screen.

Talton et al. [TGY+09] explained the limitations of using dimensionality reduction to layout the suggestions and rather let the user manually organize the suggestions. The layouts that are manually created are stored and are automatically reused during future usage of the interface.

The genetic texture scheme of Karl Sims [Sim91] or the material browser used by Ngan et al. [NDM06] arrange very similar appearances in a 2D grid. Because the appearances are quite similar, they use a random arrangement and this is sufficient in their case.

Shapira et al. [SSCO09] first use MDS to obtain an intermediary arrangement of the appearances. The intermediary arrangement is then re–arranged by recursively partitioning the space of the intermediary arrangement in half using a kd–tree. Finally the kd–tree 2D ordering is trivially used to arrange appearances in a grid. As the arranged appearance are

quite similar and only a small number is displayed, the robustness of the arrangement is not primordial.

Cao et al. [CCL12] automatically generate a manga or comic layout from a set of input images. The layout algorithm learns a distribution of layouts from a large database of comics and mangas. The user defines the semantic of the manga or the comic and a layout of suggestions is sampled from the distribution of layouts.

Sakamoto et al. [SKK04] directly sample and arrange icons of postures in the 2D grid. The authors rely on the self–organizing map (SOM) algorithm [KSH01]. The SOM algorithm is however applied on parameter space rather than appearance space. In chapter 3 we also use the SOM algorithm. In our case, the SOM algorithm is applied in appearance space and is adapted to maximize the variety of the sampling. Please refer to section 3.2.2 of chapter 3 for a comparison.

Comparison between selection in parameter space and appearance space

Kerr and Pellacini [KP10] conducted a user study to compare direct selection with sliders and thumbnails–like suggestive interfaces in the case of material shader parameters section. The study concludes that, under interactive feedback, sliders perform best in particular when precise adjustments are required while suggestive interfaces are useful for artistic exploration without the need for precise parameters adjustments.

In another user study, Kerr and Pellacini [KP09] evaluated different lighting design interfaces. They compared three types of interfaces:

- **Classical direct interface**: the properties of the light are selected through sliders and by directly moving a light source to a desired position.
- **Indirect interface [PTG02]**: Rather that manipulating lights, the user can select a bright spot or a shadow spot in the scene. Once a spot is selected, the user can scale and translate the spot and the light parameters are updated accordingly.
- **Goal based interface [PL07]**: the user paints the surfaces of the scene to achieve a rough approximation of the desired look. Light properties are then optimized automatically to much the look painted by the user.

The user study conclusion is that goal based interfaces are very inefficient, while acting directly on light parameters allows for a faster design and accurate adjustments.

From the two user studies that Kerr and Pellacini conducted, we note that abstract interfaces do not necessarily improve parameter selection tasks.

In chapter 3, we use sliders as a main interface for selecting procedural textures parameters. However, as procedural textures often exhibit complex parameters, we augment the sliders with visual previews to give more insight about the parameter settings.

2.5 Texture streaming

In chapter 5 we want to render scenes that use a large number of synthesized textures. Most data–driven techniques and substance textures do not allow for point–wise evaluation. In this case, the synthesized textures have to be generated prior to rendering and this could lead to high memory usage.

Texture streaming reduces memory usage by only synthesizing a texture when it is needed for the current frame. A good streaming system would accurately decide when to start synthesizing a texture, when to load the texture in memory and when to unload it to free the memory.

Loading all texture data, typically in video memory or more generally in a *texture pool*, is wasteful since only a small portion will be eventually displayed. In fact, the number of displayed texels can never exceed the number of pixels in the frame buffer and this, regardless from texture resolution and scene complexity. Furthermore, only the lower levels of details are usually required for the rendering of distant textured objects and this is especially the case when high resolution textures are used.

Visibility and culling [COCSD03, Won03] are standard tools to detect whether an object and its corresponding textures are whether visible or not.

Tanner et al. [TMJ98] developed a technique called *clip mapping* to render a very large texture, called a *virtual texture*, that does not fit in memory. The clip mapped virtual texture corresponds to a partial mipmap pyramid clipped around a point called the *clip center* (Figure 2.30). The clipped portion corresponds to invisible data that will not be loaded in memory. Tanner et al. also showed how to update the virtual texture when the camera moves.

Huttner [Hüt98] subdivides a virtual texture into tiles to form an *MP-Grid*. At rendering only the visible tiles are loaded in memory. Because of hardware limitations at the time MP-Grid was published, Huttner had to subdivide the triangles that are textured with multiple tiles.

Peachey [Pea90] developed *Texture On Demand*, a texture caching scheme commonly used by off–line renderers. Peachey uses a tiled decomposition of the textures. A virtual memory paging system is used to cache texture tiles. A least recently used (LRU) policy is used to determine which tiles go in and out of the cache.

Lefebvre et al. [LDN04] use a virtual tile caching scheme suitable for interactive applications running on GPU. A more recent implementation of such virtual tile caching scheme is used in the game engine Id Tech 5 [vW09] and is well known as *Sparse Virtual Texture* mapping *SVT* [Bar08, CESL10, OvWS12].

Most recent video game engines rely on a streaming system to handle texture data. A common simple technique used in video games is to pre–render the scene to detect which

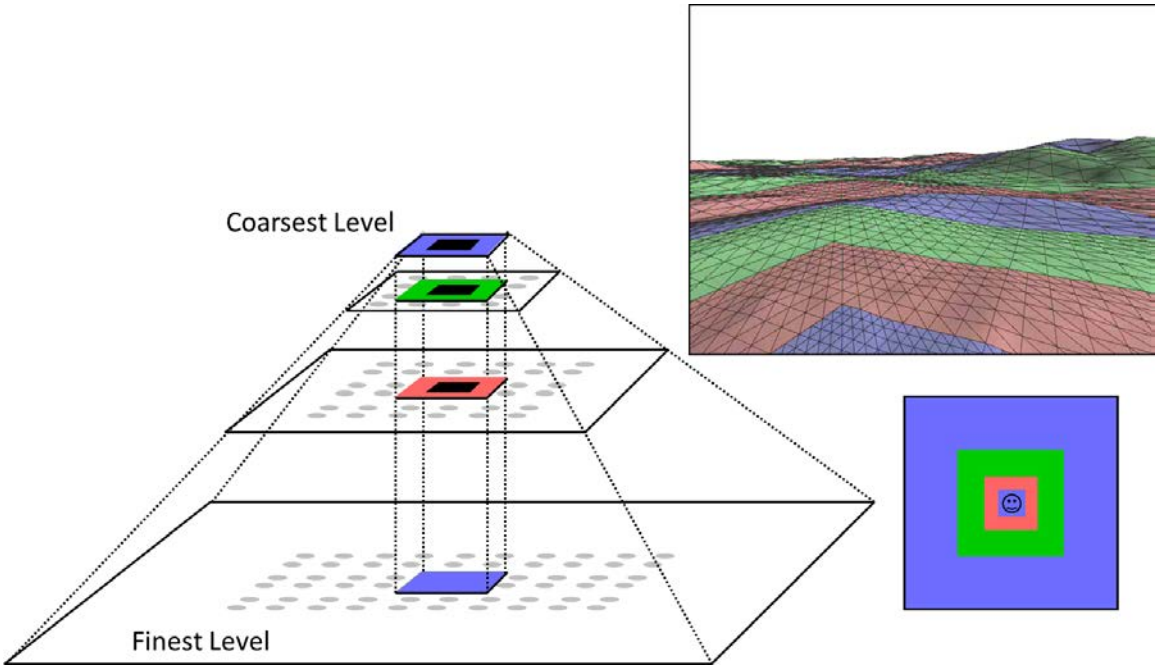

Figure 2.30: *Left:* A clip map virtual texture corresponds to a stack within the mipmaps pyramid clipped around the clip center. *Right:* A clip–mapped terrain geometry showing lower geometry resolution at lower level of detail in the clipmap stack. The clip center is illustrated with a smiley face on the bottom–right image. *Source:* Geometry clipmaps [LH04].

textures and particularly which mip–levels are visible [Can05]. The visible textures are then loaded asynchronously for rendering. In case the texture pool is full one of the invisible textures is evicted. Large textures or textures kept in the pool for a long time without being referenced are likely to be evicted.

Unreal engine 3 [Gam06] prioritizes the loading of the textures based on some user defined priorities but also based on the distance to the textured object and an approximation of how many pixels the texture occupies on screen.

Apart from detecting what texture data should be loaded for the current frame, a very important aspect of data streaming is to ensure that the data are always available at the appropriate mip–level *before* they are required for display. This process is often called pre–caching, or look–ahead policy. A pre–caching system tries to predict depending on the user motion what will become visible next and start loading the required texture data before they become visible. Such prediction needs to take into consideration the total spent time until the textures would become visible and the total spent time to load them.

Regarding video game engines, little efforts have been done in predicting the viewer state and pre–caching textures. However, many networked multi–player games try to locally predict the position and state of objects that are controlled by other machines. Usually, the prediction is performed by extrapolating information of previous states stored in the local

machine. The prediction also takes advantage of the semantic of the game in most cases. Such prediction scheme is known in navigation as *dead reckoning* [PW02]. For example, in a multi–player racing game, many cars are competing where each car is controlled by a separate machine in the network. In one machine controlling one car, rather than waiting for the state of all other cars, the dead reckoning system will advance the cars by extrapolating their previous positions and by making sure that there is a high probability that the cars will follow the path of the racing track and will avoid hitting each other. Once the correct state of one car is known, an smooth interpolation changes the predicted state towards the correct state. Pantel et al. [PW02] discuss dead reckoning and present different extrapolation schemes. An alternative that uses artificial intelligence techniques to emulate the behavior of objects controlled remotely is called *Doppelgänger* [BDL+08].

In the context of remote walkthrough applications, several pre–caching heuristics have been investigated to stream geometry. These heuristics assume a very high latency network which means that expensive on–line optimizations at the server side may be affordable.

Schmalstieg and Gervautz [SG96] stream meshes progressively starting from coarse levels of detail (LOD) to finer levels. When all geometry for the current frame is available at the correct LOD and if memory is available, the system starts pre–fetching finer LOD for objects close to the viewer.

Chim et al. [CGL+98] stream progressive meshes [Hop96] over the Internet. Each object in the scene is associated with a probability of being streamed first. The probability is computed as a function of the distance of the object to the viewer, the size that the object occupies in the scene and the historic of previous user movements.

Teler and Lischinski [TL01] consider different types of viewer states: standing still, rotating or moving forward. They predict the next visible objects by supposing that once a state starts it will continue in the near future.

Popescu and Liu [PL02] propose a dynamic random walk version of Teler and Lischinski's static prediction: the future state can be randomly changed at any time rather than always considering that a state will remain the same in the near future.

Guthe et al. [GBK06] store the viewer motion and direction of some previous frames and extrapolate these information to estimate the next state of the viewer.

Cui [CXY05] also extrapolate previous viewer information in a dead reckoning scheme built on top of Teler and Lischinski scheme.

Rather than a short time prediction, Pribyl and Zemcik [PZ10] focus on a longer time prediction to estimate the next location of the viewer. To achieve such a prediction, they build a database of paths followed by a number of viewers. To estimate the next location of a new viewer, they seek similar paths in the database and randomly sample a next location from the sought paths.

Chapter 3

Exploring synthesized textures

3.1 Introduction

Procedural textures often expose a set of parameters controlling their final appearance. This lets end users tune the final look and feel, typically through a set of sliders (Figure 3.1). However, as discussed in section 2.4 it is difficult to browse through a database of such procedures, since each one of them can generate any number of appearances. Similarly, it is often difficult to predict the changes introduced by a given slider, especially as their effects interact in non trivial ways.

In this chapter our goal is to ease the exploration of large databases of procedural textures, helping users visualize and navigate the space of possible appearances. We focus on the case of users discovering a procedural texture generator, i.e. a user without knowledge of the procedural texture.

Our first contribution that we call a *procedural texture preview*, is an algorithm that summarizes a procedural generator in a single, limited–resolution image. This makes it easy to visualize *at once* the space of possible appearances. The main challenge in creating such a preview is due to the limited pixel–space, while the texture may exhibit a large number of variations. An example of a procedural texture preview is illustrated in Figure 3.2.

Our second contribution that we call *scented sliders*, is an algorithm generating preview images for the slider interface used by many procedural texture software. Our slider previews help the user navigate parameters by providing a visual prediction of the effect of each slider. The sliders also emphasize the differences that will be produced by a parameter change. Slider previews are illustrated in Figure 3.3.

Our third contribution is to combine the procedural texture preview and the scented sliders into a unique interface. The user may select and explore appearances from the procedural texture preview, which now acts as a *texture palette*. The texture palette is useful for rapid selection of appearances. The scented sliders give further control on the

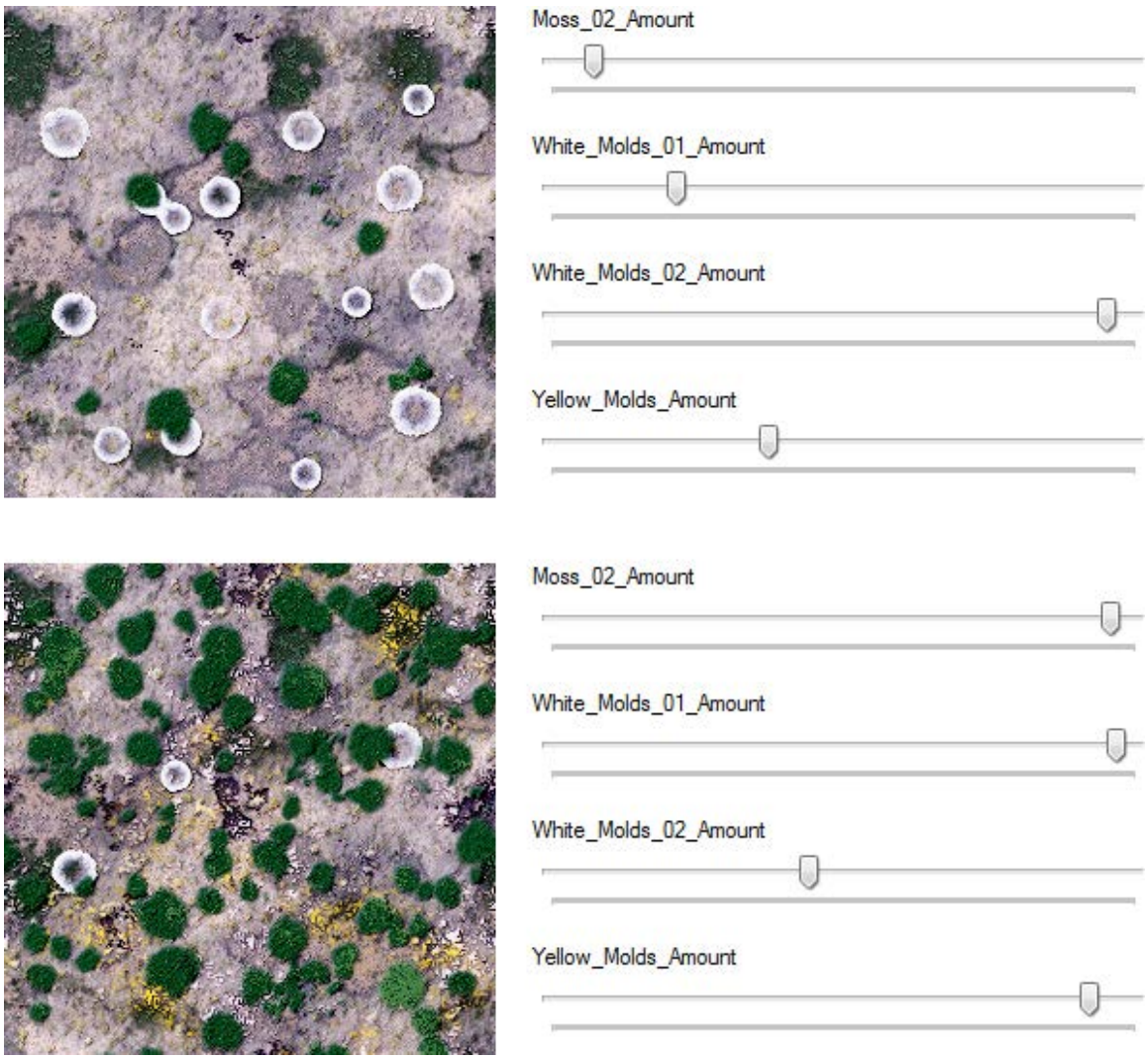

Figure 3.1: *Left:* Two textures produced by a same texture procedure. *Right:* The sliders controlling the texture appearance.

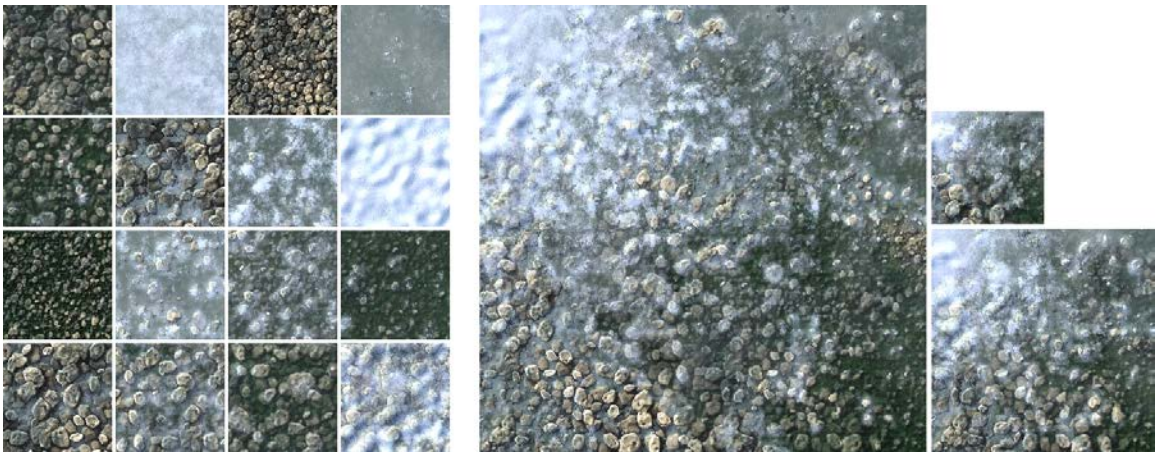

Figure 3.2: Left: Sixteen random thumbnails cropped in the same procedural texture. Right: Our procedural texture preview summarizes in a single image the possible appearances. A 512^2, 256^2 and 128^2 previews are shown. Note how the smallest preview, which has the same size as the thumbnails, shows the most important variations of the texture (the frozen water, the snow, the moss, the pebbles and transitions between them).

Figure 3.3: Left: Two variants from a procedural texture with standard sliders controlling their appearances. Right: the same procedural texture with our visual slider previews controlling their appearances.

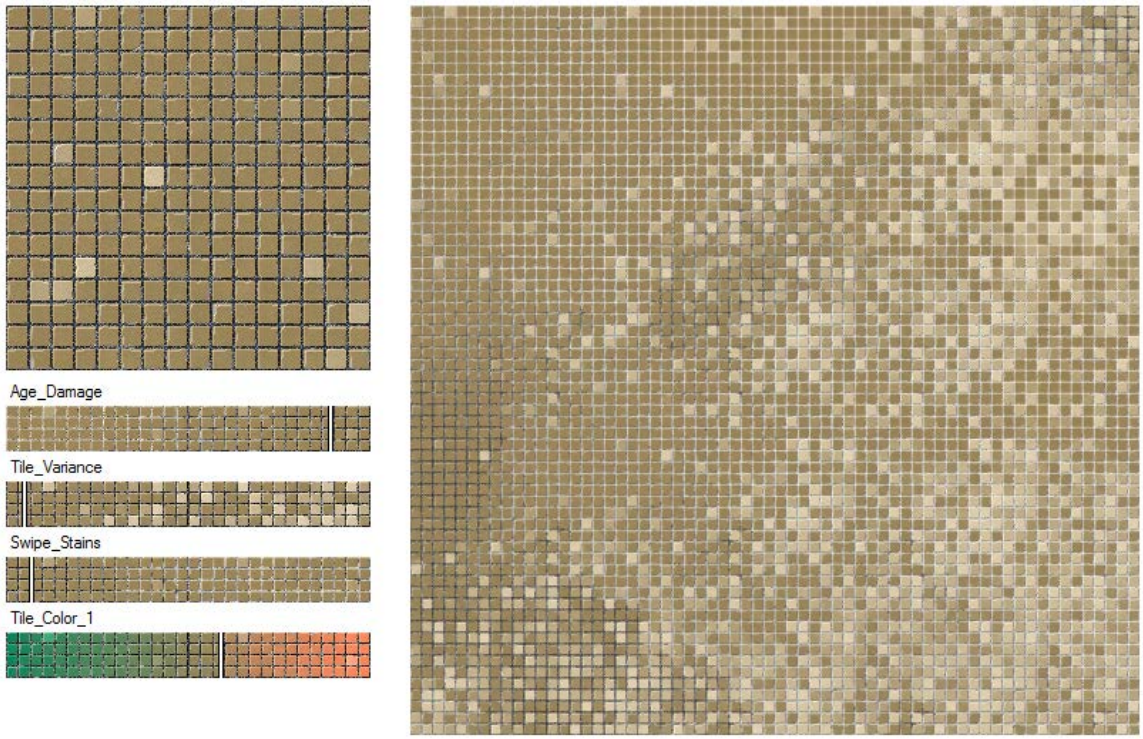

Figure 3.4: The texture palette interface lets the user directly select an appearance in the preview summary. The visual sliders allows for fine tuning the parameters.

parameters and are useful to accurately fine tune the parameters. The interface combining a texture preview and scented sliders is illustrated in Figure 3.4.

The work described here is published at Eurographics 2012 [LLD12a, LLD12b].

Hypothesis and scope of the techniques The techniques presented here focus on textures: images representing self–similar materials usually tiled over large surfaces. We exploit specificities of textures throughout the work, and hence our results do not apply to arbitrary images, such as for instance photographs or paintings.

We expect the space of all appearances to contain progressive changes in the overall appearance of the textures: the space of appearances should contain a single connected manifold. Textures not enforcing this assumption, e.g. having a parameter that turns a grass texture into a wood texture without any transition, will result in less visually continuous previews. We also expect parameters to be defined on a finite range of values.

The textures we use are made by *Allegorithmic* artists. Generic parameters affecting material properties like normals or specular intensity are not considered and are not shown in our previews. In addition all textures have parameters affecting the global tone of colors (hue, saturation, value...). We ignore such parameters to focus on the most relevant effects. Nevertheless, we consider these parameters for the slider previews.

3.2 Procedural texture preview

A texture preview is synthesized in two steps: *mapping* and *synthesis*. The mapping step produces a 2D parameter map, noted $\mathcal{X}$. This 2D map is a regular grid with a choice of parameters in each cell. It offers a regular, uniform coverage of the available pixel space. The synthesis step produces a color image – the texture preview, noted $\mathcal{T}$ – from $\mathcal{X}$.

We followed three main goals in the design of our previews:

1. The texture preview is a single continuous image, with similar appearances close to each other. This makes the preview as easy to read as a map, with continuous paths between different appearances.

2. As many appearances as possible should be represented in the preview, given the allotted pixel space.

3. Each appearance in the summary should be given a similar pixel area, avoiding over or under–representation.

3.2.1 Notations

A procedural texture is a function $g(p)$ where p is a point in the space of valid parameters $\mathcal{P}$. For any given point in $\mathcal{P}$, the procedure produces an image $I_p = g(p)$. Our goal is to compute a texture preview $\mathcal{T}$ knowing only g, and under the constraints described above.

The texture preview $\mathcal{T}$ is an image of size $W_\mathcal{T} \times H_\mathcal{T}$. It is associated with a *parameter map* $\mathcal{X}$, which is a 2D grid of size $W_\mathcal{X} \times H_\mathcal{X}$ containing points in $\mathcal{P}$. For simplicity we use the notation $\mathcal{X}$ for both the map and the set of parameters stored in it.

The parameter map has a lower resolution than the final preview, so that appearance changes occur at a lower frequency than the texture details. We typically use a ratio of 32 between the size of $\mathcal{X}$ and $\mathcal{T}$.

3.2.2 Comparing appearances

In order to avoid over– or under–representation of appearances, it is important to compare the images I_p and I_q rather than comparing their parameters p and q. Indeed, a given change of parameters is unlikely to induce the same amount of change in the corresponding appearances. Figure 3.5 illustrates the difference.

Note how the white region is over–represented in the left preview, which is created by comparing parameters instead of appearances.

We thus rely on an appearance metric $\mathcal{M}$. For now let us assume that $\mathcal{M}$ is a sum of per–pixel differences over the Gaussian pyramids of the images I_p and I_q. This is too slow for practical use, but we introduce in Section 3.2.4 a faster metric with similar results.

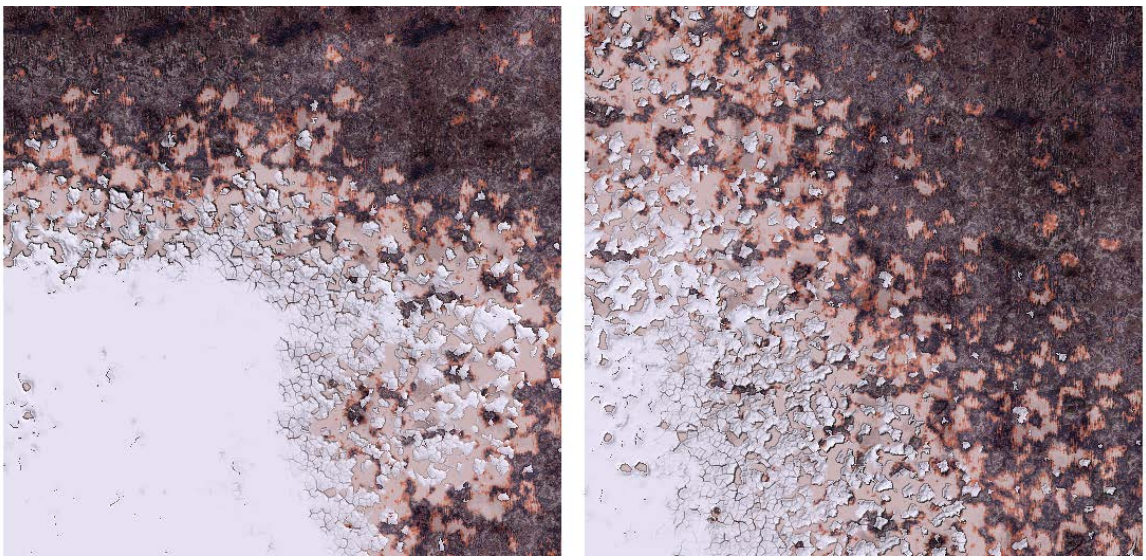

Figure 3.5: *Left:* Preview using a parameter–space metric. *Right:* Preview using an appearance–space metric.

3.2.3 Mapping

We first sample the parameter space $\mathcal{P}$ to build a dense set $\mathcal{C}$. Samples in $\mathcal{C}$ are not required to be uniformly distributed but must be dense enough to cover all possible appearances.

Next, we create the parameter map $\mathcal{X}$ by selecting a subset of parameters in $\mathcal{C}$, which we call the *representative* parameters and call their corresponding appearances: *representative* appearances. These representative appearances should best capture all appearances produced by the procedure (completeness), and should be as varied as possible (variety). The layout of representative appearances in $\mathcal{X}$ should result in progressive change of appearance when following a path (smoothness).

These criteria translate to optimizing for the following energies:

- *Completeness*: Each appearance produced by a point in $\mathcal{C}$ should be as close as possible to its closest match in $\mathcal{X}$. This ensures that each possible appearance is associated with a good representative appearance in the preview. This translates to *minimizing* the objective function:

  $$E_C(\mathcal{X}) = \max_{p \in \mathcal{C}} \min_{q \in \mathcal{X}} (\mathcal{M}(I_p, I_q))$$

  Minimizing the max ensures that points in the furthest pair are pushed closer.

  Optimizing the objective function E_C is equivalent to solving the p–center problem (section 2.4.3).

- *Variety*: Each representative appearance in $\mathcal{X}$ should be as far as possible from other points in $\mathcal{X}$, so as to enhance the overall variety. This translates to *maximizing* the

objective function:

$$E_V(\mathcal{X}) = \min_{r \in \mathcal{X}} \min_{q \in \mathcal{X}} \min_{r \neq q} (\mathcal{M}(I_r, I_q))$$

Maximizing the min ensures that points in the closest pair are pulled apart. This ensures that appearances in the preview are distant from each other and therefore exhibit a good variety. Note that contrary to E_C only points in $\mathcal{X}$ are considered.

Optimizing the objective function E_V is equivalent to solving the p–dispersion problem (section 2.4.3).

- *Smoothness*: We would like to order the representative appearances so as to produce a smooth map. This is achieved by minimizing the difference between neighboring appearances in $\mathcal{X}$. We *minimize* the objective function:

$$E_S(\mathcal{X}) = \sum_{p \in \mathcal{X}} \sum_{q \in \mathcal{N}_p} \mathcal{M}(I_p, I_q)$$

where $\mathcal{N}_p$ is the set of values within the 4–neighborhood of p in the 2D map $\mathcal{X}$. This may seem contradictory to variety. However, it only concerns direct neighbors in the map, while E_V considers the overall variety and ignores neighboring relationships.

Optimizing simultaneously for these three energies is challenging. Below we review possible strategies, their shortcomings for our scenario, and we propose a new algorithm.

Optimizing for E_C and E_V

As discussed in section 2.4.3, the heuristic proposed by Hochbaum and Shmoys [HS85] or the one proposed by Gonzalez [Gon85] are perfect to simultaneously solve the p–center and the p–dispersion problems. It is therefore ideal to use one these two heuristics to optimize E_C and E_V. However optimize only E_C and E_V selects representative samples but does not order them in a map.

Optimizing for E_S

As previously discussed, Shapira et al. [SSCO09] solve an ordering problem by using MDS to project representative appearances on the 2D plane then rearrange the projected appearances in a grid using a kd–tree ordering. This heuristic works well in many of our textures as shown in Figure 3.6.

The issue with Shapira et al. method is that it is highly sensitive to the shape of the 2D space generated by MDS and fails to produce a smooth result when it is too distant from a square grid. Figure 3.7 shows a failure case where strong discontinuities appear in the arranged grid.

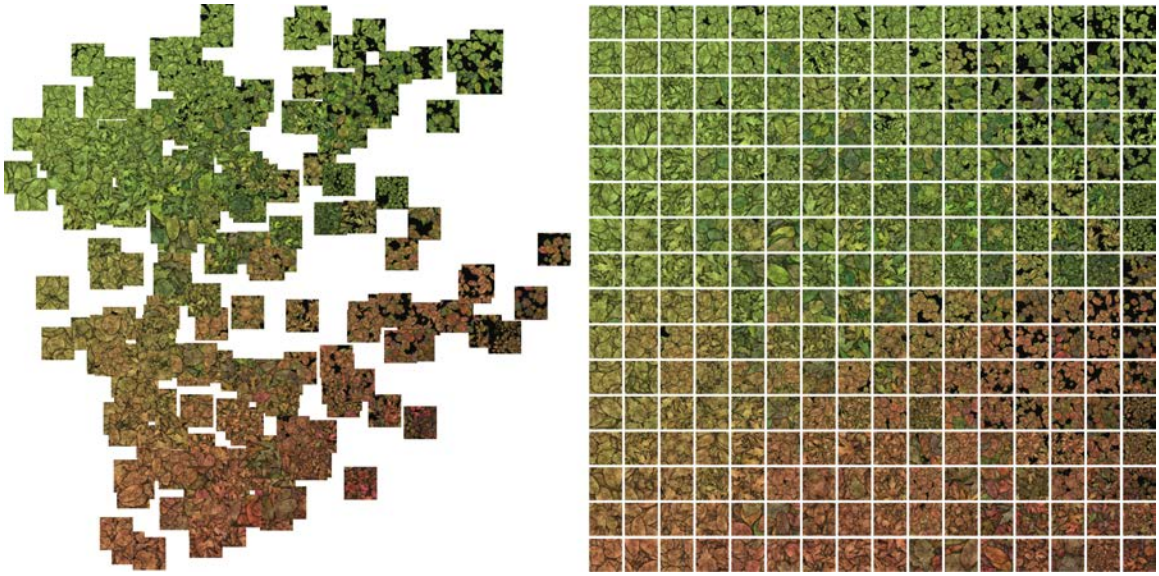

Figure 3.6: When the layout of appearances produced by MDS is close to a square grid then Shapira et al. arrangement produces a good result. The Hochbaum and Shmoys heuristic have been used to select representative appearances.

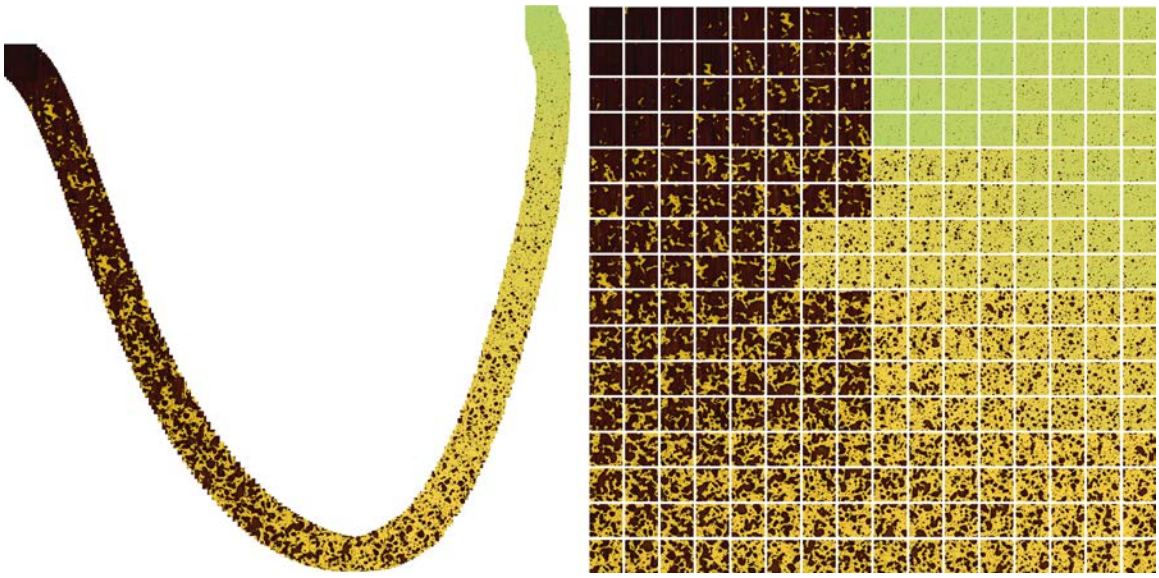

Figure 3.7: MDS fails to correctly layout appearances. MDS detects a one dimensional path embedded in the multidimensional appearance space. By projecting the one dimensional path in 2D the two ends of the path which are very different become neighbors in the grid after Shapira et al. re–arrangement. The Hochbaum and Shmoys heuristic have been used to select representative appearances.

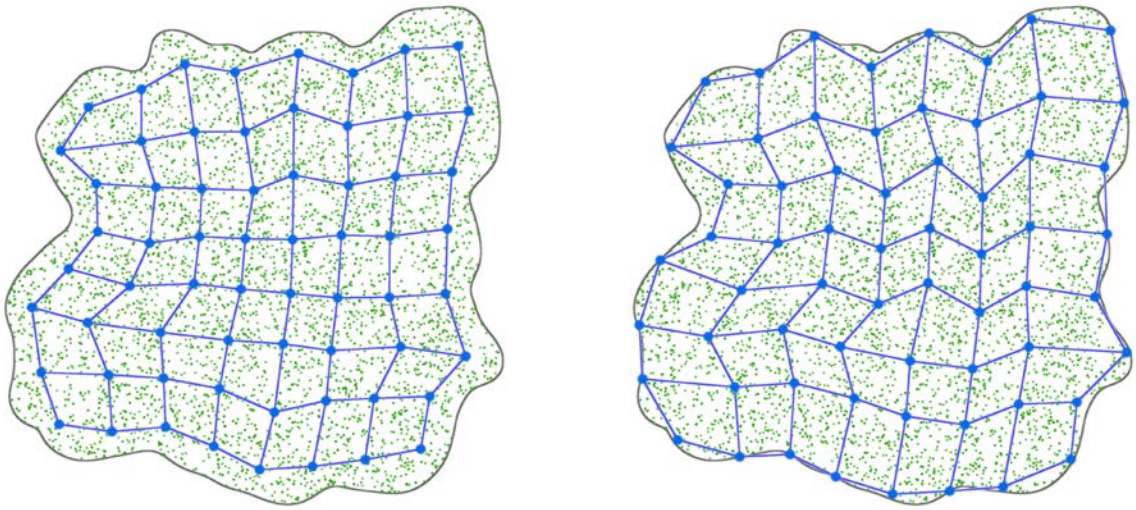

Figure 3.8: Green points are textures generated by $\mathcal{C}$, blue points are images generated by representative parameters. The connection between blue points reveals their relationship in the map $\mathcal{X}$. *Left:* Result from the SOM algorithm. *Right:* Result of our algorithm. Note how optimizing for variety brings points closer to the boundary, while preserving the good ordering of points within the grid. Note that on real cases the grid is much more distorted in appearance space.

Optimizing jointly for E_C, E_S

Sakamoto et al.[SKK04] solve simultaneously for the completeness and smoothness, in parameter space, using the self–organizing map [KSH01] (SOM). A SOM is a mapping from a high–dimensional space (i.e. $g(\mathcal{C})$) to a 2D map (i.e. $\mathcal{X}$). The algorithm starts from a random solution then iteratively updates $\mathcal{X}$ as follows:

$$\forall p \in \mathcal{C}, \forall s \in \mathcal{N}(argmin_{q \in \mathcal{X}}(\mathcal{M}(I_p, I_q))) \;\; s = s + \alpha(p - s)$$

Where α is a decreasing coefficient such as $1 > \alpha > 0$ and $\mathcal{N}(p)$ a neighborhood around node p in $\mathcal{X}$ which radius is decreasing. The SOM optimizes for the completeness energy while seeking to form an ordered (smooth) map [Hes99]. The resulting representative points are a linear combination of points in $\mathcal{C}$. When the subspace spanned by $\mathcal{C}$ has a complex concave geometry this can lead to undesirable results with points exiting the subspace of valid appearances. We avoid this by selecting representative points within $\mathcal{C}$ only.

Another important issue is that the SOM results in a lower than desired variety. As shown in Figure 3.8, the completeness energy favors representative appearances distant from the boundary of the subspace spanned by $g(\mathcal{C})$. This is expected since points minimizing the overall distance to $\mathcal{C}$ are also centroids of a centroidal Voronoi tessellation [DFG99]. Unfortunately, this means that some extreme appearances are not captured in the preview.

Figure 3.9 shows some extreme appearances obtained if only E_C is optimized. Similarly,

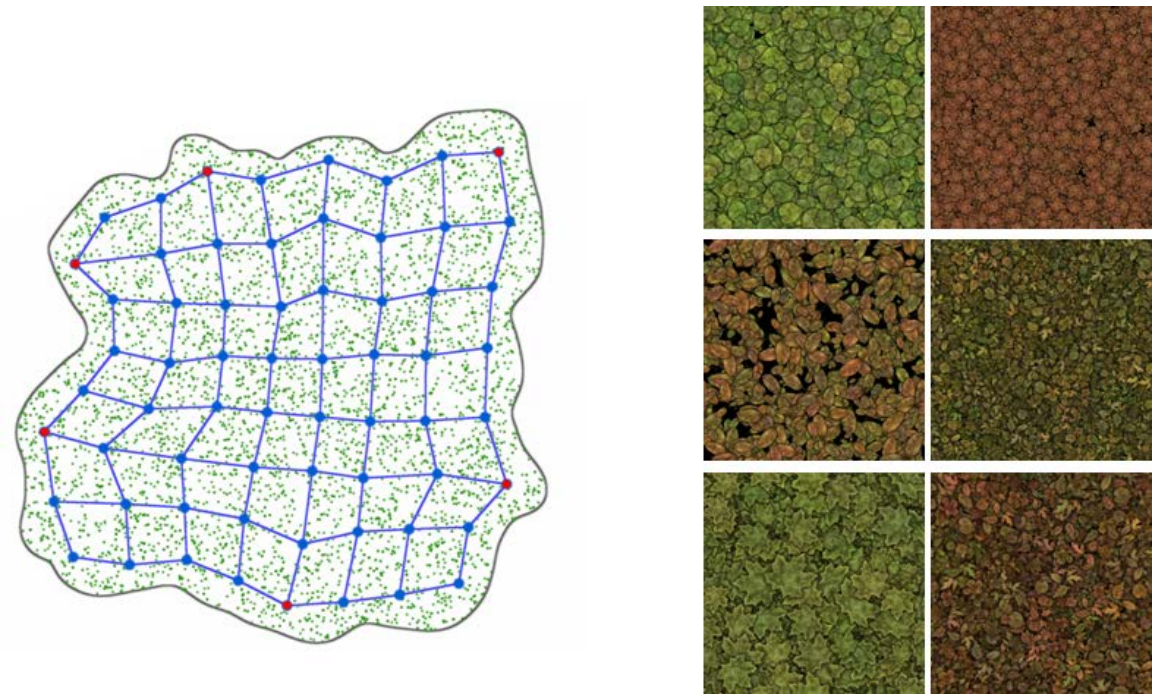

Figure 3.9: Extreme appearances if only E_C is optimized. *Left:* Illustration of the appearance space with some optimized extreme appearances highlighted in red. *Right:* Appearances corresponding to the highlighted red dots. The extreme appearances are found by seeking sample lying on the convex hull of the representative appearances.

Figure 3.10 shows some extreme appearances obtained if both E_C and E_V are optimized. These appearance will be missing if only E_C is optimized. It is interesting to note that from the few displayed thumbnails we can observe more variety when both E_C and E_V are both optimized as shown in Figure 3.11.

Since extreme appearances constitute important landmarks and are often very characteristic of the texture, we cannot afford to miss them. The variety term E_V is geared toward reducing this 'shrinking' of the representative points away from the boundary. However, the SOM algorithm cannot directly optimize for it.

Self–organizing map with variety

Our algorithm builds upon SOM but integrates the variety term E_V in addition to E_C and E_S. At each iteration we construct in every location s of $\mathcal{X}$ a cluster of samples noted $Cluster[s]$. These are the samples closest to $\mathcal{X}[s]$ in appearance space. We then iteratively select a representative sample in every cluster, in turn minimizing E_C then maximizing E_V.

First, we minimize E_C for the cluster by selecting the sample closest to all other cluster samples. Second, we maximize E_V for the cluster by selecting the sample furthest to the representative samples of all other clusters. The cyclic dependency on the representative samples is removed by updating the representative samples through interleaved sub–passes

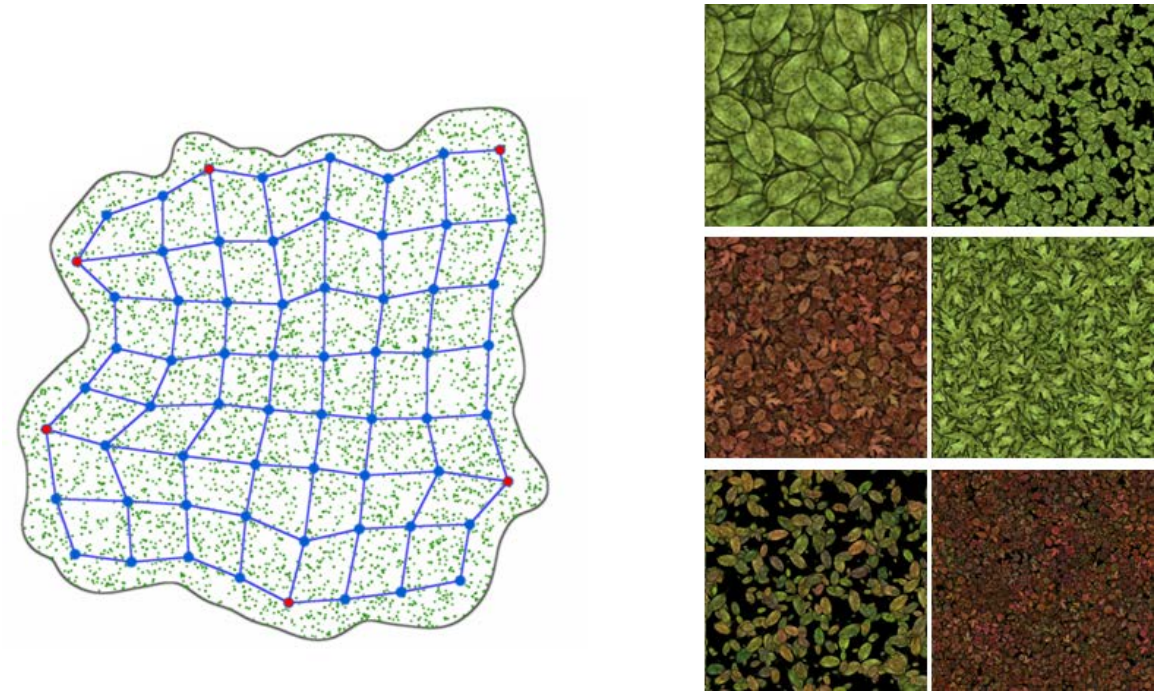

Figure 3.10: Extreme appearances if both E_C and E_V are optimized. *Left:* Illustration of the appearance space with some optimized extreme appearances highlighted in red. *Right:* Appearances corresponding to the highlighted red dots. The extreme appearances are found by seeking sample lying on the convex hull of the representative appearances. When E_V is included in the optimization the sought samples reach the appearance–space convex hull.

Algorithm 4 Computing $\mathcal{X}$

$r \leftarrow rMax$ // radius of a neighborhood $\mathcal{N}$ within $\mathcal{X}$
$t \leftarrow 0$ // iteration index
$subPassCounter \leftarrow 0$ // used during E_V optimization
initialize $\mathcal{C}$ with a dense sampling of $\mathcal{P}$
initialize $\mathcal{X}^t$ with samples randomly selected from $\mathcal{C}$
for $i \in \{1..N\}$ **do**
  // optimizing E_C
  $OptimizeCompleteness(\mathcal{X}, t, \lfloor r \rfloor)$
  // optimizing E_V
  for $j \in \{0..3\}$ **do**
    $subPassID \leftarrow mod(subPassCounter + j, 4)$
    $OptimizeVariety(\mathcal{X}, t, \lfloor r \rfloor, subPassID)$
  end for
  // change the order of sub–passes
  $subPassCounter \leftarrow subPassCounter + 1$
  // decrease r
  $r \leftarrow rMax^{\frac{-1}{N}} \times r$
end for

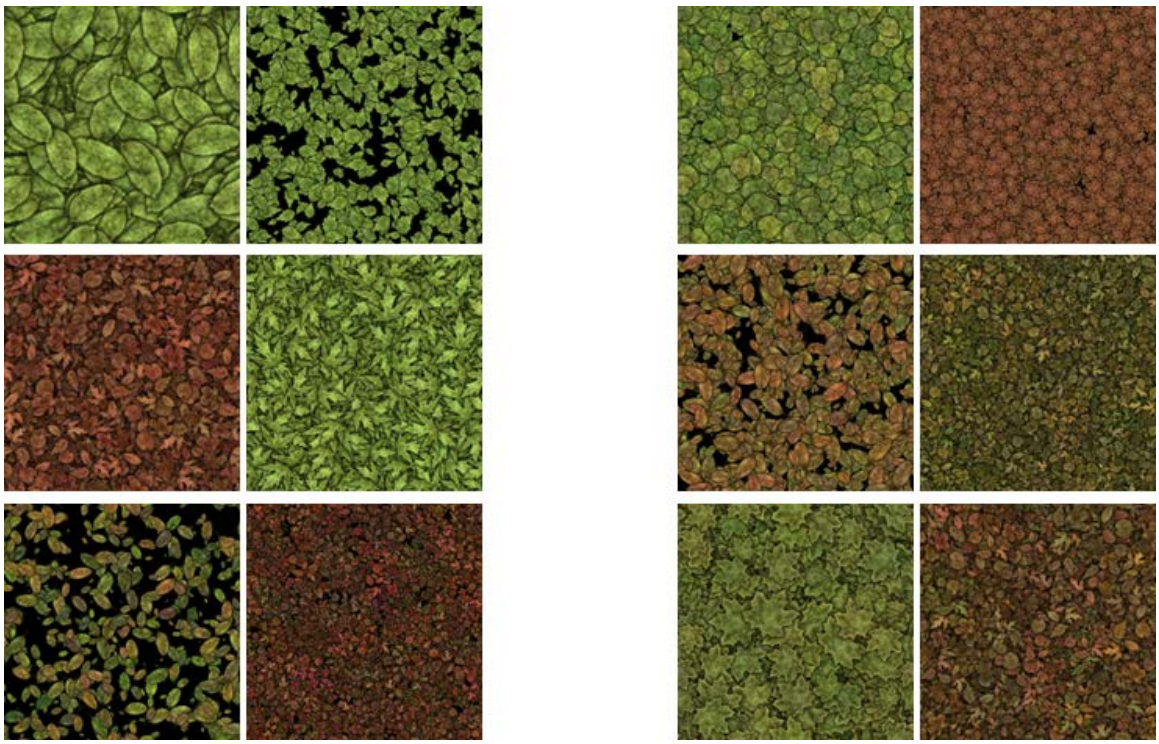

Figure 3.11: Comparing the extreme appearances of Figure 3.10 to the extreme appearances of Figure 3.9. *Left:* Extreme appearance if both E_C and E_V are optimized. *Right:* Extreme appearances if only E_C is optimized.

in the manner of [LH05]. The order of the sub–passes changes in every iteration to avoid any directional bias (Algorithm 4, line 15).

We iterate between E_C and E_V optimization, using more iterations for E_C at the beginning. This prevents variety to compete with smoothness before a good ordering is obtained (Algorithm 5, line 1).

Algorithm 4 describes our approach. Functions `OptimizeCompleteness` and `OptimizeVariety` are respectively detailed in Algorithm 5 and Algorithm 6. The function `processedAt(k, subPassID)` (Algorithm 6, line 12) returns true if k belongs to the set of coordinates used at sub–pass `subPassID`.

Quantitative results:

We measure the average energies on our entire collection of textures, for three different methods: Our approach, a standard SOM, and the Hoshbaum–Smoys heuristic combined to the arrangement method of Shapira et al (noted HSS). Results are given in Figure 3.12. We remap the energies so that the higher the bar, the better the result. For each criterion, bars are normalized between 0 and max.

As can be seen, our approach (yellow) offers the best compromise. It exhibits a better variety than the SOM (red) and much better smoothness than HSS (blue). Compared

Algorithm 5 *OptimizeCompleteness*$(\mathcal{X}, t, r)$

for $i \in \{1..r\}$ **do**
  $t \leftarrow t + 1$
  // clustering as in SOM
  initialize an empty grid $Clusters$ of size $W_{\mathcal{X}} \times H_{\mathcal{X}}$
  for all $p \in \mathcal{C}$ **do**
    $q \leftarrow argmin_{q \in \mathcal{X}^{t-1}} \mathcal{M}(I_p, I_q)$
    for all k $\in \mathcal{N}_r(q)$ **do**
      $Clusters[k] \leftarrow Clusters[k] \cup \{p\}$
    end for
  end for
  // select in every cluster, the sample minimizing E_C
  for all $k \in Clusters$ **do**
    $E_C^{min} \leftarrow +\infty$
    for all $p \in Clusters[k]$ **do**
      $e \leftarrow -\infty$
      for all $q \in Clusters[k]$ **do**
        $e \leftarrow \max(e, \mathcal{M}(I_p, I_q))$
      end for
      if $e < E_C^{min}$ **then**
        $E_C^{min} \leftarrow e$
        $\mathcal{X}^t[k] \leftarrow p$
      end if
    end for
  end for
end for

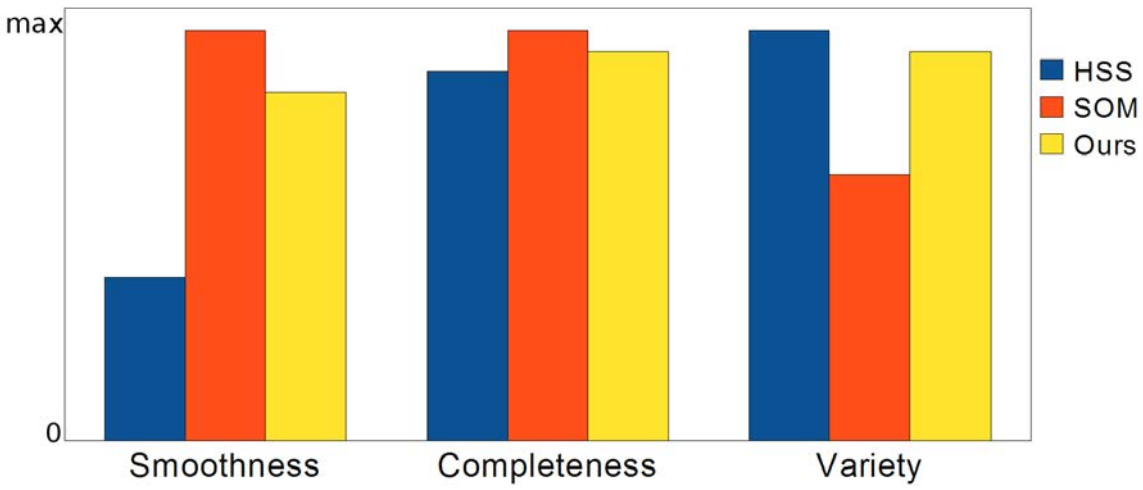

Figure 3.12: Energies for different optimization methods.

Algorithm 6 $OptimizeVariety(\mathcal{X}, t, r, subPassID)$

$t \leftarrow t + 1$
// clustering as in SOM
initialize an empty grid $Clusters$ of size $W_{\mathcal{X}} \times H_{\mathcal{X}}$
for all $p \in \mathcal{C}$ **do**
  $q \leftarrow argmin_{q \in \mathcal{X}^{t-1}\ p \neq q} \mathcal{M}(I_p, I_q)$
  for all k $\in \mathcal{N}_r(q)$ **do**
    $Clusters[k] \leftarrow Clusters[k] \cup \{p\}$
  end for
end for
// select in every cluster, the sample maximizing E_V
for all $k \in Clusters$ **do**
  if $processedAt(k, subPassID)$ **then**
    $E_V^{max} \leftarrow -\infty$
    for all $p \in Clusters[k]$ **do**
      $e \leftarrow +\infty$
      for all $q \in \mathcal{X}^{t-1}$ **do**
        $e \leftarrow \min(e, \mathcal{M}(I_p, I_q))$
      end for
      if $e > E_V^{max}$ **then**
        $E_V^{max} \leftarrow e$
        $\mathcal{X}^t[k] \leftarrow p$
      end if
    end for
  end if
end for

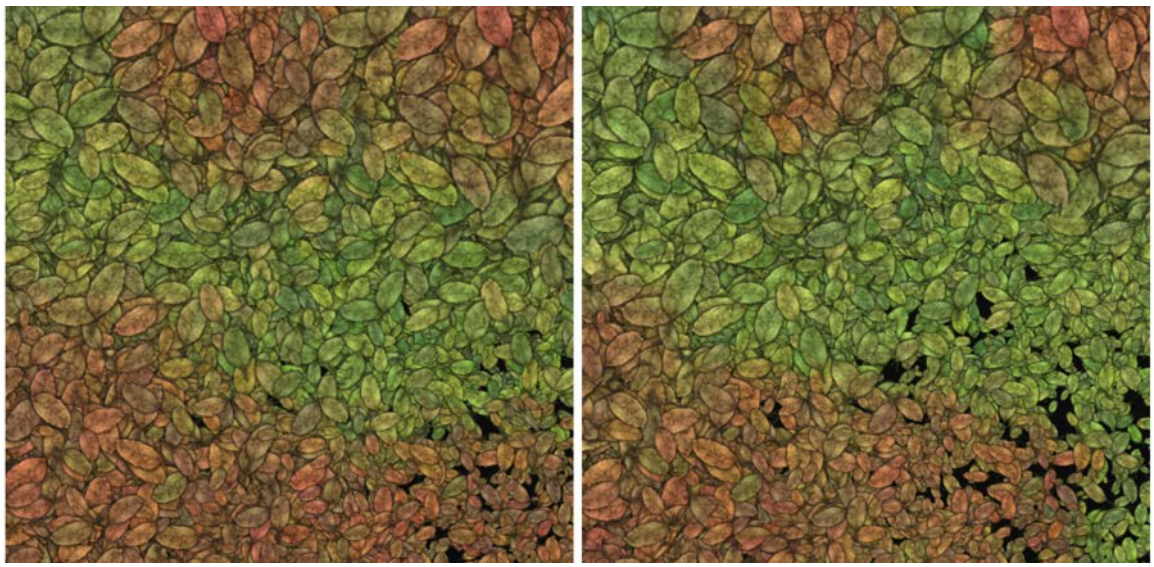

Figure 3.13: *Left:* A standard SOM. *Right:* A SOM optimizing completeness and variety

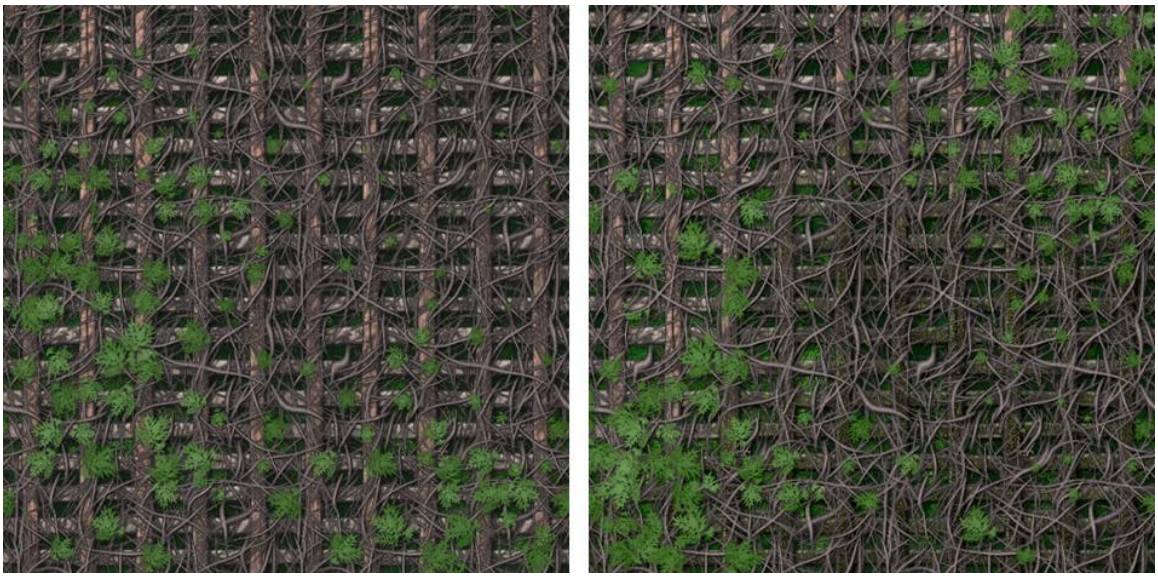

Figure 3.14: *Left:* A standard SOM. *Right:* A SOM optimizing completeness and variety

to SOM, the smoothness decreases slightly due to the increased variety. As shown in Figure 3.13 or Figure 3.14, the increased variety shows more appearances.

3.2.4 Fast texture comparisons

The metric $\mathcal{M}$ described in Section 3.2.2 is too slow in practice. The images must be re–generated many times along with their Gaussian pyramids. This adds up to impractical timings. Pre–computing $\mathcal{M}$ for all pairs of images is also impractical due to the large size of $\mathcal{C}$, so is pre–computing and storing all image pyramids.

Principal component analysis(PCA), keeping only the few first dimensions, seems well

suited. Unfortunately, the high dimensionality of the Gaussian pyramids makes it difficult to compute. Using PCA on patches or a down–sampled version of the textures unfortunately degrades the result.

We instead propose to rely on a compact signature, different for each procedure. It consists of the values of a small number of pixels located within the Gaussian pyramid. The signatures are computed once and stored.

The pixels are chosen in areas of high variations to captures localized changes (Figure 3.15). We estimate the spatial variations of the texture by computing the per–pixel variances across $\mathcal{C}$. We then sample the map of variances using Monte Carlo importance sampling with the estimator:

$$\hat{\mathcal{M}}(I_1, I_2) = \frac{1}{N} \sum_{i=1}^{N} \frac{||I_{p_{x_i}} - I_{q_{x_i}}||}{P(x_i)}$$

where x_i is a pseudo–random coordinate within the Gaussian pyramid, sampled from a distribution P proportional to the variance map.

Figure 3.15 compares our signature to PCA applied on patches of size 64^2. For each signature size the error with respect to the exact metric is averaged over a large number of texture–pair comparisons. Our signature achieves a good approximation using only a small signature size in contrast to PCA. In practice, we use a signature of size $\max(W_{\mathcal{T}}, H_{\mathcal{T}})$.

3.2.5 Tiling

In order to outline the texture variations while preserving the layout of features, we tile the procedural texture over the summary. The user can control the number of tiles by changing the resolution of the procedural texture. For example if the user chooses a resolution of 256^2 for the generated textures and a resolution of 512^2 for the output preview, the procedural texture would be tiled twice in each direction. However, for the preview to succeed in visually carrying the possible outcomes of the procedural texture, the number of tiles appearing in the summary must be chosen carefully. If some texture features do not repeat often enough, many variations could be missed. This is for instance the case with the eye texture shown in Figure 3.16. However, tiling the texture a large number of times quickly reduces resolution, hiding details as shown Figure 3.17. It is thus important to strike the right balance between number of tiles and loss of resolution.

To help the user choose the resolution of the generated textures given a fixed resolution for the preview, we propose an automatic algorithm that suggests a resolution that produces a tiling with a good balance between the amount of useful information in the preview and the quality of the generated textures.

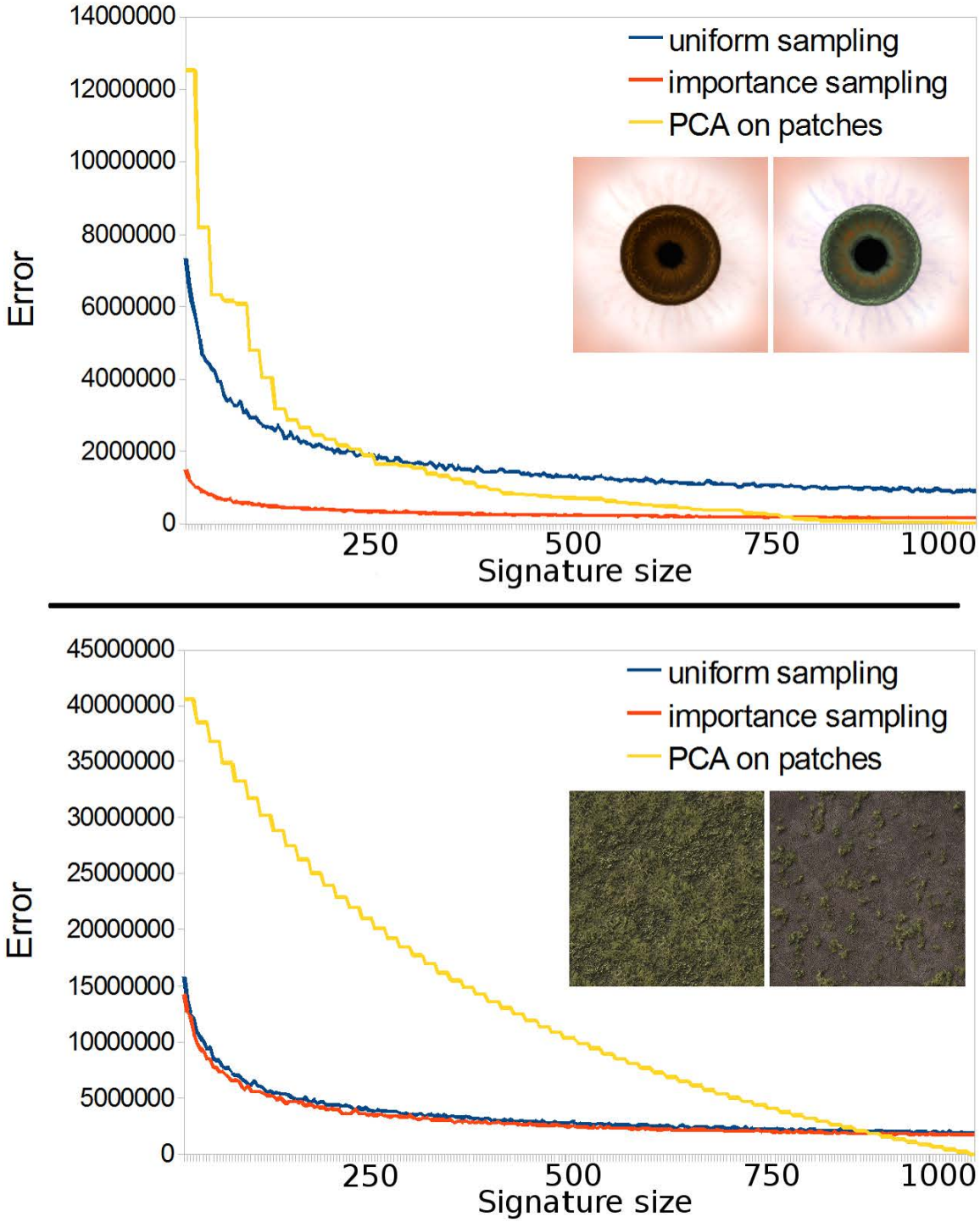

Figure 3.15: Comparison between our signature, a random signature and PCA applied on patches. *Top:* A texture with localized changes. *Bottom:* A texture with global changes.

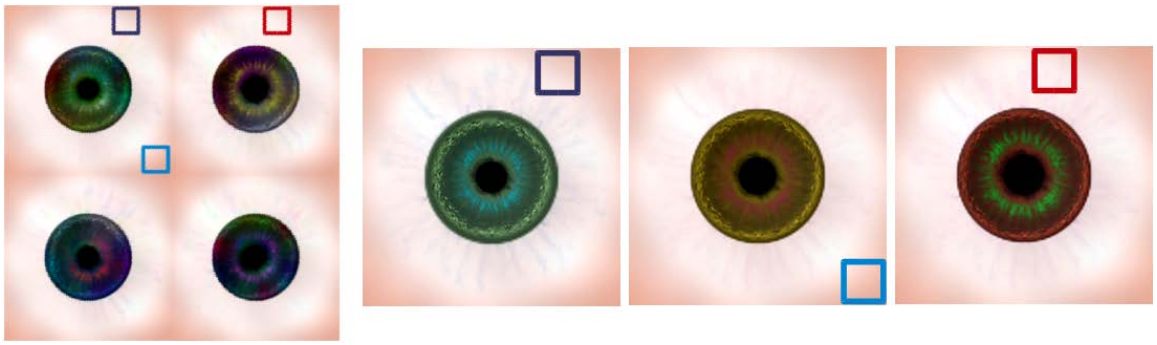

Figure 3.16: The eye texture summarized in a 2×2 tiling (left). The three outlined patches do not show any useful information. The parameters should reveal new eye colors within the iris, however none of these patches contain a part of the iris.

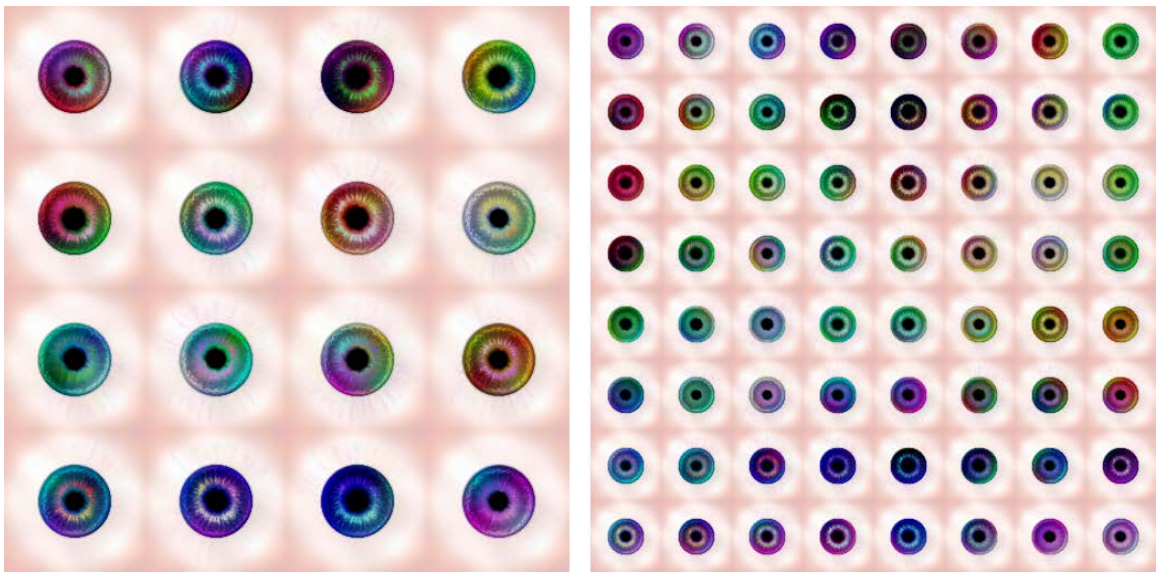

Figure 3.17: The eye texture summarized at 512^2 resolution with 4×4 and 8×8 tiles. The 4×4 tiling shows more information than the 2×2 tiling of Figure 3.16. The 8×8 tiling shows even more information but also misses some fine scale details due to loss of resolution. Our automated algorithm tries to select the 4×4 tiling since it gives a good compromise between amount of information and quality.

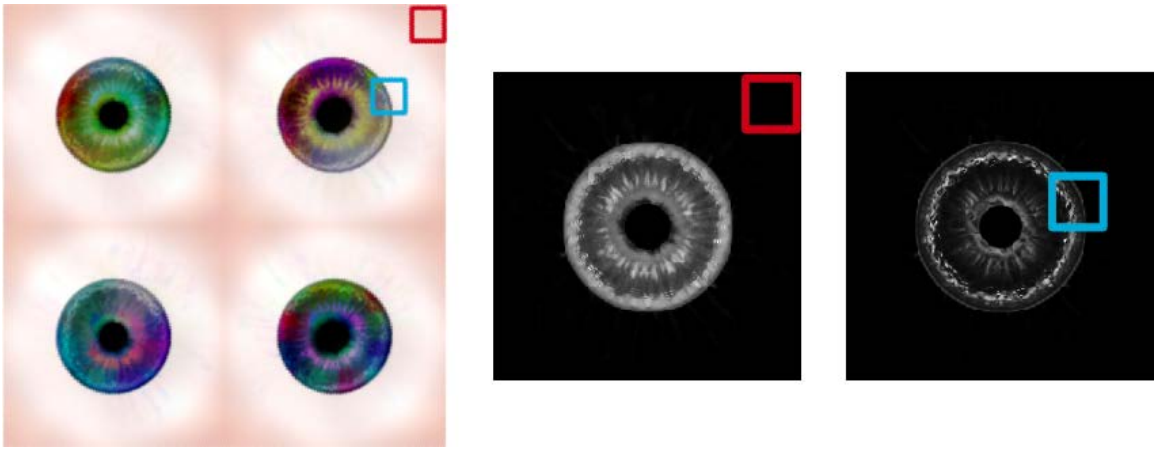

Figure 3.18: *Left:* The eye texture summarized in a 2×2 tiling. *Right:* The local difference maps for the red and blue patches. The red patch shows no useful information.

Patch score The algorithm computes, for each patch, a measure of the amount of information it reveals at different levels of details. We name this value the *multi–resolution patch information score*. Patches with a low score show no useful content with respect to the parameter variations they contain.

For the patch corresponding to position (x, y) in $\mathcal{X}$ we compute a local difference map (see Figure 3.18) as follows: We generate all images corresponding to a neighborhood around (x, y). We then compute compute the per–pixel absolute differences between the patch and the other images at the same patch location. We repeat the process for all level of details within the Gaussian pyramid of the images using the same patch size. We therefore have a local difference map for each pyramid level. The local difference map at each pyramid level is then transformed into a score by summing up all difference values.

To better interpret the the multi–resolution patch information score, Figure 3.19 illustrates two situations where in one case the score is higher at the coarse level of detail and in another case the score is higher at the fine level of detail.

Selecting the best tiling By analyzing the multi–resolution patch information score our aim is to automatically select the level of detail that increases the amount of information in the preview without too much loosing fine scale details. To do so, we consider the information scores across all patches at all level of details. The sum of all scores at each level is a first good indication of which level should be selected. However, a small area at a fine level of detail could have a very high score and the tiling selection will be mostly affected by this small area. For a more appropriate selection, we consider the standard deviation of all scores in each level of detail (see Figure 3.20). With too few tiles, the standard deviation tends to be large as patches have either a very low or a very high score. As the number of tiles increases, the standard deviation becomes smaller due to the more

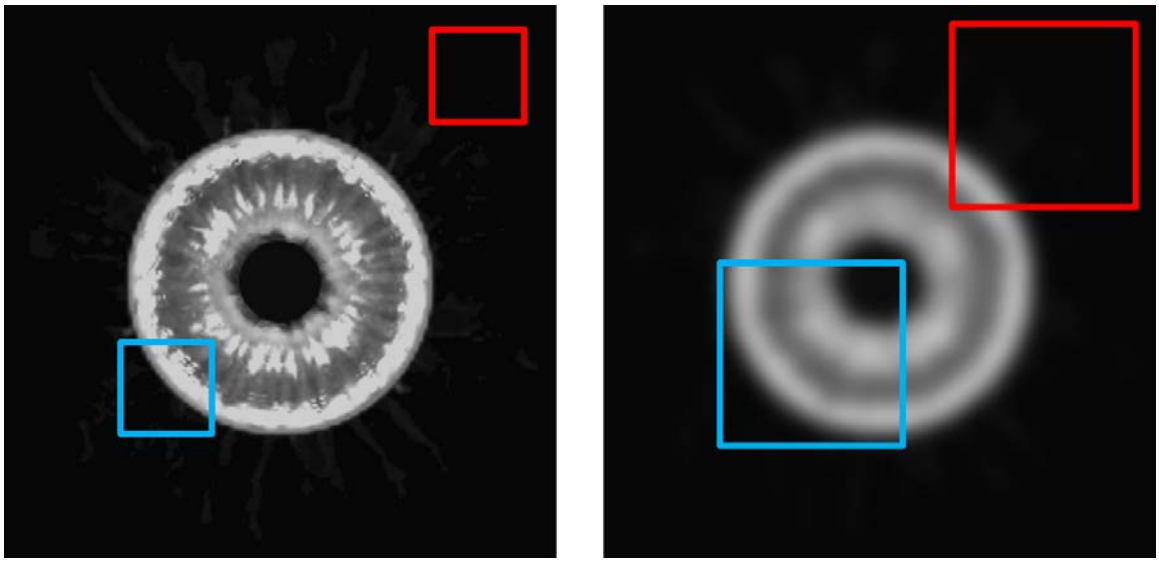

Figure 3.19: The multi–resolution patch information score. *Left:* A local difference map corresponding to the finest level of detail. *Right:* The same local difference map at a coarser level of detail (the displayed size of this map is doubled for clarity). *Red patches:* The red patch on the left corresponds to a region without any useful information. Its score is zero. The red patch on the right contains some useful information and its score is therefore higher. Note that this patch has the same size as the red patch on the left (recall that the displayed size of the difference map is doubled). *Blue patches:* About 30% of useful information occupy the area of the blue patch on the left. About 70% of useful information occupy the area of the blue patch on the right. However, due to the loss of energy within the coarse level of details, summing up the 70% of useful information occupying the blue patch on the right turns out to give a lower score than the sum of the 30% of useful information occupying the blue patch on the left.

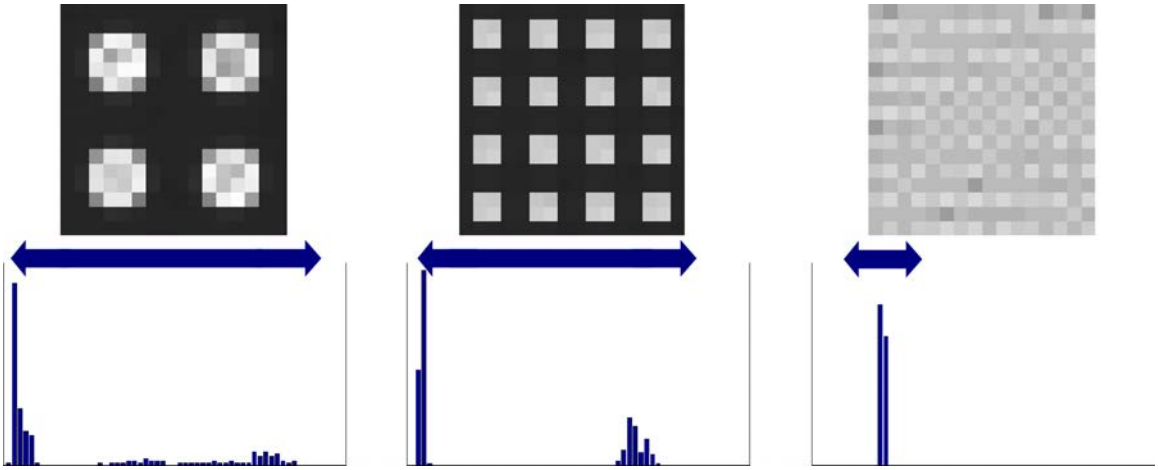

Figure 3.20: *Top:* Information score for all cells. *Bottom:* Histogram of information scores. The arrows indicate the standard deviation. (The last image has been contrast enhanced for better visualization).

even distribution of scores. We therefore normalize the sum of all scores at each level with the scores standard deviation.

We however noticed that the standard deviation decreases non linearly and at the coarsest level of details its value quickly approaches zero. To avoid selecting the coarsest level of details we ignores the score if the the standard deviation is less than a certain threshold. We empirically selected this threshold by experimenting its effect on our data set of textures.

3.3 Scented Sliders for Procedural Textures

The most widely spread interface for procedural texture parameters selection is a set of sliders with the name of the parameters next to them. Such interfaces have proven efficient for setting graphics parameters [KP10]. However, they are difficult to use for someone discovering a new procedural texture: Contrary to other applications, the set of parameters and their meaning varies strongly from one texture to another (compare parameters of Figure 3.21, Figure 3.33 or Figure 3.34). Since there is no visual clue – besides the parameter name – of what each slider exactly does, it is hard to predict and understand the influence of each parameter without trying a large number of different settings. In addition, dozens of parameters are exposed and many of them change the effect of each other. This quickly becomes a bottleneck for users exploring the possibilities offered by different textures. For instance, the parameters 'Cereals type' and 'Cereals color' in Figures 3.21 would not have any effect if the parameter 'Milk level' is set to its highest value.

Our slider previews solve this problem by illustrating the effect of each parameter *for the current settings*. In particular, our sliders dynamically adapt to user interactions by refreshing all the slider previews if one parameter changes. This way, our visual slider

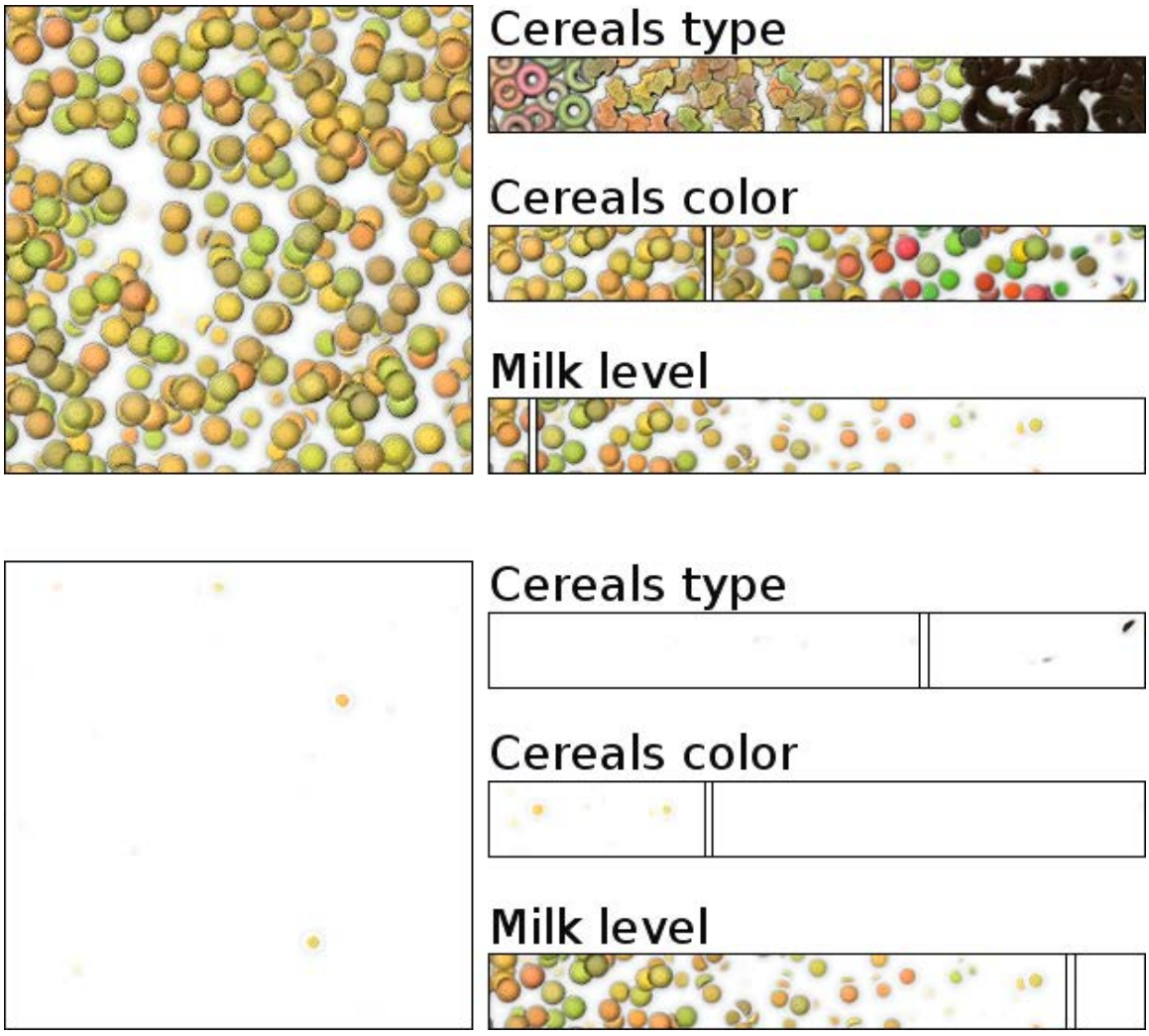

Figure 3.21: Two settings of the 'Cereals' texture with visual sliders control. *Top:* The 'Milk level' parameter has a low value. *Bottom:* The 'Milk level' parameter has a high value, masking the effect of other parameters.

previews make the user aware that the 'Milk level' is the only parameter that could affect the settings of Figures 3.21, bottom.

Designing such visual slider previews presents a number of challenges:

- The slider previews should indicate in an obvious manner all the *changes* that will occur in the texture when the slider is manipulated.
- The pixel space is very limited: We cannot afford to display a large image below each slider, or the navigation from one slider to the next would quickly become tedious. This is especially problematic when the slider impacts small features in the texture.
- The continuous refresh of slider previews imposes a fast synthesis algorithm.

3.3.1 Overview

To synthesize visual slider previews following the requirements set above, we need to concentrate in a small pixel area the important information: in this case the parts of the image that are effectively changed by the slider. We understand this as an importance–driven image–relayout problem: Starting from an image showing all changes we would like to discard non–essential information that remains unchanged when moving the slider. This is done by the following steps:

- A large image is formed by tiling the texture a number of times horizontally. The large image is then divided into a grid of patches (Figure 3.22, ***(b)***). Each column of patches reflects the appearances of one setting of parameters. The parameter corresponding to the slider is increasing from left to right while others keep their current values. This means that the leftmost column of patches reflects the appearances produced if the parameter is at its lowest value. The rightmost column reflects the appearances produced if the parameter is at its highest value.
- An importance map is computed (Figure 3.22, ***(c)***) by giving a score to each patch depending on its variance during slider manipulation. Patches that vary very often will get high scores.
- The set of patches is reduced by removing patches with low scores (Figure 3.22, ***(d)***). To limit distortions and preserve the layout of patches, seam–carving is applied on the importance map to remove patches rather than pixels.
- The remaining patches (Figure 3.22, ***(e)***) are stitched together using patch–based texture synthesis (Figure 3.22, ***(f)***). Due to some distortion produced by seam–carving, we allow for small changes in the patch positioning during synthesis.

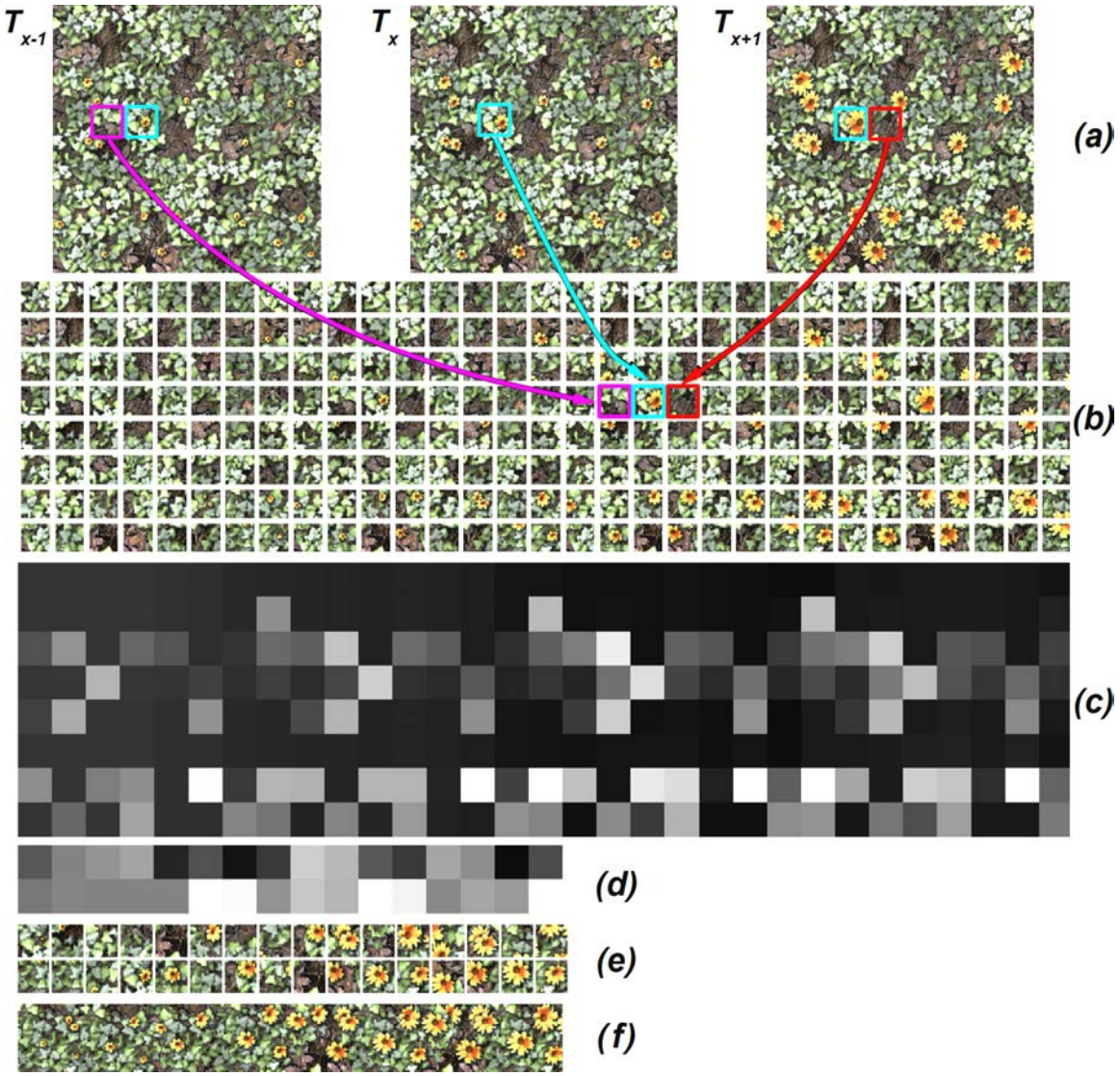

Figure 3.22: Algorithm overview: ***(a)*** Texture samples from which patches are extracted. ***(b)*** Initial grid G of patches. ***(c)*** Per-patch importance map I. This map will be carved to remove patches with low scores. ***(d)*** Carved map I. Bright pixels correspond to patches with high scores. ***(e)*** The remaining patches. Notice how the flowers, which are the most impacted, dominate the patches. ***(f)*** Resulting slider after the synthesis step.

3.3.2 Generating the grid of patches

We note $V \in [0,1]^N$ the parameter vector of size N describing the current state of the texture. There are N sliders and we are interested in synthesizing the slider indexed s ($s \in \{1..N\}$). The values in V will be fixed with the exception of $V[s]$ that varies linearly from 0(left) to 1(right). We construct a grid of patches G of size $W_G \times H_G$ in which every element $G[x,y]$, $(x,y) \in \{1..W_G\} \times \{1..H_G\}$, is a patch of size $W_P \times H_P$. The patch at $G[x,y]$ is contained in the texture T_x that has a size of $W_T \times H_T$ and parameters v such as $v[i] = V[i]\ if\ i \neq s$ and $v[s] = \frac{x-1}{W_G-1}$ otherwise. The upper–left corner of this patch is located at coordinates $((x \times W_P)\%W_T\ ,\ (y \times H_P)\%H_T))$ in T_x. Figure 3.22 ***(a)*** and ***(b)*** highlights the grid creation.

3.3.3 Importance map

The importance map I gives a score to every patch in G (Figure 3.22, ***(c)***). This score is high for patches with high visual impact. We note $I[x,y]$ the score of patch $p = G[x,y]$ that belongs to texture T_x. We note (x_0, y_0) the upper–left coordinates of the patch p in T_x. The score $I[x,y]$ of patch p is computed as follows:

$$I[x,y] = \sum_{t=1}^{W_G} \frac{e^{-\frac{||x-t||^2}{2\sigma^2}}}{1 + \sum_{i=x_0}^{x_0+W_P} \sum_{j=y_0}^{y_0+H_P} ||T_x[i,j] - T_t[i,j]||^2}$$

The denominator corresponds to a per–pixel difference between the patch p contained in texture T_x and the patch having the same location as p but contained in another texture T_t. As an example, in Figure 3.22, ***(a)***, the blue patch in texture T_x is compared to the two blue patches in T_{x-1} and T_{x+1}. The numerator controls the contribution of each texture T_t in the score. Textures with very different parameters will contribute less. These contributions can be further customized using the scalar $\sigma \neq 0$. A small σ^2 emphasizes the possible variations when little changes occur to the slider cursor. If σ^2 goes to infinity then all textures T_t will have a same contribution making the score independent from the cursor position. We typically set $\sigma = 1$ and, in order to speed up the algorithm we ignore textures with very low contributions (when the numerator is close to zero).

3.3.4 Patch carving

We shrink the grid of patches G with seam–carving [AS07], removing vertical/horizontal seams of patches. Carving at the patch–level rather than pixel–level avoids texture distortion but produces discontinuities along the boundaries of the remaining patches. The carving is done simultaneously on G and I according to the scores contained in I. The shrunk version of the map I is shown Figure 3.22, ***(d)***. It mostly contains patches with

high scores.
In principle, the carving could require to change the mapping from the slider to the parameter, as non–linear relayout may occur. However in practice we never noticed a bias strong enough to justify remapping the parameters.

3.4 Synthesis

Once the mapping obtained, we need to synthesize the preview image. Our goal is to ensure that the preview is everywhere visually similar to images generated by the parameters in $\mathcal{X}$. To do so we map the procedural texture over the preview, tiling it if necessary then use by–example texture synthesis to fill the preview with the corresponding appearances. Because we target interactive interfaces we need a fast and robust by–example texture synthesis algorithm. In section 2.3.2 we have seen that fast algorithms exist for by–example pixel–based synthesizers. These synthesizers exhibit a high degree of parallelism and map well to GPUs. Unfortunately, they often fail to preserve structures from the exemplar without the user specifying additional feature information. Furthermore pixel–based synthesizers are not well suited for synthesis from multiple exemplars as it is the case for our previews (each and every sample in our preview corresponds to a different exemplar). In Figure 3.23 we use a fast GPU pixel–based synthesis algorithm. The algorithm consists of an implementation of Lefebvre and Hoppe algorithm [LH05] with a search phase accelerated with Patch–match [BSFG09]. We use dithering to transition from one appearance to another by randomly select an exemplar proportional to the distance to the nearest sample in $\mathcal{X}$. The result however does not perfectly capture original appearances and the dithering produces unwanted noise. A feature distance map would have solved these two issues but it is very difficult to compute a good feature distance map automatically with current image processing and analysis techniques.

On the contrary to pixel–based synthesizers, patch–based synthesizers are better at capturing and preserving structural patterns. They can also naturally synthesize from multiple exemplars. However, they require relatively slow algorithms to layout the patches and stitch them together. In the remaining of this section we will use patch–based texture synthesis. We will consider that a fast patch–based synthesizer exist. Later in Chapter 4 we will describe a fast parallel patch–based synthesizer that runs entirely on the GPU and that is suitable for interactive applications like the sliders.

To use a patch–based synthesizer we divide the preview into patches, typically of 32^2 pixels. The patch size should correspond to the texture features for best quality. The parameters for the image of each patch are obtained by sampling $\mathcal{X}$ in the patch center. We then generate the part of the texture for the patch, as illustrated in Figure 3.24, left. Note that some procedural engines may not be able to directly produce the sub–part of the

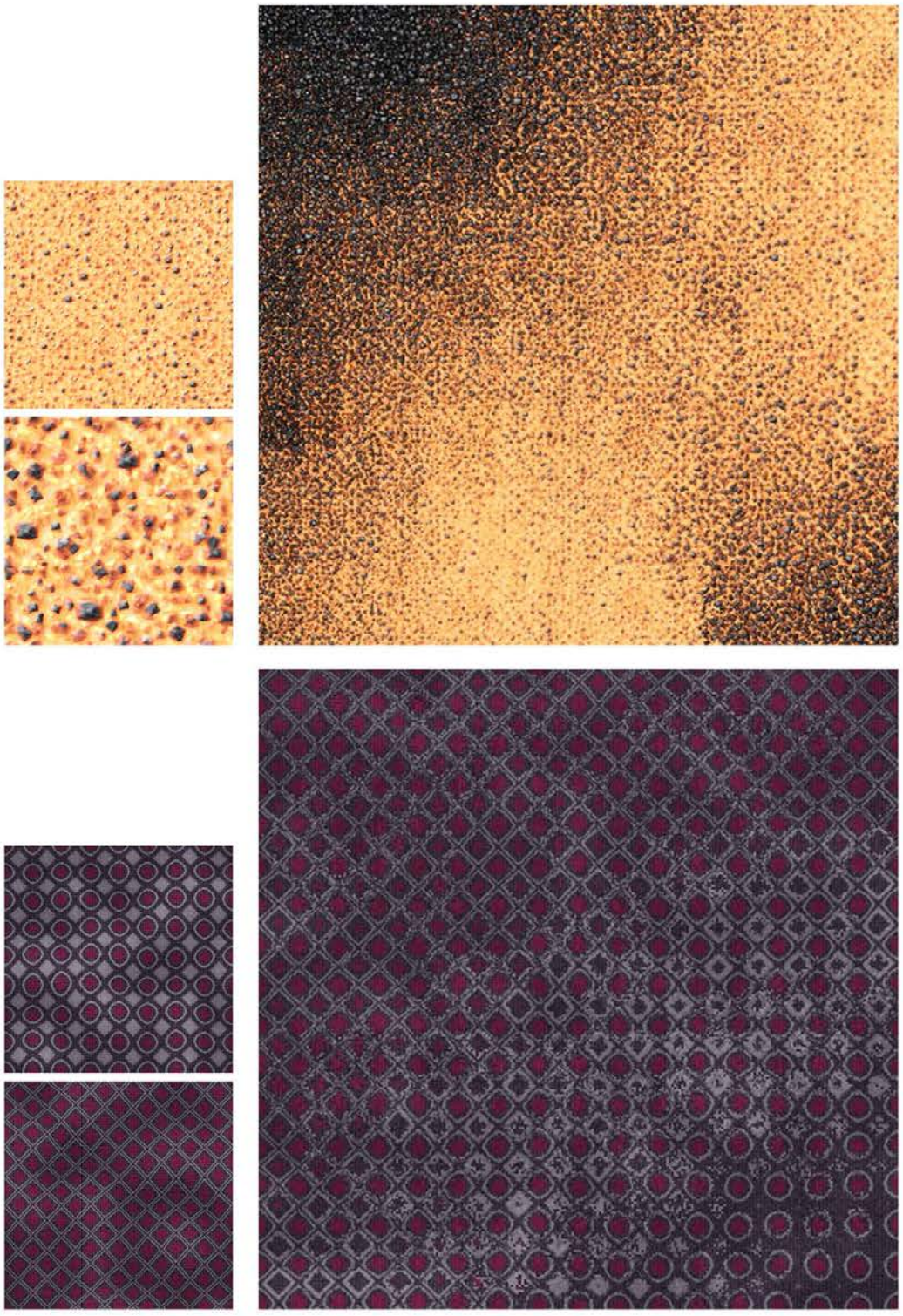

Figure 3.23: Pixel–based synthesis fails to preserve structures and smoothly transition from one appearance to the other.

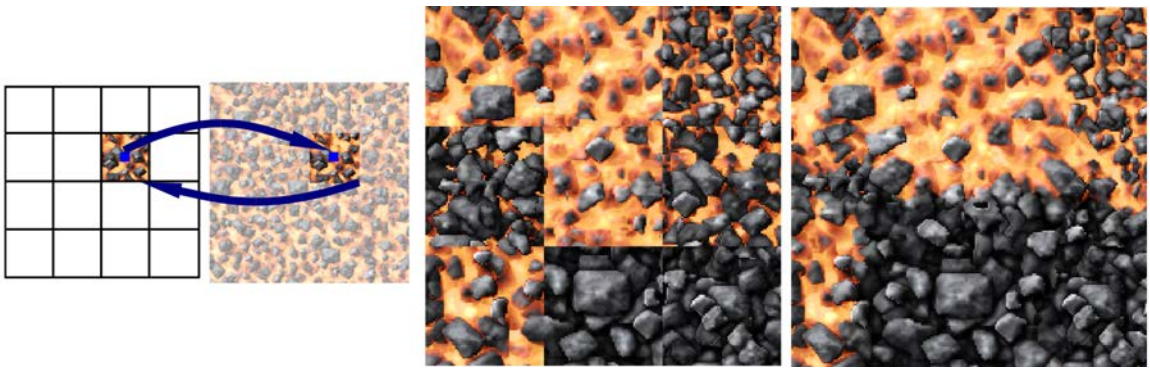

Figure 3.24: *Left:* Each patch is obtained by sampling the parameter map, generating the texture, and cropping at the corresponding location. *Middle, right:* We optimize the fine positioning of the patches and their boundaries so as to hide abrupt transitions.

texture corresponding to the patch, in which case we generate the entire texture and crop.

Due to changes of appearance through parameters, the boundaries of the grid cells appear (see Figure 3.24, middle). We apply a stitching step, we use the patch–based synthesizer to optimize the boundaries between cells as well as the fine positioning of the texture patches.

The result is a continuous summary image, where the texture appears tiled but exhibits spatial appearance variations, as illustrated in Figure 3.24, right and on the final synthesized preview of Figure 3.25.

Interestingly this approach allows the user to manually paint a map $\mathcal{X}$ to produce spatially varying textures as shown in Figure 3.26.

3.5 Texture palette

The procedural texture preview is useful to display, in a limited pixel space, a visually pleasing static overview of the possible appearances. This lets users quickly understand the content of the procedure, and conveniently explore many of them in a database of procedures. In this section we extend the procedural texture preview to make an interface where a user can conveniently select an appearance by clicking in the preview which now acts as a *texture palette*. In this texture palette we also allow the user to hover the mouse over the palette and display the texture that corresponds to the parameters at the mouse position in $\mathcal{X}$. In order to show a continuous transition of the displayed texture while the user is hovering the mouse we cannot rely on simple point sampling in $\mathcal{X}$.

To obtain a smooth continuous transitions, we consider $\mathcal{X}$ as covering the unit square and access it through normalized coordinates in $[0..1]^2$ and bi–linear interpolation as is usual for textures. In order to use bi–linear interpolation directly in parameter space, we expect the parameters to induce progressive changes in the texture, i.e. a small change of a

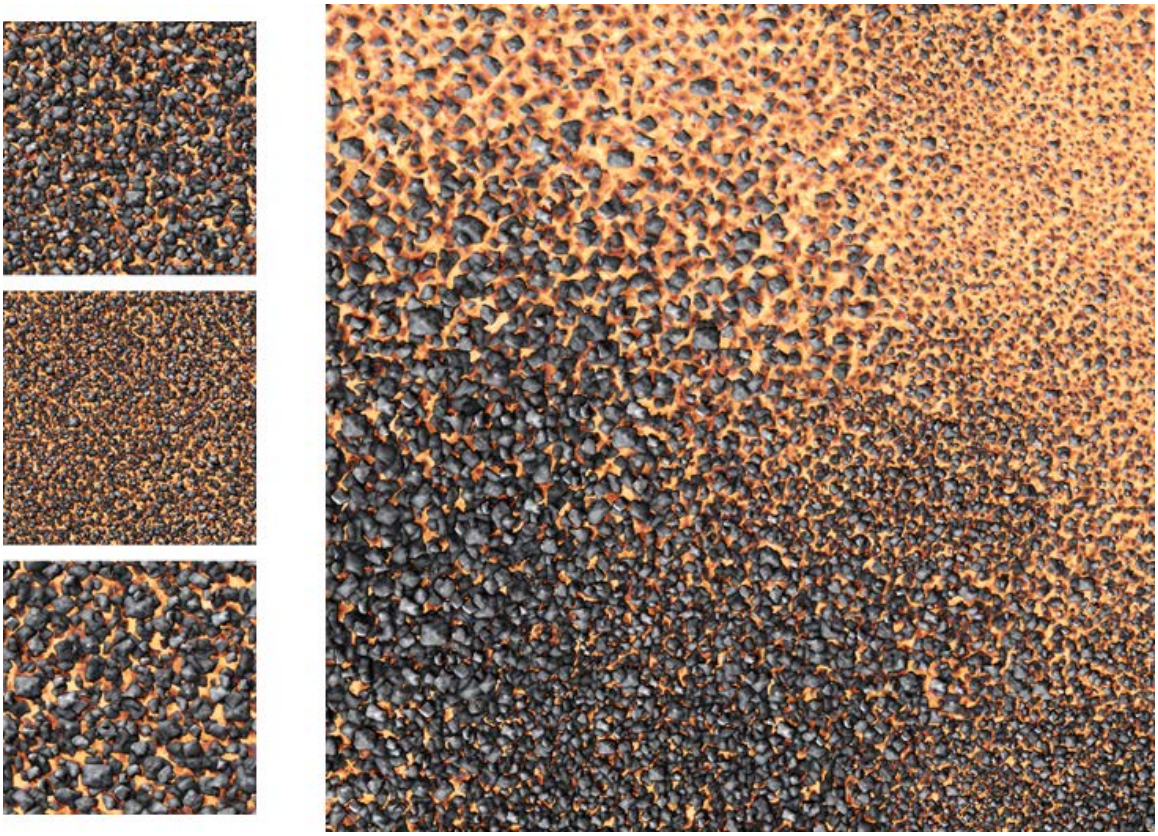

Figure 3.25: The final synthesized coal texture.

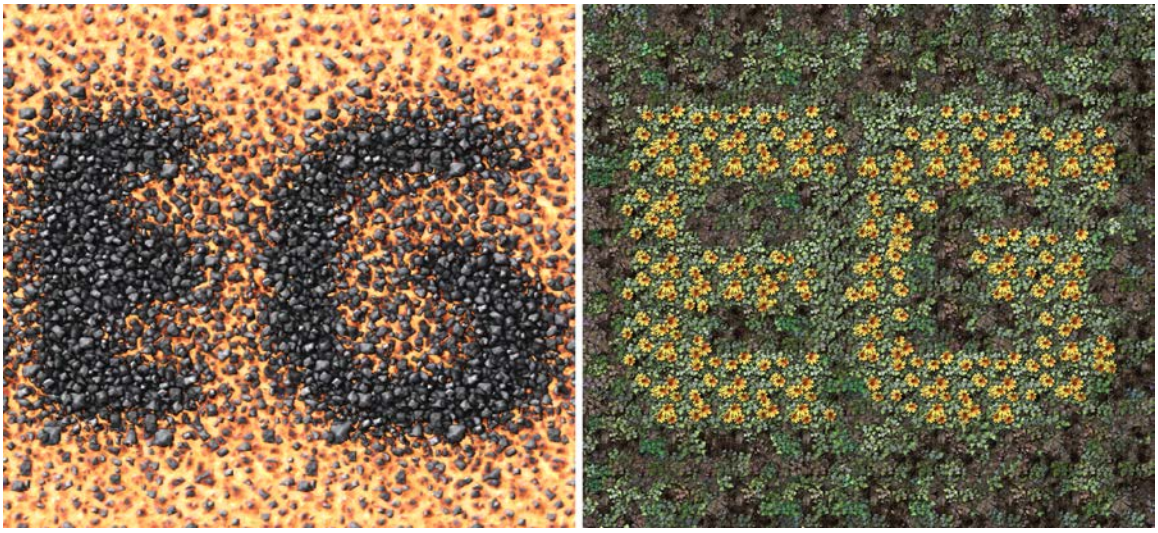

Figure 3.26: Spatially varying texture under user control.

parameter should either impact few pixels or all pixels by a small amount. This is the case for most procedural textures authored carefully by an artist and this includes the texture package we are using in this thesis. For textures not enforcing this assumption we work entirely in appearance space. We apply bi–linear interpolation on the appearance signatures surrounding the current mouse coordinates to obtain the blended signature I_b. Next, we search in $\mathcal{C}$ for the most similar signature to I_b then display the texture corresponding to this sought signature.

The palette allows for a quick selection of an appearance but as shown by Kerr and Pellacini [KP10], accurate parameters adjustment is better done with sliders. We therefore augment the palette with our scented sliders. When the user select and appearance in the palette the sliders are automatically updated with the parameters of the selected appearance. Conversely, when the user manipulates the sliders the cursor in the palette is updated by finding in $\mathcal{X}$ the closest sample to the current sliders setting.

Figure 3.4 shows a screen–shot of the palette interface.

3.6 Results

3.6.1 Procedural texture preview

In this section we first show some procedural texture preview results. These results are computed on Intel E5520 2.26 GHz CPU. The set $\mathcal{C}$ contains 2^{15} samples generated in approximately 6 minutes per procedural texture. The map $\mathcal{X}$ is computed in 1 minute for a size of 16^2. Note how the time is dominated by the generation of the textures. We normalize all parameters in $[0, 1]$ for processing. Integer parameters are correctly handled since we build the preview in appearance space.

Figure 3.27 and Figure 3.28 show previews obtained for different procedures. We can observe that the algorithm groups the visually dominant appearances in smoothly ordered regions. In Figure 3.27 (a), the bottom–right region contains cold rocks, with the temperature increasing as we move towards the top–left corner. The same effect is observed in Figure 3.28 (a) where the amount and size of flowers increase between the bottom–right and top–left regions. The leaves density, a parameter with less visual impact, increases between the bottom–right and the bottom–left. Other parameters such as leave colors or spacing can be observed, even though they do not dominate the ordering. Note that despite few parameters, textures such as the one in Figure 3.27 (b) exhibit a large variety of appearances – working in appearance space allows to properly capture them.

In Figure 3.29 we constrain the algorithm to very small previews of size 128^2 – only 16384 pixels. Note how they still capture most appearances. Such small previews are very useful to browse through a database of procedures.

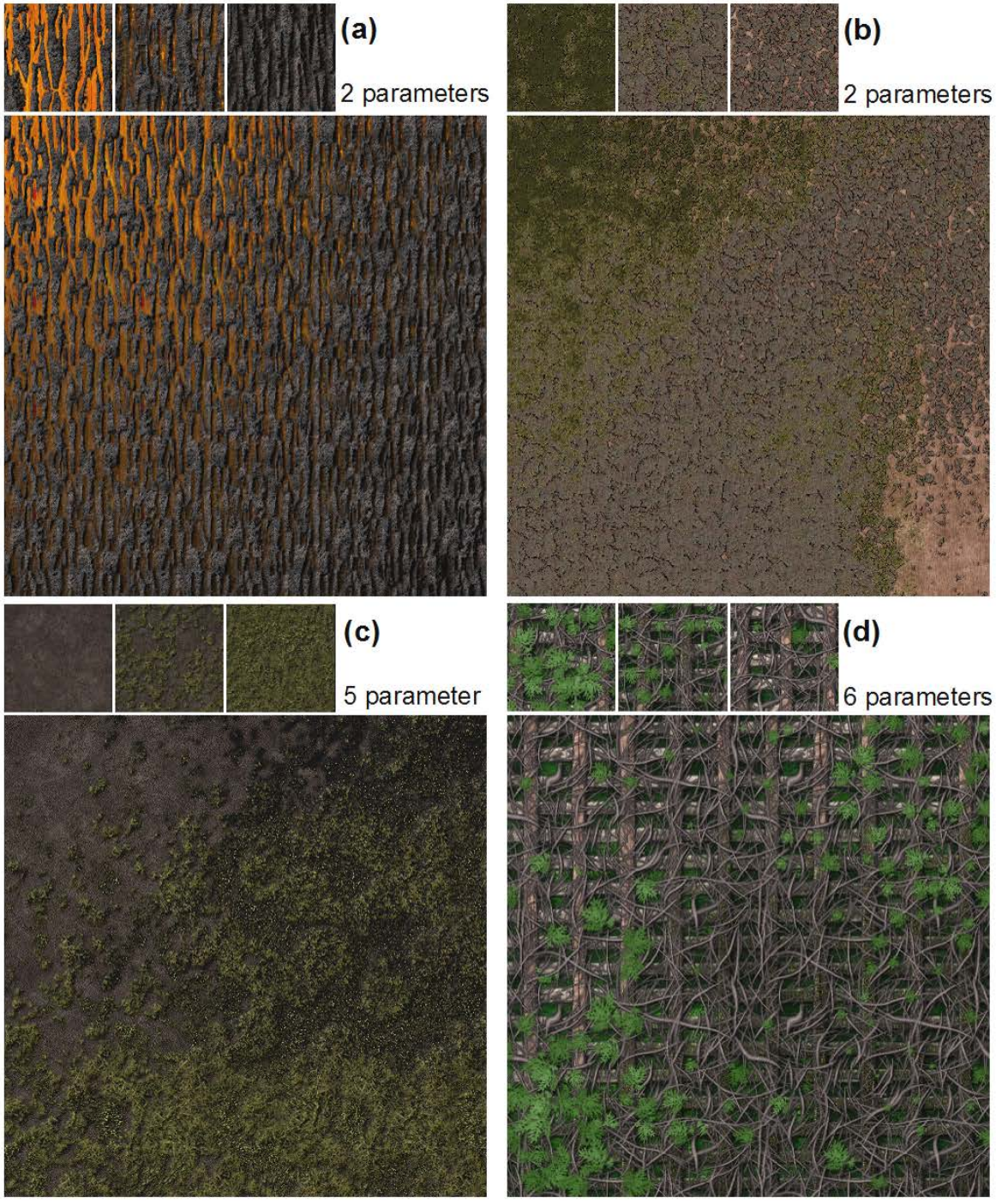

Figure 3.27: Selection of previews computed from procedural textures. The small images show random thumbnails of the textures.

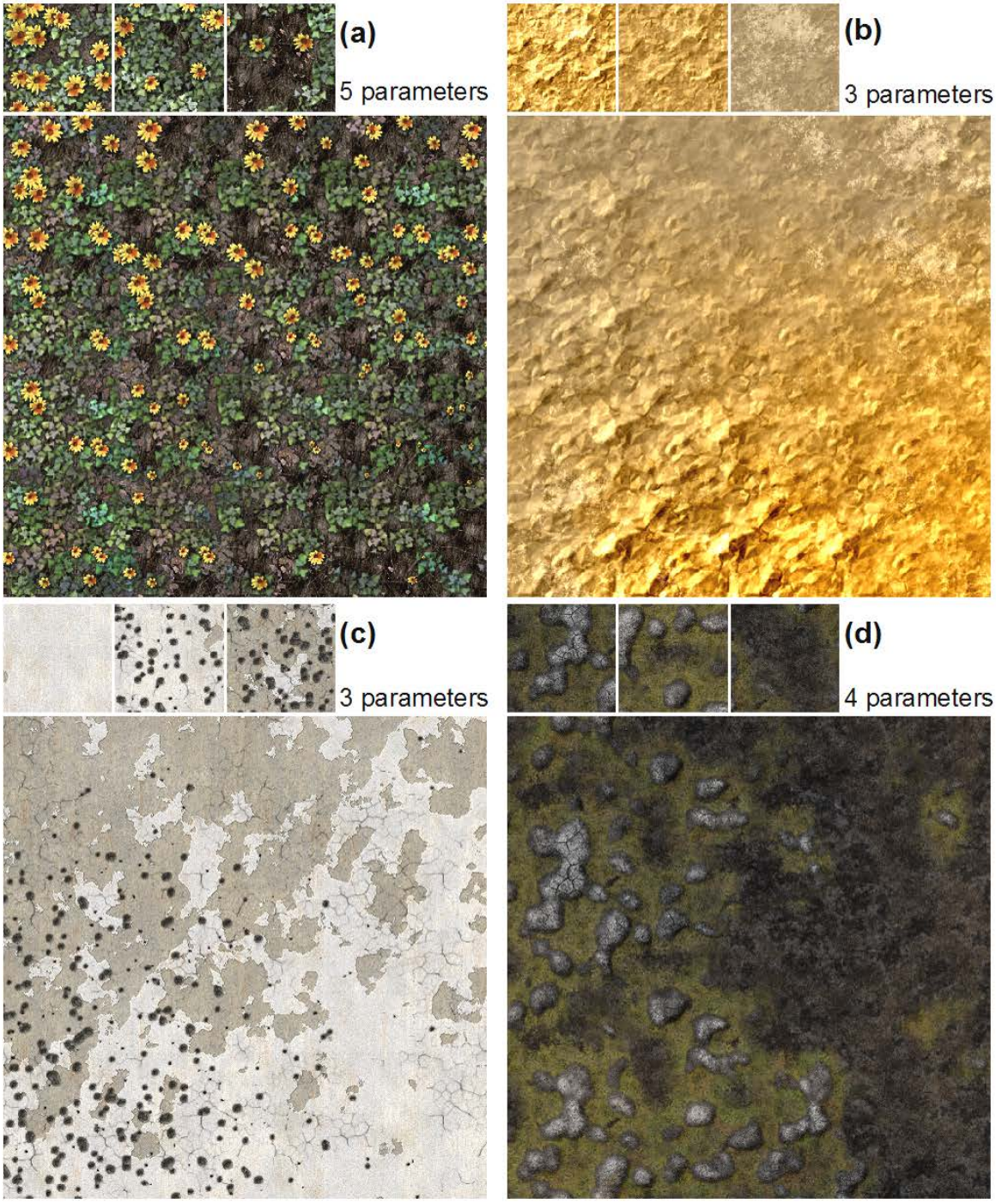

Figure 3.28: Selection of previews computed from procedural textures. The small images show random thumbnails of the textures.

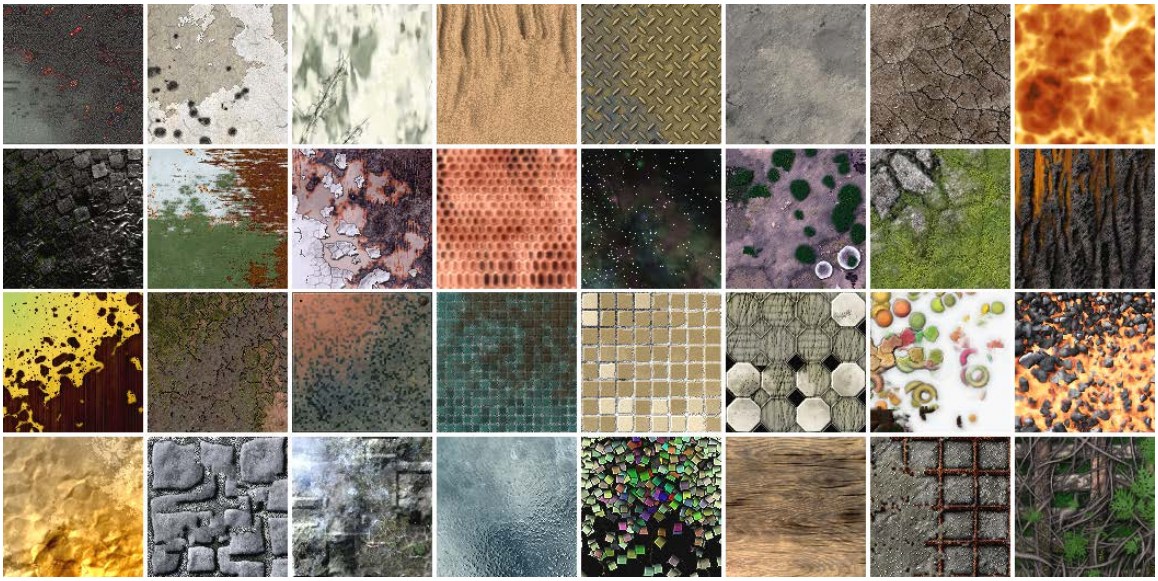

Figure 3.29: A band of 128^2 texture previews (all textures are shown in more details in the supplemental material).

User study

We conducted a user study to compare our previews to grid of thumbnails. We verify whether users observe more variety in our previews, as well as whether they find them more visually pleasing. While subjective, this last criterion is important for our application of previewing. The study is composed of three tasks:

Task 1 : Our previews are compared to projected previews where we intentionally reduce the variety. We do this by projecting parameters in a random sub–space. This serves as a sanity check.

Task 2 : Ordered grids of thumbnails are compared to random grids of thumbnails. Both grids contain the same variety as they are sampled from $\mathcal{X}$ – in one case the order is destroyed on purpose. All thumbnails are cropped to reveal changes, as in [SLBJ03].

Task 3 : Our previews are compared to ordered grids of thumbnails. The map $\mathcal{X}$ is different in each case.

Figure 3.30 shows an example of thumbnail grids and preview with reduced variety. The tasks are repeated for nine different textures. Each time a (random) left/right pair for a same texture is presented to the user for comparison. For each pair, we ask which contains more appearances, and which is visually more pleasing.

We had a total of 22 users, mostly researchers and students from our university. The results of the study are shown in Figure 3.31.

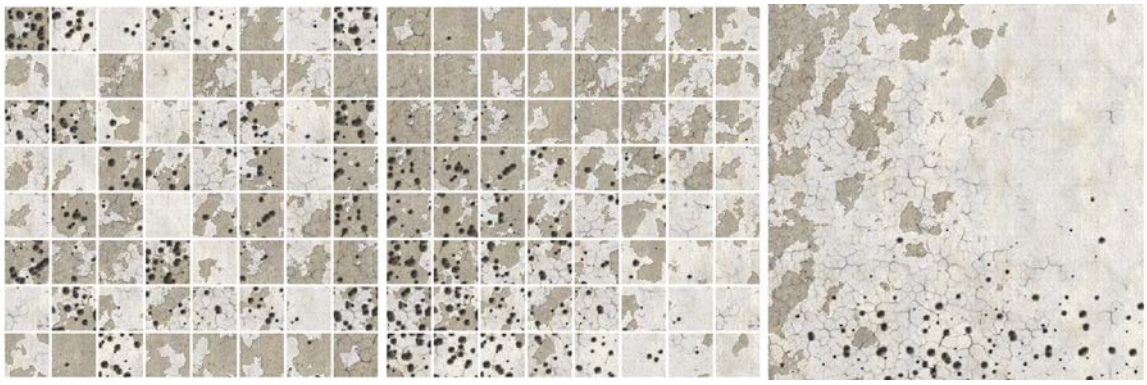

Figure 3.30: *Left:* Unordered thumbnails. *Middle:* Ordered thumbnails. *Right:* Projected preview. Note that there are no black impacts on the unpainted wall, contrary to the full preview shown in Figure 3.28 (c).

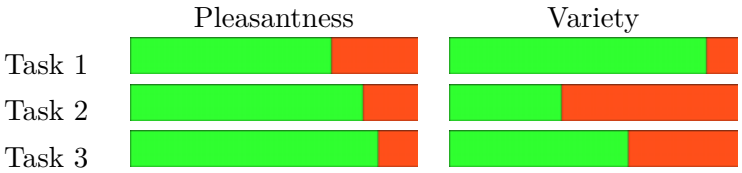

Figure 3.31: Color codes : *Task 1:* Green is 'Ours', red is Projected preview'. *Task 2:* Green is 'Ordered thumbnails', red is 'Unordered thumbnails'. *Task 3:* Green is 'Ours', Red is 'Ordered thumbnails'.

Task 1: 89% of the users have perceived the lack of variety in the projected previews. This confirms their ability to properly discriminate whether previews contain more appearances.

Task 2: 81% of the users found ordered thumbnails visually more pleasing. Surprisingly 61% found random thumbnails more varied, even though both use the same sets of parameters. We believe that the contrast between neighboring thumbnails increases the perception of variety.

Task 3: Users clearly preferred our previews over grids of ordered thumbnails: 86% found them visually more pleasing and 62% found them more varied. Note that our previews indeed contain more variations since the patch size is smaller than the thumbnails. This is a key advantage of previewing with a continuous image.

3.6.2 Scented sliders

We now show some slider preview results.

Figures 3.32 shows the benefit of our patch summarization method. In this figure, the most varying elements of the texture (the green stars) represent small features that are

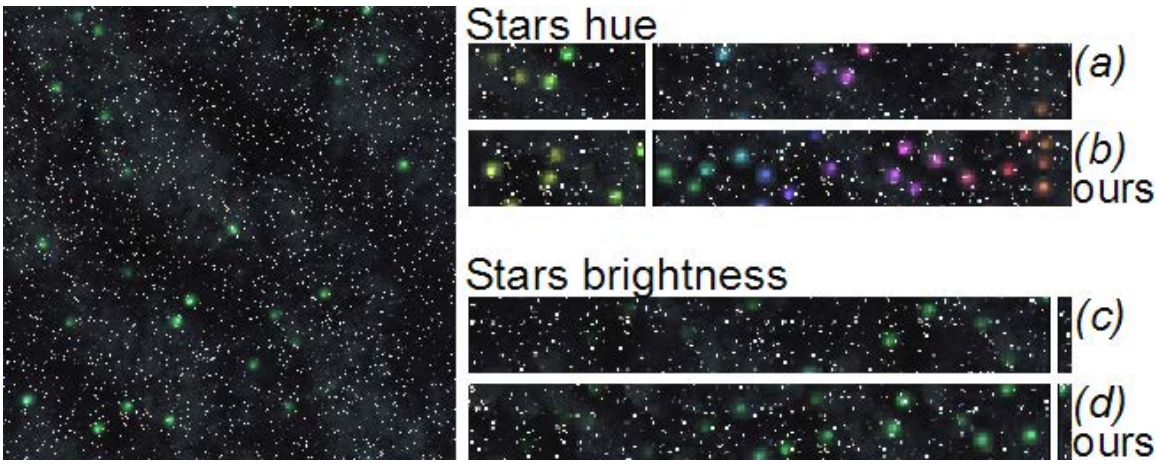

Figure 3.32: A procedural texture with two different types of sliders controlling parameters. (*a*) *and* (*c*)*:* Sliders generated by cropping the best horizontal strip from a larger initial slider. (*b*) *and* (*d*)*:* Sliders generated by our method.

far from each other. Sliders (*a*) and (*c*) are generated with a naive algorithm that starts from a large slider preview having the same width as the final slider and the same height as the procedural texture. It then crops the horizontal region that maximizes the overall variation. Sliders (*b*) and (*d*) are generated using our method. These sliders reveal more variation and give a better insight to the user.

Figure 3.33 shows an example containing texture and color parameters. Generating one slider for this texture takes on average 151 milliseconds. This timing is dominated by the generation of the 16 textures required to build the slider.

Figure 3.34 shows another example of sliders for a more complex texture. In this case, an average of 355 milliseconds is required to generate one slider.

Figure 3.35 shows how the previous change when the user changes the settings.

3.7 Limitations

Parameters introducing spatial layout changes on structured content are not handled well by our approach. This is for instance the case of changing the number of bricks in a wall. Our simple synthesizer fails to preserve the structure as shown in Figure 3.36 and Figure 3.37, top. Arguably, these parameters cannot be shown in a meaningful continuous manner, and are thus incompatible with our continuous previews.

One possible work–around is to let the artist authoring the texture decide to show these particular parameters as thumbnails. We included this possibility in our sliders. In this mode, a set of thumbnails is automatically generated by selecting high variance regions as thumbnails and ordering them on the slider Figure 3.37, bottom.

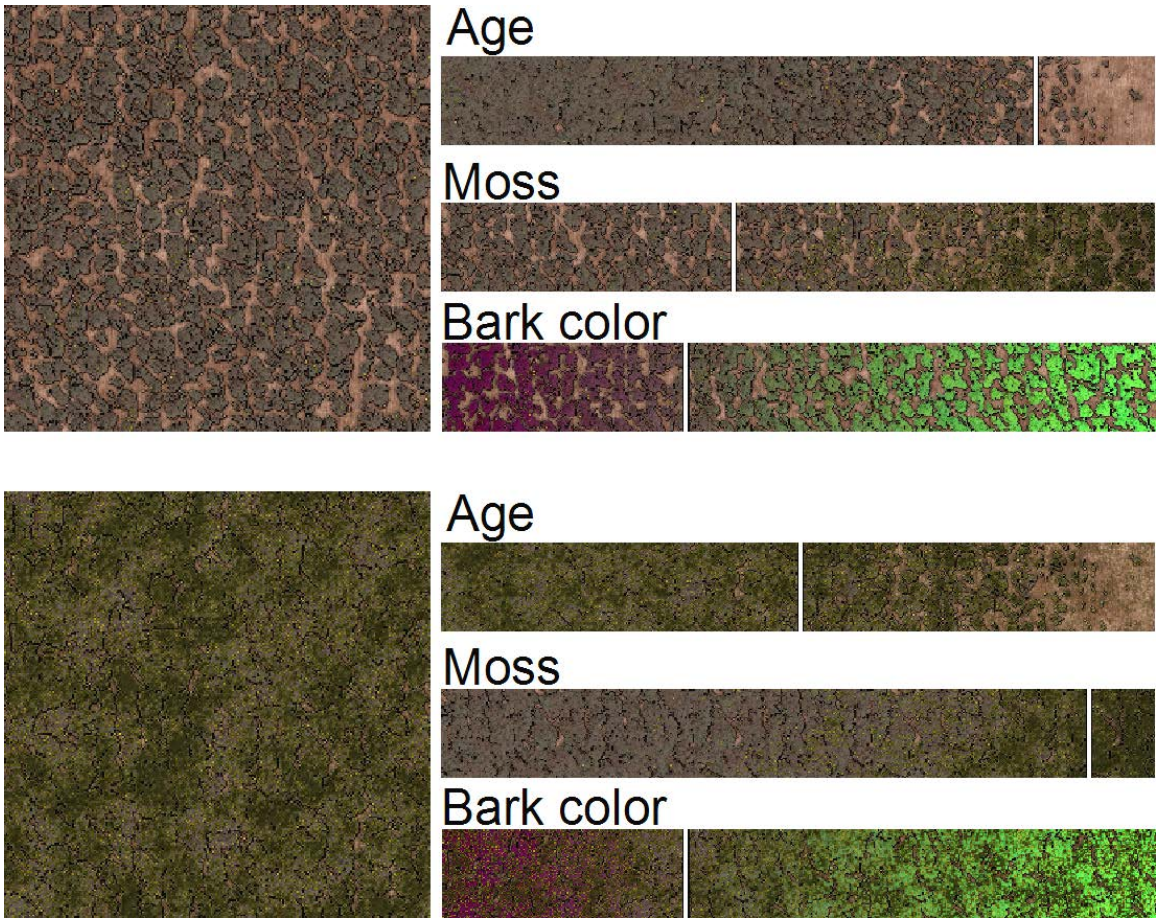

Figure 3.33: Two different settings of the 'bark' texture with visual sliders controlling parameters.

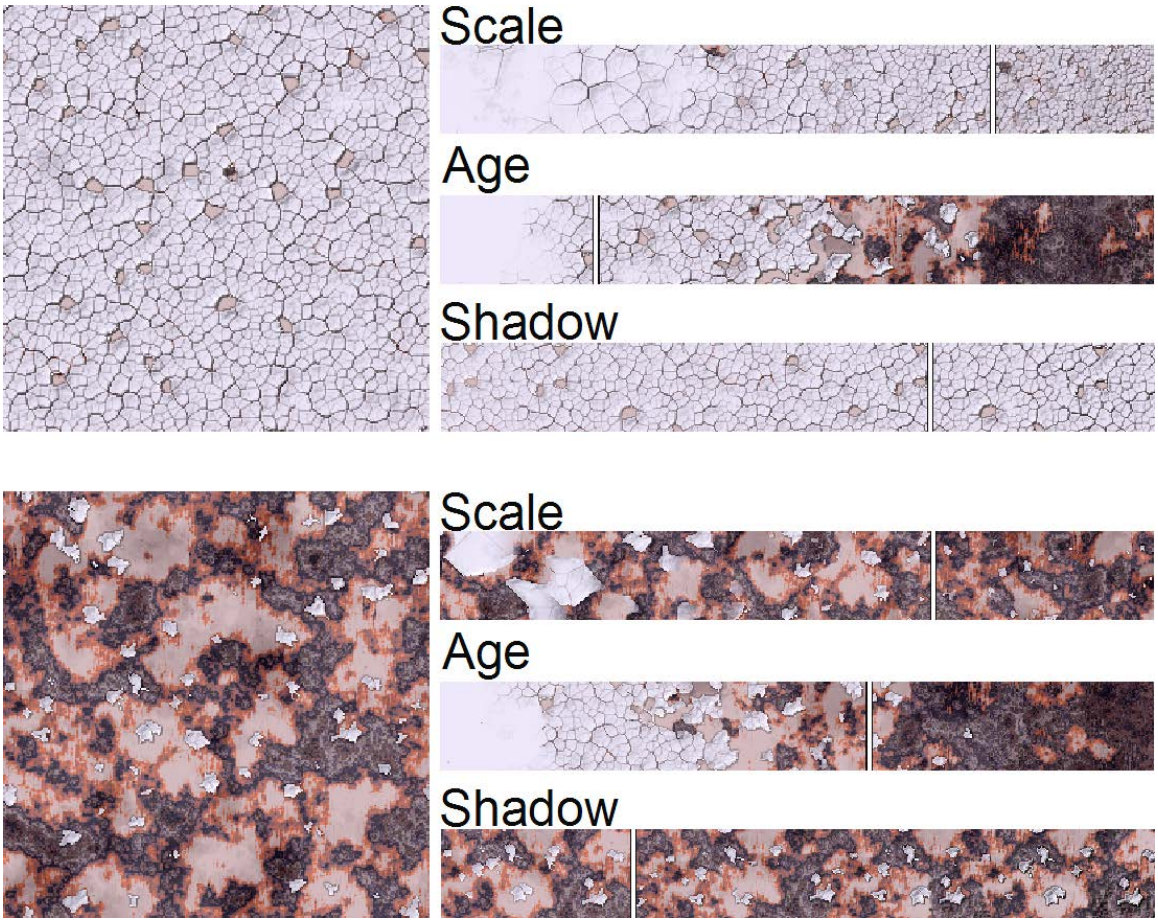

Figure 3.34: Two different settings of the 'rotten wall' texture with visual sliders controlling parameters.

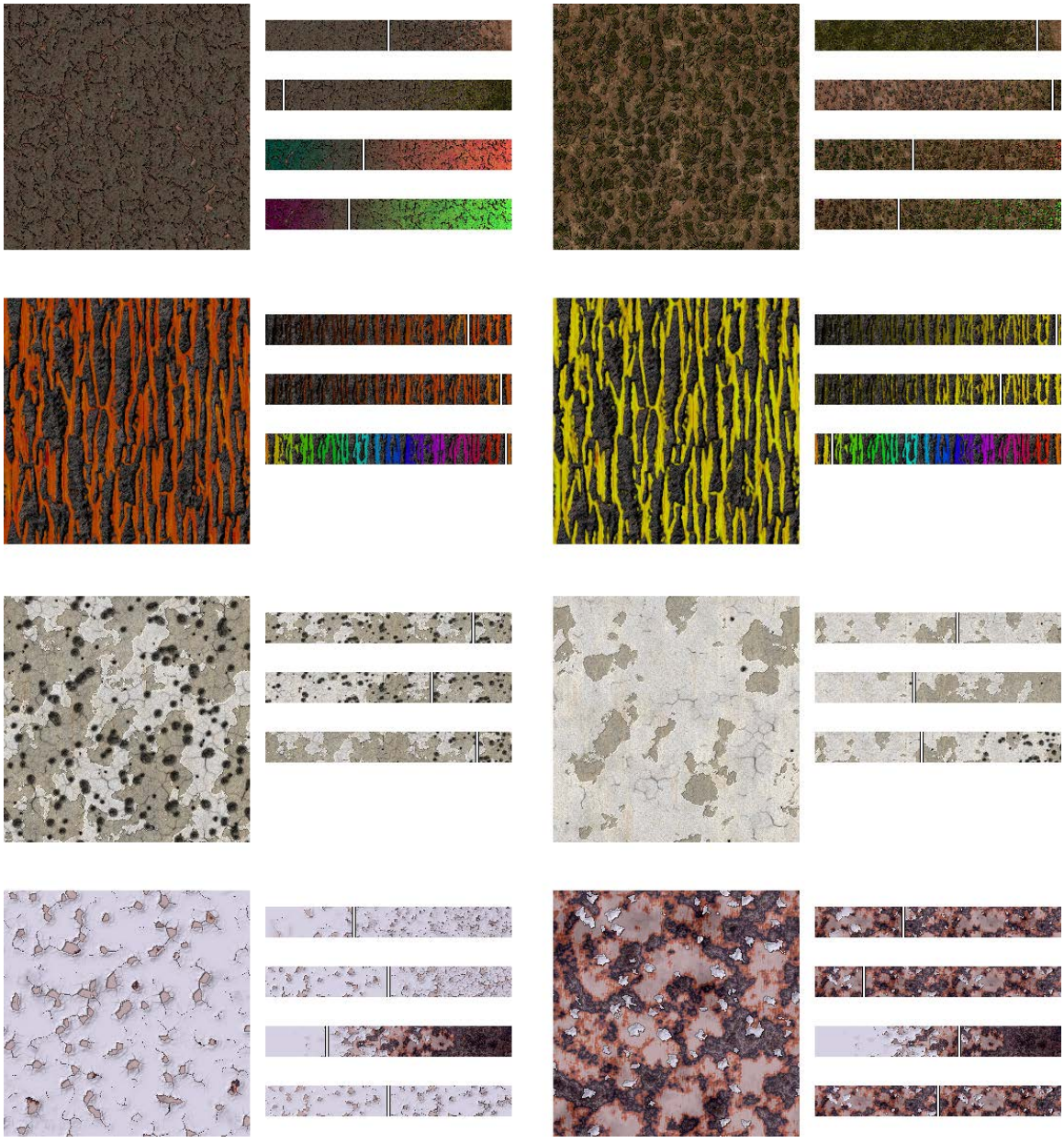

Figure 3.35: User interactions automatically refresh the previews.

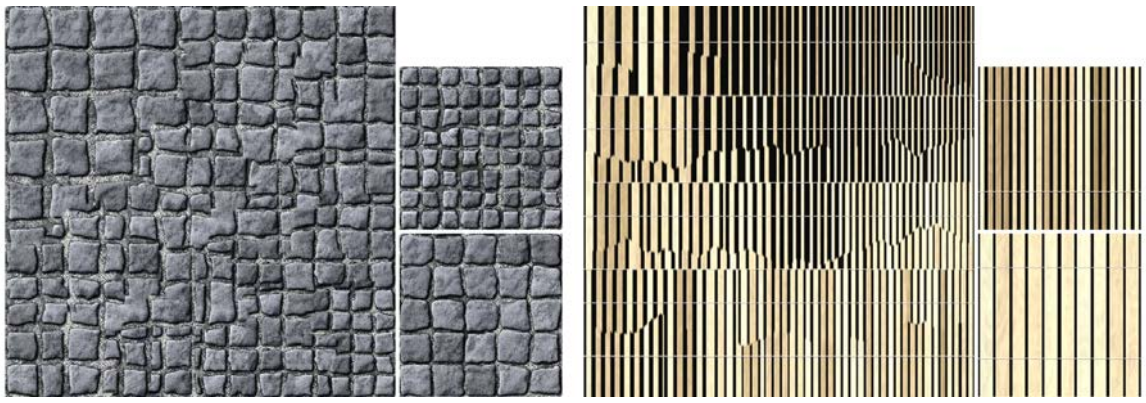

Figure 3.36: *Failure cases:* Our approach does not capture global, sudden structural changes in the textures.

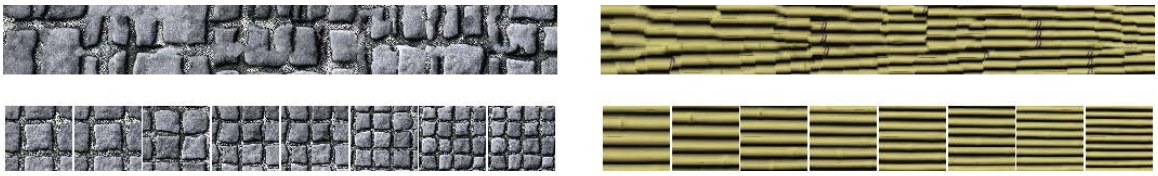

Figure 3.37: *Top:* Failure cases : our approach does not capture global, sudden structural changes in the textures. *Bottom:* Slider thumbnails constitute a possible solution for structural parameters.

3.8 Conclusion

We have introduced in this chapter the ideas of a *procedural texture preview*, *scented sliders* and a *texture palette*.

Our previews have proven effective in carrying a visually pleasing overview of the possible appearances given a limited pixel space. They allow users to rapidly understand the possible content of procedural textures in databases of existing procedures.

The visually driven slider interface generates several procedural textures at interactive rates. This allowed the creation of dynamic visual sliders that give users more control over the result especially for understanding the state of the parameters and for predicting the possible changes.

Our sliders are not only useful for final users, but also during the design of the texture. Most procedural textures are obtained by assembling image filters, between tens to hundreds of them. Each filter exposes a number of parameters (e.g. blur intensity, blending alpha value, emboss direction). Our slider previews help to quickly reveal the possible effects to the designer.

Finally, the palette which combines the procedural texture preview and the scented sliders makes the standard interface more engaging – an important factor when targeting a public versed in art and Computer Graphics. However, this palette allows fast selection

in the procedural texture preview and mostly relies on the scented sliders for precise adjustment. A more complex interaction with the procedural texture preview would require zooming continuously in and out of the palette in an intuitive way. We can draw inspiration from multi–scale texture synthesis [HRRG08] or video tapestries [BGSF10] for continuous zooming techniques. However, to make the zoom intuitive, it is important to keep the user aware of his location in the palette.

Procedural approaches are often considered difficult and non–intuitive due to the technical aspects they expose to the user. We hope this work will help improve existing suggestive interfaces and raise new interest in visualization techniques for procedural textures.

In this chapter we have assumed the existence of a fast–patch based texture synthesizer. In the next chapter we will present a novel patch–based synthesizer that entirely runs on the GPU while providing high quality results.

Chapter 4

Parallel patch–based texture synthesis

4.1 Introduction

In the previous chapter we used texture synthesis to build the final visual summaries for the procedural texture preview and for the scented sliders. In order to build these previews, we used texture synthesis to synthesize the previews from multiple exemplars. These visual previews then served as a palette interface to explore the space of appearances produced by a procedural texture. In addition to a high quality result, such an interface requires high interactivity and therefore a fast by–example texture synthesis approach.

Pixel–based synthesis approaches are known to be fast, offer a high degree of parallelism and map well to the GPU [LH05]. This makes them suitable for an interactive interface. However, as seen Section 3.4 pixel–based synthesis fails to preserve structural patterns and produces low quality results when multiple exemplars are used. In contrast to pixel–based synthesis, patch–based synthesizers are better at preserving structures, while per–pixel synthesis must be guided by the addition of a feature distance map [LH06a], which is often difficult to define. However, as described in section 2.3.2, patch–based approaches are based on relatively slow algorithms to layout the patches and stitch them together, for instance optimizing for the patch frontier (or cut) using graph–cut.

To build the procedural texture previews, we relied on patch–based synthesis but we have omitted details on our approach.

In this chapter, we present a parallel patch–based texture synthesis algorithm that relies on fast optimization methods that are suitable to implement on the GPU. This novel patch–based texture synthesizer is based on the following main contributions:

- We introduce a fast, albeit approximate algorithm to optimize for the patch boundaries. This greatly improves performance with little impact on quality.

- We propose a novel algorithm to deform the patches after boundary optimization and improve feature alignment.
- We synthesize textures by optimizing for multiple patches in parallel. Our global scheme uses newly added patches to hide existing errors. It rejects patches producing visible seams or requiring strong deformations.
- We design a full GPU implementation enabling fast synthesis and interactive user controls with the same level of quality than state–of–the–art patch based synthesizers.

This work has been published at High Performance Graphics 2012 [LL12].

4.2 Overview

Given a source exemplar E our method synthesizes a visually similar toroidal texture by repeatedly stitching multiple patches from E. The result is stored in a map S containing coordinates in E. We note $E[S]$ the final colored texture.

The synthesis is done in an iterative manner where in each iteration multiple patches are randomly selected in E and placed on S. To process patches in parallel within an iteration, the placement is made in a way such as patches placed on S do not overlap. To maximize the number of non–overlapping patches, S is overlaid with a grid where each cell contains one patch. To avoid any bias, the alignment of the grid with respect to the synthesized image randomly changes between iterations. Cells in the grid are then independently processed in parallel (Figure 4.1).

For each patch placed in each cell we optimize for the boundary of the patch so as to minimize visual seams. The optimization aims to minimize the discontinuities along the cut of the patch and to hide existing cuts in $E[S]$ that are produced during previous iterations (Section 4.3).

We further reduce seams by deforming the patch so as to align features (Section 4.4). In order to avoid altering structural patterns in $E[S]$ we use constraints that limit the deformations in the result.

We can decide to either accept or reject a patch depending on its benefit to the overall quality. The patch is rejected if the seam along its cut has more error than all existing seams inside the cut (Section 4.5).

4.3 Fast approximate cyclic cuts

In this section we describe the boundary optimization for a single patch that we note $\mathcal{P}$. Section 4.6 gives details on how to optimize simultaneously for multiple patches.

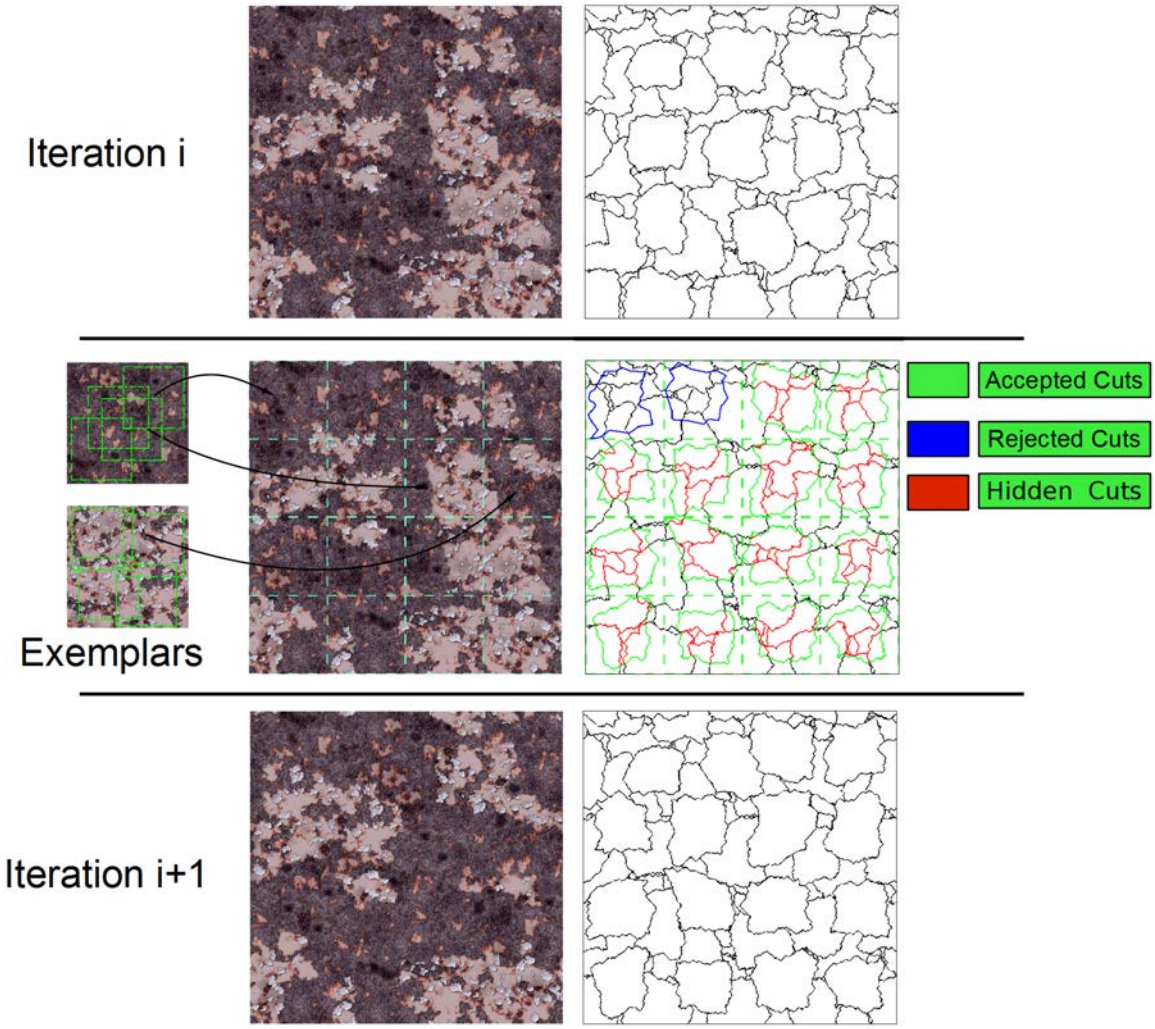

Figure 4.1: *Top:* Synthesized texture at the start of iteration i. The error map is shown on the right with dark colors representing seams. *Middle:* New patches are added to improve the texture. Each patch hides some seams (red) and introduces a new seam (green). Patches with low benefit will be rejected (blue). *Bottom:* Synthesized texture at the end of the iteration.

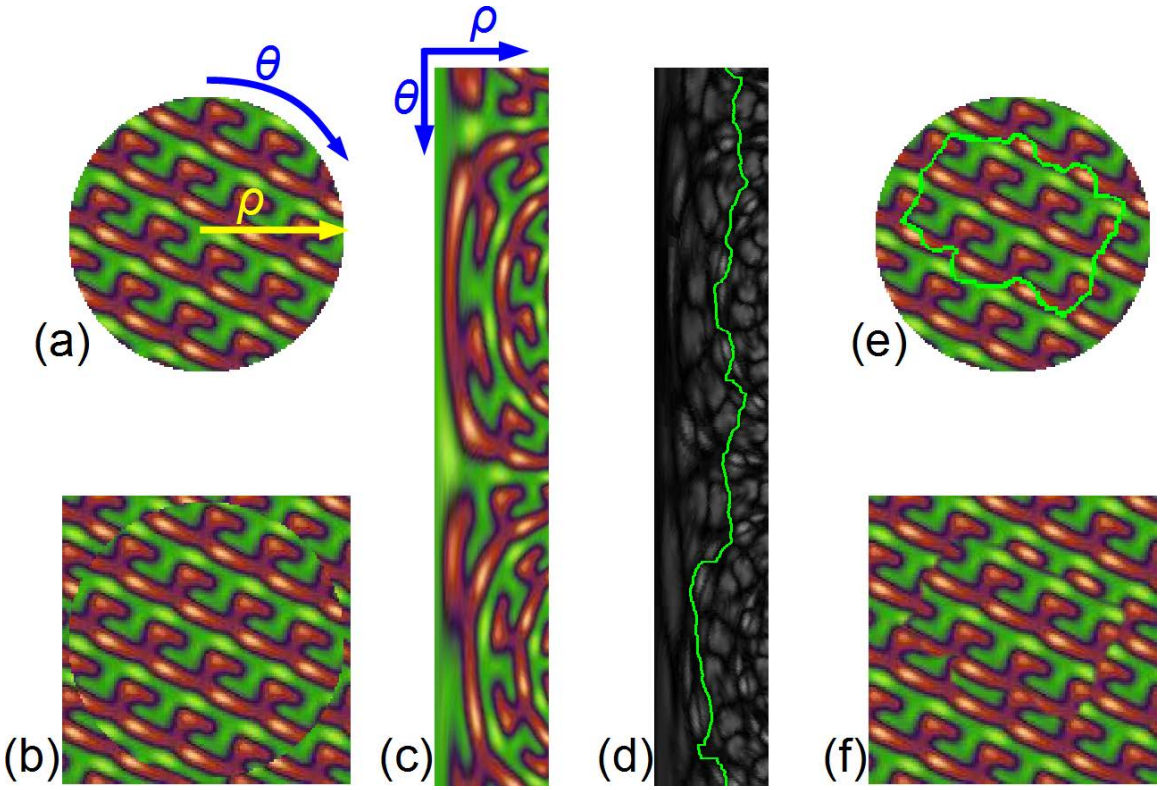

Figure 4.2: *(a):* $\mathcal{P}$ *(b):* Placing $\mathcal{P}$ on $E[S]$. *(c):* $\mathcal{P}_{polar}$ *(d):* Error map with the cut $\mathcal{C}$ in green. *(e):* The optimized boundary $\mathcal{T}^{-1}(\mathcal{C})$ on $\mathcal{P}$. *(f):* Stitching result.

We interpret $\mathcal{P}$ as a disk of radius R centered at a position o_e in E and placed at a center position o_s on S. The goal is to find a closed cut $\mathcal{C}$ in $\mathcal{P}$ that contains at least the point o_s in S. $\mathcal{C}$ should produce as little color differences as possible between $\mathcal{P}$ and $E[S]$.

Instead of using graph–cut to compute $\mathcal{C}$ we want to use dynamic programming (DP). DP is simpler, faster and relies on simple arrays suitable to a GPU implementation. However, to make the optimization compatible with DP we process $\mathcal{P}$ in polar space.

We note $\mathcal{P}_{polar}$ the parameterized version of $\mathcal{P}$ with polar coordinates (Figure 4.2). $\mathcal{P}_{polar}$ is a rectangle of size $\mathcal{W} \times \mathcal{H}$ such as:

$$\mathcal{W} = N_\rho \times R$$

and

$$\mathcal{H} = N_\theta \times 2\pi R$$

N_ρ and N_θ are two constant factors used to add some accuracy to the discrete sampling when transforming $\mathcal{P}$ into $\mathcal{P}_{polar}$. We note $\mathcal{T}$ the transformation from Cartesian to polar space and $\mathcal{T}^{-1}$ the inverse transformation.

Using $\mathcal{P}_{polar}$, the cut $\mathcal{C}$ is now a path that starts at the first row of $\mathcal{P}_{polar}$ and ends at the last row of $\mathcal{P}_{polar}$. Since $\mathcal{C}$ is closed in $\mathcal{P}$, it has to start and end at the same abscissa

in $\mathcal{P}_{polar}$ (cyclic cut).

We now define $\mathcal{C}$ as being in polar space and let $\mathcal{T}^{-1}(\mathcal{C})$ to be the actual boundary of $\mathcal{P}$. We note $\mathcal{C}[y]$ the x coordinate of the curve $\mathcal{C}$ at row y in $\mathcal{P}_{polar}$. Since $\mathcal{T}^{-1}(\mathcal{C})$ is cyclic we access $\mathcal{C}$ in a wrap mode. e.g. $\mathcal{C}[\mathcal{H}+1] = \mathcal{C}[0]$. Using this definition, the existing seams in S now lie on the left side of $\mathcal{C}$ in $\mathcal{P}_{polar}$ and will be hidden by the newly placed patch $\mathcal{P}$.

As in graph–cut textures, we define $\mathcal{C}$ as lying *between* the pixels of $\mathcal{P}_{polar}$. Also, both horizontal and vertical transition errors along $\mathcal{C}$ will be taken into account.

We relax the Y–monotony constraint of image quilting to be:

$$\forall y \in \{1..\mathcal{H}-1\} \quad |\mathcal{C}[y] - \mathcal{C}[y+1]| \leq J_{max}$$

J_{max} is a positive integer that limits the maximum offset between $\mathcal{C}[y]$ and $\mathcal{C}[y+1]$. In image quilting $J_{max} = 1$. The improvement of relaxing this constraint is shown in Figure 4.3.

4.3.1 Normalization

One can notice that in Figure 4.2 distortions appear in $\mathcal{P}_{polar}$ and this especially in the left area of $\mathcal{P}_{polar}$ which corresponds to the interior of $\mathcal{P}$. To account for this distortion a normalization $\mathcal{N}$ is necessary.

The normalization factor $\mathcal{N}$ is used to account for distortions and for the scale factors N_ρ and N_θ. $\mathcal{N}$ makes sure that the total cost inside $\mathcal{T}^{-1}(\mathcal{C})$ in $\mathcal{P}$ matches the total cost in the left area of $\mathcal{C}$ in $\mathcal{P}_{polar}$ (energy preservation).

For a position located at u in $\mathcal{P}_{polar}$, $\mathcal{N}$ must satisfy the equality:
$\sum_{u \in \mathcal{P}_{polar}} \mathcal{N}(u) \times (\mathcal{M}_{polar}(u,(1,0)) + \mathcal{M}_{polar}(u,(0,1))) =$
$\sum_{u \in \mathcal{P}} \mathcal{M}(u, u+(1,0), (1,0)) + \mathcal{M}(u, u+(0,1), (0,1))$

$\mathcal{M}$ is a cost function that measure the visible discontinuities along $\mathcal{C}$ in $\mathcal{P}$. $\mathcal{M}_{polar}$ is a cost function that measure the visible discontinuities along $\mathcal{C}$ in $\mathcal{P}_{polar}$. $\mathcal{M}$ and $\mathcal{M}_{polar}$ will be detailed in the next section.

The solution of the equality gives the final value of $\mathcal{N}$ as follows:

$$\mathcal{N}(u) = \frac{2\pi u_x}{\mathcal{H} \times N_\rho^2}$$

4.3.2 Seams cost

To quantify the visible discontinuities along $\mathcal{C}$ we use a cost function $\mathcal{M}$ similar to the one used in graph–cut textures and defined as:

$$\mathcal{M}(t_e, t_s, \delta) = \left(\frac{||E[t_e] - E[S[t_s - \delta]]||^2 + ||E[S[t_s]] - E[t_e + \delta]||^2}{\eta + ||E[t_e] - E[t_e + \delta]||^2 + ||E[S[t_s]] - E[S[t_s - \delta]]||^2} \right)^d$$

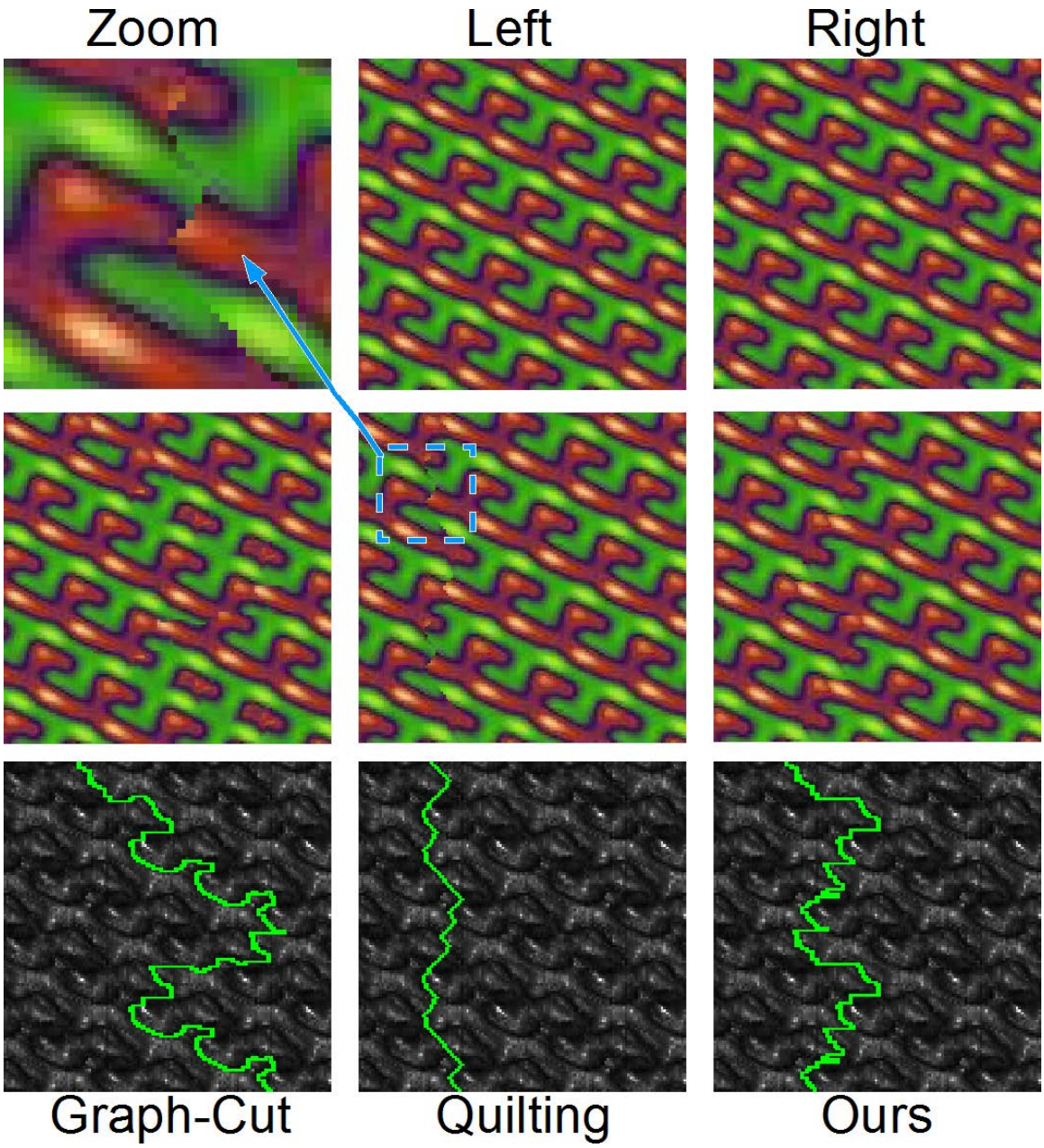

Figure 4.3: *Top:* A left/right texture regions to be overlapped. *Middle:* From left to right: separating the two regions with graph–cut, image quilting and our cut using $J_{max} = 16$. *Bottom:* Error maps produced by the overlap. Each map sums up the vertical and the horizontal transition errors. Light areas indicate high errors. Cuts are shown with a green color.

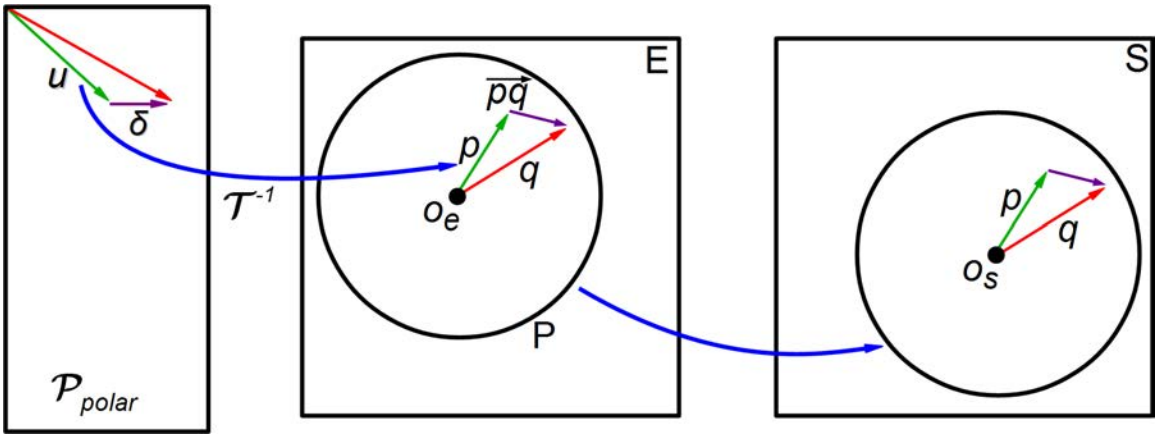

Figure 4.4: Parameters used in $\mathcal{M}_{polar}$. p is the image of u by $\mathcal{T}^{-1}$, q is the image of $u+\delta$ by $\mathcal{T}^{-1}$, p and q have different directions for clarity reasons, the actual p and q have the same direction but different lengths. Note that $\vec{pq}$ is not the image of δ and unlike δ it is not a constant due to the non–linearity of $\mathcal{T}$.

t_e are coordinates in E, t_s are coordinates in S, δ is a displacement vector. $\delta = (1,0)$ when cutting vertically and $\delta = (0,1)$ when cutting horizontally. η is a strictly positive regularization factor used to limit the effect of the denominator. The denominator allows for free transitions at high frequency locations in E and $E[S]$. The exponent d penalizes strong seams when its value is high. It is typically set to 2.

The existing errors in S can be easily computed as follows:

$$\mathcal{M}^S(t_e, t_s, \delta) = \mathcal{M}(S[t_e], t_s, \delta)$$

Because our optimization is done in polar space we define a polar version of $\mathcal{M}$ as follows:

$$\begin{cases} \mathcal{M}_{polar}(u,\delta) = \mathcal{N}(u) \times \mathcal{M}(o_e + p, o_s + q, \overrightarrow{pq}) \\ p = \mathcal{T}^{-1}(u) \\ q = \mathcal{T}^{-1}(u+\delta) \end{cases}$$

u are coordinates in $\mathcal{P}_{polar}$. As before, $\delta = (1,0)$ when cutting vertically and $\delta = (0,1)$ when cutting horizontally. Figure 4.4 shows the parameters used in $\mathcal{M}_{polar}$.

Similarly, we define a polar version of $\mathcal{M}^S$ as follows:

$$\begin{cases} \mathcal{M}^S_{polar}(u,\delta) = \mathcal{N}(u) \times \mathcal{M}(S[o_s + p], o_s + q, \overrightarrow{pq}) \\ p = \mathcal{T}^{-1}(u) \\ q = \mathcal{T}^{-1}(u+\delta) \end{cases}$$

4.3.3 Patch cost

We associate a quality cost to the patch $\mathcal{P}$. It consists in taking the cost of the seam $\mathcal{C}$ from which we subtract the existing costs on the left side $\mathcal{C}$ (costs inside $\mathcal{T}^{-1}(\mathcal{C})$). This leads to an energy function involving three terms: H, V and $\mathcal{E}$.

H represents the cost of horizontal transitions along $\mathcal{C}$ and is defined as:

$$H(u) = N_\rho^2 \times \mathcal{M}_{polar}(u, (1,0))$$

The constant N_ρ^2 cancels the term N_ρ in $\mathcal{N}$. It is used because $\mathcal{C}$ has a unit thickness and is not affected by N_ρ.

V represents the cost of vertical transitions along $\mathcal{C}$. Because a horizontal gap, that can be as long as J_{max} pixels, may appear between $\mathcal{C}[y]$ to $\mathcal{C}[y+1]$ (Figure 4.3 bottom–right), V has to sum up all the vertical costs along this gap. The gap starts at $x_1 = \min(\mathcal{C}[y], \mathcal{C}[y+1])$ and ends at $x_2 = \max(\mathcal{C}[y], \mathcal{C}[y+1])$. V is thus defined as:

$$\begin{cases} V(x_1, x_2, y) = \sum\limits_{x=min(x_1,x_2)}^{max(x_1,x_2)} \mathcal{M}_{polar}((x,y),(0,1)) \;\; if\ y < \mathcal{H} \\ V(x_1, x_2, y) = 0 \;\; otherwise \end{cases}$$

$\mathcal{E}$ represents existing errors in S. These existing errors lie on the left side of $\mathcal{C}$ and will be hidden by $\mathcal{P}$. This means that for the row u_y in $\mathcal{P}_{polar}$ all existing errors that precede $\mathcal{C}[u_y]$ will be subtracted. $\mathcal{E}$ is as follows:

$$\begin{cases} \mathcal{E}(u) = \sum\limits_{x_i=1}^{u_x} (h+v) \\ h = \mathcal{M}^S_{polar}((x_i, u_y), (1,0)) \\ v = \mathcal{M}^S_{polar}((x_i, u_y), (0,1)) \end{cases}$$

We now define the energy $\mathscr{E}(\mathcal{C})$ as:

$$\mathscr{E}(\mathcal{C}) = \sum_{y=1}^{\mathcal{H}} (H(\mathcal{C}[y], y) + V(\mathcal{C}[y], \mathcal{C}[y+1], y) - \mathcal{E}(\mathcal{C}[y], y))$$

Our goal is to find a cut $\mathcal{C}$ such as $\mathscr{E}(\mathcal{C})$ is minimized.

4.3.4 Optimization with DP

To optimize for $\mathscr{E}(\mathcal{C})$ with DP we pre–compute all sub–solutions in a table T using the following recurrence:

$$T[y,i] = \underset{j=i-J_{max}..i+J_{max}}{\arg\min} (T[y-1, j] + V(j, i, y)) + H(i, y) - \mathcal{E}(i, y)$$

$T[y, i]$ is the cost of having $\mathcal{C}[y] = i$. $y \in \{1..\mathcal{H}\}$ and $i \in \{1..\mathcal{W}\}$.

4.3.5 Approximate cuts

Our cut $\mathcal{C}$ is constrained to start and end at the same abscissa. This usually requires repeating the optimization of $\mathscr{E}(\mathcal{C})$ for all starting and ending abscissas as in drag–and–drop pasting [JSTS06]. Repeating the optimization requires an $O(\mathcal{W}^2 \times \mathcal{H} \times J_{max})$ complexity. Instead, we propose an approximation that optimizes once for $\mathscr{E}(\mathcal{C})$ and reduces the complexity to $O(\mathcal{W} \times \mathcal{H} \times J_{max})$. The approximation is based on the following property of our DP:

by backtracking all the paths from bottom to top, there exists at least one path that starts and ends at the same abscissa.

Approximate cut existence

To prove the existence of our approximate cut, we note $\mathcal{C}_{ij}$ the path starting at abscissa i and ending at abscissa j in $\mathcal{P}_{polar}$. $\mathcal{C}_{ii}$ is a closed cut starting and ending at i. In our DP two shortest paths cannot *cross* (principal of optimality [Bel54]). If two sub-paths meet, they will continue along the same sub–optimal path.

Let us assume that there is no closed cut $\mathcal{C}_{ii}$. We first prove by induction that in this case for any path $\mathcal{C}_{nk_n}$ with $n \geq 1$ we have $k_n > n$.

The very first path $\mathcal{C}_{1k_1}$ is not a closed cut, and we necessarily have $k_1 > 1$. Now, consider a path $\mathcal{C}_{nk_n}$ with $n > 1$ and assume that $k_n > n$. The next path $\mathcal{C}_{n+1k_{n+1}}$ has to be such that $k_{n+1} \geq k_n$ otherwise $\mathcal{C}_{nk_n}$ and $\mathcal{C}_{n+1k_{n+1}}$ would cross each other. If $k_n > n + 1$ then $k_{n+1} > n + 1$ since $k_{n+1} \geq k_n$. If $k_n = n + 1$ then we have $k_{n+1} \geq n + 1$. However, since there is no closed cut, it follows that $k_{n+1} > n + 1$. The first proof is complete.

Let us now consider the very last path: $\mathcal{C}_{\mathcal{W}k_{\mathcal{W}}}$. We have $k_{\mathcal{W}} > \mathcal{W}$ which is impossible since the path would exit $\mathcal{P}_{polar}$. Therefore, a path $\mathcal{C}_{ii}$ has to exist. This proves the existence of at least one cut $\mathcal{C}_{ii}$.

Approximate cut quality

We have experimentally compared our approximate cut to the optimal one by running multiple tests on a large number of textures. The table below shows the average results of the experiment.

Number of textures	3000
Optimal cut average error	50.855
Approximate cut average error	53.633
Approximate cut average ranking	8.59/256
All cuts average error	81.962
Approximate cut ranked first	17.76%

Compared to the average, our approximate cut gives a low energy. It is indeed a shortest path and it quickly joins some of the few shortest paths along low–cost channels. However, there are rare cases where our approximation produces seams stronger than the ones produced by the optimal cut. Such seams are likely to be rejected (Section 4.5).

4.4 Feature alignment

Minimizing $\mathscr{E}(\mathcal{C})$ does not always guarantee a seamless result. This is especially the case if the texture contains aligned structural patterns like bricks or straws. For instance it is possible to offset the two overlapped textures of Figure 4.3 to produce a case where *any* cut optimization produces seams. In Figure 4.5 the overlapping is made so that the error map contains high–cost strips (the bright slanted strips in the figure) and our cut as well as graph–cut are constrained to cross these strips, therefore producing visible seams.

If the cut optimization fails and produces seams, we propose to apply small deformations to align features on each side of $\mathcal{C}$. The deformation consists of two steps:

- In a first step, the colors in $\mathcal{P}_{polar}$ along $\mathcal{C}$ are offset to match the colors in $E[S]$ lying on the other side of $\mathcal{C}$ (Figure 4.6 **e** and **f**).
- In a second step a deformation is smoothly propagated to the inside of $\mathcal{P}$ (the left of $\mathcal{P}_{polar}$) based on color displacements made in the first step (Figure 4.6 **b**).

4.4.1 Offsetting colors along the cut

Recall that the curve $\mathcal{C}$ is represented as an array where $\mathcal{C}[y]$ is the x coordinate of the curve at row y in $\mathcal{P}_{polar}$. Our goal is to offset the indices of $\mathcal{C}$ in order to to align features. We note $\mathcal{D}$ the array that contains the new indices in $\mathcal{C}$ after offsetting. i.e. the color at $\mathcal{P}_{polar}[\mathcal{C}[y], y]$ is replaced by the color at $\mathcal{P}_{polar}[\mathcal{C}[\mathcal{D}[y]], \mathcal{D}[y]]$.

As $\mathcal{C}$, $\mathcal{D}$ is accessed using wrap mode. We note $\mathcal{C}[\mathcal{D}]$ the cut with the offset colors but having the same shape as $\mathcal{C}$.

Since we propagate the deformation at the next stage, we take care that no fold–over occurs when optimizing for $\mathcal{D}$. For this we ensure that: $y \leq z \Rightarrow \mathcal{D}[y] \leq \mathcal{D}[z]$.

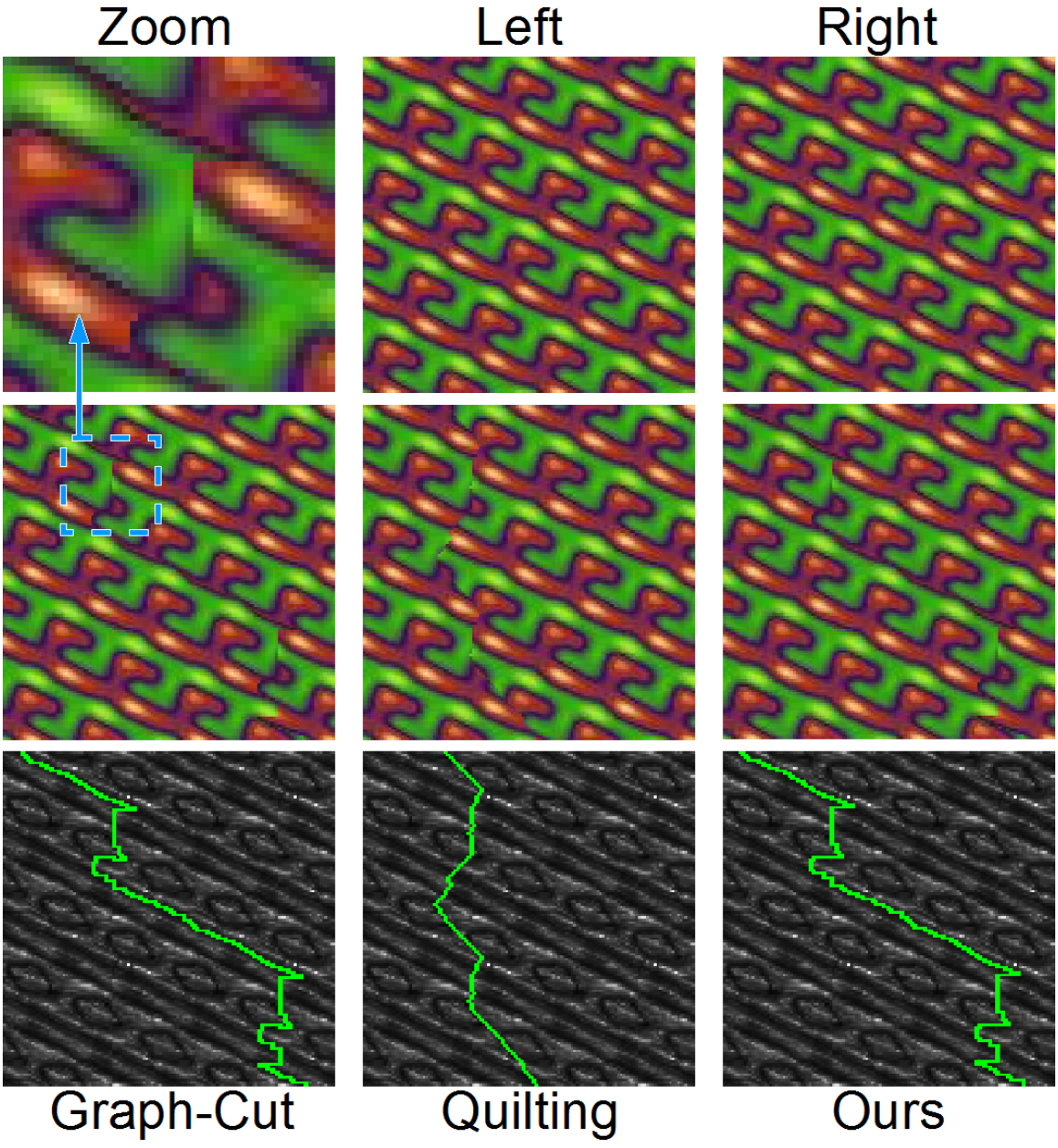

Figure 4.5: *Top:* A left/right texture regions to be overlapped. *Middle:* From left to right: separating the two regions with graph–cut, image quilting and our cut using $J_{max} = 16$. *Bottom:* Error maps produced by the overlap. Each map sums up the vertical and the horizontal transition errors. Light areas indicate high errors. Cuts are shown with a green color.

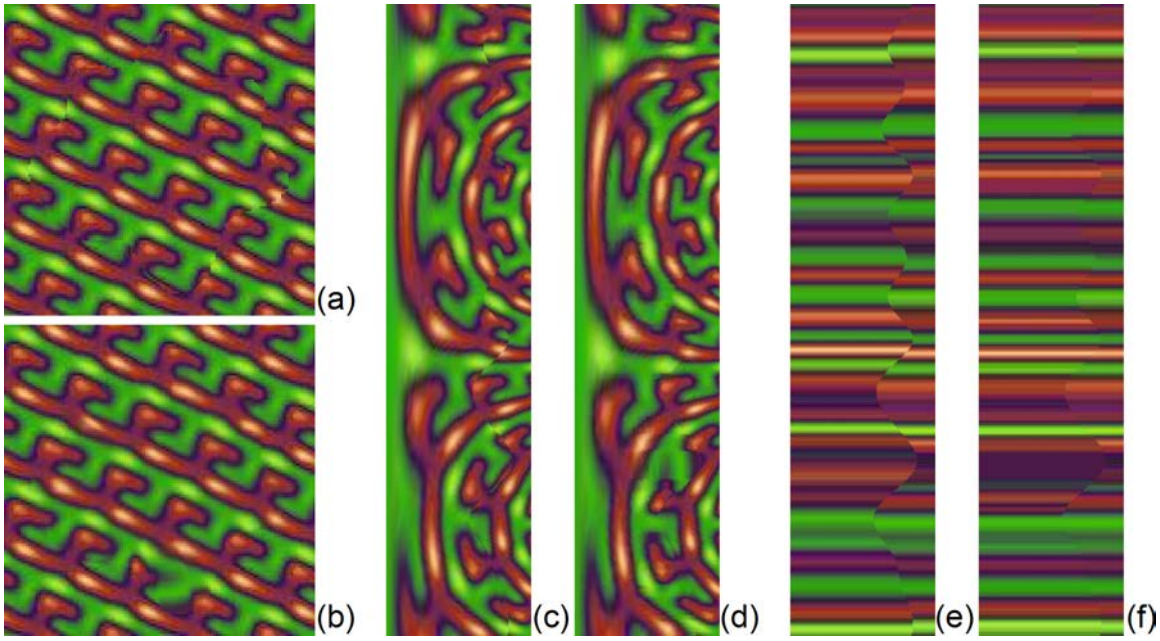

Figure 4.6: *(a)* The patch $\mathcal{P}$ produces seams when placed on $E[S]$. *(b)* Result after feature alignment. *(c)* Polar space view before feature alignment. *(d)* Polar space view after feature alignment. *(e)* Colors along $\mathcal{C}$ before feature alignment. *(f)* Colors along $\mathcal{C}$ after feature alignment.

Recall that the shape of $\mathcal{C}$ can have a gap as long as J_{max} between $\mathcal{C}[y]$ and $\mathcal{C}[y+1]$. Since $\mathcal{C}$ only encodes one x coordinate per row in $\mathcal{P}_{polar}$, offsetting colors within the gap is difficult. We therefore ignore the cost of vertical transitions. In practice, the cases where horizontal seams persist due to ignoring the cost of vertical transitions are infrequent and such seams are likely to be either rejected or replaced by another patches if the synthesis is re–iterated.

$\mathcal{D}$ is obtained by minimizing the following energy:

$$\mathscr{E}_{\mathcal{D}}(\mathcal{D}) = \sum_{y=1}^{\mathcal{H}} \mathcal{M}_{\mathcal{D}}(\mathcal{D}[y], y)$$

$\mathcal{M}_{\mathcal{D}}$ is the cost of replacing the color at coordinates $(\mathcal{C}[y], y)$ with the color at coordinates $(\mathcal{C}[\mathcal{D}[y]], \mathcal{D}[y])$. It is defined as:

$$\begin{cases} \mathcal{M}_{\mathcal{D}}(i,y) = \mathcal{M}(o_e + d, o_s + q, \overrightarrow{pq}) \\ p = \mathcal{T}^{-1}(\mathcal{C}[y], y) \\ q = \mathcal{T}^{-1}(\mathcal{C}[y]+1, y) \\ d = \mathcal{T}^{-1}(\mathcal{C}[i], i) \end{cases}$$

Since we are only interested in applying small deformations, we limit the maximum offset between two rows to be:

$$\forall y \in \{1..\mathcal{H}-1\} \;\; \mathcal{D}[y+1] - \mathcal{D}[y] \leq 2$$

The constant 2 has been chosen empirically. It represents the smallest possible offset. We also limit the maximum amount of deformation by setting the global constraint:

$$\forall y \in \{1..\mathcal{H}\} \;\; |\mathcal{D}[y] - y| \leq D_{max}$$

D_{max} is a constant set by the user to limit the maximum possible offset. By using a small value, we prevent strong deformations and reduce the required memory (The DP optimization table T has a size of $(2D_{max}+1) \times \mathcal{H}$).

4.4.2 Optimization with DP

The optimization computes all sub–solutions using the recurrence:

$$T[y,i] = \underset{j=i-2..i}{\arg\min}(T[y-1,j]) + \mathcal{M}_{\mathcal{D}}(i,y)$$

$T[y,i]$ is the cost of having $\mathcal{D}[y] = i$. Notice how the DP minimizes $\mathscr{E}_{\mathcal{D}}(\mathcal{D})$ by performing 3 actions:

- $\mathcal{D}[y] = \mathcal{D}[y-1]$: repeating the same pixel.
- $\mathcal{D}[y] = \mathcal{D}[y-1] + 1$: advance to the next pixel.
- $\mathcal{D}[y] = \mathcal{D}[y-1] + 2$: advance twice (jumping one pixel).

4.4.3 Initial state

The solution $\mathcal{D}$ must start and end at the same abscissa in T otherwise parts of the texture will be lost after propagation. We face the same issue we encountered during the optimization of $\mathscr{E}(\mathcal{C})$. Repeating the optimization for all initial states is impractical. Starting from a random initial state could lead to artifacts (Figure 4.7). We therefore re–use the same approximate solution of $\mathscr{E}(\mathcal{C})$ optimization.

4.4.4 Deformation propagation

The propagation is done by replacing the color at pixel (x,y) in $\mathcal{P}_{polar}$ with the color at pixel (x',y') using the following interpolation:

$$y' = lerp(y, \mathcal{D}[y], \left(\frac{x}{\mathcal{C}[y]}\right)^{\gamma})$$

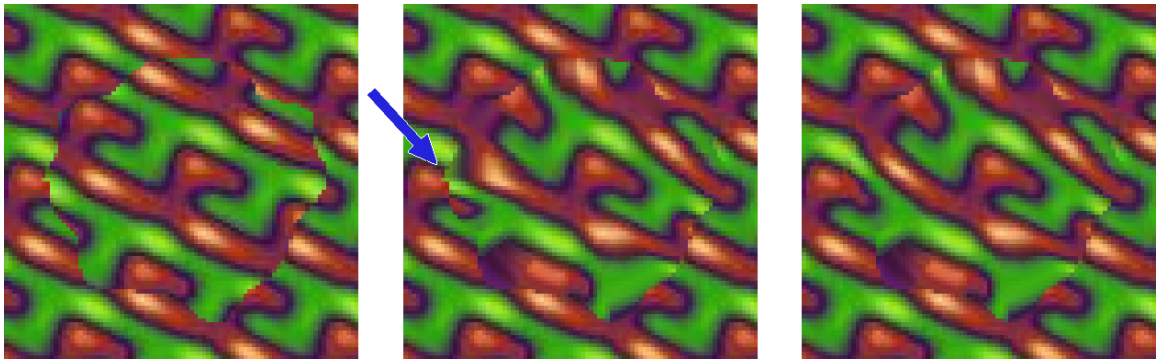

Figure 4.7: *Left:* Random cut with no deformation. *Middle:* Deformation using a fixed initial state. The starting position is pointed by the blue arrow. *Right:* Deformation using our approximate solution for $\mathcal{D}$.

$$x' = x \times \frac{\mathcal{C}[y']}{\mathcal{C}[y]}$$

γ controls the amount of deformation inside $\mathcal{P}_{polar}$. A small value will make the propagation spread further. *lerp* function represents linear interpolation. It considers the fact that $\mathcal{P}_{polar}$ is vertically cyclic and always interpolates along the shortest path. e.g. $\mathcal{C}[0]$ is closer to $\mathcal{C}[\mathcal{H}]$ than $\mathcal{C}[\mathcal{H}/2]$.

4.5 Patch rejection and placement

4.5.1 Patch rejection

After the optimization, we decide to merge the patch with the result or reject it if it has no benefit to the overall quality.

We use two rejection policies:

A constant time rejection predicate

The first rejection policy is applied before feature alignment and is based on a simple predicate noted $isImproving_{before}(\mathcal{C})$ returning *true* iff $\mathscr{E}(\mathcal{C}) \leq 0$. When $isImproving_{before}(\mathcal{C})$ is *true* the subtracted existing errors in S, i.e. $\mathcal{E}$, are greater than the errors produced by $\mathcal{C}$ and in this case the patch corresponding to $\mathcal{C}$ provides a benefit and will be accepted. The patch will be rejected otherwise. Rejecting patches before feature alignment is a good heuristic. Accepted patches would have few seams and feature alignment performs well in these cases.

A more constrained rejection predicate

The second rejection policy is applied after feature alignment and requires changing the predicate as follows:

$$isImproving_{after}(\mathcal{C}) = \mathscr{E}(\mathcal{C}[\mathcal{D}]) + \mathscr{E}(\mathcal{P}) - \mathcal{E} \leq 0$$

$\mathscr{E}(\mathcal{C}[\mathcal{D}])$ is the energy along $\mathcal{C}$ after color offsetting, $\mathscr{E}(\mathcal{P})$ is the total energy in $\mathcal{P}$ after deformation and $\mathcal{E}$ is the total existing energy on the left of $\mathcal{C}$. $isImproving_{after}$ requires some extra computations while $isImproving_{before}(\mathcal{C})$ comes practically for free. However $isImproving_{after}$ ensures a monotonically decreasing global energy.

4.5.2 Patch placement

Patches in E are randomly sampled with a uniform distribution. We chose not to use coherent techniques [Ash01, TZL$^+$02, ZG02, BSFG09] that are commonly used in pixel–based synthesizers because a patch in E is already highly coherent and the amount of coherence can be easily controlled by changing the maximum radius R.

Note that Kwatra et al. [KSE$^+$03] use an exhaustive search accelerated using an FFT–based block matching. In our case, we can simply accelerate the exhaustive search by optimizing in parallel for all patches in E then keeping the patch that produces the least error. This however requires a separate optimization table for each patch in E and this would require too much memory. We rather randomly sample N patches in E then keep the patch that produces the least error. When the value of N is high, more amount of memory is required. All results shown in this thesis use $N = 1$. We use $N = 1$ to keep memory usage as low as possible and rely on the deformation and the rejection to improve the cases where a patch produces visible seams.

Adapting N automatically

Using $N = 1$ requires a minimum amount of memory and still offers good performance. To use the least amount of memory, our final implementation is limited to $N = 1$. If the memory budget allows the use of more than one patch per cell, we present here a simple method that adapts N with the aim to focus computation in high error areas in S. We did not actually implement this strategy but we believe this is beneficial if many samples per cells are allowed.

Recall that the output S is overlaid with a grid and cells in this grid are processed in parallel (Figure 4.1). Rather that setting a constant value N for all cells, we add the possibility to automatically set N depending on the amount of error contained in the cell. Figure 4.8 shows how the value of N adapts to the error in S.

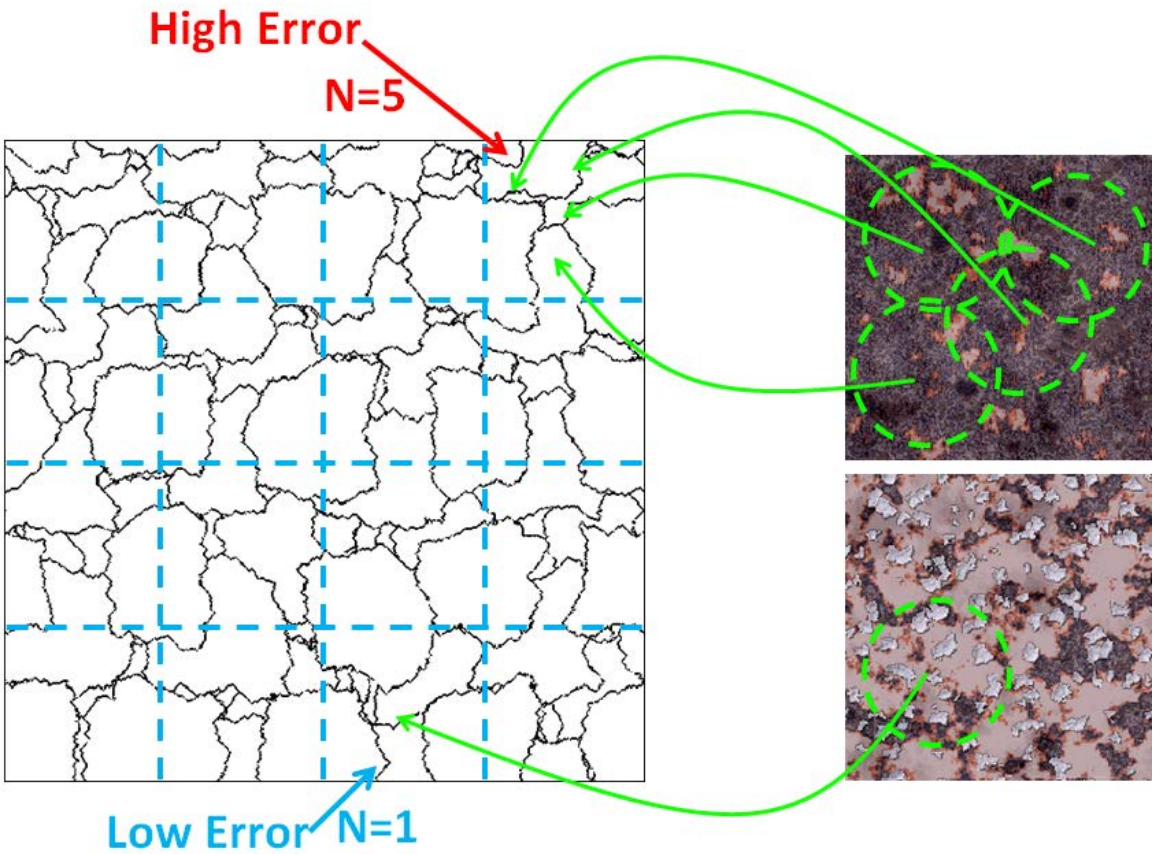

Figure 4.8: Adapting N automatically.

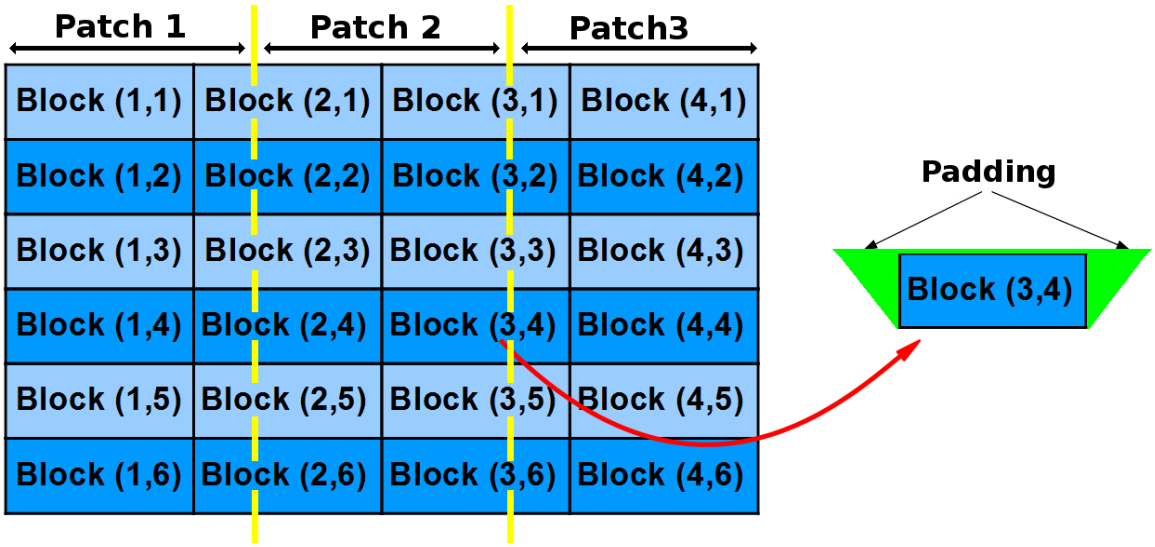

Figure 4.9: The DP table is subdivided into blocks and each block is processed by a block of threads. A block of threads needs to access some data in the left and right blocks and the first row of above blocks (the green padded regions).

$\mathcal{M}^S$ gives the error in S for one pixels. The total error $T_{\mathcal{M}^S}$ in one cell of S can be rapidly accumulated using a reduction algorithm. The number N of sampled patches is determined using the formula:

$$N = \max(N_{max}, \eta T_{\mathcal{M}^S})$$

Where N_{max} is an upper limit for N. η is a user defined positive constant that scales all values that N can take.

4.6 Implementation details

We implement the algorithm using NVIDIA CUDA.

Multiple patches are processed simultaneously by placing their optimization tables one next to the other from left to right. For n patches, the global optimization table will have a size of $w \times h$ where $w = n \times \mathcal{W}$ and $h = \mathcal{H}$. Using this layout, the optimization is quite similar when processing a single or multiple patches. A thread just needs to know which patch is being processed to set the corresponding boundaries (The yellow dashed lines in Figure 4.9). Data beyond the boundaries are not accessed by the thread (clamp mode is used).

The algorithm executes two main optimizations: the optimization of $\mathscr{E}(\mathcal{C})$ followed by the optimization of $\mathscr{E}_{\mathcal{D}}(\mathcal{D})$. Since both optimizations are similar, we mostly give details about the optimization of $\mathscr{E}(\mathcal{C})$. The optimization of $\mathscr{E}_{\mathcal{D}}(\mathcal{D})$ is simpler due to the fact that vertical transition errors are ignored (they are set to zero).

The buffers allotted for the optimization of $\mathscr{E}(\mathcal{C})$ will be re–used during the optimization of $\mathscr{E}_{\mathcal{D}}(\mathcal{D})$. We therefore make sure that there is enough memory for both optimizations.

We allocate the following *texture* buffers having a size of $w \times h$ each

- A buffer H that first stores existent horizontal costs then stores new horizontal costs,
- A buffer V that stores new vertical costs
- A buffer $\mathcal{E}_v$ that stores existent vertical costs.

Prior to optimization, we start pre–computing all transition costs by performing the following steps:

- Fill H with horizontal existing costs and $\mathcal{E}_v$ with vertical existing costs (Pass1: one thread per entry in the table).
- For each row within each patch in H and $\mathcal{E}_v$, accumulate costs from left to right (Pass2: a thread per row of a patch).
- Fill V with zeros.
- Compute the new horizontal costs in H and the new vertical costs in V and subtract the existing content before storage (Pass3: one thread per entry in the table).

To efficiently process each pass within a same kernel H, V and $\mathcal{E}_v$ share the same texture unit. Texture memory is mainly used to avoid cache conflicts when accumulating existing errors (Pass2). After pre–computing the costs, H will be used as the main optimization table. V and $\mathcal{E}_v$ will be read–only and will provide the additional terms of $\mathscr{E}(\mathcal{C})$.

The DP optimization consists of a top–down sub–solution pre–computation in H followed by backtracking all cuts and storing the result in–place in H. Cuts that do not start and end at the same abscissa will be assigned an infinite cost. A reduction algorithm then selects for each patch the cut with the minimum cost.

The DP we solve has the property that each row y can be processed in parallel and only depends on the preceding row $y-1$. For the DP accumulation task a single row parallelism suffers from 3 limitations:

- By parallelizing a single row at a time, the number of threads will be too limited to fully exploit the GPU.
- After processing one row a global synchronization (stopping and re–running the kernel) is required before processing the next row.
- The synchronization suspends the computation within a column for a long time. Meanwhile other computations will fill–up the cache losing its coherence.

We address these limitations by processing many rows before making a global synchronization. This is done by subdividing H into blocks as shown Figure 4.9. A local DP accumulation is done in each block and the global synchronization happens after processing a line of blocks.

To correctly compute the result in each block, the blocks are padded with additional data (The green parts in Figure 4.9). These data belong to the left, right and above neighboring blocks. The padded data are only used to ensure correctness within each block. The additional computations within these padded regions are wasted but the overhead is small compared to the benefit of the increased coherence. We use one block of threads to process one block of data (including the padded regions). Because the padded data can be processed simultaneously by multiple blocks, they are first stored in a read–only temporary buffer. A block of threads is thus responsible of loading the data in shared memory and a subset of the threads applies the DP accumulation in this shared memory. The threads in the block finish by copying the processed data (without the padding) from shared memory back to H. The size of the blocks is determined empirically to be $B_w = 32 + J_{max}$ and $B_h = \frac{32\sqrt{J_{max}}-16}{J_{max}}$. The number of threads per block is also determined empirically by rounding–up the constant $32 + B_h \times J_{max}$ to the next multiple of 32.

The same process is used for the bottom–up backtracking with the difference that there is no padding. The temporary memory will now store the solution which is then copied back to the table H before a global synchronization. For the backtracking operation, if memory is not an issue one can eliminate the synchronization by allocating a table having the same size as H to store all the results.

The optimization of $\mathscr{E}_{\mathcal{D}}(\mathcal{D})$ is quite similar to the optimization of $\mathscr{E}(\mathcal{C})$. We can easily re–use the same DP. However since we are ignoring vertical transition costs when optimizing $\mathscr{E}_{\mathcal{D}}(\mathcal{D})$ and since only 3 actions are performed at each step, the buffers V and $\mathcal{E}_v$ are set to zero while $J_{max} = 1$.

4.7 Results and applications

In this section we use the following synthesis settings: S is initialized with white noise texture coordinates and it has a size of 512×512. $R = 64$, $D_{max} = 32$, $N_\rho = 2$, $N_\theta = 1.5$, $J_{max} = 4$ and $\gamma = 1$. We use the predicate $isImproving_{after}$ for rejection. We notify the reader if settings are changed. In general, the user can interactively tweak any parameter to improve the quality for a specific texture.

4.7.1 Varying the maximum radius R

The parameter with most effect on quality is the maximum patch radius R. Figure 4.10 shows synthesis results for the same texture but using different values for R. The synthesis is only iterated 8 times (it is necessary to iterate the synthesis a few times to cover the whole map S; 3 to 5 iterations are often enough to cover the whole map S with patches). When R is small the synthesizer fails to capture the flower patterns. It either breaks the flowers or rejects them. Because of rejection, the flowers will potentially disappear if the synthesis is further iterated. When R is large the synthesizer copies large patches from E producing a seamless result but having little variety. Setting R to be just large enough to capture the flowers results in a nice distribution of flowers. Please keep in mind that R is the maximum possible radius. A patch can be as small as one pixel regardless of R.

4.7.2 Automatic radius R

We allow synthesis with an automatically decreasing radius R. Starting with a large value for R the synthesizer can quickly capture coarse structures like the canes in Figure 4.11. By decreasing R small patches are accepted making local variations in the result. Patch rejection ensures an unchanged structure during the late iterations.

4.7.3 Automatic D_{max}

Similar to R, automatically reducing D_{max} starting from a large value can make the synthesis converge faster. Although the result is highly deformed during earlier iterations, the late iterations will focus on removing deformed features rather than avoiding seams (Figure 4.12).

4.7.4 Texture completion

Our synthesizer allows for a straightforward texture completion application. If a texture contains holes it is easy to fill in these holes by setting an infinite existing cost in the holes. In practice we obtain the high cost by filling the holes with random coordinates. By running the synthesizer, patches are accepted within high cost areas corresponding to holes. Figure 4.13 illustrates the completion process.

In Figure 4.13 nothing prevents the synthesizer from taking patches from the flower then pasting them to S. The flower disappeared because its corresponding patches have been rejected. Using a large patch radius R that encompass the flower would probably put the flower in the result.

It is possible to prevent copying patches from the holes by providing a mask to the synthesizer. This way, masked pixels are not copied in the result. Figure 4.14 shows such

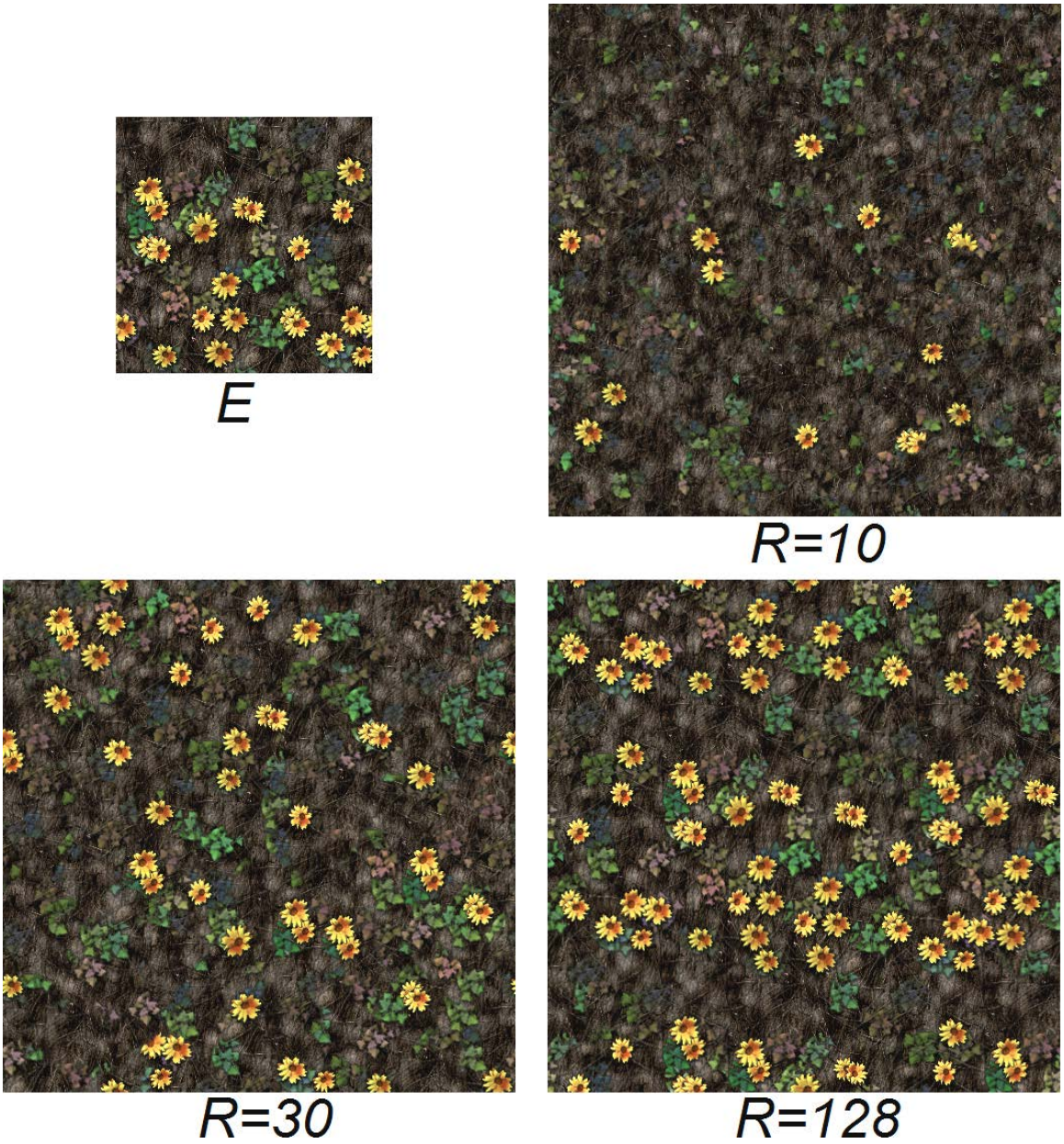

Figure 4.10: Varying the maximum radius R.

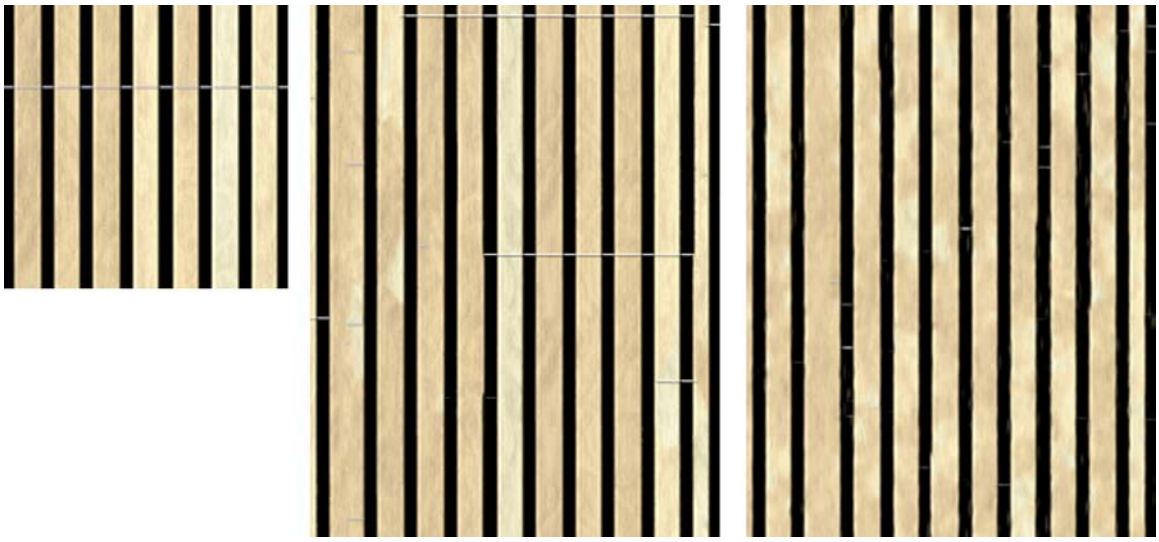

Figure 4.11: Starting from a patch radius $R = 132$, R is decremented by 4 every iteration. *From left to right:* The exemplar E. Result when $R = 72$. Result when $R = 4$. Notice the variations within the canes when $R = 4$.

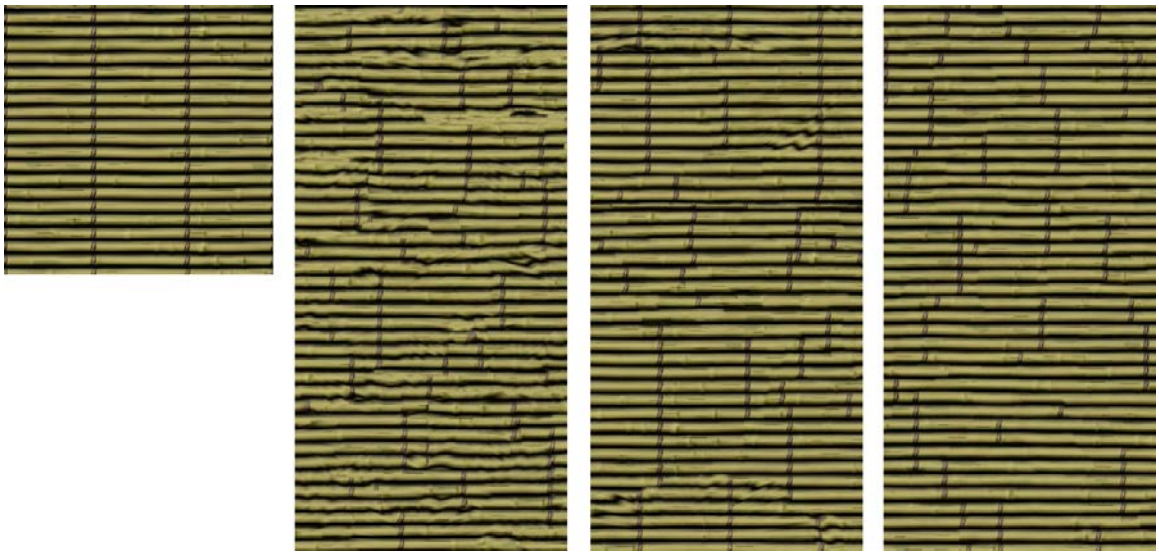

Figure 4.12: Starting from a max deformation of $D_{max} = 160$, D_{max} is decremented by 10 every iteration. *From left to right:* The exemplar E, result when $D_{max} = 100$, $D_{max} = 50$ and $D_{max} = 0$. S has a size of 256×512.

Figure 4.13: Texture completion. *Left:* E. *Middle:* Erasing the flower by filling S with random coordinates. *Right:* Running the synthesizer to fill the texture.

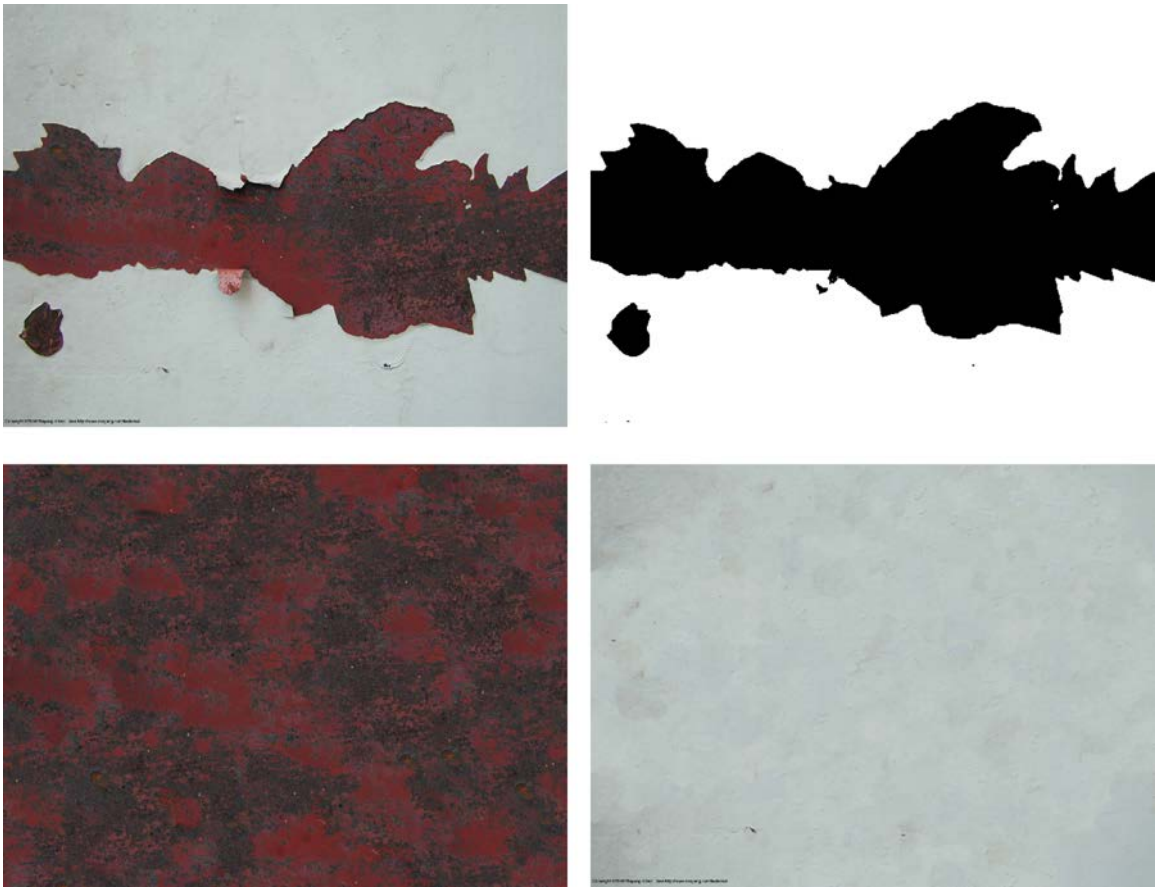

Figure 4.14: Texture completion. *Top–left:* E. *Top–right:* mask. *Bottom–left:* Using the invert mask. *Bottom–right:* Using the mask. S has size 1600×1200.

constrained completion.

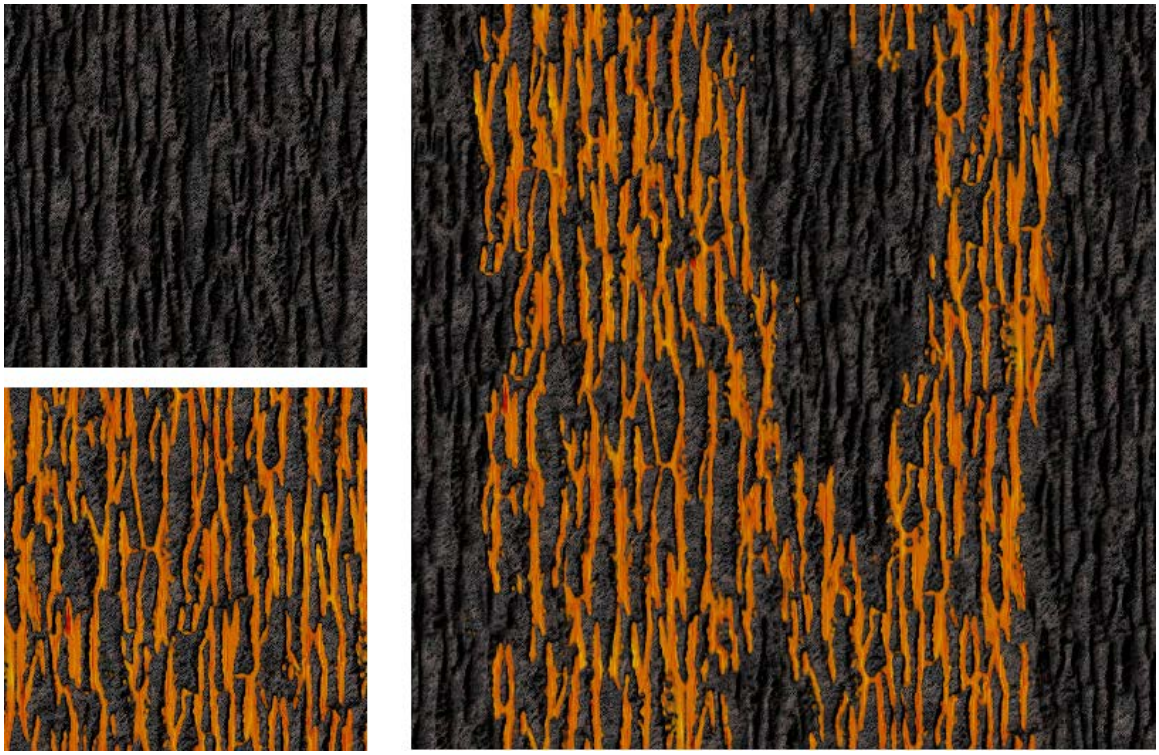

Figure 4.15: *left:* Multiple exemplars. *right:* Synthesis result.

4.7.5 Multiple exemplars and texture painting

One advantage of patch–based synthesis over pixel–based synthesis is the ease of using multiple exemplars. In pixel–based approaches the transition between different texture regions is not well defined and requires special handling. To use multiple exemplars the only requirement is to add a third coordinate in S to store the exemplar index. Figure 4.15 shows a synthesis result that uses patches from different textures.

Using multiple exemplars one can use a texture as a brush to paint on another texture. The painted zones are considered as holes and the synthesizer instantly fills these holes.

4.7.6 Patch drag–and–drop

In a drag–and–drop task our fast synthesizer allows seeing the stitching result while dragging a patch. This gives more intuition to the user on where to place the patch. We disable rejection for this task. In addition the user can set a minimum patch radius in which the existing costs are forced to be zero.

4.7.7 Synthesis convergence

The plot in Figure 4.16 shows the global cost evolution and the patch acceptance rate during the iterative synthesis process. At the tenth iteration the synthesis went near a local

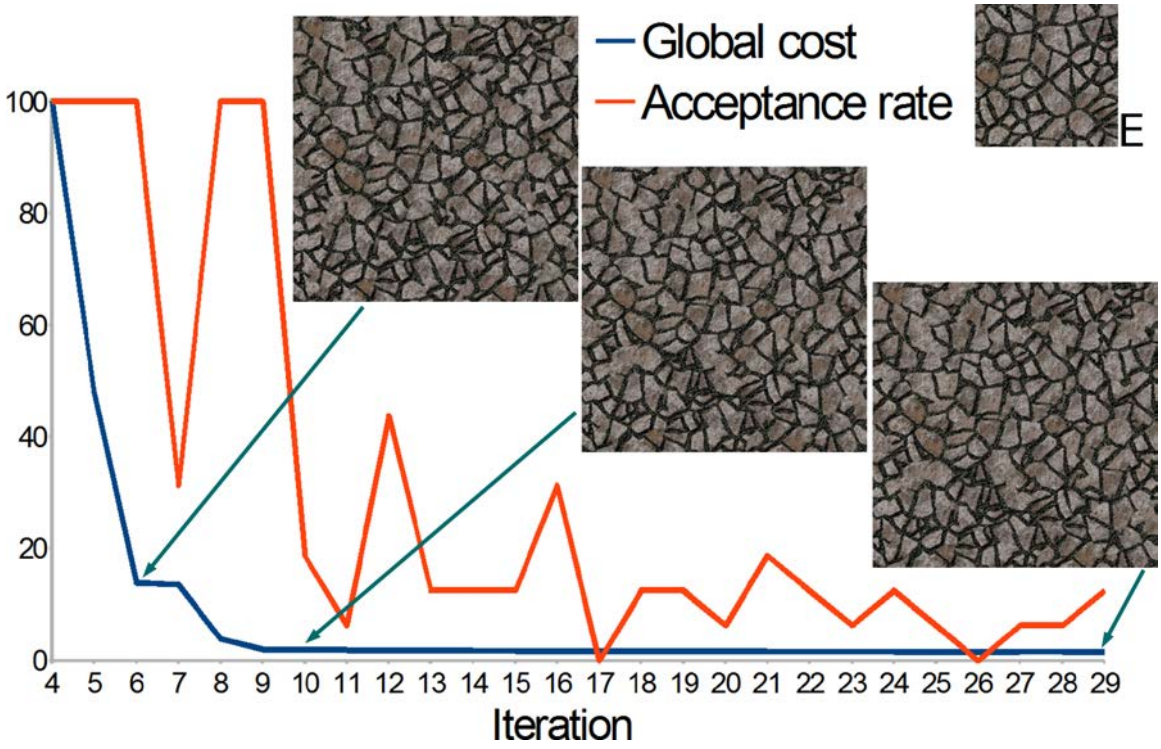

Figure 4.16: Global cost and patch acceptance rate evolution. The global error is scaled to be in range [0..100]. The first 4 iterations produce very high errors and are ignored.

minimum dropping the patch acceptance rate and making the result almost unchanged during the next iterations. Manually changing synthesis parameters or disabling rejection is a simple way to get away from local minima. Please note that convergence could vary from one texture to another. In addition, note that the algorithm starts from random coordinates in S (worst initialization) and that other initializations may improve convergence.

4.7.8 Performance

The following table lists the execution time and the used memory for one iteration running on an NVIDIA GeForce GTX580. Recall that $R = 64$.

size of S	256	512	1024	2048
used memory in MB	<4	14	56	226
iteration time in ms	6	6	25	59
$\mathcal{C}$ DP initialization	9%	19%	29%	51%
$\mathcal{C}$ DP optimization	27%	22%	11%	12%
$\mathcal{D}$ DP initialization	3%	5%	8%	14%
$\mathcal{D}$ DP optimization	28%	22%	10%	8%

In the table above one can notice that the cost of DP initialization grows quickly with S. This is due to the intensive evaluation of $\mathcal{M}_{polar}$ which requires many texture fetches in

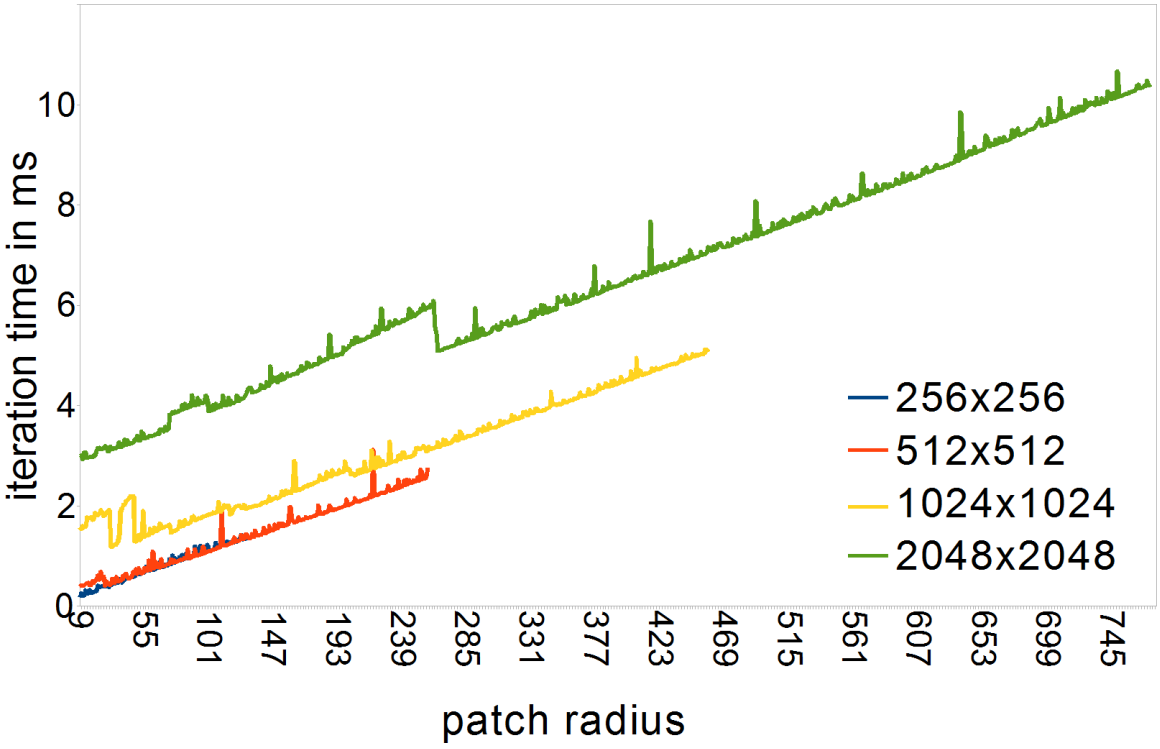

Figure 4.17: Effect of increasing the maximum patch radius R. Each curve corresponds to one resolution of S. The optimization table uses a much larger resolution because of $\mathcal{T}$ and the factors N_ρ and N_θ.

E and S in addition to multiple calls to $\mathcal{T}$. The DP optimization scales well because the number of patches increases when the size of S increases. Increasing the number of patches enlarges the optimization table x axis and therefore increases the degree of parallelism. However for a same S using more patches also requires more memory. The number of patches can be reduced by increasing R. However this increases the optimization table y axis and reduces the degree of parallelism. Figure 4.17 shows the DP performance while R increases.

4.7.9 Comparison with pixel–based synthesis

Our algorithm is roughly ten times slower than fast per–pixel synthesizers due to the number of iterations required before convergence. For instance, our GPU implementation of [LH05] synthesizes the texture of Figure 4.18 in 15 ms using only 4 iterations while our method needs 16 iterations to correctly align features and this requires 112 ms of computation.

Nevertheless, our implementation achieves interactivity and inherits the benefits of patch–based approaches. In particular, the quality achieved by fast per–pixel algorithms largely depends on the amount of jitter added during multi–resolution synthesis. This has to be carefully selected by the user and differs for each texture. Similarly, structured patterns

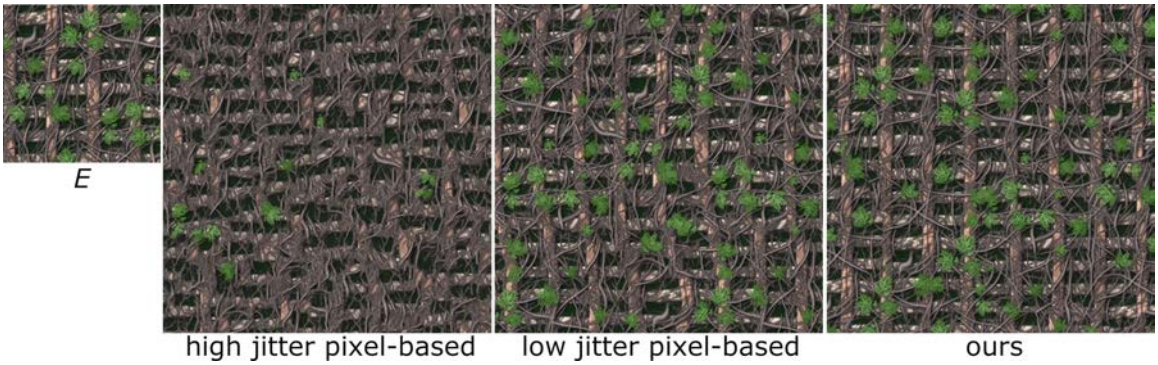

Figure 4.18: Comparison with pixel–based synthesis. *From left to right:* E having a size of 256^2. Pixel–based result computed in 17 ms using a high jitter. Pixel–based result computed in 15 ms using a low jitter. Our result computed in 112 ms using $R = 128$. S has a size of 512^2.

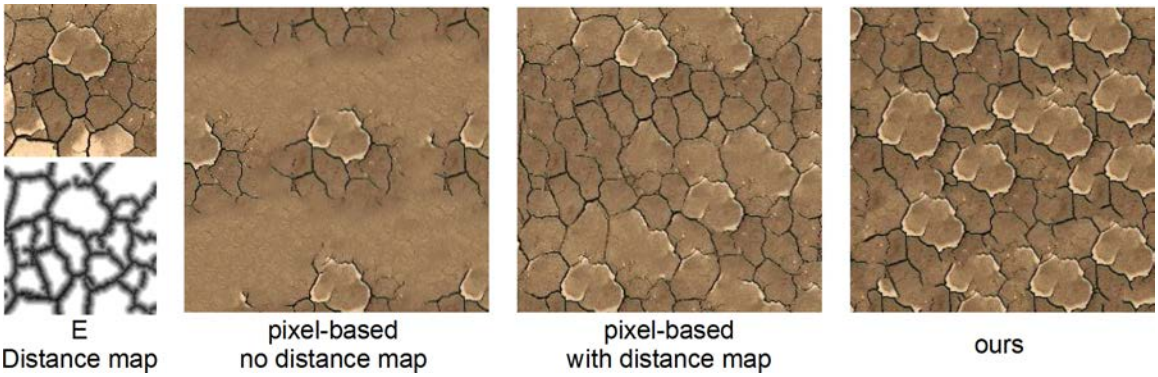

Figure 4.19: Comparison with pixel–based synthesis. Pixel–based results uses the technique of [LH06a]. Our patch–based result does not make use of the distance map. S has size 256×256 in this comparison.

require the addition of a feature distance (see Figure 4.19).

Our algorithm offers superior quality without this requirement. To the best of our knowledge no fast per–pixel algorithm *automatically* reaches the quality we demonstrate on highly regular patterns such as the ones shown in Figures 4.11,4.12.

4.8 Limitations

The main limitations of our synthesizer stem from the random patch selection and placement. Quality–wise, global structures in textures cannot be preserved as shown in Figure 4.20. Performance–wise, despite fast iterations global convergence is slowed–down by the high patch rejection rate as shown in Figure 4.16.

Figure 4.20: Our synthesizer cannot capture global structures like doors. *Left:* E. *Right:* $E[S]$.

4.9 Conclusion

We introduced a parallel patch–based texture synthesis algorithm that quickly achieves high quality results and enables interactive controls. Our algorithm relies on a parallel implementation of an approximate boundary optimization, as well as a patch deformation to align features. A patch rejection scheme ensures a progressively improving synthesis quality.

In the next chapter, we want to render large environments from a small set of texture exemplars. We will use the synthesis algorithm described here to adapt the exemplars to their supporting surfaces. However, since the synthesizer does not support point–wise evaluation, all synthesis results for all surfaces have to be generated prior to rendering and this would quickly consume all available video memory.

To avoid filling all video memory we describe in the next chapter two solutions: The first solution is to slightly modify the synthesis algorithm to generate a very compact representation and describe a scheme that directly decodes texels from the compact representation. The second solution is to use a texture streaming technique that progressively synthesizes textures when needed by the current and the next few frames. These two solutions are complementary and can be combined with other texture synthesis techniques.

Chapter 5

Rendering with synthesized textures

5.1 Introduction

In video game production, artists have to spend a significant amount of time authoring textures and customizing them to best fit their supporting surfaces. This manual authoring has two major limitations when the game environments are large and rich. First, it is costly to customize a texture to every surface in the scene. Second, even if textures can be authored and customized rapidly for many different surfaces, storing in memory a different texture for each surface would be impractical. Because of this, artists have to author a small set of textures that are reused on different surfaces. Since surfaces on which textures are reused have different dimensions, stretching, tiling and cropping are often used to adapt each texture to its supporting surface without duplicating it.

Stretching scales the texture so that its dimensions correspond to the surface dimensions. However, this approach deforms the features of the textures and makes them appear at different scales. The deformation gets even more noticeable when the scaling is not uniform in all directions. Figure 5.1, Middle, shows stretched texture facades. Notice how the result lacks realism due to deformed doors and windows.

An alternative to stretching is to tile a texture if it is smaller than its supporting surface then crop the texture when it gets beyond the surface. The main limitation of cropping is that structures in the textures get cut as shown in Figure 5.1, Right.

Tiling is often used for texturing large terrains by repeating a tillable texture. This saves a considerable amount of memory but repetition artifacts appear at far camera distances as shown in Figure 5.2. Repetition artifacts are usually diminished by modulating the repeated texture with another texture. The other texture may consist of the tiled texture itself at different resolutions or a low frequency noise, e.g. a Perlin noise. The modulation however

Figure 5.1: Three different methods used to adapt textures to their supporting surfaces *Left:* Texture synthesis. *Middle:* Stretching. *Right:* Tiling and cropping.

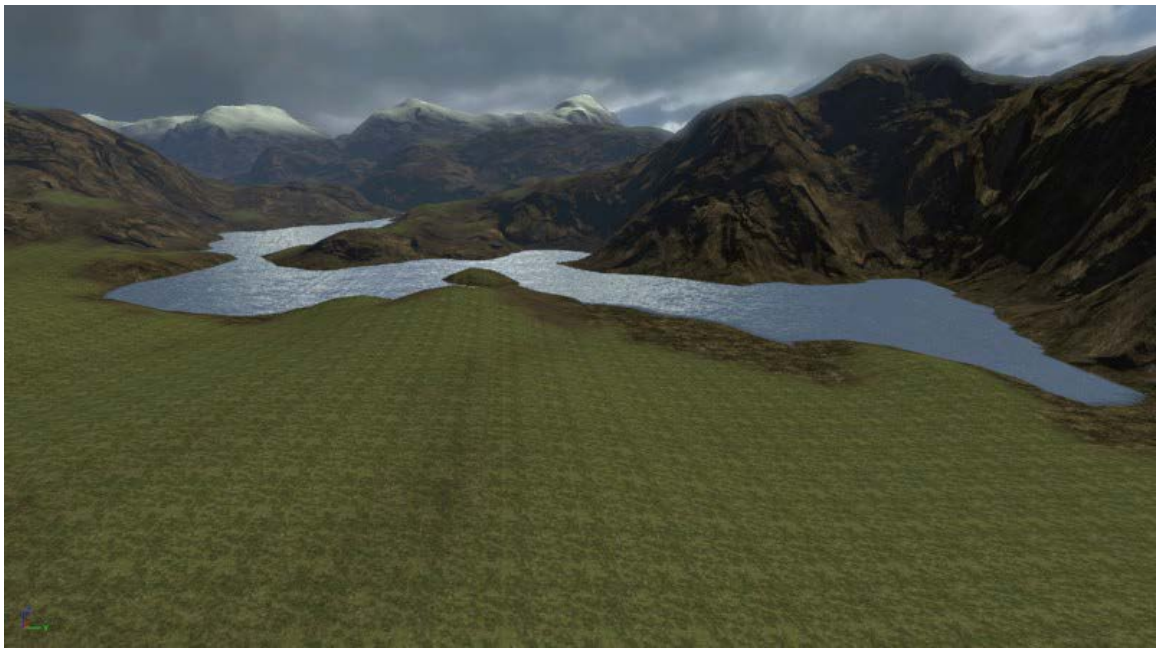

Figure 5.2: Tiling a texture over a terrain often results in repetition artifacts.

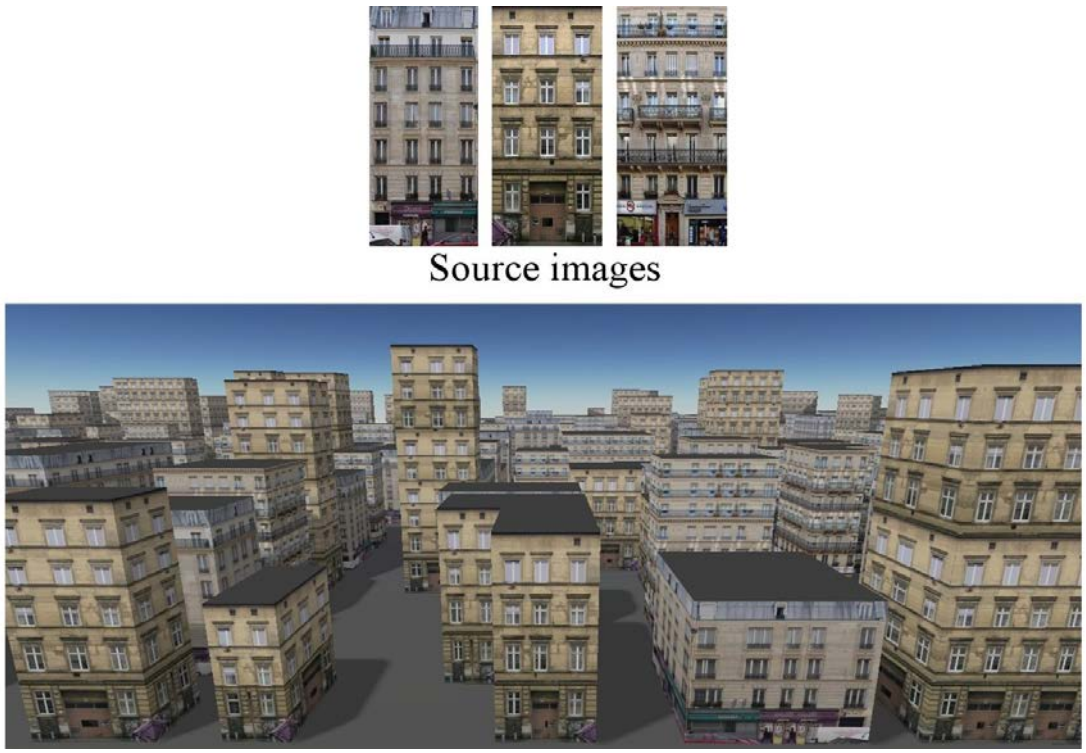

Figure 5.3: Each and every facade of this large city has its own different texture synthesized from one of the three facade exemplars shown at the top. Notice the absence of cut windows and how bricks join well at the corners of each building.

alters the content of the tiled texture losing its details. To reduce the modulation effects, a careful and time–consuming manual intervention is required.

Contribution In this chapter, we use texture synthesis to adapt textures to their supporting surfaces and therefore avoid the unpleasing stretching, cropping or repetition artifacts. We focus mostly on landscapes and city scenes like the one in Figure 5.3. This figure shows a large city where only three texture facades are stored in memory and reused on thousands of different buildings.

Using texture synthesis eliminates the need to a manual texture customization. However, all textures would have to be synthesized prior to rendering if the synthesizer does not allow for point–wise evaluation.

Similarly, in the previous chapter, we have described a fast patch–based synthesizer that would be appropriate for generating many textures from a same exemplar. But due to its lack of point–wise evaluation, one still has to generate all textures and store them in memory before rendering.

To overcome this, we propose to modify the patch–based synthesizer that we have described in the previous chapter so that it generates a compact encoded representation of the result. This representation can be used *directly* during rendering *without* the need to

ever produce the bitmap texture in memory.

We have seen in section 4.8 that the synthesis fails to preserve large scale structures. Because of that, we use this synthesizer for landscapes without highly structured textures and rely on a different patch–based synthesizer that better suits structured textures like the facades in Figure 5.3.

Both synthesizers suffer from the same limitation (lack of point–wise evaluation). We overcome this by storing the results compactly in memory and directly decode texels from the compact representation during rendering. The proposed decoding is fast, allows point–wise accesses and relies on an approximate filtering approach that takes advantage of the hardware for maximum performance. We adapt these ideas to the two synthesizers we use. We also give attention to the particular case of sparse textures. If a texture is sparse, we discuss how to efficiently encode the sparse set of texels in a hash table. The encoding and decoding of synthesized textures are described in section 5.2.

To limit the usage of video memory, we also consider another difficulty: when the number of encoded textures is too high or when synthesized textures cannot be encoded compactly, the video memory would not be enough to store all the textures. To take advantage of the fact that only a small number of textures are usually visible within a short period of time, we study a streaming scheme that synthesizes the textures only when they are needed by the renderer. The main challenge is to correctly predict which textures will be visible in the near future and when to start synthesizing them so that they are in memory when they become visible. The streaming scheme has to be compatible with any texture synthesis technique as long as the synthesis process is fast (a couple of milliseconds per texture). This is often the case with the used synthesizers and this includes most Substance textures. Streaming is discussed in section 5.3.

It should be noted that the discussed streaming approach is work in progress. We only propose a general formulation of the problem for static scenes. A heuristic has been proposed during the internship of Gurprit Singh, a student advised by Sylvain Lefebvre and myself. This heuristic is described in Appendix A.

5.2 Encoding and decoding synthesized textures

To avoid the unpleasing stretching, cropping and repetition artifacts when applying the same texture on different surfaces we use patch–based texture synthesis to adapt textures to their target surfaces. We focus on two synthesizers: First, a patch–based synthesizer that specifically targets architectural textures and that easily captures global structures. Second, we use the patch–based synthesizer that we have described in chapter 4 that better suits stochastic textures and textures with local structured patterns. To avoid any confusion between these two synthesizers, we refer to the former one as *structured synthesizer* and

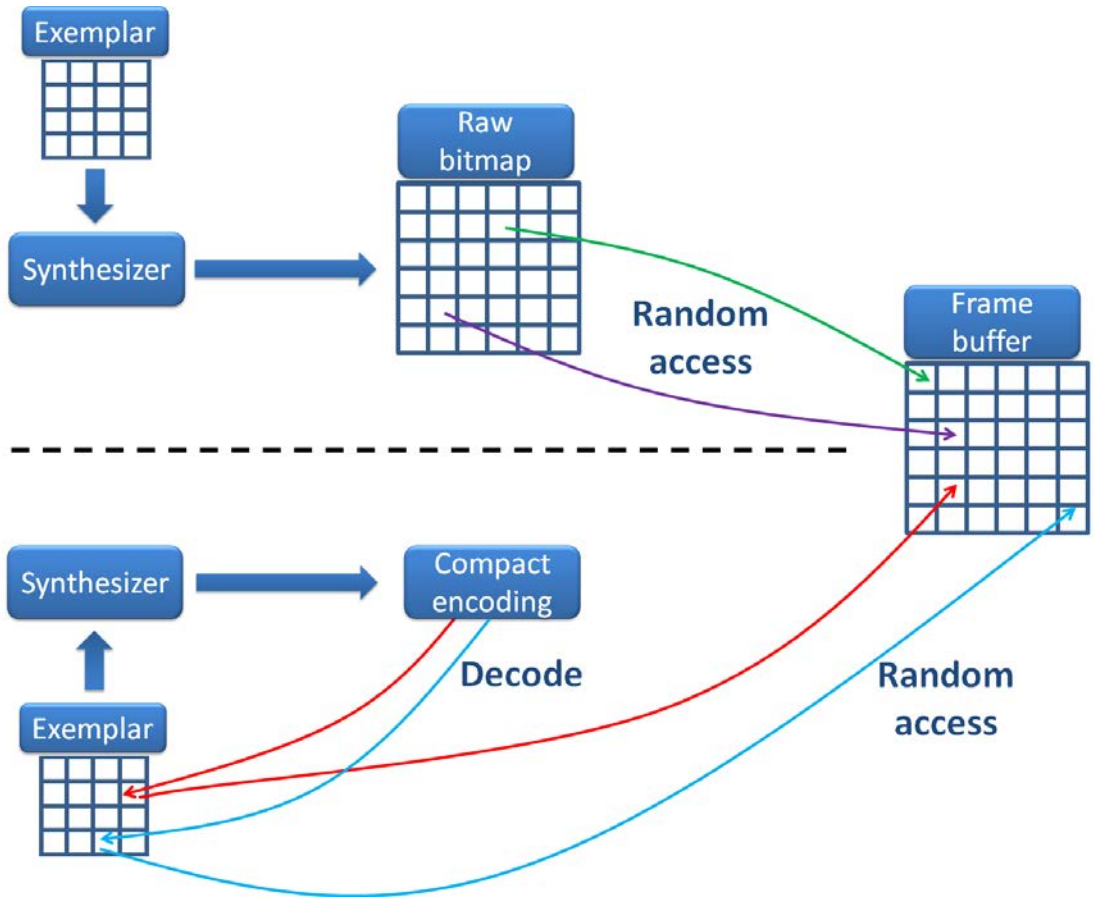

Figure 5.4: *Top:* The synthesizer generates a bitmap texture that is stored entirely in memory. The stored texture costs a considerable amount of memory but it allows a constant time random access to the texels. *Bottom:* We modify the synthesizer so that it generates a compact intermediary representation. A decoding scheme allows a random access to the texels directly from the exemplar.

refer to the latter as *stochastic synthesizer*.

For the structured synthesizer, my contribution focused essentially on the encoding and decoding of the synthesis result. Hence, I will only briefly describe how the synthesizer finds the best layout of patches, and instead focus on the output of the algorithm that we want to encode. The reader is referred to the original paper [LHL10] for details on how to optimally layout patches.

Both synthesizers are fast but still require tens and even hundreds of milliseconds to converge to a high quality result. Therefore, we cannot simply synthesize textures between two rendered frames.

To reduce the memory that is required to store all synthesis results, we propose to encode the result of both synthesizers in a compact intermediate representation and efficiently decode texels from this intermediate representation during rendering (Figure 5.4).

We first describe how to encode and decode the result of the structured synthesizer

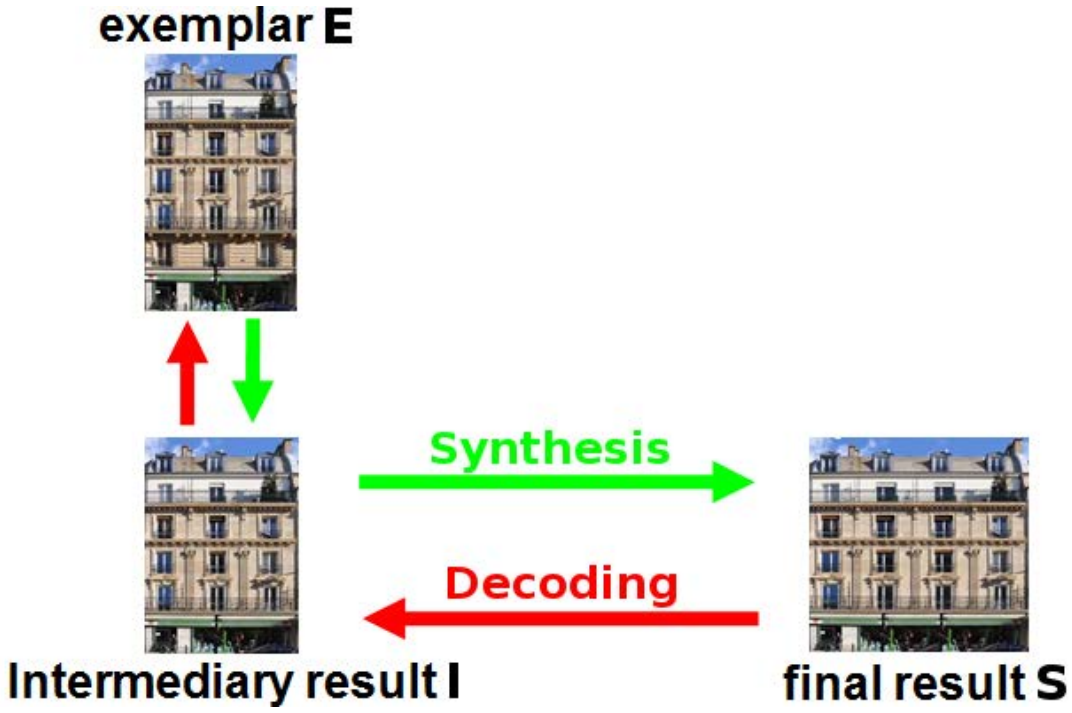

Figure 5.5: First, the synthesizer proceeds vertically to build an intermediate result I. Second, a horizontal synthesis is applied to obtain the final texture S. The decoding is done in reverse synthesis order: First texture coordinates (u_S, v_S) in S are transformed to coordinates (u_I, v_S) in I. The coordinates (u_I, v_S) are then transformed to coordinates $(u_E, v_E) = (u_I, v_E)$ in E. The final texel $S[u_S, v_S]$ is therefore mapped to $E[u_E, v_E]$.

and demonstrate a result rendering large cities with thousands of buildings. We then describe a similar encoding and decoding for the stochastic synthesizer and demonstrate a result rendering a large terrain without any texture repetition artifacts. Finally, for sparse textures, we quickly discuss how to efficiently encode texels in a hash table.

5.2.1 Structured texture synthesizer

The structured synthesizer is a by–example patch–based synthesizer that casts synthesis as a shortest path problem in a graph describing the space of textures that can be synthesized.

To synthesize a texture S of size $W \times H$ from an exemplar E of size $w \times h$, the algorithm proceeds in two steps as illustrated in Figure 5.5. First, it synthesizes an intermediate texture I of size $w \times H$. This synthesis step is done following the vertical direction. Second, the intermediate texture I is used as an exemplar and the final result S of size $W \times H$ is synthesized by repeating the same process as in the first step but this time following the horizontal direction.

In each step, the synthesizer cuts horizontal (resp. vertical) strips in the exemplar and pastes them side by side in a different order. This is illustrated along the horizontal direction in Figure 5.6.

Figure 5.6: The algorithm synthesizes textures by reordering strips of the exemplar. Each result is a path in a graph describing the space of synthesizable textures.

The source image is searched for repeated content in the form of horizontal (resp.vertical) parallel cuts along which similar colors are observed. Two such parallel cuts are indicated by the green/blue cuts in Figure 5.6. Many such parallel cuts are extracted from the exemplar image. Any pair of non–crossing cuts defines a strip, and the strips can be assembled with a number of other strips like in a jigsaw puzzle. The parallel cuts create many choices in the way strips may be assembled. This is the source of variety in the synthesized textures.

Intuitively, given a target size the algorithm automatically determines the best choice of strips to both reach the desired size and minimize color differences along their boundaries. Despite the apparent combinatorial complexity of the problem, given a set of cuts the algorithm is able to solve efficiently for it using a graph–based formulation. The optimization reduces to a simple shortest–path computation solved using Dijkstra's algorithm. The solution path describes the sequence of cuts that form the boundaries of each strip. Constraints can be set on the used graph giving the user many controls, for instance, he can force a cut or an object to appear at a desired location in S.

Synthesis output

The algorithm outputs a list of strips that when assembled together, produce the intermediary result I. The algorithm also outputs another list of strips within the intermediary result I that when assembled together, produce the final synthesis result S.

Each strip is represented by a pair of cuts: The first cut contains the starting coordinates of the strip in the exemplar. It is shown in green in Figure 5.6. The second cut contains

the ending coordinates of the strip in the exemplar. It is shown in blue in Figure 5.6.

Cuts We give here a definition for the cuts. We only define vertical cuts used when synthesizing S from the intermediary exemplar I following the horizontal direction. The very same reasoning can be used to deduce the definition of horizontal cuts.

A vertical cut is a list $\mathcal{C}$ of length H. We note $\mathcal{C}[y] \in [1, .., w]$ the y'th entry in the list $\mathcal{C}$. $\mathcal{C}$ satisfies the following constraint:

$$\forall\ y\ \in [1, .., H-1],\ |\mathcal{C}[y] - \mathcal{C}[y+1]| \leq 1$$

The couple $(\mathcal{C}[y], y)$ represents valid coordinates in I. The set $(\mathcal{C}[y], y)\ \forall\ y$ represents a Y–monotone curve that defines the left boundary or the right boundary of a vertical strip in I. We note $\mathcal{C}_l$ the left boundary and $\mathcal{C}_r$ the right boundary.

The output of the synthesizer is an ordered list $\mathscr{C}$ of N cuts:

$$\mathscr{C} : \{\mathcal{C}_l^1, \mathcal{C}_r^1,\ \ \mathcal{C}_l^2, \mathcal{C}_r^2,\ \ \mathcal{C}_l^3, \mathcal{C}_r^3,\ \ \ldots\ ,\ \ \mathcal{C}_l^N, \mathcal{C}_r^N\}$$

To distinguish between the set of horizontal cuts output in the first synthesis step and the set of vertical cuts output in the second step, we note $\mathscr{C}_h$ the list of horizontal cuts and $\mathscr{C}_v$ the list of vertical cuts.

Encoding

Our goal is to generate, from a same source exemplar, a unique texture for many different surfaces. A good example of this scenario is the virtual city of Figure 5.3 where thousands of facades are synthesized from only three exemplars.

The output cuts and the exemplar suffice to represent S. The exemplar needs to be stored once while a different set of cuts has to be stored for each synthesized texture.

The storage requirements for the cuts may be considerable if the number of synthesized textures is too important. It is therefore crucial to store the cuts as compactly as possible and yet, make sure that the decoding phase is fast and is able to randomly access the encoded information. We do so according to three observations:

Observation 1: The overall shape of the cuts is roughly vertical (Figure 5.6). If we note $\mathcal{C}_{min}$ the minimum coordinate in $\mathcal{C}$, that is $\mathcal{C}_{min} = \min_{y \in [1,..,H-1]}(\mathcal{C}[y])$ and note $\mathcal{C}_{max}$ the maximum coordinate in $\mathcal{C}$, that is $\mathcal{C}_{max} = \max_{y \in [1,..,H-1]}(\mathcal{C}[y])$ then in general, the value of $\mathcal{C}_{max} - \mathcal{C}_{min}$ will be small compared to $\mathcal{C}_{min}$ or $\mathcal{C}_{max}$.

We note $\mathcal{C}'$ the cut that contains offsets relative to $\mathcal{C}_{min}$. It is computed as:

$$\forall\ y\ \in [1, .., H-1],\ \mathcal{C}'[y] = \mathcal{C}[y] - \mathcal{C}_{min}$$

The original cut $\mathcal{C}$ can be trivially recovered from $\mathcal{C}'$ and $\mathcal{C}_{min}$ as follows:

$$\forall\ y\ \in [1, .., H-1],\ \mathcal{C}[y] = \mathcal{C}'[y] + \mathcal{C}_{min}$$

Since for any y we have $\mathcal{C}'[y] \leq \mathcal{C}_{max} - \mathcal{C}_{min}$, the value of $\mathcal{C}'[y]$ would be relatively small and hence, the information in $\mathcal{C}'$ would require fewer bits than $\mathcal{C}$.

As $\mathcal{C}$ can be recovered from $\mathcal{C}'$ and $\mathcal{C}_{min}$, we only store $\mathcal{C}_{min}$ and $\mathcal{C}'$ in memory. 16 bits are enough to store $\mathcal{C}_{min}$ and in all our results, 8 bits suffice to store each offset in $\mathcal{C}'$.

Furthermore, by noting that $|\mathcal{C}'[y] - \mathcal{C}'[y+1]| \leq 1$ only the first offset in $\mathcal{C}'$ can be stored and two bits representing the values $-1, 0, 1$ can be used to store how $\mathcal{C}'[y+1]$ can be obtained from $\mathcal{C}'[y]$. This means that a loop has to read all offsets preceding y to determine the value of $\mathcal{C}'[y]$. The loop may be slow and may increase bandwidth requirements.

A better compromise between speed and compression is to decompose the cut $\mathcal{C}'$ into smaller segments, store the first offset of the segment using 8 bits and the remaining offsets of the segment using 2 bits. By carefully choosing the size of the segment, typically 8 bits for the first offset and 24 bits for the next 12 offsets, bitwise operators (if available) can be used to access the data in constant time and more importantly, bandwidth can be significantly improved.

Recently, Nystad et al. [NLP+12] proposed *Bounded Integer Sequence Encoding*. Rather than storing 8 bits for the first offset and 24 bits for the next 12 offsets, we can use 8 bits for the first offset and use 24 bits to store 15 offsets. Precisely, each group of 5 offsets can be encoded in 8 bits rather that 10 bits. The 5 offsets can be represented by the chain $t_0t_1t_2t_3t_4$ where t_i takes the value -1, 0 or 1. The group of 5 offsets can take $3^5 = 243$ possible values meaning that 8 bits suffice to store $t_0t_2t_3t_4$. The 8 bit value B is simply obtained by adding 1 to each offset t_i and considering the result as a base–3 number: $B = (t_0+1)3^0 + (t_1+1)3^1 + (t_2+1)3^2 + (t_3+1)3^3 + (t_4+1)3^4$. $t_0t_2t_3t_4$ is recovered by re–interpreting B in base–10 then subtracting 1. While this would further improve our result, we currently do not use this technique.

Observation 2: The shape of the cut $\mathcal{C}_r^i$ is equivalent to the shape of the cut $\mathcal{C}_l^{i+1}$ (Figure 5.6) meaning that $\mathcal{C}_r'^i$ is exactly equal to $\mathcal{C}_l'^{i+1}$. Only the value of $\mathcal{C}_{min}$ differs between $\mathcal{C}_r^i$ and $\mathcal{C}_l^{i+1}$. Therefore, only the right boundaries $\mathcal{C}_r'$ need to be stored along the full list of $\mathcal{C}_{min}$ values.

The very first left boundary $\mathcal{C}_l'$ has to be stored as well but in our implementation the starting column and row are constrained to coincide with the starting column and row of the exemplar: the very first left boundary $\mathcal{C}_l'$ is not needed as it only contains zero offsets.

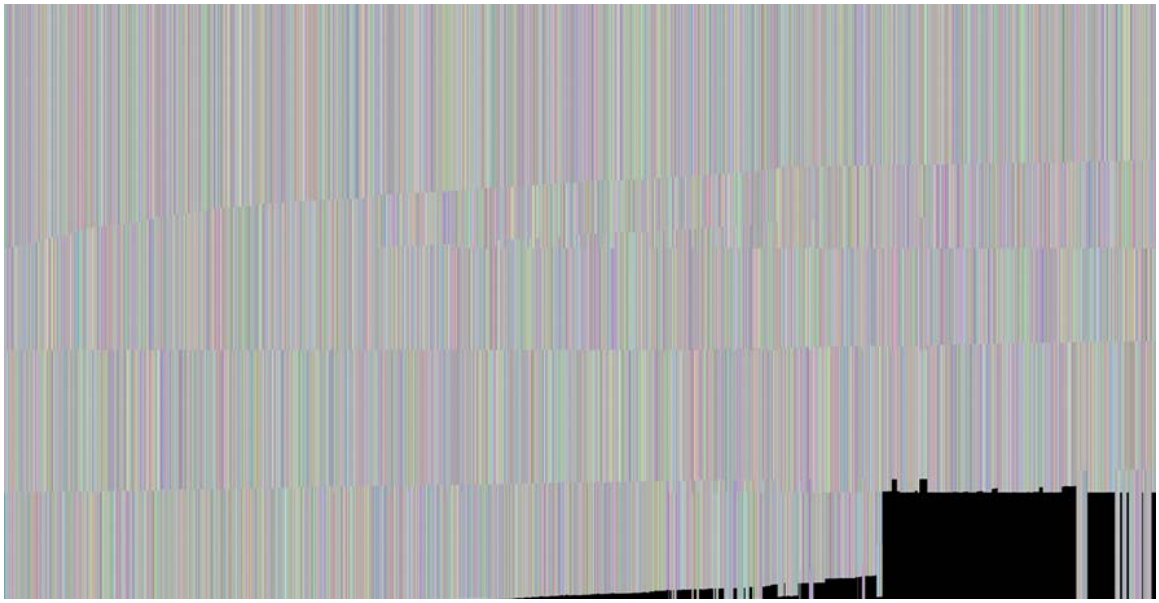

Figure 5.7: packing the global cut table in a 2D texture. Each colored segment represents a cut. This packing corresponds to the cuts stored for the city of Figure 5.3.

Observation 3: The entire set of cuts is precomputed then used during the patch layout optimization to synthesize different textures depending on the output size and other constraints set by the user. In the scenario of a large virtual city like the city in Figure 5.3, the precomputed cuts are reused many times for different facades. We therefore eliminate duplicated cuts by storing a unique instance of each cut $\mathcal{C}'$ in a global cut table shared for all facades. A list of indices for each facade indicates which cuts have to be read from the global cut table.

In case of old hardware that only support texture reads and not linear memory reads from the pixel shader, we pack the global cut table in a 2D texture by repeating the best fit decreasing packing heuristic [SL94] for different texture ratios then keeping the solution with the least waste of space. Figure 5.7 shows a global cut table packed in a 2D texture.

Decoding

During rendering, a pixel shader is executed at every pixel to determine its color. Each pixel is associated with coordinates (u_S, v_S). With standard texturing the texture image stored in memory is sampled at (u_S, v_S) to obtain the pixel color as described in section 2.2.

Our goal is to find the color of each pixel using only the cuts, without ever having to decode the entire result in memory. In each pixel we thus must directly find the coordinates (u_E, v_E) in the exemplar that correspond to coordinates (u_S, v_S) in the result S.

When performing a two steps synthesis, it is easy to see that the horizontal (resp. vertical) step changes only the x (resp. y) coordinate of the pixel. Computations on u and

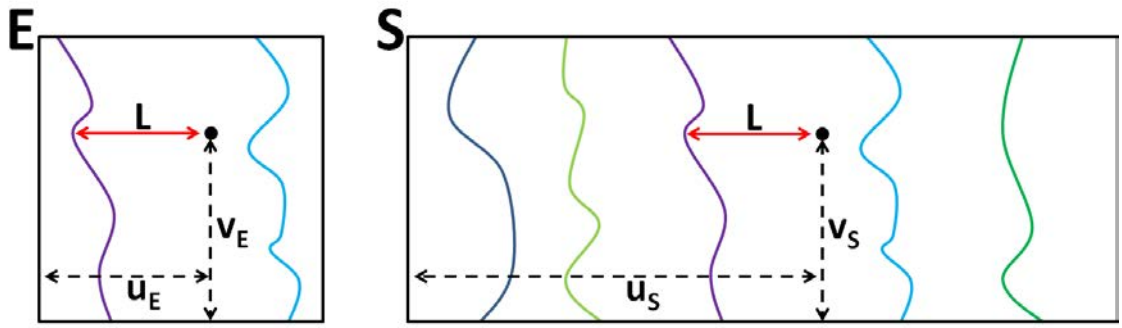

Figure 5.8: A same version of a strip shown in S and E.

v must however be performed in reverse synthesis order to take into account the intermediate result I (Figure 5.5). We obtain u_E from (u_S, v_S) and then v_E from (u_E, v_S).

In a given direction, we first locate the strip enclosing the lookup coordinate in the result. We perform a binary search using the positions of the stored cuts. From there, we easily locate the source coordinates using the left strip border (Figure 5.8).

The following pseudo–code summarizes the decoding of texels. For simplicity, we ignore the encoding of the cuts in the pseudo–code.

```
color textureLookup(u_S, v_S)
{
  float u_E = findCoordinateInSource( C_h, u_S, v_S );
  float v_E = findCoordinateInSource( C_v, v_S, u_E );
  return E[u_E, v_E];
}

float findCoordinateInSource(C, x, y)
{
  (C_l,C_r) = findStripEnclosing( C, x );
  L         = distanceToLeftStrip( C_l, x );
  return C_l[y] + L;
}
```

The function `findStripEnclosing` takes a position x and a list of strips as input and finds the strip enclosing x using binary search. The strip is returned as a pair $(\mathcal{C}_l, \mathcal{C}_r)$. The function `distanceToLeftStrip` returns the distance L to the left cut $\mathcal{C}_l$ of the found strip. L is shown in Figure 5.8.

Filtering When rendering from a pixel shader, it is important to ensure that the result is filtered: the texture must be smoothly interpolated at closeups and properly blurred when

seen at a distance. This is typically done using the GPU tri–linear and anisotropic filtering mechanism (section 2.2.2). However, the filtering mechanism breaks around cuts since it is unaware of the discontinuities and visible seams appear (texels on each side of a cut are not necessarily neighbors in the exemplar).

One approach would be to re–implement the tri–linear interpolation mechanism. However, while feasible, this would be very slow. Furthermore, anisotropic filtering which is an important feature when rendering flat surfaces such as walls and facades would be much slower if implemented manually.

To circumvent the problem, we sample four additional texels having the same coordinates as the current texel but belonging to the previous and the next strips in both horizontal and vertical directions. The five resulting texels are sampled with hardware filtering enabled. These five texels are then blended depending on the MIP–map level and the distance between the current texel and the four borders of the current patch (a patch is formed from the intersection of a horizontal and a vertical strip). This blending constitutes an approximation of the exact filtering but gives good results in practice.

For simplicity, we now limit the filtering explanation to only the horizontal direction. In this case, only three texels are blended rather than five. These three texels are shown in Figure 5.9.

We note (u, v) the current texel that lies on the i'th strip. This strip is defined by its starting and ending cuts $(\mathcal{C}_l^i, \mathcal{C}_r^i)$. We note (u_l, v) and (u_r, v) the two texels that are respectively lying on strips $i-1$ and $i+1$ and that will be blended with (u, v). Note that the v coordinate is the same for the three involved texels since we are only considering the horizontal synthesis step. (u_l, v) and (u_r, v) are computed as follows:

$$u_l = \mathcal{C}_r^{i-1}[v] + L_1$$

$$u_r = \mathcal{C}_l^{i+1}[v] - L_2$$

L_1 is the distance in texels for the current MIP–level from the current texel (u, v) to the previous cut and L_2 is the distance in texels for the current MIP–level from the current texel to the next cut. L_1 and L_2 can be seen in Figure 5.10.

A weighting factor w_l, w_r and w are respectively computed for (u_l, v), (u_r, v) and (u, v) as follows:

$$w_l = \max(1 - L_1, 0)$$

$$w_r = \max(1 - L_2, 0)$$

$$w = 1 - \max(w_l, w_r)$$

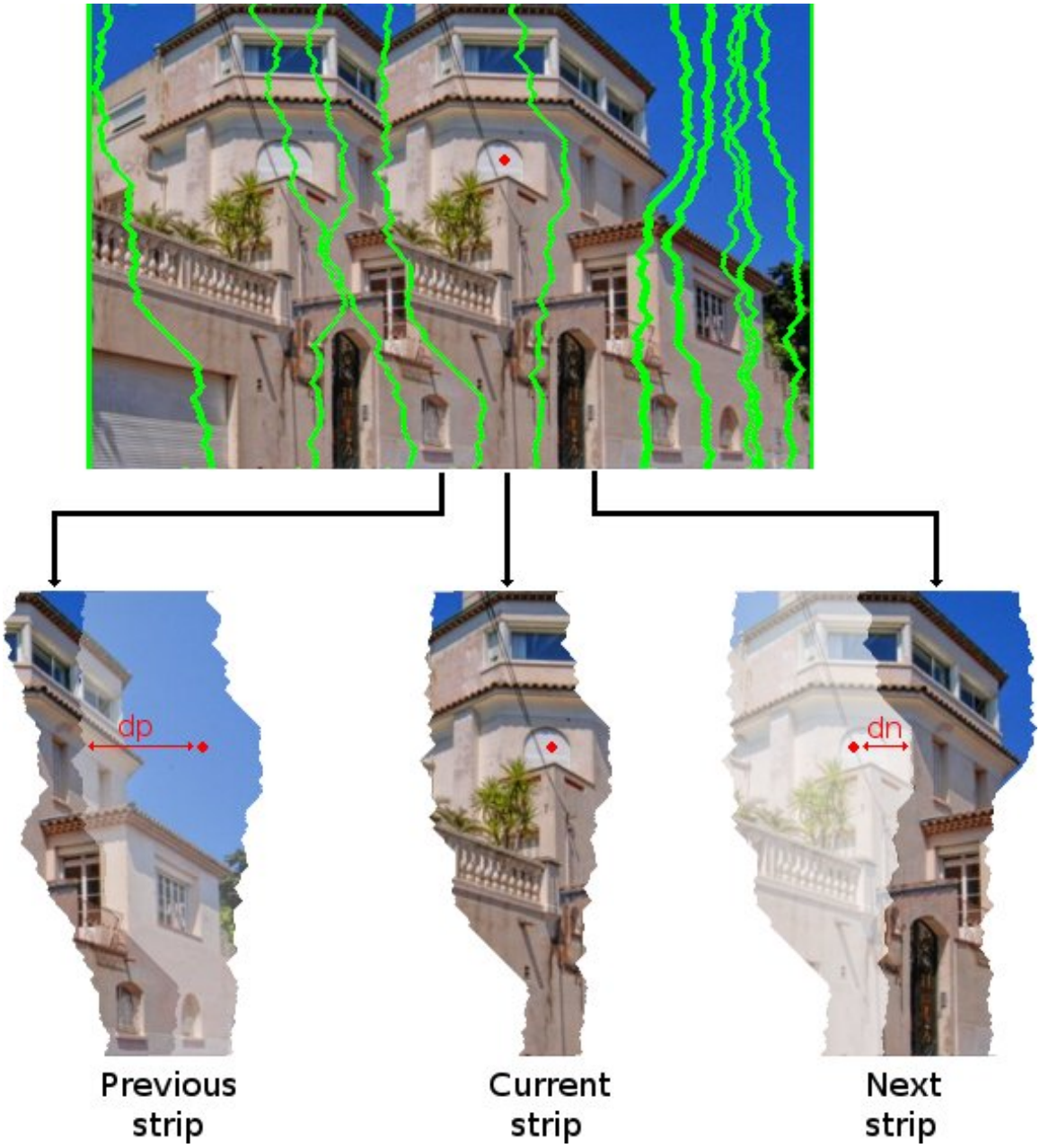

Figure 5.9: Filtering, the red dot texel in the current strip is blended with the two superimposed texels in the previous and next strips. The blending factor depends on the MIP–map level and the distance dp and dn.

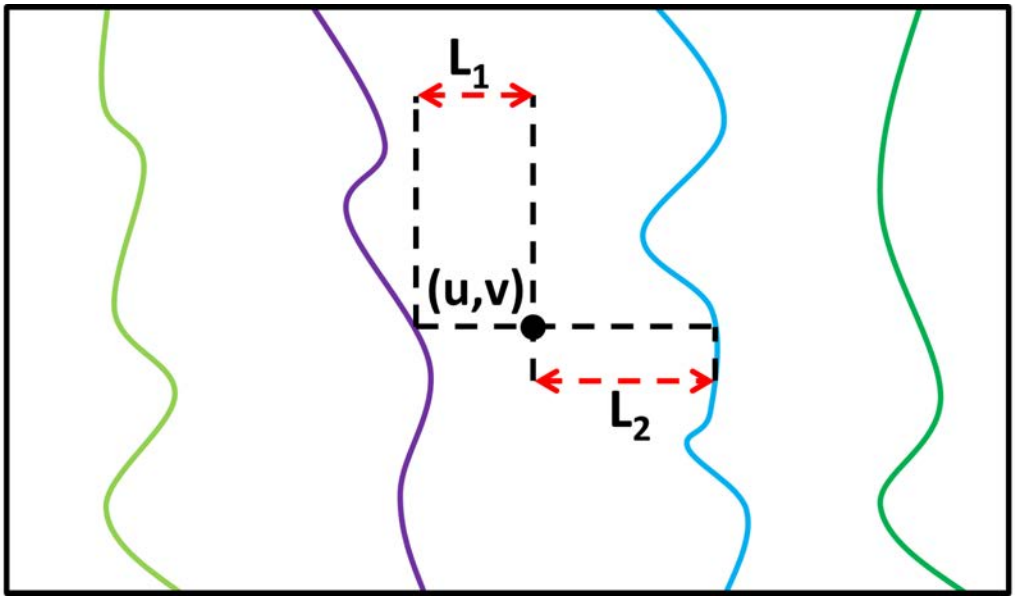

Figure 5.10: Distance to the previous and the next cuts

The final color c that follows from the blending of the three texels is computed as:

$$c = \frac{w_l \times E[u_l, v] + w_r \times E[u_r, v] + w \times E[u, v]}{w_l + w_r + w}$$

Intuitively, when $L1 \geq 1$ and $L2 \geq 1$, $c = E[u, v]$. When $L1 \leq 1$ and $L2 \geq 1$, c is a linear interpolation between $E[u_l, v]$ and $E[u, v]$. When $L2 \leq 1$ and $L1 \geq 1$, c is a linear interpolation between $E[u_r, v]$ and $E[u, v]$. Finally when $L1 \leq 1$ and $L2 \leq 1$, the size of the projected strip $(\mathcal{C}_l^i, \mathcal{C}_r^i)$ is less than a pixel. In that case, as an approximation the current texel weight is assigned the minimum weight among w_l and w_r, that is, $w = 1 - \max(w_l, w_r) = \min(w_l, w_r)$. The blending is done with more contribution to the texel with the highest weight and with less contribution to the two other texels that have the same low weight.

Please note that when multiple strips are projected within one pixel, more samples belonging to further strips would have to be considered for blending. However we do not consider this case as fetching data from more strips would become very slow.

This filtering approximation does not produce *visible* artifacts in practice, even though approximation errors still exist at coarsest levels of detail. Figure 5.11 shows the quality of the proposed filtering approach. The approach provides very good results for the first levels of detail and enables correct anisotropy. The coarsest MIP–map levels however, show some discontinuities due to multiple strips within the filter width. The discontinuities are only visible if more that one strip are projected within one screen pixel. In practice, these discontinuities happen at very far distances and are usually unnoticeable.

Such filtering works well in many situations besides RGB colors. Figure 5.12 shows a

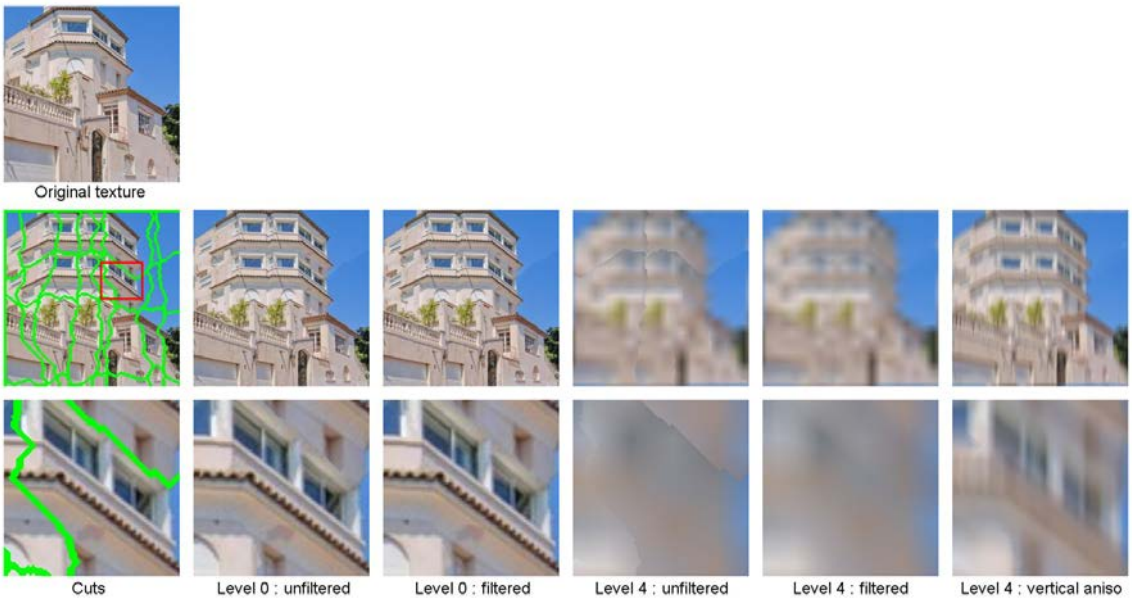

Figure 5.11: Filtering quality. *Top:* Original texture. *Middle:* Enlarged texture with different levels of detail and filtering modes. *Bottom:* Zoom–in the marked red rectangle.

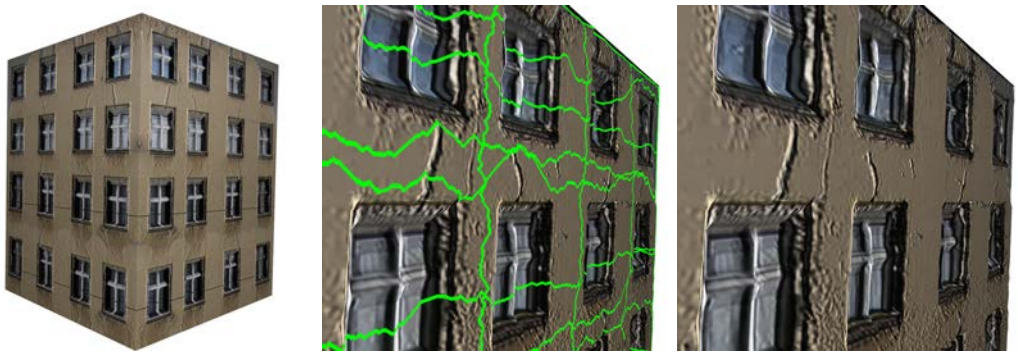

Figure 5.12: Decoding and filtering with parallax occlusion mapping. *Left:* Rendering with parallax occlusion mapping. *Middle:* Zoom–in showing cuts in green. *Bottom:* Zoom–in without showing cuts. Note how the result is correctly filtered along the cuts. The parallax mapping is computed as described by Tatarchuk [Tat06].

parallax occlusion effect using synthesis of texels containing color and relief information.

Performance

The rendering of the city of Figure E.5 reveals the key advantage of our approach compared to standard image resizing techniques. Results are obtained on an NVIDIA GeForce GTX580. Since each synthesis result is compactly encoded, we are able to customize a facade texture to each and every surface of an entire city. The city contains 15131 facades (3348 buildings), using 8 different exemplars stored in 7 MB. Storing a texture for each facade would require 12.1 GB of GPU memory. Instead, using our scheme (without *Bounded*

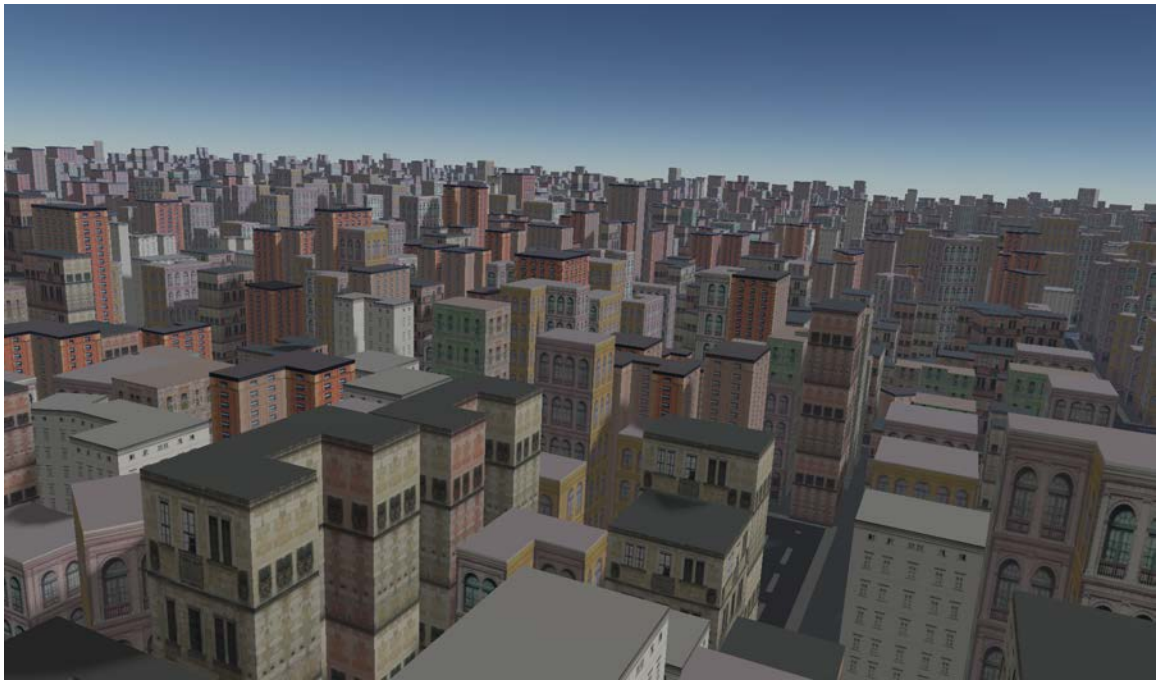

Figure 5.13: A city containing a unique texture per facade.

Integer Sequence Encoding), each facade gets a unique texture and all facades occupy 92 MB of GPU memory (excluding the source images); two orders of magnitude less. The textures are decoded in every pixel, from the pixel shader. The entire city renders at 201 FPS on average at 1600×1200 resolution. Without filtering, the city renders at 317 FPS. Using stretching rather than synthesizing a separate texture to fit each facade increases the frame rate to 733 FPS on average but reduces quality as shown in Figure E.6.

5.2.2 Stochastic texture synthesizer

We now describe how to use a similar encoding and decoding scheme for the stochastic synthesizer that we have described in chapter 4.

Unlike the structured synthesizer that represents the result with non–overlapping patches from the exemplar, the stochastic synthesizer is an iterative random process that constructs the result by overlapping patches. However, recall that the patches of a same iteration do not overlap. We made this possible by overlaying the output S with a virtual grid such as each cell encompasses one patch (Figure 5.15). Since cells in the virtual grid do not overlap, patches can be processed independently.

Assuming a toroidal synthesis result, the grid origin o_g randomly changes between iterations. o_g is computed with a hash function (Jenkins lookup3 mix function [Jen96, Jen06]) that is computed from the iteration index. For each patch $\mathcal{P}$ contained in a cell, its origin location o_e in the exemplar E is determined with the same hash function but this time

Figure 5.14: *Left:* The exemplars are simply stretched to fit the facades. *Right:* The synthesizer adapts the exemplars to best fit the facades. This greatly improves quality but rendering is more than 3 times slower.

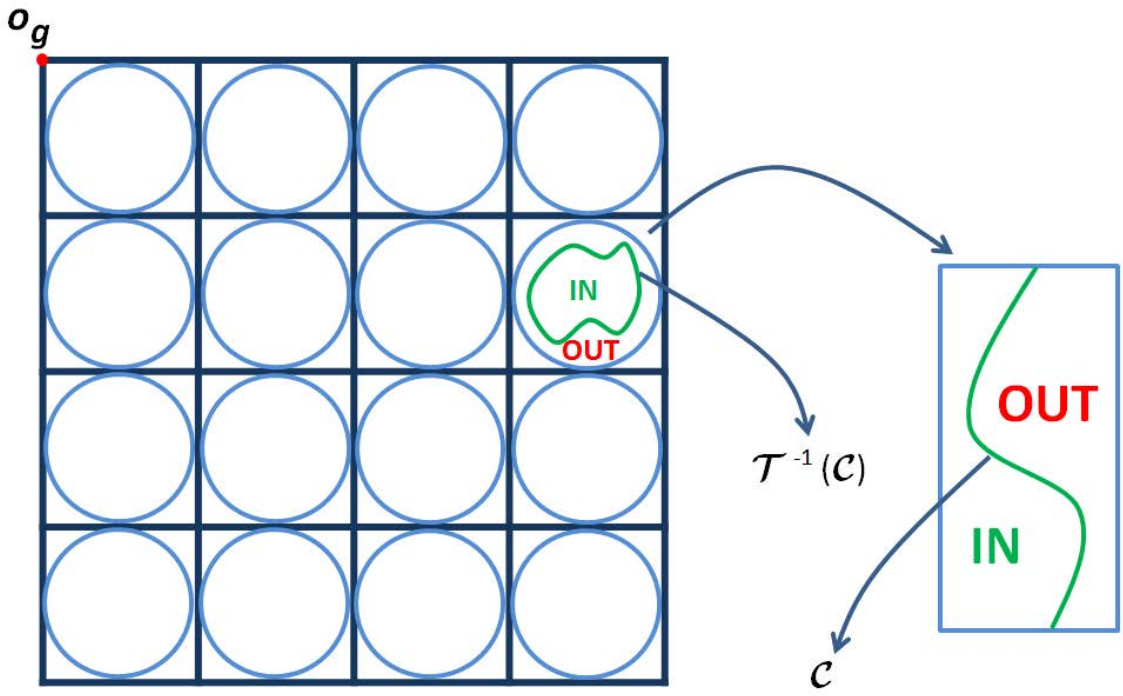

Figure 5.15: Quick overview of the stochastic synthesizer described in Chapter 4.

seeded with the iteration index and the cell coordinates. The boundary of $\mathcal{P}$ is optimized in its polar version $\mathcal{P}_{polar}$. Recall that we have defined the boundary $\mathcal{C}$ in $\mathcal{P}_{polar}$ and used the notation $\mathcal{T}^{-1}(\mathcal{C})$ to refer to the actual boundary of $\mathcal{P}$. We have also deformed the patch $\mathcal{P}$ to align features along the cut $\mathcal{C}$ by offsetting its coordinates. An array $\mathcal{D}$ containing indices in $\mathcal{C}$ gives the final cut coordinates after deformation.

Encoding

To encode the result of one iteration, we need to store for every patch the cut $\mathcal{C}$ and its indices $\mathcal{D}$ that describe the deformation along the cut. We also need to store the iteration index to determine o_g and o_e using the hash function.

We compress $\mathcal{C}$ and $\mathcal{D}$ similarly to the structured synthesizer. However, unlike the structured synthesizer $\mathcal{C}$ is not Y–monotone. Only $\mathcal{D}$ is Y–monotone. $\mathcal{D}$ is therefore stored similarly to the structured synthesizer cut. $\mathcal{C}$ may move by more than one pixel to the left or to the right. Precisely, we have $|\mathcal{C}[y] - \mathcal{C}[y+1]| \leq J_{max}$ where J_{max} is constant set by the user. Recall that J_{max} is typically small (between 8 and 32) and therefore, we directly encode every $\mathcal{C}[y]$ using 5 bits assuming that $J_{max} < 32$.

Removing unnecessary cuts Encoding then storing the information of all synthesis iterations can be expensive especially if the number of iterations is high. Also, this is wasteful since many iterations do not contribute to the final result. This is especially the case of the first iterations that are likely to be hidden by the result of subsequent iterations.

To avoid storing information from iterations that do not contribute to the final result, we locate during synthesis the iterations that do contribute to the final result then after synthesis we only encode cuts belonging to the located iterations. We do so following 3 steps:

1) We use a map I that has the same size as S and where $I[x, y]$ stores the index of the last synthesis iteration that did update the pixel (x, y) in S. This means that for the pixel (x, y) in S, $I[x, y]$ contains the iteration responsible of the final color at (x, y). I is simply constructed during synthesis as follows: Every time a pixel (x, y) is updated (lying inside the current patch), $I[x, y]$ is assigned the current iteration index.

2) After synthesis, I only contains indices of iterations contributing to the the final result. However, the indices are the same across many different pixels. We reduce the content of I to a small set of unique indices using a parallel reduction algorithm combined to an atomic reduction [HPLVdW09]. This way, I becomes a small table of few entries containing unique indices and representing iterations contributing to the the final result.

3) Finally, we only encode the result of iterations present in I and ignore all other iterations since they do not contribute to the result.

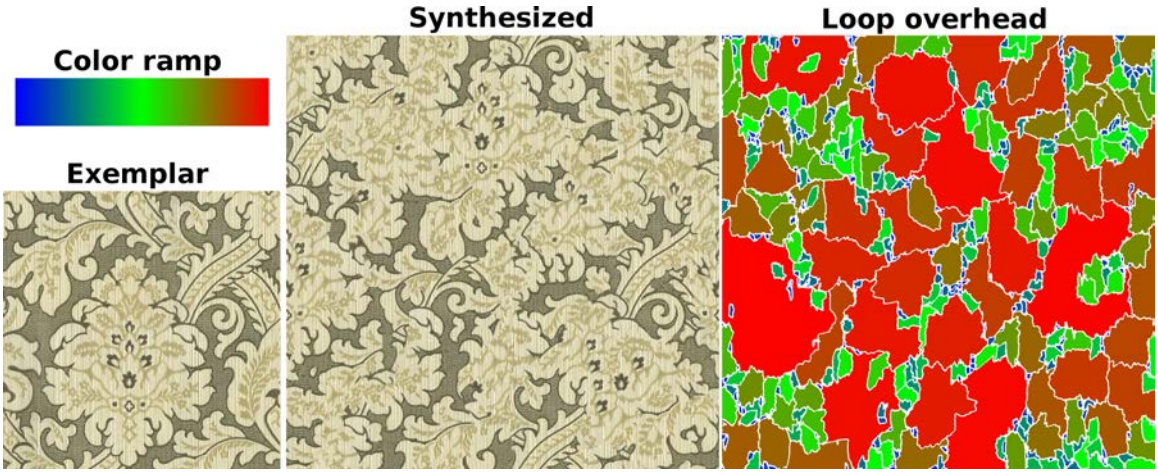

Figure 5.16: To decode a pixel in the synthesized result, a loop has to check the synthesis iterations in backward order until finding the the patch containing the pixel. The number of iterations in the loop is shown in the loop overhead map on the right. White contours represent patch borders. The red color tones in the color ramp represent high values. Conversely, the blue tones in the color ramp represent small values. This means that red regions in the loop overhead map require more iterations than green regions and green regions require more iterations than blue regions.

Decoding

For the result of a given iteration, the pixel color at any location is determined in constant time as follows: First, we determine in which cell the pixel is contained. For the patch $\mathcal{P}$ that is contained in the cell, we check whether the pixel is inside or outside the boundary $\mathcal{C}$ of $\mathcal{P}$. If the pixel is outside, it keeps its color otherwise, the pixel is inside $\mathcal{C}$ and its coordinates are relocated and smoothly displaced to take into account the deformation. Finally, the pixel color is fetched from the exemplar E.

To consider all iterations, the pixel is tested for each iteration in backward synthesis order: If the pixel is inside the patch of the current iteration, its value is fetched from the exemplar and we *stop* (Figure 5.16). Otherwise, the same test is repeated for the next iteration (the previous iteration in forward synthesis order).

Figure 5.16 shows a decoded synthesis result with a map illustrating the number of iterations required until finding the color of each pixel. In this map, red color tones represent a large number of iterations while blue color tones represent a small number of iterations. Pixels requiring few iterations belong to patches pasted during the latest iterations of the synthesis step. We clearly notice in the figure that these pixels belong to few small patches. The reason the patches are small is that during the latest iterations of the synthesis step the result already converged to a good quality: Large patches with long boundaries are more likely to produce errors and are therefore rejected. The rejection rate (section 4.5) gets higher and only parts of the result are updated increasing the decoding loop overhead

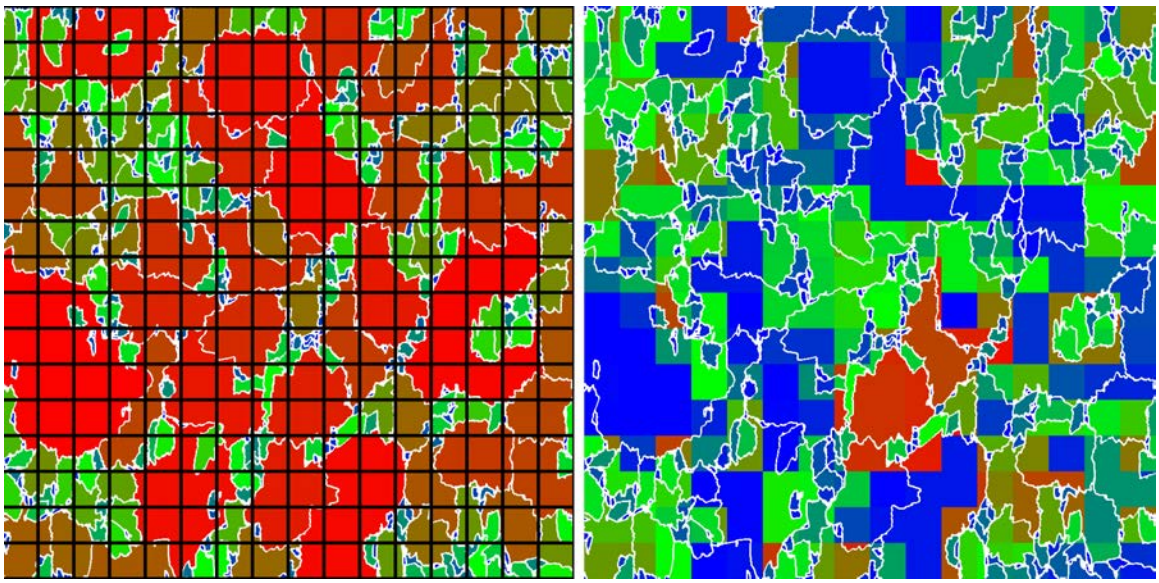

Figure 5.17: Reducing the decoding loop overhead of Figure 5.16. *Left:* The synthesis result S is overlaid with a coarse grid G. *Right:* In each cell of G the index of the first iteration contributing to the result is computed. This index is used as the starting index of the decoding loop. This results in a larger amount of blue colors in the map on the right meaning that the overall loop overhead is significantly reduced.

for pixels that are not updated during the latest synthesis iterations. We next describe how to reduce the decoding loop overhead.

Reducing the decoding loop overhead To reduce the number iterations in the decoding loop, we overlay on top of S a coarse grid G having a typical size $w \times h = 16 \times 16$ (Figure 5.17, Left). The ratio between S and G is $w_r = \frac{W}{w}$ and $h_r = \frac{H}{h}$. Each cell in this grid contains an 8 bits value storing the index from which the loop should start. This value is computed as the index of the first synthesis iteration contributing to the result. If this index cannot be stored with 8 bits, we use a 255 value. Concretely, G is computed as follows:

$$\forall x, y \in G,\ G[x,y] = \min_{i \in \{x \times w_r, \ldots, (x+1) \times w_r\},\ j \in \{y \times h_r, \ldots, (y+1) \times h_r\}} I[i,j]$$

This way, to decode pixel (x, y), the loop starts at $G[\lfloor \frac{x}{w} \rfloor, \lfloor \frac{y}{h} \rfloor]$ and stops when the pixel is lying inside a patch. This reduces the decoding loop overhead as shown in Figure 5.17.

Decoding complexity This decoding approach has an $O(N)$ time complexity where N is the number of iterations in the decoding loop. Despite an $O(N)$ complexity, the decoding approach remains reasonable in most situations where the synthesis only runs 16 to 32 iterations to converge to a good result. In the case where more synthesis iterations are used, trimming the iterations that do not contribute to the result significantly reduces

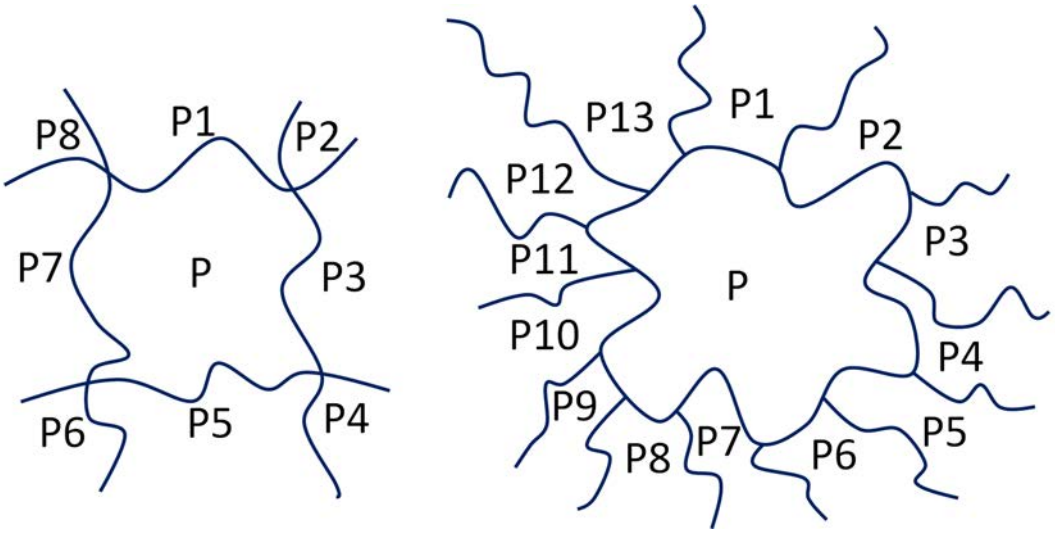

Figure 5.18: Surrounding patches in case of the structured synthesizer and the stochastic synthesizer. *Left:* In the structured synthesizer, the number of surrounding patches is 8. *Right:* In the stochastic synthesizer, the number of surrounding patches is arbitrary. When the number of surrounding patches is high, filtering artifacts may occur under minification.

decoding complexity and storage. Furthermore, using the grid G reduces the decode loop overhead when the rejection rate is high during synthesis.

It should be noted that the starting and the ending of the decoding loop is coherent among neighboring pixels as shown in Figure 5.17. In this figure, neighboring pixels often have the same color (the same number of loop iteration). This coherence is beneficial to the GPU compute pattern and means that the GPU threads are less likely to diverge.

Filtering The same filtering approximation used in the structured synthesizer is used with the stochastic synthesizer. However, this approximation is more prone to artifacts as the patch containing the current texel can be surrounded by an arbitrary number of other patches while in the structured synthesizer a patch is always surrounded by exactly eight patches (Figure 5.18, Left). For the stochastic synthesizer, when the number of surrounding patches is high, filtering artifacts may occur under minification (Figure 5.18, Right).

Performance

We test the encoding and decoding of the stochastic synthesizer for a large texture mapped on a fractal terrain shown in Figure E.7. The exemplar E has a size of 256^2 and is stored in 768 kilobytes of memory. The resulting texture S has a size of 16384^2. It is synthesized and encoded in less than one minute using a patch radius $R = 64$. The encoding requires 18.96 MB of memory excluding the exemplar. Without any encoding, the texture S would have required 768 MB, an order of magnitude more. If the exemplar is repeated,

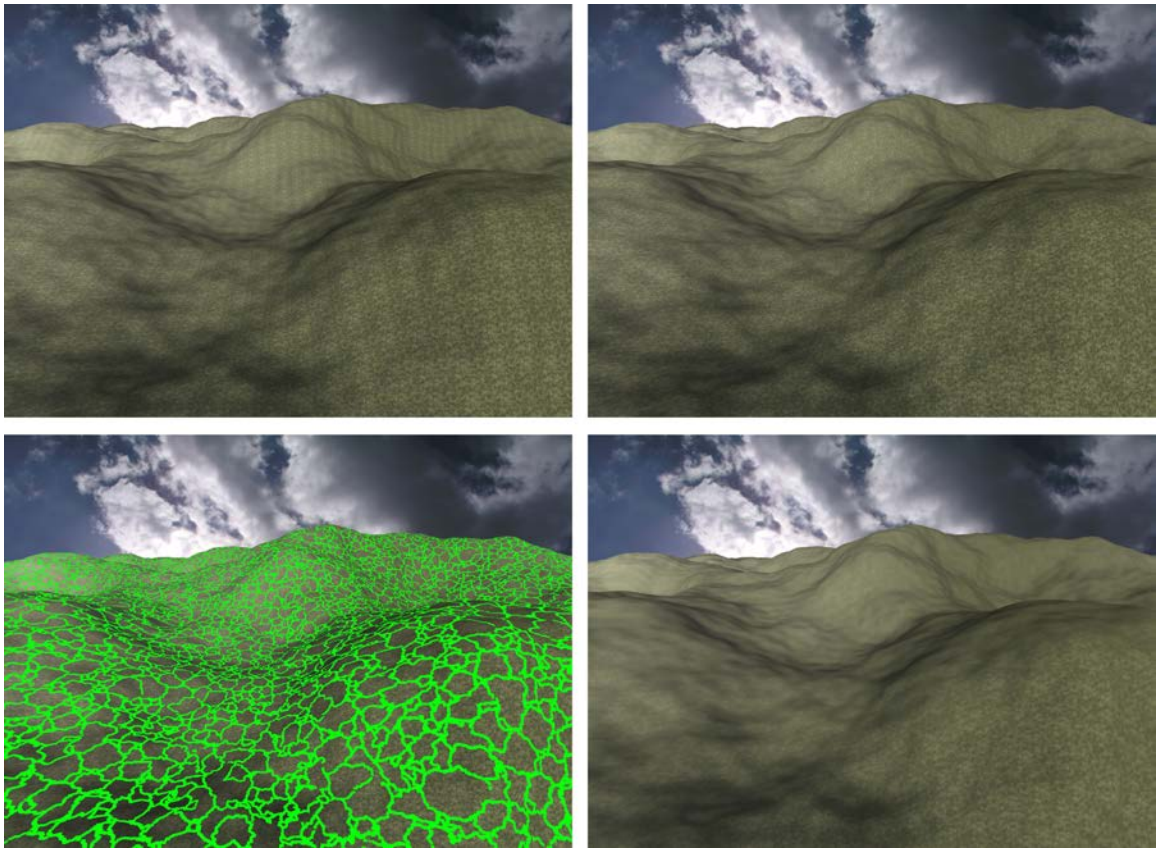

Figure 5.19: *Top left:* The exemplar is repeated on the terrain surface. The repetition produces unpleasing artifacts. *Top right:* Using our stochastic texture synthesizer removes repetition artifacts. *Bottom left:* The light green color shows the boundary of the patches. *Bottom right:* The mipmap level of detail is intentionally biased to show that the filtering approximation is free from noticeable artifacts even at coarse levels of details.

the terrain is rendered at 218 FPS but repetition artifacts occur. Using our decoded synthesized texture, the terrain is rendered without any repetition artifacts at 37 FPS with our unoptimized shader.

Possible improvements

In practice, the stochastic synthesizer requires more memory than the structured synthesizer. Also, its decoding is more expensive than the structured synthesizer. We discuss here possible improvements.

To reduce memory, we can store $\mathcal{T}^{-1}(\mathcal{C})$ instead of $\mathcal{C}$. $\mathcal{T}^{-1}(\mathcal{C})$ is a closed loop and can be decomposed into pieces and every piece can be stored compactly with a Freeman chain code [Fre61]. The number of samples on $\mathcal{C}$ have to be carefully chosen so as to prevent aliasing in $\mathcal{T}^{-1}(\mathcal{C})$.

To improve decoding performance, we have to reduce memory fetches and calls to the transformation function $\mathcal{T}$ that requires a cosine and a sine. This can be done, for instance,

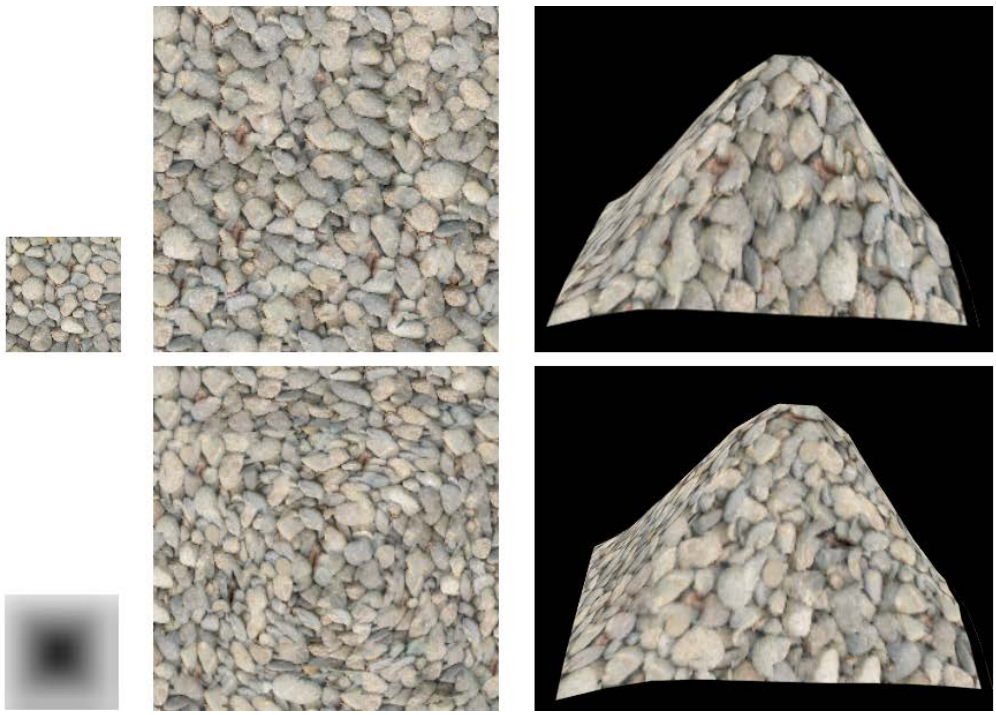

Figure 5.20: *Top:* The middle texture is synthesized from the exemplar on the left. When mapped on a terrain, this texture appears stretched. *Bottom:* The elevation map of the terrain shown on the left is considered during synthesis to produce a texture that when mapped on the terrain preserves the exemplar features. *Source:* Salman and Lefebvre [SL07].

by considering for each patch the largest circle of radius R_{in} that is contained in $\mathcal{T}^{-1}(\mathcal{C})$ and the smallest circle of radius R_{out} that encompasses $\mathcal{T}^{-1}(\mathcal{C})$. Only pixels lying between R_{in} and R_{out} need to be checked against $\mathcal{T}^{-1}(\mathcal{C})$.

As shown in Figure E.7, the synthesizer is particularly useful for texturing terrain. For this case, a possible improvement is to take into consideration distortions resulting from the terrain slope as in Salman and Lefebvre [SL07] (Figure 5.20).

5.2.3 Encoding sparse data

Some texture synthesizers output a sparse content like a sparse set of pebbles, dead leaves or footsteps that can be laid on top of a ground texture. In section 2.2.3, we discussed spatial hashing techniques to efficiently store sparse texture data. A hash table is ideal to compactly store synthesized sparse textures. To allow fast rendering with little memory usage, the hash table must ensure the following:

- The storage of the hash table must be as compact as possible, that is, the table has to store little information in addition to texels and the table should not contain unused space, that is, the load factor that corresponds to the ratio of used space must be high.

- The construction of the hash table must fast. It should be possible to construct the table during synthesis rendering time. Also, the construction must always succeed even at high load factors.
- Reading texels from the hash table has to be as fast as possible, that is, a couple of texture fetches per texel.
- To fully exploit graphics hardware, it is preferable to have a coherent spatial arrangement of the stored data, a coherent access to the memory and a coherent execution of the same code paths.

I contributed with Ismael García, Sylvain Lefebvre and Samuel Hornus in a novel hashing scheme [GLHL11]. This scheme is ideal for sparse texture data processed with the GPU: The stored data, memory accesses and code paths are all coherent. Furthermore, the storage is minimal and a load factor as high as 0.99 can be achieved without construction failure. My contribution was to demonstrate applications using this hashing scheme. We did not try to hash sparse texels generated with a synthesis algorithm. Instead, we focused on hashing texels generated with a 3D painting application. However, the manual painting can be replaced with a synthesis algorithm without any change. For instance, the same hash has been used as a cache to store procedural texture data [RLD+12].

We next give a quick description of the painting application. We refer the reader to García et al. [GLHL11] for details on the hash construction.

Painting application

A 2D atlas is updated interactively while the user paints along the surface. Only the pixels touched by the brush are stored in the hash table. This lets us paint locally at very high resolution, while maintaining a low memory usage.

When the user paints on the model we retrieve the (u, v) coordinates of the pixels touched by the brush. If pixels are already in the hash table, we simply update their colors. If new pixels are touched we rebuild the hash table entirely: We first gather the new pixel key–color pairs and concatenate them with the current hash table from which we remove empty entries. This array is used as the input for building a new hash table. The entire process is fast enough to happen seamlessly while the user paints.

The painting application is illustrated in Figure 5.21. The used atlas has a virtual size of 4096^2, among which $1M$ pixels are painted. For the viewing conditions of the figure, 389586 queries are made in a 1024^2 view–port and the display runs at 237 FPS when the load factor of the hash table is 0.99. When the load factor is reduced to 0.5, the display reaches at 446 FPS. The construction time required ton insert all the keys in the hash table is 10 ms when the load factor is 0.99 and 3 ms when the load factor is 0.5.

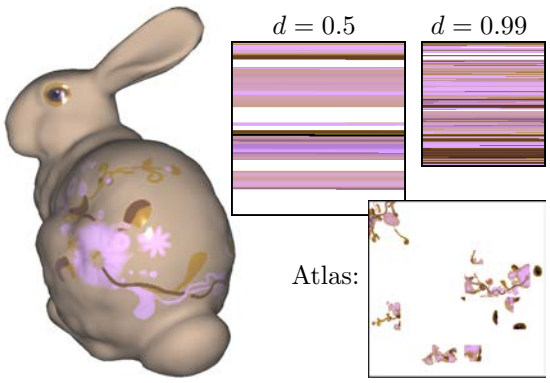

Figure 5.21: Our painting application lets users decorate an object with high resolution details. Only the sparse set of painted texels is stored.

5.3 Streaming synthesized textures

In the previous section, we discussed an encoding scheme that considerably reduces memory usage. However, for very large scenes with rich texture content, the storage required to encode all textures can be significant. For instance, if we increase the number of buildings of Figure 5.3 to 70000, the storage required to encode all the facades would require more than 2GB and this exceeds the video memory limit on most GPUs.

Furthermore, it is possible that the synthesizer has to produce a full bitmap texture, e.g. Substance textures. In this case, pre–synthesizing all textures as raw bitmaps would quickly fill video memory.

In this section, we describe a formulation of the streaming problem: how to *progressively* synthesize the textures during rendering. This way, memory is reduced by unloading any texture that will not be visible in the near future, making room for textures that will soon become visible. We only formalize the general streaming problem in static scenes and target city scenes. We investigated this topic during the internship of Guprit Singh who joined the lab for 6 months. I co–supervised him with my advisor during his stay. Initial ideas and preliminary results explored during Guprit's internship are described in Appendix A.

5.3.1 Assumptions

- We assume that the viewer only navigates on the ground moving at a constant speed and is unable to teleport from one point to another in the scene. This is typical when navigating in large landscapes or cities.

- We assume that the used synthesizers are fast enough to generate textures between rendered frames (within a couple of milliseconds) or at least that the synthesis can be done within a couple of frames in an asynchronous manner.

- As the used synthesizers may not support point–wise evaluation, we do not take advantage of the specific case where point–wise evaluation is possible.
- We assume we are given an algorithm capable of performing visibility queries, so that we can easily obtain the set of textures visible from any point in the scene. The reader may refer to [BMW+09, COCSD03] for visibility computation algorithms.
- For simplicity and without lack of generality, we suppose that all synthesized textures have the same resolution. Note that it is always possible to decompose textures with different resolutions into blocks of same size.

5.3.2 Notations

We note $\mathcal{T} = \{T_1, T_2, ...\}$ the set of all textures in the scene. $|\mathcal{T}|$ is the number of textures in $\mathcal{T}$. We overlay the ground plane with a 2D grid G of size (W, H). For each square cell (x,y) in G, we note $G[(x, y)]$ the set of all visible textures from any point within the region that is comprised in the cell (x,y). G is pre–computed and given as input. We note L a second grid superimposed over G. For each cell (x,y) in L, $L[(x, y)]$ gives the set of all textures residing in memory when the viewer is inside the cell (x,y). Note that L is unknown: Our goal is to pre–compute the sets $L[(x, y)]$ so that during the application we know which textures to load in memory as the user enters a new cell. We note $\mathbf{v}^f$ the viewer cell location in G at frame f. When transitioning from frame f to frame $f+1$ the viewer can move to one of the cells $\mathbf{v}^f + (i, j)$ where $i, j \in \{-1, 0, 1\}^2$. We note $\mathcal{M}_{max}$ the maximum number of textures that can be stored in video memory. We note $\mathcal{B}_{max}$ the maximum number of textures that can be synthesized between two frames. Finally, we note $\mathcal{M}$ the set of textures residing in memory and note $\mathcal{M}^f$ the set of textures residing in memory at frame f. Please note that $\mathcal{M}^f = L[\mathbf{v}^f]$.

5.3.3 Objective

The goal is to pre–determine the set of textures that should reside in memory for each location (x, y) in G. That is, the goal is to find the best assignment of textures in each entry of L so as to always satisfy the following constraints:

$$C1 : \forall\ (x, y),\ G[(x, y)] \subseteq L[(x, y)]$$

$$C2 : \forall\ (x, y),\ L[(x, y)] \leq \mathcal{M}_{max} - size(L)$$

$$C3 : \forall\ i, j \in \{-1, 0, 1\}^2,\ |L[(x + i, y + j)] \setminus L[(x, y)]| \leq \mathcal{B}_{max}$$

$C1$ ensures that visible textures are always residing in memory.

$C2$ ensures that textures that should reside in memory along with the structure L actually fit in the available memory size. $size(L)$ gives the size of the structure L. It is computed as follows:

$$size(L) = \sum_{(x,y)\in G} |L[(x,y)]|$$

$C3$ ensures that when the viewer moves from location $\mathbf{v}^f$ to location $\mathbf{v}^{f+1}$ there will be enough time (bandwidth) to synthesize the textures that have become visible but were not residing in memory. We assume that unloading textures from memory has no cost.

The number of combinations that L can have is extremely large. Even the number of combinations for textures residing in a single cell of L is important. The combinatorial optimization seems therefore intractable. Instead we studied a simple heuristic given in Appendix A. The heuristic tries to directly determine the content of $\mathcal{M}$ at rendering time. by relying on a priority based caching mechanism that takes into consideration the Euclidean and geodesic distances to the texture surfaced along with the texture contribution to the rendered image.

5.4 Conclusion

In this chapter we studied two approaches that overcome the problem of generating before rendering all textures in a raw bitmap format.

The first approach consisted in compactly encoding textures generated with the patch–based synthesis approaches we have used. In addition to the exemplar, the encoding required the storage of few information describing the patches that compose the final textures. The most expensive information describing a patch is its boundary (the cut $\mathcal{C}$). We used factorization and compression to store these boundaries compactly. We also proposed a fast decoding approach supporting random access to the encoded texels and relying on an approximate filtering scheme to ensure high performance rendering. With such encoding and decoding, we were able to render large cities with thousands of different facades in real time and without stressing the memory.

Our compact result represents the synthesized textures with as few as 0.2 bits per pixel (bpp) while the state of art compression scheme S3TC cannot do better than 4 bpp. The best standardized compression rate for 2D RGB textures only reaches 0.89 bpp [NLP$^+$12]. Despite such reduction in memory, our compact representation can still be decoded at interactive rates. In addition, the decoding mostly requires coherent accesses.

The second approach we considered is the synthesis of textures only when they are required for rendering. We described preliminary research on the topic of streaming and at this stage the context of synthesized textures remains a topic of further research.

Chapter 6

Summary and Perspectives

6.1 Summary

In order to create large and highly detailed virtual environments with reduced authoring time and in order to interactively render these environments without overloading video memory, we focused in this thesis on texture synthesis. This automatically generates textures using algorithms instead of storage. The automatic generation not only saves a considerable amount of time to the artists but memory is also dramatically reduced since the synthesis algorithm and its parameters are the only data that need to be stored.

To enable the use of texture synthesis to render large and detailed environments, a number of issues related to texture synthesis have been addressed:

Synthesized texture visualization and exploration As the visualization and parameter selection of synthesized textures are difficult, we proposed to summarize the possible synthesizable appearances in two types of previews:

We call the first a *procedural texture preview*. It summarizes the synthesizable appearances in a static image with a limited pixel space. The static image preview ensures that most appearances are visible, are allotted a similar pixel area, and are ordered in a smooth manner throughout the preview. Such a preview is useful for quickly interpreting the variety of appearances produced by a procedure and this especially in the case where a user is exploring a database of procedural textures.

The second preview is an image strip associated with each parameter of the synthesized texture. This image strip is dynamically refreshed to reflect changes with respect to the current setting of the parameters. The preview is built in a way that reveals in a limited pixel space the changes that will be introduced upon manipulation. The slider previews represent a simple adaptation of the procedural texture preview to a single parameter. However despite their simplicity, the previews are intuitive and may enhance the experience of selecting procedural texture parameters. The main limitation is that the interactivity may be limited by the complexity of the procedural algorithms.

In order to build the previews interactively, we proposed a fast patch–based synthesizer that allows combining the different synthesizable appearances in a continuous preview. The quality produced by this synthesizer is as high as the quality of the best existing patch–based synthesizers. Furthermore, the synthesizer achieves interactive synthesis, thanks to the parallel used algorithms that are entirely implemented on the GPU.

Compact representation of patch–based texture synthesis We have discussed the fact that a texture synthesis algorithm may not be point–wise evaluated. This is often the case of data–driven texture synthesizers that have to compute a whole texture then store it in memory as a bitmap.

For the patch–based synthesis algorithms we have considered, we avoid the storage of the result as a bitmap by encoding it in a compact representation. We then allow to rapidly decoding texels from this representation during rendering. The decoding affords for a random access to the texels and offers the possibility to use an approximate filtering scheme that relies on the hardware for maximum performance.

Rather than generating a bitmap texture then use compression to reduce its size, the developed synthesis algorithms were designed to directly produce a compact representation. We have only considered two patch–based synthesizes but it would be interesting to investigate other synthesis techniques and try to analyze their operations to possibly find a compact way of storing the result.

Streaming of synthesized textures Our compact representation cannot be directly generalized to any texture synthesis method. Without the support for compact encoding, there is no other choice but to synthesize the textures prior to rendering. This can be costly in memory especially if many textures have to be synthesized. In this case, we have discussed the importance of streaming synthesized textures. The streaming aims to synthesize only the textures that will be visible in the near future making the best use of the available memory and bandwidth. We have described a general formulation of the problem. A simple heuristic is described in appendix A.

In today's game engines, streaming a high amount of textures has become critical to render realistic scenes. However, the design of a robust streaming system would require complex algorithms that have to be general enough to handle any type of virtual scenes including dynamic ones. This makes texture streaming an open object of research with strong industrial constraints for practical solutions.

6.2 Perspectives

Procedural techniques seek to answer two important limitations faced by most interactive applications: The first limitation relates to storage and bandwidth and the second relates

to authoring time. Procedural techniques therefore represent a valuable tool for artists to efficiently create complex and detailed content. As future work, we envision several possible paths to further promote the use of procedural techniques.

Assisted texture authoring We discussed the fact that the authoring and the tweaking of procedural textures can be complex. We presented methods to ease their exploration but we did not investigate how to author them more efficiently.

During the past years, our industrial partner Allegorithmic has developed a large library of substance textures. To ease the creation of such a library, artists have access to a large set of low level generic generators like a brick or a grass generators. However, making complex materials like ceramic or concrete requires a certain effort. The effort is more considerable when creating complex structures like doors or windows.

As future work, we would like to investigate ways to analyze the large library of substance graphs and extract sub–graphs representing semantically meaningful materials and objects. For a semantic description of substances, we can take advantage of the research done in the vision community which led to the development of systems [SWRC06] capable of extracting semantically meaningful objects from visual data. These systems are particularly efficient when using the Internet as a database for supervised learning tasks [HE07, DDS+09, DBLFF10]. Using a semantic description of substances, we can automatically create a library of generic procedural generators. Because the number of generic generators can be high, we can use the procedural texture preview to visually present the possibilities to the artist. The artist can simply drag an appearance and drop it to the graph that is under construction.

Furthermore, to better assist artists, we can use the semantic description of the substances to suggest what type of operations should the artist use to grow the graph that is under construction. Of course, this would require an interactive analysis of the graph that is under construction.

Automatic texturing Throughout this thesis, we emphasized the importance of by–example synthesis techniques. Such techniques do not require a complex manual setting to define the procedure parameters. This is ideal as it is natural for artists to start from visual examples that inspire the authoring of the final textures. For instance, in film and video pre–production, artists start by defining the look and feel of the scene through drawings, sketches, photographs, pastels, animatics, sculptures or storyboards. The examples are only inspirational and artists have to develop the textures from scratch during production.

Automatic techniques for generating a graph–based texture from an example exist. For instance, genetic texturing [Sim91] aims at automatically creating a procedural texture that matches an input exemplar.

In addition to the previously discussed perspective, another research perspective is to automatize the creation of substances from an input set of exemplars. As for the substances, the exemplars can be analyzed to extract semantically meaningful objects, e.g. building, doors, trees, stones *etc.* The description of the extracted objects should be enhanced by finding similar objects using on–line information retrieval systems [BYRN+99, HK06]. Similar objects and their relations can be used to define a procedural parametrization where parameters allow transitioning from one object to another.

The decomposition of the input could be done recursively to obtain a tree structure where the leaves can be easily described with substances and the other nodes can be easily built by combining child nodes using substance operations. For example, a brick wall texture can be decomposed into three simpler objects: The brick structure, a stochastic texture representing the mortar and another stochastic texture representing the brick material.

An example of an application using such a system is the approximation of real world environments using data from databases like Google earth, Google street maps or Microsoft Bing maps. The approximation would be entirely procedural, meaning that the data that have to be stored and transferred are extremely compact. Additionally, content of higher resolution than the original can be obtained thanks to the procedural description.

Exploration for other procedural techniques In this thesis we mainly focused on texturing. Procedural techniques however generalize to any graphic representation ranging from geometric models to particle systems. Procedural techniques are heavily used by artists to model a diversity of content [EMP+02, DEF+04] varying from large environment like cities [Mul06, Whi06, WAMV11], forests [PL91, PHL+09], crowds [ABL+04, TL09], road networks [GPMG10, GPGB11, WAMV11] or terrains [Dac06, PGGM09a, PGGM09b, SDKT+09] to details and particles like fur [Gol97, PH06], foliage [AP03, PTMG08, BP08, TWT+06], feather [SAM08, LF10, BKA11], fire [FKM+07, HG09, AP11] or clouds [Har03, Bou07]. As future research, we want to investigate an adaptation of the procedural texture preview to any sort of procedural content. However, the diversity and versatility of procedural techniques make an intuitive general previewing system challenging. Furthermore, the dimensionality of the output can vary from one type of procedure to another, for example, the result can be a three dimensional surface or a volume and may vary through time. Volumetric and dynamic visualization techniques coupled with dimensionality reduction and data summarization are possible directions for future research. We can for instance draw inspiration and adapt the geometric data–driven interface of Chaudhuri and Koltun [CK10], the fit and diverse method of Xu et al. [XZCOC12], the animation interface of Brochu et al. [BBdF10] or the light transport inspection and visualization tools described by Reiner et al. [RKRD12].

Bibliography

[ABL+04] M. Aitken, G. Butler, D. Lemmon, E. Saindon, D. Peters, and G. Williams. The lord of the rings: the visual effects that brought middle earth to the screen. In *SIGGRAPH 2004 Course Notes*, 2004.

[ADA+04] Aseem Agarwala, Mira Dontcheva, Maneesh Agrawala, Steven Drucker, Alex Colburn, Brian Curless, David Salesin, and Michael Cohen. Interactive digital photomontage. 2004.

[Alc12] D.A.F. Alcantara. *Efficient Hash Tables on the GPU*. PhD thesis, University of California, Davis, 2012.

[AP03] M. Aitken and M. Preston. Grove: a production-optimised foliage generator for the lord of the rings: The two towers. In *GRAPHITE*, 2003.

[AP11] Dhanyu Amarasinghe and Ian Parberry. Towards fast, believable real-time rendering of burning objects in video games. In *Foundations of Digital Games*, 2011.

[ARB] ARB_texture_compression_bptc.

[AS02] J.C. Hart. A. Sheffer. Seamster: Inconspicuous low-distortion texture seam layout. In *IEEE Visualization 2002 Conference Proceedings*, 2002.

[AS07] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. *Transactions on Graphics*, 2007.

[ASA+09] Dan A. Alcantara, Andri Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the gpu. *Transactions on Graphics*, 2009.

[Ash01] Michael Ashikhmin. Synthesizing natural textures. In *I3D*, 2001.

[BA83] P. Burt and E. Adelson. The laplacian pyramid as a compact image code. *Communications, IEEE Transactions on*, 31(4):532–540, 1983.

[BAC96] Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha. Rendering from compressed textures. In *SIGGRAPH*, 1996.

[Bar08] S. Barrett. Sparse virtual textures. *GDC 2008 presentation*, 2008.

[BB81] H. Harlyn Baker and Thomas O. Binford. Depth from edge and intensity based stereo. In *IJCAI*, 1981.

[BBdF10] E. Brochu, T. Brochu, and N. de Freitas. A bayesian interactive optimization approach to procedural animation design. In *SCA*, 2010.

[BD02] David Benson and Joel Davis. Octree textures. In *SIGGRAPH*, 2002.

[BD04] E. Bourque and G. Dudek. Procedural texture matching and transformation. In *Computer Graphics Forum*, 2004.

[BDA04] E. Galin B. Desbenoit and S. Akkouche. Simulating and modeling lichen growth. In *Computer Graphics Forum*, 2004.

[BDL+08] A. Bharambe, J.R. Douceur, J.R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *ACM SIGCOMM*, 2008.

[Bel54] R. Bellman. The theory of dynamic programming. *Bull. Amer. Math. Soc*, 1954.

[BGSF10] Connelly Barnes, Dan B Goldman, Eli Shechtman, and Adam Finkelstein. Video tapestries with continuous temporal zoom. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 2010.

[BKA11] S.D. Bowline and Z. Kačić-Alesić. Dynamic, penetration-free feathers in rango. In *ACM SIGGRAPH Talks*, 2011.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977.

[BMW+09] J. Bittner, O. Mattausch, P. Wonka, V. Havran, and M. Wimmer. Adaptive global visibility sampling. *ACM Transactions on Graphics (TOG)*, 2009.

[Bou07] A. Bouthors. *Realistic rendering of clouds in real-time*. PhD thesis, Université Joseph Fourier, 2007.

[BP08] Jeffrey Budsberg and Scott Peterson. Beyond procedurally modeled foliage in madagascar: Escape 2 africa. In *ACM SIGGRAPH talks*, 2008.

[BPMG04] C. Bosch, X. Pueyo, S. Mérillou, and D. Ghazanfarpour. A physically-based model for rendering realistic scratches. In *Computer Graphics Forum*, 2004.

[BSFG09] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. PatchMatch: A randomized correspondence algorithm for structural image editing. *Transactions on Graphics*, 2009.

[BVZ99] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. In *ICCV*, 1999.

[BYRN+99] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*. ACM press., 1999.

[CAF10] T.S. Cho, S. Avidan, and W.T. Freeman. A probabilistic image jigsaw puzzle solver. 2010.

[Can05] I. Cantlay. Mipmap-level measurement. *GPU Gems II*, 2005.

[Cat74] Edwin E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.d. thesis, University of Utah, 1974.

[CB04] P.H. Christensen and D. Batali. An irradiance atlas for global illumination in complex production scenes. In *EGSR*, 2004.

[CBAF08] Taeg Sang Cho, Moshe Butman, Shai Avidan, and William T. Freeman. The patch transform and its applications to image editing. In *CVPR*, 2008.

[CCL12] Y. Cao, A. B. Chan, and R. Lau. Automatic stylistic manga layout. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)*, 2012.

[CD05] R.L. Cook and T. DeRose. Wavelet noise. In *ACM Transactions on Graphics*, 2005.

[CDF+86] G. Campbell, T.A. DeFanti, J. Frederiksen, S.A. Joyce, and L.A. Leske. Two bit/pixel full color encoding. In *ACM SIGGRAPH*, 1986.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 2008.

[CESL10] Mattäus G. Chajdas, Christian Eisenacher, Marc Stamminger, and Sylvain Lefebvre. Gpu pro. chapter Virtual Texture Mapping 101. 2010.

[CEVZ05] G. Chartrand, D. Erwin, D. VanderJagt, and P. Zhang. γ-labelings of graphs. *Bull. Inst. Combin Appl*, 2005.

[CGL+98] J.H.P. Chim, M. Green, R.W.H. Lau, H. Va Leong, and A. Si. On caching and prefetching of virtual objects in distributed virtual environments. In *ACM international conference on Multimedia*, 1998.

[CK10] S. Chaudhuri and V. Koltun. Data-driven suggestions for creativity support in 3d modeling. 2010.

[CKGK11] Siddhartha Chaudhuri, Evangelos Kalogerakis, Leonidas Guibas, and Vladlen Koltun. Probabilistic reasoning for assembly-based 3D modeling. *Transactions on Graphics*, 2011.

[CNLE09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *I3D*, 2009.

[COCSD03] D. Cohen-Or, Y.L. Chrysanthou, C.T. Silva, and F. Durand. A survey of visibility for walkthrough applications. *IEEE Visualization and Computer Graphics*, 2003.

[CPAM08] T. Capin, K. Pulli, and T. Akenine-Moller. The state of the art in mobile graphics research. *IEEE Computer Graphics and Applications*, 2008.

[Cro84] Franklin C. Crow. Summed-area tables for texture mapping. In *Computer Graphics*, 1984.

[CSHD03] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. *Transactions on Graphics*, 2003.

[CT82] R.L. Cook and K.E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics (TOG)*, 1(1):7–24, 1982.

[CXW+05] Y. Chen, L. Xia, T.T. Wong, X. Tong, H. Bao, B. Guo, and H.Y. Shum. Visual simulation of weathering by γ-ton tracing. *Transactions on Graphics*, 2005.

[CXY05] X. Cui, L. Xiuwen, and J. Yicheng. 3d scene transmission for web-based shiphandling training. In *IEEE, Computer Graphics, Imaging and Vision: New Trends*, 2005.

[Dac06] C. Dachsbacher. *Interactive terrain rendering: Towards realism with procedural models and graphics hardware*. Institut für Informatik, 2006.

[DBLFF10] J. Deng, A. Berg, K. Li, and L. Fei-Fei. What does classifying more than 10,000 image categories tell us? *Computer Vision–ECCV 2010*, 2010.

[DDS+09] J. Deng, W. Dong, R. Socher, L.J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[DEF+04] Oliver Deusen, David S. Ebert, Ron Fedkiw, F. Kenton Musgrave, Przemyslaw Prusinkiewicz, Doug Roble, Jos Stam, and Jerry Tessendorf. The elements of nature: interactive and realistic techniques. In *SIGGRAPH Course Notes*, 2004.

[DFG99] Qiang Du, Vance Faber, and Max Gunzburger. Centroidal voronoi tessellations: Applications and algorithms. *SIAM*, 1999.

[DGPR02] David DeBry, Jonathan Gibbs, Devorah DeLeon Petty, and Nate Robins. Painting and rendering textures on unparameterized models. In *SIGGRAPH*, 2002.

[DH96] J. Dorsey and P. Hanrahan. Modeling and rendering of metallic patinas. *Transactions on Graphics*, 1996.

[DJ05] Craig Donner and Henrik Wann Jensen. Light diffusion in multi-layered translucent materials. *Transactions on Graphics*, 2005.

[DJLW08] R. Datta, D. Joshi, J. Li, and J.Z. Wang. Image retrieval: Ideas, influences, and trends of the new age. *ACM Computing Surveys (CSUR)*, 2008.

[DLTD08] Yue Dong, Sylvain Lefebvre, Xin Tong, and George Drettakis. Lazy solid texture synthesis. In *EGSR*, 2008.

[DM79] E. Delp and O. Mitchell. Image compression using block truncation coding. *IEEE Communications*, 1979.

[DMLG02] J.M. Dischler, K. Maritaud, B. Lévy, and D. Ghazanfarpour. Texture particles. *Eurographics Conference*, 2002.

[DPH96] Julie Dorsey, Hans Køhling Pedersen, and Pat Hanrahan. Flow and changes in appearance. In *SIGGRAPH*, 1996.

[DSC03] Laurent Demanet, Bing Song, and Tony Chan. Image inpainting by correspondence maps: a deterministic approach. Technical report, 2003. UCLA.

[EF01] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. In *SIGGRAPH*, 2001.

[Eic94] S.G. Eick. Data visualization sliders. In *Proceedings of the 7th annual ACM symposium on User interface software and technology*, 1994.

[EL99] Alexei A. Efros and Thomas K. Leung. Texture synthesis by non-parametric sampling. In *ICCV*, 1999.

[EMP+02] D.S. Ebert, F.K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and modeling: a procedural approach*. Morgan Kaufmann, 2002.

[ETB+10] C. Eisenacher, C. Tappan, B. Burley, D. Teece, and A. Shek. Example-based texture synthesis on disney's tangled. In *SIGGRAPH Talks*, 2010.

[Fen03] S. Fenney. Texture compression using low-frequency signal modulation. In *Graphics hardware*, 2003.

[FKM+07] Alfred R. Fuller, Hari Krishnan, Karim Mahrous, Bernd Hamann, and Kenneth I. Joy. Real-time procedural volumetric fire. In *I3D*, 2007.

[Fre61] H. Freeman. On the encoding of arbitrary geometric configurations. *IRE Electronic Computers*, 1961.

[Gab46] D. Gabor. Theory of communication. *J. Int. Electrical Engineers*, 1946.

[Gam06] Epic Games. Unreal engine 3. *URL: http://www.unrealengine.com*, 2006.

[Gar81] D.D. Garber. Computational models for texture analysis and texture synthesis. 1981.

[GBK06] M. Guthe, P. Borodin, and R. Klein. Real-time out-of-core rendering. *International Journal of Image and Graphics*, 2006.

[GD95] Djamchid Ghazanfarpour and Jean-Michel Dischler. Spectral analysis for automatic 3d texture generation. *Computers & Graphics*, 1995.

[GD96] D. Ghazanfarpour and J.M. DISCHLER. Generation of 3d texture using multiple 2d models analysis. In *Computer Graphics Forum*, volume 15, pages 311–323. Wiley Online Library, 1996.

[GdM85] Andre Gagalowicz and Song de Ma. Model driven synthesis of natural textures for 3-D scenes. In *Eurographics Conference*, 1985.

[GDS10] G. Gilet, J.M. Dischler, and L. Soler. Procedural descriptions of anisotropic noisy textures by example. In *Eurographics Short Papers*, 2010.

[GLHL11] Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. Coherent parallel hashing. *ACM Trans. Graph.*, 2011.

[GLLD12] B. Galerne, A. Lagae, S. Lefebvre, and G. Drettakis. Gabor noise by example. *Transactions on Graphics*, 2012.

[Gol97] D.B. Goldman. Fake fur rendering. In *SIGGRAPH*, 1997.

[Gon85] T.F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 1985.

[GPGB11] E. Galin, A. Peytavie, E. Guérin, and B. Beneš. Authoring hierarchical road networks. In *Computer Graphics Forum*, 2011.

[GPMG10] E. Galin, A. Peytavie, N. Maréchal, and E. Guérin. Procedural generation of roads. In *Computer Graphics Forum*, 2010.

[GSX] Baining Guoa, Harry Shum, and Ying-Qing Xu. Chaos mosaic: Fast and memory efficient texture synthesis. Microsoft Research technical report MSR-TR-2000-32.

[GT88] A.V. Goldberg and R.E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 1988.

[GZD08] A. Goldberg, M. Zwicker, and F. Durand. Anisotropic noise. In *ACM Transactions on Graphics*, 2008.

[Har03] M.J. Harris. *Real-time cloud simulation and rendering.* PhD thesis, University of North Carolina, 2003.

[HB95] David J. Heeger and James R. Bergen. Pyramid-Based texture analysis/synthesis. In Robert Cook, editor, *SIGGRAPH*, 1995.

[HE07] J. Hays and A.A. Efros. Scene completion using millions of photographs. In *ACM Transactions on Graphics*, 2007.

[Hec89] Paul S. Heckbert. Fundamentals of texture mapping and image warping. Technical report, 1989.

[Hes99] Tom Heskes. Energy functions for self-organizing maps. In E. Oja and S. Kaski, editors, *Kohonen Maps*. 1999.

[HG09] C. Horvath and W. Geiger. Directable, high-resolution simulation of fire on the gpu. In *ACM Transactions on Graphics*, 2009.

[HH99] C. Heaps and S. Handel. Similarity and features of natural textures. *Journal of Experimental Psychology: Human Perception and Performance*, 1999.

[HK06] J. Han and M. Kamber. *Data mining: concepts and techniques*. Morgan Kaufmann, 2006.

[HLS+07] K. Hormann, B. Lévy, A. Sheffer, et al. Mesh parameterization: Theory and practice. *SIGGRAPH Course Notes*, 2007.

[Hop96] Hugues Hoppe. Progressive meshes. In *SIGGRAPH*, 1996.

[HPLVdW09] C. Hollemeersch, B. Pieters, P. Lambert, and R. Van de Walle. Accelerating virtual texturing using cuda. *GPU Pro*, 2009.

[HR00] R. Hassin and S. Rubinstein. Better approximations for max tsp. *Information Processing Letters*, 2000.

[HRRG08] Charles Han, Eric Risser, Ravi Ramamoorthi, and Eitan Grinspun. Multiscale texture synthesis. *Transactions on Graphics*, 2008.

[HS85] D.S. Hochbaum and D.B. Shmoys. A best possible heuristic for the k-center problem. *Mathematics of operations research*, 1985.

[HS86] Dorit S. Hochbaum and David B. Shmoys. A unified approach to approximation algorithms for bottleneck problems. *J. ACM*, 1986.

[Huf52] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 1952.

[Hüt98] T. Hüttner. High resolution textures. *IEEE Visualization*, 1998.

[Ibr98] Alaa Eldin M. Ibrahim. *Genshade: an evolutionary approach to automatic and interactive procedural texture generation*. PhD thesis, 1998.

[IM06] T. Inada and M.D. McCool. Compressed lossless texture representation and caching. In *Graphics Hardware*, 2006.

[INH99] K.I. Iourcha, K.S. Nayak, and Z. Hong. System and method for fixed-rate block-based image compression with inferred pixel values, 1999. US Patent 5,956,431.

[JDR04] Robert Jagnow, Julie Dorsey, and Holly Rushmeier. Stereological techniques for solid textures. *SIGGRAPH*, 2004.

[Jen96] R. Jenkins. Isaac. In *Fast Software Encryption*, 1996.

[Jen06] R. Jenkins. The hash, 2006.

[JFK03] N. Jojic, B.J. Frey, and A. Kannan. Epitomic analysis of appearance and shape. In *ICCV*, 2003.

[JGDAM12] B. Johnsson, P. Ganestam, M. Doggett, and T. Akenine-Möller. Power efficiency for software algorithms running on graphics processors. In *High Performance Graphics*, 2012.

[JSTS06] Jiaya Jia, Jian Sun, Chi-Keung Tang, and Heung-Yeung Shum. Drag-and-drop pasting. *Transactions on Graphics*, 2006.

[JT05] J. Jia and C.K. Tang. Eliminating structure and intensity misalignment in image stitching. In *ICCV*, 2005.

[JT08] J. Jia and C.K. Tang. Image stitching using structure deformation. *PAMI*, 2008.

[KCODL06] J. Kopf, D. Cohen-Or, O. Deussen, and D. Lischinski. *Recursive Wang tiles for real-time blue noise.* 2006.

[KEBK05] Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. Texture optimization for example-based synthesis. *Transactions on Graphics*, 2005.

[KKL+07] S.J. Kim, K. Koh, M. Lustig, S. Boyd, and D. Gorinevsky. An interior-point method for large-scale l1-regularized least squares. *Selected Topics in Signal Processing, IEEE Journal of*, 2007.

[KP09] William B. Kerr and Fabio Pellacini. Toward evaluating lighting design interface paradigms for novice users. *Transactions on Graphics*, 2009.

[KP10] William B. Kerr and Fabio Pellacini. Toward evaluating material design interface paradigms for novice users. *Transactions on Graphics*, 2010.

[Kru64] J.B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 1964.

[KSE+03] Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut textures: Image and video synthesis using graph cuts. *Transactions on Graphics*, 2003.

[KSH01] T. Kohonen, M. R. Schroeder, and T. S. Huang, editors. *Self-Organizing Maps.* Springer-Verlag New York, Inc., 3rd edition, 2001.

[LD11] A. Lagae and G. Drettakis. Filtering solid gabor noise. *ACM Transactions on Graphics*, 2011.

[LDN04] Sylvain Lefebvre, Jérome Darbon, and Fabrice Neyret. Unified texture management on arbitrary meshes. Technical Report 5210, INRIA, 2004.

[Lew89] John-Peter Lewis. Algorithms for Solid Noise Synthesis. In *Computer Graphics*, 1989.

[LF10] James Leaning and Damien Fagnou. Feathers for mystical creatures: creating pegasus for clash of the titans. In *SIGGRAPH Talks*, 2010.

[LH04] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *Transactions on Graphics*, 2004.

[LH05] Sylvain Lefebvre and Hugues Hoppe. Parallel controllable texture synthesis. *SIGGRAPH*, 2005.

[LH06a] Sylvain Lefebvre and Hugues Hoppe. Appearance-space texture synthesis. *Transactions on Graphics*, 2006.

[LH06b] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. *SIGGRAPH*, 2006.

[LHL10] Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. By-example synthesis of architectural textures. *Transactions on Graphics*, 2010.

[LHN05] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. *Octree Textures on the GPU*. GPU Gems II, 2005.

[LL12] A. Lasram and S. Lefebvre. Parallel patch-based texture synthesis. In *High Performance Graphics*, 2012.

[LLC+10] Ares Lagae, Sylvain Lefebvre, Rob Cook, Tony DeRose, George Drettakis, D.S. Ebert, J.P. Lewis, Ken Perlin, and Matthias Zwicker. A survey of procedural noise functions. *Computer Graphics Forum*, 2010.

[LLD12a] Anass Lasram, Sylvain Lefebvre, and Cyrille Damez. Procedural texture preview. *Computer Graphics Forum (Eurographics conf. proc.)*, 2012.

[LLD12b] Anass Lasram, Sylvain Lefebvre, and Cyrille Damez. Scented sliders for procedural textures. *Eurographics short paper*, 2012.

[LLDD09] Ares Lagae, Sylvain Lefebvre, George Drettakis, and Philip Dutré. Procedural noise using sparse gabor convolution. *Transactions on Graphics*, 2009.

[Llo82] S. Lloyd. Least squares quantization in pcm. *IEEE Information Theory*, 1982.

[LLP12] PricewaterhouseCoopers LLP. The evolution of video gaming and content consumption. 2012.

[LLX+01] Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. Real-time texture synthesis by patch-based sampling. *SIGGRAPH*, 2001.

[LN02] Sylvain Lefebvre and Fabrice Neyret. Synthesizing bark. In *Eurographics Workshop on Rendering*, 2002.

[LN03] Sylvain Lefebvre and Fabrice Neyret. Pattern based procedural textures. In *I3D*, 2003.

[LP00] Laurent Lefebvre and Pierre Poulin. Analysis and synthesis of structural textures. In *Graphics Interface*, 2000.

[LPRM02] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Least squares conformal maps for automatic texture atlas generation. *Transactions on Graphics*, 2002.

[LVLD10] A. Lagae, P. Vangorp, T. Lenaerts, and P. Dutré. Procedural isotropic stochastic textures by example. *Computers & Graphics*, 2010.

[LWL+09] Y. Liu, W. Wang, B. Lévy, F. Sun, D.M. Yan, L. Lu, and C. Yang. On centroidal voronoi tessellation–energy smoothness and fast computation. *ACM Transactions on Graphics*, 2009.

[LZPW04] A. Levin, A. Zomet, S. Peleg, and Y. Weiss. Seamless image stitching in the gradient domain. *ECCV*, 2004.

[MAB+97] J. Marks, B. Andalman, P. A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. Design galleries: a general approach to setting parameters for computer graphics and animation. In *SIGGRAPH*, 1997.

[MDG01] Stephane Merillou, Jean-Michel Dischler, and Djamchid Ghazanfarpour. Corrosion: Simulating and rendering. In *Graphics Interface*, 2001.

[MMS+05] G. Müller, J. Meseth, M. Sattler, R. Sarlette, and R. Klein. Acquisition, synthesis, and rendering of bidirectional texture functions. In *Computer Graphics Forum*, 2005.

[Mul06] Pascal Muller. Procedural modeling of cities. In *ACM SIGGRAPH Courses*, 2006.

[MWT11] Chongyang Ma, Li-Yi Wei, and Xin Tong. Discrete element textures. *ACM Trans. Graph.*, 2011.

[MYV93] Jérôme Maillot, Hussein Yahia, and Anne Verroust. Interactive texture mapping. In *SIGGRAPH*, 1993.

[MZD05] Wojciech Matusik, Matthias Zwicker, and Frédo Durand. Texture design using a simplicial complex of morphable textures. *Transactions on Graphics*, 2005.

[NDM06] A. Ngan, F. Durand, and W. Matusik. Image-driven navigation of analytical brdf models. *Eurographics Symposium on Rendering*, 2006.

[NLP+12] J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson. Adaptive scalable texture compression. In *High Performance Graphics*, 2012.

[Nyq24] H. Nyquist. Certain factors affecting telegraph speed. *American Institute of Electrical Engineers*, 1924.

[OHH+02] M. Olano, JC Hart, W. Heidrich, B. Mark, and K. Perlin. Real-time shading languages. *SIGGRAPH Course Notes*, 2002.

[Ost07] V. Ostromoukhov. Sampling with polyominoes. In *Transactions on Graphics*, 2007.

[OvWS12] Juraj Obert, J. M. P. van Waveren, and Graham Sellers. Virtual texturing in software and hardware. In *SIGGRAPH Courses*, 2012.

[Pea88] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[Pea90] D. Peachey. Texture on demand. 1990.

[PELS10] Pau Panareda Busto, Christian Eisenacher, Sylvain Lefebvre, and Marc Stamminger. Instant Texture Synthesis by Numbers. *Vision, Modeling & Visualization*, 2010.

[Per85] Ken Perlin. An image synthesizer. In *SIGGRAPH*, 1985.

[Per02] Ken Perlin. Improving noise. In *SIGGRAPH*, 2002.

[PFH00] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In *SIGGRAPH*, 2000.

[PGB03] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. 2003.

[PGGM09a] A. Peytavie, E. Galin, J. Grosjean, and S. Merillou. Arches: a framework for modeling complex terrains. In *Computer Graphics Forum*, 2009.

[PGGM09b] A. Peytavie, E. Galin, J. Grosjean, and S. Merillou. Procedural generation of rock piles using aperiodic tiling. In *Computer Graphics Forum*, 2009.

[PH06] M. Preston and M. Hill. Grooming, animating & rendering fur for king kong. In *SIGGRAPH Sketches*, 2006.

[PHL+09] Wojciech Palubicki, Kipp Horel, Steven Longay, Adam Runions, Brendan Lane, Radomír Měch, and Przemyslaw Prusinkiewicz. Self-organizing tree models for image synthesis. *ACM Trans. Graph.*, 2009.

[Pho75] B.T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 1975.

[PL91] P. Prusinkiewicz and A. Lindenmayer. The algorithmic beauty of plants (the virtual laboratory). 1991.

[PL02] G. Popescu and Z. Liu. On scheduling 3d model transmission in network virtual environments. In *Distributed Simulation and Real-Time Applications*, 2002.

[PL07] F. Pellacini and J. Lawrence. Appwand: editing measured materials using appearance-driven optimization. In *Transactions on Graphics*, 2007.

[PP93] K. Popat and R.W. Picard. *Novel cluster-based probability model for texture synthesis, classification, and compression.* MIT. Vision and Modeling Group, 1993.

[PPD02] Eric Paquette, Pierre Poulin, and George Drettakis. The simulation of paint cracking and peeling. In *Graphics Interface*, 2002.

[PS99] J. Portilla and E. Simoncelli. Texture modeling and synthesis using joint statistics of complex wavelet coefficients. In *IEEE Workshop on Statistical and Computational Theories of Vision*, 1999.

[PTG02] F. Pellacini, P. Tole, and D.P. Greenberg. A user interface for interactive cinematic shadow design. In *Transactions on Graphics*, 2002.

[PTMG08] A. Peyrat, O. Terraz, S. Merillou, and E. Galin. Generating vast varieties of realistic leaves with parametric 2gmap l-systems. *The Visual Computer*, 2008.

[PW02] Lothar Pantel and Lars C. Wolf. On the suitability of dead reckoning schemes for games. In *Workshop on Network and system support for games*, 2002.

[PZ10] J. Pribyl and P. Zemcik. Multi-resolution next location prediction for distributed virtual environments. In *Embedded and Ubiquitous Computing*, 2010.

[RK95] W. Richards and J. J. Koenderink. Trajectory mapping (tm): A new nonmetric scaling technique. *Perception*, 1995.

[RKRD12] T. Reiner, A. Kaplanyan, M. Reinhard, and C. Dachsbacher. Selective inspection and interactive visualization of light transport in virtual scenes. In *Computer Graphics Forum*, 2012.

[RLD+12] Tim Reiner, Sylvain Lefebvre, Lorenz Diener, Ismael Garcia, Bruno Jobard, and Carsten Dachsbacher. A runtime cache for interactive procedural modeling. *Computers and Graphics, SMI*, 2012.

[RRT91] S. Ravi, D. Rosenkrantz, and G. Tayi. Facility dispersion problems: Heuristics and special cases. *Algorithms and Data Structures*, 1991.

[SA79] B. Schachter and N. Ahuja. Random pattern generation processes. *Computer Graphics and Image Processing*, 1979.

[SAM05] J. Ström and T. Akenine-Möller. i packman: high-quality, low-complexity texture compression for mobile phones. In *Graphics hardware*, 2005.

[SAM08] Daniel Seddon, Martin Auflinger, and David Mellor. Rendertime procedural feathers through blended guide meshes. In *SIGGRAPH talks*, 2008.

[SCSI08] D. Simakov, Y. Caspi, E. Shechtman, and M. Irani. Summarizing visual data using bidirectional similarity. In *CVPR*, 2008.

[SDKT+09] R.M. Smelik, K.J. De Kraker, T. Tutenel, R. Bidarra, and S.A. Groenewegen. A survey of procedural methods for terrain modelling. In *Workshop on 3D Advanced Media In Gaming And Simulation*, 2009.

[SG96] D. Schmalstieg and M. Gervautz. Demand-driven geometry transmission for distributed virtual environments. In *Computer Graphics Forum*, 1996.

[Sim91] K. Sims. Artificial evolution for computer graphics. *SIGGRAPH*, 1991.

[SKK04] Y. Sakamoto, S. Kuriyama, and T. Kaneko. Motion map: image-based retrieval and segmentation of motion data. In *SCA*, 2004.

[SL94] D. Simchi-Levi. New worst-case results for the bin-packing problem. *Naval Research Logistics*, 1994.

[SL07] Nader SALMAN and Sylvain Lefebvre. Synthesis of compact textures for real-time terrain rendering. *Master thesis*, 2007.

[SLBJ03] Bongwon Suh, Haibin Ling, Benjamin B. Bederson, and David W. Jacobs. Automatic thumbnail cropping and its effectiveness. In *Symposium on User interface software and technology*, 2003.

[SSCO09] L. Shapira, A. Shamir, and D. Cohen-Or. Image appearance exploration by model-based navigation. In *Computer Graphics Forum*, 2009.

[ST08] J.O.B. Soh and B.C.Y. Tan. Mobile gaming. *Communications of the ACM*, 2008.

[SWRC06] J. Shotton, J. Winn, C. Rother, and A. Criminisi. Textonboost: Joint appearance, shape and context modeling for multi-class object recognition and segmentation. *ECCV*, 2006.

[Sze06] R. Szeliski. Image alignment and stitching: A tutorial. *Foundations and Trends® in Computer Graphics and Vision*, 2006.

[SZS+06] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, and C. Rother. A comparative study of energy minimization methods for markov random fields. *ECCV*, 2006.

[TAH+07] B. Tversky, M. Agrawala, J. Heiser, P. Lee, P. Hanrahan, D. Phan, C. Stolte, and M.-P. Daniel. Cognitive design principles for generating visualizations. *In Applied spatial cognition: from research to cognitive technology*, 2007.

[Tat06] N. Tatarchuk. Dynamic parallax occlusion mapping with approximate soft shadows. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 63–69. ACM, 2006.

[TdSL00] J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 2000.

[TFL83] B.C. Tansel, R.L. Francis, and T.J. Lowe. Location on networks: a survey. *Management Science*, 1983.

[TGY+09] J.O. Talton, D. Gibson, L. Yang, P. Hanrahan, and V. Koltun. Exploratory modeling with collaborative design spaces. 2009.

[TL01] E. Teler and D. Lischinski. Streaming of complex 3d scenes for remote walkthroughs. In *Computer Graphics Forum*, 2001.

[TL09] Edric Tse and Justin Lo. Crowd simulation in astroboy. In *SIGGRAPH ASIA Sketches*, 2009.

[TMJ98] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The Clipmap: A Virtual Mipmap. In *SIGGRAPH*, 1998.

[TMR02] D.S. Taubman, M.W. Marcellin, and M. Rabbani. Jpeg2000: Image compression fundamentals, standards and practice. *Journal of Electronic Imaging*, 2002.

[Tur91] Greg Turk. Generating textures for arbitrary surfaces using reaction-diffusion. In *SIGGRAPH*, 1991.

[TWT+06] Bruce Tartaglia, Rob Wilson, Olcun Tan, Scott Peterson, and Jonathan Gibbs. A procedural modeling workflow for "over the hedge" foliage. In *SIGGRAPH Sketches*, 2006.

[TZL+02] Xin Tong, Jingdan Zhang, Ligang Liu, Xi Wang, Baining Guo, and Heung-Yeung Shum. Synthesis of bidirectional texture functions on arbitrary surfaces. In *SIGGRAPH*, 2002.

[VTP+10] D. Vaquero, M. Turk, K. Pulli, M. Tico, and N. Gelfand. A survey of image retargeting techniques. *Proceedings of SPIE-The International Society for Optical Engineering, Applications of Digital Image Processing*, 2010.

[VW06] JMP Van Waveren. Real-time texture streaming & decompression. *Intel Software Network*, 2006.

[VW08] JMP Van Waveren. Geospatial texture streaming from slow storage devices. *Intel Software Network*, 2008.

[vW09] J.P. van Waveren. id tech 5 challenges:from texture virtualization to massive parallelization. *SIGGRAPH Talk*, 2009.

[Wal91] Gregory K. Wallace. The jpeg still picture compression standard. *Commun. ACM*, 1991.

[WAMV11] Peter Wonka, Daniel Aliaga, Pascal Müller, and Carlos Vanegas. Modeling 3d urban spaces using procedural and simulation-based techniques. In *SIGGRAPH Courses*, 2011.

[War92] Gregory J. Ward. Measuring and modeling anisotropic reflection. *SIGGRAPH*, 1992.

[WB10] D. Wesley and G. Barczak. *Innovation and Marketing in the Video Game Industry: Avoiding the Performance Trap*. Gower Pub Co, 2010.

[Wei04] Li-Yi Wei. Tile-based texture mapping on graphics hardware. In *Graphics Hardware*, 2004.

[WHA07] W. Willett, J. Heer, and M. Agrawala. Scented widgets: Improving navigation cues with embedded visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 2007.

[Whi06] C. White. King kong: the building of 1933 new york city. In *SIGGRAPH Sketches*, 2006.

[WHZ+08] Li-Yi Wei, Jianwei Han, Kun Zhou, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Inverse texture synthesis. *Transactions on Graphics*, 2008.

[Wil83] Lance Williams. Pyramidal parametrics. In *Computer Graphics*, 1983.

[WK91] Andrew Witkin and Michael Kass. Reaction-diffusion textures. In *SIGGRAPH*, 1991.

[WL00] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In *SIGGRAPH*, 2000.

[WL03] Li-Yi Wei and Marc Levoy. Order-independent texture synthesis, 2003. Technical Report TR-2002-01, Stanford University.

[WLKT09] Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. State of the art in example-based texture synthesis. In *Eurographics, EG-STAR*, 2009.

[WNC87] I.H. Witten, R.M. Neal, and J.G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 1987.

[Won03] P. Wonka. Visibility in computer graphics. *Environment and Planning B: Planning and Design*, 2003.

[Wor96] Steven P. Worley. A cellular texturing basis function. In *SIGGRAPH*, 1996.

[WWOH08] Huamin Wang, Yonatan Wexler, Eyal Ofek, and Hugues Hoppe. Factoring repeated content within and among images. *Transactions on Graphics*, 2008.

[WY04] Qing Wu and Yizhou Yu. Feature matching and deformation for texture synthesis. *Transactions on Graphics*, 2004.

[XZCOC12] K. Xu, H. Zhang, D. Cohen-Or, and B. Chen. Fit and diverse: Set evolution for inspiring 3d shape galleries. *ACM Transactions on Graphics*, 2012.

[ZG02] Steve Zelinka and Michael Garland. Towards real-time texture synthesis with the jump map. In *Eurographics Workshop on Rendering*, 2002.

[ZHWW12] Y. Zhou, H. Huang, L.Y. Wei, and R. Wang. Point sampling with general noise spectrum. *Transactions on Graphics*, 2012.

Appendix A

Streaming synthesized texture

A.1 Introduction

In chapter 5 we discussed the streaming of synthesized textures but we only described a general formalism. In this appendix we discuss a texture streaming heuristic that tries to limit the usage of video memory during user exploration of a scene. The heuristic uses the same notations and assumptions made in section 5.3.

The heuristic tries to directly determine the content of $\mathcal{M}$ at rendering time without considering L. Only the information on G are used. However, as described in section 5.3, G requires an amount of memory that is proportional to $O(\max_{(x,y)}(|G[(x,y)]|) \times W \times H)$. Directly storing G in video memory is therefore too expensive.

To overcome the high storage requirement of G, we first describe in section A.2 how to compress the grid G. We then describe in section A.3 the proposed heuristic.

A.2 Compressing G

G has a very coherent content: when moving from one cell of G to its neighbor, the set of visible textures changes a little. This is illustrated in Figure A.1.

To reduce the complexity of storing G, we rely on a lossy compression that replaces G with a smaller structure K. K consists of changing the dense regular grid representation of G into a non–regular coarse decomposition of the scene and still retain most information in G. We choose to represent this decomposition by a Voronoi tessellation of the scene. Each Voronoi cell is represented by its centroid coordinates. Each centroid is associated with the set of textures that are visible from any point within its Voronoi cell. We note K_i the i'th centroid and note $K[i]$ the set of textures associated to the i'th centroid. We note N the number of centroids.

With this representation there is a small number of centroids compared to the number

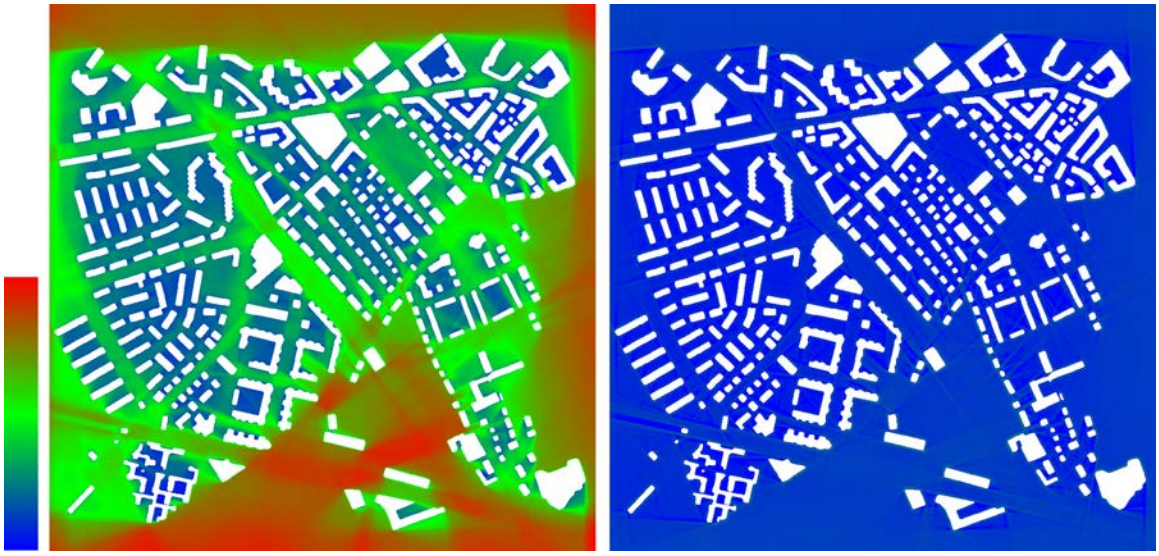

Figure A.1: A map representing a district in the city of Zurich with color information showing the coherent content of G. The white regions represent buildings. In left color ramp, the blue tone represents low values while a red tone represents high values. The map on the left illustrates the number of visible textures per cell in G. The map on the right illustrates the number of different textures when moving from one cell to its 4 neighbors in G. Note how the number of different textures is small (blue) everywhere.

of cells in G. A list of texture identifiers is only stored with a centroid rather than every cell in G. All points within a Voronoi cell share the same list of texture identifiers associated with the cell's centroid.

The compression error induced by K is computed as follows:

$$E(K,G) = \sum_{(x,y)\in G} (R(x,y) \times (|G[(x,y)] \setminus K[D(x,y)]| + |K[D(x,y)] \setminus G[(x,y)]|)$$

D returns the index of the closest visible centroid to the cell (x,y). $R(x,y)$ returns 1 if the location (x,y) is reachable by the viewer and 0 otherwise. For example, Figure A.2 shows a binary map of a district in the city of Zurich where $R(x,y)$ is 1 in black colored regions and $R(x,y)$ is 0 in white colored regions.

The total number of stored identifiers is computed as follows:

$$\mathcal{C}2(K) = \sum_{i=1}^{N} |K[i]|$$

We use a parameter K_{max} that represents an upper limit to $\mathcal{C}2(K)$. K_{max} is set manually.

For a maximum size K_{max}, the goal is to find a set of centroids that minimize the error $E(K,G)$. That is:

Figure A.2: Binary map of a district in the city of Zurich. Black colored regions are reachable by the user. White colored regions are not reachable by the user.

Minimize:

$$E(K, G)$$

Subject to:

$$\mathcal{C}2(K) \leq K_{max}$$

We use a greedy heuristic to find such centroids. This heuristic is described in Algorithm 7.

Algorithm 7 Build K

```
1: // Step1: finding centroids
2: K ← ∅ // empty set of centroids
3: while C2(K) < K_max do
4:    bestC ← (0, 0)
5:    bestE ← ∞
6:    for (x, y) ∈ G do
7:       if (x, y) ∈ K OR R(x, y) = 0 then
8:          continue
9:       end if
10:      if E(K, G) ≤ bestE then
11:         bestE ← E(K, G)
12:         bestC ← (x, y)
13:      end if
14:   end for
15:   K ← K ∪ bestC
16: end while
17: // Step2: texture identifiers assignment
18: for (x, y) ∈ G do
19:   K[D(x, y)] ← K[D(x, y)] ∪ G[(x, y)]
20: end for
```

Figure A.3 shows the result of Algorithm 7 on the map shown in Figure A.2 and Figure A.1. After building K, $K[D(x, y)]$ gives an approximate visible set of textures at location (x, y) in G.

A.3 Heuristic

We now describe a heuristic that uses K and tries to directly determine the content of $\mathcal{M}$ at rendering time. This heuristic is based on a texture priority function P.

A.3.1 Texture priority function

The priority function is defined so that textures with high priorities are synthesized and loaded in memory while textures with low priorities and residing in video memory are evicted to ensure the constraint $C1$.

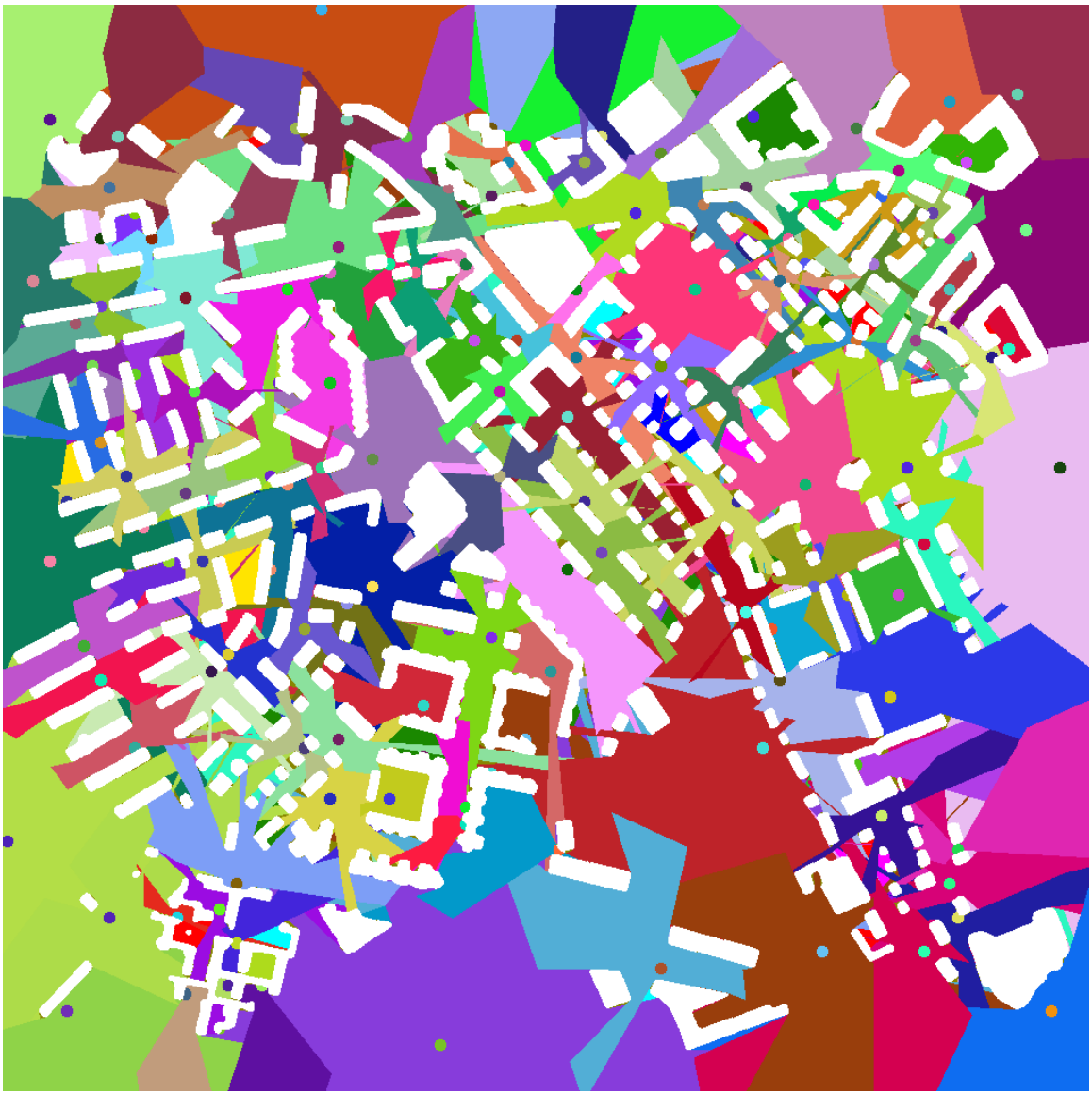

Figure A.3: Each cluster in K is colored randomly. The white describes regions not reachable by the user.

We define the priority of a texture as function of the number of visible texels in the texture, the distance from the viewer to the surface on which the texture is mapped on and the view direction.

We note $P(T_i, \mathbf{v}, l)$ the priority of texture T_i when the viewer is at position $\mathbf{v}$ and looking in direction l. P is defined as:

$$P(T_i, \mathbf{v}, l) = \omega_0 \times N_{visible}(\mathbf{v}, T_i) - \omega_1 \times L(p(\mathbf{v}, T_i)) + dot(l, v)$$

$N_{visible}(\mathbf{v}, T_i)$ is the number of texels in T_i seen by the viewer: it is proportional to the error visible on screen if the texture is missing. $p(\mathbf{v}, T_i)$ is the shortest path from the viewer to the closest surface on which the texture T_i is mapped. It is computed using Dijkstra shortest path algorithm in the 2D map G. $L(p)$ is the length of the path p. v is the direction towards the texture T_i. ω_0 is a regularization constant set by the user to give more importance to $N_{visible}$ when its value is high. ω_1 is a regularization constant set by the user to give more importance to $L(p)$ when its value is high.

Note that the Unreal Engine 3 texture streaming system uses a similar priority function but only relies on euclidean distance instead of a geodesic distance. We found that this is not optimal in highly occluded environments where the viewer may follow a long path until seeing an object.

Since P requires a shortest path computation every time it is called, calling P multiple times can be computationally expensive. This is wasteful as textures far from the viewer are likely to have a low priority. The computation of P is only relevant for a relatively small neighborhood around the viewer. We therefore only compute P in a neighboring of G around the viewer. We manually fix the size of the neighborhood depending on the complexity of the scene.

For textures outside the fixed neighborhood, we use an approximate priority consisting in the euclidean distance between the viewer and the surface on which the texture is mapped. We do not check visibility for the approximate priority.

A.3.2 Algorithm

The heuristic works following Algorithm 8. At each frame, the set $\mathcal{T}$ of all textures in the scene is ordered depending on the priorities (line 3). Similarly, the set $\mathcal{M}$ of textures in video memory are also ordered depending on their priorities (line 4). Textures in $\mathcal{T}$ are then synthesized and loaded in video memory from the highest to lowest priority (line 5). A texture is not synthesized if it is already in $\mathcal{M}$ (line 7). Also, At some point the video memory is filled and in this case, textures with low priorities are evicted to make room for textures with higher priorities (line 10). The loading process stops either if the size of the synthesized data exceeds $\mathcal{B}_{max}$ (line 17) or if high priority textures fill the memory (line

20).

Algorithm 8 Streaming heuristic

for frame f **do**
  $sz \leftarrow 0$
  $sort(\mathcal{T}, P)$ // sort $\mathcal{T}$ from highest to lowest priority
  $sort(\mathcal{M}, P)$ // sort $\mathcal{M}$ from highest to lowest priority
  for $T_i \in \mathcal{T}$ **do**
    if $T_i \in \mathcal{M}$ **then**
      continue
    end if
    while $|\mathcal{M}| \geq \mathcal{M}_{max}$ AND $P(back(\mathcal{M}), \mathbf{v}, l) < P(T_i, \mathbf{v}, l)$ **do**
      $popBack(\mathcal{M})$
    end while
    if $|\mathcal{M}| < \mathcal{M}_{max}$ **then**
      $synthesize(T_i)$
      $pushFront(\mathcal{M}, T_i)$
      $sz \leftarrow sz + 1$
      if $sz \geq \mathcal{B}_{max}$ **then**
        break
      end if
    else
      break
    end if
  end for
end for

Annexe B

Introduction

B.1 Contexte général

Les environnements virtuels sont généralement représentés par un mélange de polygones capturant la géométrie de la scène et des images, appelées *textures*, capturant les détails essentiels des différentes surfaces (Figure B.1). Les textures, représentent les détails géométriques fins des différents matériaux tels que l'albédo de la surface ou la normale géométrique. Ces informations sont nécessaires pour décrire comment la lumière interagit avec une surface texturée. La Figure B.2, par exemple, montre comment représenter une peau humaine réaliste en utilisant plusieurs couches de textures qui définissent différents types de matériaux.

Les textures dans les applications interactives Dans les applications graphiques interactives, les scènes virtuelles doivent être rendus avec un taux de rafraîchissement élevé. Cela est particulièrement le cas dans les applications contenant des animations et des mouvements rapides comme les jeux de tir à la première personne ou encore les simulateurs de course. Souvent, ces applications ont besoin d'un taux de rafraîchissement aussi élevé que 60 images par seconde pour avoir un affichage fluide. Cependant, pour atteindre un taux de rafraîchissement aussi élevé, la complexité géométrique est souvent simplifiée et les textures sont utilisées pour compenser la perte d'informations géométriques. Ce concept est décrit dans la Figure B.3 qui montre un exemple d'un personnage de jeu vidéo où les textures représentent les détails fin alors qu'une géométrie simple représente les informations grossières. Pour construire un tel personnage, l'artiste construit d'abord une version très détaillée avec des centaines de milliers de polygones. Les polygones sont ensuite simplifiés et le résidu de détails est stocké dans une texture. La texture est ensuite utilisée pendant le rendu pour permettre des interactions lumineuses réalistes.

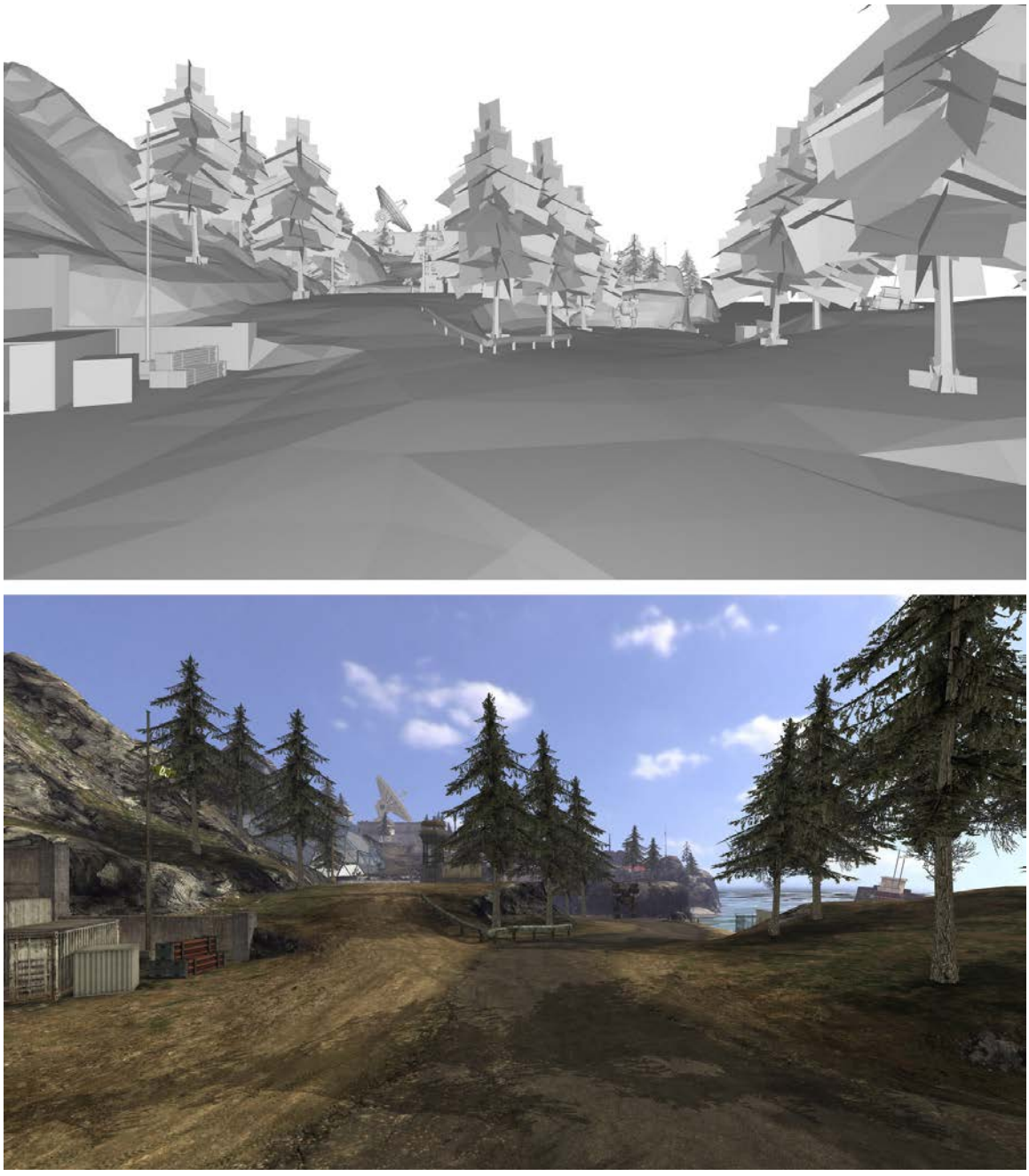

FIGURE B.1 – Une scène de jeux vidéo avec et sans textures. *Haut :* La scène sans textures. *Bas :* La même scène avec textures. *Source :* Quake wars, Splash Damage/Activision.

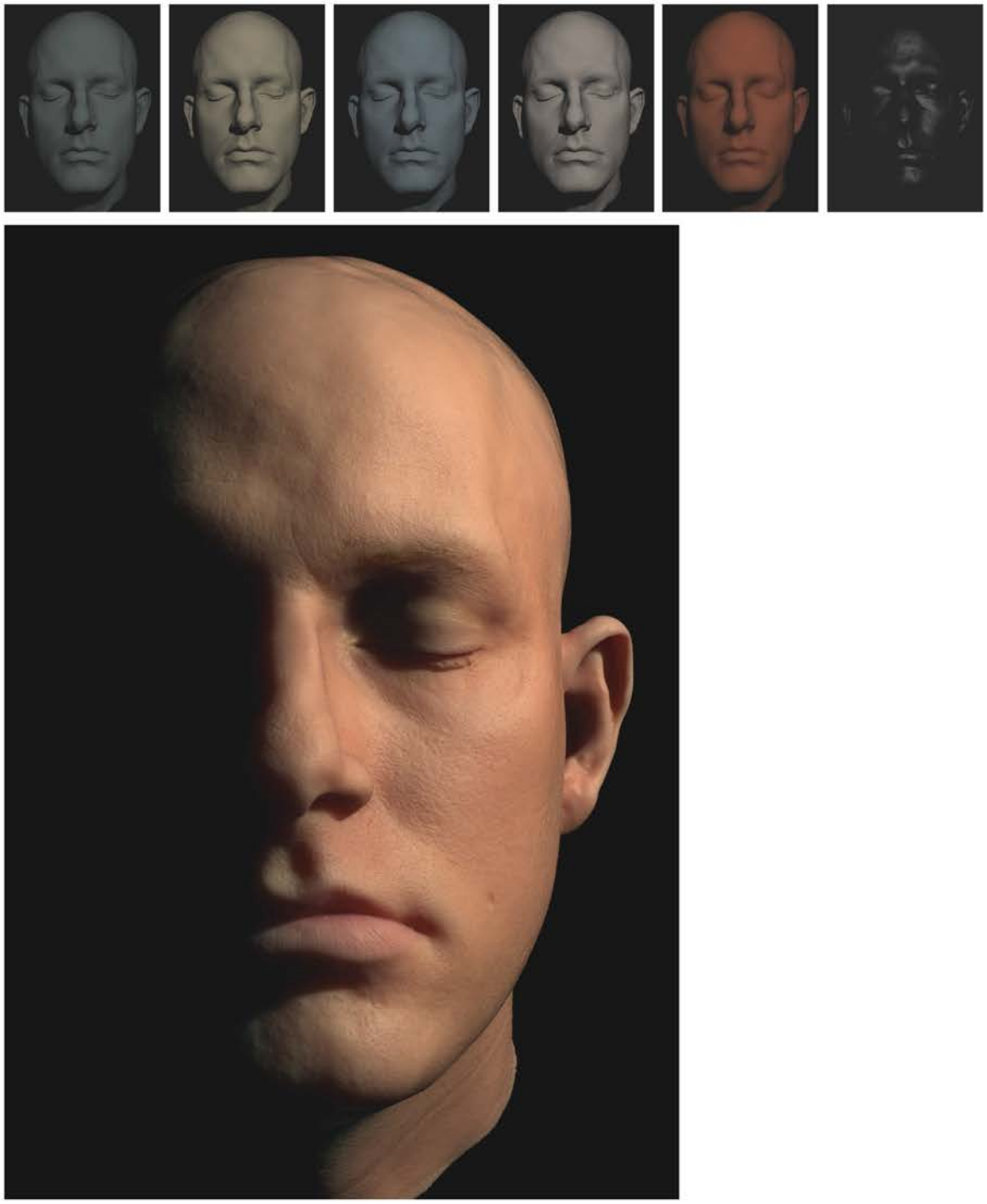

FIGURE B.2 – *Top :* Plusieurs couches de texture décrivant une peau humaine. *Bottom :* Rendu avec les différentes couches de texture. *Source :* Donner et Jensen [DJ05].

FIGURE B.3 – Textures appliquées sur un simple modèle géométrique. *Gauche :* Un modèle composé d'un petit nombre de polygones affichés en fil de fer. Ce modèle est obtenu par simplification d'une version de plus haute résolution. *Milieu :* Les matériaux et les détails géométriques du modèle de haute résolution sont stockés dans des textures 2D. La texture en haut représente l'albédo, celle au milieu représente les normales géométriques et celle en bas représente l'intensité des reflets spectaculaires. *Droite :* Rendu du modèle basse résolution texturé. *Source :* Guild Wars, ArenaNet/NCsoft.

FIGURE B.4 – Textures dans les anciens jeux vidéo. *Gauche :* Quake, un jeu sorti il y a plus de 15 ans, utilise des textures de basse résolution allant de 16^2 pixels jusqu'à seulement 256^2 pixels. *Source :* Quake, Id Software/GT Interactive. *Droite :* Quake 3 utilise les processeurs graphiques qui possèdent leur propre mémoire vidéo pour afficher des textures d'une résolution atteignant les 512^2 pixels. *Source :* Quake 3 Arena, Id Software/Activision.

Limitation des textures Dans les applications interactives, les textures représentent souvent la ressource la plus coûteuse en mémoire vidéo et en bande passante.

Pour limiter l'utilisation de la mémoire, les anciens jeux vidéo ont dû utiliser une faible quantité de textures de basse résolution. Les même textures sont répétées plusieurs fois sur différentes surfaces (Figure B.4, à gauche). Avec l'introduction des processeurs graphiques, les développeurs de jeux vidéo ont considérablement augmenté la quantité de textures utilisées (Figure B.4, à droite). Les textures ont été non seulement appliquées sur des surfaces, mais ont été également utilisées pour la création d'effets spéciaux.

Aujourd'hui, les processeurs graphiques offrent aux développeurs la possibilité de contrôler la plupart des opérations graphiques. Cela leurs a permis d'utiliser une plus grande quantité de textures de plus haute qualité. Par exemple, dans le jeu récent RAGE (Figure B.5), les développeurs d'*id Software*, ont programmé le processeur graphique pour afficher une texture avec une résolution supérieure à 128000^2 pixels.

Conséquences dans l'industrie du jeu vidéo L'utilisation d'une quantité énorme de textures implique des efforts considérables pendant la production des jeux vidéo : Les artistes doivent manuellement créer chacune des textures composant le jeu. De même, les ingénieurs doivent mettre en œuvre des moteurs graphiques permettant une gestion efficace de centaines de giga-octets de données de textures. Cela conduit souvent à des années de développement et des coûts atteignant des centaines de millions d'euros [WB10].

Textures dans les nouveaux médias La convergence entre l'informatique graphique et Internet a conduit à l'émergence de nouveaux médias allant des jeux de rôle en ligne aux

FIGURE B.5 – Quantité importante de textures dans le jeux RAGE. Chaque objet possède ses propres détails. *Source :* RAGE, Id Software/Bethesda Softworks.

programmes géographiques tel que Google Earth. Ces applications nécessitent l'affichage d'environnements encore plus vastes et plus riches ce qui nécessite encore plus de mémoire et de bande passante. Par exemple, Google Earth permet aux utilisateurs de naviguer à travers toute la surface de la terre. Les textures qui composent le monde sont obtenues à partir d'images satellitaires d'une taille totale dépassant les 70 téraoctets de données [CDG+08]. A cause de cela, ces volumes de textures sont transmis progressivement sur le réseau Internet qui est extrêmement limité en bande passante.

Un autre secteur en pleine croissance est celui des jeux sur téléphones et consoles mobiles [ST08, LLP12]. Les téléphones mobiles sont aujourd'hui capables d'afficher du contenu en 3D et en temps réel. De plus, les jeux pour mobiles sont accessibles à un public très large avec des prix accessibles (Figure B.6). Même les moteurs de jeux comme *Unreal Engine 3* d'*Epic* ou *CryENGINE 3* de *Crytek* qui visent en premier lieu les jeux sur PC et consoles sont également disponibles pour le développement de jeux sur téléphones mobiles. Cependant, des contraintes encore plus fortes surviennent lors de l'utilisation des textures sur des appareils mobiles : Les capacités de stockage et de bande passante sont très limitées. De plus, le transfert de textures consomme énormément d'énergie et de batteries [CPAM08, JGDAM12].

Approches existantes pour contourner le problème Pour faire face aux limites liées aux textures, les applications interactives sacrifient souvent la qualité visuelle des textures pour les rendre plus petites. Cela est fait aux travers de méthodes de compression agressives

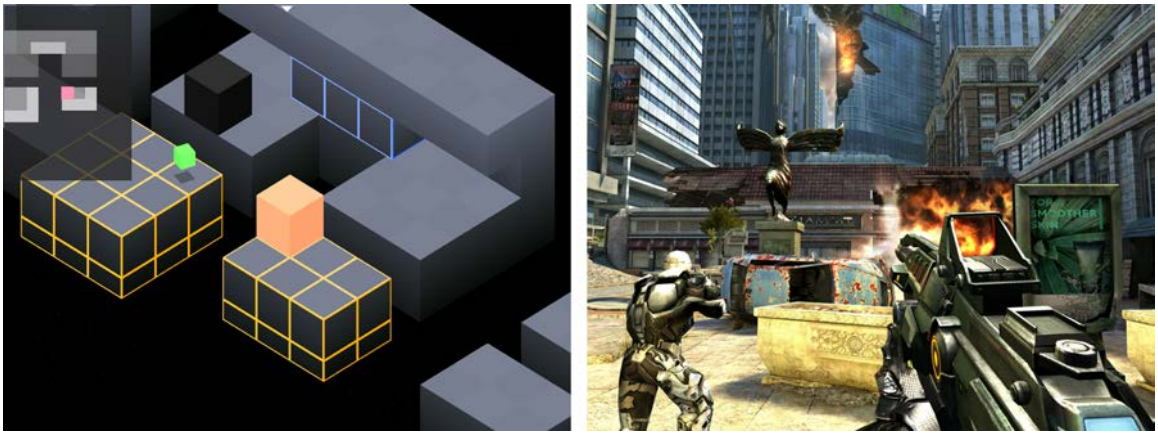

FIGURE B.6 – Évolution de jeux vidéo sur mobiles. *Gauche :* Les premiers jeux 3D sur mobiles utilisaient des éléments géométriques simples dépourvus de textures. *Source :* Edge, Mobigame. *Right :* Les jeux 3D récents sur mobiles utilisent une quantité de textures importante pour afficher des scènes réalistes. *Source :* N.O.V.A. 3, Gameloft.

FIGURE B.7 – Textures basse résolution utilisées dans Google Earth pour afficher une version virtuelle de Las Vegas. *Source :* Google Earth.

et avec perte. Par exemple, pour afficher son contenu en 3D, Google Earth applique beaucoup de compression et cela résulte en une qualité limitée comme illustré sur la Figure B.7.

Une autre approche très utilisée en pratique est la répétition des textures en pavage sur les surfaces. Cela réduit la quantité de mémoire nécessaire mais se traduit par un manque

FIGURE B.8 – Pour habiller un large terrain avec des textures sans contraindre la mémoire vidéo, des morceaux de texture sont répétés sur la surface du terrain. Vues de loin, nous pouvons remarquer la répétition des textures. Ceci affecte le réalisme de la scène. *Source :* The Elder Scrolls 5 : Skyrim, Bethesda Game Studios/Bethesda Softworks.

de variété et des effets de répétition quand les surfaces sont vues de loin (Figure B.8).

Synthèse de texture Plutôt que de réduire la résolution des textures ou encore répéter les même textures, les chercheurs ont essayé de trouver d'autres alternatives pour réduire la taille des textures et en même temps réduire le temps nécessaire pour les produire. Un des développements majeur en recherche est celui de la synthèse de texture.

La synthèse de texture représente des techniques permettant la création d'une texture d'une manière complètement automatique. Le but est de remplacer les textures par des algorithmes produisant une quantité infinie de textures. De cette façon, au lieu de stocker des images représentant les textures, seul des algorithmes et leurs paramètres ont besoin d'être stockés. La Figure B.9 montre une scène qui utilise exclusivement des textures synthétisées. Dans cette scène, les algorithmes des différentes textures ainsi que leurs paramètres sont stockés en quelques kilo-octets de mémoire au lieu de plusieurs méga-octets.

Il faut noter que les paramètres des algorithmes de synthèse de texture peuvent être contrôlés. Cela permet de générer une infinité d'apparences différentes offrant ainsi la possibilité de personnaliser chaque texture d'une manière unique. Par exemple, les paramètres correspondant aux apparences des textures de la Figure B.9 peuvent être modifiée pour obtenir les apparences de la Figure B.10. Faire de tels effets dynamiques avec des textures

FIGURE B.9 – Une scène entièrement composée de textures synthétisées crées avec l'outil *Allegorithmic Substance Designer*.

FIGURE B.10 – Les paramètres des textures de la Figure B.9 sont modifiés pour changer l'apparence de la scène.

classiques exigerait une quantité importante de mémoire pour stocker toutes les textures et leurs transitions. Cela nécessiterait également un temps considérable aux artistes pour créer les différentes textures et toutes leurs transitions.

Avec le succès des applications graphiques sur internet et sur mobiles, les industriels ont commencé à s'intéresser à la synthèse de texture pour développer d'une manière rapide des applications graphiques à faible coûts mémoires. Cela a conduit à l'élaboration de nouveaux outils de synthèse de texture comme *Substance* de la société *Allegorithmic*, *Genetica* de la société *Spiral Graphics* ou encore *Filter Forge*. Ces outils se basent sur une

représentation avec des graphes permettant aux artistes de concevoir plus facilement des textures complexes et paramétrables.

B.2 Difficultés

Malgré beaucoup de recherches en synthèse de textures et malgré le développement d'outils permettant la création de textures synthétisées, la synthèse de texture est peu utilisée en pratique dans les applications interactives. Cela est dû, en partie, aux deux difficultés suivantes :

- **Difficulté 1 :** La sélection de paramètres d'une texture synthétisée peut être complexe et fastidieux. L'espace des paramètres est généralement très important et l'utilisateur doit passer beaucoup de temps à visualiser et explorer les différentes apparences que peut prendre la texture synthétisée. Cela est particulièrement contraignant lorsque l'utilisateur considère plusieurs textures à la fois comme dans un marché virtuel de textures tel que la librairie de substances sur *TurboSquid*.

  Pour surmonter cette difficulté, il faudrait un moyen efficace pour visualiser et explorer les textures synthétisées.

- **Difficulté 2 :** Pendant le rendu, les accès se font à des sous–parties de textures et ceci d'une manière aléatoire. Nous appelons accès *par point* un accès aléatoire à un seul pixel d'une texture et ceci indépendamment des autres pixels.

  Les algorithmes de synthèse ne permettent souvent pas d'évaluer la texture par point. Souvent toute la texture doit être produite pour obtenir la valeur d'un seul pixel. Il est donc nécessaire de générer la texture et la stocké en mémoire avant l'étape de rendu. Ceci peut être très couteux en mémoire vidéo surtout quand plusieurs textures doivent être générées et stockées en même temps.

  Pour surmonter cette difficulté, il faudrait représenter le résultat de synthèse d'une manière compacte en mémoire. Aussi, une texture ne faudrait être générée que lorsqu'elle est nécessaire pour l'image rendue à l'écran.

B.3 Objectif et contributions

L'objectif de cette thèse est de promouvoir l'utilisation de la synthèse de texture dans le but d'afficher de vastes environnements riches en détails sans contraindre la mémoire vidéo et la bande passante.
À ces fins, l'objectif est de simplifier l'utilisation des textures synthétisées, de représenter le résultat de synthèse de manière compacte et décoder les données de textures d'une manière

rapide et seulement quand la texture est nécessaire pour le rendu de l'image finale.
Plus précisément, nous proposons de résoudre les deux difficultés décrites dans la section précédente au travers des deux contributions suivante :

- Pour surmonter la **Difficulté 1** et faciliter la sélection, la visualisation et l'exploration des paramètres des textures synthétisées, nous présentons deux contributions : une image résumant les apparences que peut prendre une texture synthétisée ainsi que des curseurs visuels résumant les apparences de chaque paramètre de la texture.

  L'image résumée consiste en une image statique et continue qui résume, dans un espace limité de pixels, les apparences produites par un algorithme de synthèse. Cela permet à l'utilisateur final de comprendre rapidement le contenu d'une texture et de faire une sélection rapide d'apparences directement dans le résumé.

  Les curseurs visuels remplacent les curseurs classiques utilisés pour contrôler les paramètres par des bandes visuelles qui sont dynamiques et qui révèlent les apparences qui varient le plus quand le curseur est modifié.

  Dans les deux types de résumé, le défi est de s'assurer que la plupart des apparences sont visibles, sont attribuées un espace similaire dans le résumé et sont ordonnées d'une manière continue. De plus, dans le cas des curseurs visuels, il faut révéler autant que possible les variations visuelles induites par le curseur.

  Ces résumés sont décrits dans l'annexe C.

- Pour surmonter la **Difficulté 2** nous proposons une technique de synthèse de texture par l'exemple qui est rapide, de haute qualité et implémentée de manière parallèle sur le processeur graphique.
  Plutôt que de générer le résultat de ce synthétiseur sous forme d'une image bitmap, le résultat est une structure encodée de manière compacte en mémoire. Les données de cette structure sont décodées d'une manière rapide pendant le rendu. L'algorithme de synthèse est décrit dans l'annexe D alors que l'encodage et le décodage de la structure compacte sont décrits dans l'annexe E.

  Pour éviter de stocker toutes les textures avant le rendu et ceci même sous forme d'une structure compacte, nous discutons du problème de streaming de texture qui vise à ne synthétiser une texture que lorsqu'il est probable qu'elle soit visible dans un future proche. La méthode de streaming doit s'assurer que l'opération de synthèse doit commencer assez tôt de telle sorte que la texture soit déjà synthétisée et chargée en mémoire quand elle devient visible. Le streaming est décrit dans l'annexe E.

Les contributions décrites ci-dessus apparaissent dans les publications suivantes :

- **By-example Synthesis of Architectural Textures**
  Sylvain Lefebvre, Samuel Hornus and Anass Lasram
  ACM Transactions on Graphics (SIGGRAPH conference proceedings), 2010
  patent - FR 10/02902, 2010

- **Coherent Parallel Hashing**
  Ismael Garcia, Sylvain Lefebvre, Samuel Hornus and Anass Lasram
  ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia), 2011

- **Procedural Texture Preview**
  Anass Lasram, Sylvain Lefebvre and Cyrille Damez
  Computer Graphics Forum (Eurographics conf. proc.), 2012

- **Scented Sliders for Procedural Textures**
  Anass Lasram, Sylvain Lefebvre and Cyrille Damez
  Computer Graphics Forum (Eurographics conf. proc.) - short paper, 2012

- **Parallel Patch-based Texture Synthesis**
  Anass Lasram and Sylvain Lefebvre
  High Performance Graphics, 2012

Annexe C

Exploration des textures synthétisées

C.1 Introduction

Les textures procédurales exposent souvent un ensemble de paramètres qui contrôlent leurs apparences. Cela permet aux utilisateurs finaux de régler, généralement au travers d'un ensemble de curseurs (Figure C.1) l'apparence finale d'une texture. Cependant, il est difficile d'explorer une base de données de textures procédurales tellement le nombre d'apparences possibles est élevé. De même, il est difficile de prédire les changements introduits par un curseur donné sachant que l'effet du curseur dépend souvent des autres curseurs.

Dans cette annexe, notre objectif est de faciliter l'exploration de grandes bases de données de textures procédurales permettant ainsi à l'utilisateur de visualiser et de naviguer facilement dans l'espace des apparences possibles.

Notre première contribution est un algorithme qui résume toutes les apparences d'une texture procédurale dans une seule image, rendant ainsi facile la visualisation des apparences possibles. Le principal défi est de résumer la grande variété d'apparences dans un même petit espace. Un exemple d'un résumé d'une texture procédurale est illustré dans la Figure C.2.

Notre seconde contribution est un algorithme générant une bande visuelle qui résume les apparences de chaque paramètre de la texture procédural. Ces bandes visuelles remplacent les curseurs classiques utilisés dans les interfaces de nombreux logiciels de texture procédurale. Ces bandes visuelles sont particulièrement utiles pour aider l'utilisateur à prédire l'effet visuel de chaque paramètre. Nous construisons ces bandes visuelles de manière à accentuer les changements visuelles induits par chaque paramètre. Des bandes visuelles sont illustrées dans la Figure C.3.

Notre troisième contribution est de combiner le résumé de textures procédurales et les bandes visuelles dans une même interface. L'utilisateur peut directement sélectionner une

FIGURE C.1 – *Gauche :* Deux textures produites par la même procédure. *Droite :* Les curseurs contrôlant l'apparence de la texture.

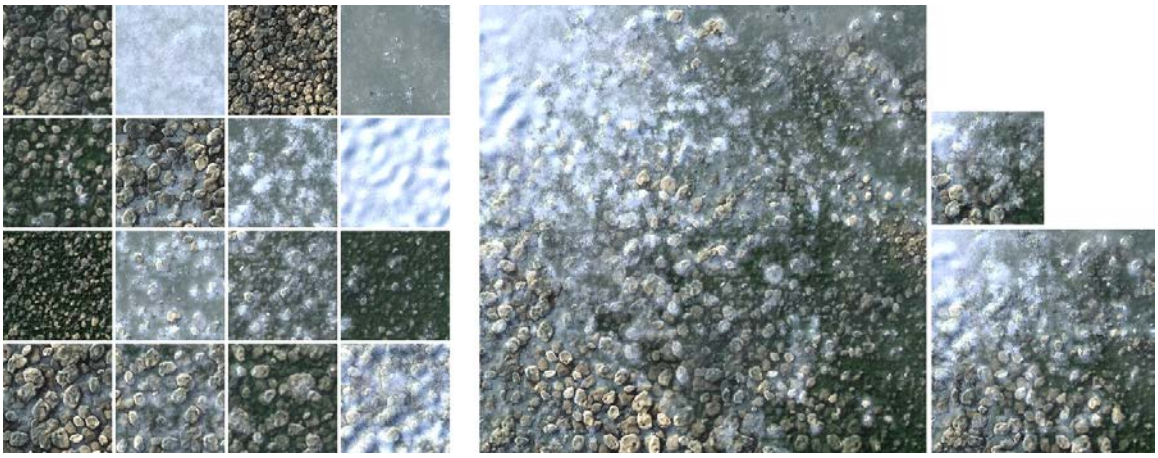

FIGURE C.2 – Gauche : 16 textures miniatures issues d'un même synthétiseur. Droite : Notre résumé de texture procédurale affiche les apparences possibles dans une même image. Des résumés de tailles 512^2 pixels, 256^2 pixels et 128^2 pixels sont affichés. Il est à noter que même le petit résumé de taille 128^2 pixels affiche bien les éléments importants de la texture à savoir : l'eau gelée, la neige, la mousse, les cailloux ainsi que des transitions entre eux.

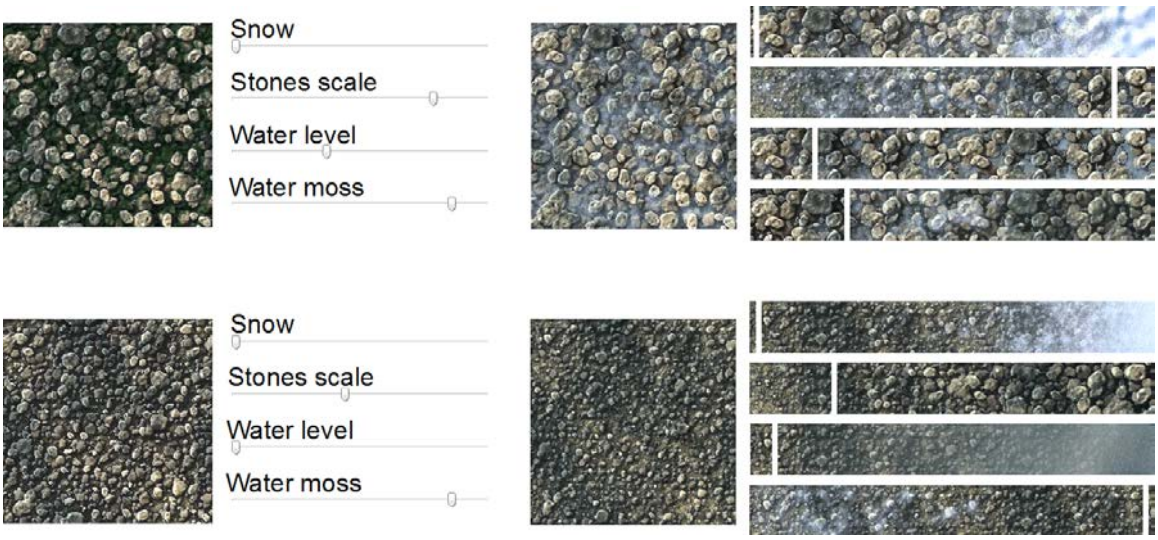

FIGURE C.3 – Gauche : Deux variantes d'une même texture procédurale avec les curseurs contrôlant les paramètres. Droite : La même texture procédurale avec des bandes visuelles contrôlant les paramètres. Les changements possibles des apparences peuvent être directement perçus dans les bandes visuelles. Ces bandes sont reconstruites automatiquement quand les paramètres de la texture changent.

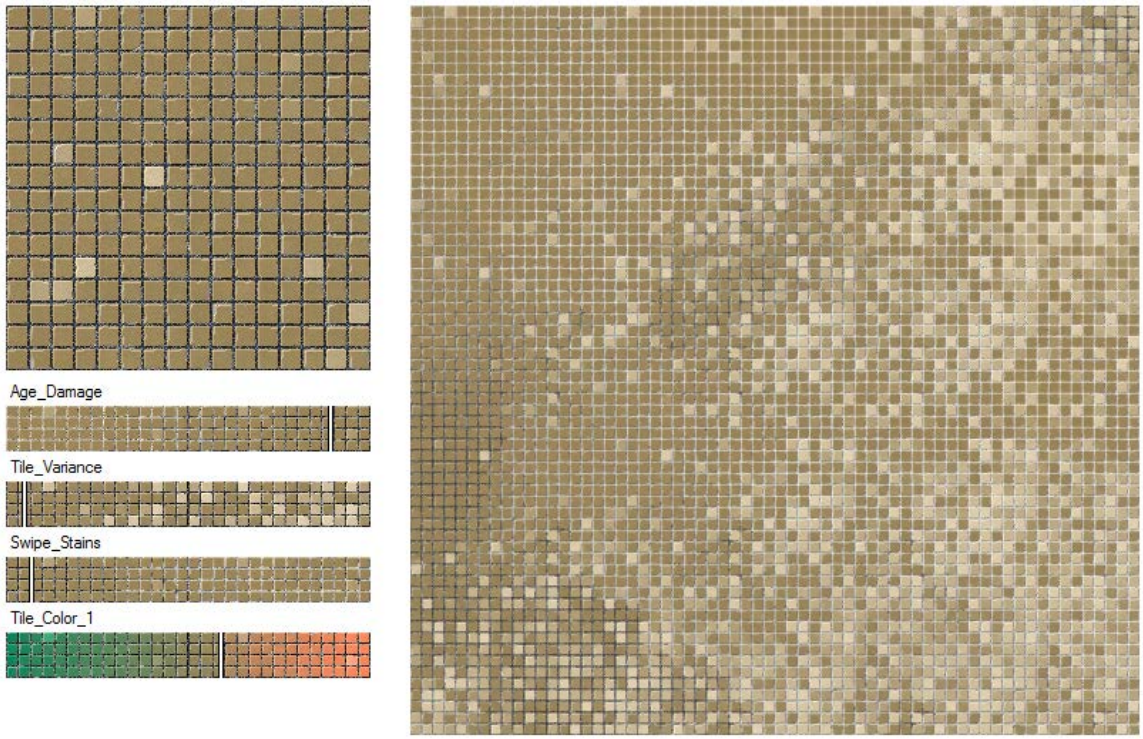

FIGURE C.4 – Interface utilisant les résumés visuels. L'utilisateur peut rapidement sélectionner une apparence dans le grand résumé à droite. Les curseurs avec bandes visuelles permettent de faire des ajustements précis.

apparence dans le résumé puis utiliser les bandes visuelles pour faire des ajustements précis. Une telle interface est illustré dans la figure C.4.

C.2 Résumé de textures procédurales

Étant donné un générateur procédural $g(p)$ permettant de générer une image I_p à partir d'un vecteur de paramètres p ($I_p = g(p)$), le but est de construire une image résumé $\mathcal{T}$ contenant la plupart des apparences produites par le générateur procédural g. Nous notons $\mathcal{C}$ l'espace des apparences produites par le générateur procédural g. Nous utilisons une fonction de similarité $\mathcal{M}(I_p, I_q)$ renvoyant une faible valeur quand les textures I_p et I_q sont similaires. Il est important de comparer les textures au lieu de leurs paramètres pour avoir une distribution uniforme d'apparences dans le résumé (Figure C.5).

Pour construire $\mathcal{T}$, nous construisons d'abord un tableau 2D $\mathcal{X}$ contenant les vecteurs de paramètres qui produisent des apparences que nous appelons : *apparences représentatives*. Les apparences représentatives doivent capturer la plupart des apparences produites par g (*Complétude*). Aussi, les apparences représentatives doivent être aussi variées que possible (*Variété*). Enfin, les apparences représentatives doivent être disposées de manière à ce que les apparences similaires se trouvent dans un même voisinage (*Disposition*). Pour choisir

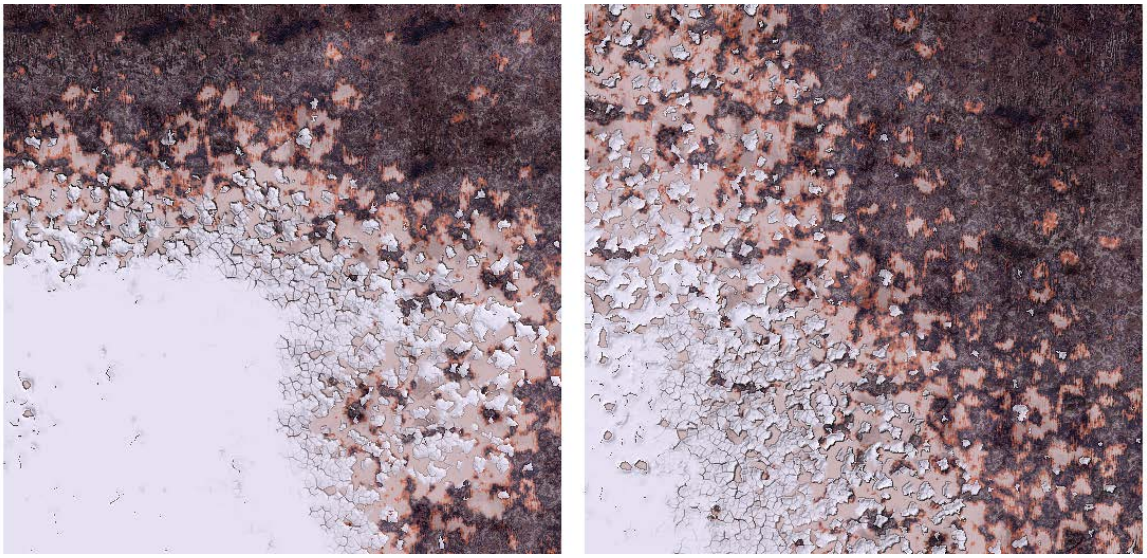

FIGURE C.5 – *Gauche :* Utilisation d'une fonction de similarité $\mathcal{M}$ comparant les paramètres d'une texture procédurale. *Droite :* Utilisation d'une fonction de similarité $\mathcal{M}$ comparant les textures directement.

ces apparences représentatives et les ordonner dans $\mathcal{X}$, nous optimisons les trois critères suivants :

- *Complétude* : Chaque apparence de $\mathcal{C}$ doit être aussi proche que possible à une apparence représentative dans $\mathcal{X}$. Cela revient à *minimiser* la fonction objectif :

$$E_C(\mathcal{X}) = \max_{p \in \mathcal{C}} \min_{q \in \mathcal{X}} (\mathcal{M}(I_p, I_q))$$

- *Variété* : Les apparences représentatives doivent être aussi variées que possible. Cela revient à *maximiser* la fonction objectif :

$$E_V(\mathcal{X}) = \min_{r \in \mathcal{X}} \min_{q \in \mathcal{X}} \min_{r \neq q} (\mathcal{M}(I_r, I_q))$$

- *Disposition* : Nous voulons ordonner les apparences représentatives de facon à ce que nous ayons des transitions lisses dans $\mathcal{T}$. Cela revient à *minimiser* la fonction objectif :

$$E_S(\mathcal{X}) = \sum_{p \in \mathcal{X}} \sum_{q \in \mathcal{N}_p} \mathcal{M}(I_p, I_q)$$

  Où $\mathcal{N}_p$ représente un voisinage autour de p dans $\mathcal{X}$.

Nous optimisons simultanément pour ces trois fonctions objectifs en utilisant une version modifiée de l'algorithme *self–organizing map* [KSH01].

Une fois la carte $\mathcal{X}$ construite, nous générons les textures associées aux paramètres des apparences représentatives. Un morceaux (patch) est sélectionné dans chaque texture

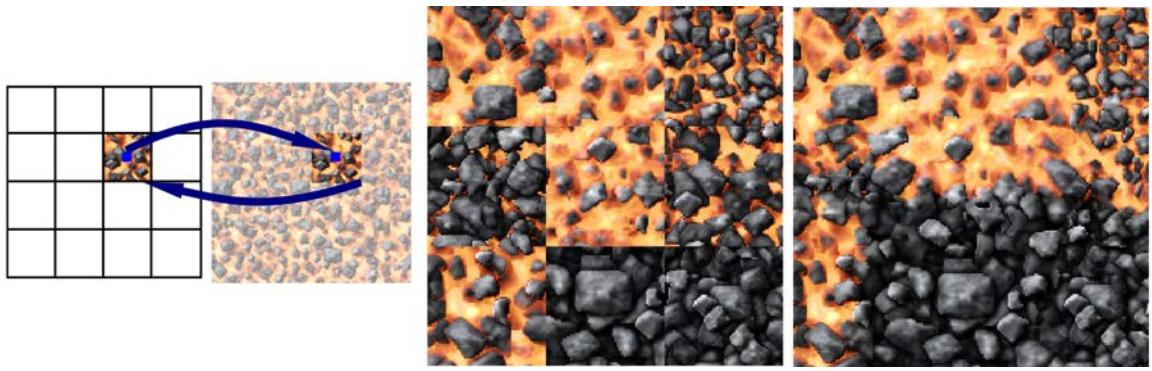

FIGURE C.6 – *Gauche :* Un patch est associé à chaque apparence représentative. *Milieu :* Sans aucun traitement, des discontinuités apparaissent aux bords des patchs. *Droite :* La synthèse par patch élimine ces discontinuités.

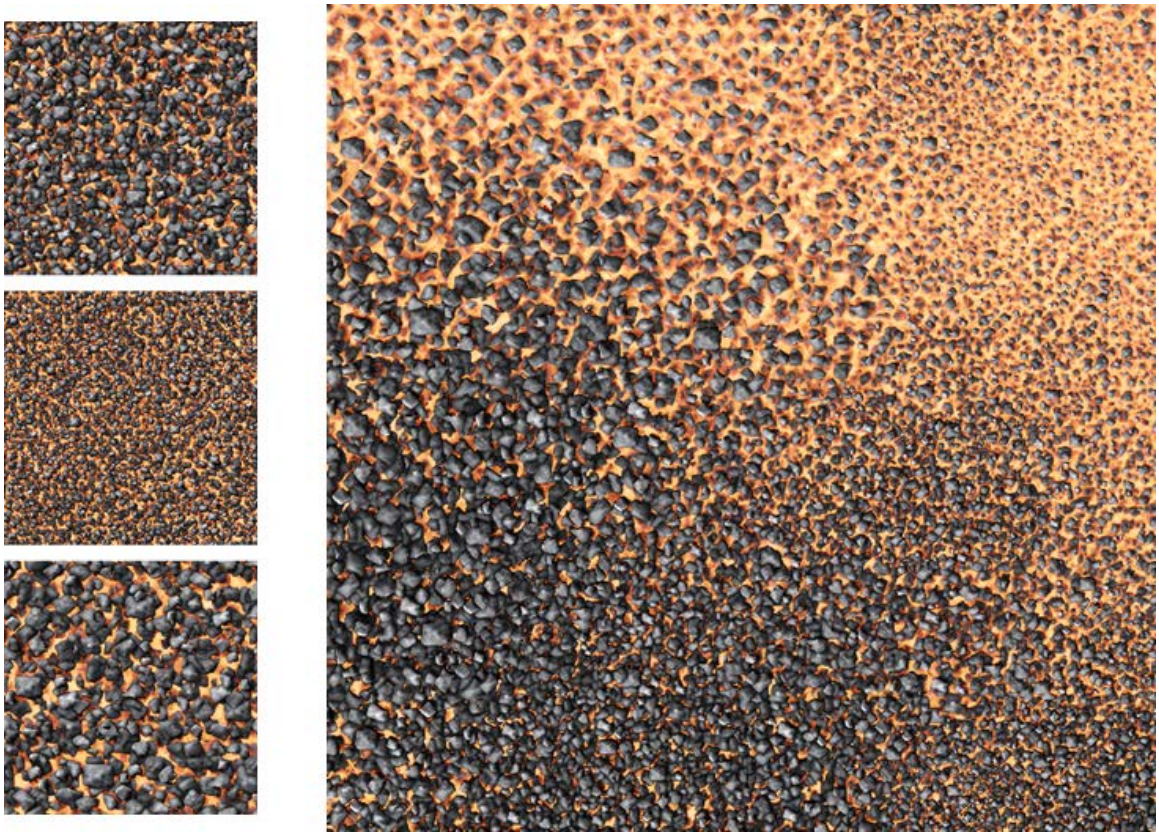

FIGURE C.7 – Résultat de synthèse.

et $\mathcal{T}$ est construite en carrelant les patchs comme dans une mosaïque. Cependant, des discontinuités apparaissent souvent aux bords des patchs (Figure C.6). Nous utilisons un synthétiseur par patch pour éliminer ces discontinuités. Le résultat final associé à la texture de la Figure C.6 est présenté dans la Figure C.7.

C.3 Curseurs à bandes visuelles

Pour synthétiser une bande visuelle pour chaque curseur, nous essayons de concentrer, dans le petit espace de la bande, les données visuelles qui varient le plus. Partant d'une grande image qui montre tous les changements visuelle, nous aimerions enlever les données visuelles qui varient peu et ne garder dans la bande finale que les données visuelles essentielles. Aussi, nous aimerions que le résultat soit consistant avec le curseur. Nous faisons cela au travers des étapes suivantes :

- Une image de grande taille est formée en répétant la texture procédurale un certain nombre de fois horizontalement. Cette grande image est divisée en une grille de patchs (Figure C.8, ***(b)***) où, selon la position du curseur, chaque colonne de patchs reflète des apparences différentes. Par exemple, la colonne de gauche reflète les apparences produites si le curseur est complètement à gauche (valeur minimale du paramètre associé). La colonne de droite par contre, reflète les apparences produites si le curseur est complètement à droite (valeur maximale du paramètre associé).

- Une carte d'importance est calculée (Figure C.8, ***(c)***) en donnant un score à chaque patch en fonction de la variance des apparences. Les patchs qui varient le plus auront des scores plus élevés.

- L'ensemble composé de tous les patchs est réduit en supprimant les patchs ayant de faibles scores (Figure C.8, ***(d)***). Pour limiter les distorsions et préserver la structure des patchs, *Seam–Carving* [AS07] est appliqué sur la carte d'importance pour la suppression des patchs.

- Les patchs restants (Figure C.8, ***(e)***) sont collés ensemble à l'aide d'une synthèse par patch (Figure C.8, ***(f)***).

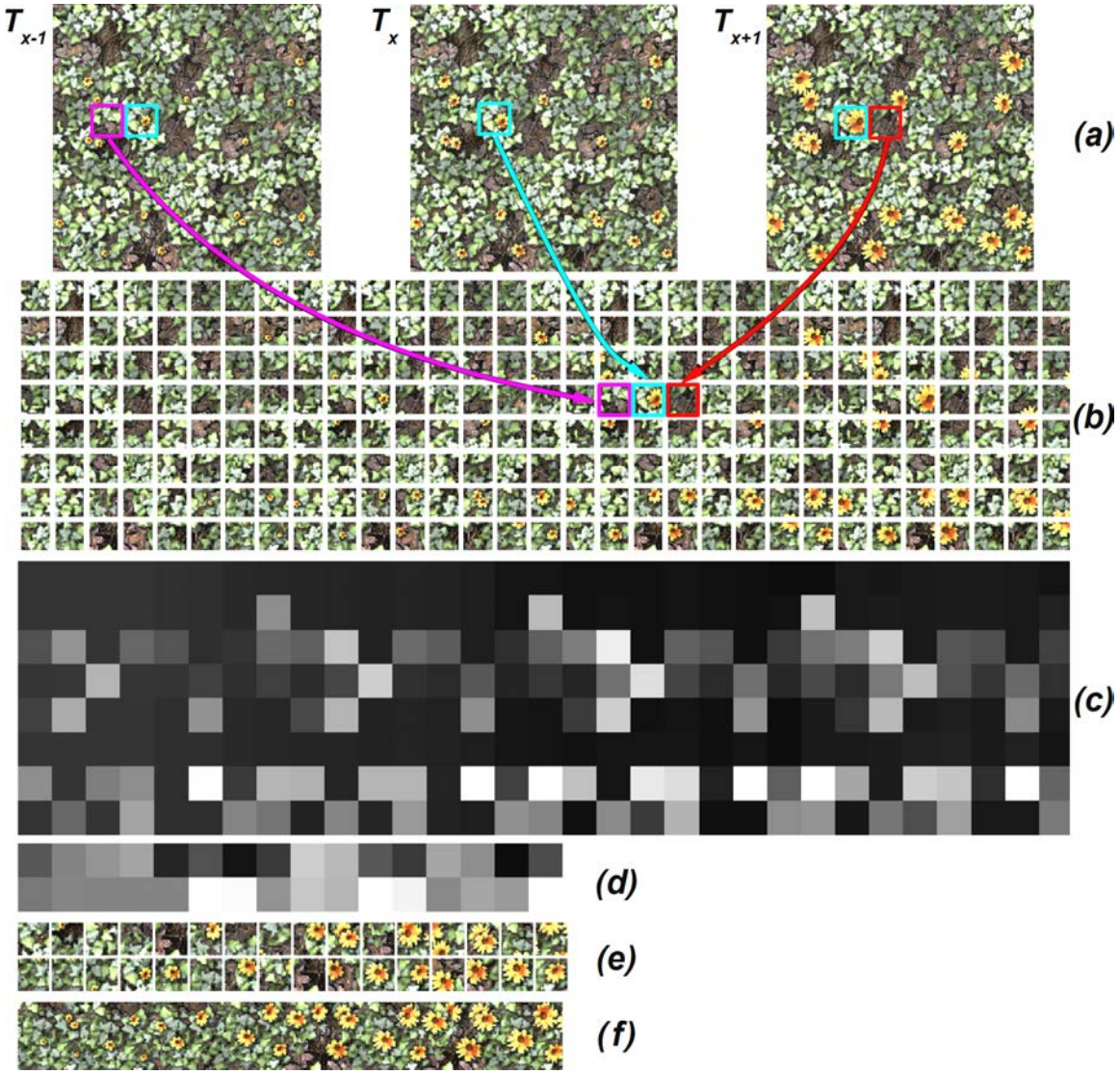

FIGURE C.8 – Création d'une bande visuelle : ***(a)*** Échantillons d'une texture procédurale à partir desquels des patchs sont extraits. ***(b)*** Grille de patchs. ***(c)*** Carte d'importance des patchs. ***(d)*** Carte d'importance réduite. Les pixels clairs correspondent aux patchs à scores élevés. ***(e)*** Les patchs restants. Les fleurs qui varient le plus sont bien représentées. ***(f)*** Synthèse finale.

Annexe D

Synthèse de texture par patch sur GPU

D.1 Introduction

Dans l'annexe précédente, nous avons utilisé la synthèse de textures pour construire les résumés finaux. Cette synthèse s'est faite à partir de plusieurs exemples (ensemble de patchs). Les résumés synthétisés ont servi comme interface pour explorer les apparences produites par une texture procédurale. En plus d'un résultat de haute qualité, une telle interface nécessite une grande interactivité et donc, une méthode de synthèse de texture rapide.

Les approches de synthèse par pixel sont connues pour être très rapide [WLKT09] et sont souvent facile à implémenter sur le processeur graphique [LH05]. Cependant, la synthèse par pixel ne parvient pas toujours à bien préserver les structures dans la texture. De plus, la synthèse par pixels produit souvent des résultats d'une moindre qualité lorsque la synthèse se fait à partir de plusieurs exemples. Contrairement à la synthèse par pixel, la synthèse par patch préserve facilement les éléments structurés et supporte d'une manière naturelle plusieurs exemples en entré. Cependant, les algorithmes de synthèse par patch nécessitent souvent beaucoup de calculs pour disposer les patchs et les coller ensembles.

Dans cette annexe, nous présentons une méthode de synthèse par patch utilisant des calculs parallèles entièrement implémentés sur le processeur graphique. Ce nouveau synthétiseur repose sur les contributions suivantes :

- Nous proposons un nouvel algorithme rapide, mais approximatif, permettant de trouver les frontières des patchs de manière à éviter toutes discontinuités.

- Nous proposons un nouvel algorithme qui déforme un patch dans le cas où une discontinuité persiste sur sa frontière.

- Nous synthétisons une texture en optimisant plusieurs patchs en parallèle. La procédure de synthèse ajoute itérativement des patchs à l'image synthétisée. La frontière des patchs ajoutés est optimisée de manière à cacher autant que possible les discontinuités produites par d'anciennes itérations. De plus, nous rejetons les patchs ne permettant pas d'améliorer le résultat final.

- La plupart des opérations se basent sur de la programmation dynamique. Nous implémentons la programmation dynamique d'une manière efficace sur processeur graphique.

D.2 Vue d'ensemble

Étant donné une texture exemple E, notre méthode synthétise une texture S visuellement similaire et composée de patchs tirés dans E. La synthèse se fait de manière itérative où à chaque itération, plusieurs patchs sont choisis au hasard dans E puis placé sur S. Au sein d'une itération, pour traiter les patchs en parallèle, les patchs sont placés dans S de manière à ce qu'ils ne se chevauchent pas. Afin de maximiser le nombre de patchs utilisés dans une itération, S est recouvert d'une grille où chaque case contient un patch (Figure D.1). De plus, afin d'éviter tout biais, l'alignement de la grille par rapport à S change aléatoirement d'une itération à une autre.

Pour chaque patch placé dans chaque case, on optimise la frontière de manière à minimiser les discontinuités . Pour optimiser les frontière en utilisant la programmation dynamique, on travail dans l'espace polaire du patch comme illustré dans la Figure D.2.

Après avoir optimisé la frontière d'un patch, nous réduisons encore plus les discontinuités en déformant le patch de manière à aligner les gradients de la texture. Cette déformation se base sur une procédure similaire à celle de l'optimisation de la frontière. La Figure D.3 illustre cette déformation.

Une fois le patch déformé, le patch est effectivement ajouté au résultat s'il améliore la qualité globale de la texture, sinon il est rejeté.

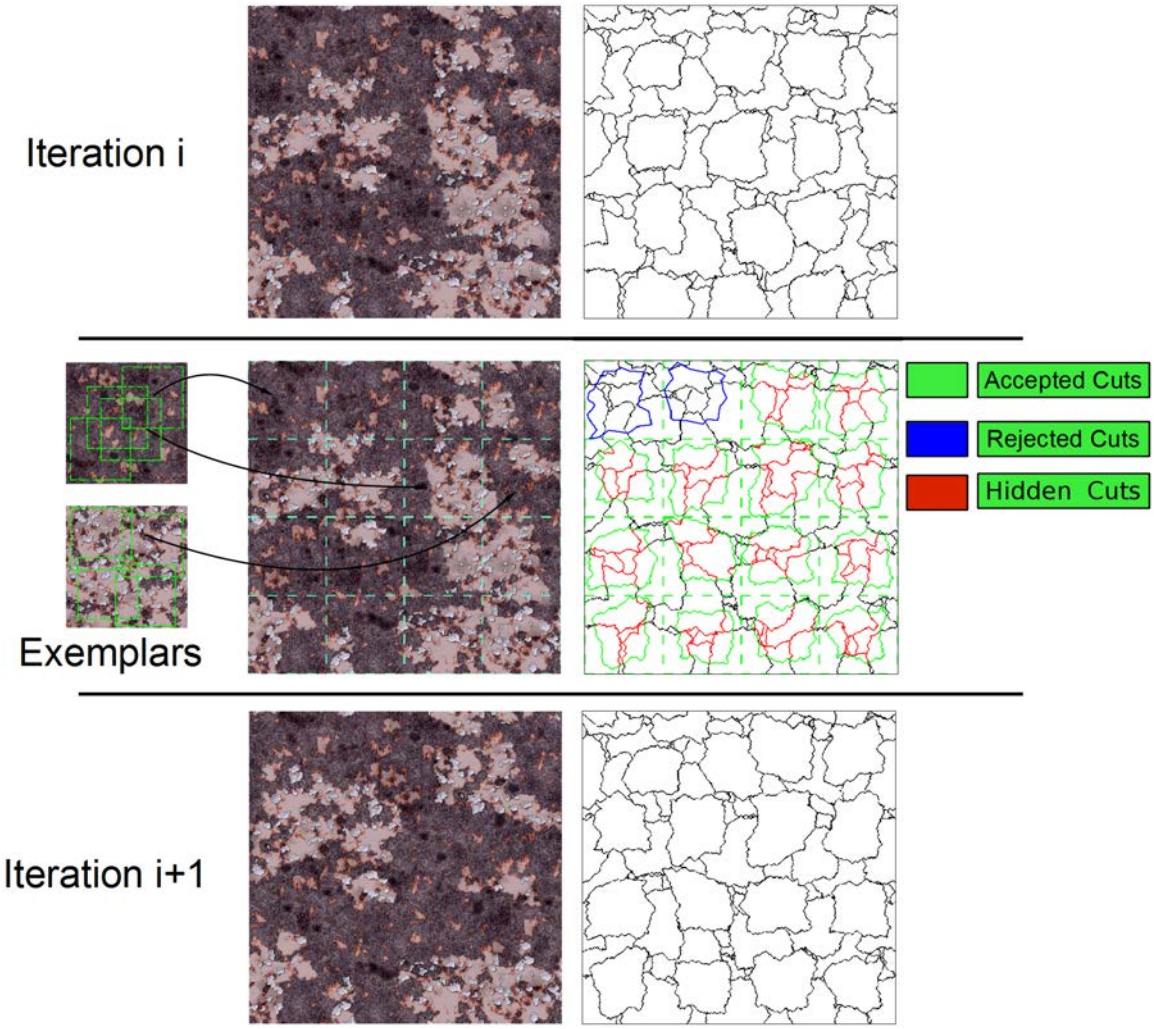

FIGURE D.1 – *Haut :* Texture synthétisée à l'itération i. Les erreurs associées sont affichées en noir sur la carte de droite. *Milieu :* De nouveaux patchs sont ajoutés pour améliorer le résultat. Dans la carte à droite, les erreurs en *rouge* sont cachées par les patchs ajoutés. Les nouveaux patchs ajoutent eux même de nouvelles erreurs affichées en *vert*. Les erreurs en *bleu* sont produites par des patchs n'améliorant pas le résultat. Ces patchs sont rejetés. *Bas :* Texture synthétisée à l'itération $i + 1$.

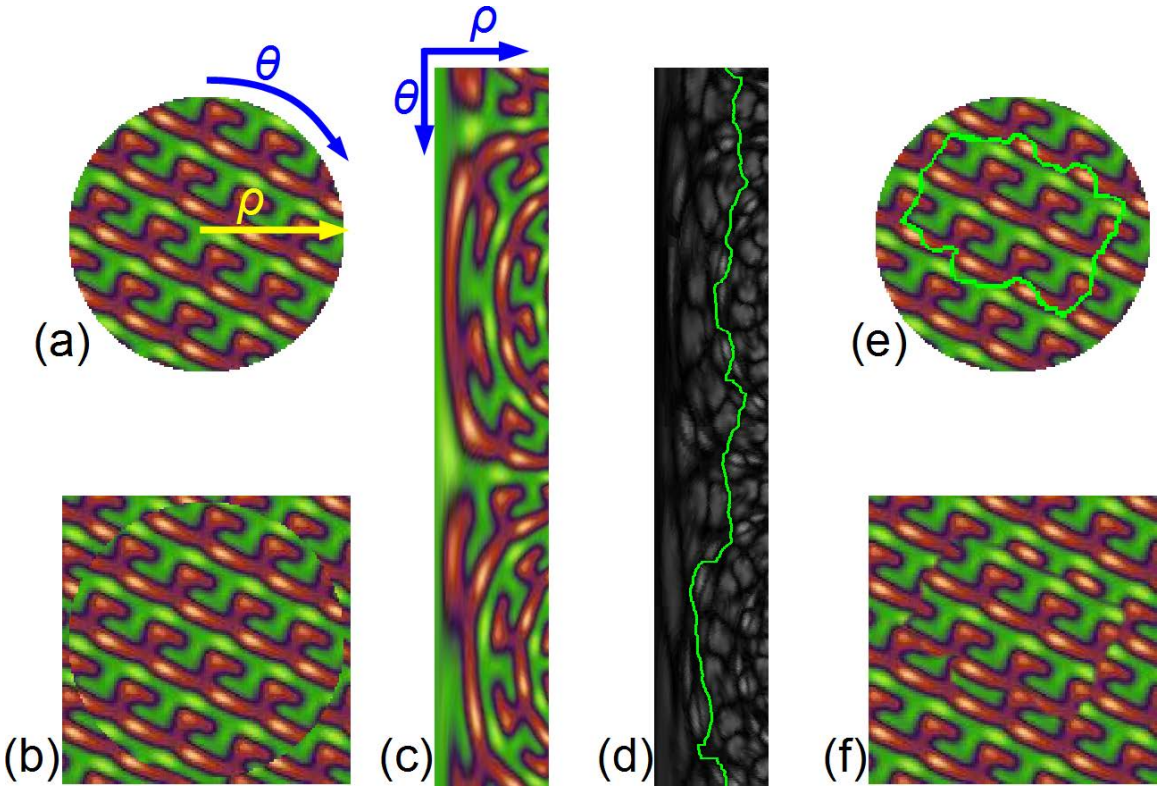

FIGURE D.2 – *(a) :* Un patch $\mathcal{P}$. *(b) :* Placement de $\mathcal{P}$ sur S. *(c) :* Version polaire du patch $\mathcal{P}$. *(d) :* Carte d'erreurs avec la frontière optimale $\mathcal{C}$ en vert. *(e) :* La frontière optimisées dans l'espace cartésien du patch $\mathcal{P}$. *(f) :* Résultat final.

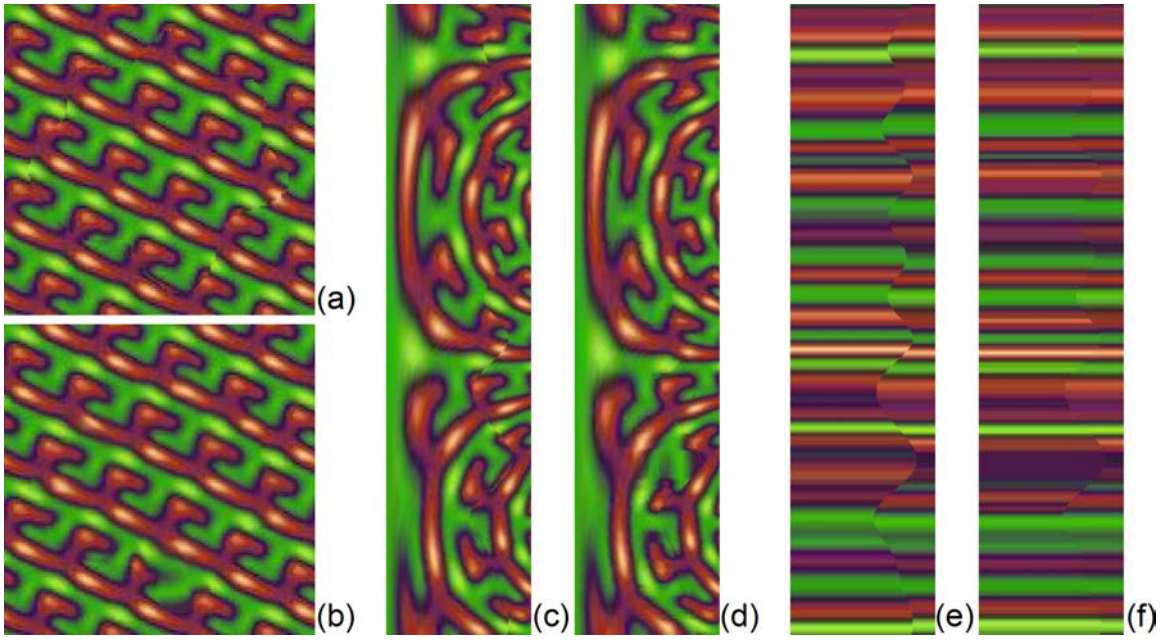

FIGURE D.3 – *(a)* Le patch $\mathcal{P}$ résulte en une discontinuité. *(b)* Résultat final après la déformation . *(c)* Vue polaire du patch avant la déformation. *(d)* Vue polaire du patch après la déformation. *(e)* Couleurs le long de la frontière $\mathcal{C}$ avant la déformation. *(f)* Couleurs le long de la frontière $\mathcal{C}$ après la déformation.

FIGURE E.1 – Trois méthodes différentes utilisées pour adapter les textures à leurs surfaces. *Gauche :* Synthèse de texture. *Milieu :* Etirement de texture. *Droite :* Découpage de textures.

Annexe E

Rendu de textures synthétisées

E.1 Introduction

En production de jeux vidéo, les artistes doivent généralement passer un temps considérable pour créer les textures. Il doivent aussi personnaliser chaque texture pour qu'elle soit adaptée à surface sur laquelle elles sera appliquée. Cette opération manuelle présente deux limitations majeures et ceci surtout lorsque l'environnement du jeu est vaste et varié : Premièrement, beaucoup d'efforts sont nécessaires pour personnaliser chacune des textures. Deuxièmement, le stockage en mémoire de chaque adaptation d'une même texture serait très coûteux.

A cause de cela, les artistes travaillent généralement sur une faible quantité de textures. Ces textures sont ensuite réutilisées sur différentes surfaces sans duplications. Mais étant données que différentes surfaces ont des dimensions différentes, les textures sont souvent étirées, répétées ou encore découpées afin qu'elles soient adaptées sur différentes surfaces. L'étirement résulte en des déformations des éléments composant la texture (Figure E.1 au milieu). Le découpage résulte en des structures découpées dans les textures (Figure E.1 à droite). Enfin, la répétition d'une même texture serait visible lorsque la surface texturée est vue de loin (Figure E.2).

Dans cette annexe, nous utilisons la synthèse de texture pour adapter les textures à leurs surfaces et ainsi éviter les effets indésirables résultant de l'étirement, du découpage

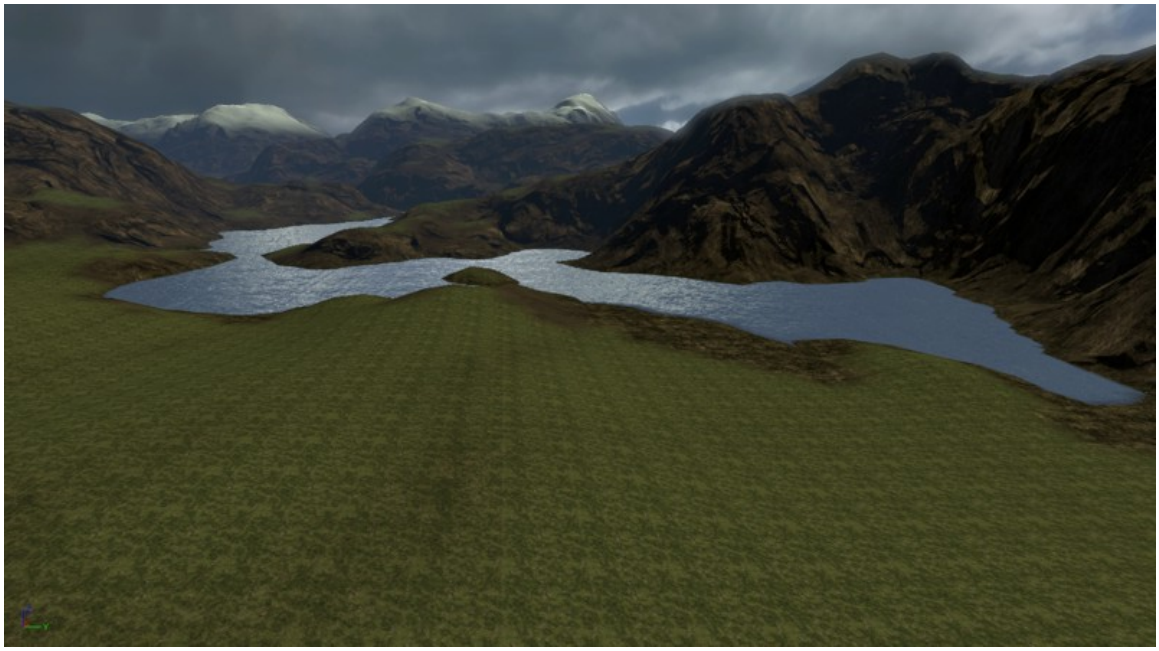

FIGURE E.2 – Répétition d'une texture sur un terrain.

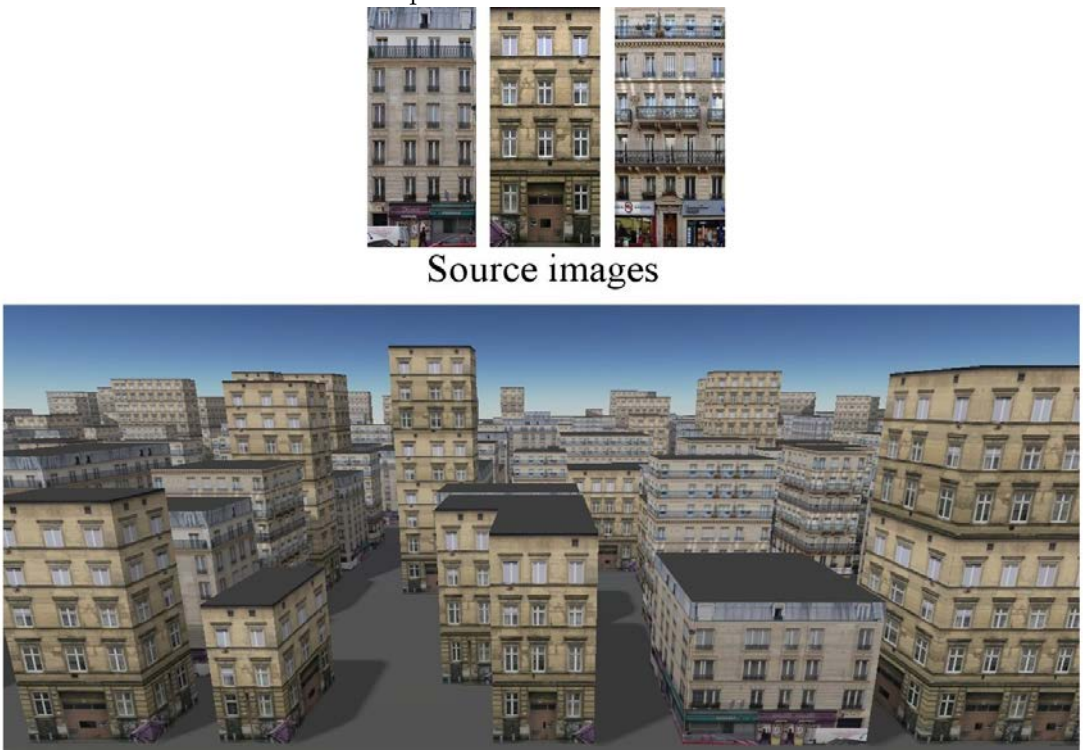

FIGURE E.3 – Chaque façade dans cette large ville possède sa propre texture synthétisée à partir d'un des trois exemples affichés en haut.

ou de la répétition des textures. Cela est particulièrement utile pour les paysages vastes ou les grandes villes comme celle de la Figure E.3. Dans cette figure, chaque façade possède sa propre texture qui est synthétisée à partir d'une texture exemple. Seulement trois textures exemples sont utilisées.

Dans l'annexe précédente, nous avons décrit une méthode rapide de synthèse par patch qui serait appropriée pour générer différentes textures à partir d'un même exemple. Cependant, cette méthode ne permet pas une évaluation par point. Cela signifie que toutes les textures doivent être synthétisées et stockées en mémoire avant le rendu.

Pour surmonter ce problème, nous proposons deux solutions :

Premièrement, nous proposons de modifier le synthétiseur par patch de sorte qu'il génère une représentation compacte du résultat. Cette représentation peut être utilisée directement pendant le rendu sans la nécessité de produire la texture sous forme d'une image bitmap. Le synthétiseur par patch ne parvient cependant pas à synthétiser correctement des structures à grandes échelles. A cause de cela, nous l'utilisons pour synthétiser des textures peu structurées ou stochastiques. Dans le cas des textures structurées, nous utilisons un autre synthétiseur par patch conçu pour gérer ce genre de textures. Pour différencier les deux synthétiseurs, nous utilisons le terme *synthétiseur stochastique* pour décrire le premier synthétiseur et le terme *synthétiseur structuré* pour décrire le deuxième. Pour les deux synthétiseurs, nous représentons le résultat de manière compacte puis nous décodons les pixels pendant le rendu directement à partir de la représentation compacte. L'opération de décodage est rapide, permet un accès par point aux pixels des textures et utilise un filtrage approximatif qui exploite l'accélération matériel pour des performances maximales. La Figure E.4 illustre les opérations d'encodage et de décodage du résultat de synthèse.

Deuxièmement, lorsque le nombre de textures encodées est trop élevé ou lorsque les textures ne peuvent pas être encodées, la mémoire vidéo ne serait probablement pas suffisante pour stocker toutes les textures. Nous proposons dans ce cas un schéma de streaming qui ne synthétise une texture que lorsqu'elle sera visible dans un futur proche. Le principal défi consiste à prédire correctement quelles textures vont devenir visibles et quand démarrer l'opération de synthèse pour qu'une texture soit en mémoire quand elle devient visible.

E.2 Résultats avec le synthétiseur structuré

Le rendu de la ville de la Figure E.5 révèle le principal avantage de notre approche où chaque façade possède sa propre texture. La ville contient 15131 façades (3348 bâtiments). Les textures sont synthétisées à partir de 8 images exemple stockés en 7 Mo. Le stockage du résultat sous forme de textures bitmaps nécessiterait 12.1 Go de mémoire vidéo. Cependant, en utilisant notre méthode d'encodage, les textures synthétisées occupent 92 Mo seulement. Les textures sont décodés pendant le rendu grâce à un *pixel shader*. Le rendu se fait, sur une NVIDIA GeForce GTX580, à 201 images par seconde en moyenne avec une résolution d'écran de 1600×1200. Sans filtrage, le rendu se fait à 317 images par seconde. Quand les textures sont étirées au lieu d'être synthétisées, le rendu se fait à 733 images par seconde mais la qualité est fortement affecté comme le montre la Figure E.6.

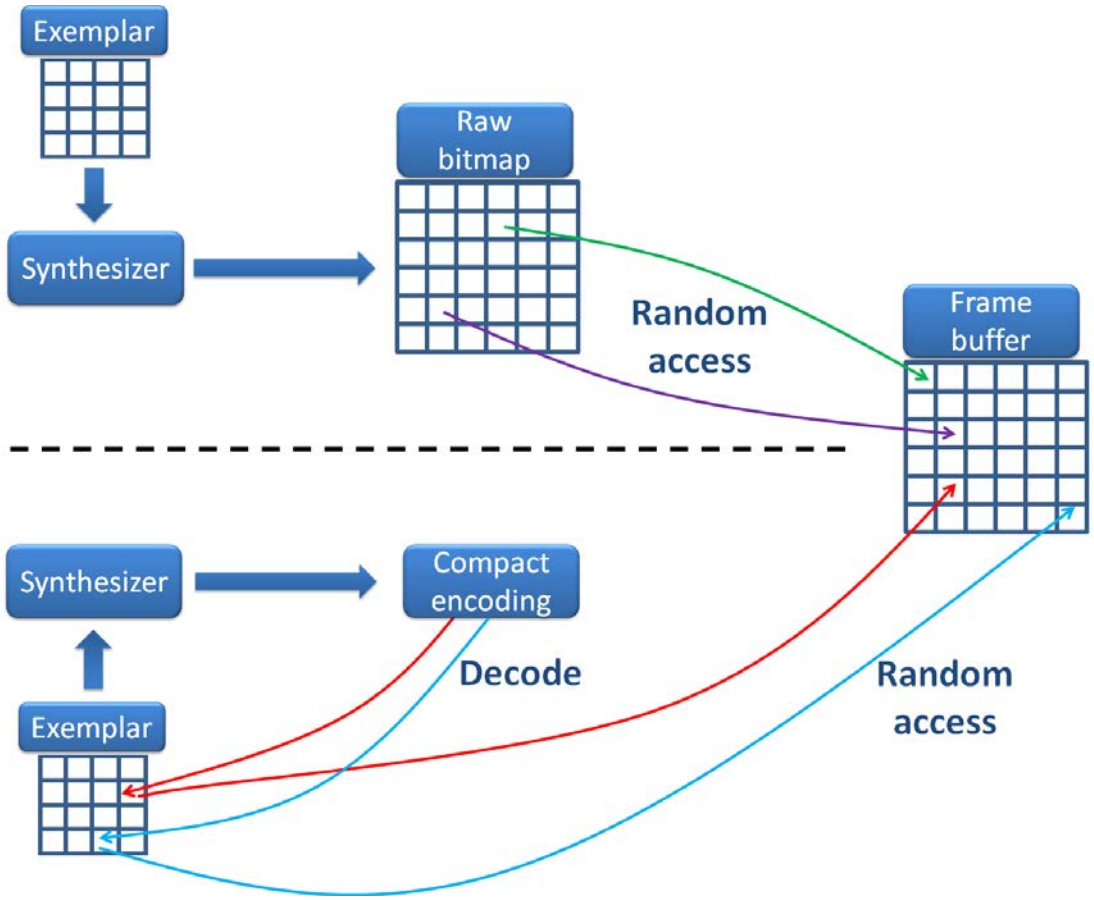

FIGURE E.4 – *Haut :* Le synthétiseur génère une image bitmap stockée entièrement en mémoire. *Bas :* Nous modifions le synthétiseur pour qu'il génère une représentation compacte. Le décodage à partir de cette représentation supporte un accès par point aux données de la texture.

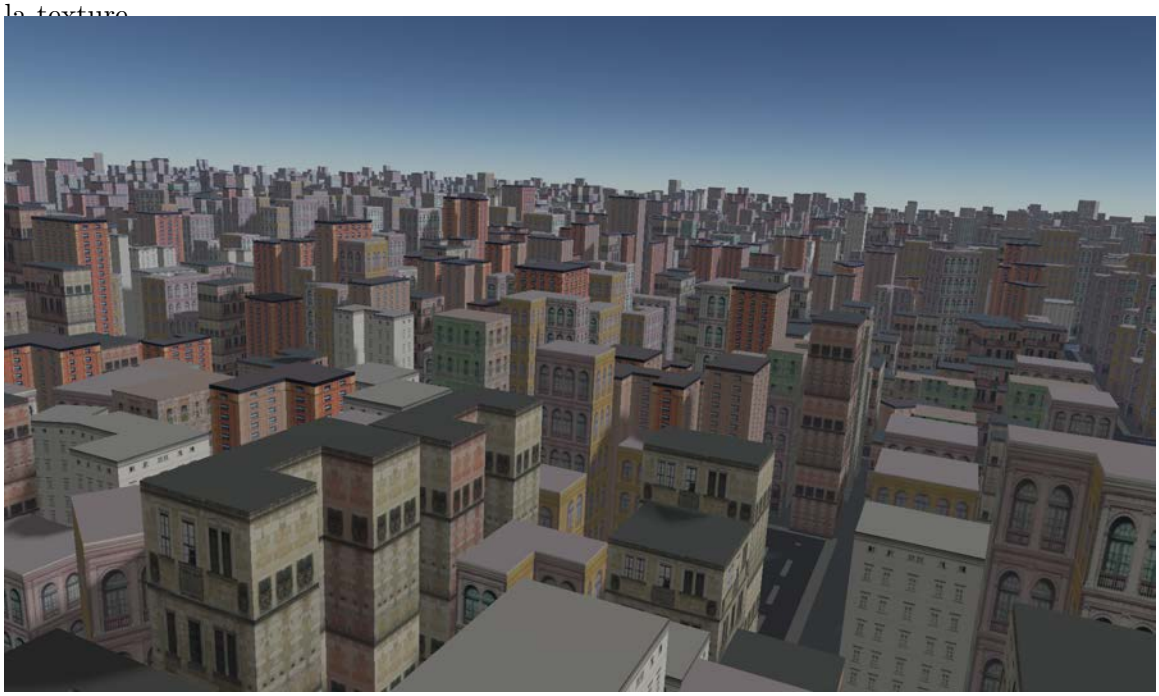

FIGURE E.5 – Une ville contenant une texture unique par façade.

FIGURE E.6 – *Gauche :* Les textures exemples sont étirées pour s'adapter aux façades. *Droite :* Le synthétiseur adapte les textures exemples aux différentes façades.

E.3 Résultats avec le synthétiseur stochastique

Nous utilisons le synthétiseur stochastique pour synthétiser une très large texture sur un grand terrain comme illustré sur la Figure E.7. La texture exemple qui a une taille de 256^2 est stocké en 768 Ko de mémoire vidéo. La texture synthétisée à une résolution de 16384^2 pixels. Elle est synthétisée et encodée en moins d'une minute et le résultat est stocké en 18.96 Mo de mémoire vidéo. Sans aucun encodage, il aurait fallu 768 Mo pour stocker le résultat. Le décodage se fait pendant le rendu à 37 images par seconde avec un *pixel shader* peu optimisé. Quand la texture exemple est répétée au lieu d'être synthétisée, le rendu se fait à 218 images par seconde mais des effets de répétitions sont bien visible comme le montre la Figure E.7.

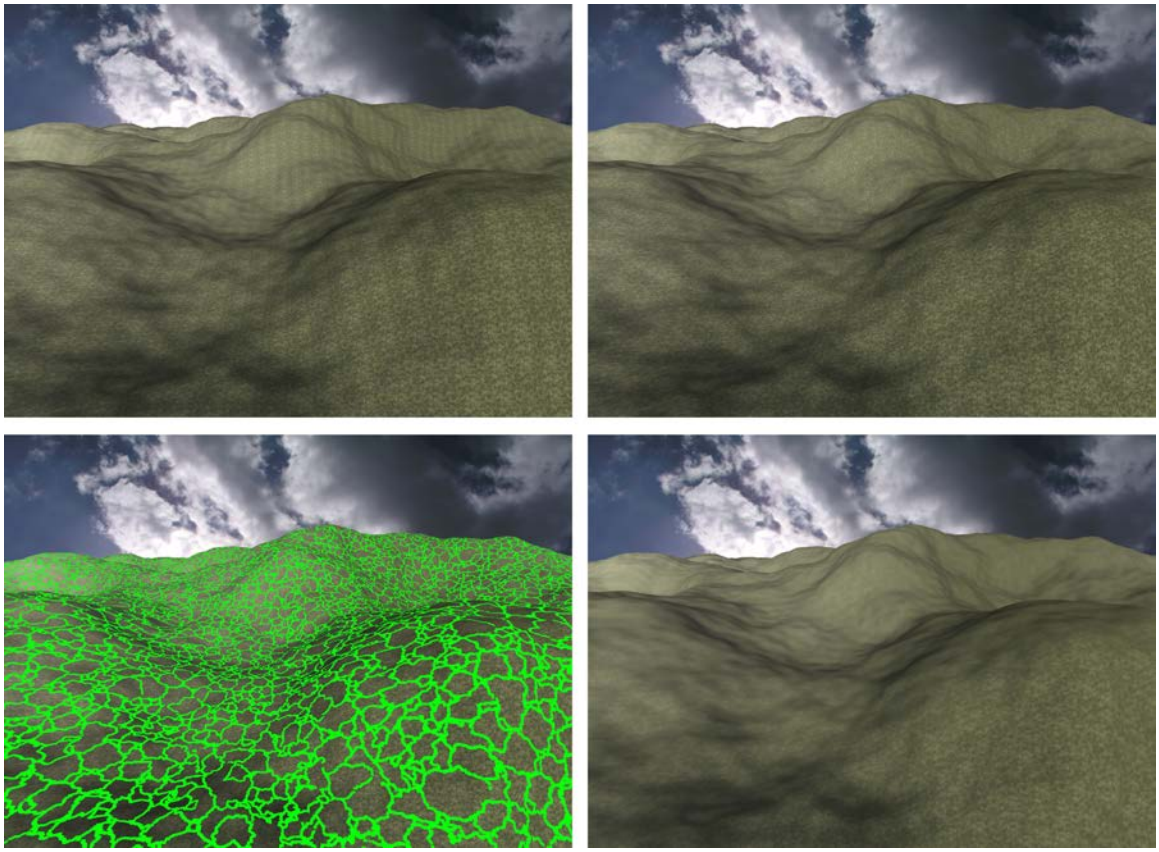

FIGURE E.7 – *En haut à gauche :* La texture exemple est répétée sur la surface du terrain. *En haut à droite :* Le synthétiseur stochastique produit une texture sans répétitions. *En bas à gauche :* Les contours en verts montrent les bords des patchs utilisés pour la synthèse. *En bas à droite :* Le niveau de détail de la texture (LOD) est intentionnellement biaisé pour montrer que le filtrage ne produit pas de discontinuités qui soient visibles.

Résumé

La *synthèse de textures* est une technique qui génère une texture automatiquement grâce à un algorithme. Cette technique permet de réduire le temps de création des textures ainsi que le coût mémoire étant donné que seuls les algorithmes et leurs paramètres ont besoin d'être stockés. Cependant, des difficultés sont souvent rencontrées lors de l'utilisation des textures synthétisées : Tout d'abord, les paramètres des textures synthétisées sont difficiles à manipuler. Ensuite, l'algorithme de synthèse génère souvent les textures sous forme de tableaux de pixels nécessitant beaucoup de mémoire. Nous essayons dans cette thèse de résoudre ces difficultés à travers les contributions suivantes : D'abord, pour améliorer la visualisation de l'espace des textures synthétisées, nous proposons de construire un résumé de cet espace : une seule image statique qui résume, dans un espace limité de pixels, les apparences produites par un synthétiseur donné. De plus, pour améliorer la sélection de paramètres, nous augmentons les curseurs qui contrôlent les paramètres avec des bandes visuelles révélant les changements qui se produisent quand l'utilisateur manipule les curseurs. Pour permettre à l'utilisateur d'interagir de manière interactive avec les résumés visuels, nous nous reposons sur un algorithme de synthèse par patch permettant de générer les textures de façon rapide grâce à une implémentation parallèle sur le processeur graphique. Au lieu de générer le résultat de ce synthétiseur sous forme d'un tableau de pixels, nous représentons le résultat dans une structure compacte et nous utilisons une méthode rapide permettant de lire des pixels directement à partir de cette structure.

Mots-clés : Texture, Synthèse de textures, Texture Procédurale, Rendu en temps réel.

Abstract

Texture synthesis is a technique that algorithmically generates textures at rendering time. The automatic synthesis reduces authoring time and memory requirements since only the algorithm and its parameters need to be stored or transferred. However, two difficulties often arise when using texture synthesis : First, the visualization and parameters selection of synthesized textures are difficult. Second, most synthesizers generate entire textures in a bitmap format. This leads to high memory usage if many textures are synthesized. To address these difficulties we propose the following contributions : First, to improve visualizing the space of synthesized textures we propose the idea of a procedural texture preview : A single static image summarizing in a limited pixel space the appearances produced by a given synthesizer. The main challenge is to ensure that most appearances are visible, are allotted a similar pixel area, and are ordered in a smooth manner throughout the preview. Furthermore, to improve parameters selection we augment sliders controlling parameters with visual previews revealing the changes that will be introduced upon manipulation. Second, to allow user interactions with the procedural texture preview and the sliders previews we rely on a fast patch–based synthesizer. This synthesizer achieves a high degree of parallelism and is implemented entirely on the GPU. Finally, rather than generating the output of the patch–based synthesizer as a bitmap texture, we encode the result in a compact representation and allow to rapidly decoding texels from this representation during rendering.

Keywords : Texturing, Texture synthesis, Procedural texture, Real time rendering.