Synthèse de textures par l'exemple pour les applications interactives

Sylvain Lefebvre

UNIVERSITÉ DE LORRAINE

Ecole Doctorale IAEM

Synthèse de textures par l'exemple pour les applications interactives

Mémoire d'habilitation à diriger des recherches

Sylvain Lefebvre

Soutenue le 30 juin 2014

Composition du jury:

- Kavita Bala, Professeur associé, Université de Cornell, USA *(rapporteur)*
- Mathias Paulin, Professeur, Université de Toulouse, France *(rapporteur)*
- John C. Hart Professeur, Université d'Illinois, USA *(rapporteur)*
- Sylvain Lazard, Directeur de recherche, LORIA *(examinateur)*
- Bruno Lévy, Directeur de recherche, LORIA *(examinateur, parrain HDR)*

Contents

Préambule

Ce document constitue mon mémoire d'Habilitation à Diriger des Recherches. Il contient un récapitulatif de mes travaux de recherche les plus importants, mis en perspective les uns par rapport aux autres. Le document commence par un résumé en Français de mes travaux, puis est rédigé en Anglais.

Foreword

This document was prepared to defend my *Habilitation à Diriger des Recherches*. It provides an overview of my research work of (roughly) the past ten years. The document has been edited from its original version to remove confidential information.

Research is rarely done alone and I would like to thank all my collaborators, in particular Samuel Hornus, Jérome Darbon, Christian Eisenacher, Ares Lagae, Carsten Dachsbacher, Chongyang Ma, Xin Tong, Li-Yi Wei, Ismael García, Matthäus Chajdas, Marcio Cabral, Nicolas Bonneel, Cyrille Damez, Frédéric Claux as well as my PhD students Anass Lasram, Jean Hergel, Jérémie Dumas, and my postdocs Shizhe Zhou and Jonas Martinez.

One of the greatest aspect of being a young researcher is that you get to be advised by some of the brightest people in the field, and I got the chance to collaborate closely with some of them: Fabrice Neyret (my PhD advisor), Hugues Hoppe (postdoc advisor at Microsoft Research), George Drettakis and Bruno Lévy (who leads of my current research team). They all had a deep influence on how I understand our field today.

I take this thesis as an opportunity to provide additional hindsight and context to my work. Rather than duplicating the already published papers – which would be somewhat redundant and not so exciting – I will try to provide the big picture, revealing the strong links between the various projects we undertook. There will be less details here than in each of the publications, since I will instead focus on what I believe, in hindsight, are the most important ideas.

Chapter 1

Résumé des travaux de recherche en Français

1.1 Contexte

Les univers virtuels ont longtemps été réservés aux applications industrielles et scientifiques : Conception assistée par ordinateur, design, architecture, études d'impact, archéologie, simulateurs d'entraînement (aviation, armée, chirurgie), éducation. Cependant, leur essor annoncé auprès du grand public ne s'est pas démenti.

Ces univers sont de plus en plus fréquentés, qu'ils soient à but ludique comme *World of Warcraft* de *Blizzard Entertainment* qui compte plus de 10 millions d'abonnés actifs, ou bien qu'ils proposent des services comme *Google Earth* ou *Virtual Earth*. Ces applications sont également de plus en plus ouvertes aux utilisateurs, qui peuvent non seulement interagir avec le monde représenté, mais également créer de nouveaux contenus : *Google Earth* permet l'envoi de photographies géo-localisées, alors que les mondes tels que *Second Life, Habbo Hotel* ou encore *There* permettent de créer de nouveaux objets et de leur associer des comportements. L'un des enjeux majeurs de ces applications est de pouvoir faire face à la complexité des contenus graphiques : Ils doivent être à la fois variés et détaillés, tout en permettant un affichage rapide sur des ordinateurs grand public. De plus, afin de permettre aux utilisateurs de modifier et créer de nouveaux éléments il faut proposer des outils ayant un fort pouvoir d'expression, mais restant simple à utiliser.

L'objectif de mes recherches est de maîtriser les coûts de création, stockage et affichage des contenus graphiques, tout en permettant leur manipulation par des utilisateurs non experts. Mon approche consiste à considérer l'ensemble de la chaine plutôt que chaque aspect du problème séparément. En particulier, grâce à des méthodes capables de générer automatiquement tout ou partie du contenu, que ce soit à partir d'exemple ou à partir d'une représentation compacte, il devient possible de simplifier la création, de réduire le stockage, et de n'afficher que les parties visibles d'un environnement.

Après mes travaux de thèse, j'ai essentiellement exploré deux thèmes de recherche : la simplification de la chaine de création graphique, et l'accroissement de la qualité et des performances de rendu par une meilleure gestion du contenu. De part la nature transversale de mon approche, mes travaux s'inscrivent souvent dans ces deux thèmes. Je détaille ci-dessous mes principales contributions.

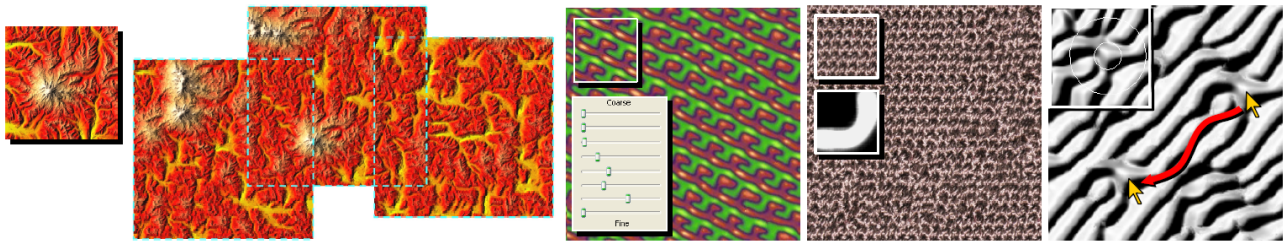


Figure 1.1: *De gauche à droite: Une carte de hauteur synthétisée au fur et à mesure des déplacements du point de vue, l'interface de contrôle des variations dans la texture, contrôle localisé de l'aspect et interface de 'copier-coller'. Ces images sont issue de notre article 'Parallel Controlable Texture Synthesis' [LH05]*

Figure 1.2: *Différents objets texturés par notre méthode procédurale 'Gabor Noise' [2]. Ces textures ne sont pas stockées sous forme d'images mais calculées à la volée, par notre algorithme, sur processeur graphique.*

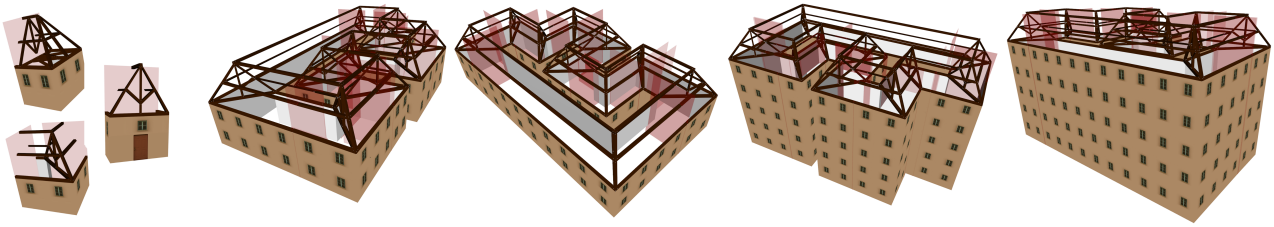


Figure 1.3: *Différents bâtiments construits à partir de trois briques de base. Les briques de base sont déformées (géométrie et texture) de manière à créer un ensemble consistant. Résultat de notre article [CLDD09].*

1.2 Synthèse de textures à partir d'exemples pour les applications interactives

Ma principale contribution au domaine de la synthèse d'images est constituée par mes travaux sur la synthèse de textures à partir d'exemples, notamment dans le contexte des applications interactives et des processeurs graphiques parallèles (GPU). Les textures sont des images appliquées aux objets 3D et leur donnant un aspect de surface. L'originalité de mon approche consiste à générer les textures pendant le cours de l'application, plutôt que de les préparer à l'avance et de les stocker en mémoire.

J'ai débuté cet axe de recherche durant mon post-doctorat. Ces travaux, illustrés Figure 1.1, ont débouché sur deux publications dans la revue TOG en 2005 [LH05] et 2006 [LH06a] (actes de la conférence SIGGRAPH), ainsi que sur deux publications au journal CGF (Computer Graphics Forum) en 2008 [ELS08, DLTD08] (actes des conférences Eurographics et Rendering Symposium). Les deux principales publications [LH05, LH06a] ont été citées respectivement 200 et 170 fois à ce jour (source : Google scholar, 28/05/2013, arrondi dizaines inférieures). Nous avons récemment étendu ces approches à des phénomènes animés. Ce travail a été présenté à la conférence SIGGRAPH 2013 et publié dans le journal Transactions on Graphics [MWLT13].

J'ai poursuivis ces travaux dans plusieurs directions. En premier lieu, la génération de motifs aléatoires dont le contenu fréquentiel est précisément contrôlé – ceci est utilisé comme primitive de base par les infographistes pour construire des textures, qui sont alors calculées plutôt que stockées sous forme d'images, comme illustré Figure 1.2. Ces recherches ont été publiées en 2009 dans le journal TOG (Transactions on Graphics) [LLDD09] (actes de la conférence SIGGRAPH), et le logiciel est déposé à l'APP (Agence de Protection des Programmes) en vu de transferts industriels (nous avons eu plusieurs contacts avec Pixar, Weta Digital, Luxology). Nous avons proposé une seconde approche permettant de déterminer les paramètres du modèle de bruit procédural à partir d'une image donnée en exemple [GLLD12].

Une seconde direction de recherche concerne la manipulation interactive conjointe de géométrie et texture, illustrée Figure 1.3. Ces travaux ont donné lieu à une publication en 2009 dans le journal Computer Graphics Forum [CLDD09] (actes de la conférence Eurographics) et une publication à la conférence internationale I3D

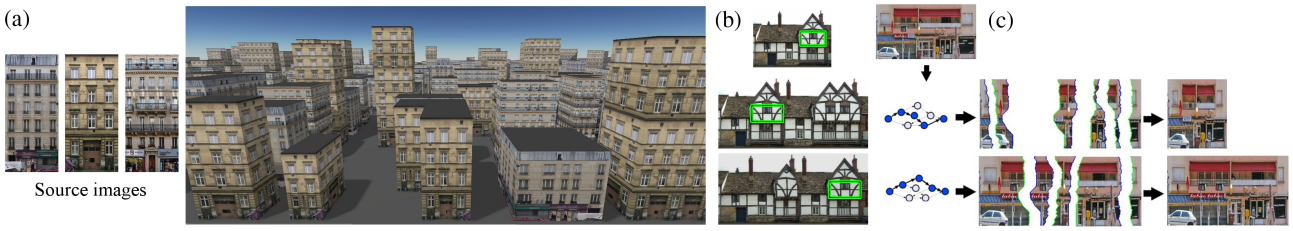


Figure 1.4: *De gauche à droite: (a) Trois images sources utilisées pour produire automatiquement toutes les façades d'une ville. (b) Contrôle par glisser-déplacé interactif du résultat. (c) La synthèse est formalisée sous la forme d'une recherche de plus court chemin dans le graphe décrivant l'espace des images synthétisables. Images de l'article [LHL10].*

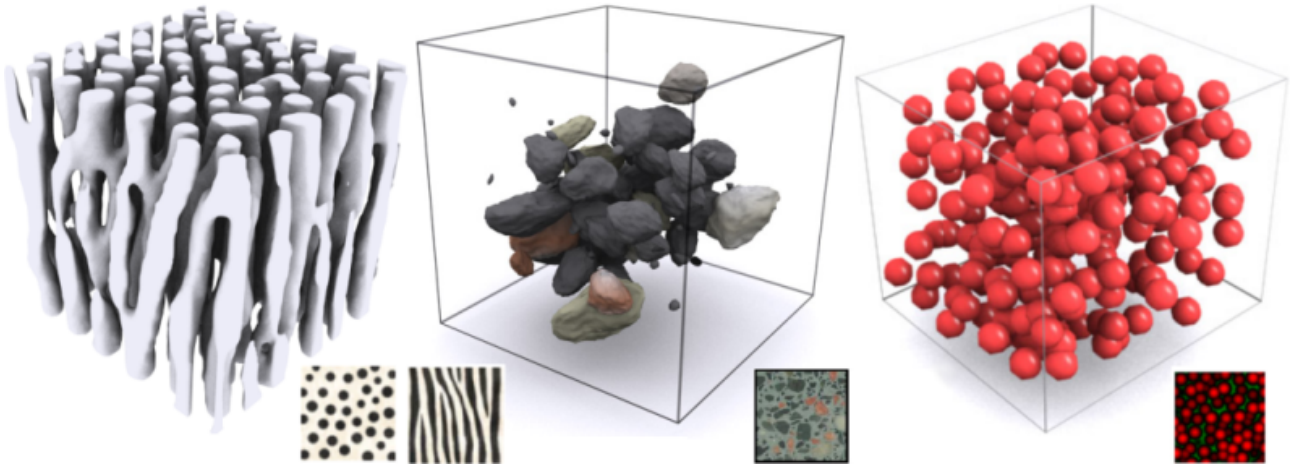


Figure 1.5: *De gauche à droite: Trois volumes synthétisés automatiquement à partir des images exemples situées en dessous.*

en 2010 [CLS10]. Nous avons récemment proposé d'assembler automatiquement des environnements complets à partir des éléments de base manipulés par notre méthode de synthèse, en suivant des spécifications de haut niveau fournies par un designer [MVLS14].

Ces travaux sur la manipulation conjointe de géométrie et texture ont inspiré mon approche de la synthèse de textures fortement *structurées* (façades, portes, fenêtres, panneaux, éléments architecturaux), travaux publiés dans la revue internationale TOG [LHL10] (actes de la conférence SIGGRAPH en 2010). Cette nouvelle approche, illustrée Figure 1.4, permet de synthétiser des textures structurées qui sont impossibles à synthétiser correctement avec les algorithmes de synthèse existants. La particularité de notre approche, qui illustre bien ma méthodologie, est de stocker les résultats sous forme très compacte. Cette représentation compacte est utilisée directement pour peindre les couleurs sur la surface des objets et n'est donc jamais décompressée en mémoire. Ces travaux ont été effectués dans le cadre de l'ANR SIMILAR-CITIES (voir ci-après). En 2013, nous avons appliqué une méthodologie similaire pour syntétiser des motifs structurés le long de courbes. Ces travaux ont été publiés dans le journal Computer Graphics Forum [ZLL13].

La troisième direction j'ai explorée, illustrée Figure 1.5, concerne la génération de textures en volume. Ces travaux sont aussi emblématiques de mon approche transversale. L'idée de la synthèse volumique est de générer un volume de couleurs à partir d'une image : Toute tranche extraite du volume ressemble à l'image. Ces volumes sont ensuite utilisés pour donner une apparence à un objet. Contrairement aux approches existantes, notre algorithme ne génère que la sous partie du volume effectivement utile pour appliquer une couleur sur la surface. Il s'agir d'une avancée significative car la taille du problème est grandement réduite : La complexité de notre algorithme ne dépend que de l'aire de la surface à texturer. Notre algorithme est de plus parallèle et implémenté sur les processeurs graphiques de dernière génération. Ceci nous permet de générer les textures à la volée lorsque de nouvelles surfaces apparaissent, par exemple si un objet se brise. Afin d'appliquer la texture sur les objets dont la forme évolue au cours du temps, nous utilisons le TileTree, une méthode que nous avons publiée en 2007 [LD07] (voir ci-dessous). Ces travaux sont le résultat d'une collaboration avec Microsoft Research Asia et ont été publiés à la conférence EGSR en 2008 [DLTD08]. Ce travail met bien en évidence la dualité de mon approche : D'une part la méthode simplifie la création de contenu (ie. générer

un volume de couleurs à partir d'une image) ; d'autre part elle permet de ne générer que les données utiles pour l'affichage de la scène.

1.3 Application de textures sans paramétrisation planaire

Ma seconde contribution importante en informatique graphique concerne l'application de textures sur des modèles 3D sans avoir recours à une paramétrisation planaire. J'ai très tôt contribué dans ce domaine tout d'abord en 2004 avec un chapitre dans le livre GPUGems2 [LHN05a], édité par la société NVidia et destiné à diffuser vers l'industrie les dernières avancées algorithmiques autour des processeurs graphiques. Ce chapitre décrit l'implémentation d'une structure hiérarchique sur processeur graphique. Ceci a été suivi d'une publication à la conférence I3D en 2005 [LHN05b], exploitant cette hiérarchie pour positionner des éléments de texture en volume autour d'un objet. Nous avons ensuite publié en 2006 dans le journal TOG (actes de la conférence SIGGRAPH) un travail sur du hachage spatial parfait [LH06b], qui est aussi l'objet d'un brevet (US Patent 11405953). Nous avons poursuivi cette direction de recherche avec des publications en 2011 [GLHL11] et 2013 [LHL14]. Nous avons enfin proposé une nouvelle méthode pour les textures sans paramétrisation, à la fois plus simple et plus compacte en mémoire, publiée en 2007 à la conférence I3D [LD07]. Nous avons proposé plusieurs améliorations dans un chapitre du livre ShaderX6 [DL08], à destination de l'industrie.

Chapter 2

Research activities: overview

2.1 Context and overview

> *Hiro is approaching the Street. It is the Broadway, the Champs Elysees of the Metaverse. [...] It does not really exist. But right now, millions of people are walking up and down it.* Neal Stephenson, Snow Crash

Millions of individuals explore virtual worlds every day, for entertainment, training, or to plan business trips and vacations. Video games such as *Eve Online*, *World of Warcraft*, and many others popularized their existence. Sand boxes such as *Minecraft* and *Second Life* illustrated how they can serve as a media, letting people create, share and even sell their virtual productions. Navigation and exploration softwares such as *Google Earth* and *Virtual Earth* let us explore a virtual version of the real world, and let us enrich it with information shared between the millions of users using these services every day.

Virtual environments are massive, dynamic 3D scenes, that are explored and manipulated interactively by thousands of users simultaneously. Many challenges have to be solved to achieve these goals. Among those lies the key question of content management. How can we create enough detailed graphical content so as to represent an immersive, convincing and coherent world? Even if we can produce this data, how can we then store the terra–bytes it represents, and transfer it for display to each individual users?

Rich virtual environments require a massive amount of varied graphical content, so as to represent an immersive, convincing and coherent world. Creating this content is extremely time consuming for computer artists and requires a specific set of technical skills. Capturing the data from the real world can simplify this task but then requires a large quantity of storage, expensive hardware and long capture campaigns. While this is acceptable for important landmarks (e.g. the statue of Liberty in New York, the Eiffel tower in Paris) this is wasteful on generic or anonymous landscapes. In addition, in many cases capture is not an option, either because an imaginary scenery is required or because the scene to be represented no longer exists. Therefore, researchers have proposed methods to *generate* new content programmmatically, using captured data as an example. Typically, building blocks are extracted from the example content and re–assembled to form new assets. Such approaches have been at the center of my research for the past ten years.

However, algorithms for generating data programmatically only partially address the content management challenge: the algorithm generates content as a (slow) pre–process and its output has to be stored for later use. On the contrary, I have focused on proposing models and algorithms which can produce graphical content while minimizing storage. The content is either generated *when it is needed* for the current viewpoint, or is *produced under a very compact form* that can be later used for rendering. Thanks to such approaches developers gain time during content creation, but this also simplifies the distribution of the content by reducing the required data bandwidth.

In addition to the core problem of content synthesis, my approaches required the development of new data–structures able to store sparse data generated during display, while enabling an efficient access. These data–structures are specialized for the massive parallelism of graphics processors. I contributed early in this domain and kept a constant focus on this area.

The originality of my approach has thus been to consider **simultaneously** the problems of **generating, storing and displaying** the graphical content. As we shall see, each of these area involve different theoretical and

technical backgrounds, that nicely complement each other in providing elegant solutions to content generation, management and display.

2.2 Contributions

Interactive rendering requires a very fast access to the graphical content. For instance, colors synthesized along the surface of an object are accessed millions of times during the rendering of the surface on screen. To achieve such speed, it is usually assumed that the data (the color points) is static: it does not change over the course of the application. This affords for a number of pre-computations to store the data in a way that facilitates rendering. However, when this data is generated on–the–fly, during the course of the application, these assumptions no longer hold. Therefore, in addition to the problem of content synthesis, we needed to answer the problem of storing and accessing the produced data. Since rendering is performed on data–parallel Graphics Processing Units (GPUs), we designed our synthesizers, data–structures and algorithms for these platforms. This adds strong requirements on the algorithms and data layouts.

My research therefore resulted in two lines of work: on the one hand algorithms to synthesize graphical content; on the other hand algorithms to store and render the data generated by these algorithms. The first line of work is more theoretical, while the second involves designing parallel algorithms and data–structures for graphics processors.

These two lines of research are reflected in the organization of this document: Chapter 3 introduces my contributions in texture synthesis, while Chapter 4 introduces my work on Computer Graphics data–structures for parallel processors. I present my research program for the coming years in a separate document.

In this document I focus only on a selected number of publications. My full list of publications is available on my website [1].

[1] http://webloria.loria.fr/~slefebvr/

Chapter 3

Runtime texture synthesis

> *There is a moment in every dawn when light floats, there is the possibility of magic. Creation holds its breath.* Douglas Adams, Life, the Universe and everything

Texture synthesis is the process by which an algorithm generates an arbitrarily large image representing a particular appearance, from little input data. It is a fascinating process, as complete landscapes and even planets can be produced by such algorithms [EMP+94].

Texture synthesis is a wide research area, involving different fields. Mathematicians have proposed the first statistical models for characterizing a texture. In Computer Vision, textures provide visual cues to understand shapes or detect defects in manufacturing processes. In Computer Graphics, we seek to generate automatically high quality, detailed images of a large variety of materials, thus alleviating the task of manually acquiring and painting this data.

Runtime texture synthesis adds a new set of challenges: not only do we seek to synthesize a texture from an exemplar, but we aim at generating these textures as the user explores the scene. This avoids having to store in memory more content than is actually displayed. Whenever textures cover a very large spatial extent, such as the ones used on landscapes, we would like to only synthesize the visible subpart.

The following sections discuss our main contributions to the field of by–example texture synthesis. Rather than repeating the content of our papers, I try to cast a new view on the algorithms we proposed as well as provide hindsights on these approaches.

However, before going any further a better understanding of what is meant by *a texture* is required.

3.1 What are textures?

> ***Texture*** *Noun: The feel, appearance, or consistency of a surface or a substance.* Google dictionary
>
> ***Indefinite*** *Adj.: Not clearly expressed or defined; vague.* Google dictionary

The term *texture* takes different meanings in Computer Graphics. Most often, it refers to an image which is mapped onto an object, so as to define details along an otherwise smooth surface. This operation, named *texture mapping*, is intuitively similar to wrapping an object in a decorative paper. It is responsible for most of the complexity perceived in computer generated images, as illustrated Figure 3.1. We introduce important notions related to texture mapping in Section 3.5.1.1.

The term *texture* also designates a range of appearances, typically images of *homogeneous materials* having similar visual characteristics in the plane. Not all images elect as a texture in that sense. For instance, the photograph of a human face, a landscape, or a door frame is not referred to as a texture. Figure 3.2 shows examples of textures in the sense of homogeneous materials. Often, textures used for Computer Graphics are *toroidal*, that is their left/right and top/bottom border match seamlessly. This lets the image be repeated through a simple tiling, thus giving the illusion of a larger amount of the same material. This simple approach – immensely popular in practice – unfortunately gives poor results.

In this document, we essentially use the term *texture* in the sense of a material, and will refer to a *texture map* for images mapped onto a surface.

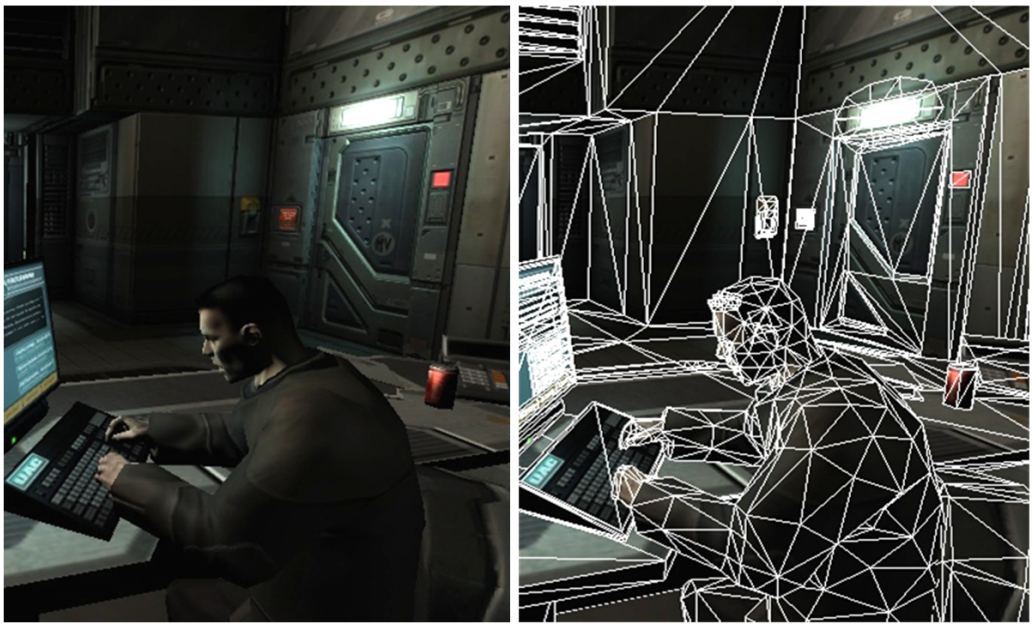


Figure 3.1: Left: *A very detailed scene from the video game Doom 3 (© Id Software).* Right: *The wireframe view reveals the low number of polygons. Textures are in fact responsible for most of the details in the scene.*

In the following we introduce the most common types of textures, as well as tools to analyze them. We will then introduce key notions of texture mapping which are important to understand constraints unique to texture synthesis for Computer Graphics.

3.1.1 Main texture types

There is no widely accepted classification of textures. However, researchers often refer to a number of categories. These do not have clearly defined boundaries and sometimes overlap, but are nonetheless useful to designate ranges of textures. Please note that the terminology below is my own, and that many other categories are possible depending of the properties of interest. The purpose of this classification is simply to give insights about the main types of textures and their properties.

It is also very important to keep in mind that a texture may belong to different categories depending at which *scale* it is observed. Some textures are structured at close inspection but seem random when seen from afar.

Stochastic–unstructured Stochastic textures are the most widely studied. They typically exhibit a random aspect with no edges or contours, as illustrated Figure 3.2. Stochastic textures fall into two main categories: isotropic and anisotropic (see Figure 3.6). Isotropic textures have no preferred orientation, that is, rotating them does not change their appearance. On the contrary, anisotropic textures exhibit an orientation.

Stochastic textures are of particular interest in Computer Graphics: on the one hand they occur often in nature and are difficult to paint and to compress – due to their intrinsic randomness. On the other hand, they seem easier to apprehend in particular thanks to the understanding of random processes developed in mathematics and physics.

An important property of these textures is their *locality* and *stationarity*. Locality implies that the value of the texture at a given point only depends on the values within a limited size neighborhood around it. Stationarity implies that this dependency between a pixel and its neighborhood is overall the same throughout the image. These properties can be easily understood: let us consider an image of many tiny pebbles. The color of an individual pixel is determined by the pebble to which it belongs. This information is local and independent from the surrounding pebbles. Now, because there is a very large number of pebbles their variations are uniformly distributed everywhere the texture has a similar appearance in different places. These properties led researchers to model textures as Markov Random Fields (MRF) [PL95]. This model relies on a set of sites – pixels in our case – depending on each others through only local relationships – a neighborhood. The key step in defining a MRF is to define the probability of a given value at a site, knowing the values at its neighbors. We will further discuss this model in

Figure 3.2: *Typical examples of stochastic–unstructured textures. Textures from cgtextures.com*

Section 3.3.

It has also been conjectured that stochastic textures are solely determined by their power spectra – or equivalently second order statistics [Jul62]. Julesz gave several examples of textures with identical power spectra and showed that subjects could not discriminate them under brief inspection. However, he also proposed counter examples in subsequent works [JGSF73]. This is nevertheless a useful observation in practice which led to a variety of synthesis methods: summation of noise bands [Per85], histogram matching at different scales [HB95, LVLD10], Wavelet coefficients matching [DB97, BJEYLW01], matching of several statistical constraints [PS99], and phase randomization [GLLD12].

Stochastic–structured Stochastic–structured textures exhibit an overall stochastic aspect, but have some structures at finer scales. A typical example is a stone wall: the individual stones are structured elements, but all together they form a stochastic pattern. In general, this category contains all textures having an overall random aspect, but containing well defined contours and edges. These textures are typically stochastic when considered at a coarser scale. Figure 3.3 gives several examples.

Depending on the texture content, the MRF assumptions may still hold, and some of the techniques mentioned above may give reasonable results. However, it is often necessary to guide the synthesis process so as to preserve contours and shapes present in the textures [WY04] [LH06a].

The approaches giving the best results for synthesizing stochastic–structured textures are so–called patch based methods [EF01, KSE$^+$03]. Contrary to what is done for stochastic textures, these approaches do not try to obtain a statistical model of the texture content. Instead they seek to re–organize the content, similarly to a jigsaw puzzle. Pieces of the original image are put together, one by one, to form a larger texture. The frontiers between the pieces is optimized so as to minimize visible seams. This is done by hiding the color differences through an optimal cut [KSE$^+$03] followed by Poisson stitching [PGB03], and sometimes by deforming the patches [WY04], [LL12].

Another set of particularly successful methods are those based on particles [PFH00, DMLG02]. The

Figure 3.3: *Typical examples of stochastic–structured textures. Textures from cgtextures.com*

texture is modeled as a set of texture elements, called *textons*, which are put together to form the texture. These approaches first extract the textons – often manually. Textons are then reordered so as to synthesize a new texture with a similar aspect. The main difficulty with these approaches is to define what a good texton is. However they have proven very effective on stochastic–structured textures.

Stochastic–structured texture may also be modeled with noise. For instance, [Wor96] proposes a model based on generalized Voronoi diagrams to capture the structured but irregular aspect of these textures.

Structured–organized Structured–organized textures exhibit structure as well as a higher level organization. Many decorative patterns and ornaments fall into this category, as illustrated Figure 3.4.

Figure 3.4: *Typical examples of structured–organized textures. Textures from cgtextures.com*

Near–regular and regular Near–regular and regular textures are in fact a subset of the previous category where the higher level organization exhibits some degree of periodicity. As illustrated Figure 3.5 typical examples are brickwalls, textiles, or regular architectural patterns. Some techniques specialize in detecting and synthesizing such patterns [LLH04]. Interestingly, many approaches can synthesize this category while they cannot address structured–organized textures in general: the periodicity can be exploited to guide the synthesis process, injecting prior knowledge into the texture model.

Figure 3.5: *Typical examples of near–regular textures. Textures from cgtextures.com*

3.1.2 Characterizing textures

In this Section I introduce some of the most common tools to analyze and characterize textures: histograms and histograms of co-occurences (Section 3.1.2.1), power spectrum and auto–correlation (Section 3.1.2.2), and finally tools to analyze the color space (Section 3.1.2.3). This is far from exhaustive, but this set of fundamental tools is the core toolbox for anyone interested in texture analysis and synthesis.

Many synthesis schemes seek to produce new images under the constraint that they must have the same characteristic as a given example (for instance, the same distribution of colors). The minimal set of characteristics *completely* characterizing a texture has yet to be discovered [PS99]. The very existence of such a characterization remains an open question and is tightly linked to the precise definition of what textures are.

3.1.2.1 Histograms

One of the most universally used tool for basic image analysis is to consider the histogram of pixel values. On colored images, this is often done on the luminance only – the eye being more sensitive to changes in luminance – or separately for each color channel. The histogram is a *first–order* statistic: it captures the values of pixel independently from each others.

The synthesis scheme of Eager and Bergen [HB95] synthesizes textures by matching their luminance histograms at all scales, from coarse to fine. This produces convincing results for highly stochastic textures, but fails as soon as edges or any sort of identifiable texture elements are present. This scheme was later extended to runtime synthesis by Lagae *et al.* [LVLD10].

Interestingly, Kopf *et al.* [KFCO$^+$07] note that the histogram is often not well preserved by neighborhood synthesis schemes (Markov Random Fields), and include an additional histogram matching term for synthesis.

Multi–dimensional histograms and co–occurences Histograms can be generalized to multiple dimensions, counting how many times specific vector of values appear. Multi–dimensional histograms can for instance be applied onto the RGB color space. They have also been used early in texture synthesis to determine the likelihood of a pixel taking a value knowing the values of its neighbors [PP93, PL95, EF01]. However, these histograms tend to be very sparse when built from an example image (see Section 3.3.1): as the number of dimension increases, even a large image contains few individual examples of specific color neighborhoods. Exploiting such histograms usually requires additional assumptions to interpolate between the observed values.

Histogram of co–occurences count how many times specific pixel values appear together within a neighborhood. For instance, this implies counting the number of times a pixel of value 127 was observed at the top left of a pixel of value 13, independently of the other neighbors. Such histograms have been used for the purpose of texture synthesis [QY05].

3.1.2.2 Power spectrum

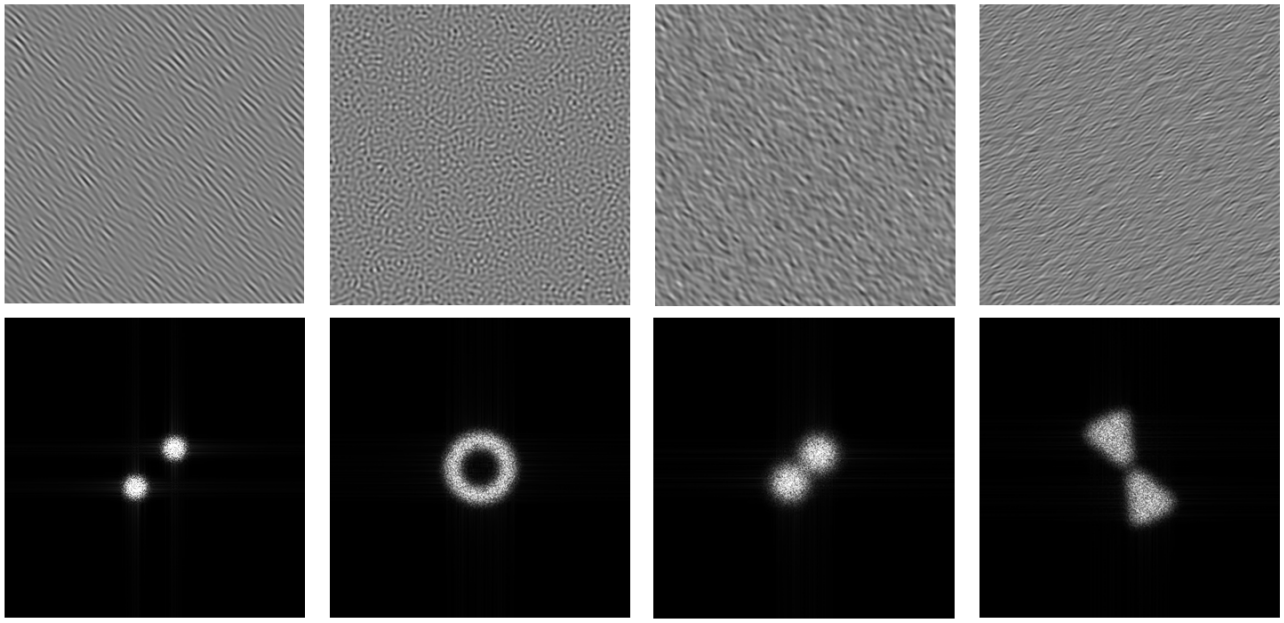
The power spectrum of a texture is a central tool for texture analysis and synthesis. It is defined as the squared modulus of the Fourier transform of the texture. It is best computed using a fast Fourier transform library. The power spectrum captures second–order statistics – that is, the relationship between pairs of pixels throughout the image (intuition on this is given below).

The power spectrum of an image is a 2D array of values. Each entry corresponds to a sine wave in the image, whose frequency and orientation depends on the position of the entry within the power spectrum. The value stored in each entry is the *energy* that the signal contains for this given frequency and orientation. The center point in the power spectrum is the continuous component – the image average. Points increasingly further away from the center correspond to increasingly higher frequencies. The power spectrum of an image is always symmetric (real valued signals). Figure 3.6 shows several examples of power spectra for stochastic textures.

The power spectrum also has strong links with the auto–correlation image – that is the convolution of the image with itself. In fact the power spectrum is the Fourier transform of the auto–correlation image. This is easily seen by the convolution theorem, which states that the Fourier transform of a convolution is the pointwise product of Fourier transforms. The convolution theorem is very important for texture synthesis and filtering, and will be used throughout Section 3.4.2.3. The link to the auto–correlation image also explains why the power spectrum describes second–order statistics. The auto–correlation image measures the correlation between pixel values for all offsets in the image.

A texture is sometimes understood as a window of values taken from a larger process: in fact a random process generating values in the infinite plane. In this case the power spectrum of the texture gives a noisy estimation of the periodogram of the process. In order to achieve a more robust estimate of the periodogram of the process it is not enough to use a larger window: one must average the power spectra of several uncorrelated windows of same size.

The power spectrum describes only part of the signal: contrary to the Fourier transform it does not capture any information about the phases of each frequency. Thus, it cannot capture phase interactions between frequencies which typically create edges and contours in an image. Texture synthesizers based solely on the power spectra



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

Figure 3.6: *Various gray stochastic textures (top row) and their power spectra (bottom row). These textures are produced by our Gabor noise method, which is discussed Section 3.4.2.3.* (a) *Anisotropic texture: note the strong orientation. In the power spectrum we can see that this texture has energy around a peek frequency. This explains the strong orientation and wavy aspect of the texture.* (b) *Isotropic texture: note the lack of orientation. In the power spectrum we can see that the energy is focused around a circle. This texture has energy in all direction, explaining the isotropic aspect. The energy is also limited within a band, explaining why the texture seems made of a pattern having a defined size.* (c) *and* (d) *Other examples with arbitrary power spectra.*

are therefore inherently limited to textures without contours and edges. Galerne *et al.* [GGM11] exploited this fact further to synthesize textures by randomizing their phase. They demonstrated that a large class of textures, namely *Gaussian textures*, are *phase–independent*: phase randomization produces variations of these textures without breaking their visual appearance. We extended this work to by–example procedural synthesis of Gaussian textures [GLLD12] using Gabor noise [LLDD09].

Dischler *et al.* [GD95b, DGF98] proposed to synthesize textures using directly their Fourier spectrum, accounting for the phase. This allows edges to be preserved. The main challenge is that only a small number of frequencies can be reproduced efficiently by these techniques.

3.1.2.3 Color space

Texture analysis and synthesis often requires to compare vectors made of color triples. These comparisons typically rely on the L^2 norm. However, the RGB color space is not necessarily the best suited. Several color spaces are perceptually motivated, taking into account – for instance – the fact that the eye is more sensitive to a change in luminance than a change in chrominance. Popular color spaces besides RGB are HSL, HSV, XYZ, Lab, but there are many others available.

Many of the early texture synthesizers were dealing with luminance only. An interesting approach to extend monochromatic synthesis to color, introduced by Heeger and Bergen [HB95], is to decorrelate the three color channels and synthesize them independently. The decorrelation is done through Principal Component Analysis (PCA) of the color points in the example image. This finds a transformation of the color space minimizing linear dependencies between the color channels. However, as noted by Kopf [KFCO$^+$07], this only works on a limited set of textures where colors are only *linearly* correlated. In many cases colors have a strong and complex dependency and may not be synthesized separately.

Other approaches seek to define an *appearance space*, in which distances better capture the image content at a pixel. For instance in our 2006 work *Appearance space texture synthesis* [LH06a] we do not directly rely on the RGB color at a pixel but replace it by a small vector characterizing the appearance surrounding it. The appearance vector is computed by concatenating RGB colors of the pixel neighbors, and possibly

other information such as the distance of each pixel to an edge – also called *feature distance*. This high dimensional vectors are analyzed through PCA and reduced to a smaller number of dimensions for greater efficiency. Indeed, PCA orders the dimensions by importance, the first axis being the one along which the data has greater variance. We typically keep 90% of the variance which is captured by a small number of the initial dimensions. Other approaches consider the manifold defined by connecting nearby color points within the color space. For instance Wang *et al.* [WTL+06] map a single parameter along the color manifold to control the variation of colors in an aging exemplar. Lu *et al.* [LDR09] diffuse heat in the manifold to later segment out the main appearance from an image comprising several textures.

As of today it is unclear whether a color space is truly superior in terms of synthesis quality. We know however that image color points and color neighborhoods often lie on low dimensional sub–spaces. This is why PCA and other dimensionality reduction techniques (Isomap [TdSL00], LLE [RS00]) are very effective acceleration techniques when comparing these otherwise high dimensional vectors. Similarly, we have observed that synthesis quality greatly improves by including additional information at each pixel, such as the feature distance [LH06a]. In this particular case the additional information compensates for the inability of the L2 norm to compare neighborhoods along sharp edges and contours.

3.2 Texture synthesis: objectives and challenges

Texture synthesis has been a topic of interest for several decades. However, the objectives pursued by researchers vary greatly: the constraints on the process synthesizing the textures will be very different depending on whether we seek for a theoretical analysis, or whether we target very fast synthesis of large texture maps. This resulted in a variety of models and algorithms. I refer the interested reader to our state of the art [WLKT09].

In Computer Graphics, there are essentially two scenarios for texture synthesis: off–line and on–line synthesis. In an off–line synthesizer, a high-resolution texture is synthesized and stored for later use. The expected synthesis time is in the order of tens of seconds. The main benefit is to simplify/accelerate the authoring process. An on–line synthesizer is typically used in an interactive application (simulation, game). The textures are either quickly synthesized when the user enters a new area, or are computed on–the–fly as the surfaces are painted on screen.

In all cases, it is important for the practical value of the algorithms that they offer a strong control over the results to the artists.

3.2.1 Data–driven, by–example and procedural synthesizers

Due to its long history, the terminology around texture synthesizers is sometimes confusing. For the sake of clarity, I define here how to understand the main properties of texture synthesizers in the remainder of this document:

- **Procedural synthesizer**: a procedural synthesizer is able to *compute* a texture from only a small set of parameters. It is expected that the number of parameters is several orders of magnitude less than the number of pixels required to visualize the texture properly (i.e. all its frequency content). In the Computer Graphics community, a *procedural texture* is often understood as a function taking a coordinate as parameter and returning the texture properties at this location (color, normal, etc.) in constant evaluation time. This is the meaning we use throughout this document, unless otherwise specified.

- **Data–driven (sometimes *non–parametric*) synthesizer**: we refer to a synthesizer has being *data–driven* when it exploits an input example to produce an output. Typically, small parts of the input are copied and combined together to form the result. These algorithms are by definition *by–example* synthesizers.

- **By–example synthesizer**: a synthesizer which takes an example image as input, and produce a different, possibly larger output. The quality of the output is judged in terms of its visual similarity with the input. The distinction with *data–driven* synthesizers is important – in particular, we introduce Section 3.4.3.2 a *procedural*, by–example synthesizer.

3.2.2 Summary of contributions

My work has essentially focused on *on–line* synthesizer. In particular, my post–doctoral work made data–driven synthesizers available for on–line synthesis, while algorithms were previously limited to off–line synthesis. This contribution is described Section 3.3.2.

We also investigated *procedural–texturing*. In this context, it is expected that the texture is represented by a procedure *computing* in constant time the color of any point of the surface [Per85]. This is very challenging since the computation must be very fast, use little memory, and has to be done for a single point at a time. This makes any scheme relying on second–order statistics very challenging as the value of neighboring points is not easily determined during the evaluation at a given point. We contributed a new model for procedural texturing which is nevertheless able to control the produced texture through its power spectrum, and hence its second-order statistics. It is described Section 3.4.2.3. We later extended this approach to learn the parameters of the procedure directly from an example texture (see Section 3.4.3), therefore introducing a by–example procedural texture synthesizer.

Throughout this work, I kept a constant focus on making our synthesizers available for Computer Graphics applications. In particular, this means being able to produce the textures *along the surface of objects*, as well as being able to filter out unnecessary details, avoiding frequency aliasing. This brings another set of challenges, discussed Section 3.5.

This chapter is concluded by introducing a synthesizer addressing a different set of textures. While most of my approaches focused on stochastic textures, I also proposed approaches for structured, organized content. The context remains the same, and our approach produces results which remain very compact in memory.

3.3 Data–driven synthesis of stochastic textures

My main contributions to the field of content generation are data–driven, parallel, controllable texture synthesis algorithms. The motivation for these algorithms directly stems from the context of my research: can we generate graphical content while the user explores a scene, rather than pre–computing and storing it?

While several data–driven algorithms for high–quality texture synthesis were available prior to our work, most of them required seconds to synthesize a unique texture: these algorithms have been designed to synthesize an entire texture at once, using either an inherently sequential generation processes or a global optimization procedure [WLKT09]. This prevents parallelism and does not allow the generation of only a sub–part of the content. In addition, most algorithms did not offer explicit control over the amount of randomness introduced. Our algorithms improve each of these aspects.

The first algorithm I developed introduces a parallel, controllable synthesizer [LH05]. The structure of the computations map well to the GPU, but also allows for explicit control over the randomness introduced during synthesis. This offers two important advantages. First, this allows for unprecedented control over the results. Second, this progressively introduces variations into the result: high quality results are obtained in less iterations. We later extended this approach to the synthesis of complex, view–dependent textures over surfaces [LH06a]. Our synthesizer directly generates a texture in the parametric domain of the surface and properly handles discontinuities and distortions. In both cases our algorithm is parallel and spatially deterministic. It can synthesize any window from an infinite texture plan with the properties that 1) computational complexity only depends on the synthesized window size 2) the content generated at given coordinates in the plan always remains the same. In later work, we exploited these properties for the lazy synthesis of solid (3D) textures. This is especially interesting when using volume textures on surfaces, as the surface typically use only a small percentage of the solid texture. Our synthesizer only synthesizes what is required for the surface.

Other extensions of this algorithm include the synthesis from and into photographs [ELS08], the synthesis of textures following guidance [BvdPLD10, PELS10]. Finally, other researchers used our scheme as the basis of their work: synthesis of multi–scale scenery [RHDG10], structure preserving synthesis [HRRG08]. Our technique was also used in the production of the animated movie Tangled by Disney [ETB+10], for instance to synthesize details along the chameleon, pavements in the cities and tree barks.

In this document I focus on the core ideas and will not detail the content of all related publications. I describe

the algorithm in a generic way, and give insights and additional information gathered since we introduced this approach. Since our algorithms belong to the family of data–driven texture synthesizers, I recall in the next section the fundamental ideas behind such schemes. In particular, I will re–emphasis the connections between the different approaches.

3.3.1 Fundamentals of data–driven texture synthesis

Data–driven texture synthesis approaches distinguish themselves from other approaches by the fact that they rely directly upon the example data to synthesize new images. This is in contrast to *parametric* synthesizers (see Section 3.4.2.3) where the texture model is specified by a few parameters that may be directly controlled by the user.

Intuitively, the process of synthesizing a novel texture can be understood as choosing colors in each pixel so as to locally reproduce features visible in the example image. The assumption is that if the synthesized image is *locally* similar to the example everywhere, then it will *globally* exhibit the same appearance, despite being organized differently at large scale. As we shall see, there are different ways to formalize this objective, as well as different ways to produce images under this objective.

In the Computer Graphics texture synthesis literature, it is often hard to distinguish between *the formulation* of the problem and *the algorithm* which attempts to optimize for images. Both are however equally important for understanding the challenges in runtime texture synthesis. In the following sections, I will therefore re–emphasize how most synthesizers formulate the problem of texture synthesis (Sections 3.3.1.2 to 3.3.1.4), before discussing algorithms (Section 3.3.1.5).

3.3.1.1 Patch–based or pixel–based?

> *Understanding is the knowing of misunderstanding* Zivarnna Smithies

There are many different schemes in the literature, which are often classified in two categories: pixel–based and patch–based approaches [WLKT09]. These categories however do not always properly reflect the fundamental differences between data–driven texture synthesis approaches.

Pixel–based approaches are approaches which synthesize a texture by operating at the pixel level. Each pixel is considered in turn, and its color is determined from already synthesized neighboring pixels.

Patch–based approaches have been named after schemes *explicitly* copying large patches[1] from an example texture into the result [PFH00, GSX00, EF01]. For instance in image quilting [EF01] the patches are added in each cell of a regular grid covering the output. The frontier between patches is hidden after the copy, often by finding a cut hiding differences in an overlap area. Therefore, these schemes are understood as a sequence of operations: 1) select a region not yet covered, 2) find a large patch matching the region of overlap, 3) paste it into the result and hide the seams. While this appears a reasonable greedy approach, graph cut textures [KSE+03] introduced a formal model for patch–based synthesis. Unfortunately, this is often under-looked due to the fact that the implementation appears similar to the sequence of operations described above.

However, as I will discuss Section 3.3.1.3, patch–based schemes as formulated by Kwatra *et al.* [KSE+03] are indeed per–pixel approaches. The fact that patches are observed in the results arises naturally from the formulation of the objective, which is defined on the pixels of the output. In a sense, early patch–based approaches were explicitly trying to create the patches that we expect to observe in the results: the algorithms are designed to explicitly manipulate patches so as to construct solutions of the correct form. The downside, however, is that the formulation behind the solver remains hidden. This sometimes gave the impression that the algorithms were somewhat arbitrary, when in fact Kwatra *et al.* [KSE+03] already provided a well–posed framework.

I propose below a different distinction between texture synthesis approaches: those formulating the problem as matching neighborhoods of colors, and those formulating the problem as a *geometric* optimization, exploiting pixel coordinates. In both cases, two point of views are discussed: a Markov Random Field point of view,

[1] In this document we use *neighborhood* to refer to small patches (e.g. 7×7 pixels) and *patch* to refer to large pieces of an image (e.g. 32×32 pixels and above).

and an energy minimization point of view. Finally, since many recent schemes use in fact a mixture of both approaches, I will discuss the links between the two formulations Section 3.3.1.4.

3.3.1.2 The neighborhood of colors formulation

I spent an interesting evening recently with a grain of salt. Mark V. Shaney

Markov Random Field point of view Among the most successful methods for by–example texture synthesis are those modeling the texture as a Markov Random Field (MRF) (see also Section 3.1.1). These approaches seek to reproduce color pixel–neighborhoods similar to those of the exemplar across space and scale. The size of the neighborhoods depends on the texture itself, even though this problem is alleviated by a multi–scale approach [PP93, PL95]. Typically neighborhoods of size 5×5 or 7×7 are considered across scales. The neighborhood size is an important parameter to the texture model. Results vary greatly for different values.

Formally, the MRF is defined by a local conditional probability density function (LCPDF) which gives, for every pixel, the probability of the pixel taking a given color knowing the color of its neighbors. For a synthesized image S, at a pixel coordinate p, we note the probability of the pixel taking value x knowing the values of its neighbors as:

$$P(S[p] = x \mid \mathcal{N}_S(p) \backslash p)$$

where $\mathcal{N}_S(p) \backslash p$ is the color neighborhood around p in S excluding the center color at p. The local probability function is either parametric and fitted to the data, or directly sampled from the example image. We discuss this aspect in more details Section 3.3.1.5.

The set of all local probabilities leads to a global probability $P(I)$ measuring how likely any given image I is to be similar to the texture. This defines a *model* of the texture appearance. Assuming the texture appearance can indeed be captured by an MRF, an image with high probability will be visually similar to the exemplar. On the contrary an image with low probability is unlikely to be visually similar to the exemplar. Synthesis will therefore amount to finding images with high probability.

Energy minimization point of view Following the point of view of matching color neighborhoods, synthesis can also be written as an energy minimization. Instead of starting from the conditional probability, we define a global energy measuring how well the neighborhoods of a synthesized image match the neighborhoods of an example. The neighborhoods are compared through a dissimilarity metric, most often a simple L^2 norm in RGB space. Other norms are of course possible [BvdPLD10, RCOL09]. Given a synthesized image S, we define the single resolution neighborhood matching error as:

$$\mathcal{E}(S) = \sum_{p \in S} \left(\min_{z_p \in E} ||\mathcal{N}_S(p) - \mathcal{N}_E(z_p)||^2 \right) \tag{3.1}$$

$\mathcal{N}_S(p)$ is the neighborhood around p in S and $\mathcal{N}_E(z_p)$ the neighborhood around z_p in E. The min searches for the location z_p giving the neighborhood best matching $\mathcal{N}_S(p)$ in E. This formulation is similar to the ones used in [WSI04] and [KEBK05]. Its goal is to minimize the distance between color neighborhoods in S and their best matches in E.

3.3.1.3 The geometric formulation

When adopting the geometric point of view, synthesis does not consist in computing colors. Instead, it consists in *copying* pixels from the example into the synthesized image. Thus, the synthesized image is not an array of colors, but an array of pixel coordinates. This subtle difference leads to a change in the texture model, with results of a different visual quality.

Energy minimization point of view Let us first formulate the problem as an energy minimization. We now seek to select coordinates in S so as to minimize the following energy:

$$\mathcal{E}(S) = \sum_{p \in S} \left(\sum_{\Delta \in \{-1,0,1\}^2} ||E[S[p]] - E[S[p + \Delta] - \Delta]||^2 + ||E[S[p] + \Delta] - E[S[p + \Delta]]||^2 \right) \tag{3.2}$$

Intuitively, this energy states that the choice of coordinates is good if the colors at respectively $S[p]$ and $S[p]+\Delta$ match well the colors at $S[p+\Delta]-\Delta$ and $S[p+\Delta]$.

This energy can also be interpreted as minimizing color difference at the frontiers of patches. Let us write L_{ij} the *label* which corresponds to translating a copy of E at position (i,j) in S. We can rewrite any selection of coordinates in S as a selection of labels in a table T: $T[p] = L_{ij}$ so that $(i,j) = p - S[p]$. Each label L_{ij} now corresponds to a patch, and it is easy to verify that the energy only takes non–zero values at the frontier between different labels (patches). Hence, *this energy will have a natural tendency to grow geometric patches in the result*. The patches *arise from the formulation*, and are *not* explicitly introduced during synthesis.

This formulation as a labeling problem can be found in several papers: graph cut textures [KSE+03], a report by Demanet *et al.* [DSC03], as well as in several following papers [Kom06, RB07, PKVP09] (it is quite common in the Computer Vision literature). We discuss Section 3.3.1.5 the typical approaches used to optimize for the best choice of labels. The reader interested in a formal analysis of the energy in Equation 3.2 can also refer to the work of Aujol *et al.* [ALM10].

Markov random field point of view The energy $\mathcal{E}$ may be used to define a specific type of MRF known as a *Gibbs field*. The energy is turned into the probability that a synthesized image S belongs to the model of the texture. It can be seen that the energy has the form of a Gibbs energy by rewriting it in terms of pairwise relationships:

$$\mathcal{E}(S) = \sum_{(p,q)\in C_2} D_{pq} + D_{qp}$$

where $D_{pq} = (E[S[p]] - E[S[q] + (p-q)])^2$ and C_2 is the set of all size–two cliques formed by the neighborhoods: the unordered pairs $\{(p,q)|q = p+\Delta\}$ with $\Delta \in \{-1,0,1\}^2 \backslash (0,0)$.

The probability corresponding to $\mathcal{E}$ is:

$$P(S) = \frac{1}{Z} e^{-\beta \mathcal{E}(S)}$$

where β is a temperature parameter and Z a normalizing constant. The normalizing constant implies computing over all possible S and therefore cannot be computed (it is fortunately not necessary to know its actual value, see Section 3.3.1.5).

By the Hammersley–Clifford theorem it is known that the above probability defines an MRF. We can derive the conditional probability at any pixel, which is computed using the standard relationship $P(A|B) = P(A \bigcap B)/P(B)$ as:

$$P(S[p] = x \mid S[p+\Delta]) = \frac{1}{Q} e^{-\beta \left(\underbrace{\sum_{q=p+\Delta} D_{pq}}_{\text{1st term}} + \underbrace{\sum_{q \text{ st. } p=q+\Delta} D_{qp}}_{\text{2nd term}} \right)}$$

with $\Delta \in \{-1,0,1\}^2 \backslash (0,0)$, and Q a normalizing constant which is computed over all possible values of x. The first term measures how visible the transitions around p in S are. The second term corresponds to the influence of p in the transitions to its neighbors. In essence, this probability states that x is a more likely value at p if it does not introduce color differences when going from p to one of its neighbors.

3.3.1.4 Link between neighborhood of colors and geometric formulations

The two formulations discussed above are not incompatible. In particular, it is possible to include a geometric term into the energy for matching neighborhoods–of–colors. Interestingly, this idea was first introduced by Ashikhmin [Ash01] as an acceleration technique for texture synthesis. While results were produced faster, the quality was also significantly improved on several texture types (named *natural textures* by Ashikhmin). This improvement, which may seem surprising, originates in fact from the change in the energy being optimized: it now includes a geometric term. The algorithm proposed by Ashikhmin can be understood as a fast approximate solver for this energy (see also Section 3.3.1.5).

Similarly the geometric energy can be modified to take color neighborhoods into account. This is the formulation we used in our first parallel texture synthesis paper [LH05] (see Section 3.3.2). An interesting feature of such an approach is that it offers a controllable, continuous tradeoff between both formulations. This lets the user choose between encouraging patches, or relying only on color neighborhood matching. This is useful since these two formulations do not perform equivalently on all textures. Other schemes include a similar idea, for instance the formulation of [Kom06] through the *incoherence penalty* term.

3.3.1.5 Synthesis algorithms

We have previously seen two main formulations, each either as an energy optimization problem, or as a Markov Random Field model. We now discuss in more details how images can be synthesized from each variant of these formulations.

The link between the MRF and energy point of views is worth mentioning. The energy minimization point of view seeks for the best possible image: the global minimum of the energy. It is interesting to note that in some cases this image does exist. If the input example is a cyclic image, as is sometimes the case, the best possible image for the energy Equation 3.2 is a regular tiling of the example. However this result is not desirable due to the highly periodic content. The very reason for which we synthesize images is to avoid such artifacts and obtain large scale variations in results. This outlines an interesting issue: the formulation itself does *not* explicitly avoid periodicities. This is questionable since this is a major objective when synthesizing textures for rendering. In fact, the global optimum is not even desirable – this implies we actually want to avoid the best possible result! The key here is that even though the best image may be bad, the energy landscape is in fact very complex. One can expect that there exists a large number of local minimum which actually correspond to visually convincing images. Therefore, starting from a random initialization, a local search algorithm is likely to discover and remain in such a local minimum.

The MRF point of view seems to better capture this idea. Rather than searching for the single best image it simply associates a probability to each image. Therefore, generating results involves sampling high probability images. This corresponds better to our initial goal of generating a variety of images. Nevertheless, properly sampling the space of images is no less difficult than performing a search over local minima of the energy landscape.

In an excellent discussion on this issue in the context of inpainting, Demanet and colleagues [DSC03] however argue for the energy optimization approach, noting than when sampling a distribution the typical outcomes are often quite different from the most probable outcome. In my experience, this often boils down to a tradeoff between noise (sampling) and bad local minima such as obvious cuts between patches (local minimization). Whether one approach truly is superior remains an interesting open question.

We next discuss various algorithm for MRF sampling and energy minimization in the context of data–driven texture synthesis.

Markov Random Fields synthesizers MRFs are a well known formalism and generic techniques exist for their sampling. They both apply to formalization using neighborhoods or using the geometric approach.

Textures are typically synthesized by optimizing for the maximum a–posteriori of the Markov field, that is the image maximizing $P(I)$. Another approach is to *sample* the set of all possible images according to the probability distribution defined by P. The goal is to generate a sequence of images from the MRF, with the property that after a *sufficient time* the produced sequence follows the probability distribution. (In fact, the sequence of images itself is a Markov chain).

Several methods are available, most of which involve sampling the local probability function at a given pixel knowing the color of its neighbors. For instance, stochastic relaxation [PL95], Gibbs sampling (Metropolis Hastings), or Iterated Conditional Modes (ICM) [Bes86] have all been applied to texture synthesis. The later approach is one of the most popular, due to its simplicity of implementation.

Popat [PP93] applied these ideas early to by–example texture synthesis. The local probability is parametric and fitted on the example data. In Paget and Longstaff [PL95] the local probability is approximated from a multi–dimensional histogram of the pixel values in the neighborhoods. Since the observation of the texture is limited

to the example image, this histogram is very sparse and has to be smoothed to obtain a usable local probability function. Sampling of the MRF is performed with stochastic relaxation. Because the local probability is based on example data rather than on a parametric description, the scheme is said to be *non–parametric*.

Another popular MRF approach introduced by Efros and Leung [EF01] is to generate the image one pixel at a time in sequence, using a partial neighborhood of already synthesized pixels. The conditional probability is used to sample the value of the next pixel. In Efros and Leung the conditional probability is approximated by searching for a limited number of *close* neighborhoods in the exemplar, forming an histogram of possible values for the pixel. A selection threshold is determined from the distance to the *closest* neighbor. The pixel value is sampled at random following the histogram distribution.

Wei and Levoy [WL00] further simplified this process by directly selecting only the best matching neighborhood – an approach which closely resembles the ICM update rule of Besag [Bes86]. Synthesis time is greatly reduced by relying on an acceleration data structure for approximate nearest neighbor queries. Note that this scheme is closer to an optimization technique, in the sense that it seeks the image maximizing the joint probability of the Markov random field rather than trying to *sample* the probability distribution. See also the introduction of Section 3.3.1.5 for a discussion on this topic.

For all these schemes best results are obtained through a multi–scale approach: a Gaussian pyramid of the example image is computed. The top of the pyramid is a single pixel having the average color of the image, and each subsequent pyramid level increases in resolution and details. Synthesis is performed from coarse to fine, each next level being initialized with the result of the previous. This improves quality and alleviates the issue of carefully selecting the neighborhood size.

Energy optimization for labels The energy formulation is in essence a discrete label assignment problem. While tractable in 1D, for instance through dynamic programming, belief propagation (BP) [SYJS05], or Dijkstra [LHL10], there is no polynomial time exact optimizer for the 2D case. Several good approximations however exists, such as loopy BP [CBAF08], prioritized BP [Kom06], and alpha–expansion [KSE+03, RB07, PKVP09]. All are applicable to the problem of texture synthesis. A survey by Szeliski *et al.* [SZS+08] compares several of these approximate approaches.

Faster approaches exist to quickly obtain approximate solutions by exploiting the form of the problem, in particular the coherence due to patches. These ideas can be found in the early works of Ashikhmin [Ash01], Tong *et al.* [TZL+02] and Demanet [DSC03].

Energy optimization for neighborhoods of colors Optimizing the energy for the neighborhoods of colors is difficult due to the best–match term z_p in Equation 3.1, which prevents the computation of a smooth gradient.

In the work of Wexler *et al.* [WSI04] pixels are updated iteratively by the following rule: all patches covering a pixel are extracted, and a best matching patch is found for each in the exemplar. Each best match suggests a color for the pixel, and the final color is a weighted average of these suggestions. The weights are proportional to the matching distances of the patches.

Kwatra *et al.* [KEBK05] proposed a scheme inspired by Expectation Maximization [MK97], where the variables z_p and the colors are optimized in two different steps of an iteration. First, given initial colors, the best–matching neighborhoods z_p are computed. Second, given the z_p, the colors are optimized to reduce the matching distances. This scheme was later implemented on the GPU for faster synthesis [HTW07], as well as extended to solid texture synthesis [KFCO+07]. It is quite versatile since other objectives such as temporal coherence on dynamic textures may be mixed in the energy being optimized.

All these approaches share a common bottleneck: the search for best–matching patches. *PatchMatch* [BSFG09] may be used to accelerate this step. Similarly to approaches for label optimization, the choice of neighboring pixels is used to suggest natural candidates for the current pixel, while a random search helps escaping local minima.

3.3.2 A fast and controllable texture synthesis algorithm

The generation of random numbers is too important to be left to chance. Robert R. Coveyou

The main motivation for our algorithm was the need for a fast algorithm synthesizing high–quality textures during the course of the application, as the user explores the scene. An important difference between our work and prior schemes is that we seek to synthesize textures and make them readily available for rendering. This implies the following requirements:

- The algorithm has to be able to synthesize sub–parts of very large textures, in a deterministic manner. For instance, if the user is walking a landscape, only the parts of the textures required for the current viewpoint would be synthesized. This is not possible using sequential algorithms or algorithms following a global optimization strategy, since the color of a given pixel depends on *all* other pixels. In addition, if the user comes back later to a same location, the texture should of course remain unchanged (*spatial determinism*). This requires taking extra care when dealing with boundary conditions.
- Synthesis time should be in the order of tens of milliseconds for textures of a resolution typically found in real–time applications (512×512 in 2005, up to 2048×2048 in 2013). Considering the amount of computations and memory bandwidth required this can only be achieved through an efficient, parallel, GPU implementation.

In addition CG artists need to have a strong control over the synthesis process, for instance by fixing the appearance of the texture in specific locations.

Our scheme is inspired both by the neighborhoods of colors and geometric formulations. However, the most important aspect of our work is not the formulation – which is essentially following previous work – but the overall structure of our algorithm, which affords for direct control over randomness, spatial determinism, sub–part synthesis and an efficient parallel implementation.

A key hindsight of our approach is that by starting close to a good solution we can quickly recover a high–quality result. This is why in absence of explicitely introduced randomness our algorithm produces the best possible image: A perfect tiling on the input (assuming the input is toroidal – see Section 3.3.1.5). The user controls the amount of randomness (perturbation) that is introduced. Thanks to an instantaneous feedback, the user can select the most appropriate perturbation for a given input texture: The best compromise between quality and removal of periodicities.

After introducing notations in Section 3.3.2.1, I describe Section 3.3.2.2 our algorithm. I provide a generic description, since our approach is modular and may be easily modified for different use. I later discuss possible variants of each individual components.

3.3.2.1 Notations

Let us first introduce notations. We note E the exemplar image and S the synthesized image. We note $S[p]$ the value stored at pixel p in S, and use a similar notation for E. Our scheme relies both on neighborhood of colors and on pixel coordinates. Internally, instead of storing a color value directly in the synthesized image S, we store the coordinate of the chosen pixel in E. That is, the exemplar coordinate $x_p = S[p]$ at p in S corresponds to the color value $E[x_p]$. This enables several advantages in both performance and quality.

Our synthesis algorithm performs a coarse to fine synthesis, synthesizing increasingly higher resolution coordinate arrays S_i, where i is the resolution level. We note L the finest resolution level and 0 the coarsest. We note $W_L \times H_L$ the final resolution of the synthesized image, with the property that $W_i = \frac{W_L}{2^{L-i}}$ (and similarly for H_i). The number of resolution levels depends solely on the resolution of the example image. We compute a Gaussian (MIPmap) pyramid from the example image. We treat each level as an independent example image, using level E_i when synthesizing S_i.

3.3.2.2 Algorithm overview

A generic algorithm relying on our approach performs the following steps:

```
1 S_0 = array of size W_0 × H_0 initialized to (0,0)
2 for resolution level l from 1 to L do
3   S_l = upsampling( S_{l-1} )
4   S_l = jitter( S_l )
5   S_l = correction( S_l )
```

```
6 end
```

The three main steps of the algorithm are *upsampling*, *jitter* and *correction*. Contrary to previous approaches, we explicitly distinguish between the increase in resolution (upsampling), the addition of randomness (jitter) and the actual synthesis process (correction), enabling control over each step.

Upsampling Upsampling is the process of taking the previous resolution level and increasing its resolution. This can be done very efficiently by relying on coordinates within the example image pyramid. We upsample each pixel as $S_l[(i,j)] = S_{l-1}[(i/2, j/2)] + (i\%2, j\%2)$ with integer arithmetic and $\%$ the modulo operator. Intuitively, each pixel from E_{l-1} is replaced by its four children at the next resolution level E_l.

Later work in the community modified this step to jump between different exemplars across resolution, enabling effects such as infinite zooms [HRRG08].

Jitter The jitter step introduces variations in the synthesized texture. In absence of jitter, a simple tiling is obtained – this is a direct tiling of the texture if it is toroidal, otherwise a seamless tiling is obtained as a side effect of the synthesis (next step). This also means that in absence of jitter, the synthesis step has little to no work to do.

This property is an important aspect of our work. Variation is only explicitly introduced during the jitter step. In our original work [LH05], randomness is introduced by adding an offset to the synthesized coordinates. The offset is a random direction and its magnitude is exposed to the user. Large scale changes may be introduced by using large offsets at a coarse resolution, while large offsets at a fine resolution impact the fine scale details. This lets artist preserve large scale structures while introducing fine variations. In order to enforce determinism pseudo–random numbers are generated from the pixel coordinates.

An important aspect of jitter is that it lets prior knowledge be used to guide the synthesis process. In our original scheme, we exploited this to preserve regular patterns by constraining the offsets. However, this required users to specify the frequency of the patterns to be preserved. Later, Risser *et al.* [RHDG10] extended this idea and proposed a synthesis scheme synthesizing images of structured objects. It relies on a modification of the jitter step, ensuring that the introduced randomness does not break the overall structure of the image. This approach is only possible thanks to our explicit jitter strategy.

Correction The third and final step is the correction step. It ensures that the texture appearance remains similar to the exemplar, keeping only the variations introduced by jitter which are compatible with it. The correction algorithm seeks to replace each pixel by one improving its local neighborhood, similarly to MRF schemes (see Section 3.3). This is done by searching the exemplar for best matching neighborhoods. We use the output of the jitter step as the initial image.

Instead of a sequential traversal we process pixels *simultaneously*, using the graphics processor or a multi–core CPU. We perform several passes, each pass *reading from the previous* pass result while *writing to the current* pass result. The result for pass i at level l is noted S_l^i.

Naively processing all pixels simultaneously within a pass however reduces synthesis quality. Indeed, the correction of a single pixel relies on neighbors which are being updated simultaneously. Therefore, the *previous* value of the neighbors is used, instead of the one most recently computed. To avoid this issue, we split pixels in a small number of independent sets, processed in sequence. We call these additional iterations *sub–passes*. This process is similar to the parallel Iterated Conditional Modes update rule of Besag [Bes86]. Please refer to our original paper for more details on how to efficiently perform this mechanism, and how to best choose the ordering of sub–passes.

The pseudo–code below gives a generic view of the correction algorithm. The sub–pass mechanism is not included for clarity. The components of this algorithm are detailed next.

```
1 for i = 1 to P passes do
2   for each pixel p in parallel
3     c         = first_candidate (S_l^{i-1},p)
4     best_dist = ∞
5     best_c    = c;
```

```
6    while c not null do
7      dist = distance ( N(c,E_l) , N(p,S_l^{i-1}) )
8      if dist < best_dist then
9        best_dist = dist
10       best_c    = c
11     end
12     c    = next_candidate (p,c, best_c)
13   end
14   S_l^i[p] = best_c
15 end
16 end
```

S_l^0 is the output of the jitter step. In a naive implementation, `first_candidate` and `next_candidate` would enumerate all exemplar pixels. Different search strategies are discussed Section 3.3.2.3.

3.3.2.3 Search strategies

We discuss in this section the various possibilities to implement the search in the correction step (see Section 3.3.2.2, Correction). The tradeoff is essentially between performance and synthesis quality.

Exhaustive search The exhaustive search simply consists in parsing through all the pixel neighborhoods of the exemplar, searching for the best matching one. Clearly a lot of computations are wasted, for two reasons: First, many of the neighborhoods are very dissimilar and could be safely ignored. Second, among the well matching neighborhoods we are not necessarily interested in the absolute best one: At this stage we are sampling the local probability function for the pixel (see Section 3.3.1), and we need to find a *most likely* value for the pixel. The two strategies below improve over this.

Coherent and k-coherent search Interestingly, the largest improvement in performance was initially motivated by improving the synthesis quality – two objectives considered antagonist. Observing that neighborhood matching synthesis does not work well on some stochastic textures – in particular those with identifiable features – Ashikhmin [Ash01] proposed to encourage the formation of entire patches from the exemplar image. This intuition has strong links with the works on patch–based synthesis [EF01, KEBK05] which are synthesizing new images by copying entire patches from the exemplar. The idea of Ashikhmin is to exploit, during sequential synthesis, the already synthesized neighbors of the considered pixel. Each of these neighbors can be used to propose a candidate. For a given neighbor, a natural candidate is simply the pixel next to it *in the exemplar*. That is, the pixel which would appear next to the neighbor if a patch was grown. This idea is illustrated Figure 3.7, (a). This avoided one of the main artifact of MRF texture synthesis, known as the melting effect: Entire areas locking into using neighborhoods from a small region of the exemplar. In addition, synthesis speed greatly improved since so few candidates are considered at every pixel.

The scheme of Ashikhmin is only considering direct neighbors as candidates. These are called *coherent* candidates. This leads to tearing artifacts on many textures, and the scheme is only performing well on textures having well identifiable texture elements and high frequencies. Tong [TZL+02] bridged this idea with standard MRF synthesis by introducing the idea of k-coherence. Each exemplar pixel is now associated with a (small) set of its k-most similar pixels within the exemplar. We note $C_i(p)$ the i-th most similar pixel of p. These sets are pre–computed once for the exemplar. When the coherent candidates are determined, all their k-most similar pixels are also included into the set of candidates. The best match is searched amongst these. A parameter κ weights the coherent candidates differently from the others so as to control whether to favor or not the apparition of patches. With a proper setting of κ the resulting synthesizer performs well on many types of stochastic textures, and synthesis speed is still orders of magnitude faster than the exhaustive search.

Our work [LH05, LH06a] uses a similar k-coherence mechanism. More precisely, for a pixel p being synthesized we obtain the set of candidates as:

$$\mathcal{C}(p) = \{C_i(S[p+\Delta]) - \Delta | i = 1...k, ||\Delta|| < 2\}$$

For best results, the k-most similar pixels are chosen to be spatially spread across the exemplar. This is done by selecting more than k candidates and keeping the k-most distant amongst those. Additionally our scheme

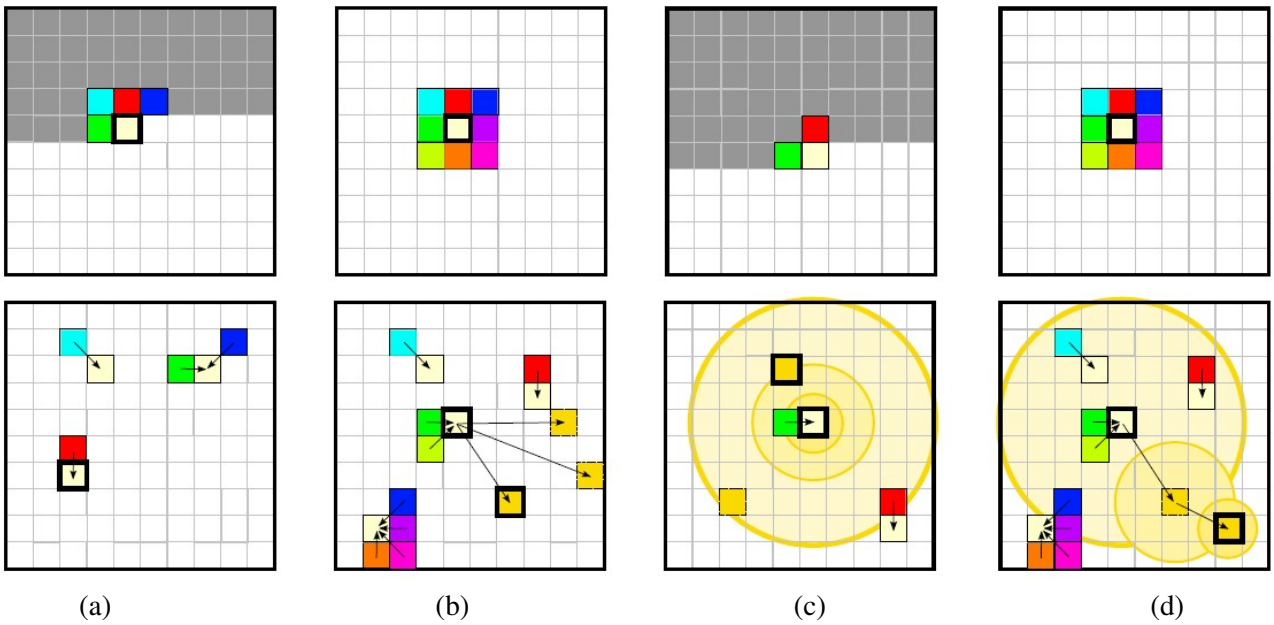


Figure 3.7: (a) *Coherent candidates.* (b) k*-coherent candidates.* (c) *Search performed by PatchMatch: A random search is carried around each coherent candidate with decreasing radius.* (d) *Random search strategy of Busto et al. : A random walk is performed with decreasing radius. The walk steps if an improving candidate is found.* Figure from Busto *et al.*

projects all neighborhoods into a lower dimensional space obtained by principal component analysis. This lets us store and compare neighborhoods more efficiently.

The main drawback of the k-coherence scheme comes from the required pre–computation and storage of the k-most similar pixels. This can be a severe issue when going to a production environment where very large textures are used and a quick cycle for testing textures is required [ETB+10].

Randomized search Eisenacher *et al.* [ETB+10] proposed a randomized pre–computation for building the k-nearest sets. This approach reduces pre–computation time by orders of magnitude, and was successfully used for the production of the Disney motion picture *Tangled*, released in 2010 – to my knowledge one of the first large–scale use of by–example texture synthesis in a movie. The deterministic sub–part synthesis property of our algorithm is exploited to allow for out–of–core synthesis of high–resolution textures which would otherwise not fit in memory.

In a 2009 SIGGRAPH paper, Barnes *et al.* [BSFG09] proposed to exploit coherent candidates combined to a randomized search to rapidly find approximate nearest neighbors. In Busto *et al.* [PELS10] we followed a similar idea to propose a scheme without any pre–computation. The idea is best captured by Figure 3.7. The intuition behind both schemes is that good matching neighbors tend to be close to the considered pixels in the exemplar. This is in fact a consequence of the locality property of stochastic textures.

This last approach is recommended for its simplicity and efficiency.

3.3.2.4 Using the algorithm at run–time

Our algorithm defines an infinite two–dimensional texture domain. A fascinating property is that the final content of the synthesized texture is solely determined by the randomness introduced in the jitter step. By shifting the pixel coordinates appropriately at each level, we can thus give the illusion of panning over an infinite texture plane. Similarly, we may generate any window of texture, including overlapping ones, and the content will always perfectly correspond. The synthesis cost only depends on the size of the synthesized texture. This is in contrast to sequential algorithms where the cost depends on the *location* of the synthesized texture within the texture space, since all pixels located between the origin and this location must be synthesized to enforce determinism.

In order to achieve perfect determinism we must be careful to synthesize enough content: Neighborhoods at the edge of the synthesized texture exit the synthesized area. Therefore, at each correction pass wrong content

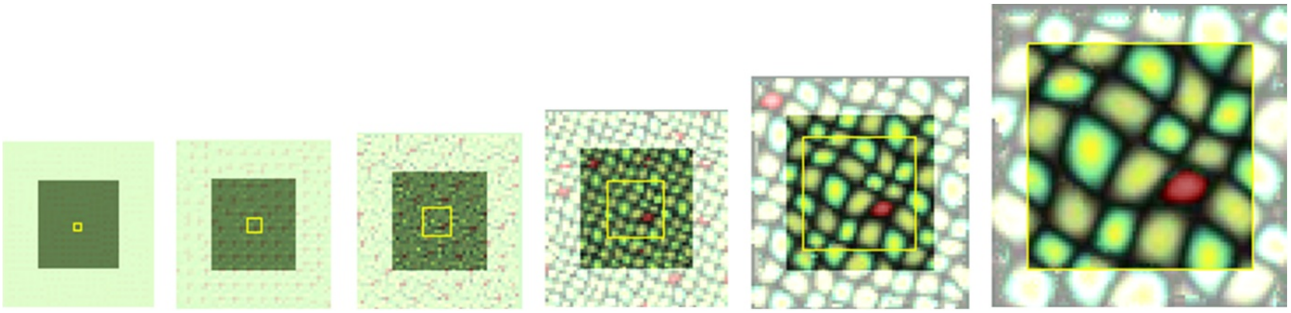


Figure 3.8: *Result of spatially deterministic multi–resolution synthesis. Each image is the result of correction from coarser (left) to finer (right) resolution. The yellow outline is the requested content. At finest resolution, the white border outside the yellow window is the required padding. At the next coarser level this content must be deterministically synthesized, requiring additional padding. This continues from fine to coarse, resulting in a larger padding at the coarsest level. The size of the padding border is constant regardless of the position within the texture space.*

bleeds into the result. To avoid this issue, we must pad the synthesis window at each scale. Figure 3.8 reveals the necessary padding across resolutions for synthesizing a deterministic window of texture.

Generating content at run–time requires temporary buffers to hold the data. I present Chapter 4 the specialized data–structures we proposed to this effect. An important specific case however is terrain synthesis. We followed the approach of the geometry clipmaps [LH04] to synthesize directly elevation and color data at runtime. A clipmap is a multi–resolution stack of textures, similar to a MIP–mapping pyramid [Wil83], storing the elevation data at different resolutions (i.e. the terrain seen from above, with varying degrees of precision). In a clipmap however, each level has the same resolution than the previous. Each increasingly coarser level thus covers a larger spatial extent. The clipmap follows the user as he moves within the scene. The user is thus always at the center of the clipmap, and sees high resolution data around him, while resolution decreases in the distance. In the original work, data is progressively loaded into the clipmap. When the user slowly moves, only a few rows and columns must be updated – those entering at the edges of the clipmap. We simply replace the data loader by our synthesizer. Our synthesizer very efficiently synthesizes the new rows and columns. This lets us explore an infinite terrain synthesized in real time from an example. Figure 3.9 shows a screenshot using the Puget Sound elevation data as an exemplar for terrain synthesis.

3.3.3 Layer–based texture synthesis

The scheme we proposed in Section 3.3.2 has proven efficient and versatile. It considers neighborhoods of colors, but also exploits pixel coordinates to prune the search space during texture synthesis. However, the storage of pixels coordinates during synthesis incurs a memory overhead. Furthermore, the technique only supports a single exemplar image. While it is possible to add a third coordinate to select between several exemplar images, this significantly increases memory and computational cost.

In an effort to further simplify this process, I came across a novel fast algorithm for synthesizing textures at runtime. While the scheme has never been published, I briefly described it during an invited talk at SIBGRAPI 2012. I take the opportunity of this HDR thesis to further describe it.

Rather than optimizing for pixel coordinates as in the geometric formulation, this new approach is similar in spirit to algorithms for Photomontage [ADA+04]: The synthesized texture is formed by selecting pixels amongst several possible layers of images. More precisely, we are given as input a set of K texture layers. Each texture layer covers the entire plane, for instance by an infinitely repeating tiling. A simple way to obtain the layers is by repeating a single texture in the plane with a different affine transform in each layer.

Let $I_l[p]$ be the color of the pixel at coordinate p in layer l. The output is a 2D array C of integers in $[0, K-1]$ selecting in each pixel from which layer its color should be copied. The array C is optimized so as to minimize:

$$\mathcal{E}(C) = \sum_{p \in C} \left(\sum_{\Delta \in \{-1,0,1\}^2} ||I_{C[p]}[p] - I_{C[p+\Delta]}[p]||^2 + ||I_{C[p]}[p+\Delta] - I_{C[p+\Delta]}[p+\Delta]||^2 \right) \tag{3.3}$$

Note that this energy bears strong similarity with Equation 3.2. However, instead of having a large number of

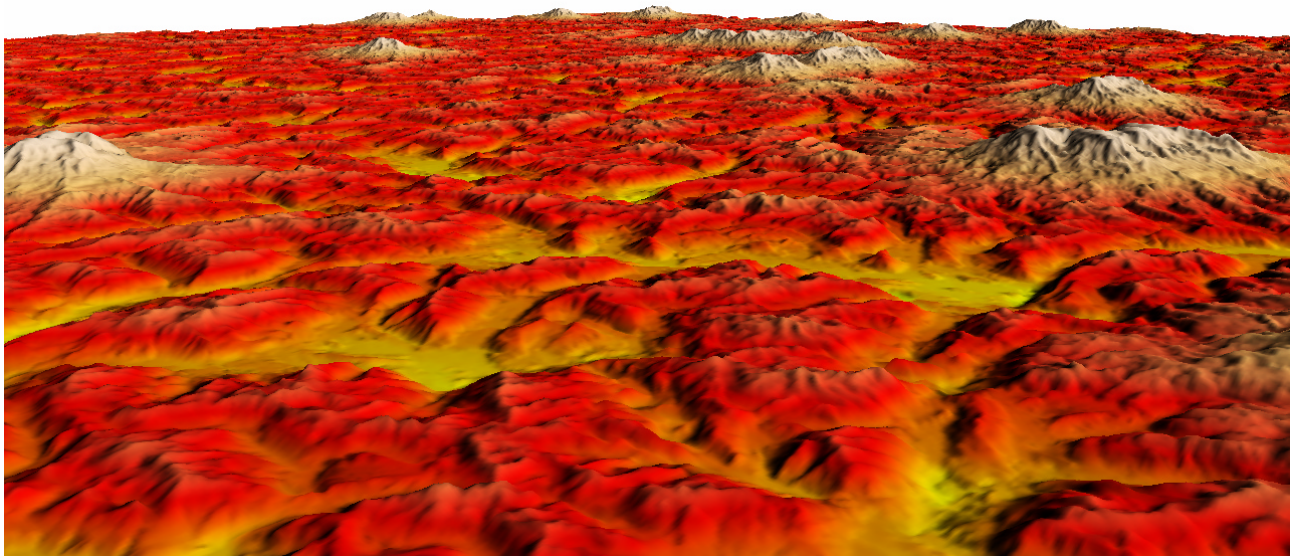


Figure 3.9: *Terrain synthesized at run–time in a clipmap hierarchy. Real–time synthesis speed was achieved in 2005 on top–of–the–line graphics hardware.*

labels L_{ij} we only keep a relatively small number of texture layers K. In a typical implementation the number of texture layer is set to $K = 256$ so that the array C is stored with only 8 bits per entry. This has important practical considerations compared to storing pixel coordinates which typically require at least 24 bits.

This energy may be optimized with techniques such as alpha-expansion [KSE+03]. However this would not provide fast run-time synthesis, since this requires several iterations of graph–cut (at least one per label). Instead, we use a simple scheme adapted from our parallel synthesizer (see Section 3.3.2). The scheme is multi–resolution: There is one array C_l for each resolution level l, where l increases with resolution. The coarsest resolution is randomly initialized with layer numbers in $[0, K-1]$. At each resolution we perform the same three steps:

- **Upsample**: The choice in each parent pixel is replicated for the children: $C_l(i,j) = C_{l-1}(i/2, j/2)$.
- **Jitter**: The choice in each pixel is randomly changed with probability τ_l, where τ_l is a user parameter controlling how much variation is desired in the result (it is set for each resolution level separately).
- **Correction**: A number of candidate layers $\mathcal{C}(p)$ is selected for each pixel. The candidates are the layers used by the 8 neighbors, as well as an equal number of random layer numbers. The candidate which results in the best local score according to Equation 3.3 is kept.

Each of these steps is performed in parallel on the GPU using pixel shaders in GLSL. Since they are very simple we can quickly perform a large number of iterations.

This scheme inherits all the benefits from our previous synthesizer, in particular the ability to generate quickly a window of an infinite texture plane. This is achieved through the use of a pseudo-random generator in each pixel, seeded by the output coordinates. Results of this approach are shown Figure 3.10. The last row illustrates a limitation: The technique is better suited to textures with high–frequency content, and performs worse on

smoothly varying features. Nevertheless, performance is very good and the implementation is simpler than prior schemes. The technique can easily be extended with several input exemplar, local control via a–priori selection of labels, as well as synthesis onto surfaces by considering planes in space surrounding the object.

3.3.4 Conclusion: how bad can the optimizer be?

Performance and memory footprints are important concerns for runtime texture synthesis. In many ways, coming up with a good algorithm for this purpose consists in finding an optimizer which is just good enough for the task, but simple and fast enough to run in real time. We have also seen that explicitly and progressively deviating from the optimal result – jitter on a tiling – greatly improves the efficiency of the optimizer compared to random initializations.

The formulations we proposed can be intuitively understood as asking the computer to create and solve a giant Jigsaw puzzle, where the shape of the puzzle pieces is free. In absence of structure and organization, such puzzles are easier to solve since well matching pieces are not rare. This is why, despite being local optimizers, the algorithms proposed in Computer Graphics for texture synthesis are so effective. The absence of structure makes the problem much easier to solve. Non-local structures make the problem significantly harder, and requires guiding the algorithms [RB07]. In Section 3.6 I will describe our synthesizer for structured textures. In contrast to the approaches described here, relying on an optimizer finding the optimal solution was key to the success of this approach.

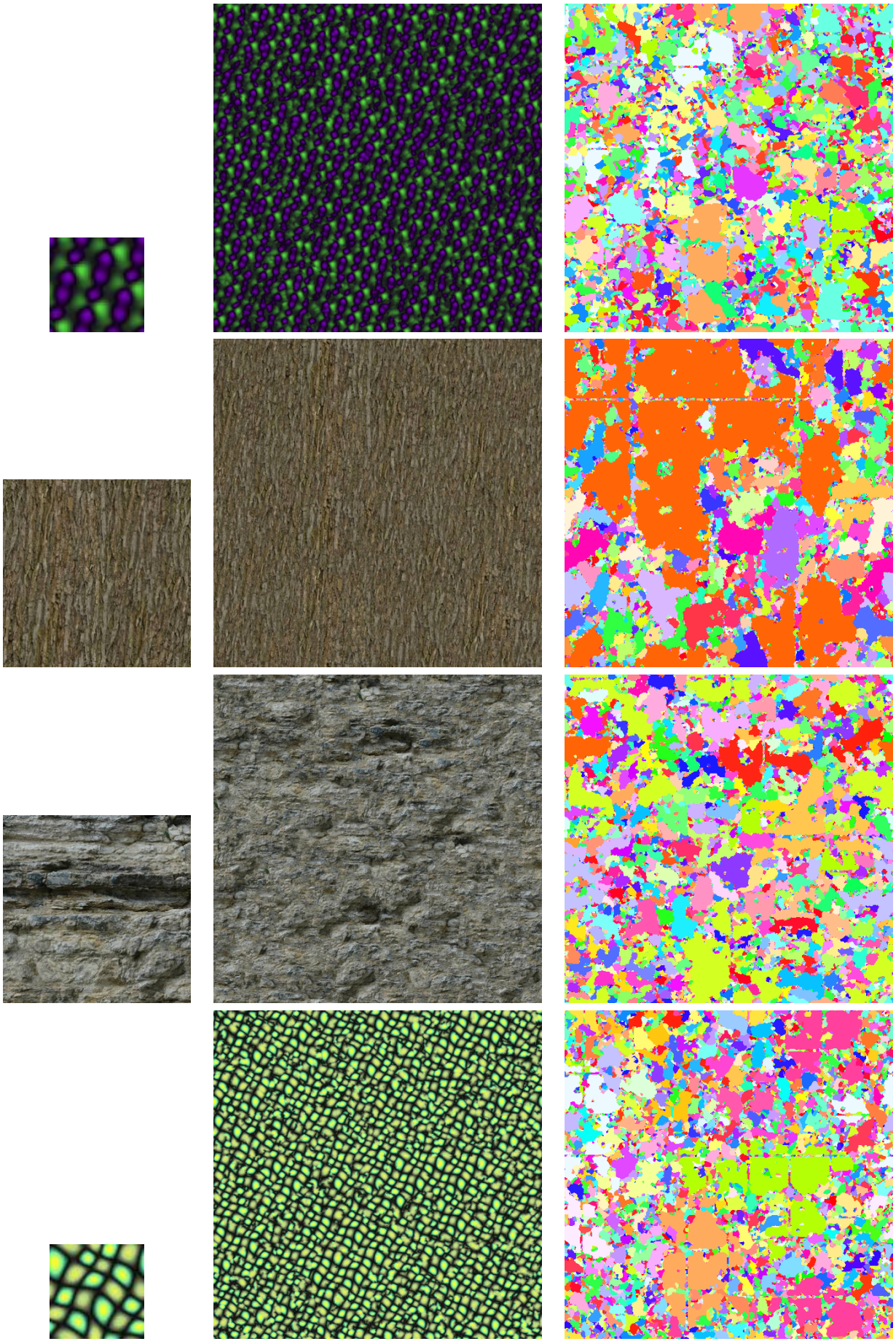


Figure 3.10: *Results from layer–based texture synthesis.* Left: *Example,* Middle: *our result,* Right *choice of layer in false color. Synthesis resolution is* 512^2*, synthesis time is below 10 ms on a GeForce GTX 480.*

3.4 Procedural synthesis of stochastic textures

> Proceduralism *as a discipline in computer graphics involves the abstraction of underlying form and/or behavior through time into a compact, elegant procedure or algorithm which can then manifest the form or changes when and where needed, i.e., through lazy evaluation. Large, detailed, explicit specifications of complex models are reduced to encodings in relatively short and simple routines. As the compliment of tablelookup schemes, very little explicit descriptive information is stored; rather, a specification for derivation of the data is developed, and that functional description is evaluated when and where specifics are required. Thus the overhead is shifted from storage space to computation time.* F. Kenton Musgrave, PhD dissertation, 1993

The concept of procedural content in Computer Graphics is intrinsically linked to the idea of run–time content generation. Instead of pre–generating and storing content, a compact procedure describing the content is written. For instance, in the case of a texture such a procedure *computes* the color at any given point, from a few parameters, in constant time. This idea has been successfully applied to model terrains, clouds, oceans, and a variety of texture appearances such as wood, marble and rocks [EMP+94]. Defining such procedures is however a difficult task, and there is no automated way to obtain them from examples in the generic case. Recently, solutions appeared however for specific classes of textures (see Section 3.4.3).

It is important to distinguish between procedural approaches and methods which are generating content offline. For instance the grammar systems used for the generation of buildings and plants [PLH88] are not capable of lazy evaluation and point wise evaluation of the content in constant time: the content is either entirely generated, or does not exist. Our work on run–time texture synthesis described Section 3.3 may be understood as bridging the gap between off–line by–example texture synthesis and procedural synthesis. However it is not entirely the case. First, our by-example synthesizer requires a relatively large amount of data to be stored: the exemplar image. Even though researchers attempted to reduce the size of the exemplar data [WHZ+08], this remains orders of magnitudes more than the few parameters of a procedural texture. Second, while synthesis is extremely fast, our algorithm requires a temporary memory buffer to compute the color of a single point. Therefore, it cannot be used as conveniently as a procedure, which is simply called at any point where content is required.

My focus in this area has been to improve the expressiveness of procedural textures, enabling new uses and simplifying their use. I describe our contributions in the following sections.

3.4.1 Fundamentals

> *Naturalistic visual complexity is built up by composition of nonlinear functions [...]. Powerful primitives are included for creating controlled stochastic effects [...].* Ken Perlin, [Per85]

Procedural textures have been initially conceived as a modeling tool, letting CG artists create patterns as rich and complex as those found in nature. A natural question which arises is how to model and control randomness.

Perlin first proposed to rely on *band–limited* noise functions. These functions produce random images with frequencies in a localized area of the power spectrum. Perlin proposed isotropic functions producing a ring of frequencies in the power spectrum. An example of such an ideal noise band can be seen Figure 3.15. A very important property of Perlin noise is that it is implicit: the noise value is quickly evaluated from the function, at any point in space, while producing an overall pattern with band–limited frequency.

These functions can be used as building blocks for more complex patterns: several independent weighted bands are added together. An example is given Figure 3.11. These noises are typically used to perturb a color function, producing complex visual effects such as a texture of wood or marble, as illustrated Figure 3.12.

Since Perlin, several noise functions have been proposed, essentially improving the spectral properties of the band–limit. I refer the interested reader to our survey on the topic [LLC+10]. In the following section I focus on a different type of procedural noises, obtained by procedurally combining many small primitives together.

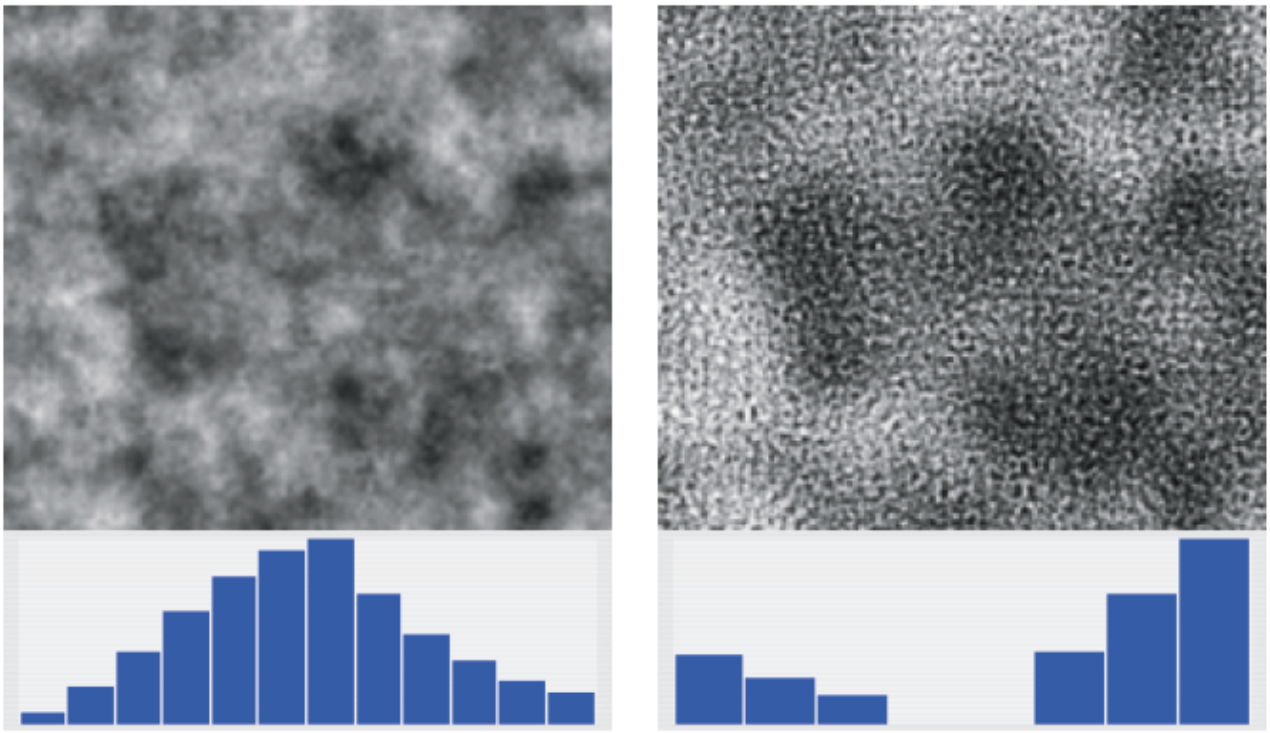


Figure 3.11: *Different patterns obtained by summing band–limited noises of different scales. The weight of each band is graphically shown as the blue histogram.*

3.4.2 Impulse textures

In this section I discuss our work around procedural textures and noises. We focused on a specific class of procedural textures, obtained from the combination of many randomly positioned kernels. I introduce below the fundamental ideas, and later discusses both our early work on the topic as well as our most recent advances.

3.4.2.1 Sparse convolutions

Textures obtained by sparse convolution have been introduced to Computer Graphics by Lewis [Lew89] and van Wijk [vW91]. The core idea of a sparse convolution is to define the texture as the summation of many *kernels* of limited spatial extent. The kernels are positioned following a random impulse process.

Formally, a sparse convolution noise N is written as the convolution of a kernel h and a Poisson impulse process:

$$N(x,y) = \left[h * \sum_i w_i \delta_{(x_i,y_i)} \right](x,y), \tag{3.4}$$

where $*$ denotes the convolution, where δ is the impulse and where $\delta_{(x_i,y_i)}(x,y) = \delta(x - x_i, y - y_i)$. The Poisson impulse process consists of impulses of uncorrelated intensity $\{w_i\}$ at uncorrelated locations $\{(x_i,y_i)\}$. It is characterized by the mean number of impulses per unit area, noted λ.

The sparse convolution takes its name from the fact that the process in Equation 3.4 is similar to convolving the kernel with a (sparse) white noise. As a consequence, the power spectrum of a sparse convolution is that of the kernel scaled by a constant, which follows from the Convolution Theorem. We exploit this property in our work on Gabor noise, presented Section 3.4.2.3.

Sparse convolutions can be procedurally evaluated provided that the kernel has a finite support. The evaluation, as described by Lewis [Lew89], consists in considering a grid overlaid on the texture domain. The grid is chosen so that the size C of one cell matches the diameter of the kernel support. The Poisson process is obtained by seeding a random generator, in each cell, with the coordinates of the cell inside the grid. A number of impulses are then randomly positioned inside each cell. The number of impulses per cell has to follow a

Figure 3.12: *A texture resembling marble (left) is obtained from a color ramp in space (middle) perturbed by a noise pattern (right). The texture is obtained by evaluating the procedure at every point along the vase.*

Poisson distribution with mean λ^2. When evaluating the value of the sparse convolution in a given point, it is enough to consider the impulses within the 3×3 cells surrounding the evaluation point. Indeed, all other kernels are too distant to have an influence. This locality enables a procedural evaluation. The sampling algorithm is given below:

```
1  float sparseConvol(u,v)
2   i = ⌊u/C⌋, j = ⌊v/C⌋
3   value = 0
4   for ni = i−1 ... i+1
5    for nj = j−1 ... j+1
6     seedRandom(ni,nj)
7     N = poissonDistribution (λ)
8     for k=0 ... N−1
9      s,t = randomImpulse(ni,nj,k);
10     w   = randomWeight(ni,nj,k);
11     value = value + w · kernel(u−s,v−t)
12    end
13   end
14  end
15  return value;
16 end
```

`poissonDistribution` generates a random integer following a Poisson distribution of mean λ.
`randomImpulse(ni,nj,k)` generates the k-th random coordinate within cell i, j.
`randomWeight(ni,nj,k)` generates the k-th uniform random weight in $[-1, 1]$. All these random generators are *pseudo–random*: after a call to `seedRandom` with same parameters they generate the exact same sequence of uncorrelated numbers. In particular, a given i, j cell always generates the same sequence of weighted impulses.

In the next section I describe our early scheme based on a similar idea of generating textures from a random spread of basic elements.

3.4.2.2 Distribution of texture elements

In 2003 we proposed a first hardware implementation of a mechanism similar to sparse convolution [LN03]. Implementing such a mechanism on modern graphics hardware is a relatively trivial task. However, due to very

[2] In practice it is often chosen as a fixed number for performance concerns – this has little impact on the quality of the noise

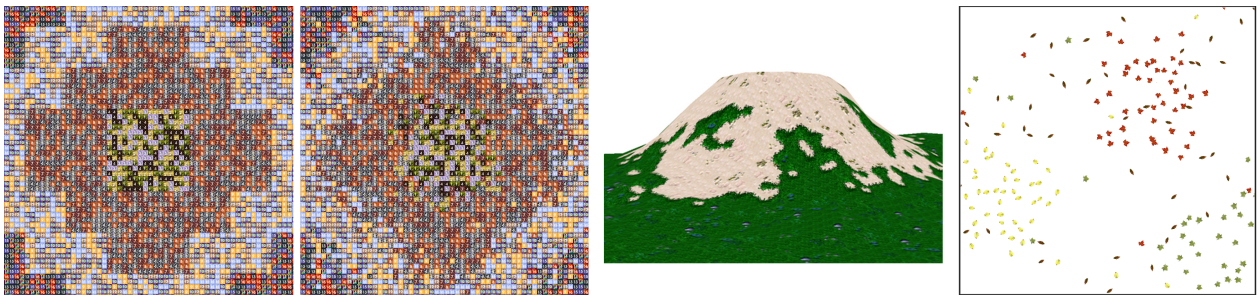


Figure 3.13: *From left to right:* First: *Random choices of tiles, with different histograms.* Second: *Same with interpolation between histograms.* Third: *Technique applied to generate random tilings between different materials.* Fourth: *Technique applied for random placement of texture elements (yellow, green, brown and red leaves).*

strong hardware constraints, achieving this back in 2003 was a challenging task[3].

We demonstrated how to exploit the hardware to generate random tilings as well as random placement of texture elements. The main interest in doing so is to achieve a truly procedural placement: the only cost was to store the base texture elements in memory. In addition, our scheme offers various controls over the generated random choices: histogram control and interpolation, interactive cut–and–paste, automatic choice of tiles to transition between different materials. These are illustrated Figure 3.13. I later extended these ideas to produce textures animated in real–time, such as drops of water flowing along a surface [Lef03]. This work introduced the idea that the texture itself could contain animated elements. This later led to the *texture sprites* [LHN05b] and to the filtering technique employed in Gabor noise [LLDD09].

The tiling approach we proposed led to more elaborate schemes for hardware accelerated random tilings [Wei04], including tilings with correct texture filtering [Lef08].

3.4.2.3 Gabor noise

In Gabor noise [LLDD09] we proposed a novel approach for synthesizing noise patterns, based on sparse convolution (Section 3.4.2.1). Perhaps the biggest advantage of Gabor noise is that it offers a precise control upon the power spectrum of the synthesized noise. In fact, the control is so precise that it affords for on–the–fly filtering (Section 3.5.3.2) and by–example procedural synthesis (Section 3.4.3).

I give here the definition of *random-phase* Gabor noise [LD11], which differs from the original in that the phase is explicitly randomized. This lets noise be filtered across dimensions, such as when slicing a 3D noise with a 2D surface.

Two-dimensional anisotropic random-phase Gabor noise is defined as

$$n\left(\mathbf{x};K,a,\omega\right)=\frac{1}{\sqrt{\lambda}}\sum_i g\left(\mathbf{x}-\mathbf{x}_i;K,a,\omega,\phi_i\right), \tag{3.5}$$

where K, a and ω are the amplitude, bandwidth and frequency of the noise, g is the phase-augmented Gabor kernel (defined below), the random positions $\{\mathbf{x}_i\}$ are distributed according to a Poisson process with mean λ, and the random phases $\{\phi_i\}$ are distributed according to a uniform distribution on the interval $[0,2\pi)$.

The phase-augmented Gabor kernel is defined as

$$g\left(\mathbf{x};K,a,\omega,\phi\right)=Ke^{-\pi a^2|\mathbf{x}|^2}\cos\left(2\pi\mathbf{x}\cdot\omega+\phi\right), \tag{3.6}$$

where K, a, ω and ϕ are respectively the amplitude, bandwidth, frequency and phase of the kernel.

The variance of the noise is $K^2/4a^2$.

The power spectrum of the noise is

$$S_n\left(\xi;K,a,\omega\right)=\frac{K^2}{8a^2}\mathcal{G}\left(\xi;\pm\omega,\frac{a}{2\sqrt{\pi}}\right), \tag{3.7}$$

[3] At the time there were no programming languages for GPUs and we relied on hardware extensions called *register combiners* and *offset textures*.

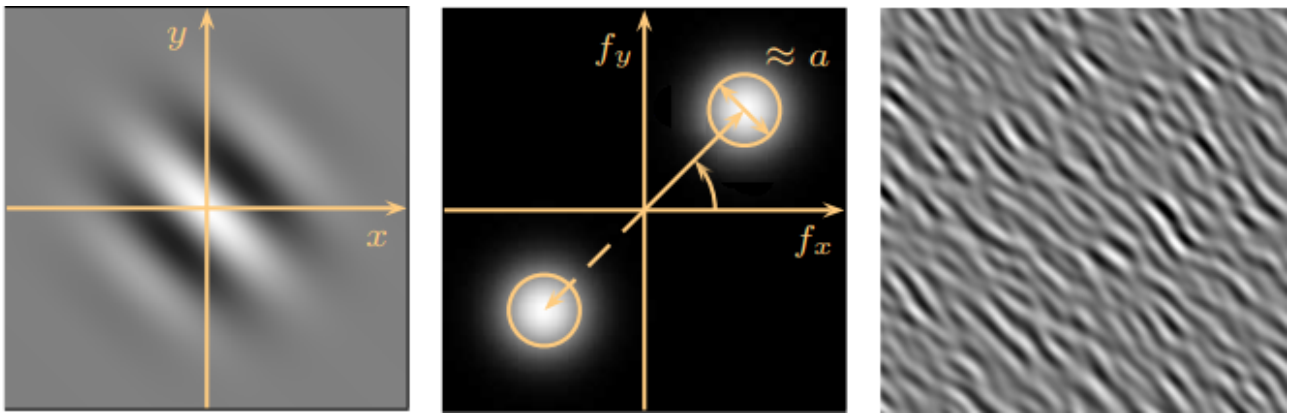


Figure 3.14: Left: *Gabor kernel in the spatial domain: the product of a sine wave and a Gaussian.* Middle: *Power spectrum of the Gabor kernel.* Right: *Gabor noise pattern from a single kernel.*

where $\mathcal{G}(\mathbf{x}; \mu, \sigma)$ is the 2D normalized Gaussian function with mean μ and standard deviation σ, and $\mathcal{G}(\mathbf{x}; \pm\mu, \sigma)$ is a shorthand notation for $\mathcal{G}(\mathbf{x}; \mu, \sigma) + \mathcal{G}(\mathbf{x}; -\mu, \sigma)$. The power spectrum of Gabor noise is thus a symmetric pair of Gaussians in the frequency domain with magnitude K, frequency $\pm\omega$ and bandwidth a. Figure 3.14 illustrates a Gabor kernel, its power spectrum, as well as the noise obtained by using it in the sparse convolution process.

Mixing Gabor kernels An important property of Gabor noise is that it is possible to generate complex spectra by mixing different Gabor kernels. For instance, choosing a random orientation for each kernel during evaluation leads to an isotropic noise, as illustrated Figure 3.15, left.

Another way to understand this procedure is simply by considering that we are summing layers of uncorrelated noises having different orientations. The final power spectrum is the sum of all spectra. However, this is not equivalent: by performing a random sampling we avoid an explicit discretization of orientation, and therefore we avoid the associated bias.

We exploited this idea to create noise *widgets*, shown Figure 3.16. These widgets let us manually control the final noise spectrum. Each widget describes a range of parameter values from which the bandwidth, orientation and frequency of each kernel is randomly sampled. Combined with color maps, this let us create interesting and complex texture patterns, as shown Figure 3.17.

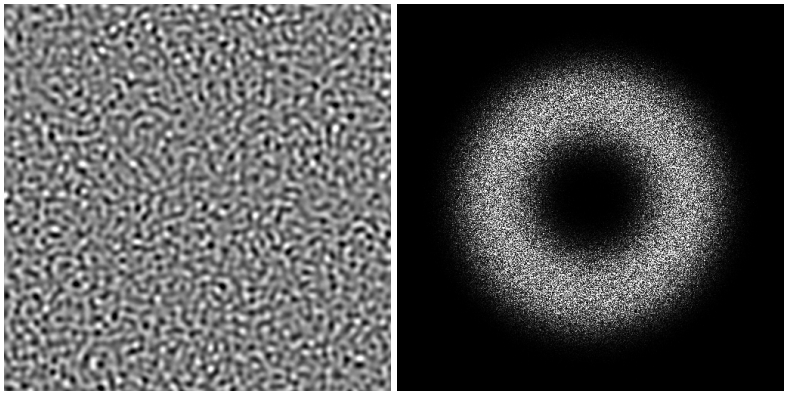


Figure 3.15: Left: *Gabor kernels are selected with a random orientation, producing an isotropic pattern with no preferred orientation.* Right: *The periodogram clearly shows the bandlimited isotropic property of the noise.*

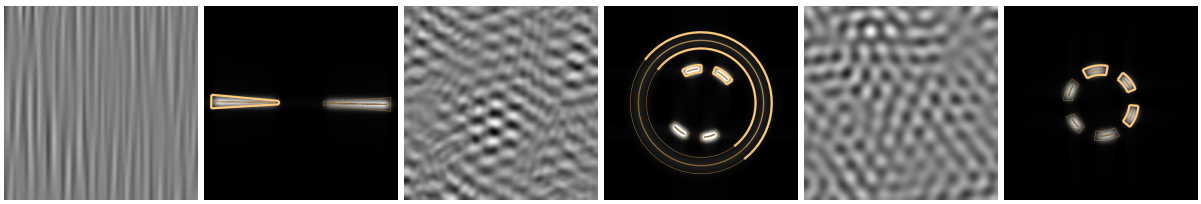


Figure 3.16: From top to bottom: *A noise pattern on the left is obtained by randomly sampling the parameters of Gabor kernels as specified by the widget on the right.*

Figure 3.17: *Using a combination of Gabor noise patterns and color maps complex textures can be created. See Section 3.5.3.2 for more details on how the noise is applied to the surfaces.*

3.4.3 By–example procedural textures

The main difficulty in using the aforementioned methods for generating noise remains the choice of parameters. The link between the parameters, the noise pattern, the color map and the final texture is not straightforward. While experienced artists can get used to these notions, it requires significant training and experience.

Therefore there has been a lot of effort since the early days of procedural textures to automatically select parameters of a noise generator, given an example texture. Of course, this raises a key question: can we characterize the set of textures for which it is indeed possible to find parameters properly reproducing its appearance?

3.4.3.1 By–example noises

Ghazanfarpour and Dischler [GD96] presented a method for the automatic generation of 3D textures using spectral analysis of two or three 2D slices of a 3D texture, based on an earlier method [GD95a] that only takes into account a single 2D texture. The approach is based on the selection of a small number of high-magnitude peaks in the Fourier transform, later resynthesized by a summation of cosine waves. The same authors later introduced a method for automatically generating a geometric texture from a 1D profile [DG97]. The approach is however limited to isotropic patterns.

Lagae et al. [LVLD10] presented a method for procedural isotropic stochastic textures by example. The approach relies on a multi-octave wavelet noise model, computing weights from an exemplar. Xue et al. [XDR11] presented a similar method for simulating rough appearance for stone weathering in a photograph. Both methods only address isotropic patterns.

Gilet et al. [GDS10] presented a method to generate procedural descriptions of anisotropic noisy textures by example. A Gabor noise model with a small number of Gaussians and several cosines is fitted to the periodogram of the example. This was later extended to synthesize stochastic volumetric and procedural details [GD10]. The main drawback of this approach is that it still requires significant manual intervention and trial and error for each exemplar.

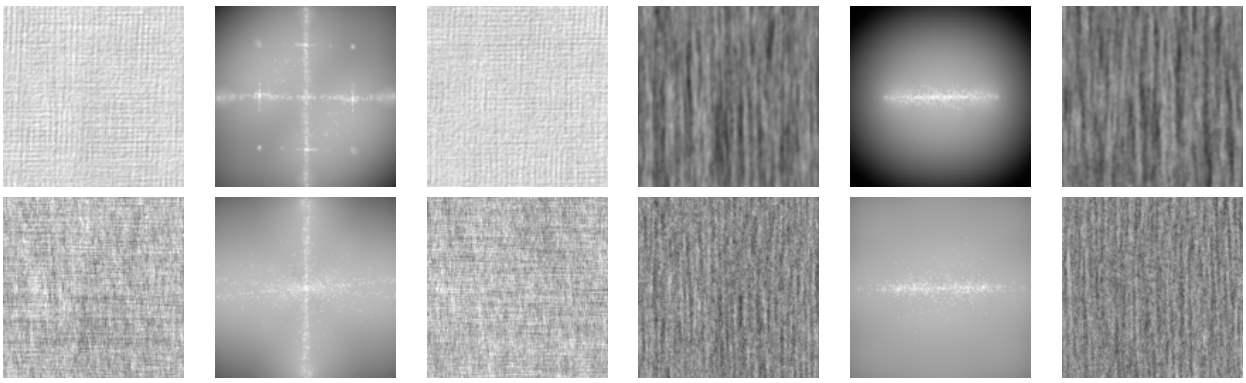


Figure 3.18: *Example of Gaussian textures. For each triple of images, left is input, middle is power spectrum, right is the Gaussian version after phase randomization.*

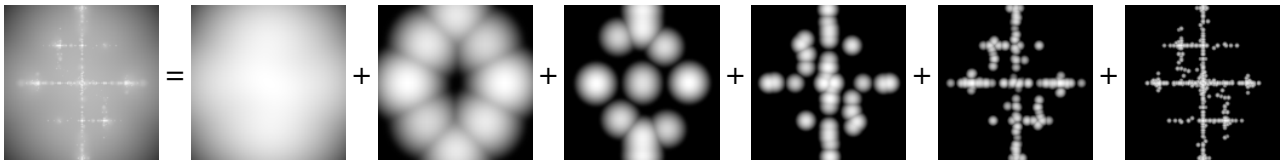


Figure 3.19: *The power spectrum on the left is a weighted sum of the Gaussians on the right. Note how the Gaussians are grouped by bandwidth. The weights are automatically optimized by our algorithm so as to obtain a sparse solution.*

For most of these approaches, it remains unclear which specific class of textures can be properly captured. In the next section I describe our work on Gabor noise by example [GLLD12]. In this work we characterize a class of textures that can be properly represented by Gabor noise, as well as a technique to fit the noise to an example image.

3.4.3.2 Gabor noise by Example

A very important question when considering procedural texture by example, is to decide upon a class of textures to consider. While it is generally very difficult to precisely characterize textures, at least one category is better understood than others: *Gaussian textures*.

Gaussian textures Gaussian textures are defined by what is known in statistics as a Gaussian random field [PP02]. Such random fields are completely characterized by their power spectrum – implying that the phase contains no information. Intuitively, these are textures containing no precisely defined contours or edges. Visually, they roughly correspond to the class of stochastic textures.

Galerne et al. [GGM11] presented methods that can be used to generate the Gaussian version of a texture. These methods explicitly destroy the information contained in the phase of the spectrum, only preserving the modulus part (the power spectrum). This may be used to produce the Gaussian version of any input texture: the original texture from which the phase information has been randomized. If the input texture falls within the Gaussian texture category, this operation will not adversely impact its appearance. Otherwise, the visual content will significantly change. Figure 3.18 illustrates the Gaussian version of a number of textures.

The model of Gaussian textures ideally fits noise representations in Computer Graphics: most noises such as Anisotropic noise [GZD08] and Gabor noise [LLDD09] seek to generate procedural textures with arbitrary power spectra, ignoring the phase. Most of the noise generators indeed produce patterns with random phases.

We applied these ideas to Gabor noise. It is ideally suited to this task since the final power spectrum is represented by a sum of Gaussians, each corresponding to a choice of parameters for Gabor kernels. We built upon this idea to approximate any arbitrary power spectrum as a sum of weighted Gaussians, as illustrated Figure 3.19. The spectrum may be formed by summing from a large number of Gaussians (in principle Gaussians of any bandwidth centered at all frequencies). The weights of the Gaussians are the unknowns of the problem.

In this process, care must be taken to ensure that only a limited number of Gaussians are used to reconstruct the power spectrum – otherwise we would loose the property of a compact representation of the texture. For this

Figure 3.20: *Results of our approach on colored textures. Each pair of images has the input on the left and the image synthesized with Gabor noise on the right.*

reason, we formulate the optimization as a *basis pursuit denoising* problem, which encourages the solution to be sparse: most of the Gaussians receive a null weight. A parameter lets the user choose between accuracy and compactness. The details of this process can be found in our publication [GLLD12].

The result of the optimization is a compact representation of the Gabor noise, from which Gabor kernel parameters are sampled. This is akin to the widgets introduced Section 3.4.2.3. The noise obtained from the example therefore *is* a Gabor noise and inherits all its properties: it is evaluated on–the–fly in each point, on the GPU.

3.4.3.3 From noise to colored textures

Noise generators only produce scalar fields of (controlled) random values. Addressing color is not straightforward in general since in most cases the color channels are strongly correlated.

Since the seminal work of Heeger and Bergen [HB95] a typical approach is to decorrelate the color channels and apply synthesis independently to each. This amounts to computing the principal component analysis (PCA) transform that will *linearly* decorrelate the colors. However, removing linear correlation does not make the color channels *independent*. It is enough to consider RGB color values taken on a sphere in the RGB space to see this: the PCA will have no effect while the RGB channels are strongly linked. This implies that any complex, non–linear dependency will remain. Failure to capture these dependencies result in color artifacts when independently synthesizing the color channels. An example of such a failure is described in the work of Kopf *et al.* [KFCO$^+$07].

Nevertheless, this decomposition is acceptable in a number of cases. In particular, for Gaussian textures it is often a reasonable approximation. A perhaps better decomposition is to search for a *maximally independent* color space, as described in our Gabor noise by example paper [GLLD12]. While this is more general than PCA from a theoretical point of view, it unfortunately only provides limited benefits in practice. Figure 3.20 gives color results of our technique.

An interesting question related to this problem would be to automatically decompose a texture into an arbitrary color map and a Gaussian noise.

3.5 Texture synthesis along surfaces

So far we discussed texture synthesis only in the context of images. However, textures are often applied to non–planar surfaces. In this section I first recall important notions of texturing, and then I describe how we adapted both by–example synthesis and procedural Gabor noise to perform directly on surfaces. This section assumes that the reader is familiar with the algorithms introduced in previous Sections.

3.5.1 Background on texturing

Illusion is needed to disguise the emptiness within. Arthur Erickson

Texturing refers to a range of technique creating an appearance along the surface of an otherwise smooth object. The most common technique consists in wrapping an image around the object surface (Section 3.5.1.1), but other techniques plunge the surface into a 3D volume of matter (Section 3.5.1.2)

3.5.1.1 Texture mapping in a nutshell

There are excellent surveys and courses on texture mapping, in particular I strongly encourage anyone interested in the topic to read the masters thesis of Heckbert [Hec86]. I will thus only briefly recall the necessary tools to understand the challenges in synthesizing textures directly on surfaces.

Texture maps are applied to objects through a planar parameterization. A planar parameterization maps triangles from the object space onto the image plane. From any point within the triangle, barycentric coordinates may be used to obtain a point in the image plane, and thus a color. In effect this paints a part of the image onto the triangle. This operation is a core component of any graphics API and is fully accelerated by graphics hardware.

It is important to understand that the image is painted along the triangle very late in the rendering pipeline, and in particular after the geometry has been processed: the visible facets and pixels have been determined through visibility culling and z-buffer test. This makes texture mapping very efficient: we only pay the price for what we actually see on screen.

Computing a planar mapping Good planar mappings have low angular and area distortion. Unfortunately these goals are antagonist and a compromise must be found. A very popular approach to make the problem easier is to cut the shape into several pieces, called *charts*, and parameterize each independently in the plane. The planar charts are then packed together to form the final texture space. Such a construct is called a *texture atlas* [MYV93, LPRM02]. Parameterizing charts is easier as cutting releases some of the constraints, at the expense of discontinuities (cuts) in the mapping. An interesting extreme case consists in splitting the mesh into single–triangle charts: each triangle is trivially parameterized in its own plane [CH02]. As we shall see these discontinuities are unfortunately not without consequences and their number has to be kept low.

Computing good planar parameterizations is an important research area and powerful automatic methods have been proposed. A review or description of parameterization techniques falls outside the scope of this document, and we refer the interested reader to the survey by Floater and Hormann [FH05]. In practice CG artists often perform this process semi–manually with the assistance of specialized tools, since they prefer to enforce the symmetries and semantics of the object: they later have to paint the surface appearance into the image plane.

Filtering Filtering deals with situations where the texture map is either oversampled or undersampled by the screen pixels. Oversampling typically occurs during closeups on an object surface: there are many screen pixels to display only a few texture pixels. Undersampling occurs when objects are in the distance: many texture pixels project in only a few screen pixels.

Filtering is essential to image quality: most of a rendered image is covered by textures. While geometric aliasing is difficult and expensive to achieve, texture filtering is in contrast relatively simple, thanks to the explicit mapping between the surface and the image plane. The typical techniques for filtering are bi–linear interpolation and MIP-mapping [Hec86, Wil83]. Both are natively supported by the graphics hardware. However, any deviation from standard texture mapping will break these mechanisms. Therefore, in most of our techniques we have to consider the problem of achieving proper filtering.

3.5.1.2 Parameterization–free texture mapping

Texturing is possible without having to resort on a texture map and a planar mapping. A typical approach is to consider that the surface is carved out a solid volume of matter [PH89]. The texture is therefore a 3D block of colors. Of course, storing such a block would be impractical: a 1024^3 volume texture already requires 3 GB of memory, and would produce a relatively small resolution along a surface. If we disregard this problem, texturing through a volume has a key advantage: the reliationship between the surface and the texture is straightforward. In particular there are no discontinuities due to cuts in the surface. Area and angular distortions are also limited and, in most cases, can be ignored.

In order to enable this methodology for on–the–fly synthesis we developed several data–structures to efficiently store and access volume texture data around surfaces. This is the topic of Chapter 4, and thus in the remainder of this section we will assume that this approach is practical.

3.5.2 Data–driven synthesis on surfaces

We explored two main approaches for data–driven synthesis along surfaces. Both take 2D exemplar images as input. However, while the first approach seeks to generate a 2D color field along the surface, the second defines a full volume of 3D matter.

3.5.2.1 Synthesizing texture atlases

The basic elements for synthesizing directly into a texture atlas were introduced by Yin *et al.* [YHBZ01] for subdivision surfaces. We generalized this approach to standard texture atlases in our 2006 paper *Appearance space texture synthesis* [LH06a]. The key idea is to synthesize colors directly into the texture atlas, ensuring that once mapped onto the surface the correct appearance is achieved. The space in the texture atlas is both distorted and discontinuous. Distortion is due to necessary tradeoffs in the mapping between angle and scale preservation. Discontinuities are due to the cuts in the surface introduced by the planar parameterization (Section 3.5.1.1).

For this reason, we cannot simply synthesize a large image and use it as the texture atlas: while the image is undistorted and continuous in texture space, it will appear distorted and discontinuous once seen along the surface through the mapping. Instead, the synthesized image has to compensate for these defects: we have to *introduce* a distortion into the texture we synthesize so that once mapped on the surface it will *appear* undistorted. We introduce in fact the inverse distortion of the mapping: where the mapping stretches the texture space, we shrink the appearance so that features maintain their size. Similarly, the texture can be synthesized so that the appearance on each side of a discontinuity appears continuous along the surface. To achieve this, it is necessary to distinguish between the (undistorted) *exemplar space*, the *surface space*, and the *texture space* which is subject to distortions and discontinuities. The surface space is defined by the user as a surface vector field t, b of tangent and binormal vectors. This lets the user additional specify the natural scale and orientation of features he wishes to synthesize along the surface.

As explained Section 3.3.1, texture synthesizers operate by re-assembling small parts of an exemplar – down to the pixel – while *locally* preserving its stochastic appearance. This absence of global features in the exemplar is what enables synthesis under distortion: since we introduce distortions locally, the synthesizer only has to locally correct the appearance to achieve a globally consistent result.

Distorted neighborhoods The first ingredient in synthesizing along surfaces is to distort the neighborhoods sampled from the texture. The neighborhoods are sampled distorted, but are still being compared without distortion in the exemplar. This direct consequence is that the synthesized textures appear distorted. A simple scenario to understand the idea is to consider for distortion a 90 degree rotation: all neighborhoods in texture space will be sampled with a 90 degree angle, effectively synthesizing a rotated version of the exemplar. There is no reason for the distortion to be constant across texture space, and in practice we compute the distortion from the texture mapping, in each pixel.

Our goal is to synthesize textures anisometrically in the parametric domain such that the surface vectors t, b (tangent and binormal) are locally identified by the exemplar space axes $\hat{u}_x, \hat{u}_y$. We note D the texture space, M the surface space, E the exemplar space. From Figure 3.21 we see that $(t, b) = J_f J^{-1} I$, where J_f is the

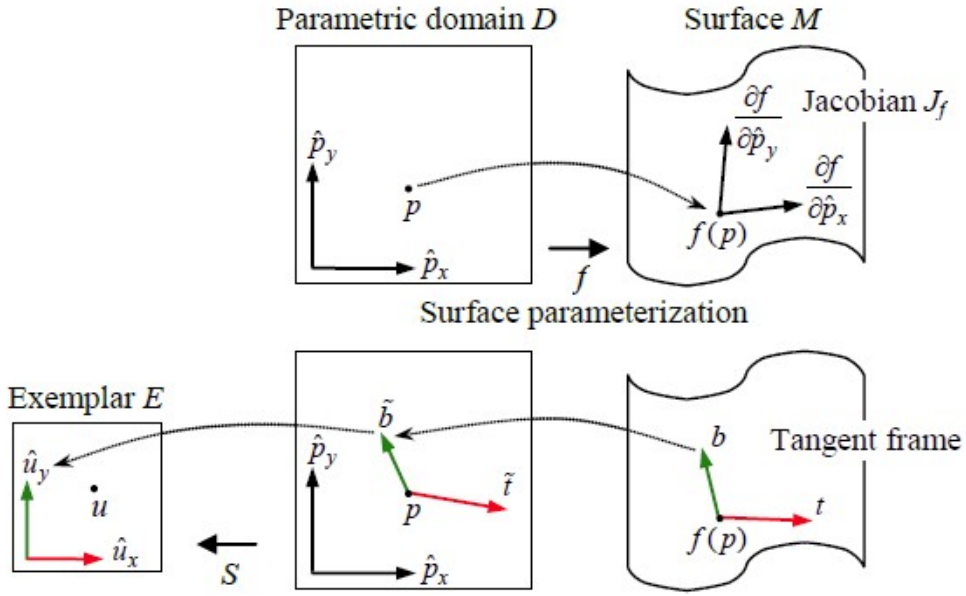


Figure 3.21: *The different spaces used by anisometric texture synthesis*

3×2 Jacobian of the surface parameterization $f : D \to M$, and J the desired 2×2 Jacobian for the synthesis distortion $S : D \to E$. It follows:

$$J = (tb)^{+} J_f = ((tb)^T (tb))^{-1} (tb)^T J_f$$

where $^+$ denotes matrix pseudoinverse. The tangential frame as well as the mapping vary per-pixel in the texture space. Note that J_f is rarely available under analytical form, but is well approximated by finite differences form the map of 3D surface points in texture space.

Sampling neighborhoods We follow the energy minimization point of view described Section 3.3.1.5. We modify Equation 3.2 to incorporate J. In the following and for the sake of clarity we only describe how single neighborhoods are modified. We define an undistorted neighborhood in D at point p as:

$$N_S(p; \Delta) = E[S[p + \Delta]]$$

A first approach would be to sample distorted neighborhoods, similarly to Ying *et al.* [YHBZ01]. Distorted neighborhoods are then obtained as:

$$N_S(p; \Delta) = E[S[p + J^{-1}(p)\Delta]]$$

That is, the sampling pattern in synthesis space is transformed by the inverse Jacobian at the current point. However, such a transformation requires filtered resampling since the samples no longer lie on the original grid. More significantly, if the Jacobian has stretch (i.e. spectral norm greater than unity), the warped samples become discontiguous, resulting in a breakdown in texture coherence.

Instead, we seek to *estimate* an anisometrically warped neighborhood vector $N_S(p)$ by only accessing immediate neighbors of p. We use the direction of each offset vector $\varphi(\Delta) = J^{-1}(p)\Delta$ to infer which neighboring pixel to access, and then to use the full offset vector $\varphi(\Delta)$ to transform the neighbor's synthesized coordinate.

More precisely, we gather the appearance vector $N_S(p; \Delta)$ for each neighbor as follows. We normalize the Jacobian transformed offset as $\delta = \hat{\varphi}(\Delta) = |\frac{\varphi(\Delta)}{||\varphi(\Delta)||} + 0.5|$, which keeps its rotation but removes any scaling. Thus $p+\delta$ always references one of the 8 immediate neighbors of pixel p. We retrieve the synthesized coordinate $S[p + \delta]$, and use it to predict the synthesized coordinate at p as $S[p + \delta] - J(p)\delta$, adjusting for anisometry. Finally, we offset this predicted synthesized coordinate by the original exemplar-space neighbor vector Δ. The final formula is

$$N_s(p; \Delta) = E[S[p + \delta] - J(p)\delta + \Delta]$$

where $\delta = \varphi(\Delta)$. It is interesting to note that in the above expression choosing any small δ would still provide a correct estimation. In practice we average between several estimations taken around $p + \delta$ for improved stability. Please refer to the original paper.

We also modify the k-coherent search to account for anisometry, picking candidates as:

$$\mathcal{C}(p) = \{C_i(S[p + \Delta]) + S[p + \Delta] - J(p)\Delta | i = 1...k, ||\Delta|| < 2\}$$

Indirection maps To form a seamless texture over a discontinuous atlas, the synthesis neighborhoods for pixels near chart boundaries must include samples from other charts. Here we exploit the property that our anisometric synthesis estimates a neighborhood from the direct neighbors of a pixel in the synthesized texture.

We read samples across charts using an indirection map I, replacing each access $S[p]$ with $S[I[p]]$. These indirection maps depend only the surface parameterization, and are precomputed by marching across chart boundaries. We reserve space for the necessary 2-pixel band of indirection pointers around each chart during atlas construction.

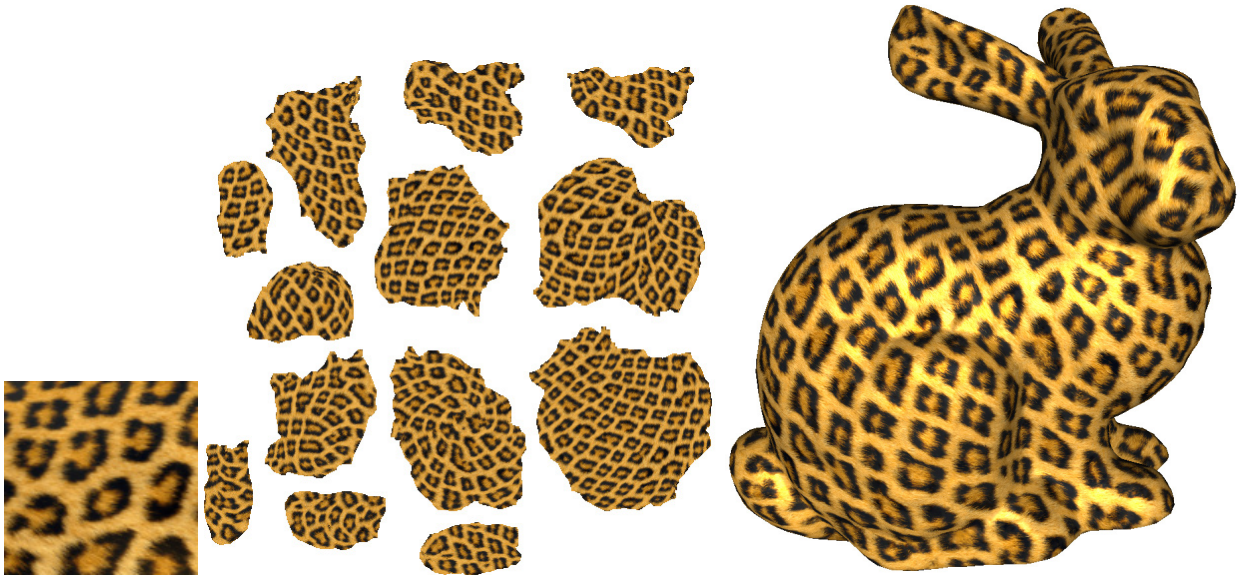


Figure 3.22: Left: *Exemplar texture provided by the user.* Middle: *Texture atlas synthesized automatically. Note the distortion performed by the synthesizer.* Right: *Once mapped onto the surface the texture appears undistorted and continuous, despite the multiple charts and distortions of the mapping.*

3.5.2.2 Synthesizing a volume texture

In Section 3.5.1.2 we explained how volume textures may be used to texture a surface. This approach can be used for texture synthesis on surfaces. A first methodology would be to synthesize a full volume of colors. The by-example synthesizer we discussed before could be trivially adapted from 2D to 3D, extending all formulations with a third dimension.

However, contrary to the 2D case, obtaining 3D examples of texture data is not easy. Even if this may be achieved through destructive layer-by-layer scanning processes, this would only apply for real materials. It is unreasonable to expect CG artists to *paint* volumes of textures.

Instead, several authors proposed to generate volume of colors by synthesizing in 3D from 2D examples [Wei02, QY07, KFCO$^+$07]. The core idea is to optimize the 3D neighborhoods so that they remain similar to 2D neighborhoods *along specific directions*: a 2D exemplar is used for the X,Y slice, and two other for the Y,Z and X,Z slices. Of course, this process can only work well if the provided example images are compatible: inconsistent 2D features in different directions will prevent the synthesizer to form properly defined 3D features.

One drawback of these approaches, however, is that they can only generate a full volume of color data. This is especially unfortunate when the main purpose is in fact to texture a surface: synthesizing a full volume is very expensive due to the cubic increase in size with resolution, whereas the surface only uses a small portion of this

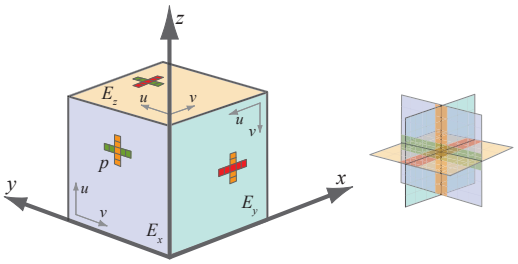


Figure 3.23: *Each candidate consists of three exemplar coordinates defining a 3-neighborhood. Consistent triples have low color differences along the crossbar. We seek to minimize the color differences along the three pairs of pixel strips, shown here with the same color.*

information. We contributed in this area by proposing a novel scheme able to quickly synthesize the volume texture *only around the surface*, at run-time. The result is equivalent to first synthesizing the volume and then texturing the surface. This work was a collaboration with Microsoft Research Asia and has been published in 2008 [DLTD08].

Overview The key idea making our approach possible is to pre-compute a small number of carefully selected *3D candidates*, later used in a volume synthesis algorithm. Each candidate is formed by interleaving three well-chosen 2D neighborhoods from the example images.

Once we have these 3D candidates, everything happens as if we had been given a 3D example instead of separate 2D images. We can therefore adapt our parallel synthesizer for volume synthesis. The synthesizer requires several changes so as to efficiently generate sparse voxels. One such change is to rely on a spatial data–structure – the TileTree – that we had introduced in prior work. I will describe this data–structure in Chapter 4 (Section 4.1.2).

Building 3D neighborhoods from 2D examples In each pixel of each exemplar we compute a small candidate set of 3-neighborhoods represented as coordinate triples (see Figure 3.23). These sets are used during synthesis as candidates for best matching neighborhoods. Hence, they must capture the appearance of the 3D neighborhoods implicitly described by the input images.

Given exemplars E_x, E_y, E_z (on in each major direction), the search space for possible candidates is huge: it contains all possible triples of coordinates. The major difficulty is that we cannot explicitly test whether a candidate triple forms a neighborhood representative of the not-yet-synthesized volume. This information is only *implicitly* given by the input images.

We hence propose to select candidate triples following two important properties: First, a good triple must have matching colors along the crossbar of the 3-neighborhood. This provides an easy way to only select triples with good *color consistency*. The second property is less obvious. A major step forward in 2D texture synthesis speed and quality was achieved by giving a higher priority to candidates likely to form coherent patches from the example image [Ash01, HJO$^+$01]. This notion is however not trivial to extend to 3D, as three exemplars are interleaved. Candidates providing coherence in one exemplar are not likely to provide coherence in the other two. Here, our approach of forming candidates prior to synthesis gives us a crucial advantage: we are able to consider coherence across *all three exemplars*, keeping only those triples likely to form coherent patches with other neighboring candidates *in all three directions*.

Color consistency Our first observation comes from the fact that a suitable candidate should be consistent along the crossbar where the 2D neighborhoods intersect. We use this observation to build first sets of potential candidates in each pixel of each exemplar.

As illustrated in Figure 3.23, we seek to minimize the color disparity between the lines shared by interleaved 2D neighborhoods. We compute a L^2 color difference between pairs of 1-dimensional "strips" of pixels (i.e., a Nx1 or 1xN vector) from the appropriate exemplars (E_x, E_y, or E_z). The sum of color differences for the three pairs of pixel strips defines our crossbar error CB for any candidate triple.

In each pixel of each exemplar, we form triples using the pixel itself and two neighborhoods from the other two exemplars. We select the triples producing the smallest crossbar error. For efficiency, we approximate this

process first by separately extracting the S most-similar pixel strips from each of the two other exemplars, using the ANN library [MA97]. For the example of Figure 3.23, assuming we are computing a candidate set for p in E_x, we would first find in E_y the S pixel strips best matching the orange line from E_x, and in E_z the S pixel strips best matching the green line from E_x. We then produce all possible S^2 triples - using the current pixel as the third coordinate - and order them according to the crossbar error CB. In our results, we keep the 100 best triples and typically use a value of $S = 65$, experimentally chosen to not miss any good triple.

Triples of coherent candidates Color consistency is only a necessary condition and many uninteresting candidate triples may be selected. As a consequence, if we directly use these candidates our algorithm will be inefficient as many will be always rejected.

After constructing candidates based on color consistency, we obtain a set of candidate triples at each pixel of each exemplar. Our key idea is to check whether a candidate may form coherent patches in *all* directions with candidates from neighboring pixels. This is in fact a simple test. We consider each coordinate within a candidate triple and verify that at least one candidate from a neighboring pixel has a continuous coordinate. Figure 3.24 illustrates this idea for pixels in E_x. We only keep candidates finding continuous coordinates for all three entries of the triple. Note that one is trivially true, i.e. by definition neighboring candidates in E_x have a continuous coordinate in E_x.

To formalize this notion, let us only consider exemplar E_x without loss of generality. We note ${}^xC$ (resp. ${}^yC$, ${}^zC$) the set of candidates for exemplar E_x (resp. E_y, E_z). ${}^xC^k[p]$ is the k-*th* candidate triple for pixel p in E_x. We note ${}^xC^k[p]_y$ and ${}^xC^k[p]_z$ the coordinates in respectively exemplar E_y and E_z for the candidate triple.

In a given pixel p, we iteratively update the set of candidates as:

$$ {}^xC_{i+1}[p] = \left\{ c \in {}^xC_i[p] \colon \exists k_1, k_2, |\delta_1| = 1, |\delta_2| = 1 \text{ s.t. } \left[\begin{array}{l} |{}^xC_i^{k_1}[p+\delta_1]_y - c_y| = 1 \\ \text{and} \\ |{}^xC_i^{k_2}[p+\delta_2]_z - c_z| = 1 \end{array} \right] \right\} $$

where i is the iteration counter, k_1, k_2 indices in candidate sets and δ_1, δ_2 offsets to direct neighbors.

We perform several iterations of this process, reducing the number of candidates with every pass. In our current implementation we iterate until having no more than 12 candidates per pixel, which typically requires 2 iterations. If more candidates remain, we keep the first 12 with the smallest crossbar matching error. While it is possible that no candidate coherency exists, this happens rarely in practice.

Candidate Slab After candidate generation, we obtain a set of candidate triples at each pixel. These candidates are not only useful for neighborhood matching, but also provide a very good initialization for the synthesis process.

Let us consider a single exemplar. Recall that each candidate triple defines a 3-neighborhood, that is three interleaved $N \times N$ 2D neighborhoods. One 2D neighborhood is in the plane of the exemplar, while the two others are orthogonal to it (see Figure 3.25, left) and intersect along a line of N voxels *above* and *below* the exemplar. This provides a way to *thicken* the exemplar and to form *candidate slabs*. To initialize synthesis we create such a slab using the best (first) candidate at each pixel (see Figure 3.25, right).

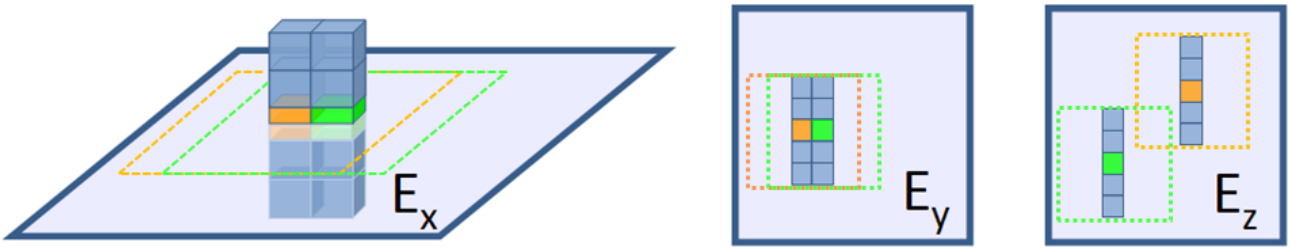


Figure 3.24: *Two candidates of neighboring pixels in E_x. Each candidate is a triple with coordinates in E_x, E_y and E_z. The first triple is shown in orange, the second in green. Notice how the coordinates of the candidates in E_y are contiguous: along both vertical pixel strips, the candidates form a coherent patch from E_y. This is not the case in E_z. Note that the orange and green triples will only be kept if another neighboring triple with a contiguous coordinate in E_z is found.*

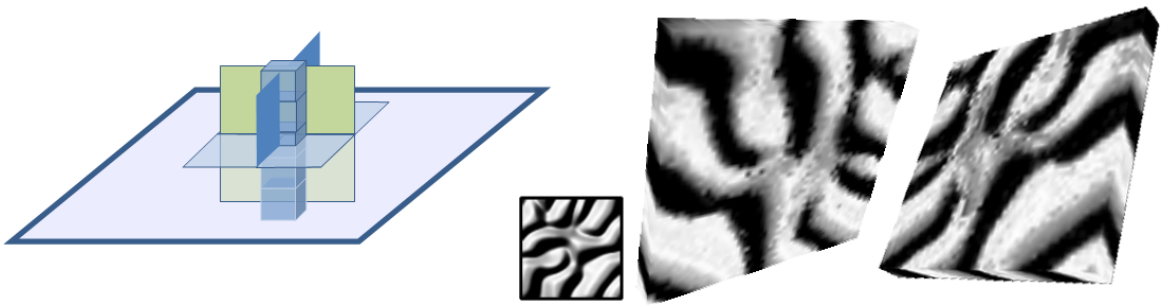


Figure 3.25: Left: *Exemplar thickening.* Right: *Candidate slab obtained from the first candidates. The slab is shown from top and bottom view. Notice how coherent structures appear around the exemplar. (This is not a synthesis result - simply a visualization of the candidates).*

Please note, however, that the slab is formed using a *single* candidate among the several available per exemplar pixel. Using the slab directly as a 3D exemplar would be very limiting: this would ignore all other candidates. Instead, our algorithm exploits the full variety of the candidates for neighborhood matching and uses a slab only for initialization. This is very different from using a 3D exemplar as input, which would require a large example volume to offer a similar variety of candidates.

Volume synthesizer The volume synthesizer is inspired by our 2D parallel algorithm, adapted and extended in several ways. A major change is that our algorithm performs synthesis in a multi-resolution 3D volume pyramid, instead of operating on a 2D image pyramid. Only a subpart of this volume pyramid will be visited by the algorithm, depending on the subset of desired voxels. In order to synthesize the smallest number of voxels, we determine, from a requested set of voxels at the finest resolution, the entire dependency chain throughout the volume pyramid. This guarantees all necessary information is available to ensure spatial determinism. Figure 3.26 illustrates this idea on a simple 2D example.

In order to minimize memory consumption, we performed synthesis into a TileTree data structure [LD07]. We introduced this data–structure in earlier work (see Section 4.1.2). Synthesis is performed at run-time, during display. We demonstrate several examples were an object is cut interactively. Synthesis occurs only once for the newly appearing surfaces.

For more details on the algorithm please refer to our publication [DLTD08].

Results In Figure 3.27, for the 64^2 examples of the first row our method requires a total of 7.22 seconds for synthesizing the 64^3 volume (7 seconds for pre-computation and 220 milliseconds for synthesis). The memory requirement during synthesis is 3.5MB. For the 128^2 image of the last row, our method requires a total of 28.7 seconds for synthesizing the 128^3 volume (27 seconds for pre-computation and 1.7 seconds for synthesis). In comparison, [KFCO$^+$07] reported timings between 10 and 90 minutes.

Figure 3.29 shows results of synthesizing various solid textures on complex surfaces. Performing synthesis using our lazy scheme results in a very low memory consumption compared to the equivalent volume resolution.

Synthesis speed ranges from 4.1 seconds (dragon) to 17 seconds (complex structure), excluding pre-computation. Storage of the texture data requires between 17.1MB (statue) and 54MB (complex structure), while the equivalent volume resolution is 1024^3 which would require 3GB. While being slower than state-of-the-art pure surface texture synthesis approaches, our scheme inherits all the properties of solid texturing: no distortions due to planar parameterization, a unique feeling of a coherent block of matter, consistent texturing when the object is cut, broken or edited. None of the pure 2D synthesis approaches can enforce these properties easily. Our timings nonetheless allow for on demand synthesis when cutting or breaking objects. Figure 3.29 shows four frames of a real-time explosion. The texture has an equivalent resolution of 256^3, while storage requires 1.3MB. The average time for synthesizing a 256^2 texture for a new cut is 8 ms. In terms of memory, synthesizing a 256^2 slice of texture content requires 14.4MB. The overhead is due to the necessary padding to ensure spatial determinism (see Section 3.3.2).

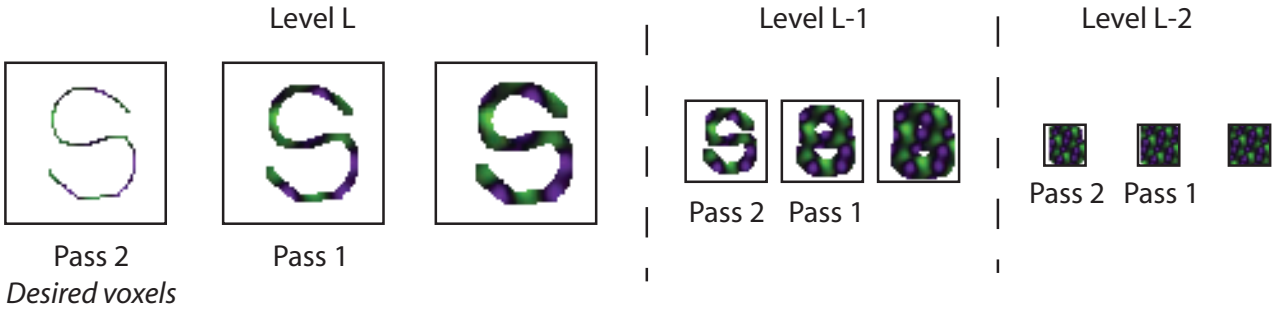


Figure 3.26: *From a set of desired voxels at finest resolution, we have to determine the entire dependency chain throughout the volume pyramid. The neighborhood matching passes performed at every level imply the dilation of the set of voxels to synthesize.*

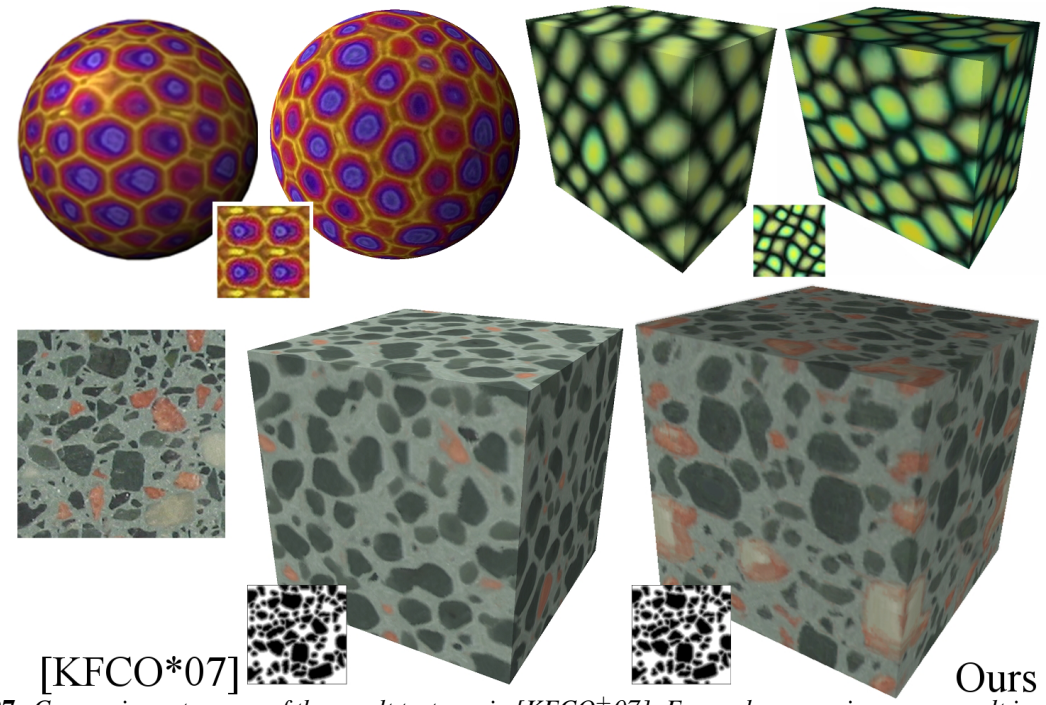


Figure 3.27: *Comparisons to some of the result textures in [KFCO$^+$07]. For each comparison, our result is shown on the right.*

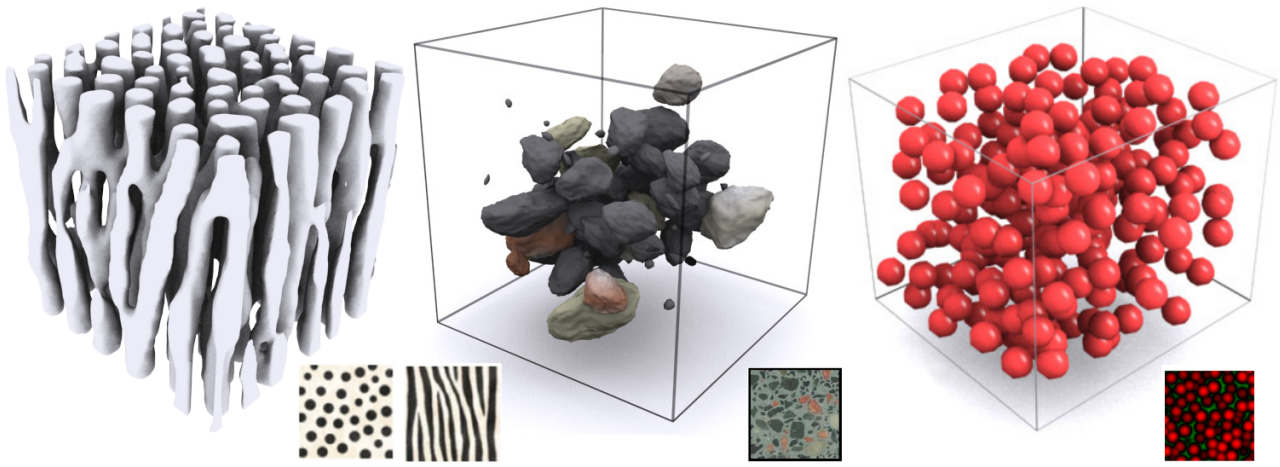


Figure 3.28: Left: *A volume generated from two different images.* Right: *Transparency reveals shape distribution. Background gray voxels are transparent for the stones, green voxels for the spheres. For clarity we removed shapes crossing the volume boundary.*

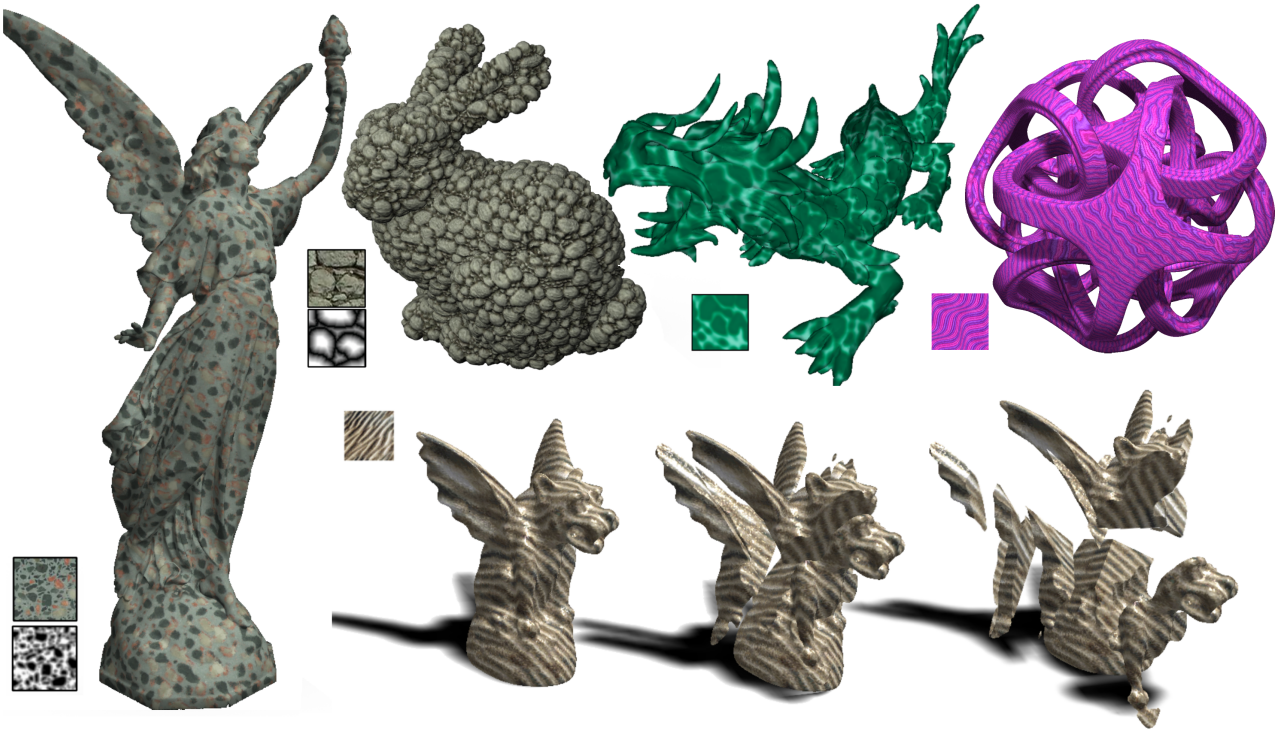


Figure 3.29: *Results on complex surfaces.* Top row: *Surfaces textured with the equivalent of a* 1024^3 *volume. The bunny is rendered with displacement mapping using synthesized feature distance.* Bottom row: *Three frames of the real-time explosion of a gargoyle model. The interior surfaces are synthesized on demand, in 8 milliseconds on average, while the object explodes. Notice the coherent structures across cut boundaries.*

3.5.3 Procedural synthesis on surfaces

In addition to the data–driven techniques mentioned above, we explored procedural texture synthesis along surfaces. Our initial ideas are inspired by the notion of lapped textures [PFH00] and texture particles [DMLG02]. Both of these techniques propose to synthesize textures by overlapping small texture elements. These are typically small patches with irregular boundaries taken from the example. While this is limited to specific textures where such elements are easily extracted, this offers unique opportunities for synthesizing along a surface.

We first proposed an efficient encoding of such textures along a surface (Section 3.5.3.1). Later, this initial work lead us to realize that a similar approach could in fact define a purely procedural surface texture, based on Gabor noise (Section 3.5.3.2).

3.5.3.1 Texture sprites

Figure 3.30: *Surfaces covered with* sprites *that locally adapt to the surface. Overlapping sprites are blended in a variety of ways: additive, max, implicit blending.*

In lapped textures [PFH00], texture patches are directly applied on the surface, giving the illusion of a continuous texture. This is possible because the texture patches tend to be small compared to the surface curvature: a local parameterization of the patch is thus easily obtained.

However, this technique *encodes* the patches as sets of texture coordinates in the 3D model. The main drawback is to introduce a dependency between the mesh tessellation and the texture. Any change to the mesh (e.g. vertex merge or split) would break the synthesized texture. In addition, in a number of rendering pipelines surfaces are not represented through meshes, but rather through a composition of analytical functions. In these cases, there are no vertices to store texture coordinates. We therefore investigated other ways to encode such textures, with the main objective to decorrelate the mesh from the texture. The patch locations and sizes are an input to our technique. They can, for instance, be generated by the lapped texture method.

The core idea is to encode the texture patches, which we call *sprites*, in a data–structure which is accessible from a fragment shader. Therefore, the colors are retrieved from the 3D coordinates of the surface points: this does not necessitate any knowledge about the mesh, its connectivity or tesselation level. Interestingly this provides a technique analog to a volume texture (see Section 3.5.2.2), but where the data stored in the volume is tailored for a specific surface.

The core of our data-structure is an octree texture [BD02, DGPR02] implemented on the GPU [LHN05a]. By todays standards, this is unimpressive. However, in 2004/2005 such data–structures where difficult to obtain within the limited programmability of GPUs. Our implementation is described in more details Section 4.1.1.

Each leaf of the octree stores a number of sprites. Each sprite is encoded as a plane (normal, distance) and a transformation mapping the sprite image onto this plane. This is illustrated Figure 3.31. The sprites have an influence limited to the box. Whenever a 3D point has to be textured, we retrieve the set of sprites that possibly have an influence. We then project the point on each plane using the plane normal direction, and obtain a texture coordinate inside the sprite image. We finally sample each sprite image. The final color is the combination of the contribution of all sprites.

The contribution of a sprite is attenuated along the plane normal by a falloff so that it becomes null outside of the box and progressively increases inside. We also attenuate the sprite's contribution when the normal at the surface point becomes very different from the plane normal (cosine attenuation). This hides distortions due to the simple planar projection.

The interesting aspects of this technique are:

1. The combination of the sprite's color allows for a wide variety of effects: it does not have to be linear. For instance, Figure 3.30 illustrates max and implicit blends.

2. The transformations applied to the sprites can vary through time. This can be used in synch with a surface animation, for instance to produce the illusion that snake scales are sliding on top of each other (Figure 3.32).

3. Filtering is properly dealt with by recomputing correct texture mapping gradients for each individual sprite. This results in correctly filtered textures, up to the point where multiple octree leaves project in a same pixel (a rare occurrence since the tree leaves tend to be large wrt. surface size).

This work was key in our understanding of how to define a surface texture into a volume. It lead to our procedural definition of surface Gabor noise, described next.

3.5.3.2 Gabor noise on surfaces

The Gabor noise presented Section 3.4.2.3 generalizes to arbitrary dimension. However, many applications require noise on a surface, and high quality rendering requires anisotropic filtering.

We discuss here how to extend the two dimensional Gabor noise to operate directly onto surfaces. We obtain surface noise at a point on a surface p with surface normal n by projecting a three-dimensional Poisson distribution on the plane determined by p and n, and evaluating our two-dimensional noise in that plane (Figure 3.33, (a)). Isotropic surface noise is obtained by randomly orienting the local surface frame of each Gabor kernel (Figure 3.33(b)-(c)), and anisotropic surface noise by aligning the local surface frame with a vector field that guides the anisotropy (Figure 3.33(d)-(e)). The two-dimensional Poisson distribution in the tangent plane with impulse density λ is obtained by projecting all points of a three-dimensional Poisson distribution with impulse density $\lambda/2r$ inside the cylinder with radius r, height $2r$, center p and orientation n onto the plane. The points

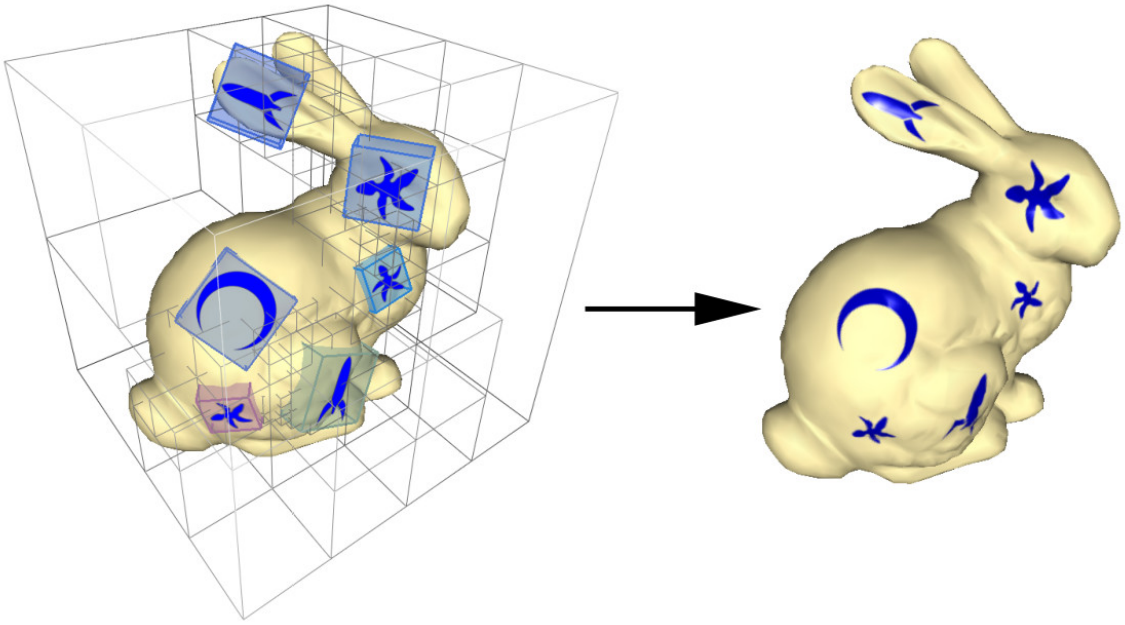


Figure 3.31: *A set of texture patches – called sprites – are stored in an octree surrounding the surface.*

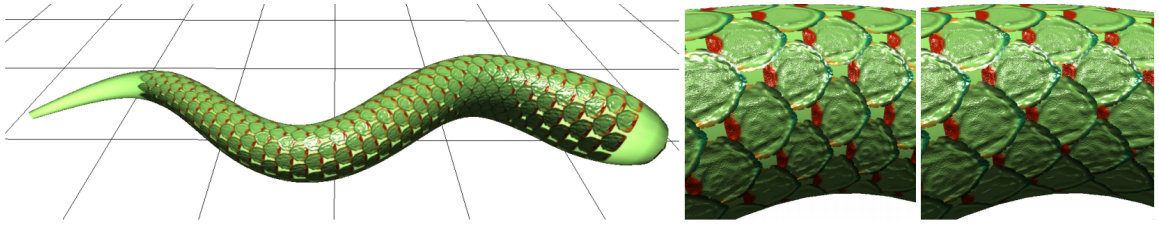


Figure 3.32: *Undulating snake mapped with 600 overlapping texture sprites whose common pattern (color+bump) have a* 512^2 *resolution. The virtual composite texture has a* 30720×5120 *resolution. One can see the correct filtering at sprite edges. This demonstrates the independent tuning of each scale aspect-ratio in order to simulate rigid scales.* Middle: *Without stretch compensation. The texture is stretched depending on the curvature.* Right: *With stretch compensation. The scales slide onto each other and overlap differently depending on the curvature.*

are generated on-the-fly, as for the Gabor noise (Section 3.4.2.3) but using a three-dimensional grid. The weight of a point is defined as one minus the distance of the point to the plane divided by r.

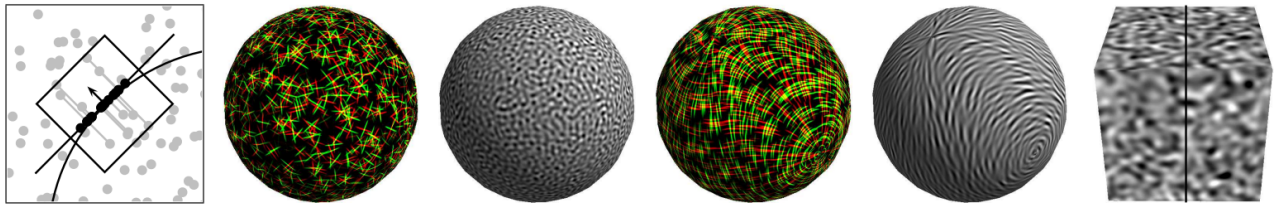
Our isotropic surface noise is setup free, and is not tied to a specific geometry representation. In contrast to the surface noise of Goldberg et al., it does not require precomputation, such as a surface parameterization. Our anisotropic surface noise is setup free as well. We generate the vector field that guides the anisotropy procedurally by computing a local surface frame from the surface normal and a global direction, but the vector field can also be designed by the user [FSDH07].

Our surface noise implicitly assumes that texture detail is small compared to geometric detail, a common assumption in texturing. A limitation of surface noise is that a discontinuity in the surface normal results in a discontinuity in the noise. If this behavior is not desired, then a continuous normal should be used instead (Figure 3.33 (f)). This is similar to the difference between geometric normals and shading normals.

Anisotropic Filtering Anisotropic texture filtering can be formulated as a convolution of the texture with a filter in the spatial domain, or as a multiplication in the frequency domain. We exploit the spectral control of our noise to achieve high quality anisotropic filtering in the frequency domain.

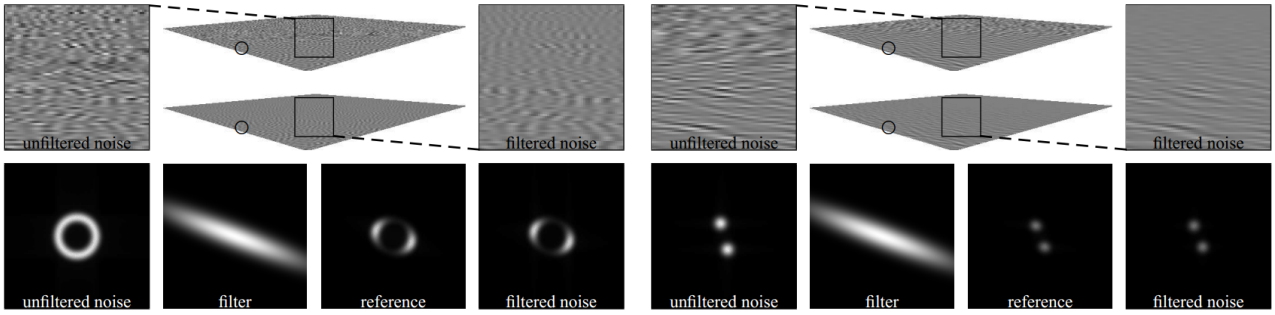
We use the analytic formulation of anisotropic filtering of Heckbert [Hec89], similar to Goldberg et al. [GZD08]. The filter in image space is the Gaussian

$$f(x,y) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2\sigma^2}\left(x^2+y^2\right)}, \tag{3.8}$$



(a) Projection. (b) Iso. surf. frames. (c) Iso. surf. noise. (d) Aniso. surf. frames. (e) Aniso. surf. noise. (f) Discontinuity.

Figure 3.33: *Setup-free surface noise. (a) Projection of the three-dimensional Poisson distribution onto the tangent plane. (b) Randomly oriented local surface frames for isotropic surface noise. (c) Isotropic surface noise. (d) Local surface frames aligned with a vector field for anisotropic surface noise. (e) Anisotropic surface noise. (f) The discontinuous face normals used on the left result in a noise discontinuity, while the smooth vertex normals used on the right ensure continuity.*



(a) Isotropic noise. (b) Anisotropic noise.

Figure 3.34: *Anisotropic filtering. (a) Isotropic noise. (b) Anisotropic noise. (Top row) Tilted plane with unfiltered and anisotropically filtered noise with close-ups. (Bottom row) Power spectrum of the unfiltered noise, frequency domain filter in texture space, reference power spectrum and power spectrum of the anisotropically filtered noise of the circled pixel.*

where σ is the width of the Gaussian. The corresponding frequency domain filter in texture space is the Gaussian

$$F\left(J^T \left[f_u \; f_v\right]^T\right) = e^{-2\pi^2\sigma^2 [f_u \; f_v] J J^T [f_u \; f_v]^T}, \tag{3.9}$$

where J is the Jacobian of the mapping from image to texture coordinates. The power spectrum of anisotropically filtered noise is obtained by multiplying the power spectrum of unfiltered noise with the frequency domain filter (see Figure 3.34). In the frequency domain, our noise is a sum of Gaussians. The frequency domain filter is also a Gaussian. Because Gaussians are closed under multiplication, anisotropically filtered noise is again a sum of Gaussians. This means that we can generate anisotropically filtered noise by simply adjusting the parameters of each Gabor kernel based on J and σ when evaluating the noise. This property lets us achieve unprecedented high-quality filtered noise along surfaces. Please refer to our paper for more details.

Filtered Gabor noise is included in Pixar Renderman as the `knoise` function[4].

[4] `http://renderman.pixar.com/resources/current/rps/rslFunctions.html#knoise`

3.6 By–example synthesis of structured textures

The texture synthesis techniques we have discussed so far focus on homogeneous materials. However, many textures used in virtual environments present a strong organization – some examples are given Figure 3.35 and Figure 3.36. Applying synthesis methods with stochastic assumptions leads to poor results since these are unable to capture the organization of such images.

Approaches have been proposed to analyze the structure existing in the image, so as to treat it explicitly (for instance for facades [MZWG07]). However, these techniques tend to be limited to a given application, such as facade synthesis. Instead, our goal was to propose an as generic as possible approach, producing plausible results on a wide range of images.

3.6.1 Synthesizing architectural textures

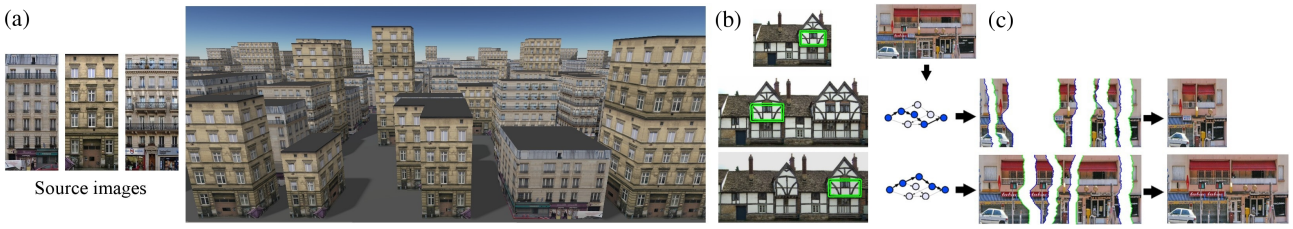


Figure 3.35: (a) *From a source image our synthesizer generates new textures fitting surfaces of any size.* (b) *The user has precise control over the results.* (c) *Our algorithm synthesizes textures by reordering strips of the source image. Each result is a path in a graph describing the space of synthesizable images. Only the path need to be stored in memory. Decoding is performed during display, in the pixel shader.*

This work was done in the context of the SIMILAR-CITIES project, that I coordinated between 2009 and 2013. The project was funded by the Agence Nationale de la Recherche (ANR). In this project we focused on textures applied to urban scenery and buildings – even though the resulting approach tends to be more general. We refer to such textures as *architectural textures*.

A typical difficulty is that architectural textures are created with a given surface size in mind. For instance, facade textures captured from photographs will only apply to buildings of the appropriate size. Door images will only apply to the corresponding door panels. Enlarging or changing the size of the surfaces results in a stretched, unpleasing and unrealistic appearance. In addition, it is often not possible to customize every texture to every surface. A first issue is of course limited authoring resources. A second major issue comes from the extremely large quantity of data that would result from having a unique texture per surface.

Following the same methodology than with stochastic synthesizers, our approach addresses both issues at once: it synthesizes, from an example, new textures fitting exactly the support surfaces *and* stores the result in a very compact form, typically just a few kilobytes. Results are directly used under their compact form during rendering. Decoding is simple enough to fit in a pixel shader. Using our technique, modelers only have to prepare a limited database of high quality textures: a new, unique texture resembling the original is synthesized for each surface, typically in less than a second. If the user is not satisfied with the result, several interactive controls are possible, such as explicit feature positioning through drag-and-drop. This all comes at a small cost since results are compactly stored during display, and efficiently rendered.

To achieve this, we exploit specificities of architectural textures. Our assumption is that the textures tend to be auto-similar when translated along the main axes. This is easily verified on architectural textures. However our approach only applies to this subclass of images, and will be less efficient on arbitrary images. The technique is fully automatic, unless the user wants to achieve a particular goal, such as putting a door or window at a given location. We use the self-similarities in the textures to define a generative model of the image content. In that sense, each image is turned into an image synthesizer able to generate visually similar images of any size. Once constructed, the synthesizer quickly generates textures of any width and height and outputs the result in a compact form.

The synthesizer proceeds successively in vertical and horizontal directions, the second step using the intermediate result as the source image. In each direction, it synthesizes a new image by cutting horizontal (respec-

tively vertical) strips of the source image and pasting them side by side in a different order. This is illustrated along the x direction in Figure 3.35 (c). We search the source image for repeated content in the form of horizontal (respectively vertical) parallel paths along which similar colors are observed. Two such parallel paths are indicated by the green/blue triangles in the inset. We call these paths *cuts* as they define the boundaries of the strips that we "cut out" from the source image. We extract many such parallel cuts from each source 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 (Figure 3.35 (c)). 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 our 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 we are able to solve efficiently for it using a graph-based formulation. The optimization reduces to a simple shortest-path computation. The solution path describes the sequence of cuts that form the boundaries of each strip; it is thus very compact. Please refer to our publication for more details [LHL10].

In addition to these features, our approach enables a number of user controls, for instance letting the user drag and drop part of the result to a precise location. Results of our approach are illustrated Figures 3.36 and 3.37.

Figure 3.36: *Various results of our synthesizer. Sources are marked with a pink square.* Right: *Source is slightly downscaled, bottom results have cuts overlaid.*

Figure 3.37: Top: *Synthesis results. Source images are marked with a pink square. The histogram window size parameter is* 250 *for the leftmost image. Drag-and-drop control is used on the image marked with a blue square.* Bottom, left to right: *The Copenhagen harbor. Automatic enlargement. 2 control cuts (in pink) to add one floor. 4 control cuts to set the building sizes. Final result.*

3.7 Exploring procedural textures

```
COMPILE THIS:
main(k){float i,j,r,x,y=-16;while(puts(""),y++<15)for(x
=0;x++<84;putchar(" .:-;!/>)|&IH%*#"[k&15]))for(i=k=r=0;
j=r*r-i*i-2+x/25,i=2*r*i+y/10,j*j+i*i<11&&k++<111;r=j);}
}
```

Ken Perlin, `http://mrl.nyu.edu/~perlin/`

Appearances produced by procedural textures are hard to predict. Therefore, in addition to better synthesizers, we also need the means to properly explore and reveal to the user the possible outcomes of a texture generator.

We studied techniques to automatically generate *visual previews* for procedural textures. We focus on two different types of previews: static visualization of the space of appearances produced by a given procedure [LLD12a], as well as dynamic previews meant as visual clues for slider–based parameter interface [LLD12b]. We explored these ideas with my PhD student Anass Lasram who defended his PhD in December 2012, in partnership with the company *Allegorithmic* specialized in procedural texturing. This was done within the scope of the ANR project SIMILAR-CITIES which I coordinated.

3.7.1 A space of appearances

Throughout this section, we consider procedural textures in their most general sense: functions producing an image given a set of input parameters. Such procedures are often described through graph–based interfaces, as illustrated Figure 3.38. Allegorithmic provided us with a database of procedures which contains dozens of complex textures from grass fields to brick walls, tilings and space ships doors. Each texture is associated with a set of parameters controlling semantic features such as the apparition of texture elements (e.g. flowers, stones), the size of these features, or the age of the material (e.g. new or weathered, cracked), see Figure 3.40.

For a user, it is quite difficult to realize the expressiveness of a given procedure without spending time randomly changing the parameters. In addition, because the parameters can have intricate interactions it is sometimes a frustrating process (e.g. changing the stone size when stones are hidden has no effect).

3.7.2 Static texture previews

In this section I describe our precise formulation of the texture summary problem – my intent is to show how we went from what seemed an ill-posed problem to the final formulation. I will not detail our algorithm for producing the previews, and refer the interested reader to our publication *Procedural Texture Preview* [LLD12a].

Our goal is to produce a single continuous image summarizing all the appearances that can be taken by a given procedure. An example of a procedural texture preview is illustrated in Figure 3.41. We follow 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.

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 $\mathcal{T}$.

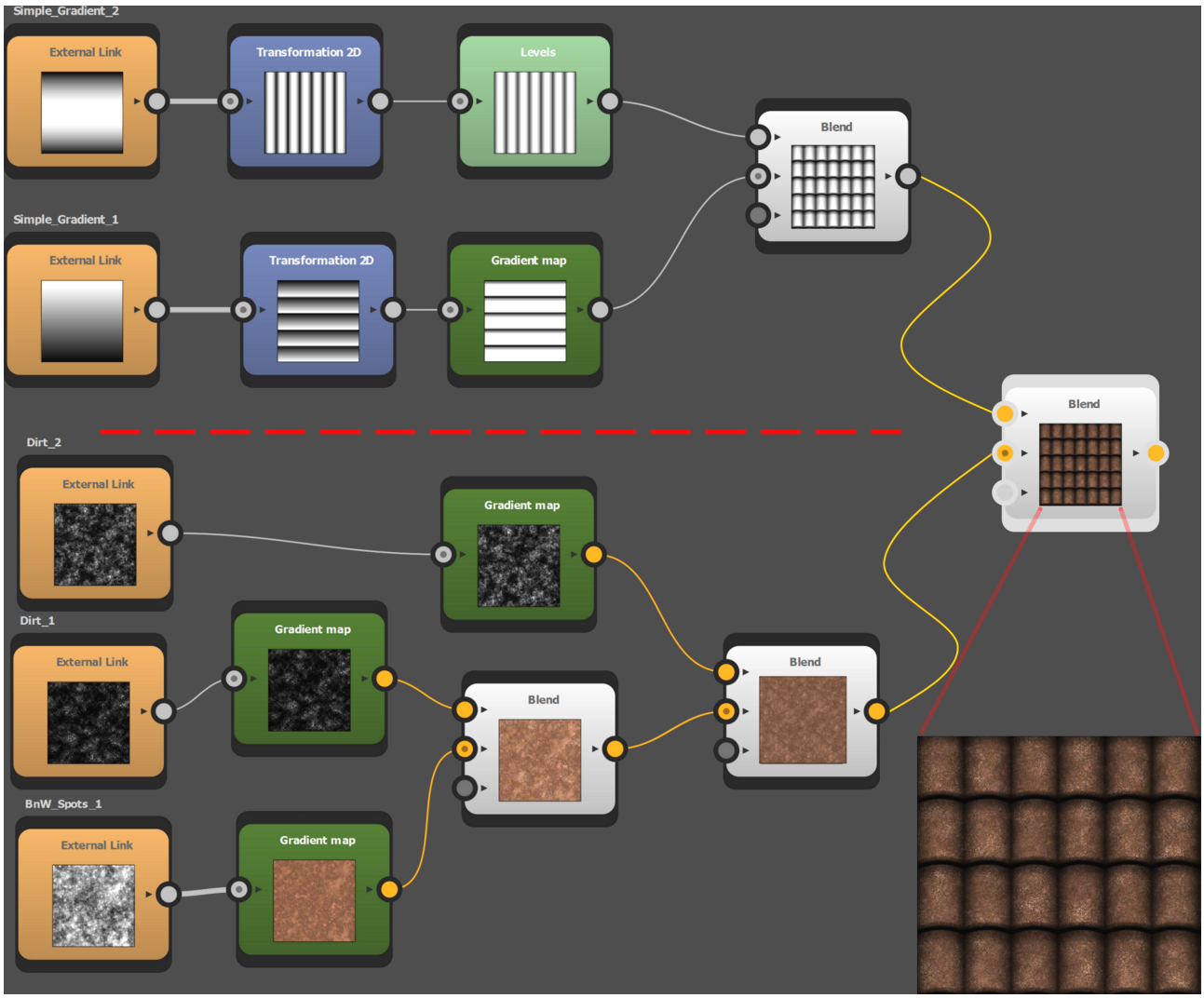


Figure 3.38: *The graph interface of Substance Designer, the procedural texture middleware developed by Allegorithmic.*

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 3.39 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 $\mathcal{M}$. Our metric is a fast approximation of the sum of per–pixel differences over the Gaussian pyramids of the images I_p and I_q. Please refer to our publication for more details [LLD12a].

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 objectives:

- *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 appear-

ance 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.

- *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:

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

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.

- *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.

Our technique optimizes these three energies simultaneously through a modified self organizing map algorithm (SOM). Please refer to our publication for details [LLD12a].

Results Our approach generates texture previews that adapt to the available pixel space. The results shown here have been computed on an 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. We normalize all parameters in $[0, 1]$ for processing. Integer parameters are correctly handled since we build the preview in appearance space.

Figure 3.47 and Figure 3.48 show previews obtained for different procedures. We can observe that the algorithm groups the visually dominant appearances in smoothly ordered regions. In Figure 3.47 (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 3.48 (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 3.47 (b) exhibit a large variety of appearances – working in appearance space allows to properly capture them. In Figure 3.42 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.

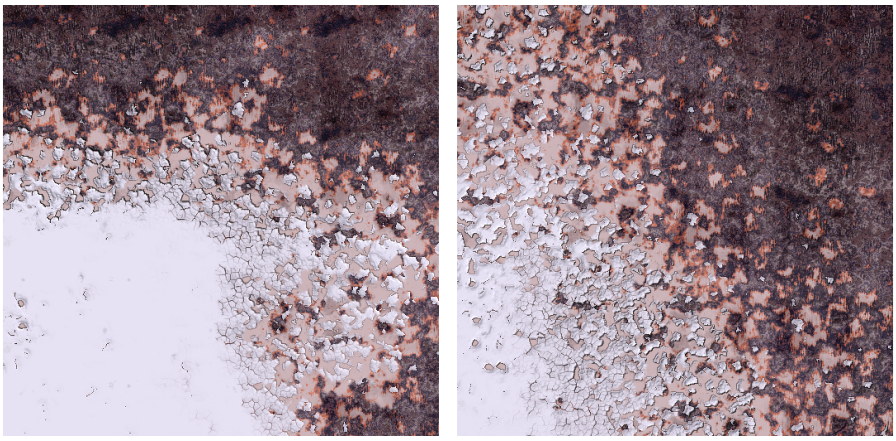


Figure 3.39: Left: *Preview with a parameter–space metric.* Right: *Preview with an appearance–space metric.*

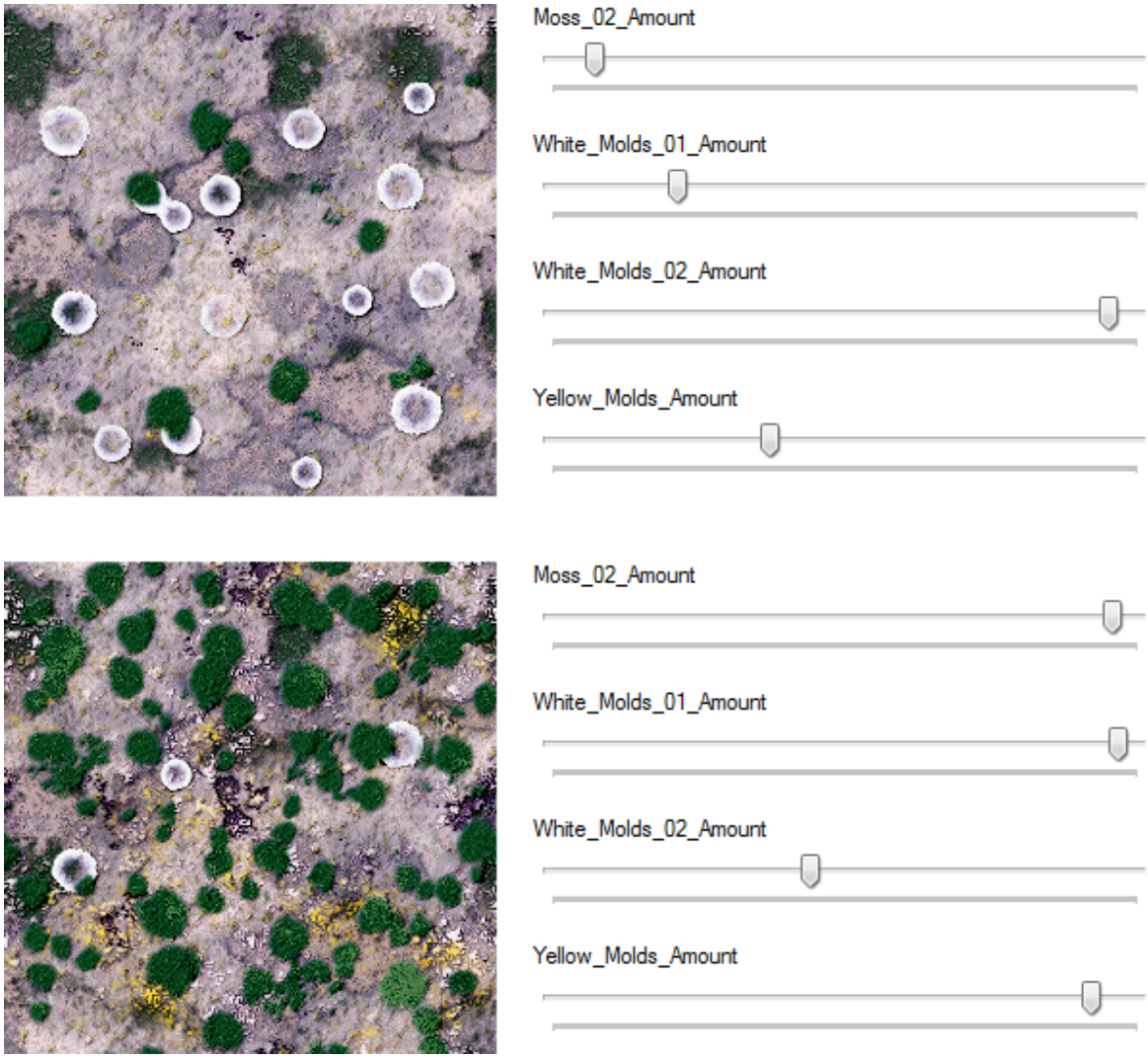


Figure 3.40: Left: *Two textures produced by a same texture procedure.* Right: *The sliders controlling the texture appearance.*

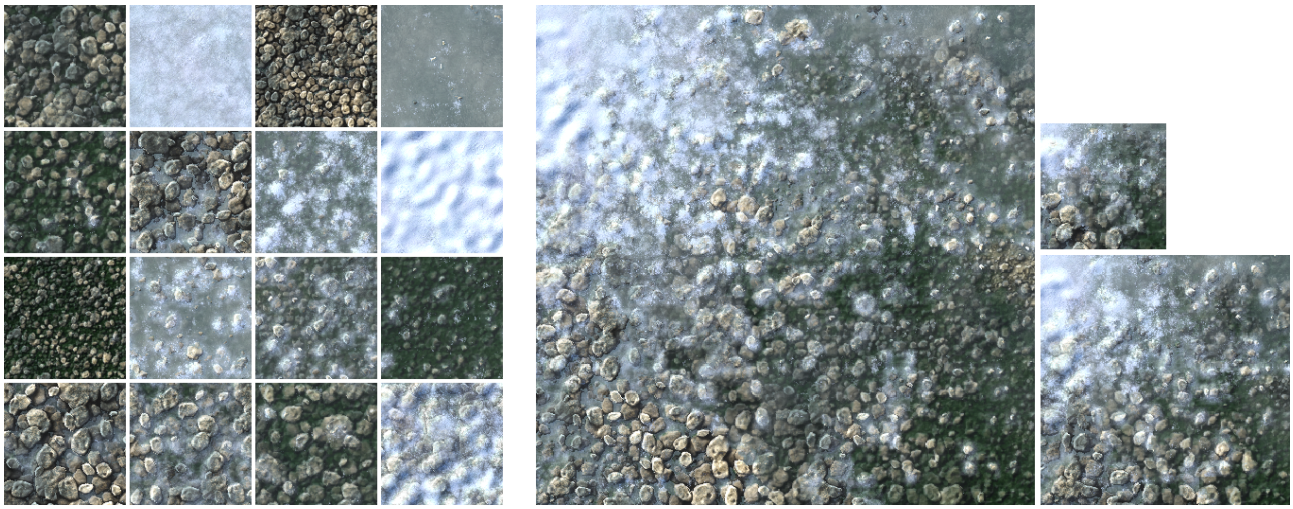


Figure 3.41: *Left: Sixteen random thumbnails cropped in the same procedural texture. Right: Our procedural texture preview summarizes in a single image the possible appearances. A* 512^2*,* 256^2 *and* 128^2 *previews are shown. Note how the smallest preview, which has the same size as the thumbnails, shows the most important variations of the texture (the frozen water, the snow, the moss, the pebbles and transitions between them).*

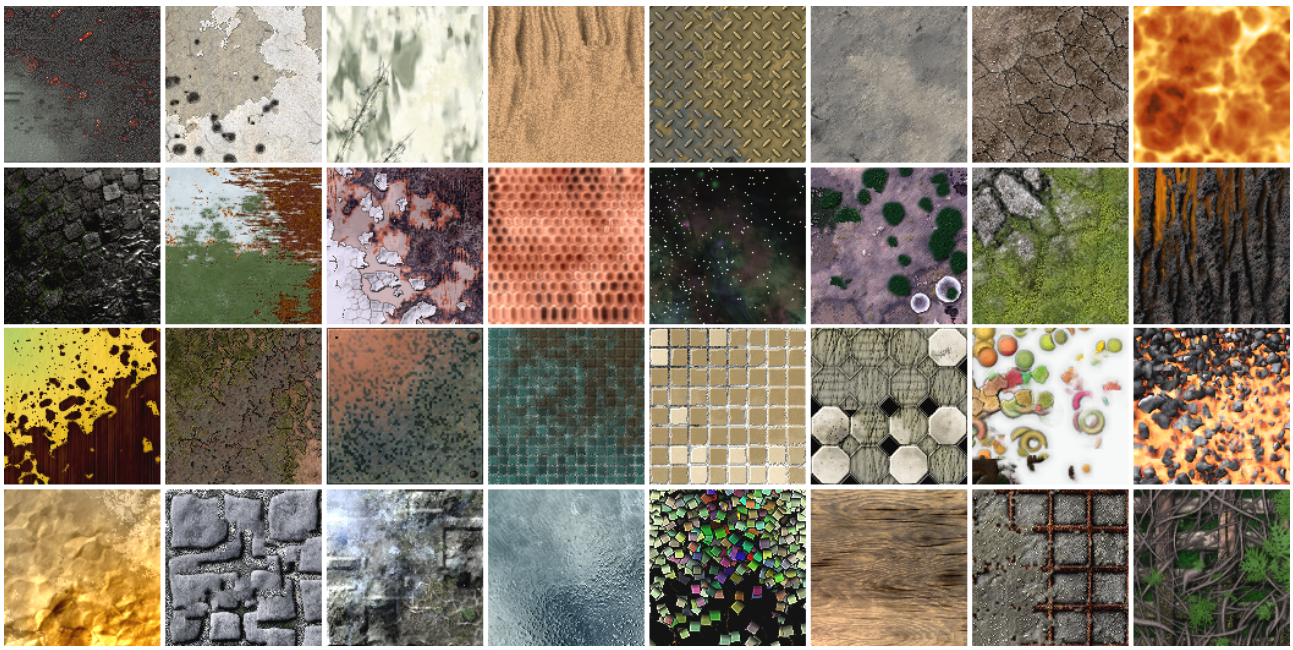


Figure 3.42: *A band of* 128^2 *texture previews. Despite their small size, the main appearance variations are captured in the previews.*

3.7.3 Dynamic slider previews

Building upon our work on texture previews, we proposed a user interface facilitating the selection of procedural texture parameters.

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. 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 have interdependencies. 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 3.43 will have no effect if the parameter 'Milk level' is set to its highest value.

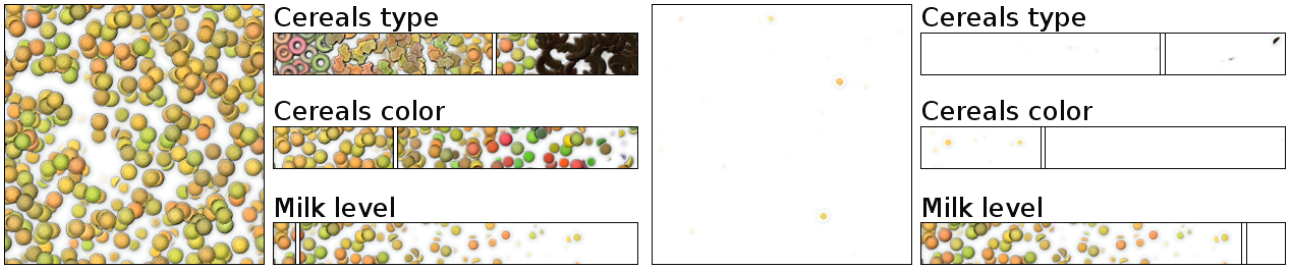


Figure 3.43: *Two settings of the 'Cereals' texture with visual sliders control.* Left: *The 'Milk level' parameter has a low value.* Right: *The 'Milk level' parameter has a high value, masking the effect of other parameters.*

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 previews make the user aware that the 'Milk level' is the only parameter that could affect the settings of Figures 3.43, bottom. This interface is inspired by the concept of scented widgets [WHA07].

Designing such visual sliders for textures 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.

We proposed a solution based on our texture previews to quickly generate such slider previews. The approach is detailed in our 2012 EUROGRAPHICS short paper *Scented Sliders for Procedural Textures* [LLD12b].

Figures 3.44 shows the benefit of our summarization method. In this figure, the most varying elements of the texture (the green stars) represent small features that are 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 variations and give a better insight to the user. Figure 3.45 (left) 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 3.45 (right) 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 3.46 illustrates how the previews evolve when the user changes the settings.

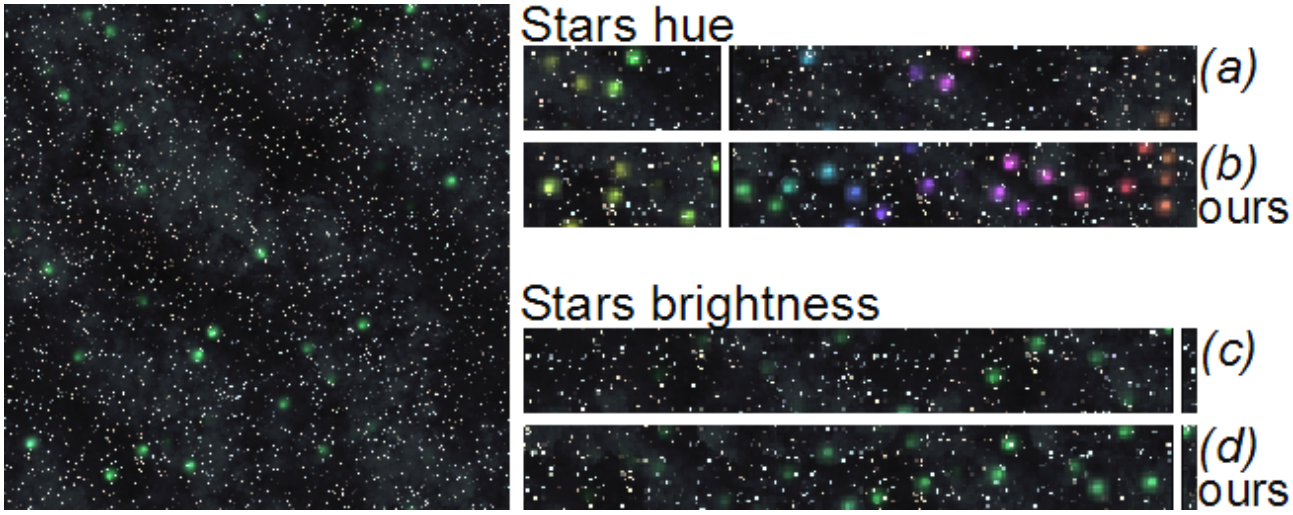


Figure 3.44: *This figures compares a naive technique for sliders to ours. Note how our sliders* concentrate *predicted differences into the slider thumbnails. Here, the sliders contain many stars to outline that the sliders impact their colors and brightness. A simple crop could not capture enough stars to reveal the change.*

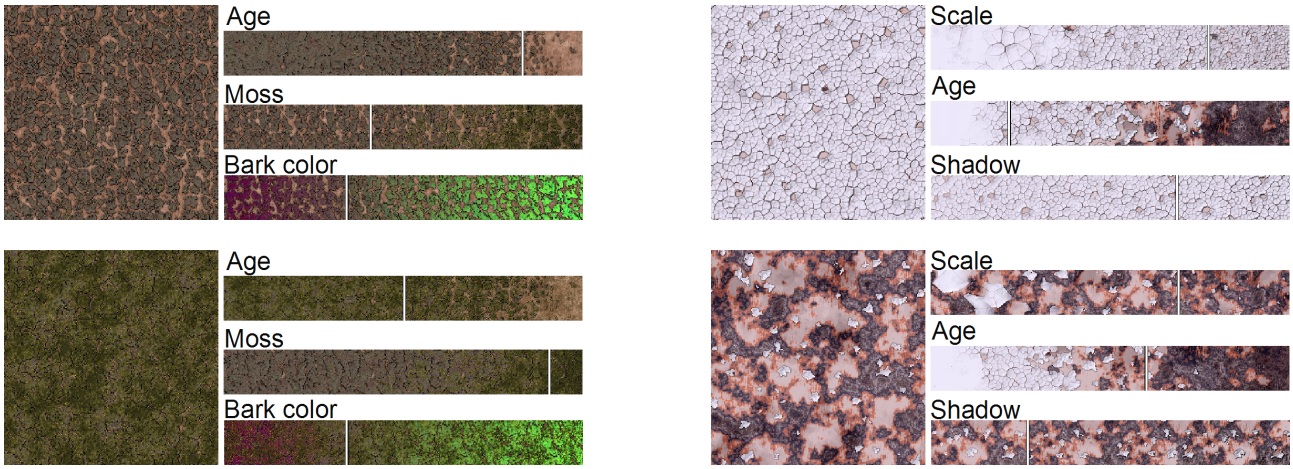


Figure 3.45: Left: *Two different settings of the 'bark' texture with visual sliders controlling parameters.* Right: *Two different settings of the 'rotten wall' texture with visual sliders controlling parameters.*

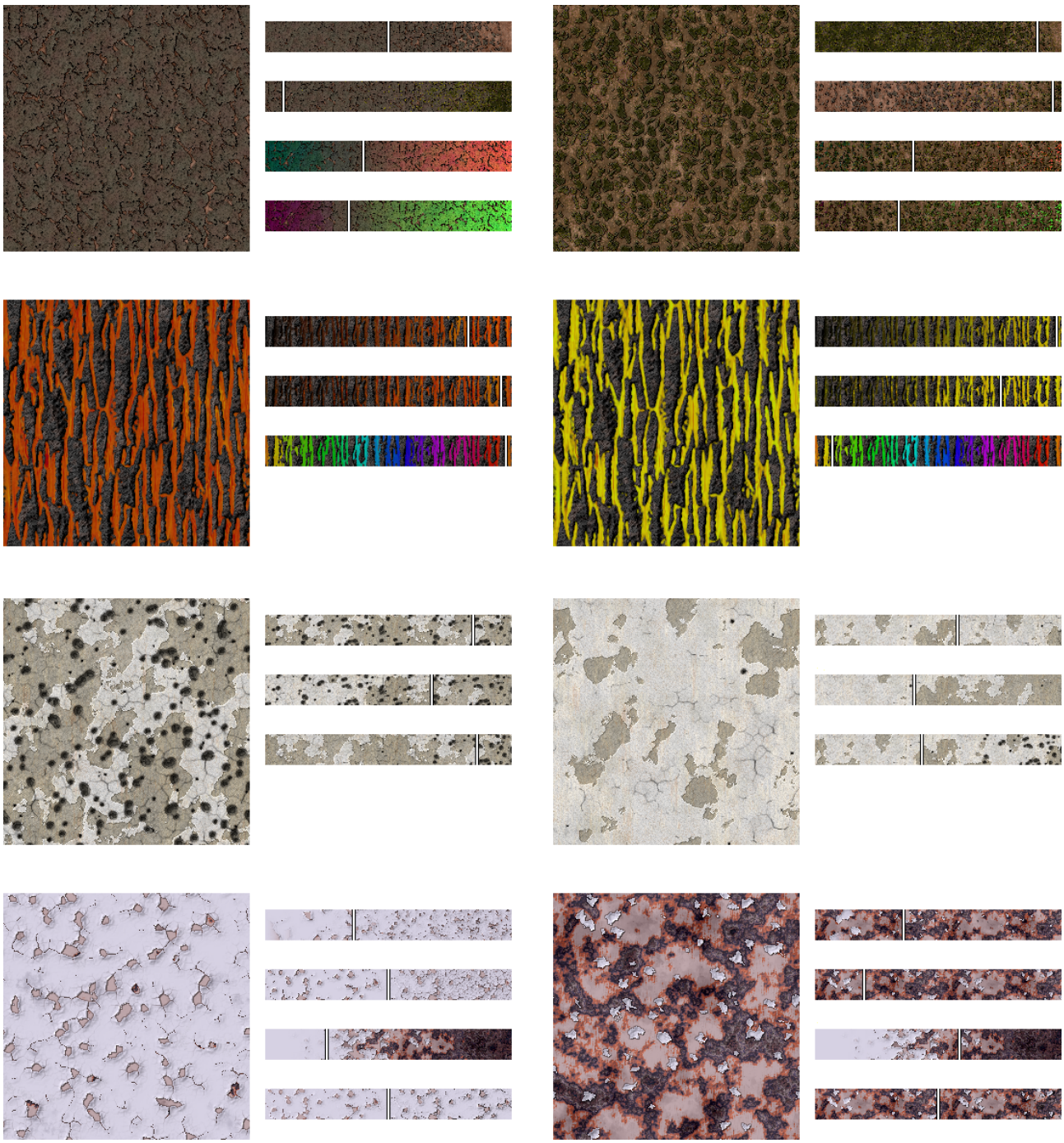


Figure 3.46: *User interactions automatically refresh the previews.*

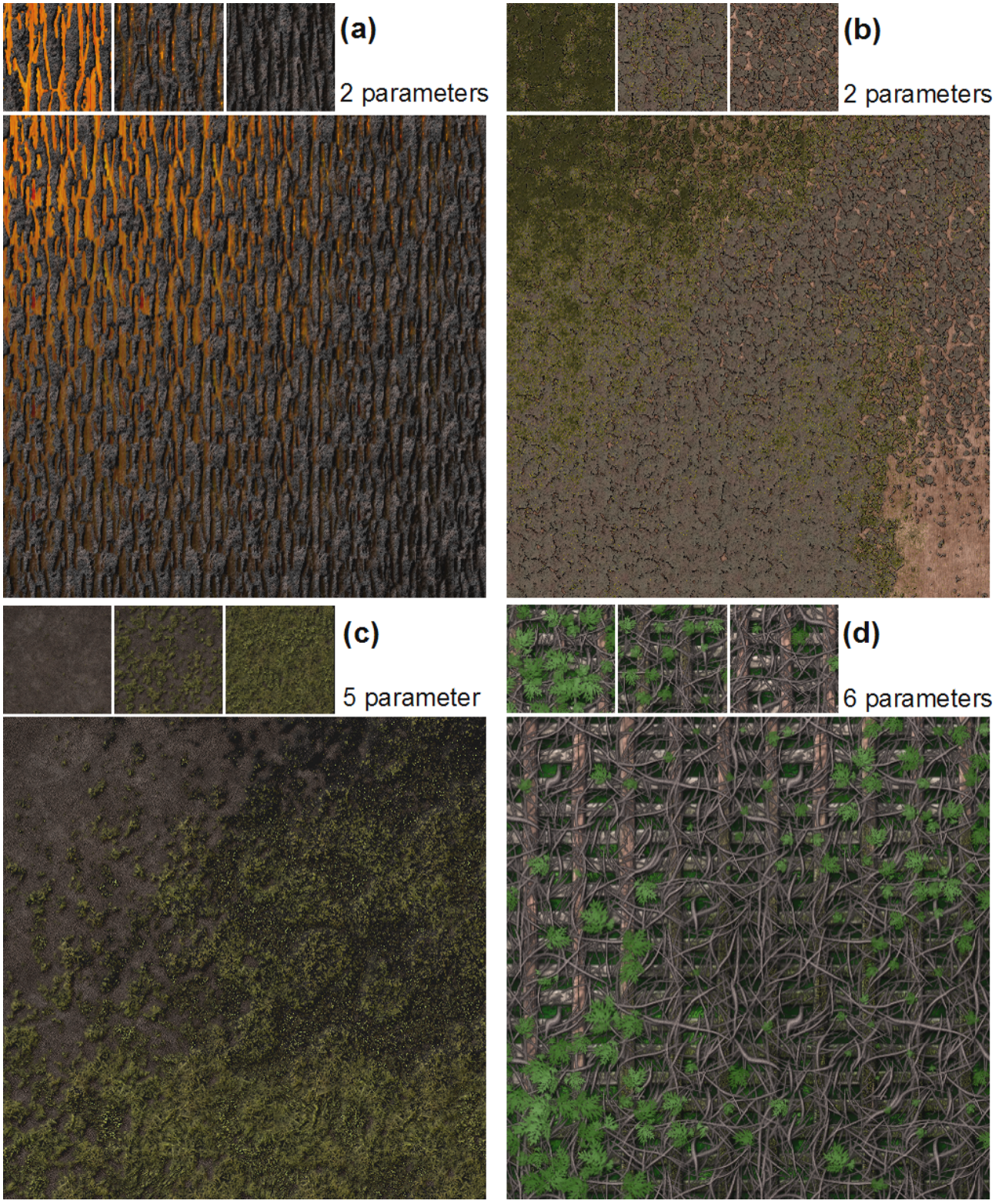


Figure 3.47: *Selection of previews computed from procedural textures. The small images show random thumbnails of the textures.*

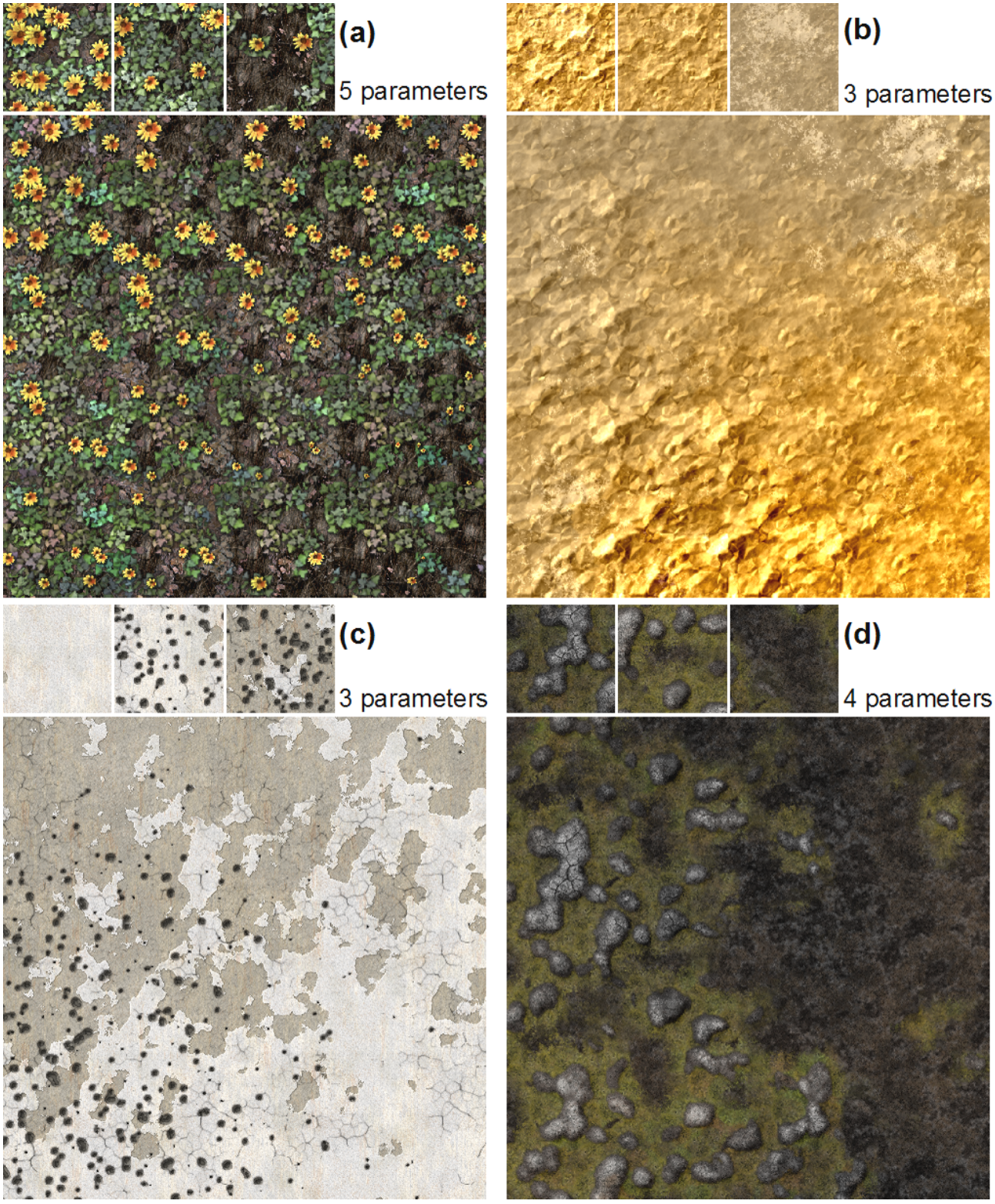


Figure 3.48: *Selection of previews computed from procedural textures. The small images show random thumbnails of the textures.*

Chapter 4

Data–structures for content generation

Real–time exploration of large environments requires to very quickly generate highly–detailed images of complex 3D scenes. The typical framerate is 30 frames per second (FPS) but this number tends to grow both with the adoption of higher display refresh rates (60 FPS) and the advent of stereoscopic displays, which require to compute two views every frame. Thus, it is not uncommon to target 120 FPS, that is 8 msec per frame.

Rendering from textures requires fetching millions of color samples every frame[1]. Therefore, the access performance is critical. In addition these accesses must support interpolation and filtering (see Section 3.5.1.1). This is crucial for image quality since aliasing is mostly dealt with through texture filtering. For these reasons, many texturing techniques rely on very simple data–structures. The data is prepared offline so as to allow for faster access during display.

In the context of on–the–fly content synthesis, it is expected that the generated data will be displayed immediately. Therefore, we cannot rely on expensive computations to prepare the data for rendering. The performance of inserting new data thus has to be fast, otherwise the rate of data synthesis may not be sufficient to follow the user.

During my PhD I developed a technique implementing a paged memory system in the context of texture synthesis [LDN04]. Data is only loaded or generated as the user explores the scene. The data can also exist at different resolutions, making the best use of memory to focus resolution near the user. Similar approaches have been later adopted by the industry, in particular by id Software for the video game *Rage* and the engine id tech 5 [vW09]. We published an up to date version of this work in the book GPU Pro [CESL10]. This line of work inspired a full volume rendering pipeline, developed by Cyril Crassin during his PhD [CNLE09].

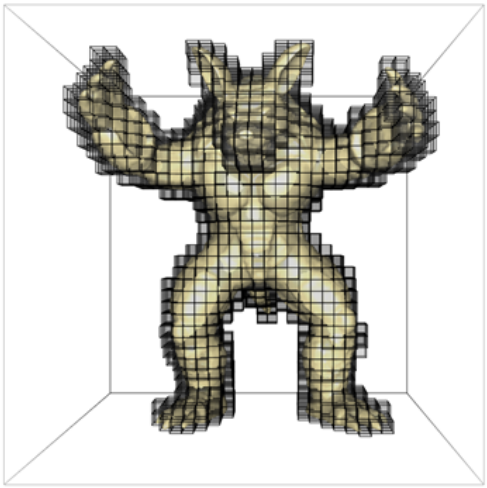


Figure 4.1: *A surface is textured by a regular 3D grid and only a small portion of the grid is intersected by the surface.*

In this document I focus on the two main types of data–structures that I explored in my work. The first are spatial hierarchies, and the second are spatial hashes. Both may be used for either 2D or 3D data, and in both cases I applied these techniques for the purpose of texturing objects (other applications are described in the publications). Figure 4.1 illustrates how a volume grid may be used to store color information around a surface. As is obvious in the figure, the useful information is *only* around the surface. Our techniques are therefore specialized to the case of *sparse* spatial data.

[1]For the sake of clarity we often use the term *color* to designate texture data. This may however be normals, specular coefficients or any other value stored along the surface.

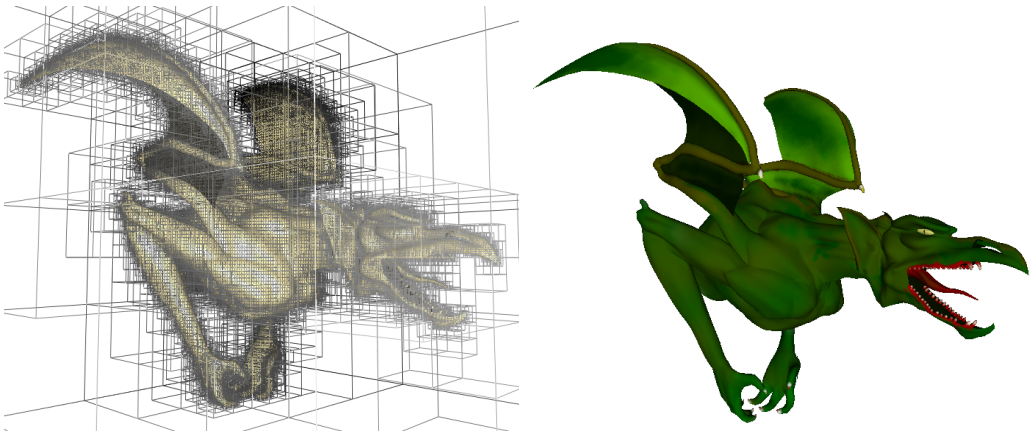


Figure 4.2: Left: *The 3D hierarchy. Each node is regularly subdivided in* 2^3 *nodes (octree). Each internal node stores the average color of the children. Leaf nodes are painted by the artist.* Right: *Rendering using the texture stored in the octree.*

4.1 Trees

In 2002, two SIGGRAPH papers simultaneously introduced the idea of *octree textures* [BD02, DGPR02]. These works have shown how 3D hierarchical data structures can be used to efficiently store color information along a mesh surface without texture coordinates. This has several advantages. First, color is stored only where the surface intersects the volume, thus reducing memory requirements. Figure 4.2 illustrates this idea. Second, the surface is regularly sampled in space and the resulting texture suffers from less distortions. Third, the texture may be dynamically and locally updated or refined, without requiring solving for a planar parameterization problem. These advantages are key in the context of on-the-fly content synthesis, where texture data may not be entirely known at the start of an application. However, in 2002, these works were limited to high–quality off–line rendering pipelines for movie production.

4.1.1 Octree Textures on the GPU

A brief historical note Our first contribution regarding texturing data–structures has been to introduce a GPU implementation of an octree texture [LHN05a]. Nowadays, GPUs have gone through an unprecedented evolution, becoming fully programmable with high-level languages. When we started to consider such data-structures in 2003 programmable GPUs were in their infancy. In fact, we started the work using low–level vendor specific instructions to modify the fixed pipelined. At the end of 2004 we were able to switch to an emulator, provided by NVidia, of the very first GPU programmable through a high-level language (NVidia Cg). The actual GPU was in fact released *after* the completion of our work. Thus, while being applied and quite technical, our work was at the edge of this evolution. The breakthrough of being able to store hierarchies on the GPU and access them quickly opened the door to several future works in texturing [LHN05b], shadowing [LSK+05], and compression [LH07] [LKS+06].

Hierarchical data–structure The main component of a texture hierarchy is the node data–structure. Each node stores either pointers or colors. For the sake of compactness, our work relies on so–called *autumnal trees* [FM86]. In such a tree, each node record contains one entry for each child, each being used either as a pointer or as the color of the child. An autumnal tree has the advantage that the data of a leaf is stored *inside* its parent record. This saves one pointer per leaf, a significant storage reduction when considering that leaves represent roughly half the nodes of the tree.

Each node stores one `RGBA` record per child. A special alpha value indicates that the `RGB` triple points to a child node, instead of being a color. The alignment between parent and child node follows standard texture conventions, similar to a MIP–mapping hierarchy. We will later see that this is actually not the best choice in the context of texturing hierarchies (Section 4.1.3).

Because pointers to children are encoded in a (24 bits) `RGB` triple, the number of nodes in the tree is limited to 2^{24} – already large enough for most applications. The maximal tree depth is not limited. On modern GPUs

it is possible to use other texture formats with larger bit–width. However, 8–bit textures have the advantage of reducing bandwidth consumption and of course result in a more compact data–structure.

All nodes are stored in a same 3D texture, called the *node pool*. Thus, `RGB` triples used as pointers are referring to a coordinate within this node pool. The size of the node pool is doubled each time it is entirely filled: A new pool is allocated, the tree is entirely copied inside, and the old pool is erased. This can be done efficiently on the GPU – pointers do not have to be modified during this process.

N^3–trees The access cost in a tree largely depends on how many nodes have to be traversed down the tree to reach a color. This is especially the case since successive memory accesses are *dependent*: The result of the previous access has to be known to compute the address of the next. This prevents any latency hiding between successive accesses. By increasing the number of subdivisions from 2^3 to N^3 (typical values are $N = 4, N = 8$), the depth of the tree is reduced at the cost of a memory overhead: Larger nodes contain more cells which are not intersected by the surface.

This degree of freedom was introduced in our initial work [LHN05a], but we later extended it so has to change the number of subdivisions at each tree depth. Given a target depth, the choice of subdivisions at each level is then optimized so as to minimize the memory overhead [DL08]. We found this approach very effective in the context of texture hierarchies.

Filtering Since we use the N^3–tree to store texture data, the access algorithm has to perform filtering.

The equivalent of MIP–mapping is achieved by augmenting each node with its average color. These colors are computed from the colors painted by the artist in the leaves of the tree.

Linear interpolation is more difficult. First, the tree must be built so as to ensure that enough colors are defined around the surface. This is illustrated Figure 4.4 (left). To this end we have to consider *interpolation cells*: The cubes defined by eight neighboring nodes. Each interpolation cell that intersects the surface has to be included in the tree. Second, the interpolation has to take into account the uneven sampling: The tree may be more refined in some regions than in others. This is possible following [BD02] but it ends up being quite expensive due to the several additional memory accesses that are required. We improved this in later work [LH07] (see Section 4.1.3). The interested reader can refer to our original publication on the octree texture for more details on the method we initially employed [LHN05a].

Dynamic updates Dynamic updates to both the stored data and the tree structure are simple.

The tree structure is maintained both on the CPU and on the GPU. Most editing operations are local, impacting only a sub–part of the tree. Changes are first applied on the CPU side, to determine which nodes have to be modified. Only the subpart of the tree which has changed is then sent to the GPU. Because the tree is based on pointers, it is easy to delete and replace entire sub–trees.

In many cases changes are applied only to the texturing data, with no impact on the tree structure. Such changes can be performed directly on the GPU. For instance, in a color paint application, a spherical brush is used to determine which leaves of the tree should be painted. All leaves are tested through a render–to–texture operation in the texture storing the nodes. The colors of the leaves enclosed in the brush are then updated. If filtering is required, one pass per–parent level computes the average color of the internal nodes. These operations are performed very efficicently on the GPU.

Applications We applied our GPU octree in a variety of situations. Our initial paper demonstrates interactive painting as well as cellular automatons defined along the surface – this is used to simulate a fluid flowing down the surface.

In later work, we used the tree to store texture elements locally projected onto the surface [LHN05b]. This work is described in more details Section 3.5.3.1. This inspired another development called the *TileTree*, which was also used to store the results of a volume texture synthesizer (see Section 3.5.2.2). The TileTree is described in more details in the next section (Section 4.1.2).

Finally, we revisited our work on GPU hierarchies with a focus on compressing both the tree structure and the texturing data. This work is described Section 4.1.3.

4.1.2 The TileTree

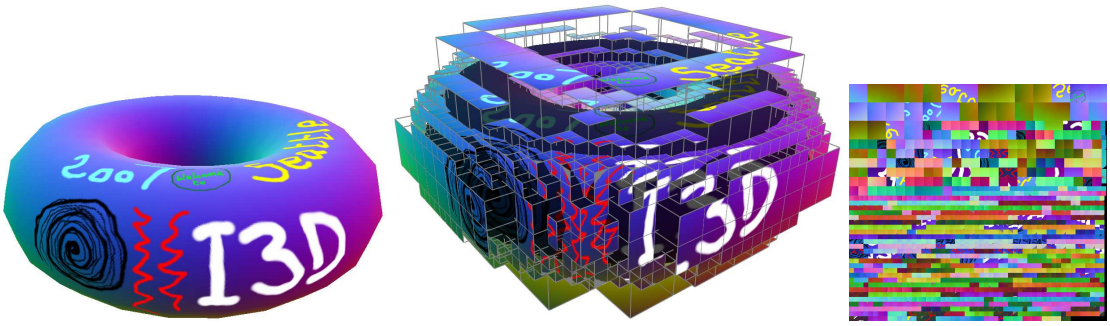


Figure 4.3: *Left: A torus is textured by a TileTree. Middle: The TileTree positions square texture tiles around the surface using an octree. At rendering time, the surface is projected onto the tiles. Right: The tile map holding the set of square tiles.*

The TileTree [LD07] is based on an octree stored on the GPU. However, it is designed to address one of the main limitations of storing data in a volume around a surface: Generally, the surface is represented as a thick layer into the volume, thus requiring to store and access 8 samples for proper tri-linear interpolation. This is illustrated Figure 4.4. However, interpolating over a surface should only require *4 samples*: These approaches always store and access at least twice the data required to texture a given surface. In contrast, our TileTree maps 2D texture tiles onto the surface and therefore strongly reduces both storage and access requirements.

Our approach starts by building an octree around the surface to be textured, similarly to previous octree-based texturing methods. However - and this is the key idea of our work - instead of storing a single color value in the leaves, we map 2D *tiles* of texture data onto the *faces* of the leaves (up to six tiles per leaf). The tiles are compactly stored into a regular 2D texture, the *tile map*. During rendering, each surface point is projected onto one leaf face. This produces texture coordinates then used to access the corresponding texture tile. This idea is illustrated Figure 4.5.

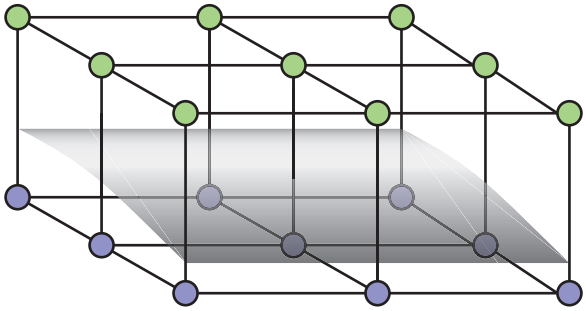


Figure 4.4: *Correct tri-linear interpolation in a volume requires storing the surface as a thick layer. However, the bottom (blue) samples should be sufficient to texture the surface, leading to a 2x saving.*

We only subdivide the octree until no more than one surface fold exists in each leaf along the main axes. Most leaves of the tree actually contain a single tile, with only a few complex cases, as illustrated Figure 4.7. Since with most geometry the texture detail is much finer than the geometric features, the octree leaves tend to be much larger than the texture pixel size: Many neighboring pixels share the same leaf, which guarantees a good access coherence.

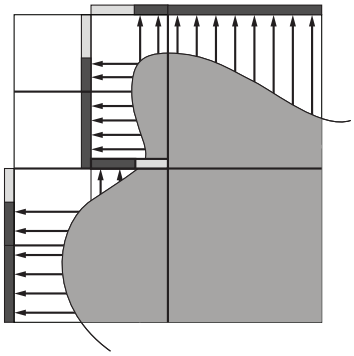


Figure 4.5: *We position texture tiles around the surface using an octree. During rendering the surface is projected onto the tiles of closest orientation. The figure shows the 2D equivalent of a tile tree: A quadtree positioning 1D tiles around a curve.*

The surface is projected onto the faces of the leaves with a simple parallel projection. The face to project onto is locally determined from the surface normal, as illustrated Figure 4.6. Note that this projection only requires knowing the surface normal and enclosing leaf. It is performed dynamically *at rendering time*: Thus our approach does not require to store additional information in vertices. In fact, it does not even require vertices at all: It can be used on implicitly defined surfaces.

Overall the TileTree proves a very efficient data–structure when it comes to texturing surfaces. In particular, it is one of the most compact parameterization free technique to date. Even the hashing schemes that we will present later remain larger at comparable performance levels, as

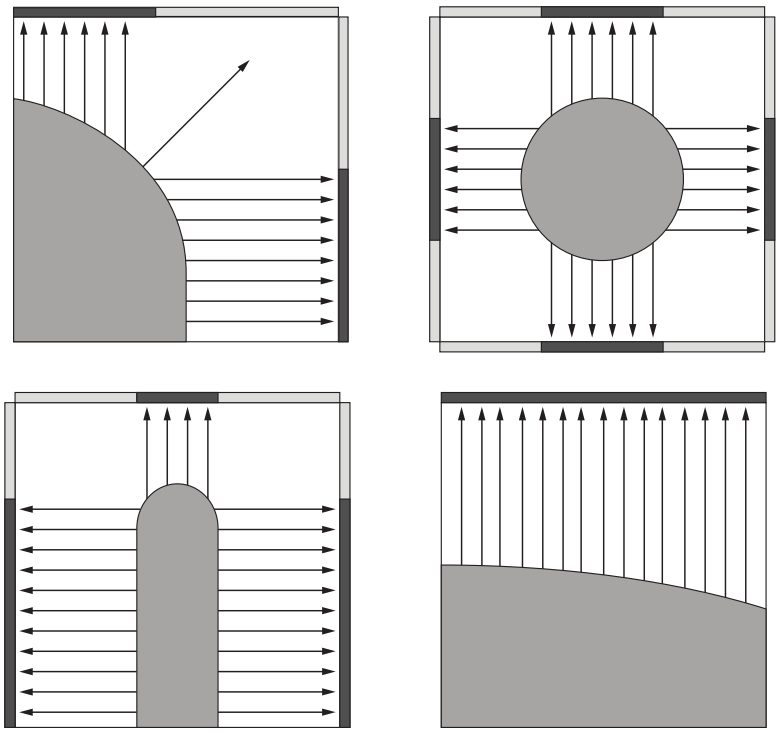


Figure 4.6: *The normal to the surface is used to select on which face to project. The surface point is then mapped to the face with a simple parallel projection.*

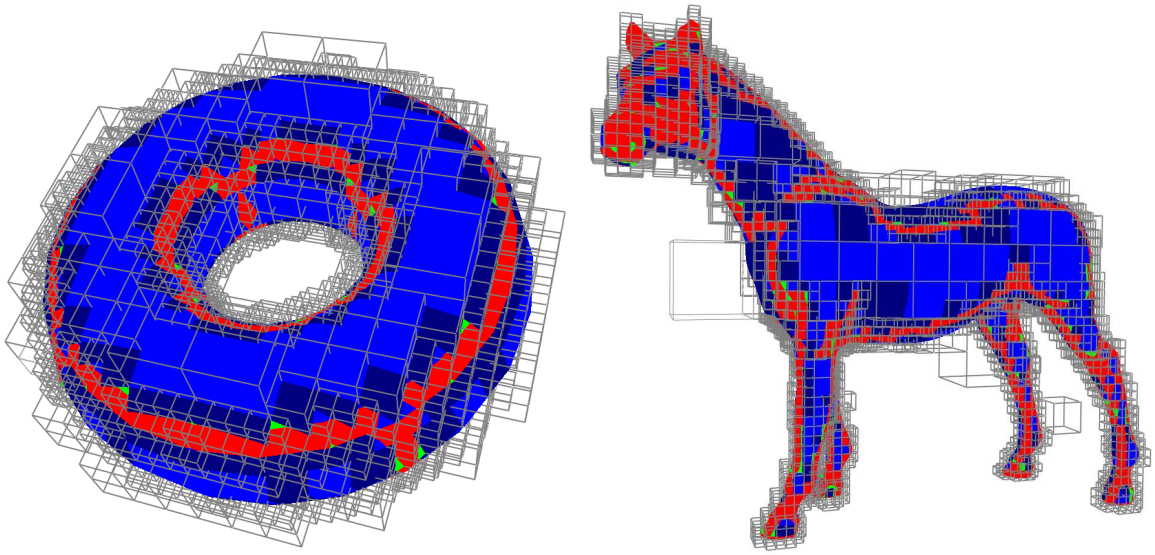


Figure 4.7: *Leaves in blue and dark-blue have a single tile.*

shown in Table 4.1. Of course, the TileTree is also less general and requires an off–line construction process. Most of this process could be mapped onto the GPU however. The only difficult part is the color duplications to ensure a seamless interpolation across tiles.

Note that access to the tree can be improved: In a follow up work we explained how to collapse the tree into a shallow nested grid hierarchy [DL08].

4.1.3 Compressed Random-Access Trees

The GPU octree texture presented Section 4.1.1 has for main purpose the hierarchical encoding of texture data around a surface. The focus was on versatility, dynamic update and access efficiency. Most modern applications are however severely limited by the amount of available texture memory. It is therefore desirable to provide very compact data–structures, able to store compressed data while enabling very fast access during rendering.

	Memory size	Frame rate
TileTree	11.4 MB	91 FPS
Hashed texture *8 lookups for tri-linear*	15.7 MB	34 FPS
Octree texture *8 lookups for tri-linear*	32.6 MB	25 FPS
Hashed texture *blocking for tri-linear*	45.9 MB	135 FPS

Table 4.1: *Comparison of a TileTree with octree texture [BD02] and hashed textures [LH06b] on the armadillo model with an equivalent volume texture resolution of* 1024^3*. Frame rate is measured with the viewpoint of Figure 4.8.*

Figure 4.8: *Left: Armadillo model textured with a uniform resolution of* 1024^3*. The entire TileTree fits in 11.4 MB. Right: Dragon model textured with a uniform resolution of* 1024^3*. The entire TileTree fits in 11.3 MB.*

This is the focus of the approach described here. Note however that this data–structure is static and optimized offline. It is therefore less interesting from the perspective of on–the–fly content generation. Nevertheless, it is a good illustration of other approaches relying on our GPU hierarchies.

Most existing compression schemes for textures are based on fixed ratio block–based compression [BAC96, SAM04, S3T]. This is essentially due to the strong requirements on random access efficiency which prevent the use of sequential entropy coding. These compression approaches are very effective when textures contain high frequency patterns – a common situation with textures of homogeneous materials. There are, however, many other textures which exhibit a smooth, slowly varying content. This is for instance the case for light–maps, terrains, distance fields, environment maps. Such a texture is illustrated Figure 4.9. On these images block–based compression is less effective than schemes which can locally adapt to the frequency content.

Our approach enables stronger compression of such images through the use of a tree hierarchy. The scheme is based on a hierarchical prediction of data values, each level of the tree encoding residuals with respect to the prediction of the parent level. A quality–size tradeoff is easily achieved by selecting which residuals to keep, for instance through a user adjustable bit rate.

I will not further describe the compression and quantization of the data stored within the tree, and refer the interested reader to our publication [LH07]. Instead, I would like to emphasize two important aspects of this work which find applications beside texture compression: The use of a primal subdivision for interpolation, as well as the compression of the tree topology (as opposed to the data stored inside it).

Primal subdivision In a traditional region quadtree (dimension d=2), nodes correspond to spatial cells that are properly nested across resolution levels, forming a dual-subdivision structure [ZS01]. An important drawback of a dual tree is that mipmap filtering becomes expensive when the tree is adaptive. Indeed, Benson and Davis [BD02] explain that mipmap interpolation requires a total of 3^d lookups per level. The problem is that pruned tree nodes must be interpolated from the next-coarser level (as shown by the red boxes in Figure 4.10,

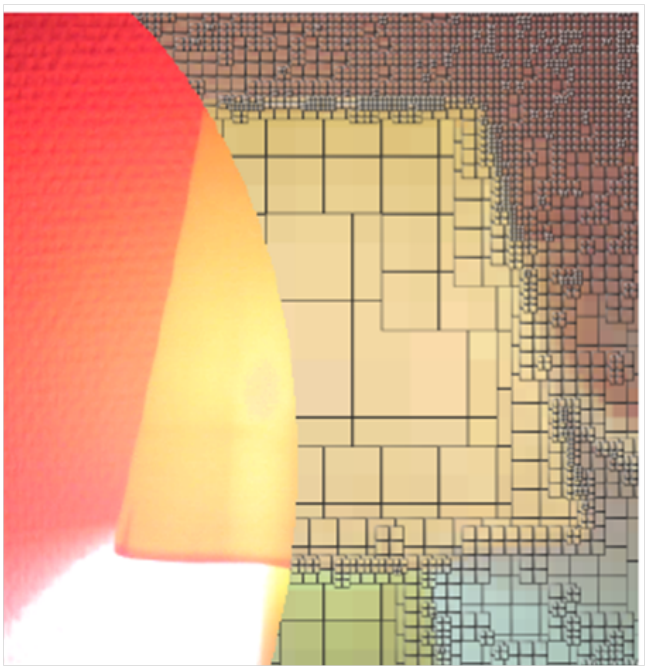


Figure 4.9: *This smooth high-dynamic range image is compressed and encoded using our compact randomly–accessible tree. Each square outlines one node of the tree.*

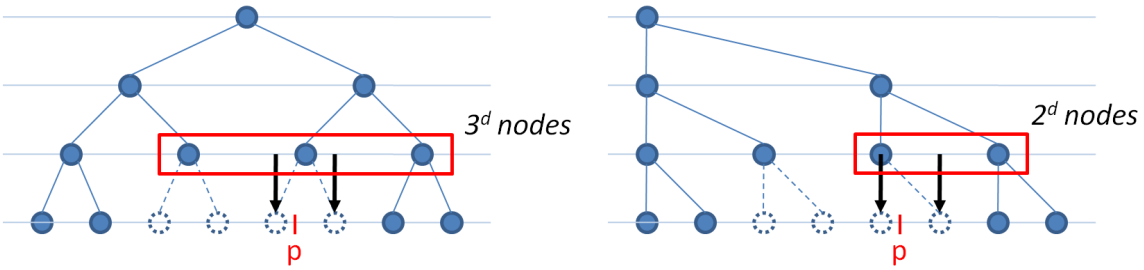


Figure 4.10: *Left: The dual subdivision scheme is the most common for texture data. Unfortunately, interpolating in between level of resolutions requires up to 3^d lookups (with d the dimension). Right: We instead rely on a primal subdivision scheme. This requires only 2^d lookups when interpolating accross level, thus reducing bandwidth.*

left), and this interpolation requires a large support. Also, in pruned areas of the tree, the successive interpolations of the 3 local nodes is equivalent to a multiquadratic B-spline, which is nicely smooth but expensive to evaluate.

We instead associate tree nodes with the cell corners, so that finer nodes have locations that are a superset of coarser nodes (though their values may differ). This corresponds to a primal subdivision structure, and allows continuous interpolation over an adaptive tree using only 2 lookups per level (e.g. Figure 4.10, right). Moreover, refinement can terminate with simple multilinear interpolation when all 2 local nodes are pruned. To our knowledge these advantages of primal trees had not been explained previously.

To represent a primal tree, we *slant* the tree structure as shown in Figure 4.10, right. Thus, in 2D, the children of a node at location (x,y) have locations (x, y), $(x, y + 2^{-l})$, $(x + 2^{-l}, y)$ and $(x + 2^{-l}, y + 2^{-l})$ in level l. In 1D, the pseudo code to evaluate interpolated values without recursion (GPU-friendly) is:

```
1 class Tree
2 {
3  Tree L, R;
4  float val;
5 }
6
7 float evaluate1D(Tree root, float x, float level)
8 {
9  Tree l = root.l;
```

```
10  Tree r = root.r;
11  float vl = l.val;
12  float vr = r.val;
13  float vC = interp(vl, vr, x); // value at current level
14  for (;;) {
15    if (!l && !r) {
16      return vC; // early exit if pruned
17    }
18    vm = interp(vl, vr, 0.5); // default midpoint value
19    if (x<0.5) { // select left/right subtree
20      x = (x−0.0)∗2;
21      l = (l ? l.L : 0);
22      r = (l ? l.R : 0);
23      vr = vm;
24    } else {
25      x = (x−0.5)∗2;
26      l = (l ? l.R : 0);
27      r = (r ? r.L : 0);
28      vl = vm;
29    }
30    if (l) vl=l.val; // set values if not pruned
31    if (r) vr=r.val;
32    float vF = interp(vl, vr, x); // value at current level
33    if (level<=1.0) {
34      return interp(vC, vF, level);
35    }
36    vC = vF;
37    level = level−1.0;
38  }
39 }
```

As can be seen from the code above, only two nodes of the tree are simultaneously active (`vl, vr`). This principle generalizes to higher dimensions.

Using a primal tree reduces the bandwidth requirements during access and results in a simpler access code using less registers. Both are critical for efficiency when accessing such a hierarchy in every pixel of an image, on the GPU.

We further reduce bandwidth and memory requirement by compressing the tree topology.

Compressed tree topology I now describe our scheme for compressing the topology of the tree.

Terminology. A tree node with at least one child is an internal node; otherwise it is a leaf. The depth of a node is the length of the path to the root, so the root node has depth zero. Level l of the tree refers to all nodes at depth l. The tree height L is the maximum level. A complete tree has all its leaves at the same depth, and hence a total of 2^{dL} leaves. In an *arbitrary tree*, each node may have any number (0 to 2^d) of children. We focus on full trees, in which all internal nodes have a full set of (2^d) children. Note that a complete tree is always full, but not conversely.

Traditional tree data structures. We begin by reviewing structures for tree topology. In the simplest case, each node contains a data record and 2^d pointers to child nodes, any of which can be NULL:

```
1 struct Node {
2   Data data;
3   Node∗ children[2d];     // NULL if the child is pruned
4 };
```

Assuming 32-bit pointers, the tree topology requires $4 \cdot 2^d$ bytes per node, with much wasted space at the leaf nodes.

Sibling tree. An improvement for full trees is to allocate sibling nodes contiguously (forming a brood), and to store a single pointer from the parent to the brood [HW91], thus reducing topology encoding to 4 bytes/node:

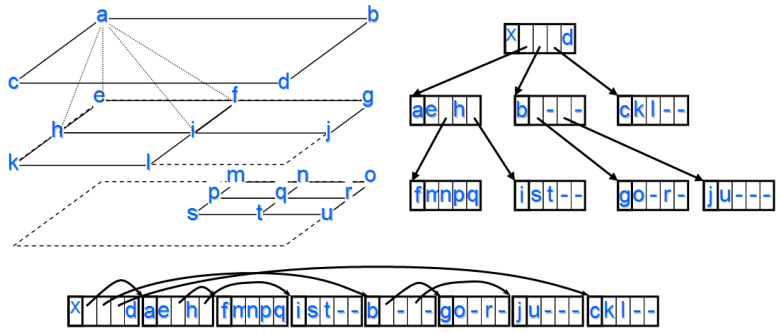


Figure 4.11: *Example of a primal autumnal tree and its memory packing. Dashes denote undefined data values.*

```
1  struct Node {
2   Data data;
3   Brood* brood;        // pointer to first child, or NULL
4  };
5  struct Brood {         // children nodes allocated consecutively
6   Node nodes[2d];
7  };
```

Autumnal tree. If pointers and data records have the same size, an even better scheme is to raise the data from leaf nodes into their parents, to form an autumnal tree [FM86]. Hence the Node structures are only allocated for internal nodes. A single bit identifies if a child is a leaf, and is often hidden within the pointer/data field. Tree topology is reduced to $4 \cdot 2^(- d) + \frac{1}{8}$ bytes/node. For a quadtree, this is 1.125 bytes/node, much less than the sibling tree.

```
1  struct PointerOrData {
2   bit leafchild;
3   union {
4    Node* pointer; // if not leafchild
5    Data data; // if leafchild
6   };
7  };
8  struct Node { // only for internal nodes
9   Data data;
10  PointerOrData children[2d];
11 };
```

Encoded local offsets. Even in an autumnal tree, the pointers remain the limiting factor for memory size. Our contribution is to replace such pointers by local offsets. Hunter and Willis [HW91] consider replacing absolute pointers by offsets, but define offsets from the start of the tree data structure. Instead, we define offsets locally, such that an offset of zero refers to memory just after the current node. Starting with an autumnal tree, we replace each absolute 32-bit pointer by a local scaled offset encoded into 7 bits. The tree data values will also be encoded into 7 bits, so that the PointerOrData structure fits nicely in one byte.

We pack the nodes in memory in preorder as shown in Figure 4.11. At fine levels, parent nodes are close to all of their children. At coarser levels, the children become separated by their own subtrees, so the offset from the parent to its last child grows. Our idea is to encode each offset y into a 7-bit code $x \in [0, 127]$ as $y = s_l \cdot x$ where s_l is a per-level scaling parameter. At finer levels where offsets are small, this encoding is wasteless with $s_l = 1$. At coarser levels where $s_l > 1$, if the desired offsets cannot be encoded exactly (i.e. are not a multiple of s_l), we leave some padding space between the subtrees.

We perform the packing in a fine-to-coarse order. For each level, having already packed the finer subtrees into memory blocks, we iteratively concatenate these subtree blocks after their respective parents such that they are addressable as encoded offsets from the parents – leaving padding space as needed. We exhaustively search for the integer scaling factor $s_l \in \{\lceil \frac{y_{max}}{127} \rceil ... y_{max}\}$ that gives the best packing (where y_{max} is the largest offset). Table 4.2 gives the optimized numbers for an example.

Level l	Num. nodes	Scaling s_l	Padding (bytes)
0	1	500	58
1	4	316	354
2	9	135	897
3	25	61	960
4	76	26	1016
5	202	10	555
6	486	5	0
7	1228	1	0
8	3218	1	0
9	8322	1	0

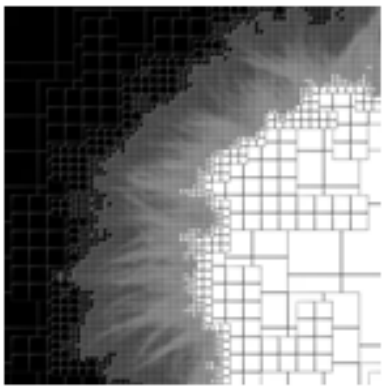


Table 4.2: *Result of offset encoding for the data shown right. A forested mipmap replaces tree levels 0-4*

Another strategy would be to pack nodes in level-order (equivalent to breadth-first search). However, such ordering would give offsets that are larger and less predictable.

Forested mipmap. Maintaining the coarsest levels as a tree structure has a number of drawbacks: (1) These levels are usually dense, so adaptivity is unnecessary; (2) The traversal of these coarse levels adds runtime cost; (3) Much of the padding space introduced by our offset encoding occurs there. For these reasons, we collapse the coarsest tree levels $0..4$ to form a (non-adaptive) mipmap pyramid. At the finest of these pyramid levels, we also store an indirection table with pointers to the resulting clipped subtrees. We call this overall structure a *forested mipmap*. For the same example in Table 4.2, the forested mipmap results in a decrease of 1847 bytes. Overall the tree topology requires 0.36 bytes/node for this quadtree.

Results. A typical result of our method is given Figure 4.12. It is important to recall that our scheme is competitive *only* when the image has smooth areas. Please refer to our publication for results on terrain height fields, high dynamic range imagery and distance fields.

I did not describe here an important contribution of our work: The way the data itself is compressed in the tree. While important from a compression point of view, in hindsight I believe the primal subdivision and tree

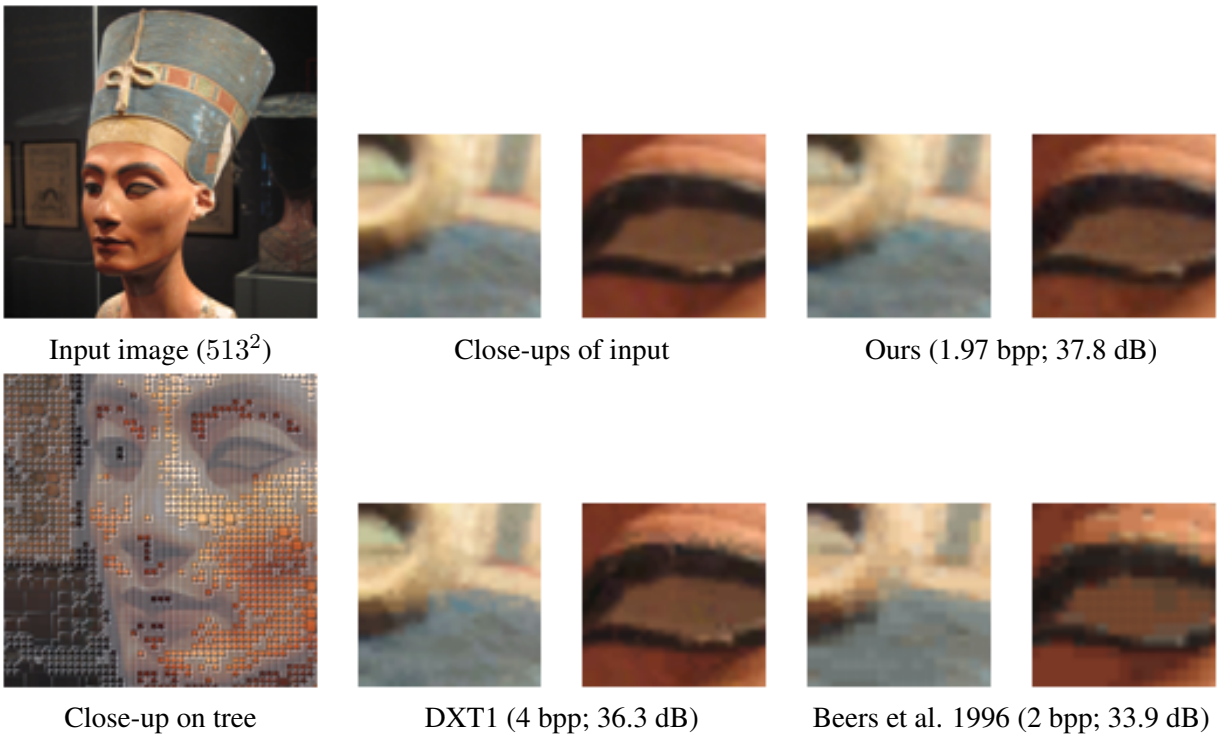


Figure 4.12: *Compression of a relatively smooth color image, compared with DXT1 compression and with uniform* 2×2 *block VQ using a 256-entry codebook.*

topology compression to be the most applicable elements of this work. They for instance directly apply to texturing applications, such as the one previously described in Section 4.1.1. In addition, while in our work the topology was compressed as a pre-computation on the CPU, it would be amenable to a parallel GPU algorithm using (for instance) a parallel scan primitive. This would provide on–the–fly compression of dynamically generated trees.

However, we will see next a number of techniques based on spatial hashing which even further reduce the overhead of the data–structure with respect to the data.

4.2 Spatial Hashing

Hierarchical spatial subdivisions are very popular in Computer Graphics to encode irregular distributions of primitives. These techniques encompass for instance octrees, kd–trees, bounding volume hierarchies (BVH) and BSP trees. In CG applications, such data–structures are often built off-line, and later used for efficient rendering; even though there has been a recent push for efficient construction of hierarchies [ZHWG08]. For texturing applications, these techniques can be used to store colors in the leaves of the hierarchy, only along the surface (see previous sections). Given a point p, the hierarchy can be traversed down to find the colors stored closest to p and compute the final color by interpolation. For interactive applications the access to the colors is performed from a pixel shader, for each and every pixel drawn on screen.

The main interest in hierarchical data-structures is their ease of construction, encoding and access. In particular, accessing the data stored at a point p in space typically takes $O(\log N)$ time, with N the number of color points. While this is efficient enough for many applications, there are a number of drawbacks:

- A pixel shader has to shade millions of fragments every frame, and therefore even a $O(\log N)$ complexity is a significant loss of efficiency.
- Accessing the hierarchy requires following pointers through *dependent* memory accesses. These dependencies imply that the memory access latencies cannot be hidden (amortized) behind computations.
- The pointers within the hierarchy incur a significant memory overhead.

Our work on parallel spatial hashing seeks to address these limitations. We introduced a first approach which pre-builds a hash table providing access to the data in constant time. This is achieved at the expense of a slow construction process. We later revisited this work to increase coherence of the data accesses and accelerate the construction process.

4.2.1 Notations and terminology

Spatial hashing takes as input a set of coordinates or *keys* $K \subset U$, where U is a d-dimensional grid of size $|U|^d$: each key is a d-uple of integers in $[0..(|U|-1)]^d$. U is called the universe, and is typically 2D or 3D. The keys are often encoded on 32 bits integers. Each key is also associated with a data record, for instance a 32 bits RGBA color.

K is the set of *defined* keys while $U \setminus K$ is the set of *empty* keys. Generally $|K| \ll |U|$. The hashing scheme encodes K in a hash table H and affords for fast random queries to discover whether a key $k \in U$ belongs to K and retrieve its associated data.

In our work the hash table is a d-dimensional grid of size $|H|^d$. Some entries of H may not be used: The *load factor* of a hash table is measured by the ratio $\frac{|K|}{|H|^d}$. The table H stores both keys and data packed together in integer records. Let $\texttt{key}(H[x])$ and $\texttt{data}(H[x])$ be respectively the key and the data record stored in $H[x]$. Access is performed through a *hash function* $h(i) : [0..(|U|-1)]^d \rightarrow [0..(|H|-1)]^d$. By definition we have for all $i \in K, \texttt{key}(H[h(i)]) = i$. Note that this assumes a *perfect* hash function, where $\forall p, q \in K, h(p) = h(p) \Rightarrow p = q$. Storing the keys alongside data incurs a significant overhead: This doubles the required size of the hash table. This can be avoided when the access is *constrained*, that is when H is only accessed from $k \in K$ (see Section 4.2.2).

4.2.2 Perfect spatial hashing

Our scheme revisits prior work in hashing in light of the specificities of texturing applications. We consider a fixed resolution 3D grid of colors surrounding the surface. For simplicity let us ignore interpolation and consider that a surface point p is colored by the nearest sample in the grid. The only useful information in the grid are the colors stored in cells intersecting the surface. This is a very sparse set, typically a few percent of the total grid size. We seek to store these colors in a hash, and retrieve them from a pixel shader during rendering. The set of defined keys are the 3D coordinates of the grid cells intersecting the surface, while their associated data are the RGBA colors.

We seek to build a *perfect*, *minimal* hash: Different defined keys map to different locations in H, and the load factor is 1. We further target a hash function that can be evaluated in constant $O(1)$ time, and with minimal additional storage.

An unusual aspect of our problem is that *only defined keys are accessed* in a texturing application. Indeed, only the surface points will be used by the pixel shader accessing the data encoded in the hash table. This has important practical consequences: the keys do not have to be stored alongside the data in this context. We call this scenario *constrained access*.

Theoretical bounds The probability that randomly assigning n elements in a table of size m results in a perfect hash is:

$$Pr_{PH}(n,m) = (1) \cdot (1 - \frac{1}{m}) \cdot (1 - \frac{2}{m})...(1 - \frac{n-1}{m})$$

When the table is large (i.e. $m \gg n$), we can use the approximation $e^x \simeq 1 + x$ for small x to obtain:

$$\begin{aligned} Pr_{PH}(n,m) &\simeq 1 \cdot e^{-\frac{1}{m}} \cdot e^{-\frac{2}{m}} ... e^{-\frac{n-1}{m}} \\ &= e^{-(1+2+...+(n-1))/m} = e^{-(n(n-1)/2m)} \simeq e^{-n^2/2m} \end{aligned}$$

Thus, the presence of a hash collision is highly likely when the table size m is much less than n^2. This is an instance of the well-known *birthday paradox* – a group of only 23 people have more than 50% chance of having at least one shared birthday. The probability of finding a minimal perfect hash (where $n = m$) is:

$$\begin{aligned} Pr_{PH}(n) &= \frac{n}{n} \cdot \frac{n-1}{n} \cdot \frac{n-2}{n} ... \frac{1}{n} \\ &= \frac{n!}{n^n} = e^{\log n! - n \log n} \simeq e^{(n \log n - n) - n \log n} = e^{-n} \end{aligned}$$

which uses Stirling's approximation $\log n! \simeq n \log n - n$. Therefore, the expected number of bits needed to describe these rare minimal perfect hash functions is intuitively:

$$\log_2 \frac{1}{Pr_{PH}(n)} \simeq \log_2 e^n = (\log_2 e)n \simeq (1.443)n$$

as reported by Fox [FHCD92] and based on earlier analysis by Melhorn [Meh82].

This theoretical bound is of course very hard to achieve, however it gives us a good baseline to evaluate the performance of our hashing scheme.

Perfect spatial hashing We draw inspiration from prior work in hashing character strings. The key hindsight from these works, in particular the works of [Sag85, FHCD92], is to first generate a hash with collisions, and then use an auxiliary table to resolve these collisions. Sager [Sag85] defines a hash $h(k) = h_0(k) + g_1[h_1(k)] + g_2[h_2(k)] \mod m$, where h_0, h_1, h_2 map string keys to $\mathbb{Z}_m, \mathbb{Z}_r, \mathbb{Z}_r$, respectively and g_1, g_2 are two tables of size r. Fox et al. [FHCD92] adapt this approach to create the first scheme with good average-case performance ($\sim 11n$ bits) on large datasets. Their insight is to assign values of auxiliary tables g_1, g_2 in decreasing order of number of dependencies. They also describe a second scheme that uses quadratic hashing and adds branching based on a table of binary values; this second scheme achieves $\sim 4n$ bits for datasets of size $n \sim 10^6$.

We follow this insight but exploit specificities of our problem to further simplify the hash function and reduce the additional storage required. We also propose a scheme to improve coherence of the accesses, although the results were somewhat disappointing in practice. This motivated our subsequent work on hashing described Section 4.2.3.

Access Let us start by describing access to the data through our hash function. We will later describe the hash construction process.

We define our perfect hash function as:

$$h(p) = h_0(p) + \Phi[h_1(p)] \mod |H|$$

where:

- Φ is the *offset table*: a d-dimensional array of size $|\Phi|^d$, containing d-dimensional vectors.
- $h_0 : p \to M_0 p \mod |H|$ is a simple mapping through a linear $d \times d$ transformation M_0, modulo $|H|$. We typically set $M_0 = I$, the identity matrix.
- $h_1 : p \to M_1 p \mod |\Phi|$ is a similar mapping from U to Φ. We typically set $M_1 = I$.

All operators (sum, modulo) are applied component per component on vectors.

The pseudo code in GLSL flavor for accessing our hash in 3D is:

```
1
2  sampler3D SOffset, SHData; // tables Φ and H.
3  mat3 M[2]; // M0, M1
4
5  // evaluates h(p)
6  vec3 ComputeHash(vec3 p)
7  {
8   vec3  h0 = M[0] * p;
9   vec3  h1 = M[1] * p;
10  vec3  offset = texture(SOffset, h1).xyz;
11  return h0 + offset;
12 }
13
14 void main()
15 {
16  // gl_TexCoord contains normalized 3D coordinates in U
17  ivec3 p = ivec3(gl_TexCoord[0].xyz * SizeU);
18  vec3 h = ComputeHash(vec3(p));
19  return texture(SHData, h);
20 }
```

Note that all modulo operations are implicit through the texture lookup functions. Accessing the data therefore requires only two memory accesses, and most arithmetic operations – especially the expensive modulos – are performed directly by the graphics hardware. In addition, the matrices M_0 and M_1 are often set to identity.

Hash construction We seek to assign table sizes $|H|$ and $|\Phi|$, hash matrices M_0, M_1 and offset values in table Φ such that:

1. $h(p)$ is a perfect hash function,
2. Φ and H are as compact as possible,

Given these parameters, the construction algorithm may not succeed. Our strategy is to let the hash table be as compact as possible to contain the given data, and then to construct the offset table to be as small as possible while still allowing a perfect hash function. More precisely, we introduce a greedy algorithm for filling the offset table, and iteratively call this algorithm with different offset table sizes. For a large enough offset table, success is guaranteed, so our iterative process always terminates with a perfect hash.

Selection of table sizes For simplicity, we assume that the hash table is a square or cube. We let its size $|H|$ be the smallest value such that $|H|^d \geq |K|$. Consequently, the perfect hash function is not strictly minimal, but the number of unused entries is small.

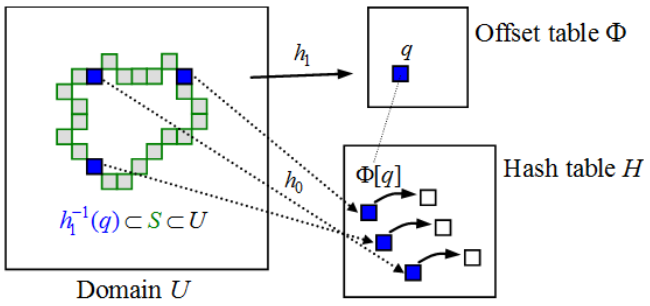


Figure 4.13: *The assignment of offset vector $\Phi[q]$ corresponds to a uniform translation of the points $h_1^{-1}(q)$ within the hash table.*

Next, we seek to assign the offset table size $|\Phi|$ to be as small as possible while still permitting a perfect hash. We describe different strategies in our paper. A simple one is to initially set the offset table to an optimistic size, such as 4 bits per entry, and increase in a geometric progression whenever construction fails.

Selection of hash matrices To our great amazement, we found that letting M_0, M_1 just be identity matrices does not significantly hinder the construction of a perfect hash. The functions h_0, h_1 then simply wrap the spatial domain multiple times over the offset and hash tables, moving over domain points and table entries in lockstep. This not only removes arithmetic operations, but also makes all memory accesses in table Φ perfectly coherent.

One necessary condition on h_0, h_1 is that they must map the defined data to distinct pairs. That is, $p \in K \rightarrow (h_0(p), h_1(p))$ must be injective. Indeed, if there were two points $p_1, p_2 \in K$ with $h_0(p_1) = h_0(p_2)$ and $h_1(p_1) = h_1(p_2)$, then these points would always hash to the same slot $h(p_1) = h(p_2)$ *regardless* of the offset stored in $\Phi[h_1(p)]$, making a perfect hash impossible. The condition for injectivity is similar to a perfect hash of n elements into a table of size $|H| \times |\Phi|$. Applying the same formula as for the theoretical bounds, we derive a probability of success of $Pr_{PH} \simeq e^{(-n^2/(2mr))} \simeq e^{(-n/2r)}$, which seems ominously low – only 12% for $\frac{|\Phi|^d}{|K|} = 0.25$. We believe that this is the main reason that previous perfect hashing schemes resorted to additional tables and hash functions.

However, unlike prior work we do not select h_0, h_1 to be random functions. Because our functions h_0, h_1 have periodicities $|H|$ and $|\Phi|$ respectively, if these periodicities are coprime then we are guaranteed injectivity when the domain size $|U| \leq |H| \cdot |\Phi|$. In practice, $|\Phi|$ is typically large enough that this is always true, and thus we need not test for injectivity explicitly. This is a major ingredient in achieving such a simple, efficient hash function for spatial data.

Creation of the offset table As shown in Figure 4.13, each entry q of the offset table is the image through h_1 of a small number of data points – namely the set $h_1^{-1}(q) \subset K$. The assignment of the offset vector $\Phi[q]$ determines a uniform translation of these points within the hash table. The goal is to find an assignment that does not collide with other points hashed in the table.

A naïve algorithm for creating a perfect hash would be to initialize the offset table Φ to zero or random values, look for collisions in $h(p)$, and try to 'untangle' these by perturbing the offset values, for instance using random descent or simulated annealing. However, this naïve approach fails because it quickly settles into imperfect minima, from which it would take a long sequence of offset reassignments to find a lower total number of collisions.

A key insight from [FHCD92] is that the entries $\Phi[q]$ with the largest sets $h_1^{-1}(q)$ of dependent data points are the most challenging to assign, and therefore should be processed first. Our algorithm assigns offset values greedily according to this heuristic order (computed efficiently using a bucket sort). For each entry q, we search for an offset value $\Phi[q]$ such that the data entries $h_1^{-1}(q)$ do not collide with any data previously assigned in the hash table, i.e., $\forall p \in h_1^{-1}(q), H[h_0(p) + \Phi[q]] =$ undef. If no valid offset $\Phi[q]$ is found, one could try backtracking, but we found results to be satisfactory without this added complexity. Note that towards the end of construction, the offset entries considered are those with exactly one dependent point, i.e. $|h_1^{-1}(q)| = 1$. These are easy cases, as the algorithm simply finds offset values that direct these sole points to open hash slots.

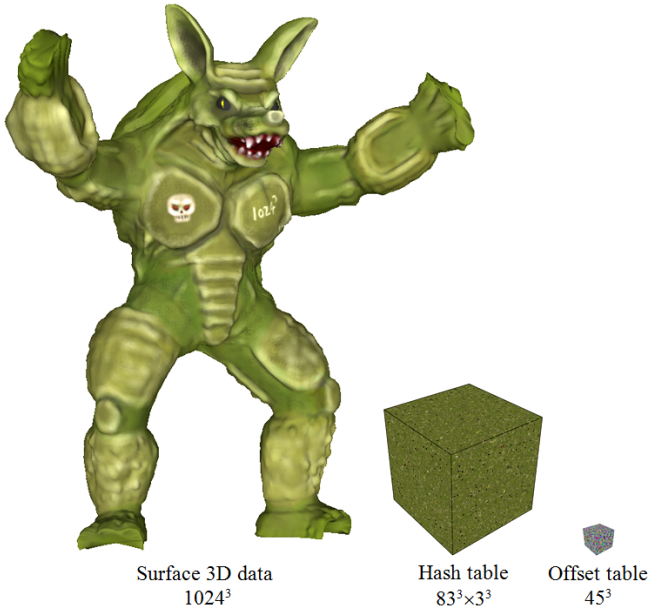


Figure 4.14: *Spatial hashing of 3D surface data. Block size* $b = 2$.

Results We applied this approach to encode a variety of sparse spatial data: Vector images, where information is stored only at discontinuities, alpha channel compression, 3D textures, and collision detection. On average, we were able to encode sparse data with an overhead of only 4 bits per defined key. Please refer to our paper for extensive analysis.

I report here only the results regarding 3D texturing, which is the main point of comparison with the texturing techniques presented earlier. Perfect hashing provides an efficient packed representation. Rather than storing individual colors we store blocks of size $b \times b \times b$ to exploit native trilinear filtering. This however implies an overhead since texels at the boundaries of the blocks are duplicated. The example of Figure 4.14 requires a total of 45.9 MB at optimal block size $b = 2$. The hashed representation renders at 530 fps, or 300 fps if including mipmapping, on an NVIDIA GeForce 7800 GTX with 256MB, in an 800^2 window.

For the same model, an octree constructed for nearest sampling occupies 18.0 MB and achieves only 180 fps. However, for a fair comparison, an unblocked spatial hash also constructed for nearest sampling takes just 7.5 MB and renders at 370 fps. Thus, a spatial hash can be more than twice as compact than an octree, which is in turn generally more compact than N^3-trees with $N > 2$.

It is interesting to note that our TileTree, introduced Section 4.1.2 is more efficient in space than the blocking scheme on this example (11.4 MB vs 45.9 MB). Hashing without blocking is more compact (7.5 MB) but is also slower to access with tri–linear interpolation.

Discussion Perfect spatial hashing provides a very simple hash function by exploiting specificities of both spatial data and the graphics processor. To my knowledge, it remains to this day the sparse spatial datastructure with the simplest access, both in terms of memory and arithmetic instructions.

In addition, in the constrained access scenario of 3D texturing the keys need not be stored alongside the data. Allow me to re–emphasize that the overhead is then of only 4 bits per *color sample* stored in the hash. Later works on hashing (including our own) often *require* to store 32 bits keys alongside the data, even in a constrained access scenario. This incurs a much larger penalty. This property does not hold however in the unconstrained access scenario. We proposed several strategies for this case in our publication.

The drawbacks of perfect spatial hashing are two–fold: First, the construction process is slow. Parallelizing the construction is made very difficult by the varying sizes of the conflicting sets $h_1^{-1}(q)$. We actually attempted it later, but it resulted in a complex implementation and the performance has been disappointing. Second, there is little coherence in the accesses. Two neighboring keys in U will end up in very different locations in H. We describe in our publication a technique to significantly improve the coherence. While the improvement can be clearly measured, it however did not translate in a significant improvement of performance. In hindsight, I

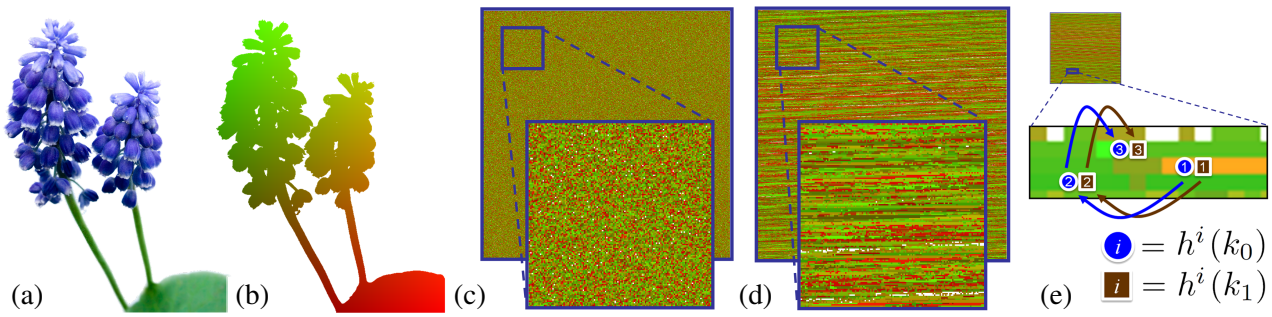


Figure 4.15: (a) *The flower image is* 3820×3820 *image (*14.5 *million pixels) and contains* 3.7 *million non–white pixels. The coordinates of these pixels are shown as colors in* (b)*. We store the image in a hash table under a* 0.99 *load factor: the hash table contains only* 3.73 *million entries. These are used as keys for hashing.* (c) *The table obtained with a typical randomizing hash function: Keys are randomly spread and all coherence is lost.* (d) *Our spatially coherent hash table, built in parallel on the GPU. The table is built in* 15 *ms on a GeForce GTX 480, and the image is reconstructed from the hash in* 3.5 *ms. The visible structures are due to preserved coherence. This translates to faster access as neighboring threads perform similar operations and access nearby memory.* (e) *Neighboring keys are kept together during probing, thereby improving the coherence of memory accesses of neighboring threads.*

believe this is due to the granularity at which we were able to provide coherence. In order to achieve efficient coalesced memory accesses, a relatively large number of threads should access neighboring memory locations. Unfortunately, our coherence optimization was only able to create small clusters of coherence; these were likely too small to result in a clear advantage.

A few years later, Alcantara et al. [ASA+09] introduced a hashing scheme based on a different type of hash functions, known as Cuckoo hashing. This scheme is able to construct a hash at runtime, inserting millions of keys in few milliseconds. This however comes at the expense of a slightly more complex access. Unfortunately, this scheme is also suffering from a lack of coherence in the accesses. Another drawback is that the failure rate quickly increases at high load factors (typically above 80%). I next present our hashing scheme addressing these issues.

4.2.3 Coherent dynamic hashing

Most available spatial hashing schemes suffer from two drawbacks: Multiple runs of the construction process are often required before success, and the random nature of the hash functions decreases access performance.

The first spatial hashing schemes focused essentially on reaching good load factors while having a constant time and simple access to the data from the GPU. Our perfect spatial hashing scheme [LH06b] proposed a static hash construction enabling access to the data with as little as two memory accesses and one addition. However, to achieve this result the hash has to be *perfect*: All keys corresponding to defined data should map to different locations in the hash table. In other words, there are no *collisions*. Building such a constrained hash function requires an off–line construction process, limiting our approach to static cases.

Alcantara *et al.* [ASA+09, AVS+11] propose less constrained hashing schemes that achieve fast, parallel construction on the GPU. These schemes may produce collisions. However, querying a key never requires more than four independent memory accesses. The particular hash mechanism they use is known as *Cuckoo hashing*. We detail it in the next section.

All of these approaches achieve constant query time with a fixed number of instructions. Unfortunately, these constraints imply that construction is difficult and the process may fail, requiring several restarts especially at high load factors. In coherent hashing, we relax the constraint of accessing data with a fixed number of instructions. Instead, we implement queries with few memory fetches *on average*. The increased flexibility in the access enables a more robust construction process. However, a small average does not guarantee good performance as empty keys are generally detected only after trying the *maximum* number of accesses required to find a defined key. We propose a mechanism to quickly reject empty keys, thereby significantly reducing their negative impact.

In addition, we tailor our scheme to exploit the spatial coherence of rendering algorithms. In existing schemes,

neighboring keys are often mapped to distant locations in the hash table. This is an issue since graphics hardware is designed to benefit from spatial coherence: Threads are organized in a grid and best access performance is achieved when nearby threads access nearby memory locations. Linial and Sasson analyzed a *non-expansive* hashing scheme to bring similar keys close to each other in the hash table [LS96]. That scheme, however, necessitates too much space to be practical in graphics applications.

Cuckoo hashing Alcantara *et al.* [ASA+09] introduced the first algorithm enabling fast, parallel hash table construction on the GPU. Millions of keys are efficiently hashed in milliseconds, outperforming previous schemes by several orders of magnitude.

A Cuckoo hash [PR04] maintains two or more different tables of the same size, each accessed through a different hash function—Alcantara *et al.* [ASA+09] use three tables. Keys are inserted in the first table, evicting already inserted keys in case of collision. Evicted keys are in turn inserted in the second table, then from the second to the third, and from the third back to the first. The process loops around until all keys are inserted or until the number of iterations reaches a maximum—which triggers a construction failure. Upon failure the process is restarted with different hash functions. Unfortunately, given a number of tables the failure rate abruptly increases above a limit load factor. Using more tables increases this limit. However, using too many tables becomes wasteful at lower load factors. In contrast, our scheme automatically adapts to these various situations.

The parallel construction algorithm builds a Cuckoo hash in shared memory—a small but very fast memory. It starts by randomly distributing the keys in equally sized buckets using a first level hash [BZ07]. Any bucket overflow triggers a construction failure. This limits the maximum possible load factor to 0.7: higher load factors give a too large failure rate for this key distribution phase. Next, all buckets are hashed independently in parallel with Cuckoo hashing. In subsequent work Alcantara *et al.* [AVS+11] build a Cuckoo hash in a single pass. The single pass approach is made possible by latest hardware capabilities (efficient atomic operations on NVidia Fermi devices). Their new hashing scheme also reaches higher load factors thanks to the use of four hash functions. Both schemes further introduce handling of multi-value hashing and duplicate keys in the input.

The Cuckoo scheme behaves very well in practice, and the guarantee of a constant number of memory accesses to query a key is well adapted to GPUs. Its main drawback stems from an increasing failure rate at high load factors requiring to manually select more hash functions, and the loss of coherence due to randomization. Another less obvious issue is that while a fixed number of lookups are required, the average number of lookups is often higher than that of our scheme as keys tend to be uniformly distributed in all the tables, even at lower load factors. We compare our approach to Cuckoo hashing in our publication, and I will discuss the common points and differences in the concluding paragraph (Section 4.2.3).

Parallel coherent hashing Our hash is designed to reach high load factors at low failure rate, and to provide fast queries regardless of the load factor. The key insight is to exploit dynamic branching to release the constraints on the construction process, and to exploit coherence in the access patterns when available.

Our algorithm builds a unique, large hash table in one pass. Parallelism is obtained by launching many thread groups simultaneously. Each thread is responsible for inserting exactly one key.

Our hash is at heart an open addressing scheme [Pet57]: Each input key k is associated with a sequence of probing locations $h^1(k), h^2(k), \ldots$ in the hash table. Ideally, this *probe sequence* should enumerate all locations in finite time. For now, we assume that the sequence is given. We introduce our coherent probe sequence later.

In order to add a key, the insertion algorithm iterates along the probe sequence until an empty location is found. The key is then inserted. We call the number of steps required for successful insertion the *age* of a key. If the hash table can fit all the defined keys, then the process is guaranteed to succeed as long as the sequence $h^i(k)$ enumerates all locations. Therefore the algorithm is quite robust to changes in the hash function. Querying a key proceeds similarly to construction, by walking along the probe sequence until the key is found in the hash table.

However, open addressing suffers from a severe drawback for our purpose: The age of the keys is very low on average but typically has a large maximum. This maximum age, noted M, is crucial: When querying an empty key, its absence from the hash table can only be verified by walking along the sequence of keys at least

Figure 4.16: *Hashing 2^{20} defined keys randomly distributed in a universe of 2^{24} keys into a hash table of 1.3 million keys (the load factor is 0.8). Histogram of insertion ages for open addressing (top) and Robin Hood (bottom). Gray bars correspond to very few items but are non–zero. The maximum age goes down from 46 to only 5.*

M steps. Since the data set is sparse, a large number of queries to empty keys is expected in many applications. The overall performance can dramatically suffer. Note that hitting an empty location during a query before reaching M steps is unlikely, especially under high load factors.

We next discuss how to efficiently reduce the maximum age and reject empty keys.

Reducing the maximum age The maximum age issue has already been identified and studied in previous work. A very effective solution to it is known as Robin Hood hashing [Cel86], which is based on open addressing.

The idea is to store the age of the keys in the hash table during its construction. This additional data is discarded afterwards. Consider the case of inserting a key k_{new} at a location $h^i(k_{\text{new}})$ already occupied by a key k_{prev}. The age a of k_{prev} is compared to i. If the key being inserted is older ($i > a$), then k_{prev} is evicted. The current key is inserted at $h^i(k_{\text{new}})$ and k_{prev} is recycled into the set of keys to be inserted. Intuitively, the keys which are difficult to insert push away the keys which have been easier to insert. This has a drastic effect on the histogram of key ages, and in particular on the maximum age as illustrated Figure 4.16.

There are two particularly important theoretical facts about Robin Hood hashing making it especially well suited to our purpose: The expected maximum age in a *full* table of size n is $\Theta(\log n)$ [Cel86]. Devroye *et al.* [DMV04] further refine the bound to $\Theta(\log_2 \log n)$ on non full tables. Furthermore, the expected query time complexity can be made constant if starting the accesses at the average age. Note that these facts are derived assuming uniform random sparse data, but our experiments show that they hold in practice on our datasets.

In our current implementation we do not start the accesses at the average age: For simplicity we always start from age one, searching for the key until the maximum age for the sequence is reached.

Empty key rejection While Robin Hood hashing strongly reduces the maximum age M, it remains quite large compared to the few memory accesses of Cuckoo hashing. This is especially important in applications querying many empty keys, as they always require M steps along the sequence. We therefore introduce a new mechanism to accelerate the rejection of empty keys.

We note that most keys have a very small age, with only a few outliers ever reaching M. Therefore, in most cases a much smaller value than M would suffice to detect empty keys. To benefit from this we store in each entry of the hash table the maximum age of all the keys mapping *first* to this location. More precisely, let $H[p]$ be the key stored at location p in the hash table H, let $\text{MAT}[p]$ be the maximum age starting from p, then we have:

$$\text{MAT}[p] = \max_{\{k \in D | h^1(k) = p\}} \left(i \text{ st. } H[h^i(k)] = k\right) \tag{4.1}$$

When querying a key k we iterate at most $\text{MAT}[h^1(k)]$ times along the probe sequence. This guarantees that defined keys are found, and affords for fast detection of empty keys.

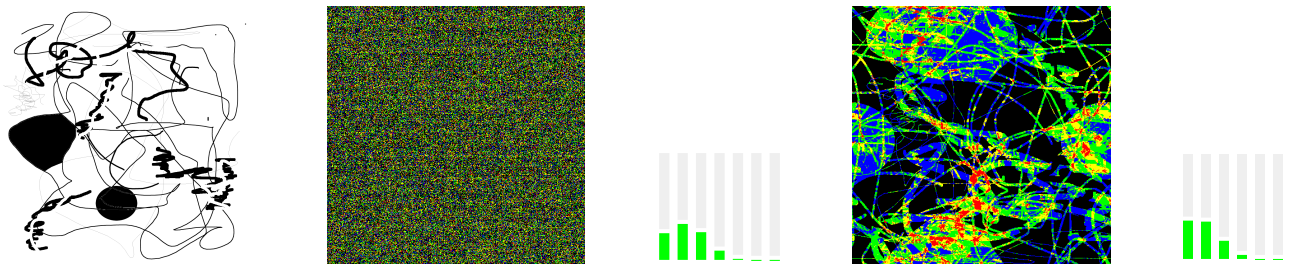


Figure 4.17: *Maximum age table for* data$test1$ *hashed at* 0.8 *load factor. Color code: 0, black; 1, blue; 2, green; 3, yellow; greater, red.* Left: *Test image.* Middle: *Random probe sequence.* Right: *Coherent probe sequence. Key age histograms are comparable, but note how coherence is maintained in the map on the right. We can expect more efficient dynamic branching when branching with respect to this map.*

Encoding the maximum age Storing the maximum age table MAT requires additional memory. Fortunately, the maximum age values are small and only require a few bits. In all our tests, the maximum age was below 16. Thus, we reserve 4 bits in each hash table cell to store the maximum age. The latter can optionally be quantized to either accommodate larger values or to reduce further the number of bits to 3 or less.

Let us assume each key is stored on k bits and their age is stored on a bits. When the user targets a data structure p times larger than the defined keys, we allocate $pk|D|$ bits of memory. In fact, due to the additional storage of the maximum age, this corresponds to a hash table storing $\frac{pk|D|}{(k+a)}$ keys. Thus, the keys will be hashed at load factor $\frac{(k+a)}{pk}$. For example, in a typical situation, we have $k = 28$ bits, $a = 4$ bits. Then, targeting a storage of $1.2 \times k \times |D|$ bits—that is 1.2 times the size of the defined keys, or a 0.83 key-density—requires hashing at 0.95 load factor, while targeting a 0.7 key-density requires a 0.82 load factor. With 64 bit cells and 60 bit keys, these load factors become 0.89 and 0.76 respectively.

The hashing scheme of Alcantara *et al.* [AVS+11] requires no additional information apart from the key, in which case the load factor and the key-density are equal.

Exploiting coherence In many computer graphics applications, data is queried in a coherent manner: Either spatial coherence within a frame, or temporal coherence due to limited motion between frames. We design a new probe sequence tailored to exploit coherence in the queries.

Note that we seek coherence of memory accesses *between neighboring threads*. This is quite different from the typical CPU notion of cache coherence where one seeks to access nearby memory locations *in sequence*. On the GPU, groups of threads access memory *simultaneously*, and it is important to group the accesses in nearby locations. This is also known as *coalesced* accesses. Similarly, the hardware performs faster when groups of threads take similar branching decisions.

Our objective is to design a different probe sequence for the keys, which favors coherence. Our new probe sequence h_{coh} preserves coherence by making neighboring keys test neighboring locations—while still ensuring that the *successive* locations of a same key are uncorrelated and perform a random walk. It corresponds to random translations of the *entire* hash table at each step i. That is, for a key k at step i in a hash table of size $|H|$:

$$h^i_{\text{coh}}(k) = k + o_i \mod |H|$$

where o_i is a sequence of offsets, independant of k. We typically set $o_0 = 0$ and use large random numbers as offsets.

An important property of this probe sequence is that neighboring keys remain neighbors at each step, as illustrated Figure 4.15(e). Therefore, if two neighboring threads attempt to find neighboring keys, they will both always access neighboring memory locations until one terminates. Note that these keys do not have to be present in the hash table for coherence to happen at the thread level during queries. In fact, access coherence during queries is orthogonal to the distribution of the defined keys. Sparse random data encoded with our hash will still benefit from being queried in a coherent manner.

The map MAT encoding the maximum ages also benefits from a coherent hash function: It exhibits a much stronger spatial coherence. This implies that neighboring threads will perform similar data–dependent loops,

reducing divergence. In Figure 4.17, for illustration purposes we hash a random image in a 2D hash table – we extend the hash to 2D by applying the same computations independently to each dimension. We display the maximum age table MAT for a random and a coherent probe sequence. The second image (coherent) exhibits structure and coherence, and thus affords for more efficient dynamic branching than the first.

Our coherent probe sequence outperforms the random one when coherence is present. It strongly reduces cache misses, thread branch divergence and results in significantly faster queries. We measure these effects on various data sets next.

Results and discussion The performance of our scheme is impacted by several factors: The number of keys to be hashed, the target load factor, and whether coherence exists in the data and the access. All our tests are performed with a NVidia Fermi GTX 480 GPU. Please refer to our publication for the implementation details.

While an in-depth analysis is provided in our paper, let me focus here on the most significant result. Our scheme is best suited when data is coherent – defined keys tend to be neighboring – and when data is also accessed in a coherent manner. Coherence in the data helps the construction process; However a random set of keys still benefits from a coherent access due to thread locality.

In a typical Computer Graphics application the hash table stores a sparse, structured, set of elements (texels, vector primitives, voxels, particles, triangles) which are accessed with some degree of spatial coherence. In most scenarios, a large number of empty keys are also queried (unconstrained access scenario).

2D data

We first consider 2D data sets. Our test consists in hashing a sparse subset of the pixels of a 2D image (e.g. all the non white pixels), and then query *all* pixels of this image to reconstruct it. In the tests below, keys are computed from the 2D pixel coordinates with a row-major ordering. We later discuss the impact of different pixel orderings.

Figure 4.18 reports construction times for both a random data set and the `data`$fish$ data set. Under a 0.8 load factor and for 16M (million) keys, our scheme achieves an insertion rate of 249 Mkeys/sec. In comparison, [AVS$^+$11] achieves 318 Mkeys/sec and [ASA$^+$09] achieves 268 Mkeys/sec. Therefore, in absence of coherence our insertion scheme is slower than both versions of the cuckoo scheme. This is essentially due to the update of the max-age table `MAT`, and the emulation of the 64 bits `atomicMax`. The important observation is that coherence in the `data`$fish$ data set—the existence of many neighboring keys—directly results in improved construction performance. The `data`$fish$ data set contains $20.5M$ keys. Under a 0.85 load factor, the random sequence reaches 142 Mkeys/s while our coherent hashing scheme achieves 368 Mkeys/s – an improvement of 159%. Thanks to coherence our scheme is on par with parallel cuckoo hashing for construction times. In contrast, on random data the coherent sequence has similar performance as the random sequence.

Figure 4.19 reports the time taken to retrieve *all* the keys, both defined and empty. The distinction between querying empty or defined keys is important since empty keys are typically the most expensive to retrieve. In our scheme their cost is greatly reduced by using the the max-age table. The results are shown in Figure 4.19 for both a random set of keys and the `data`$fish$ dataset. Clearly, both the `data`$fish$ and random data sets benefit from coherence in the access. These results are consistent across all the datasets we tested. Under a 0.85 load factor our coherent hash retrieves 5324 Mkeys/s, while all other schemes achieve less than 1000 Mkeys/s: Coherence brings a very significant improvement in query performance.

The benefit of our coherent probe sequence is also clearly revealed by the *percentage of global cache hit* during queries, as shown Figure 4.20. No other scheme in our tests made any significant use of the cache. Figure 4.20 shows only L1 cache data. We ran additional tests to reveal further improvements in the number of L2 cache read requests and DRAM read requests, as reported in Figure 4.21 together with the measured branch divergence rate. In Figures 4.20 and 4.21, the data is taken from the above experiment with the `data`$fish$ data set.

Key layout

We now analyze the effect of different orderings of the 2D data in the 1D key universe. This is important as in many graphics applications of hashing, the keys are queried in a systematic and coherent way. For example, threads in a same group rasterize neighboring pixels that have neighboring texture coordinates, so we should

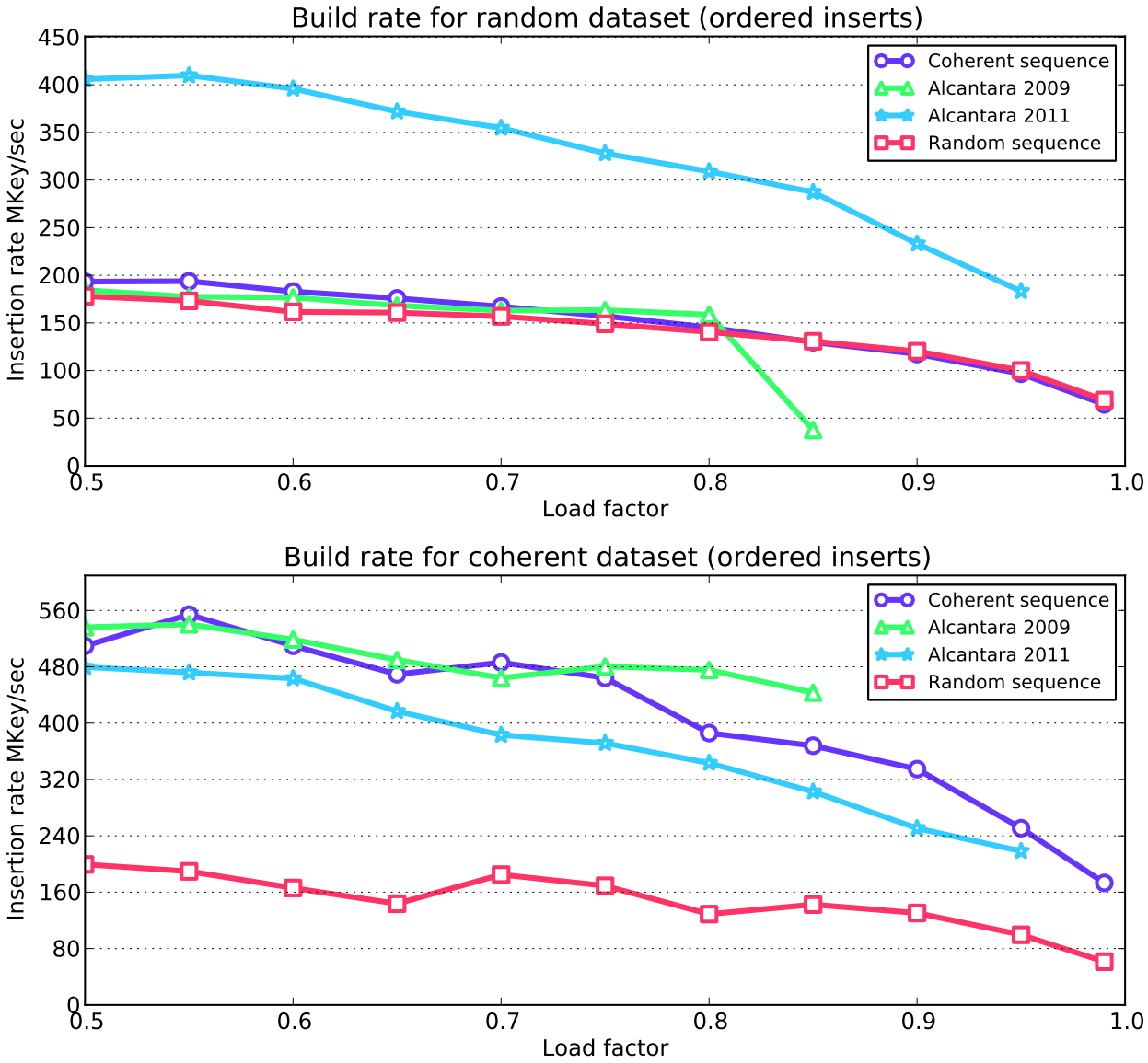


Figure 4.18: *Construction times for h_{rand} and h_{coh} and earlier schemes. Times are averaged over several runs.* Top: $1M$ *random keys are inserted, taken from a universe of size* $8K \times 8K$. Bottom: *Timings for the* `data`$fish$ *image:* $20.5M$ *are defined in a universe of size* $8K \times 8K$.

strive to keep this coherence when translating the position or the texture coordinates into 1D keys. We test three orderings:

- The *linear row-major* order maps (x, y) to $x + Wy$ when W is the width of the (rectangular) domain: we should benefit from the coherent hash when we query neighboring keys on a same line of the domain.
- The *Morton* order maps (x, y) to $\mathcal{M}(x, y)$, the integer obtained by interleaving the bits of the binary representation of x and y. This improves locality along both X and Y axes.
- The *bit-reversal* permutation $\sigma = (\sigma_i, i \in [0, 2^w))$ is obtained by reversing the bits of the binary representation $b_1^i b_2^i ... b_w^i$ of integer i: $\sigma_i = b_w^i b_{w-1}^i ... b_1^i$. For the experiment, we map (x, y) to $\mathcal{M}(\sigma_x, \sigma_y)$. This mapping exhibits no coherence at all.

The test consists in drawing a full-screen rotating image using a custom GLSL pixel shader to query the hash map. The latter is stored as a 2D texture and encodes the image color data using the three orderings above. Since GLSL offers fewer opportunities for optimization, the test reports overall lower performance than the CUDA implementation. The results are shown in Figure 4.22. The random probe sequence behaves roughly the same for each ordering and even gives slightly faster queries with the highly incoherent bit-reversal ordering. On the contrary, the coherent probe sequence gives significantly faster queries on the two other orderings since it

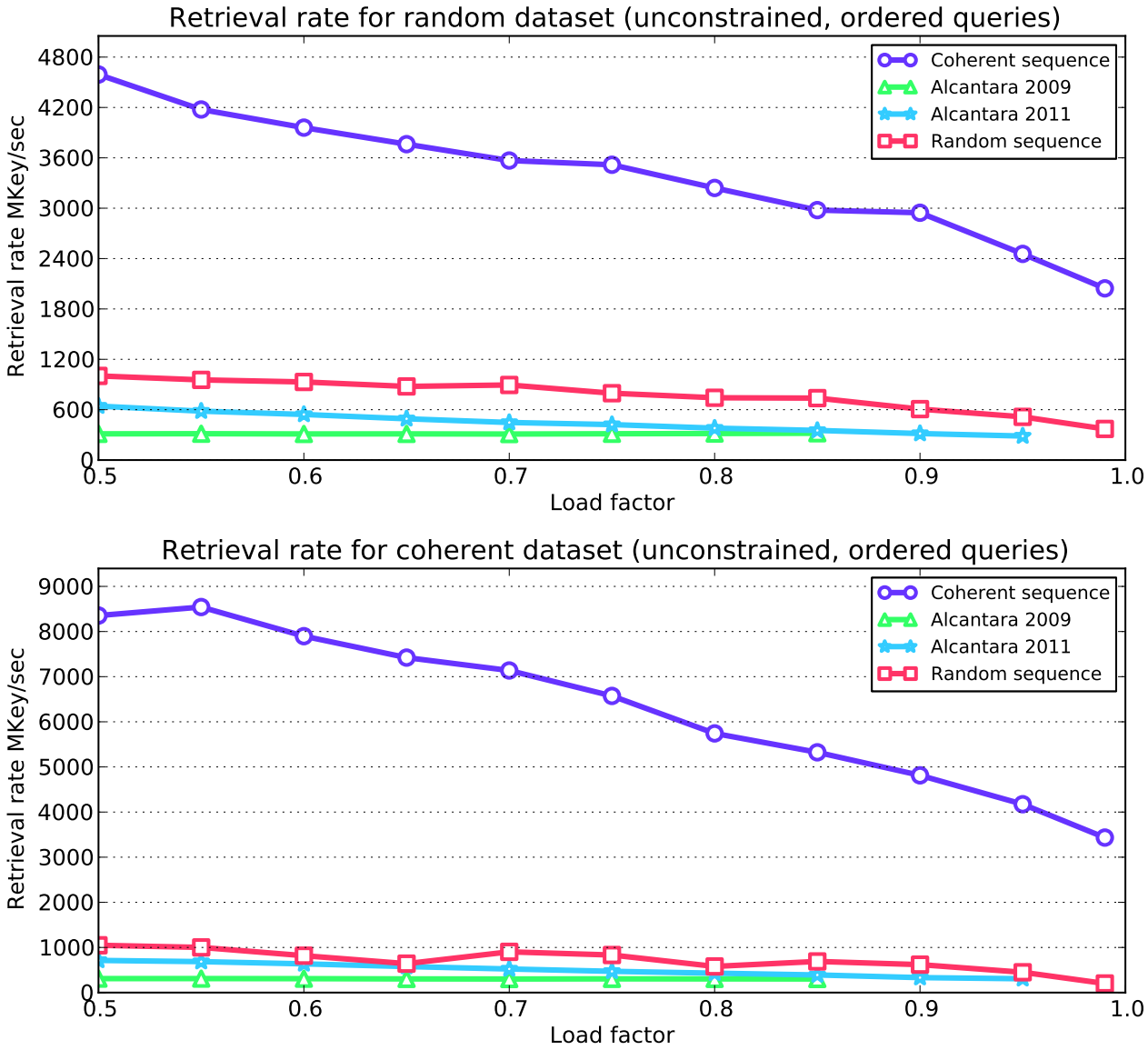


Figure 4.19: *Access times for h_{rand} and h_{coh} and earlier schemes. Times are averaged over several runs. Missing data for Alcantara09 is due to construction failure at high load factors.* Top: *$8K \times 8K$ keys are queried, $1M$ of which, chosen at random, are defined.* Bottom: *Timings for the* `data`$fish$ *image: $8K \times 8K$ keys are queried, of which $20.5M$ are defined.*

leverages coherence in the access pattern. One can clearly see how increasingly coherent orderings translate into higher performance.

3D data

We have experimented with 3D data as well, consisting in voxelizations of the `data`$armadillo$ and `data`$hairy$ models (see Figure 4.23) in a grid of size 512^3. We hash 64 bits key-data pairs. Our experiments consisted in drawing slices of the volume at random orientations. The `data`$armadillo$ voxel data results in 9.2M keys. Insertion rate is 280 Mkeys/s under load factor $d = 0.8$ and 254 Mkeys/s at $d = 0.99$. Retrieval rate is 1905 Mkeys/s at $d = 0.8$ and 1182 Mkeys/s at $d = 0.99$. The `data`$hairy$ voxel data results in 24M keys. Insertion rate is 455 Mkeys/s at $d = 0.8$ and 182 Mkeys/s at $d = 0.99$. Retrieval rate is 1736 Mkeys/s at $d = 0.8$ and 1208 Mkeys/s at $d = 0.99$.

Regarding the key layout, we obtain similar results as the 2D results shown in Figure 4.22, with an even stronger advantage to the Morton ordering.

Hindsight: links between Cuckoo hashing and our scheme There are interesting connections between Cuckoo hashing and our approach. In a sense, Cuckoo hashing corresponds to an open addressing scheme

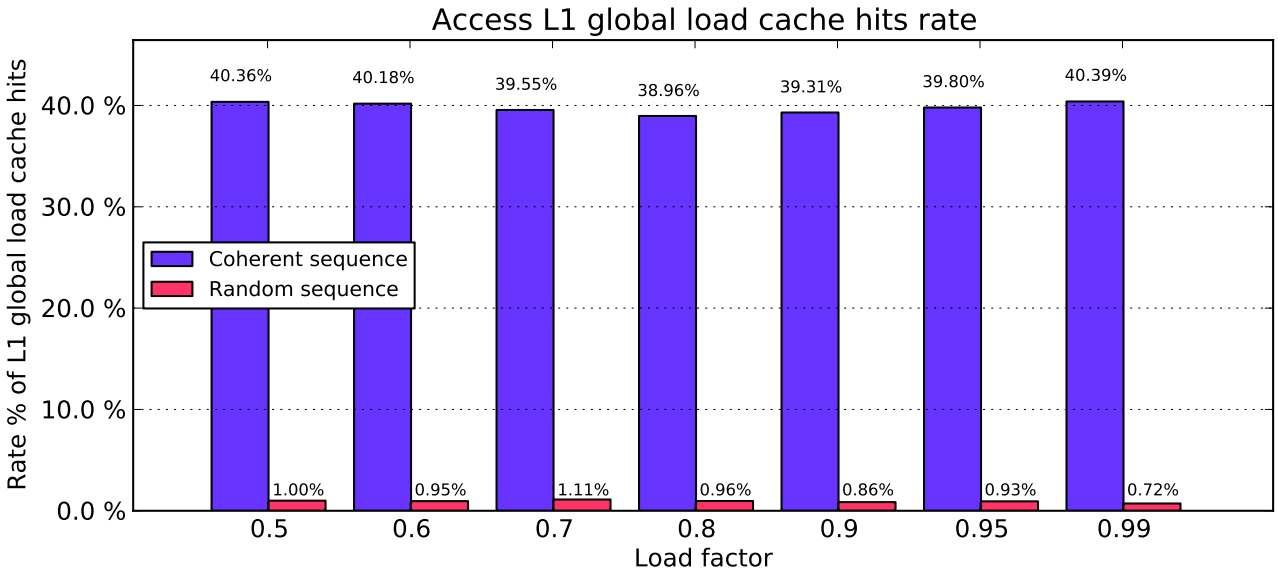


Figure 4.20: *Percentage of L1 global cache hit during queries for the* `data`$fish$ *data set. The higher the better. Note that only our coherent probe sequence exhibits a significant cache reuse.*

	Time	L2 requests	DRAM requests	Branch divergence
h_{rand}	97.1 ms	458.6 M	458.6 M	21.8 %
h_{coh}	12.6 ms	44.2 M	42.9 M	10.4 %

Figure 4.21: *L2 and DRAM read requests for the* `data`$fish$ *data set under 0.85 load factor.*

where the maximum age would be bounded. Indeed the N tables can actually be merged into a single larger table and a fixed number of N hash functions [AVS+11]. In Cuckoo hashing, whenever a key reaches the last hash function, it restarts from the first. This is akin to limiting the insertion age by a modulo operator in our scheme.

Cuckoo could be modified to use our coherent hash. However, because of this reset of the maximum age the insertion process is much more likely to fail. This is in particular due to the lack of Robin hood strategy in Cuckoo hashing. In addition our scheme automatically adapts: At high load factors the maximum age will increase to ensure all keys can be inserted. Both approaches share similarities, but we believe our scheme to be better suited for a coherent hash sequence.

In applications where coherence does not exists, neither in the data nor in the access pattern, Cuckoo hashing is to be recommended: its main advantage is that it does not need to keep track of the maximum age during the construction process.

4.2.4 Applications

In addition to the core algorithms we proposed new approaches based on hashing for specific long standing problems in Computer Graphics:

- In a collaboration with the KIT we developed a novel algorithm for caching data being continuously synthesized during rendering. In this particular case we cache the evaluation of a distance field function $\Phi(p) : \mathbb{R}^3 \to \mathbb{R}$ which computes in any point of space the distance to a surface. This distance field is used for interactive rendering of implicitly defined surfaces: A ray is traced in each pixel, and the distance field is used to find the intersection with the surface. We rely on the sphere–tracing algorithm [Har94] which is very efficient but still requires several evaluations of Φ along the ray (typically in $O(\log L)$ with L the distance to the surface from the viewpoint along the ray). The function Φ can be expensive to compute. Therefore, we store its evaluations in a cache: Prior to computing Φ we verify whether the value at p is cached and skip the evaluation if possible. The originality of our approach is to propose a caching mechanism that seamlessly integrate with the rendering pipeline. This is achieved through a

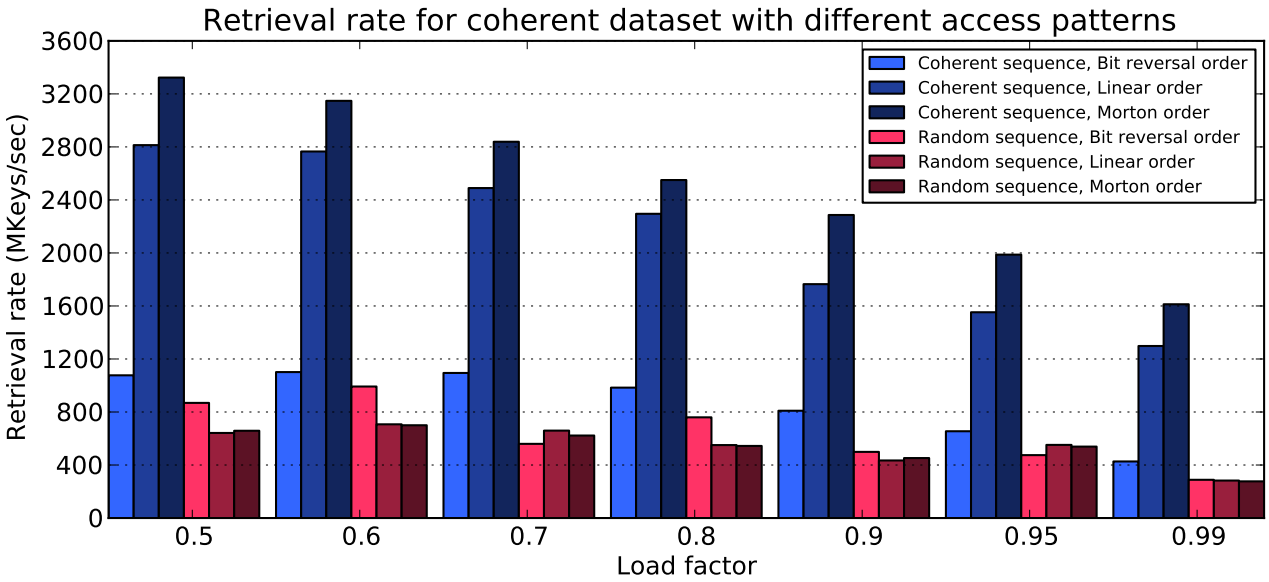


Figure 4.22: *Query timings for h_{coh} using different ordering of the pixels, for the* `data`$fish$ *data set.*

hashing mechanism similar to our coherent hash (see Section 4.2.3), but we exploit the age of a key to define an automatic cache eviction policy. This work has been published in the journal Computer and Graphics in 2012 [RLD+12].

- We recently introduced a novel approach for A–buffer rendering. The A–buffer [Car84] was introduced as a technique to render objects with transparency, but it has other applications such as per–pixel CSG and voxelization. The A–buffer records in each pixel *all* the surfaces which appear below, as opposed to only the closest one. Recording all the fragments as they are produced is very challenging: The bandwidth is huge (Giga pixels per seconds) and the number of fragments per pixel is not known in advance. Furthermore, it is necessary to order the fragments by depth before rendering. While there is a large body of work in this area, to the best of our knowledge no technique could simultaneously insert and sort the fragments: Most approaches require several passes or iterations to achieve this, resulting in implementations more difficult to integrate in the rendering framework. Instead our approach inserts and sorts in a single pass. This is made possible by exploiting some very specific properties of the coherent hash. This work is currently published as a technical report [LHL13]. It has been accepted for publication as a chapter in the upcoming GPU Pro 5 book [LHL14].

4.3 Discussion

My work on GPU datastructures has been a perfect companion research to my work on runtime content generation. It provided all the necessary algorithms to efficiently store and access the data that was produced by our synthesis algorithms.

This line of work is also rich in spin-off applications, including applications outside of texturing. For instance, our work on hashing recently resulted in a specialized algorithm for performing CSG–modeling and preparing

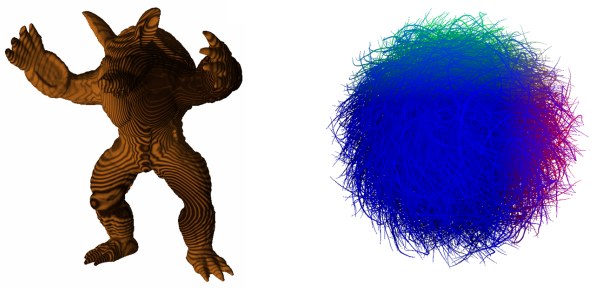


Figure 4.23: *The* armadillo *and* hairy *color voxel data.*

objects for fabrication on 3D printers. This algorithm is now at the heart of my 3D modeling and 3D printing software IceSL `http://webloria.loria.fr/~slefebvr/icesl`.

Many of our data–structures are based on well known principles (octrees, hashing, etc.). Our contributions have been to revisit them in light of specific properties of Computer Graphics rendering algorithms: Extremely high bandwidth requirements, huge datasets (volume textures), dynamism with an unbalance between insertion cost and the highly critical access cost, need to avoid memory dependent accesses and necessity of performing coalesced memory reads. All of these criteria are hardware **in**dependent, and all of them apply across many variants of parallel processing platforms.

We constantly discover new opportunities to expand our set of algorithms. I therefore intend to continue this line of research. We also expand the applicability of our techniques, in particular by exploiting parallel processors to perform discrete optimization, for instance solving a variety of dynamic programming problems [LL12, ZLL13].

Chapter 5

The next steps

> *Things are only impossible until they're not.* Hannah Louise Shearer, Star Trek: The Next Generation, When The Bough Breaks, 1988

As researchers we are given the unique opportunity to attempt to push the boundaries of what we think is impossible. Amongst my most vivid memories are the first real time texture synthesis results (especially the drag and drop results!), our paint application using the first GPU accelerated hierarchy and spatial hashing, and the always renewed amazement in front of an algorithm actually producing novel images from examples.

A recent development of my research is that we are now applying the ideas we developed for runtime texture synthesis to 3D printing and additive manufacturing. It was one thing to visualize our results on screen, but creating tangible objects out of our algorithms is simply fascinating.

To explore these ideas further I proposed the project *ShapeForge*, which was funded in 2012 by an ERC Starting Grant. Please visit my website to see our latest results!

Bibliography

[ADA+04] Aseem Agarwala, Mira Dontcheva, Maneesh Agrawala, Steven Drucker, Alex Colburn, Brian Curless, David Salesin, and Michael Cohen. Interactive digital photomontage. *ACM Transactions on Graphics*, 23(3):294–302, August 2004.

[ALM10] Jean-François Aujol, Saïd Ladjal, and Simon Masnou. Exemplar-based inpainting from a variational point of view. *SIAM*, 42(3):1246–1285, 2010.

[ASA+09] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the GPU. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2009)*, 28(5), December 2009.

[Ash01] Michael Ashikhmin. Synthesizing natural textures. In *Proceedings of the ACM Symposium on Interactive 3D Graphics*, pages 217–226. ACM Press, 2001.

[AVS+11] Dan A. Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Building an efficient hash table on the GPU. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 2, chapter 1. Morgan Kaufmann, 2011.

[BAC96] Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha. Rendering from compressed textures. In *Proceedings of SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 373–378, August 1996.

[BD02] David Benson and Joel Davis. Octree textures. In *Proceedings of SIGGRAPH*, pages 785–790, 2002.

[Bes86] Julian Besag. On the Statistical Analysis of Dirty Pictures. *Journal of the Royal Statistical Society. Series B (Methodological)*, 48(3):259–302, 1986.

[BJEYLW01] Ziv Bar-Joseph, Ran El-Yaniv, Dani Lischinski, and Michael Werman. Texture mixing and texture movie synthesis using statistical learning. *Transactions on Visualization and Computer Graphics*, pages 120–135, 2001.

[BSFG09] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. PatchMatch: A randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics*, 28(3), 2009.

[BvdPLD10] Nicolas Bonneel, Michiel van de Panne, Sylvain Lefebvre, and George Drettakis. Proxy-guided texture synthesis for rendering natural scenes. In *Proceedings of Vision Modeling and Visualization*, 2010.

[BZ07] Fabiano C. Botelho and Nivio Ziviani. External perfect hashing for very large key sets. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, CIKM '07, pages 653–662. ACM, 2007.

[Car84] Loren Carpenter. The a-buffer, an antialiased hidden surface method. In *Proceedings of SIGGRAPH*, volume 18, pages 103–108, 1984.

[CBAF08] Taeg Sang Cho, Moshe Butman, Shai Avidan, and William T. Freeman. The patch transform and its applications to image editing. In *Proceedings of the Conference on Computer Vision and Pattern Recognition*, 2008.

[Cel86] Pedro Celis. *Robin Hood Hashing*. PhD thesis, University of Waterloo, Ontario, Canada, 1986.

[CESL10] Mattäus G. Chajdas, Christian Eisenacher, Marc Stamminger, and Sylvain Lefebvre. Gpu pro - advanced rendering techniques. ShaderX Book Series, chapter Virtual Texture Mapping 101. A.K. Peters, 2010. ISBN 1-56881-472-0.

[CH02] Nathan A. Carr and John C. Hart. Meshed atlases for real-time procedural solid texturing. *ACM Transactions on Graphics*, 21(2):106–131, 2002.

[CLDD09] Marcio Cabral, Sylvain Lefebvre, Carsten Dachsbacher, and George Drettakis. Structure preserving reshape for textured architectural scenes. *Computer Graphics Forum*, 28(2), 2009.

[CLS10] Matthäus Chajdas, Sylvain Lefebvre, and Marc Stamminger. Assisted texture assignment. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*, 2010.

[CNLE09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*, 2009.

[DB97] Jeremy S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. In Turner Whitted, editor, *Proceedings of SIGGRAPH*, pages 361–368. ACM SIGGRAPH, Addison Wesley, August 1997.

[DG97] J.-M. Dischler and D. Ghazanfarpour. A procedural description of geometric textures by spectral and spatial analysis of profiles. *Computer Graphics Forum*, 16(3):129–139, 1997.

[DGF98] Jean-Michel Dischler, Djamchid Ghazanfarpour, and R. Freydier. Anisotropic solid texture synthesis using orthogonal 2d views. *Computer Graphics Forum*, 17(3):87–96, 1998.

[DGPR02] David DeBry, Jonathan Gibbs, Devorah DeLeon Petty, and Nate Robins. Painting and rendering textures on unparameterized models. In *Proceedings of SIGGRAPH*, pages 763–768. ACM SIGGRAPH, ACM Press, 2002.

[DL08] Carsten Dachsbacher and Sylvain Lefebvre. *Efficient and Practical TileTrees (in Shader X6)*. Shader X6. Charles River Media, 2008. ISBN 1-58450-544-3.

[DLTD08] Yue Dong, Sylvain Lefebvre, Xin Tong, and George Drettakis. Lazy solid texture synthesis. *Computer Graphics Forum*, 27(4), 2008.

[DMLG02] J.M. Dischler, K. Maritaud, B. Lévy, and D. Ghazanfarpour. Texture particles. *Proceedings of EUROGRAPHICS*, 21(3):401–410, 2002.

[DMV04] Luc Devroye, Pat Morin, and Alfredo Viola. On worst-case robin hood hashing. *SIAM Journal on Computing*, 33:923–936, 2004.

[DSC03] Laurent Demanet, Bing Song, and Tony Chan. Image inpainting by correspondence maps: a deterministic approach. Technical report, 2003. UCLA CAM Report 03-40.

[EF01] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of SIGGRAPH*, pages 341–346, 2001.

[ELS08] Christian Eisenacher, Sylvain Lefebvre, and Marc Stamminger. Texture synthesis from photographs. *Computer Graphics Forum*, 27(2), 2008.

[EMP+94] David Ebert, Kent Musgrave, Darwyn Peachey, Ken Perlin, and Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, October 1994. ISBN 0-12-228760-6.

[ETB+10] Christian Eisenacher, Chuck Tappan, Brent Burley, Daniel Teece, and Arthur Shek. Example-based texture synthesis on disneys tangled. In *SIGGRAPH 2010, Production Talks*, 2010.

[FH05] M. S. Floater and K. Hormann. *Surface Parameterization: a Tutorial and Survey*. 2005.

[FHCD92] Edward A. Fox, Lenwood S. Heath, Qi Fan Chen, and Amjad M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, January 1992.

[FM86] F. Fabbrini and Claudio Montani. Autumnal quadtrees. *The Computer Journal*, 29(5):472–474, 1986.

[FSDH07] Matthew Fisher, Peter Schröder, Mathieu Desbrun, and Hugues Hoppe. Design of tangent vector fields. *ACM Transactions on Graphics*, 26(3):56, 2007.

[GD95a] Djamchid Ghazanfarpour and Jean-Michel Dischler. Spectral analysis for automatic 3-d texture generation. *Computer and Graphics*, 19(3):413–422, 1995.

[GD95b] Djamchid Ghazanfarpour and Jean-Michel Dischler. Spectral analysis for automatic 3d texture generation. *Computers & Graphics*, 19(3), 1995.

[GD96] Djamchid Ghazanfarpour and Jean-Michel Dischler. Generation of 3d texture using multiple 2d models analysis. *Computer Graphics Forum*, 15(3):311–323, 1996.

[GD10] G. Gilet and J-M. Dischler. An image-based approach for stochastic volumetric and procedural details. *Computer Graphics Forum*, 29(4):1411–1419, 2010.

[GDS10] Guillaume Gilet, Jean-Michel Dischler, and Luc Soler. Procedural descriptions of anisotropic noisy textures by example. In *EUROGRAPHICS short paper*, pages 77–80, 2010.

[GGM11] B. Galerne, Y. Gousseau, and J. Morel. Random phase textures: Theory and synthesis. *IEEE Transactions on Image Processing*, 20(1):257–267, 2011.

[GLHL11] Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. Coherent parallel hashing. *ACM Transactions on Graphics*, 2011. to appear.

[GLLD12] Bruno Galerne, Ares Lagae, Sylvain Lefebvre, and George Drettakis. Gabor noise by example. *ACM Transactions on Graphics*, 31(4):73:1–73:9, July 2012.

[GSX00] Baining Guoa, Harry Shum, and Ying-Qing Xu. Chaos mosaic: Fast and memory efficient texture synthesis, 2000. Microsoft Research technical report MSR-TR-2000-32.

[GZD08] A. Goldberg, M. Zwicker, and F. Durand. Anisotropic noise. *ACM Transactions on Graphics*, 27(3):54:1–54:8, 2008.

[Har94] John C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12:527–545, 1994.

[HB95] David J. Heeger and James R. Bergen. Pyramid-Based texture analysis/synthesis. In Robert Cook, editor, *Proceedings of SIGGRAPH*, pages 229–238. ACM SIGGRAPH, Addison Wesley, August 1995.

[Hec86] Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics & Applications*, 6:56–67, November 1986.

[Hec89] Paul S. Heckbert. Fundamentals of texture mapping and image warping. Master's thesis, 1989.

[HJO+01] Aaron Hertzmann, Charles E. Jacobs, Nuria Oliver, Brian Curless, and David H. Salesin. Image analogies. In *Proceedings of SIGGRAPH*, pages 327–340. ACM Press, 2001.

[HRRG08] Charles Han, Eric Risser, Ravi Ramamoorthi, and Eitan Grinspun. Multiscale texture synthesis. *ACM Transactions on Graphics*, 27(3):51:1–51:8, 2008.

[HTW07] Hao-Da Huang, Xin Tong, and Wen-Cheng Wang. Accelerated parallel texture optimization. *Journal of Computer Science and Technololgy*, 22(5):761–769, 2007.

[HW91] Andrew Hunter and Philip Willis. Classification of quad-encoding techniques. *Computer Graphics Forum*, 10(2):97–112, 1991.

[JGSF73] B. Julesz, E. N. Gilbert, L. A. Shepp, and H. L. Frish. Inability of humans to discriminate between visual textures that agree in second-order statistics-revisited. *Perception*, (2):391–405, 1973.

[Jul62] Bela Julesz. Visual pattern discrimination. *IEEE Transactions on Information Theory*, 8(2):84–92, 1962.

[KEBK05] Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. Texture optimization for example-based synthesis. *ACM Transactions on Graphics*, 24(3):795–802, 2005.

[KFCO+07] Johannes Kopf, Chi-Wing Fu, Daniel Cohen-Or, Oliver Deussen, Dani Lischinski, and Tien-Tsin Wong. Solid texture synthesis from 2d exemplars. In *Proceedings of SIGGRAPH*, 2007.

[Kom06] Nikos Komodakis. Image completion using global optimization. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 1*, CVPR '06, pages 442–452, Washington, DC, USA, 2006. IEEE Computer Society.

[KP10] William B. Kerr and Fabio Pellacini. Toward evaluating material design interface paradigms for novice users. *ACM Transactions on Graphics*, 2010.

[KSE+03] Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut textures: Image and video synthesis using graph cuts. *ACM Transactions on Graphics*, 22(3), 2003.

[LD07] Sylvain Lefebvre and Carsten Dachsbacher. Tiletrees. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*, 2007.

[LD11] Ares Lagae and George Drettakis. Filtering solid Gabor noise. *ACM Transactions on Graphics*, 30(4):51:1–51:6, July 2011.

[LDN04] Sylvain Lefebvre, Jérome Darbon, and Fabrice Neyret. Unified texture management on arbitrary meshes. Technical Report 5210, INRIA, May 2004.

[LDR09] Jianye Lu, Julie Dorsey, and Holly Rushmeier. Dominant texture and diffusion distance manifolds. *Computer Graphics Forum*, 28(2):667–676, 2009.

[Lef03] Sylvain Lefebvre. *Shaderx2: Shader Programming Tips & Tricks*, chapter Drops of water texture sprites. Wordware Publishing, 2003. ISBN 1-55622-988-7.

[Lef08] Sylvain Lefebvre. *Filtered Tilemaps (in Shader X6)*. Shader X6. Charles River Media, 2008. ISBN 1-58450-544-3.

[Lew89] John-Peter Lewis. Algorithms for Solid Noise Synthesis. In Jeffrey Lane, editor, *Proceedings of SIGGRAPH*, volume 23, pages 263–270, July 1989.

[LH04] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Transactions on Graphics*, 23(3):769–776, 2004.

[LH05] Sylvain Lefebvre and Hugues Hoppe. Parallel controllable texture synthesis. *ACM Transactions on Graphics*, 24(3), 2005.

[LH06a] Sylvain Lefebvre and Hugues Hoppe. Appearance-space texture synthesis. *ACM Transactions on Graphics*, 25(3), 2006.

[LH06b] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. *ACM Transactions on Graphics*, 25(3), 2006.

[LH07] Sylvain Lefebvre and Hugues Hoppe. Compressed random-access trees for spatially coherent data. In *Rendering Techniques (Proceedings of the Eurographics Symposium on Rendering)*. Eurographics, 2007.

[LHL10] Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. By-example synthesis of architectural textures. *ACM Transactions on Graphics*, 29(4), 2010.

[LHL13] Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. HA-Buffer: Coherent Hashing for single-pass A-buffer. Rapport de recherche RR-8282, INRIA, 2013.

[LHL14] Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. Per–pixel lists for single pass A-buffer. *GPU Pro 5*, 2014. (to appear).

[LHN05a] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. *Octree Textures on the GPU (in GPU Gems II)*. Addison–Wesley, 2005. ISBN 0-32133-559-7.

[LHN05b] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. Texture sprites: Texture elements splatted on surfaces. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*, 2005.

[LKS+06] A. Lefohn, J. Kniss, R. Strzodka, S. Sengupta, and J. Owens. Glift : Generic, efficient random-access gpu data structures. *ACM Transactions on Graphics*, 25(1):60–99, 2006.

[LL12] A. Lasram and S. Lefebvre. Parallel patch-based texture synthesis. In *High Performance Graphics conference proceedings*, 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*, 29(8), 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. *ACM Transactions on Graphics*, 28(3), 2009.

[LLH04] Yanxi Liu, Wen-Chieh Lin, and James Hays. Near-regular texture analysis and manipulation. *ACM Transactions on Graphics*, 23(3):368–376, 2004. Proceedings of SIGGRAPH.

[LN03] Sylvain Lefebvre and Fabrice Neyret. Pattern based procedural textures. In *Proceedings of the ACM Symposium on Interactive 3D Graphics*, 2003.

[LPRM02] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Transactions on Graphics*, 21(3):362–371, July 2002. Proceedings of SIGGRAPH.

[LS96] Nathan Linial and Ori Sasson. Non-expansive hashing. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, STOC '96, pages 509–518. ACM, 1996.

[LSK+05] Aaron Lefohn, Shubhabrata Sengupta, Joe M. Kniss, Robert Strzodka, and John D. Owens. Dynamic adaptive shadow maps on graphics hardware. In *ACM SIGGRAPH 2005 Conference Abstracts and Applications*, August 2005.

[LVLD10] Ares Lagae, Peter Vangorp, Toon Lenaerts, and Philip Dutré. Procedural isotropic stochastic textures by example. *Computer and Graphics*, 34(4):312–321, 2010.

[MA97] D. Mount and S. Arya, 1997. ANN: A library for approximate nearest neighbor searching. CGC 2nd Annual Fall Workshop on Computational Geometry, `//www.cs.umd.edu/~mount/ANN`, 1997.

[Meh82] Kurt Mehlhorn. On the program size of perfect and universal hash functions. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 170–175, Washington, DC, USA, 1982. IEEE Computer Society.

[MK97] Geoffrey J. McLachlan and Thriyambakam Krishnan. *The EM Algorithm and Extensions (Wiley Series in Probability and Statistics)*. Wiley-Interscience, 1997.

[MVLS14] Chongyang Ma, Nicholas Vining, Sylvain Lefebvre, and Alla Sheffer. Game level layout from design specification. In *Computer Graphics Forum*, 2014. (to appear).

[MWLT13] Chongyang Ma, Li-Yi Wei, Sylvain Lefebvre, and Xin Tong. Dynamic element textures. *ACM Transactions on Graphics*, 32(4):to appear, 2013.

[MYV93] Jérôme Maillot, Hussein Yahia, and Anne Verroust. Interactive texture mapping. In *Proceedings of SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 27–34, August 1993.

[MZWG07] Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. Image-based procedural modeling of facades. *ACM Transactions on Graphics*, 26(3), 2007.

[PELS10] Pau Panareda Busto, Christian Eisenacher, Sylvain Lefebvre, and Marc Stamminger. Instant Texture Synthesis by Numbers. *Vision, Modeling & Visualization 2010*, pages 81–85, 2010.

[Per85] Ken Perlin. An image synthesizer. In B. A. Barsky, editor, *Proceedings of SIGGRAPH*, volume 19(3), pages 287–296, July 1985.

[Pet57] W. W. Peterson. Addressing for random-access storage. *IBM J. Res. Dev.*, 1:130–146, April 1957.

[PFH00] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In *Proceedings of SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 465–470, July 2000.

[PGB03] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. In *Proceedings of SIGGRAPH*, SIGGRAPH 2003 Proceedings, pages 313–318, 2003.

[PH89] Ken Perlin and Eric M. Hoffert. Hypertexture. In Jeffrey Lane, editor, *Proceedings of SIGGRAPH*, volume 23(3), pages 253–262, July 1989.

[PKVP09] Y. Pritch, E. Kav-Venaki, and S. Peleg. Shift-map image editing. In *Proceedings of the International Conference on Computer Vision*, Kyoto, Sep-Oct 2009.

[PL95] Rubert Paget and I. D. Longstaff. Texture synthesis via a nonparametric markov random field. In *Proceedings of DICTA-95, Digital Image Computing: Techniques and Applications*, volume 1, pages 547–552, December 1995.

[PLH88] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, and James Hanan. Developmental models of herbaceous plants for computer imagery purposes. In *Proceedings of SIGGRAPH*, 1988.

[PP93] Kris Popat and Rosalind W. Picard. Novel cluster-based probability model for texture synthesis, classification, and compression. In *In SPIE, Visual Communications and Image Processing*, pages 756–768, 1993.

[PP02] Athanasios Papoulis and Unnikrishna Pillai. *Probability, Random Variables and Stochastic Processes*. McGraw-Hill, 4rd edition, 2002.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[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*, Fort Collins, CO, 1999.

[QY05] Xuejie Qin and Yee-Hong Yang. Basic gray level aura matrices: theory and its application to texture synthesis. In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, volume 1, pages 128–135, 2005.

[QY07] Xuejie Qin and Yee-Hong Yang. Aura 3d textures. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):379–389, 2007.

[RB07] Ganesh Ramanarayanan and Kavita Bala. Constrained texture synthesis via energy minimization. *IEEE Transactions on Visualization and Computer Graphics*, 13:167–178, 2007.

[RCOL09] Amir Rosenberger, Daniel Cohen-Or, and Dani Lischinski. Layered shape synthesis: automatic generation of control maps for non-stationary textures. *ACM Transactions on Graphics*, 28(5):107:1–107:9, 2009.

[RHDG10] Eric Risser, Charles Han, Rozenn Dahyot, and Eitan Grinspun. Synthesizing structured image hybrids. *ACM Transactions on Graphics*, 29(4):85:1–85:6, 2010.

[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 (Proceedings of Shape Modeling International)*, 2012.

[RS00] S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290:2323–2326, December 2000.

[S3T] SGI OpenGL Extension Registry: ARB_texture_compression & EXT_texture_compression_s3tc `http://oss.sgi.com/projects/ogl-sample/registry/`.

[Sag85] Thomas J. Sager. A polynomial time generator for minimal perfect hash functions. *Communications of the ACM*, 28(5):523–532, May 1985.

[SAM04] Jacob Strom and Tomas Akenine-Moller. Packman: Texture compression for mobile phones, 2004. ACM SIGGRAPH Technical Sketch.

[SYJS05] Jian Sun, Lu Yuan, Jiaya Jia, and Heung-Yeung Shum. Image completion with structure propagation. In *Proceedings of SIGGRAPH*, pages 861–868, 2005.

[SZS+08] Richard Szeliski, Ramin Zabih, Daniel Scharstein, Olga Veksler, Vladimir Kolmogorov, Aseem Agarwala, Marshall Tappen, and Carsten Rother. A comparative study of energy minimization methods for markov random fields with smoothness-based priors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30:1068–1080, 2008.

[TdSL00] J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290:2319–2323, 2000.

[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 '02: Proc. 29th annual conference on Computer graphics and interactive techniques*, pages 665–672, New York, NY, USA, 2002. ACM Press.

[vW91] Jarke J. van Wijk. Spot noise-Texture Synthesis for Data Visualization. In *Proceedings of SIGGRAPH*, volume 25 (4), pages 309–318, Las Vegas, Nevada, July 1991. ACM SIGGRAPH. ISBN 0-201-56291-X.

[vW09] J.M.P. van Waveren. id tech 5 challenges. In *SIGGRAPH Courses*, 2009.

[Wei02] Li-Yi Wei. *Texture synthesis by fixed neighborhood searching*. PhD thesis, 2002. Adviser-Marc Levoy.

[Wei04] Li-Yi Wei. Tile-based texture mapping on graphics hardware. In *Proceedings of Graphics Hardware*. Eurographics Association, August 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.

[WHZ+08] Li-Yi Wei, Jianwei Han, Kun Zhou, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Inverse texture synthesis. *ACM Transactions on Graphics*, 27(3):52:1–52:9, August 2008.

[Wil83] Lance Williams. Pyramidal parametrics. *Proceedings of SIGGRAPH*, 17(3):1–11, 1983.

[WL00] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In Kurt Akeley, editor, *Proceedings of SIGGRAPH*, pages 479–488. ACM SIGGRAPH, ACM Press, 2000.

[WLKT09] Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. State of the art in example-based texture synthesis. In *Eurographics 2009, State of the Art Report*, 2009.

[Wor96] Steven P. Worley. A cellular texturing basis function. In *Proceedings of SIGGRAPH*, pages 291–294, 1996.

[WSI04] Yonatan Wexler, Eli Shechtman, and Michal Irani. Space-time video completion. In *CVPR*, pages 120–127, 2004.

[WTL+06] Jiaping Wang, Xin Tong, Stephen Lin, Minghao Pan, Chao Wang, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Appearance manifolds for modeling time-variant appearance of materials. *ACM Transactions on Graphics*, 25(3):754–761, July 2006.

[WY04] Qing Wu and Yizhou Yu. Feature matching and deformation for texture synthesis. *ACM Transactions on Graphics*, 23:364–367, 2004.

[XDR11] Su Xue, Julie Dorsey, and Holly Rushmeier. Stone weathering in a photograph. *Computer Graphics Forum*, 30(4):1189–1196, 2011.

[YHBZ01] Lexing Ying, Aaron Hertzmann, Henning Biermann, and Denis Zorin. Texture and shape synthesis on surfaces. In *Proc. 12th Eurographics Workshop on Rendering Techniques*, pages 301–312, London, UK, 2001. Springer-Verlag.

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. In *ACM Transactions on Graphics*, pages 126:1–126:11, 2008.

[ZLL13] Shize Zhou, Anass Lasram, and Sylvain Lefebvre. By-example synthesis of curvilinear structured patterns. *Computer Graphics Forum*, 2013.

[ZS01] Denis Zorin and Peter Schröder. A unified framework for primal/dual quadrilateral subdivision schemes. *Computer Aided Geometric Design*, 18(5):429–454, June 2001.