IMAGE-BASED PROCEDURAL TEXTURE MATCHING AND TRANSFORMATION

Eric Bourque

Doctor of Philosophy

Computer Science

McGill University

Montréal, Québec

August 31^{st} 2005

A Thesis submitted to McGill University in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Your file Votre référence
ISBN: 978-0-494-25106-5
Our file Notre référence
ISBN: 978-0-494-25106-5

for Lucia

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Gregory Dudek, for his leadership and direction of this research. Many of the ideas presented in this thesis evolved from discussions while walking home together, and were later worked out in the tiny free spaces on his used-to-be-white board.

I would also like to express my gratitude to the members of the mobile robotics laboratory, who all, in one way or another, have contributed to this research.

Thanks also to Pierre Poulin for inviting me to give a talk on this work during the "Graphics Lunches" seminar series at l'Université de Montréal, which has led to a post-doctoral position in his LIGUM lab.

Nicholas Roy contributed several LaTeX and Emacs gems which simplified the preparation of this thesis.

This work was funded through doctoral scholarships from the Natural Sciences and Engineering Research Council of Canada (NSERC) and "les Fonds Québécois de la Recherche sur la Nature et les Technologies" (FQRNT) , as well as through grants to Gregory Dudek from NSERC and the Institute for Robotics and Intelligent Systems (IRIS) center of excellence.

Finally, I would like to thank the members of my committee, Gregory Dudek, Frank Ferrie, Michael Langer, Michael McCool, Joelle Pineau, and Kaleem Siddiqi for their feedback on the research presented in this thesis.

ABSTRACT

In this thesis, we present an approach to finding a procedural representation of a texture to replicate a given texture image which we call *image-based procedural texture matching*. Procedural representations are frequently used for many aspects of computer generated imagery, however, the ability to use procedural textures is limited by the difficulty inherent in finding a suitable procedural representation to match a desired texture. More importantly, the process of determining an appropriate set of parameters necessary to approximate the sample texture is a difficult task for a graphic artist.

The textural characteristics of many real world objects change over time, so we are therefore interested in how textured objects in a graphical animation could also be made to change automatically. We would like this automatic *texture transformation* to be based on different texture samples in a time-dependant manner. This notion, which is a natural extension of procedural texture matching, involves the creation of a smoothly varying sequence of texture images, while allowing the graphic artist to control various characteristics of the texture sequence.

Given a library of procedural textures, our approach uses a perceptually motivated texture similarity measure to identify which procedural textures in the library may produce a suitable match. Our work assumes that at least one procedural texture in the library is capable of approximating the desired texture. Because exhaustive search of all of the parameter combinations for each procedural texture is not computationally feasible, we perform a two-stage search on the candidate procedural textures. First, a global search is performed over pre-computed samples from the given procedural texture to locate promising

parameter settings. Secondly, these parameter settings are optimised using a local search method to refine the match to the desired texture.

The characteristics of a procedural texture generally do not vary uniformly for uniform parameter changes. That is, in some areas of the parameter domain of a procedural texture (the set of all valid parameter settings for the given procedural texture) small changes may produce large variations in the resulting texture, while in other areas the same changes may produce no variation at all. In this thesis, we present an adaptive random sampling algorithm which captures the texture range (the set of all images a procedural texture can produce) of a procedural texture by maintaining a sampling density which is consistent with the amount of change occurring in that region of the parameter domain.

Texture transformations may not always be contained to a single procedural texture, and we therefore describe an approach to finding transitional points from one procedural texture to another. We present an algorithm for finding a path through the texture space formed from combining the texture range of the relevant procedural textures and their transitional points.

Several examples of image-based texture matching, and texture transformations are shown. Finally, potential limitations of this work as well as future directions are discussed.

ABRÉGÉ

Cette thèse présente une approche permettant de trouver la représentation procédurale d'une texture dans le but de reproduire une texture imagée donnée. Nous référerons à cette approche sous le nom de *image-based procedural texturing*. Les représentations procédurales sont fréquemment utilisées dans plusieurs aspects de l'infographie. Toutefois, l'habileté à utiliser les textures procédurales est limitée par la difficulté inhérente à trouver une représentation procédurale acceptable s'apparentant à la texture désirée. De plus, l'identification des paramètres appropriés et nécessaires à l'approximation de l'échantillon de textures s'avère une tâche difficile pour le graphiste.

Puisque les caractéristiques texturales de plusieurs objets réels changent avec le temps, nous nous intéresserons à la façon dont les objets texturés peuvent être modifiés automatiquement à l'intérieur d'une animation graphique. Cette notion représente le prolongement naturel de l'appariement de textures procédurales et implique la création d'une séquence fluide d'images texturées tout en permettant au graphiste de contrôler les diverses caractéristiques de la séquence de textures.

A partir d'une librairie de textures procédurales, notre approche utilise une mesure de similitude perceptuelle de textures afin d'identifier quelles textures procédurales pourraient correspondre à la cible. La présente recherche assume que la texture-cible peut s'apparenter à au moins une texture procédurale de la librairie. Compte tenu que la recherche exhaustive de toutes les combinaisons de paramètres pour chaque texture procédurale n'est pas réalisable par calcul informatique, nous effectuerons une recherche en deux étapes sur les textures procédurales selectionnées. Dans un premier temps, une recherche globale sera réalisée auprès des échantillons déjà traités provenant d'une texture procédurale donnée afin d'identifier un jeu de paramètres potentiellement satisfaisants. Dans

un deuxième temps, ce jeu de paramètres sera optimisé par le biais d'une méthode de recherche locale afin de raffiner la texture correspondante à celle désirée.

Les caractéristiques d'une texture procédurale ne varient généralement pas de façon uniforme et ce, même lorsque les changements de paramètres sont uniformes. Par exemple, dans certaines régions du domaine de paramètres d'une texture procédurale (soit l'ensemble de toutes les données de paramètres valides pour une texture procédurale donnée), de petits changements peuvent entraîner d'importantes variations au niveau de la texture résultante tandis que ces mêmes changements peuvent ne pas produire de variations dans d'autres régions. Cette thèse présente un algorithme d'échantillonnage aléatoire adaptable qui extrait l'expression de l'étendue de la texture (soit l'ensemble de toutes les images que peut produire une texture procédurale) à partir d'une texture procédurale en conservant une densité d'échantillons qui est consistante avec la quantité de changements qui se produisent dans la région du domaine de paramètres.

Les transformations de textures ne se limitent pas toujours à une seule texture procédurale. Par conséquent, nous décrirons une approche qui trouve les points transitionnels d'une texture procédurale à l'autre. Nous présenterons un algorithme permettant de trouver un chemin à travers l'espace de texture formé en combinant l'étendue de texture des textures procédurales pertinentes et de leurs points transitionnels.

Plusieurs exemples d'appariement de textures imagées ainsi que de transformations de textures seront présentés. Finalement, les limitations potentielles et les directions futures de cette recherche seront discutées.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iv

ABSTRACT vi

ABRÉGÉ viii

LIST OF FIGURES xii

1 Introduction 1

1.1 Contribution 7
1.2 Outline 8

2 Background 9

2.1 Geometric Models 10
2.2 Surface Properties 12
2.2.1 Illumination and Shading Models 13
2.2.2 Ray Tracing 17
2.2.3 Texture Mapping 18
2.2.4 Procedural Techniques 23

3 Previous Work 31

3.1 Traditional Texture Synthesis 32
3.1.1 Hypertexture 32
3.1.2 Reaction-Diffusion Textures 33
3.2 Sample-Based Texture Synthesis Techniques 36
3.2.1 Steerable Pyramid Statistical Matching 36
3.2.2 Markov Texture Synthesis 39
3.2.3 Bi-directional Texture Function 43
3.3 Procedural Texture Matching 44

4 Procedural Texture Matching 47

4.1 Approach 48
4.2 Searching in Texture Space 49
4.3 Global Search 50
4.4 Local Search 57
4.4.1 Downhill Simplex Method 59
4.4.2 Gradient Ascent Method 62

4.5 Evaluating Texture Similarity 62
4.6 Examples 73

5 Procedural Texture Transformation 86

5.1 Approach 87
5.2 Transformation Within a Shader 88
5.3 Transformation Between Different Shaders 94
5.4 Examples 97

6 Conclusion 106

6.1 Future Work 108

A Software Architecture 112

A.1 System Design 112
A.2 Graphical User Interface 115

B Shading Language Code Example 1 120

C Shading Language Code Example 2 124

References 128

LIST OF FIGURES

Figure page

1–1 An example of a scene with complicated interactions. 2

1–2 An image rendered using procedural shaders for all surfaces. 4

1–3 An example of procedural texture matching. 5

1–4 A few frames from an example texture transformation. 6

2–1 Phong illumination model. 14

2–2 BRDF illumination model. 15

2–3 An example of a ray traced image. 16

2–4 A simplified ray tracing example. 17

2–5 Mapping from a pixel in screen space to a region of the texture map. 18

2–6 Two-part texture mapping. 19

2–7 A solid texture example. 21

2–8 A vertical bar shader. 24

2–9 Example images rendered using procedural shaders for all surfaces. . 25

2–10 Example of the diversity available from one shader. 26

3–1 Hypertexture 32

3–2 Reaction diffusion textures. 33

3–3 Reaction-diffusion textures *grown* directly on a surface. 35

3–4 Textures synthesised using De Bonet's technique. 38

3–5 Textures synthesised using Efros and Leung's technique. 39

3–6 Comparison of different texture synthesis techniques. 41

3–7 Textures synthesised using Lefebvre and Poulin's technique. 44

4–1 The construction step of the adaptive random sampling method. . . . 51

4–2 The refinement step of the adaptive random sampling method. 53

4–3 The pruning step of the adaptive random sampling method. 54
4–4 An example of uniformly sampling an individual shader. 55
4–5 An example of adaptively sampling an individual shader. 56
4–6 Local phase of the search strategy. 57
4–7 Different results after a step in the downhill simplex method. 60
4–8 Steepest ascent along a narrow crest. 62
4–9 An example of texture segregation. 65
4–10 An example of the texture similarity measure. 68
4–11 Power spectrum example. 70
4–12 An example image pyramid. 71
4–13 An example of a Laplacian image pyramid. 72
4–14 An example of a deterministic and a stochastic texture. 73
4–15 Texture matching based on deterministic synthetic target textures. . . 78
4–16 Texture matching based on stochastic synthetic target textures. . . . 79
4–17 Texture matching using deterministic Brodatz textures. 80
4–18 Texture matching using stochastic Brodatz textures. 81
4–19 Matching real textures from the sky during the day and at night. . . . 82
4–20 An example of texture matching from a sketch. 83
4–21 Examples of failed texture matches. 84
4–22 A degenerate texture for the star shader. 85
5–1 Shortest path calculated using the graph from the shader catalogue. . 89
5–2 An illustration of our path cost function. 91
5–3 An example of the adaptive linear subdivision technique. 93
5–4 Finding jump points between two shaders. 95
5–5 An example of path through a connective shader. 96
5–6 An example of a texture transformation within a single shader. . . . 99
5–7 Another single shader texture transformation. 100

5–8 Another single shader texture transformation. 101

5–9 A texture transformation between two different shaders. 102

5–10 Another texture transformation between two different shaders. . . . 103

5–11 Another texture transformation between two different shaders. . . . 104

5–12 A transformation which uses a connective shader. 105

6–1 An example of motion between texture frames. 108

A–1 Texture class hierarchy. 113

A–2 Metric class hierarchy. 114

A–3 Screen shots of the texture matching application. 117

A–4 More screen shots of the texture matching application. 118

CHAPTER 1

Introduction

WE are living in a time where computer generated imagery (CGI) is ubiquitous. We find CGI in all forms of visual media – print, motion picture and television; and these images are becoming increasingly convincing. In fact, these images are so realistic that we are often unaware that they are generated synthetically. This realism has come as a result of numerous advances in the field of computer graphics, many of which will be outlined below. It is one of these techniques, namely procedural texturing, which will be the subject of this thesis.

The creation or *rendering* of computer generated imagery follows a basic formula, with certain necessary elements: a scene, one or more light sources, and a camera. These elements are represented in a mathematical model, with surfaces in the virtual scene being composed of geometric primitives such as polygons and spheres, a camera model which determines the type of projection (normally perspective), and a model for how light interacts with the surfaces in the scene. These elements can be specified using either implicit or explicit representations (terms only loosely related to the mathematical notions of explicit and implicit

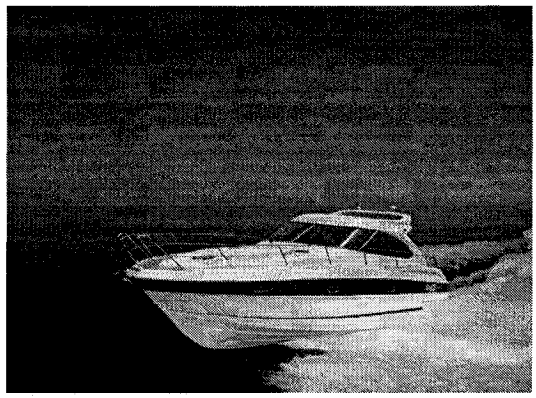


Figure 1–1: An example of a real world scene with complicated interactions between rigid and non-rigid (deformable) elements.

functional forms). Implicit representations are commonly referred to as procedural representations or procedural techniques.

A procedural technique is essentially an approach based on an algorithmic, or implicit description of some element of a scene (i.e., shading, lighting, or geometry). For example, if one wanted to render a scene containing a mountain range in the background, the individual faces of the mountains could be specified explicitly as a mesh of thousands of connected polygons, or could instead be represented implicitly in the form of a simple algorithm. This algorithm could produce a similar landscape when given suitable input parameters to control aspects such as the maximum height of a mountain, minimum height, starting height for snow caps, age of the mountain range (sharpness of the peaks), etc.

Procedural techniques began to be used in the field of computer graphics when scenes became too complex to be specified explicitly. Procedural modelling was an obvious choice for animators who wanted to create scenes with both stationary and moving objects interacting in accordance with physical laws. This type of modelling is even more appropriate when it is necessary to model the complex interactions of non-rigid, deformable elements present in a scene. Consider trying to model a scene consisting of a motor boat speeding across a

body of water (see Fig. 1–1). Increased computing power has allowed animators to render such scenes without worrying about explicitly modelling all the details of how the waves propagate, how splashes form, and how the boat bounces along on the water [36].

The simulated interaction of light and objects in a scene is described by the illumination model. Modern renderers allow this model to be specified on a per-object basis, in the form of what are called *procedural shaders*. These are parameterised functions which calculate the colour and intensity of a given point queried on a surface. Their parameters control various aspects of the object's appearance, and vary per shader. Because the functions are queried for each individual location on the surface being shaded[1] , it is possible to use them to create a texture on the surface. This is referred to as *procedural texturing*.

An alternative method used to apply a texture to an object is called *texture mapping*. The main idea behind texture mapping is to *paste* a two dimensional picture onto a surface and have it stretch and bend accordingly (see Sec. 2.2.3). In this way, texture maps can be thought of as decals. This kind of approach can be used to create appearance-based models as exemplified by our previous work on environment modelling for off-line virtual navigation. That method used a mobile robot to automatically select salient locations in an environment and to capture all relevant scene data explicitly in the form of digital images [17, 13, 15, 14]. These images were then post-processed into image-based virtual reality (VR) scenes. The method worked well for environments which were not too large, but larger

[1] The sampling of surface locations is generally determined by the projection of a pixel from the image plane onto the corresponding patch on the object's surface. This is discussed further in Sec. 2.2.3.

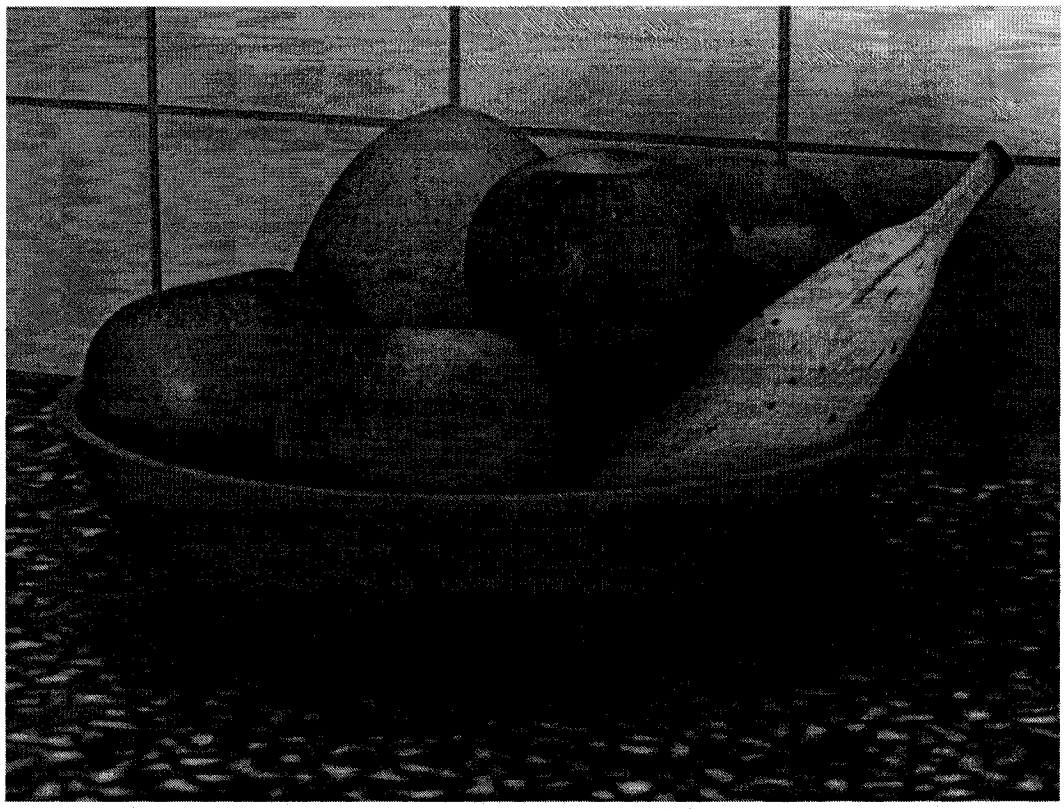


Figure 1–2: An example of an image rendered using procedural shaders for all surfaces (i.e., the wall tiles, fruit, bowl, and the counter top). Image by Jonathan Merritt. Used with permission.

environments placed higher demands on the storage of image data, and also added to the complexity of using image based rendering for the VR scenes.

Procedural texturing, on the other hand, has minimal storage requirements since it allows an algorithm to describe how a textured surface should appear as a function of its local environment. There are many advantages to using procedural textures over texture mapping, the details of which are further elaborated in Sec. 2.2.4. For the remainder of this thesis, we will use the terms *procedural texture*, and *shader* interchangeably. Figure 1–2 shows an example of an image rendered using procedural textures for all surfaces.

In this thesis, we will describe a technique we refer to as *image-based procedural texture matching*; a technique which combines the simplicity of traditional texture mapping with the flexibility and strengths of procedural

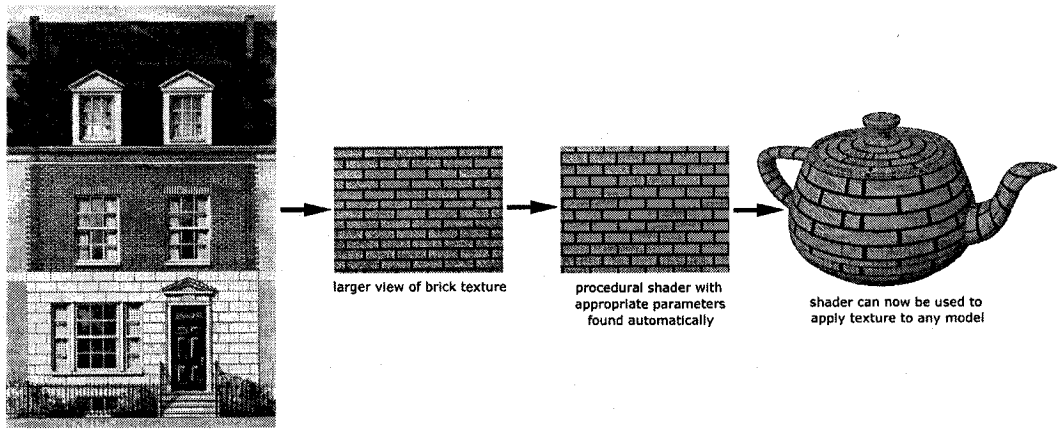


Figure 1–3: An example of procedural texture matching. An input texture is acquired from an architect's drawing, and a suitable procedural shader and parameters are found to replicate the appearance of the texture so that the shader can be applied to an arbitrary model.

texturing. When texturing using this method, a graphic artist specifies a digital input image, and a procedural texture which generates a similar[2] texture is found. This gives the graphic artist much more freedom to be creative, minimising the arduous time spent tweaking various aspects of the procedural framework. An example of the procedural texture matching process is shown in Fig. 1–3.

To exemplify why this problem needs to be solved, consider that a large library of shaders can be available to a sophisticated user. A typical shader can have half a dozen parameters, and some can have substantially more. Further, the texture that is produced by a shader can vary substantially over the range of these parameters. Thus, to find a desired texture, a user must search over a complicated and high-dimensional space. Finding the right combination of parameters to define a point in this space is clearly problematic.

[2] The notion of similarity will be explored further in Sec. 4.5.

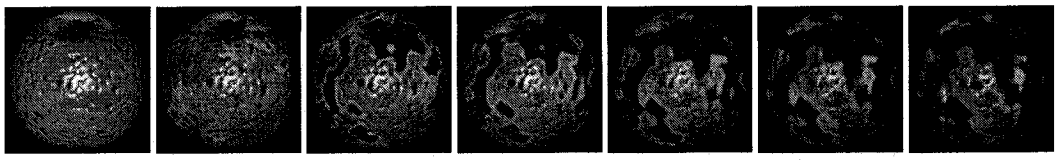


Figure 1–4: A few frames from an example texture transformation. This transformation exhibits the desired perceptual smoothness between adjacent frames.

Our work considers the problem of starting with an initial description of a target texture (in the form of a sample image) and finding a procedural representation to match the target texture. In this thesis, we do not discuss the creation of new shaders, but how to optimise the choice of a shader and its parameter settings to produce the desired appearance. This work assumes the availability of a library of shaders, and that the desired texture can be approximated by at least one shader in the library.

Given a solution to the procedural texture matching problem described above, we can then consider the problem of finding a *sequence* of procedural textures that will produce a gradually varying series of textures that accomplish a transition between two specified textures. We refer to this sequence as a *texture transformation*. In general, a texture transformation may involve using more than one shader and using varying parameter settings for each shader contained in the transformation. An important criterion for a desirable texture transformation is that it should take a *smooth* path from the initial texture to the final one. This smoothness is not measured with respect to the variations in the parameters, but with respect to the perceptual variations the texture must traverse. An example texture transformation is shown in Fig. 1–4.

Note that even with many available shaders, the space of possible target images is far larger than the set of textures that can be synthesised, so some texture images will be hard to approximate. Likewise, a good texture transformation will not always be possible, particularly if the repertoire of available shaders is limited.

Our texture matching approach is based on four key stages:

1. A global search strategy over a library of shaders to select the ones that might produce interesting results.
2. A global search over a single shader to obtain a rough estimate of suitable input parameter settings.
3. A local search strategy to optimise parameter settings given a rough guess from the previous stage.
4. A perceptually motivated texture comparison function that allows us to estimate the quality of our solution.

Likewise, our texture transformation method is based on the following elements:

1. A strategy for finding an appropriate path through a sparse collection of samples within an individual shader.
2. A technique for smoothing the transformation between adjacent samples from the path determined above.
3. A method for determining the points of maximum similarity between two different shaders to be used to transition from one shader to another.
4. A method for creating a smooth path *through* different shaders by combining the elements above.

1.1 Contribution

In the research presented in this thesis, we have established a new generic solution to image-based procedural texture matching. This solution allows a graphic artist to specify a texture sample, and to have a procedural representation of the desired texture found automatically. The primary advantage of this technique over other image-based techniques is that the final texture can be rendered at an arbitrary resolution which is necessary for photo-realistic rendering. In addition,

a procedural representation affords the artist the ability to make minor changes in the appearance of the procedural texture if necessary.

We also present an extension of this solution to the time domain in order to produce procedural texture animations. These are smoothly varying sequences of procedural textures based on a particular set of starting and ending texture samples. This work has been published in *Computer Graphics Forum* [16], one of the top international journals specialising in computer graphics.

Finally, we have experimentally evaluated our approach using a software framework. This framework, while developed as a vehicle for performing this research, is very generic, and as such can be re-used for other problem domains as discussed in chapter 6.

1.2 Outline

The outline of the remainder of this thesis is as follows. In chapter 2 we present the relevant background material necessary for an understanding of the work presented in this thesis, and chapter 3 outlines the relevant prior computer graphics research related to texturing. In chapter 4, we present our technique for approximating a given image sample procedurally. Chapter 5 describes our method for creating smooth texture transformations based on matches found using procedural texture matching or from manually specified procedural textures. Finally, in chapter 6 there is a discussion of the work presented in this thesis as well as future directions which can be explored.

CHAPTER 2

Background

BEFORE we can delve into the details of procedural shading and texture synthesis, we must first give a quick overview of how images are actually created (rendered) using a computer.

Definition 2.0.1 (Image Synthesis)

Image synthesis is the methodology of the creation of images using a computer. In three-dimensional computer graphics the image is generated by a computer program from a three dimensional mathematical description or a model by calculating a two dimensional projection for display [78].

Because we are only interested in photo-realistic images produced by 2D projections from 3D scenes, we will not consider issues specific to other domains such as cartoon generation, or technical illustrations. We will instead focus on traditional physically motivated rendering methods.

Methods for 3D image rendering fall into two main classes: those that are rasterisation based, and those which are referred to as ray-tracers. Each rendering

method has its advantages[1] , but for the purposes of our discussion, they can be considered to be equivalent since each has similar requirements.

There are three distinct components required for rendering a photo-realistic image:

1. A specification of the scene geometry.
2. A specification of the properties of all the surfaces contained within the scene geometry.
3. A specification of the lighting in the scene.

The camera position, or viewing parameters are elements which may not seem to be contained in the aforementioned components, however we can think of them as being contained within the scene geometry without loss of generality. Conceptually, we can think of the second two elements above as being linked since the lighting parameters can be specified in the surface properties of the lights, which are objects in the scene geometry. In chapter 5 we will discuss a fourth component necessary for animation, namely the variation of the scene geometry and surface details over time.

2.1 Geometric Models

A model of the scene can be represented using many different techniques, depending on the desired geometric accuracy, as well as the permissible overall size of the model (the classic time/space trade-off). The model is used to specify the surfaces of the objects in the scene, as well as to provide the ability to calculate the surface normal for arbitrary points on these surfaces. Several different representations of scene geometry are outlined below. Note that in most

[1] For example, scan-line based renderers can make extensive use of hardware acceleration leading to interactive (high frame rate) graphics while ray-tracers generally produce more photo-realistic images, but at non-interactive rates.

real cases, a mixture of these representations is used depending on their suitability to the particular objects being modelled.

- Polygonal : Objects are represented by a mesh of polygons, often triangles. Using this method, the desired accuracy is controlled by the number of polygons (the level of subdivision) present in the model. Polygonal representations are advantageous when interactive frame rates are desired since modern graphics hardware generally implements the entire rendering pipeline for polygons and scan-line rendering.
- Parametric Patches : This representation is similar to that above, with the exception that the elements of the meshes are curved surfaces. Common types of parametric patches are the Bézier patch (a specialised form of the Hermite patch), and NURBS (non-uniform rational B-spline) patches. Parametric patch representations are advantageous when attributes such as curvature, surface normal, etc., need to be computed for arbitrary points on the patch. One disadvantage with parametric patches is that they are typically difficult to specify and control during the modelling phase.
- Constructive Solid Geometry : In CSG, an object is composed of boolean set operations on geometric primitives (which can be CSG objects themselves). Objects which are difficult to describe using other modelling techniques often have a very compact representation using CSG. For example, the object formed by subtracting a unit sphere from a unit cube is represented very compactly in CSG, however, its representation using parametric patches is much larger.
- Spatial Subdivision Techniques : This representation is somewhat related to that above – here the 3D space is divided into cells which are either marked as empty or full. The spatial subdivision can be regular (into cubes called *voxels*), or binary space partitioning (BSP) techniques can be used to save

space [76]. The latter are generally only useful if the scene is static since a minor change in the space can result in the BSP representation changing drastically.

- Implicit Representations : Objects can also be expressed implicitly, for example, $x^2 + y^2 + z^2 = r^2$ defines a sphere of radius r. Implicit representations in computer graphics are generally only useful for ray-tracing, and for calculating bounding objects for mesh representations since their mathematical form allows for a relatively low computational cost solution to intersection tests.

Because we are primarily interested in the appearance of surfaces in synthesised images, we will not dwell any further on methods for specifying scene geometry. Rather, we will focus on the appearance of the surface itself, and assume an appropriate geometric specification.

2.2 Surface Properties

Initially, computer graphics researchers focused on the pragmatics of generating synthetic images containing objects which were geometrically similar to their real-life counterparts. This presented enormous challenges and lead to many new techniques for specifying scene geometry. The advance of computer processing speed was also a key contributor to the increased complexity which modern renderers were able to handle. Once the objects in a scene were being represented accurately, researchers began to investigate how the properties of the surfaces in the geometric models could be specified to permit increased realism.

The study of how light interacts locally with surfaces led to several illumination and shading models. Many of the original simple models are still used today, especially in cases where interactive frame rates are a prerequisite, since their basic shading calculations can often be performed in hardware. The frequent

use of these models is what is responsible for the general "plastic" appearance of many early computer generated images since these models did not allow for specifications other than one colour per surface. With the introduction of *texture mapping*, it became possible to take digital images (either rendered themselves, or digitised real-world images) and *paste* them onto the surfaces. This advance went a long way towards increasing the realism of rendered scenes. Unfortunately, there are many drawbacks involved with texture mapping as will be shown in Sec. 2.2.3. In order to alleviate some of these problems, as well as to allow more complex textures to be represented, procedural shading was introduced. We will explore these three areas below, namely illumination models, texture mapping, and procedural techniques.

2.2.1 Illumination and Shading Models

An illumination model is a characterisation of how light interacts locally with a surface. Some such models are based on physical laws, while others are perceptually motivated, and make only moderate attempts at physical reality while providing simplified calculations. A shading model determines which illumination model is used, and how it is applied across a surface. We will give examples of several illumination and shading models in this section. It is worth noting that many of the illumination and shading models are approximations of the underlying rules of optics and thermal radiation, usually to simplify calculations.

The most commonly used illumination model is the Phong illumination model due to Phong Bui-Tuong [21]. In this model, there are three factors which contribute to the total light at a point on a surface: (1) ambient light, (2) diffuse light and (3) specular light. If $\mathbf{N}$, $\mathbf{L}$, $\mathbf{R}$ and $\mathbf{V}$ are the (unit length) normal to the surface, incoming light direction, reflection direction, and viewing direction (see Fig. 2–1), then we have the following familiar equation for the total light at a point

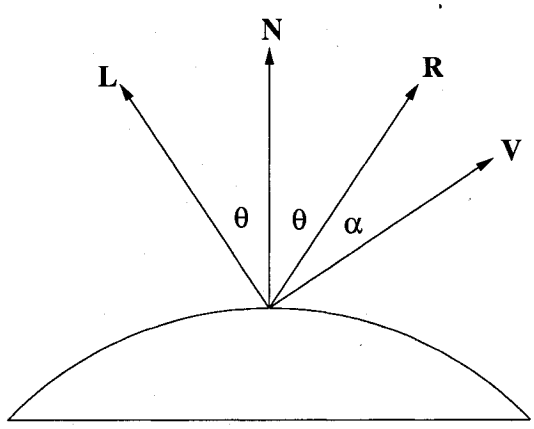


Figure 2–1: Phong illumination model: α is used to approximate the specular highlights, and θ is used to approximate diffuse light.

on the surface in the direction $\mathbf{V}$:

$$I = I_a k_a O_d + \sum_{1 \leq i \leq m} I_{p_i} \left[k_d O_d \left(\mathbf{N} \cdot \mathbf{L}_i \right) + k_s O_s (\mathbf{R}_i \cdot \mathbf{V})^n \right] \tag{2.1}$$

where I_a denotes the ambient light source, k_a, k_d and k_s the ambient, diffuse and specular coefficients, I_{p_i} the m point light sources, O_d and O_s the object's diffuse and specular components and n is the specular reflection exponent. Note that by convention all vectors point away from the surface.

One of the main drawbacks of the previous illumination model is that it cannot be used to model surfaces with an anisotropic reflectance function. These are surfaces which exhibit a directionally dependent reflectance. Velvet is an example of such a material: a swath of velvet changes appearance as it is rotated under a constant light source (it goes from shiny to matte depending on the orientation of the fibres). In general, light reflected from a point on the surface can be specified by a bi-directional reflectance distribution function or BRDF (see Fig. 2–2) [62, 71]:

$$\text{BRDF} = f(\theta_{in}, \phi_{in}, \theta_{ref}, \phi_{ref}) = f(\mathbf{L}, \mathbf{V}). \tag{2.2}$$

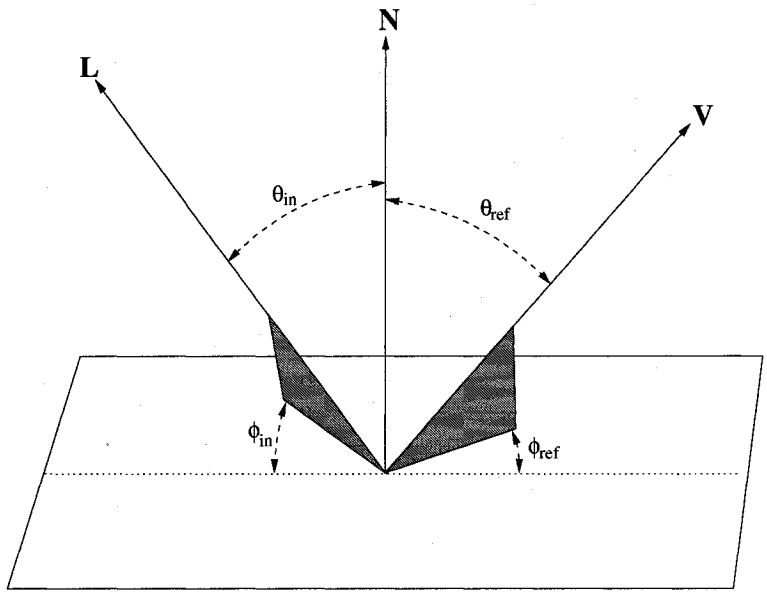


Figure 2–2: The BRDF illumination model relates light incident in direction **L** to light reflected along **V** as a function of the angles $\theta_{in}, \phi_{in}, \theta_{ref}, \phi_{ref}$.

The bi-directional reflectance function is defined for a single light ray of a single wavelength and is itself a function of four parameters. This makes the use of a complete BRDF impractical for most occasions so various approximations have been used. One popular technique is to sample the actual reflectance values in a controlled environment for a discrete set of lighting and viewing directions using a gonioreflectometer, and to then estimate the surface reflectance for lighting and viewing directions which were not sampled through interpolation [56, 26].

As mentioned above, a shading model determines how a local illumination model is applied to the underlying geometry of the scene. Three standard techniques are flat shading, Gouraud shading, and Phong shading. In flat shading, the colour of a surface is determined for only one point on the surface, and the resulting colour is applied to the entire surface. This model obviously lacks realism, but is extremely efficient. Gouraud shading calculates the colour at each vertex of the underlying geometry, and linearly interpolates the shading

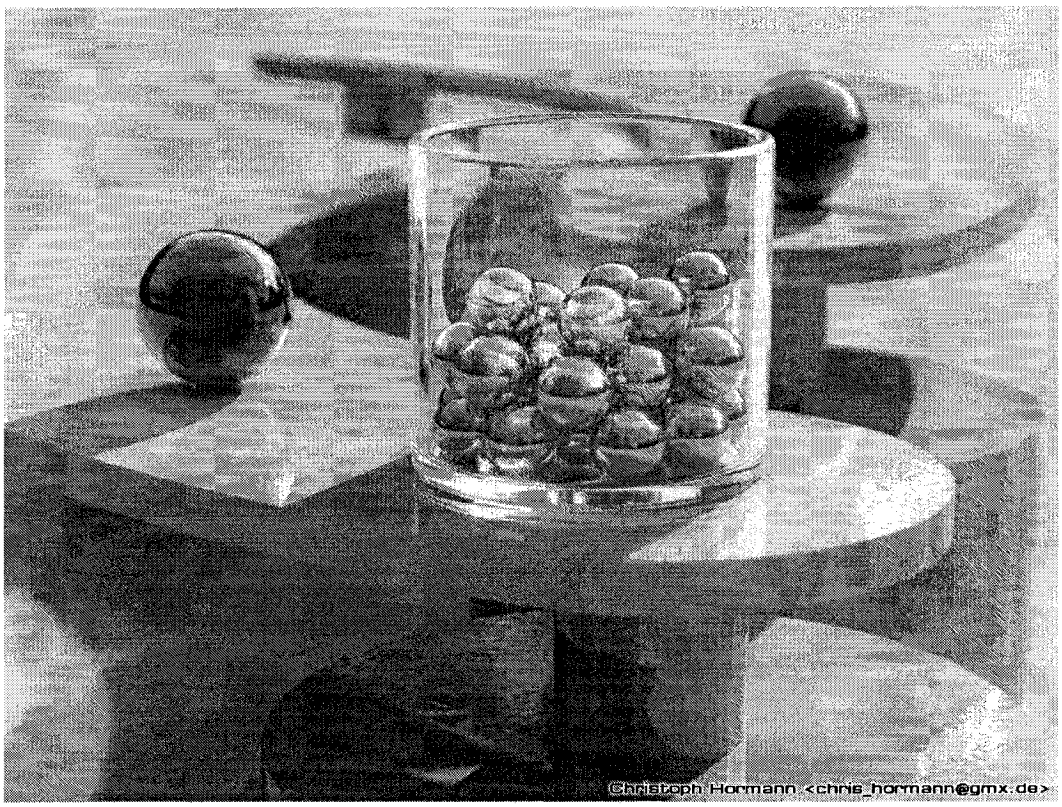


Figure 2–3: An example of a ray traced image. Notice all the reflections in the marbles, the transparency of the glass, and the refractive distortion of the blue marble in the back when viewed through the glass. ©2002 Christoph Hormann.

values across the surface. This shading method is an improvement over flat shading, however, artifacts such as specular highlights can be missed entirely.

Phong shading is similar to Gouraud shading except that it is the surface normals which are calculated at each vertex, and then interpolated across the surface. The illumination model is then evaluated for each point to be shaded. The main optimisation of this illumination model is that the surface normal does not need to be computed for each point on the surface and polygonal models can be rendered as if they were curved. In photo-realistic rendering, the local illumination model is usually applied at each shading point, using the actual surface normal for the point being shaded.

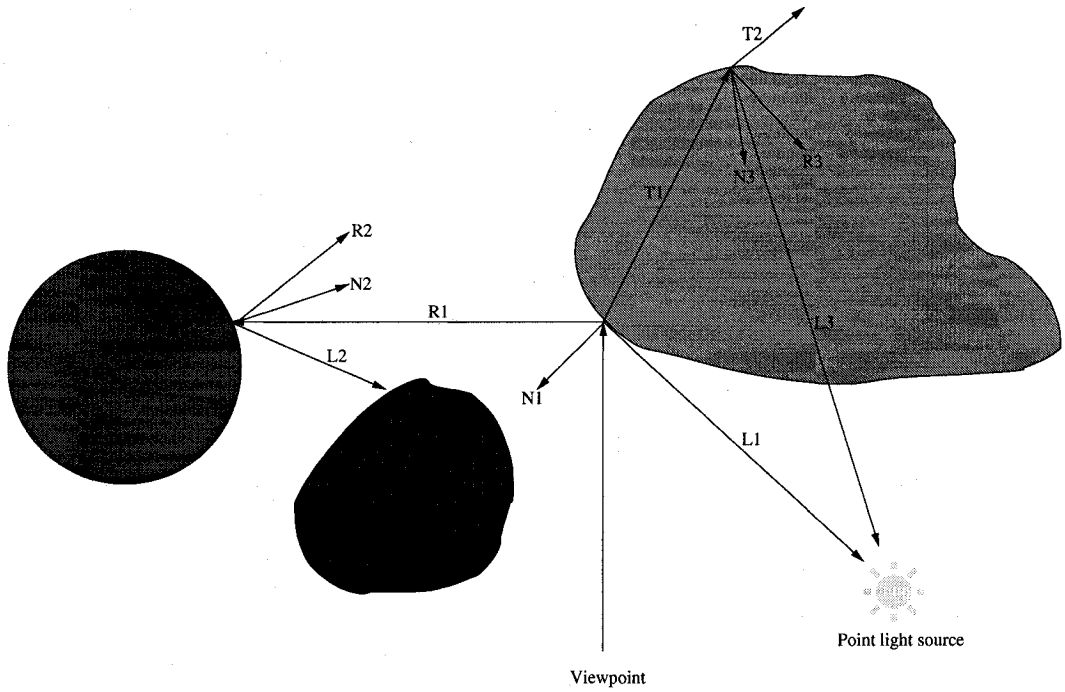


Figure 2–4: A simplified ray tracing example. The initial ray is cast from the viewpoint where it intersects with the first object, producing a reflection ray R_1, and a transmission ray T_1. The surface normals are indicated by N, and the shadow rays by L. The reflection ray R_1 then intersects a second object which itself spawns a reflection ray R_2, and a shadow ray L_2. The transmission ray T_1 intersects the other side of the first surface, where it spawns another reflection ray R_3 and another transmission ray T_2. The pixel value at the viewpoint will be determined by the bottom-up accumulation of the intensity values calculated at each intersection point in the ray tree.

2.2.2 Ray Tracing

In order to produce photo realistic images with reflections, and transmissive effects (see Fig. 2–3), it is necessary to compute the complicated paths that light will travel within a scene [81]. This process is referred to as *ray tracing*. The basic idea is to project a ray from each pixel position into the scene, and to reflect a specular ray based on the local geometry, as well as to cast shadow rays and transmissive rays dependent on the surface properties (Fig. 2–4).

The collection of these rays produces a *ray tree* which allows us to trace the light backwards from the pixel through the scene. For each pixel ray, we must test

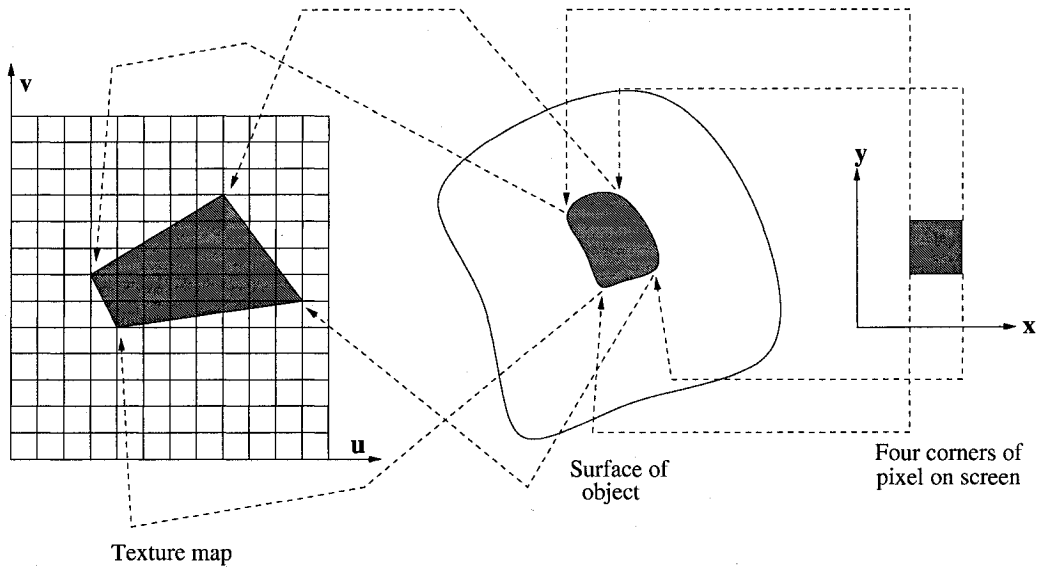
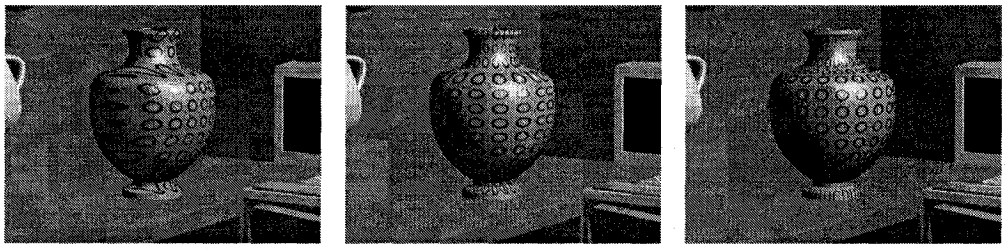


Figure 2–5: Mapping from a pixel in screen space to a region of the texture map.

the intersection of that ray with all the objects in the scene. These intersections are then sorted by depth to find the closest intersection. This will determine the first object the ray will intersect which will be the visible surface for the given pixel. This intersection ray is then reflected along a specular path, as well as a refractive path if the surface is transparent. The entire process is repeated for these *secondary rays* until either a preset tree depth has been reached, or until a set of storage constraints has been met. The intensity contributions from each intersection are then accumulated bottom-up through the tree to determine the pixel intensity for the initial ray.

2.2.3 Texture Mapping

In order to increase the realism of rendered images, the notion of texture mapping was introduced [23, 10]. The main idea behind texture mapping is to *paste* a two dimensional picture onto a surface and have it stretch and bend accordingly. In this way, texture maps can be thought of as decals. Texture mapping can be used to modulate surface properties other than colour; specular



(a) (b) (c)

Figure 2–6: An example of two-part texture mapping of an object using (a) a plane, (b) a cylinder, and (c) a sphere as intermediate objects. Notice how the texture on the neck of the vase changes depending on the geometry used for the intermediate object.

colour, normal vector perturbation, and transparency are all properties which could be modulated by a texture map. Below we will describe texture mapping in terms of modulating colour without loss of generality.

During texture mapping, a pixel in screen space is mapped to a corresponding curvilinear surface patch (consisting of four points from the four corners of the pixel) in world space. This naturally defines a set of points in the surface's (s, t) parameter space which can then be mapped to the (u, v) texture map space (see Fig. 2–5). The resulting area in texture space will hopefully span more than one pixel, and will therefore be filtered to choose a colour for the original pixel in screen space. The (u, v) parameter space of the texture T is defined within the unit square so the texture element (*texel*) lookup is specified by $T(s - \lfloor s \rfloor, t - \lfloor t \rfloor)$ to allow the texture to be tiled if necessary.

There are two predominant problems with texture mapping: (1) it is often difficult to find a suitable surface parameterisation, and (2) aliasing (caused by a fixed resolution texture). It is also worth noting that finding a suitable image to use for the texture map can be problematic. We will elaborate on these below.

Unfortunately, we will not always have a well defined (s, t) parameter space for the object being textured. This is the case, in fact, for most non-parametric representations such as polygonal meshes, or when the object to be textured is

composed of several smaller geometric entities as is frequently the case with CSG. Two-part texture mapping was introduced by Bier and Sloan to handle precisely this situation [8]. In two-part texture mapping, the texture is first mapped to an intermediate parametric surface such as a cylinder or a sphere which encloses the object to be textured using the conventional method described above. The second stage maps the new three dimensional texture pattern onto the object's surface, usually using a form of ray-tracing to determine the texture element which is closest to the object point being shaded. Although this works well, it has the disadvantage that the intermediate shape must be chosen manually. An example of two-part texture mapping using a plane, cylinder and sphere as intermediate objects is shown in Fig. 2–6.

As mentioned above, aliasing is a serious problem encountered when texture mapping. The difficulty stems from the pre-image in texture space of the pixel being shaded in screen space: because we do not know the exact shape of the curvilinear pre-image in texture space, and moreover, this shape changes for adjacent pixels, proper filtering of the texture is very computationally intensive since for each pixel we need to calculate an average over all the relevant texels. A largely unsolved problem occurs when the pre-image of a pixel maps to a *sub-texel* (a unit smaller than a texel) in texture space, since magnification is necessary. This can occur, for example, when a textured object in the scene approaches the viewer and hence the texture itself needs to be magnified in accordance with the camera motion.

An approximation to the filtered pixel value is often computed using a technique known as *mip-mapping*[2] [82]. With mip-mapping, the desired texture is pre-filtered to several smaller versions of the texture (an assumption

[2] MIP stands for *multum in parvo* – many things in a small place.

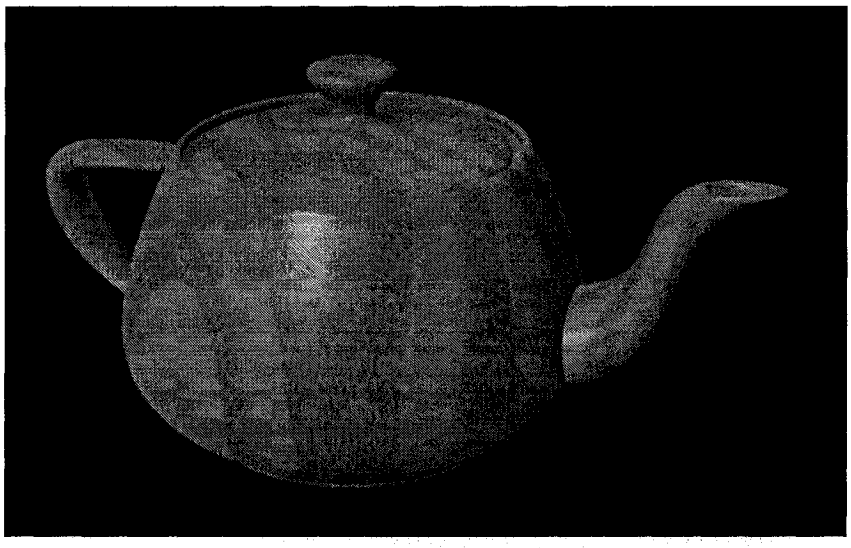


Figure 2–7: An example of a teapot which is textured using a 3D (solid) wood texture.

is made that the pre-image is very close to a square) forming an image pyramid. When selecting the texels, the appropriate size texture from the mip-map is extracted. This saves computation time since each texel at the appropriate scale is already a filtered version of the original larger scale texture. Mip-mapping aids computationally when the texture image must be compressed, but does not handle the problem of magnification, that is, when the desired (u, v) range of the texture does not contain many (if any) pixels. In this relatively common scenario, the same pixel of the texture map is used repeatedly for adjacent pixels in screen space, leading to aliasing. If we allow the world to be dynamic, these kinds of problems become quite evident in the form of textures "jumping" around on surfaces based on the camera position. Procedural textures do not suffer from the magnification problem, as will be outlined in Sec. 2.2.4.

Another problem with texture mapping is that linear interpolation of the texels will cause distortion when using a perspective projection camera model. This distortion is most noticeable in the form of features in the texture not being correctly foreshortened.

For some types of texture, it makes sense to think in terms of three dimensional textures[3] [65, 63]. For example, an object made of wood will look more realistic if the wood texture is truly 3D instead of a 2D texture map (see Fig. 2–7). In this case, we can think of the texture being addressed by $T(x, y, z)$ of some local coordinate frame scaled accordingly for the object. Three dimensional textures are prohibitively large if stored explicitly and are therefore usually defined procedurally. An advantage of 3D textures is that objects of arbitrary complexity can be textured in a coherent fashion, without seams or singularities, unlike their 2D counterparts.

An alternative use of texture maps is for bump-mapping [9]. For this technique, instead of modulating the colour of the current pixel, the normal used in the local illumination calculation is perturbed according to a value in an associated bump map following the same addressing methods outlined above. The resulting shading changes will give the appearance of surface detail not present in the object's geometry. Note, however, that silhouette edges will still follow the underlying geometry of the model.

A major disadvantage of sampled texture maps is that they must be stored. This can lead to a vast increase in storage requirements, particularly when the scene is complex and many textures are used. This problem is slightly worsened by the use of mip-maps described above as they are $\frac{4}{3}$ over-complete. Textures used in texture mapping also do not easily allow for subtle variations, and generally do not support temporal variations. These issues will be addressed in the following section.

[3] Three dimensional textures are sometimes referred to as *solid textures*.

2.2.4 Procedural Techniques

Procedural techniques are an active area of research in computer graphics in domains including shading, texturing, modelling and animation:

Definition 2.2.1 (Procedural Technique)

A Procedural Technique is a code segment or algorithm that specifies some characteristic of a computer-generated model or effect. D. Ebert [31].

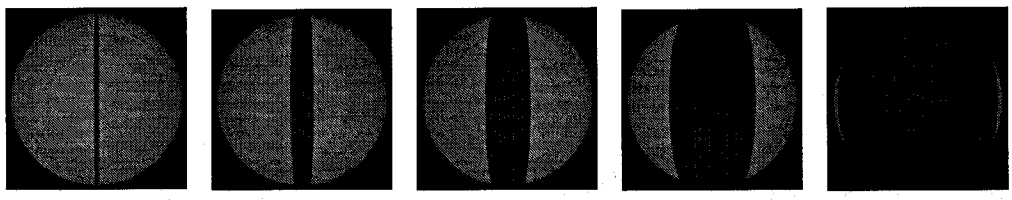
Abstraction is one of the key advantages of a procedural technique: rather than explicitly storing the complex details common to an explicit model, these details are abstracted into a function or algorithm. As mentioned above, modern renderers allow the illumination model to be specified on a per-object basis, in the form of what are called procedural shaders. This allows each object to exhibit arbitrarily complex light interaction. If there are many objects in the scene, one could use a more simplified illumination model like the Phong illumination model for less important or less noticeable elements, and could use a bi-directional distribution reflection function for objects whose lighting details are more important. Again, because procedural shaders are queried for each individual location on the surface being shaded, it is possible to use them to create a texture on the surface.

An important characteristic of procedural techniques is that of parameterisation: in a procedural model, we can assign a parameter to a meaningful concept. For example, a procedural texture for a cloth weave may have a parameter which specifies the tightness of the weave, or a procedural model for representing a stucco ceiling may have a "bumpiness" parameter. There are numerous advantages to the various procedural representations, however, in this section, we will focus exclusively on procedural texturing.

A procedural texture is a function which, given a set of input parameters $\mathbf{x} = (x_1, \ldots, x_n)$ which control the appearance of the texture, returns the colour of

Algorithm 1 A simple shader algorithm which creates a vertical red bar based on the value of the parameter ω.

Require: $(u, v) \in [0, 1]^2, \omega \in [0, 1]$
  if $0.5 - \frac{\omega}{2} \leq u \leq 0.5 + \frac{\omega}{2}$ **then**
    return Red
  else
    return White
  end if



(a) $\omega = 0.01$ (b) $\omega = 0.05$ (c) $\omega = 0.17$ (d) $\omega = 0.5$ (e) $\omega = 0.9$

Figure 2–8: An example of using the red bar shader described in Alg. 1 to texture a sphere using various values for ω.

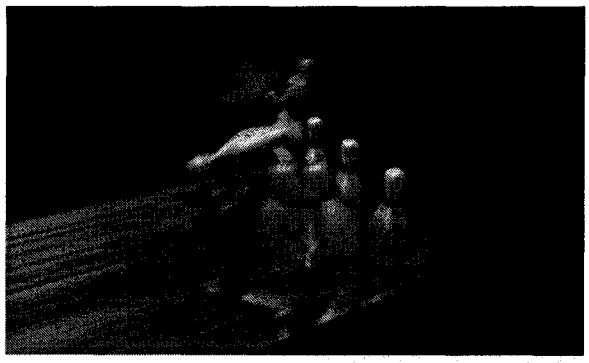
the surface at the point (u, v) queried:

$$p(u, v, \mathbf{x}) = f(u, v, \mathbf{L}, \mathbf{N}, O_d, O_s, \ldots, \mathbf{x}) \tag{2.3}$$

where p is a procedural texture having a parameter vector $\mathbf{x}$ and like a texture map is indexed by $(u, v) \in [0, 1]^2$. In general p is a function not only of the parameter vector $\mathbf{x}$ of the texture itself, but also of the light direction ($\mathbf{L}$), surface normal ($\mathbf{N}$), object diffuse and specular colours (O_d, O_s), etc., as illustrated with the function f above. We can think of these additional parameters as being functions of (u, v). For the work in this thesis, we consider procedural textures only in terms of their coordinates (u, v) and their parameter vector $\mathbf{x}$. That is, we seek only to recover textures under constant lighting conditions on a plane.

Each procedural texture will be represented by a unique function requiring a distinct set of parameters relative to that texture. For example, a trivial single parameter shader which draws a centred red vertical bar on a white background might be formulated as in Alg. 1. Here the single texture parameter ω controls the width of the red bar. Figure 2–8 shows some images rendered using the red bar

(a) (b)



(c)

Figure 2–9: Example images rendered using procedural shaders for all surfaces. (a) ©Michel Joron 2004, (b) ©Jonathan Merritt 2004, (c) Pixar studios stock image.

shader. For an example of the power of procedural shaders, consider the images shown in Fig. 2–9 which were rendered using procedural shaders exclusively. An example of the range of one procedural texture is shown in Fig. 2–10.

There are several advantages to using procedural textures as opposed to the traditional texture mapping methods described in Sec. 2.2.3:

- Compact representation: Because a procedural texture is an implicit representation of some textural phenomenon in the form of an algorithmic description, the amount of storage required is negligible compared to what is required to store texture maps.

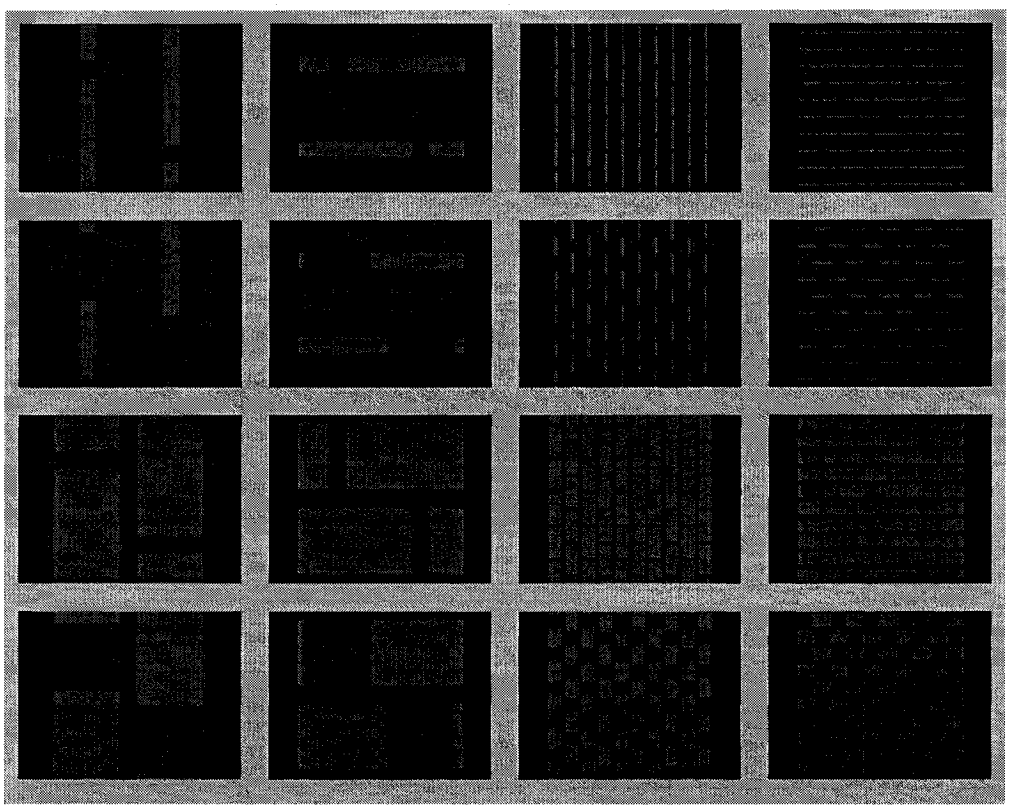


Figure 2–10: Several example textures showing the diversity of an individual shader with varying input parameters. Here the parameters were the frequency of the bars, the thickness of both the vertical and horizontal bars, and the overall orientation.

- Unlimited resolution: Again, due to the implicit definition, procedural textures are resolution independent and can therefore be used to generate textures of arbitrary resolutions.
- Parameterisation: One can assign values to meaningful aspects of the given texture such as the age of the wood in a parquet tile, or the frequency of horizontal bars in a weave pattern.
- Support minor changes: Slight changes to the resulting texture are often generated by slight parameter changes since each procedural texture generally represents an entire class of similar textures.

- Support temporal changes: A procedural framework allows for the texture of an object to change over time (see chapter 5), which would be very impractical using traditional textures.
- Expression of object properties: It is relatively simple to generate procedural shaders which will show curvature, or other object properties (through false colour) often necessary for visualisation and CAD/CAM applications. It is also possible to use any desired illumination model since this is generally part of the procedural shader.
- High-dimensional textures: As mentioned in Sec. 2.2.3, solid textures are easily represented using procedural techniques, while they have excessive space requirements when represented using traditional textures. Another example of a high-dimensional texture, called *hypertexture* will be described below.

Despite the numerous advantages of using procedural techniques for texturing, there are, unfortunately, some shortcomings. The most notable is that once one has a procedural shader, it can be quite difficult to obtain the correct parameters to synthesise the desired texture. In addition, parameterised textures can sometimes be unstable, that is, a small change in a parameter can lead to a significant change in the synthesised texture. Even when the parameterised texture is largely stable, some shaders have a non-trivial number of parameters which can simply be too unwieldy to specify manually[4] . These factors can make it difficult for the end user to obtain the desired results.

[4] The water surface shader in the film *The Perfect Storm* had over 200 parameters. Apparently, there was no single individual at ILM who knew what each parameter controlled [34].

The specification of procedural shaders is not a task suited to everyone since it requires an algorithmic formulation in a given shading language of how the desired texture should appear. This is obviously more difficult than using a typical paint program to create a single texture to be used as a texture map. Moreover, the implementers of procedural shaders must worry about problems such as anti-aliasing. For example, the simple shader presented above in Alg. 1, will alias badly since there is very high frequency content (step edges). In general, the author of a procedural shader does not know at which points the texture will be sampled, and must therefore minimise the high frequency content through the use of smooth edges. This is an example of one of the many issues which must be resolved during the specification of a procedural shader.

Procedural textures are also not a panacea; there are many textures which simply cannot be easily formulated procedurally. Complex visual structures which do not seem to have an underlying pattern (such as a human face) are very difficult, if not impossible, to represent procedurally.

Image synthesis is usually slower when using procedural textures since the procedure must be evaluated for every pixel in addition to the illumination model. There has been recent work which addresses this issue and the results are promising [57, 64, 58]. Rendering speed is, however, perhaps the smallest concern when weighed against the advantages of using procedural textures, especially considering the dizzying pace at which graphics hardware improves. This is perhaps most interesting for the video game industry where high quality real-time rendering is always the goal. Until now, these applications have had to make use of many texture maps for their virtual environments which are less visually compelling, often produce aliasing effects during motion, and have extremely high storage demands. Photo-realistic applications such as motion pictures are

not rendered in real-time and are therefore able to take advantage of the increased quality and flexibility of procedural texturing.

Note that by using procedural textures for this work, we obtain several advantages over either image samples as texture maps, or the use of stochastic image-based texture synthesis as proposed by Efros *et al.* [33, 32], Wei and Levoy [79] and others (see chapter 3). Namely, as mentioned above, procedural textures are very compact, extremely flexible, can be evaluated in arbitrary order (useful for variable level of detail applications) and are resolution independent. Using a procedural texture also allows us to generate textures that are akin to a target in some desired way, while still allowing us the freedom to make useful changes.

CHAPTER 3

Previous Work

THE previous work in the field of texture synthesis can largely be divided into two classes: synthesis which is not based on a texture sample (herein referred to as *traditional texture synthesis*), and synthesis which is based on a small texture sample where the desire is to *grow* a larger texture field of that particular texture. For the remainder of this chapter we will refer to the latter as *sample-based texture synthesis.*

In this chapter we will give an overview of the most popular techniques for both traditional texture synthesis, as well as sample-based texture synthesis. Finally we will describe previous work directly related to the automated specification of procedural textures based on a sample texture.

Our work deals with the selection of one or more procedural textures and associated parameters given a specification in terms of an sample texture. This is loosely related to research which seeks to synthesise a large texture field given only a small sample of the desired texture. While texture synthesis methods share with our work the ability to generate arbitrary texture fields from a small sample,

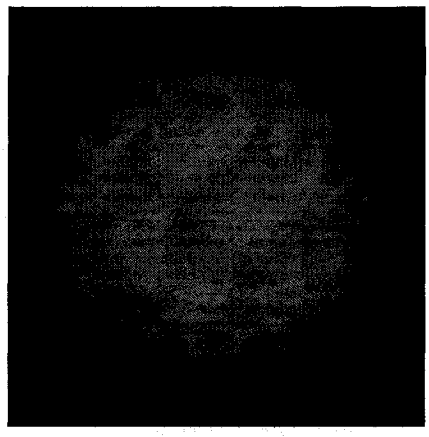


Figure 3–1: A fireball made using hypertexture. (Image by Ken Perlin.)

they differ in terms of the compactness of the description, the scientific objectives, and the manner in which the results can subsequently be re-configured.

There are many specialised techniques for texture synthesis but few of them are general solutions, that is, most are best suited to a specific type of texture.

3.1 Traditional Texture Synthesis

Traditionally, textures which were synthesised for use in texture mapping were not necessarily based on real-world phenomena, but rather were designed with very specific applications in mind. These early synthesis techniques were sometimes able to produce elaborate textures, but it was difficult to effectively control their appearance.

We will give an overview of two such methods below, namely hypertexture, and reaction-diffusion textures.

3.1.1 Hypertexture

In 1989, Perlin and Hoffert described a concept, which is an extension to procedural solid texture synthesis applied to volumetric regions, called *hypertexture* [66]. Their motivation was that many objects have surfaces which are very difficult, if not impossible, to model explicitly. Examples include fur, hair,

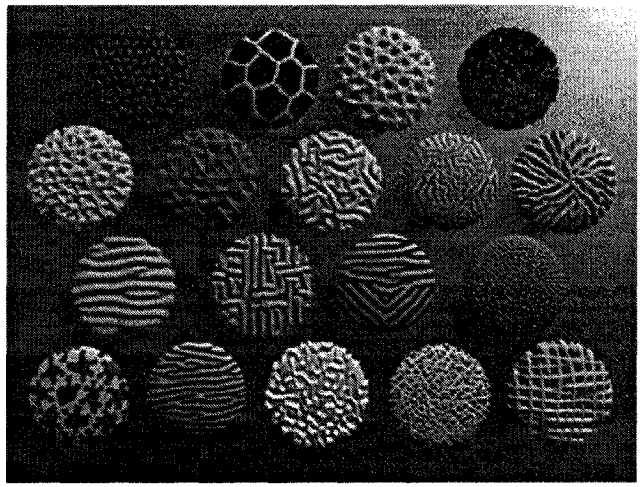


Figure 3–2: An example of several reaction diffusion textures. (Image by Andrew Witkin.)

fire, fluid flow, and erosion effects (see Fig. 3–1). They modelled such objects as *soft objects*, i.e., objects which had a density function $D(x, y, z) \in [0, 1]$ which described the density of a 3D shape for all points in $\mathcal{R}^3$. The *soft region* consisted of all points such that $0 < D(x, y, z) < 1$, the outside by $D = 0$, and the inside by $D = 1$. In addition to the density function, they defined density modulation functions (bias, gain, noise, and turbulence) which were used to modulate the object's density within its soft region. They also defined the boolean operations (union, difference, complement, and intersection) for these modulation functions thus forming a toolkit which could be used to model these soft objects. Although some highly successful images were produced, the selection and combinations of density functions, as well as parameter tweaking, still needed to be performed manually.

3.1.2 Reaction-Diffusion Textures

Reaction-diffusion (RD) texture generation is a technique which can be used to simulate a class of natural textures, or patterns, which arise from local, non-linear interactions of excitation and inhibition. Examples of such textures include

various kinds of stripes, weaves, lattices, and mazes [83, 77]. Some examples of RD textures are shown in Fig. 3–2.

The principal idea behind reaction-diffusion systems is to simulate the evolution of a concentration of *morphogens*[1] , $C(x, y)$, through two concurrently operating processes: *diffusion* of morphogens, and *reactions* that produce and destroy morphogens depending on their concentrations. This evolution is simulated until a stable pattern of concentrations is reached, at which time these concentrations are interpreted as textures, usually by assigning colours, or intensities depending on the underlying concentrations, although these patterns have also been used for bump mapping and for opacity maps.

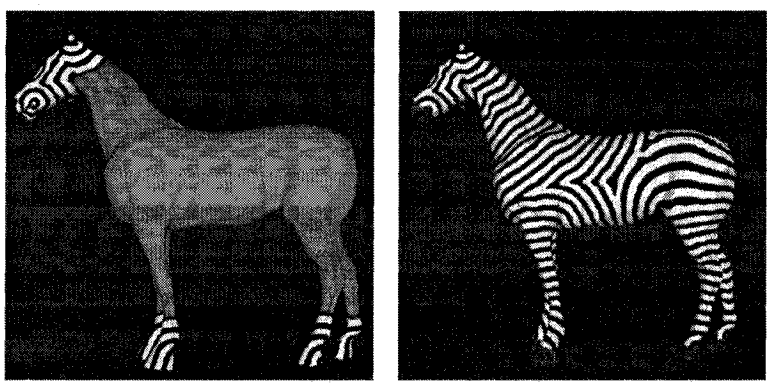
The reaction-diffusion model proposed by Witkin and Kass incorporates three processes – diffusion, dissipation, and reaction [83]. Diffusion controls the transport of morphogens from higher to lower concentrations, dissipation causes concentrations of morphogens to decay exponentially in the absence of other influences, and reaction controls the rate of morphogen production:

$$\dot{C} = \underbrace{a^2 \nabla^2 C}_{\text{diffusion}} - \underbrace{bC}_{\text{dissipation}} + \underbrace{R}_{\text{reaction}}, \tag{3.1}$$

where $\dot{C}$ is the time derivative of C, $\nabla^2 C$ is the *Laplacian* of C, a is the rate constant for diffusion, b is the rate constant for dissipation, and R is the reaction function.

In order to create new patterns, Witkin and Kass extended the RD model in several ways. First, they allowed the diffusion to be anisotropic. In order for C to diffuse at different rates, the a^2 in Eq. 3.1 can be separated into independent rate constants in x and y. They also propose a method for arbitrary non-axis

[1] Morphogens are hypothetical chemical agents which take part in *morphogenesis*, the formation and differentiation of tissues and organs.



(a) (b)

Figure 3–3: An example of reaction-diffusion textures *grown* directly on a surface. The zebra stripes were started on the hooves and head as shown in (a). The final image is shown in (b). Images ©Greg Turk.

aligned principal directions, the details of which are beyond the scope of this overview. Second, they allowed the diffusion rates to change in different areas of the concentration, a technique they refer to as space-varying diffusion. This is accomplished through the use of a diffusion map which corresponds to the diffusion rates and directions for each point in the concentration. In practise, the diffusion map is only defined for selected points, and values are interpolated for other areas of the concentration. Witkin and Kass also demonstrated that RD texture patches could be sewn together seamlessly through the use of shared boundary conditions, a special case of which allows textures to repeat periodically.

Turk made two important contributions to reaction-diffusion texturing: (1) the creation of patterns more complex than had previously been attributed to RD systems, and (2) he described a technique for *growing* RD textures directly on polygonal surfaces (see Fig. 3–3) [77]. He achieved more complex patterns by having one RD system create an initial concentration, and then using this concentration as the initial condition for a second RD system. Striking patterns were created when he stopped the initial concentrations prior to reaching stability, and then allowed the second system to stabilise using the result of the first system.

He also allowed several concentrations to be simulated together with reaction functions defined for each of the different morphogens in relation to the others. These two ideas can obviously be used to create a wide variety of patterns. Turk's technique for growing patterns directly on polygonal surfaces allowed RD textures to be immune to the surface parameterisation problems inherent with traditional texture mapping.

Unfortunately, reaction-diffusion textures remain complicated to use due to the necessary specification of the initial conditions (the initial morphogen concentrations), the rate constants, the diffusion map (if anisotropic and space varying diffusion are to be used), the reaction functions, not to mention, the point at which to stop the simulation if various RD textures are being mixed. Despite various optimisations in the numerical solution of partial differential equations, RD textures are still computationally expensive.

3.2 Sample-Based Texture Synthesis Techniques

Sample-based texture synthesis techniques can be used to create a synthetic texture which resembles an input image according to a particular texture model. There are two classes of texture models, namely deterministic and stochastic. A deterministic texture is characterised by a set of primitives, and a set of rules which governs their placement. Examples include a tile floor, or wallpaper with a spotted pattern. A stochastic texture, however, does not have any primitives which can be easily identified (tree bark, sand, stucco). In practise, many real-world textures have some combination of these characteristics.

3.2.1 Steerable Pyramid Statistical Matching

In 1995, Heeger and Bergen proposed a texture analysis and synthesis model for generating stochastic textures based on an input sample [42]. Their method is able to synthesise an arbitrary amount of the sample texture by coercing a noise

Algorithm 2 Heeger and Bergen's Texture Matching Algorithm

```
MatchHistogram(noise, texture)
analysis-pyramid = MakePyramid(texture)
for several iterations do
  synthesis-pyramid = MakePyramid(noise)
  for matching sub-bands in analysis and synthesis pyramids do
    MatchHistogram(synthesis-band, analysis-band)
  end for
  noise = CollapsePyramid(synthesis-pyramid)
  MatchHistogram(noise, texture)
end for
```

image of the desired resulting texture size to have the same intensity histograms within specific bands of frequency space as the input image. This is accomplished by using two fundamental image operations: (1) decomposition of an image into an image pyramid (and collapsing an image pyramid back into an image), and (2) histogram matching.

The histogram matching is in fact a generalisation of histogram equalisation. It is accomplished by creating two lookup tables: the cumulative distribution function (cdf) of one image, and the inverse cdf of the other image. These two functions are then used to match the histogram of one image to the other.

The entire texture analysis/synthesis algorithm is shown in Alg. 2. In practise they used a steerable pyramid which kept four images for each level of the pyramid, each a response to an oriented filter so that anisotropic textures could be synthesised, the details of which are beyond the scope of this discussion. Inspection will reveal that there is no fixed number of iterations in the algorithm, and in fact there is no formal evidence that this algorithm converges. Heeger and Bergen claim, however, that 5 iterations are usually sufficient to produce a satisfactory synthesised texture. Using this algorithm, it is also possible to synthesise textures which are similar to several input textures by using the freshly synthesised texture instead of a noise image when synthesising the second texture. This extension is similar to those mentioned in Sec. 3.1.2.

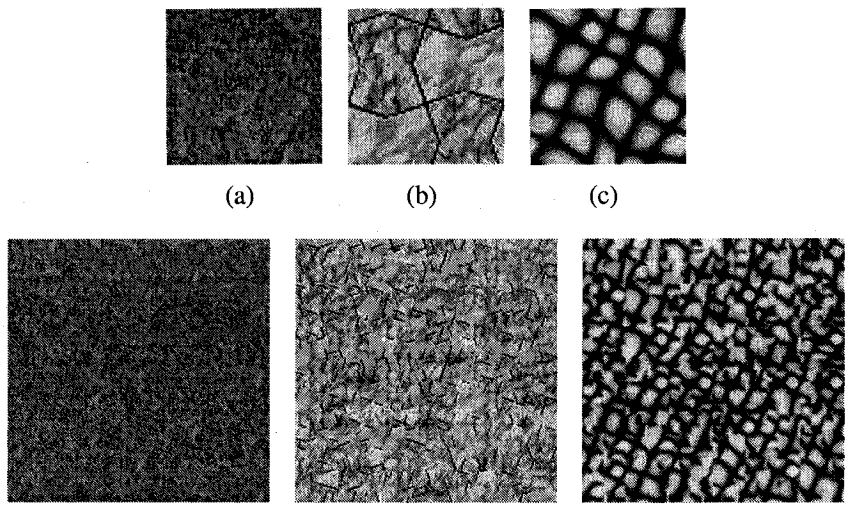


Figure 3–4: Textures synthesised using De Bonet's technique. The top row shows the texture samples, and the bottom row shows the resulting synthesised textures. Notice that the method performs acceptably for a stochastic texture sample (a), but fails for textures exhibiting even a slightly deterministic pattern as shown in (b) and (c).

De Bonet also proposed a texture synthesis technique based on image pyramids [27]. Instead of trying to match the histograms at each level of the pyramid, he samples the levels of the analysis pyramid where psychophysically motivated features have strong responses. These features are simple edge and line filters as well as Laplacian response, and must be present at each parent level of the input pyramid to be replicated in the synthesis pyramid. Similar regions of the levels in the synthesis pyramid are also randomly rearranged to increase visual difference from the input texture while maintaining minimal perceptual difference in terms of texture. After the completion of the sampling process for each level of the synthesis pyramid, it is collapsed to produce the synthesised texture. Due to the randomness, this method seems to work well only for purely stochastic textures. Another drawback exists in the way that images larger than the input are generated: the sampling is performed by simply tiling the input image without regard for whether it may be tile-able. This method also cannot model complex

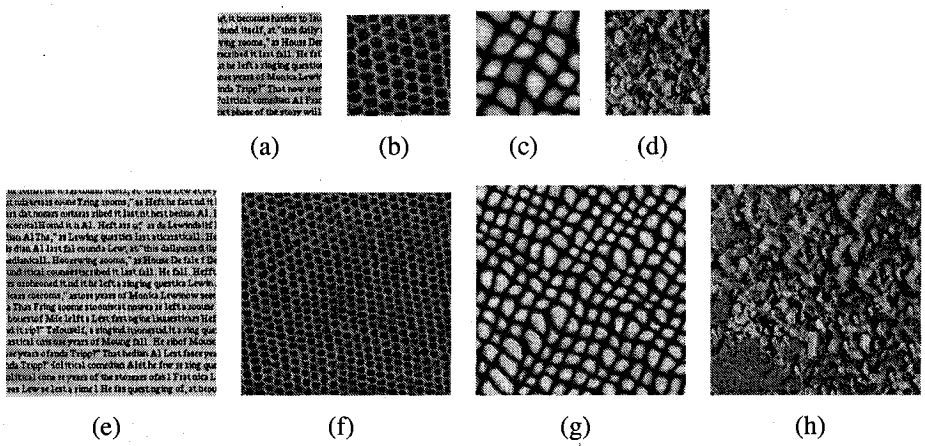


Figure 3–5: Textures synthesised using Efros and Leung's technique. The first row contains the texture samples, and the second row displays the synthesised textures. Notice how if incorrect choices are made early, it is impossible to recover as shown in the bottom left corner of (h).

visual structures since it uses only local constraints during sampling. Examples of textures synthesised using this method are shown in Fig. 3–4.

3.2.2 Markov Texture Synthesis

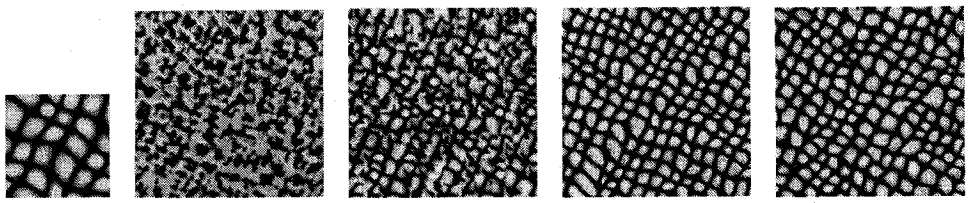
Markov texture synthesis techniques are based on the assumption of locality. That is, that the appearance of a certain texture element (texel) can be determined by its surrounding neighbourhood of texels. In other words, that the texture being synthesised can be considered to be a Markov process if time is reinterpreted as space within the texture and therefore has the following property: given that its current state (texel) is known, the probability of any future event of the process is not altered by additional knowledge concerning its past behaviour (more than n pixels in a local neighbourhood).

The specification of the neighbourhood varies according to the technique being used. These techniques can be used to *grow* an unlimited amount of texture based on a small sample, or can be used to fill in holes in an image as is necessary for image in-painting [6, 7, 52].

A technique due to Efros and Leung grows a texture, pixel by pixel outwards from an initial seed until the desired size is reached [33]. Their algorithm essentially finds a set of candidate regions in the sample texture which are similar to the area centred at the current (unknown) pixel in the synthesised texture. They compute the similarity using a sum of squared differences (SSD) of the pixel intensity values from the two neighbourhoods, weighted by a two-dimensional Gaussian kernel. A histogram of pixel values is computed from the best neighbourhoods, and the final pixel value is determined by sampling this histogram either uniformly or weighted according to the neighbourhood similarity value.

The final pixel value is then used in the synthesised texture, and the algorithm continues outward (or inward for in-painting) until all pixels have been filled in. Of course, this only works when a single pixel needs to be determined, since when there is more than one pixel to be synthesised, the neighbourhood in the synthesised image will not be complete. To correct for this situation, when neighbourhoods are compared, only known pixels are used for the similarity measure, and the error is normalised by the total number of known pixels. In the case where there are no known pixels yet in the synthesised texture (the starting condition when not doing hole-filling) a 3×3 seed is taken randomly from the sample texture and is used as a starting point for synthesis.

Due to the bootstrapping nature of this algorithm, if a pixel is filled in incorrectly, the resulting texture can have a large region which is not similar to the input texture. Assuming these incorrect pixels could be located, a possible solution to this problem would be to allow limited backtracking to choose a more appropriate pixel. Textures synthesised using this technique are shown in Fig. 3–5.



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

Figure 3–6: Comparison of different texture synthesis techniques. The sample texture is shown in (a), the result using Heeger and Bergen's technique is shown in (b), (c) is synthesised using De Bonet's technique, (d) is synthesised using Efros and Leung's technique, and (e) is synthesised using Wei and Levoy's technique. There is little difference in appearance between (d) and (e), however Wei and Levoy's method is an order of magnitude faster than Efros and Leung's method.

Wei and Levoy have shown a method similar to that of Efros and Leung which produces textures which are of equal or higher quality and can be synthesised much faster [79]. The main differences from previous techniques is that the texture is filled in progressively, from top to bottom, left to right, and they define their neighbourhood for search in the input sample to have dependence only on previous pixels which have been added to the synthesised texture (the system is *causal*). In addition, in order to capture texture elements whose scale is larger than the search neighbourhood they use a multi-resolution approach: synthesis is performed from low resolution to high resolution levels in a Gaussian pyramid, with the neighbourhood definition updated to include the levels below the level currently being synthesised.

They have also increased the search speed of matching neighbourhoods between the sample and target textures by considering each neighbourhood to be a point in a higher dimensional space, and using a nearest point searching algorithm (tree structured vector quantisation). These algorithms typically involve preprocessing the point set, but provide much faster search times. Results using their method are shown in Fig. 3–6.

Wei and Levoy also propose a method for synthesising *temporal textures*. Temporal textures are visual sequences with indeterminate extent both in space and time. For this type of synthesis, the neighbourhood definition is augmented to include the texel values from nearby frames within a texture animation. Searching in this higher-dimensional space is clearly computationally expensive, and hence results take a long time. It is also worth noting that in order to use this method, one must already have a sample of a temporal texture.

Schödl *et al.* have developed a technique for creating temporal textures of arbitrary length based on small video sequences which they call *video textures* [70]. Their method analyses a video sequence to extract its structure, and is then able to synthesise a new, similar looking, non-repeating video. The videos are represented as Markov processes with each state corresponding to a frame in the video, and the probabilities correspond to the transition likelihoods from one frame to another.

The creation of a video texture consists of an analysis phase to extract the structure from the video, followed by a synthesis phase. For the first part of the analysis phase, the similarity of all pairs of (brightness equalised) frames in the sample video sequence is computed and stored in a matrix $D_{ij} = ||\mathcal{I}_i - \mathcal{I}_j||_2$ for all pairs of video frames I_i and I_j. These distances are then mapped to probabilities, $P_{ij} \propto \exp(-D_{i+1,j}/\sigma)$, with each row of P being normalised so that $\sum_j P_{ij} = 1$. When synthesising the new video sequence, a new frame is selected according to the distribution of P_{ij}, with the value of σ determining the smoothness between adjacent frames. This work was extended so that individual pictures could contain moving elements [24], and was also applied to panoramic images in the form of panoramic video textures [2].

If we consider texture synthesis methods based on Markov models of texture as presented above, then although these techniques produce compelling results,

they have several limitations. For instance, the Markov framework does not easily allow for minor changes in the characteristics of the texture being generated (wider bricks, puffier clouds, etc.), although there has been promising work in this area [84, 19]. Another shortcoming of these techniques is that the output resolution can never be higher than the resolution of the input image: while increasing the desired size of the synthesised texture will produce more texture, its resolution (distance between samples) can never be higher than the given sample texture. This means that these texture synthesis methods all suffer from the magnification drawbacks described in Sec. 2.2.3, and are therefore not very well suited to photo-realistic rendering.

3.2.3 Bi-directional Texture Function

Dana *et al.* have a somewhat different approach to the texture synthesis problem, and propose a method which is similar to the techniques for measuring the BRDF discussed in Sec. 2.2.1 [26]. They define a bi-directional texture function (BTF) analogously to the BRDF: for each possible viewing and illumination direction, the BTF of a particular texture returns an image. This model accommodates both isotropic and anisotropic textures very well. Actual textured materials (carpet, velvet, stucco, etc.) are imaged for each possible viewing and lighting angle, and the resulting texture images are stored for later use. At rendering time, the correct texture samples are retrieved and blended together. Although this approach has produced some nice demonstration images, the time and machinery necessary to accurately sample each desired texture for all orientations, as well as the space needed to store these samples is quite prohibitive.

Suen and Healey have lowered the storage requirements via a form of subspace modelling to yield highly realistic reproductions of specific physical surfaces once the requisite measurements have been acquired [74]. This method

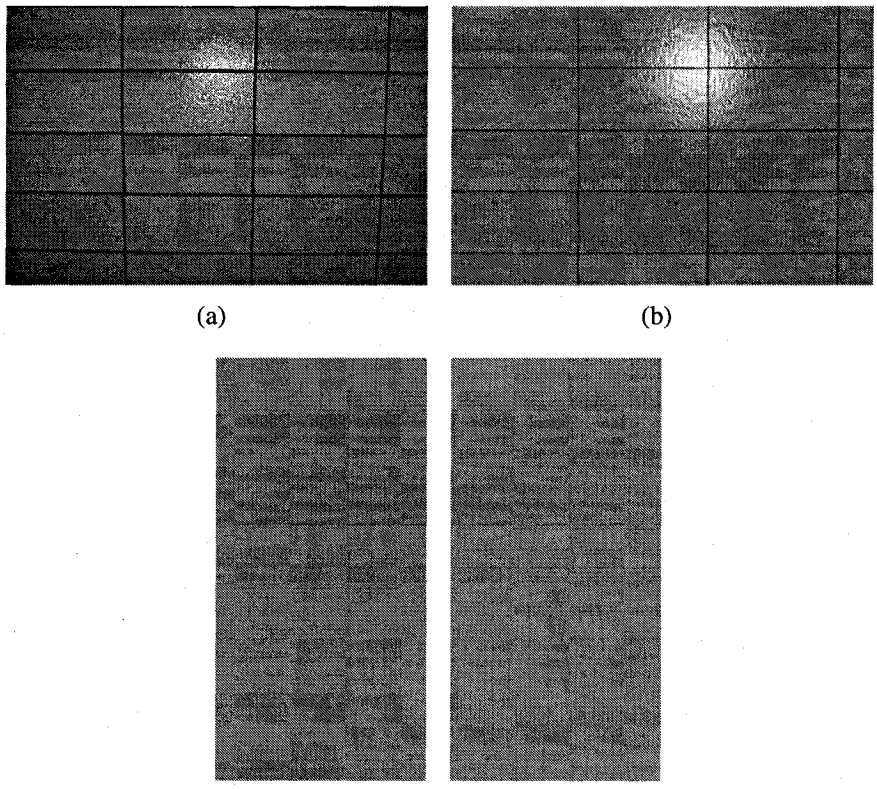


Figure 3–7: Textures synthesised using Lefebvre and Poulin's technique. The input texture for the rectangular tiling method is shown in (a), and (b) is the synthesised texture. The input texture for the wood texture method is shown in (c), and (d) is the synthesised texture.

continues to be impractical due to the necessity of physical sampling the desired texture in a very controlled environment.

3.3 Procedural Texture Matching

Lefebvre and Poulin have developed a procedural method for texture analysis and synthesis for highly structured textures [51]. This method combines some of the features discussed above with the final goal of producing a procedural texture similar to the input sample. This allows the graphic artist to use a high resolution texture, as well as to have the ability to tune some of the parameters of the resulting texture should they desire a slightly different appearance.

Their work focuses on two types of structured textures: rectangular tilings (such as can be found in brick walls or ceramic tile floors), and wood. In the case of rectangular tilings, their method is based on the Fourier analysis of a (manually tuned) segmented texture. They measure features such as orientation and the height of rows in the tiling directly from the phase and amplitude images of the texture transformed to the frequency domain. Given this information, they can then measure other parameters such as tile width and row offsets by tracing horizontal scan lines within the original image in the spatial domain. As a final step, the user must select a region in the sample texture which represents the centre of the tiles (the brick itself in a brick wall texture), and a region which is representative of the inter-tile area (mortar in the case of a brick wall). They then use Heeger and Bergen's method (discussed in Sec. 3.2.1) to synthesise these regions in the final texture. A similar approach is presented to synthesise wood textures based on an input sample. There are 10 parameters for their wood procedural texture model which are estimated using various scan-line and Fourier methods similar to those used for rectangular tilings. Two examples of their method can be seen in Fig. 3–7.

Although the results for these two specific types of texture are very positive, it is clear that this method can not be generalised to handle arbitrary textures. In addition, even in their restricted domain of textures there are some limitations: the rectangular tiling must be regular, and due to the method used to synthesise the various texture elements, arbitrary resolution is not really possible for reasons discussed above.

The approach to texture matching presented in this thesis is better able to handle generic texture samples, and the resulting procedural textures can be used to render true high resolution photo-realistic images. This approach is demonstrated below for both stochastic and deterministic texture samples.

In addition, our method uses procedural textures written in a standard shading language[2] and as such does not need to be part of a special rendering framework unlike some of the methods presented in this chapter. Our method for procedural texture synthesis based on a given texture sample will be outlined in the following chapter.

[2] We use Pixar's RenderMan® shading language since it is the prevalent shading language used in the computer graphics industry; however, our system can be easily extended to use other shading languages. For details, see appendix A.

CHAPTER 4

Procedural Texture Matching

In this chapter we will describe our method for synthesising a texture procedurally based on a given sample texture. This method contrasts with those described in chapter 3 in that rather than modelling the texture statistically or measuring specific features within the texture, we seek to find a shader from a given library of shaders which can approximate the sample texture. Given a potentially similar shader, our method then fine tunes the shader parameters in order to improve the similarity between the two textures. Computing the similarity of textures based on a human psychophysical model is an open problem, however, our method can easily use any supplied texture comparison metric. Although a more extensive discussion of texture similarity is beyond the scope of this thesis, in Sec. 4.5 we present two such similarity measures we have used with success.

Rather than returning a texture image, our method returns a shader, and a specific set of parameters for that shader which can then be used in an arbitrary rendering environment for image synthesis. While there are many advantages to this approach, the most notable are that the shader can be rendered at arbitrary resolutions, and that the graphic artist can manually fine tune the parameters to

achieve a slightly different appearance should they so desire. This work assumes the availability of a library of shaders, and that at least one shader in the library is capable of approximating the appearance of the desired texture. In chapter 6 we discuss a possible approach to relaxing these assumptions.

4.1 Approach

Recall from Sec. 2.2.4 that a procedural texture is a function which, given a set of input parameters $\mathbf{x} = (x_1, \ldots, x_n)$ which control the appearance of the texture, returns the colour of the surface at the point (u, v) queried:

$$p(u, v, \mathbf{x}) = f(u, v, \mathbf{L}, \mathbf{N}, O_d, O_s, \ldots, \mathbf{x}) \tag{4.1}$$

where p is a procedural texture taking a parameter vector $\mathbf{x}$ and like a texture map is indexed by $(u, v) \in [0, 1]^2$. Note that p is a function not only of the parameters of the texture itself, but also of the light direction ($\mathbf{L}$), surface normal ($\mathbf{N}$), object diffuse and specular colours (O_d, O_s), etc., as illustrated with the function f above. We can think of these additional parameters as being functions of (u, v). For the work in this thesis, we consider procedural textures only in terms of their coordinates (u, v) and their parameter vector $\mathbf{x}$. That is, we seek only to recover textures under constant lighting conditions on a plane.

Given an input target texture T, we wish to approximate its appearance using a procedural texture $p(u, v, \ldots)$. The solution to this problem will entail a multi-stage search strategy over the space of shaders in the shader library, as well as over the parameter domain of the shaders likely to produce desirable matches to the target texture. The details of this search technique will be given below. For the remainder of this thesis, we will refer to a procedural texture without specifying the (u, v) texture coordinates, but rather just the parameter vector, as in

$p(\mathbf{x})$. When the particular values of the parameter vector are not relevant, we will simply refer to the procedural texture as p.

Consider a *set* $\mathcal{P}$ of procedural textures $\{p_1, \ldots, p_n\}$, where each element p_i is a shader of arbitrary dimension, that is, it takes an arbitrary number of parameters. Given a texture target T, we wish to find the element $p_i \in \mathcal{P}$, and the associated parameter vector $\mathbf{x}_i$ such that $p_i(\mathbf{x}_i)$ produces a texture perceptually similar to T. That is, we want to maximise a similarity measure $S()$ between the procedural candidate and the target texture: $S(p_i(\mathbf{x}_i), T)$. The process for finding p_i and $\mathbf{x}_i$ is outlined below, and the similarity measure $S()$ is discussed in Sec. 4.5.

4.2 Searching in Texture Space

To find an appropriate shader and parameters, we need to search across the span of each shader's input parameters for a suitable match to the target texture T. For the remainder of this thesis, we will refer to the set of all valid parameters for a particular shader as its *parameter domain*, and to all the texture images a shader can produce as its *texture range*.

Unfortunately, it is unlikely that the similarity hyper-surface $s(\mathbf{x}) = S(p_i(\mathbf{x}), T)$ resulting from evaluating the target texture against the texture range of a particular shader will be convex (or even continuous for that matter). Of course, if there is a particular parameter setting for a shader which provides a good match to the target texture, theoretically, exhaustive search of the parameter domain would eventually find it, however, we desire a tractable solution. This suggests a space-time compromise consisting of a two-stage approach: a preliminary search using pre-computed data and an on-line refinement stage.

4.3 Global Search

As a pre-computation step, for each new shader that is added to the shader library, we generate a catalogue of samples in the parameter domain of that particular procedural texture p_i. We refer to the catalogue of samples for a particular shader as its *sample set* V_i, and the global set $\mathcal{V}$ is composed of all such sample sets for the shaders in the library.

Note that generating a sample for a particular parameter vector $\mathbf{x}$ entails rendering a new image of a plane textured using the shader $p_i(\mathbf{x})$. The texture samples in the catalogues are stored in an image database using a lossless compression format. Each sample is rendered at 256×256 pixels, with an average catalogue size of approximately 200 samples, combining for an average storage cost of 11MB per catalogue. The number of samples required for each catalogue is dependent on the number of parameters for the given shader. However, this relation is rarely exponential because most shaders contain a subset of parameters which are semantically motivated and hence control a wide range of the shader's output while the remaining parameters account for little variation. This pre-computation phase typically takes on the order of 4 hours for each new shader added to the library. While creating the shader catalogue is a computationally costly step, it must only be performed once for each shader.

Because it is possible that several parameter vectors will produce similar textures, we choose to sample the parameter domain of each procedural shader using an adaptive random sampling technique. This allows us to retain in our sample set only the parameter values which give us information about the *interesting areas* of the parameter domain – that is, the areas where the resulting texture range is not predominantly self-similar. A key issue, of course, is to sample densely enough to capture the expressiveness of the procedural texture while not over sampling and creating very large shader catalogues. The sampling

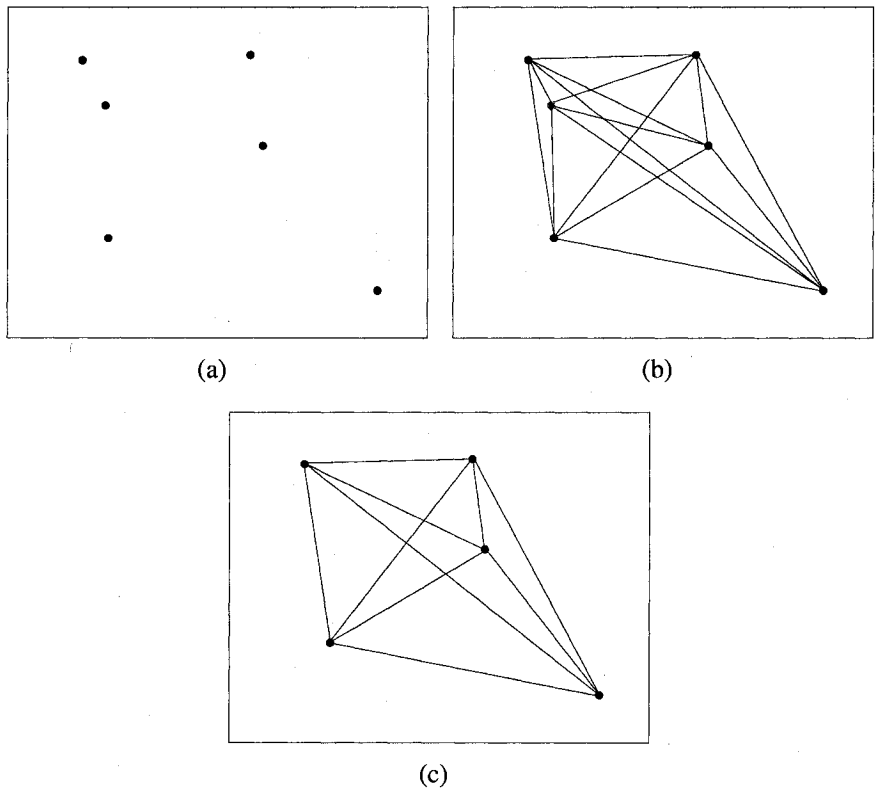


Figure 4–1: The construction step of the adaptive random sampling method. For the sake of illustration, we have limited the dimensionality of the shader to 2, and assume that the measure $S()$ is inversely proportional to the Euclidean distance between samples (i.e., samples which appear similar are closer together). In (a), samples are added randomly within the parameter domain of the shader. The fully connected graph is then constructed as shown in (b), and since one vertex is above the closeness threshold (shown in red), it is removed as shown in (c).

density (size of the sample set V_i) is determined by the end-user and can be adjusted per-shader if necessary.

Our adaptive random sampling is performed as follows. We build a graph G_i containing the vertices V_i and edges E_i for each shader p_i. Each vertex $v_l \in V_i$ corresponds to a sample point (parameter vector $\mathbf{x}_i$) in the parameter domain of the shader p_i, and we will therefore refer to the texture $p_i(v_i)$ and $p_i(\mathbf{x}_i)$ interchangeably. The edge weights $e(v_l, v_m) \in E_i$ correspond to the similarity measure between the vertices: $e(v_l, v_m) = S(p_i(v_l), p_i(v_m))$. The maximum number of samples for a shader p_i is determined by the per-shader constant N_i.

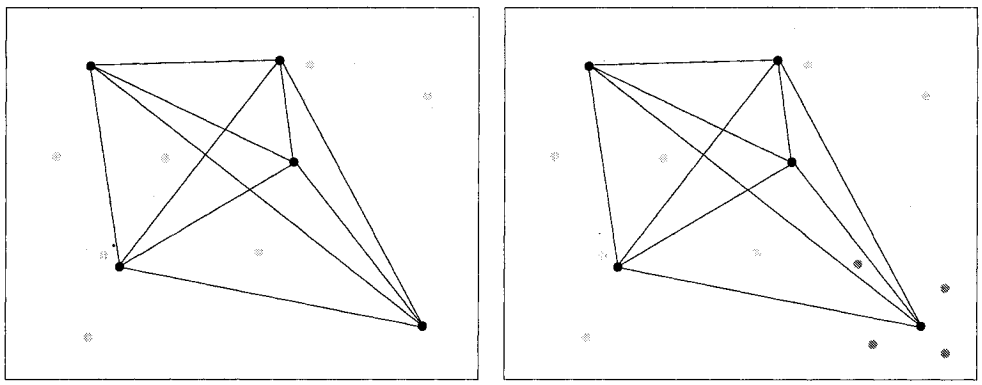
The adaptive random sampling technique involves a three step process: there is an initial construction step, a refinement step, and a pruning step. The construction step adds a number of random vertices to the graph, and the refinement and pruning steps iteratively improve the coverage of the parameter domain.

For the construction step, k random samples are added to the vertex set V_i. For each new vertex, edges are added to each existing vertex so that the graph is always fully connected. Full connectivity is not strictly necessary, however, it is desirable for the texture transformations which will be discussed in chapter 5. During the construction step, as each new vertex and its associated edges are added to the graph, it is checked to ensure that it is not too similar in appearance to some pre-existing vertex as determined by the similarity measure. That is, for a new vertex v_l if there exists a vertex v_m such that $e(v_l, v_m) > \omega$ where ω is a constant threshold, v_l is removed from the vertex set. The construction step is illustrated in Fig. 4–1.

Once it has been determined that a vertex v_l will be retained, we compute a measure of the smoothness in its local neighbourhood: given a reference point $\mathbf{x}$ and a set of points $D = \{d_1, \ldots, d_n\}$ distributed in the local neighbourhood of $\mathbf{x}$, we can compute a heuristic function measuring the *local smoothness* around the sample point $\mathbf{x}$ as follows:

$$H(\mathbf{x}) = \frac{1}{||D||} \sum_{d_i \in D} \frac{1 - S\left(P(\mathbf{x}), P(d_i)\right)}{||\mathbf{x} - d_i||_2} \tag{4.2}$$

This measure compares each point from the sample set D to the sample point $\mathbf{x}$ and these similarities are weighted by the L_2 norm between $\mathbf{x}$ and each point d_i under consideration. New points can be added incrementally as compute time permits in order to improve the measure. This function is a heuristic because it is



(a) (b)

Figure 4–2: The refinement step of the adaptive random sampling method. Again, for the sake of illustration, we have limited the dimensionality of the shader to 2, and assume that the measure $S()$ is inversely proportional to the Euclidean distance between samples. More samples are added globally as shown in yellow in (a). In addition, the vertex shown in blue in (b) was identified as being isolated, and hence random samples were added locally (shown in green). The new edges are not shown for clarity.

a discrete approximation of the local surface properties based on limited sample points, and is thus not equivalent to the derivative at the point $\mathbf{x}$.

For the refinement step of the adaptive random sampling method, we evaluate the current graph, and determine whether new points need to be added to provide better coverage of the texture range of the given shader. Points which may be too isolated are also identified during the refinement step.

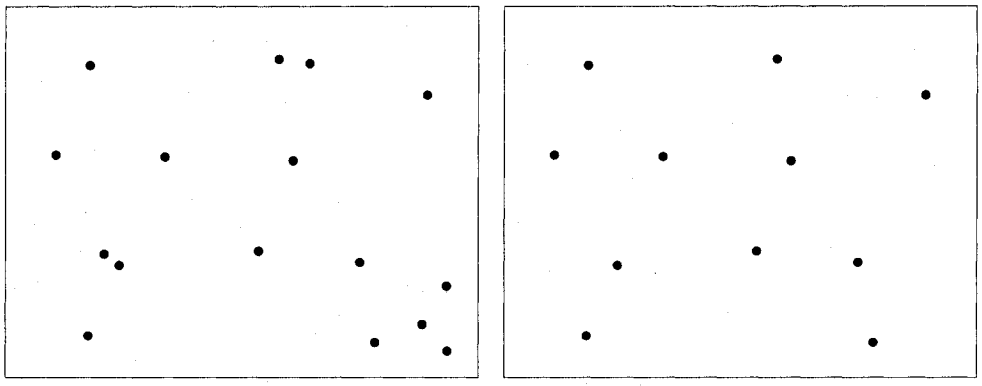
First, we compute a global measure of coverage of the current sample set:

$$\kappa(G_i) = \frac{1}{||E_i||} \sqrt[\sigma]{\sum_{l,m \in V_i \; l \neq m} \left(1 - e(v_l, v_m)\right)^{\sigma}} \tag{4.3}$$

where higher values of σ penalise graphs with isolated vertices. If $\kappa(G_i)$ is above some threshold, k more random points are added to the graph as in the construction step described above.

Secondly, we identify the most isolated vertices in the graph as defined by:

$$\gamma(v_l) = \alpha H(v_l) + \beta \max_{v_f \in V_i - \{v_l\}} \left(1 - e(v_l, v_f)\right) \tag{4.4}$$



(a) (b)

Figure 4–3: The pruning step of the adaptive random sampling method. For the sake of clarity, no edges are shown. (a) shows the vertices before pruning, and (b) shows the result after removing the vertices which were deemed to be too similar to provide sufficient novel information. Again, we assume that the measure $S()$ is inversely proportional to the Euclidean distance between samples.

that is, the measure $\gamma(v_i)$ is a mixture of the local smoothness (as determined by the heuristic $H()$) around v_i, as well as the distance to the nearest neighbour in the graph.

The vertices in V_i are then sorted according to $\gamma()$ and the top q (that is the q most isolated) points are selected. For these selected vertices, more random samples are added *locally*. This step is shown in Fig. 4–2.

The final step of the adaptive random sampling method is the pruning step. In this step we want to limit the number of vertices in the graph ($||V_i||$) to the user specified N_i. If $||V_i|| > N_i$, this is accomplished by calculating a measure to find vertices which are most similar to others, and can hence be removed:

$$\lambda(v_l) = \sum_{v_m \in V_i - \{v_l\}} e(v_l, v_m) \tag{4.5}$$

The vertices in V_i are sorted according to $\lambda()$ and the top $||V_i|| - N_i$ are removed. This step is shown in Fig. 4–3.

We then iterate over the refinement step and the pruning step until the sampling coverage threshold is satisfied, or a maximum iteration has been

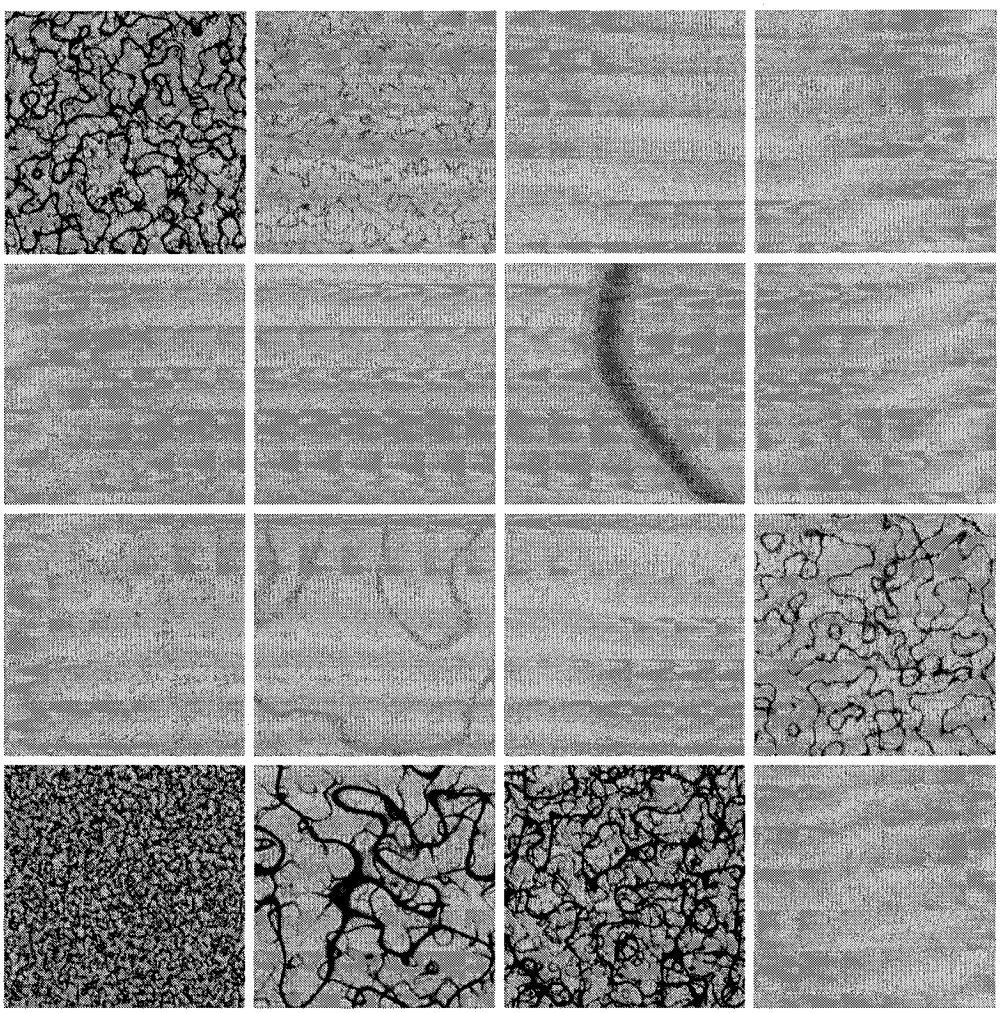


Figure 4–4: An example of uniformly sampling an individual shader. Note how there are many samples which are very similar, and when compared against the sampling of the same shader shown in Fig. 4–5, we also see that much of the texture range of the shader has not been captured.

reached. This adaptive random sampling method is similar to that used by Kavraki *et al.* for a randomised path planner for use with mobile robots [47]. An example of the adaptive random sampling of a particular shader as compared to uniform sampling is shown in Figs. 4–4, and 4–5.

In practise, we actually keep two versions of the graphs for each shader. One is as described above, the other is a highly pruned (and hence much smaller) version of the same graph. Because each vertex is stored as an index into the

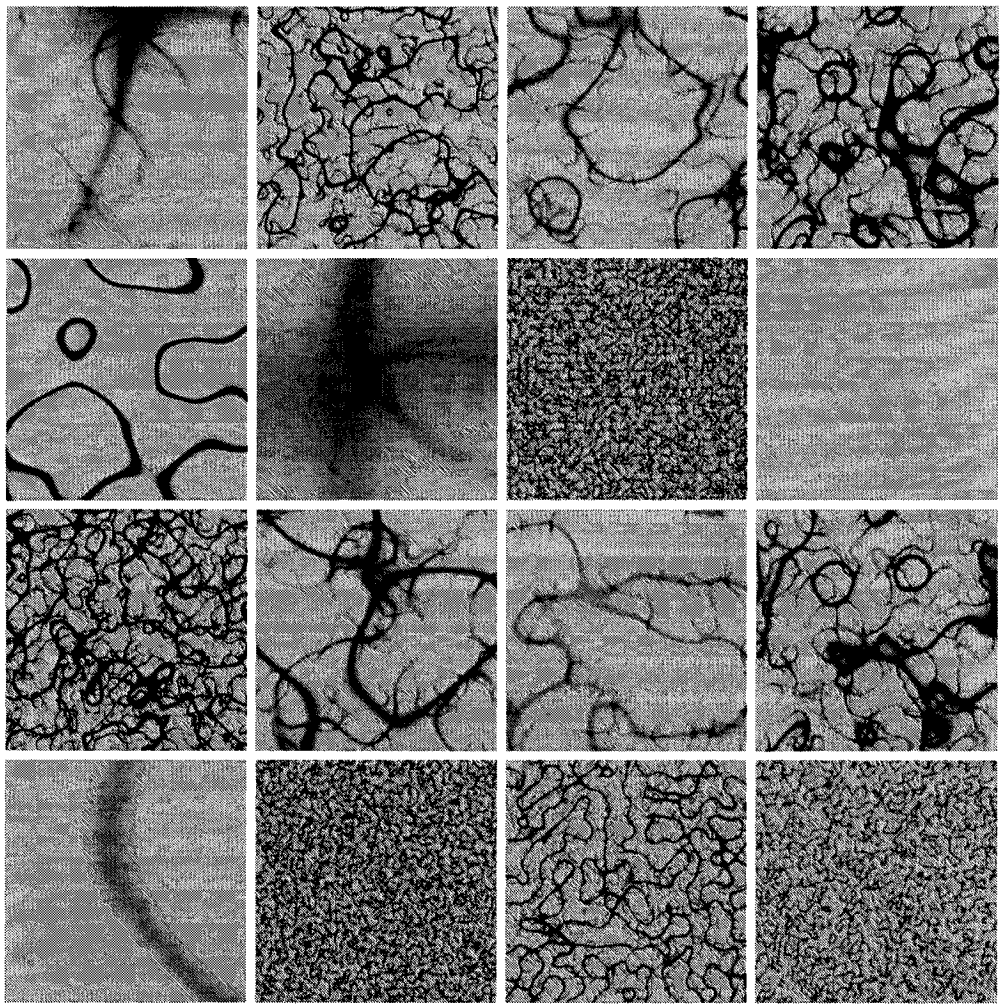


Figure 4–5: An example of the sampling due to our adaptive random sampling technique. When compared against the uniform sampling of the same shader shown in Fig. 4–4, we can see that this method captures much more of the texture range.

image database, the increased storage requirements for keeping more aggressively pruned versions are negligible.

As the first phase of our global search, we seek to identify the shaders which may produce textures similar to our target texture T. This can be accomplished by evaluating the similarity function over the highly pruned catalogue for each of the shaders in the library. For each shader, if the best match from the pruned catalogue is above a given threshold, it is added to the set of candidate shaders $P_C \subset \mathcal{P}$ to be searched globally, otherwise the shader is not searched any further

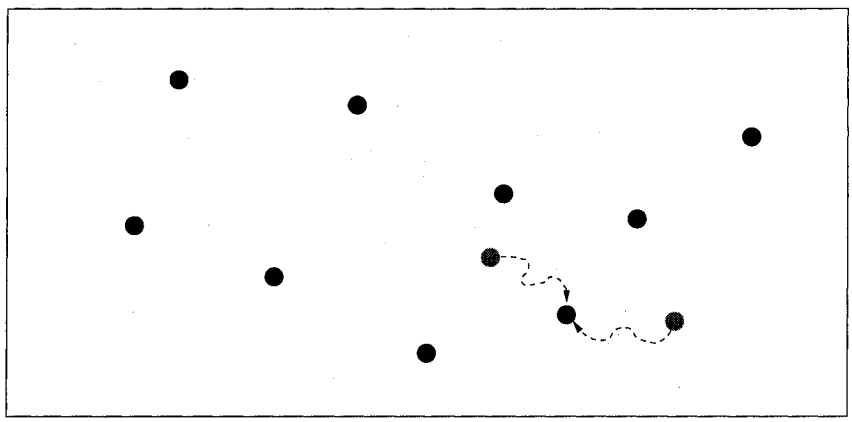


Figure 4–6: The second, local, phase of the search strategy. The points in red were identified during the global search phase as likely candidates to start a local search for the ideal match shown in blue. For the sake of illustration, we have limited the dimensionality of the shader to 2, and again assume that the measure $S()$ is inversely proportional to the Euclidean distance between samples.

for this particular target T. Alternatively, the end user can easily specify which shaders comprise P_C if they have some higher level knowledge of which shaders are likely to produce similar textures.

Once we have a set of candidate shaders P_C which have a potential for producing textures similar to the target, we perform a more exhaustive search – the similarity measure is evaluated for each sample in the sample set for each shader $p_i \in P_C$ to find the best overall match:

$$\max_{V_i \in P_C} \left(\max_{v_l \in V_i} S\left(p_i(v_l), T\right) \right) \tag{4.6}$$

This match then becomes the starting point for the local search phase of our texture matching algorithm which is described below. In practise we look not only at the best match, but at several of the top matches in order to maximise the likelihood of success during the next phase.

4.4 Local Search

The second phase of our search strategy consists of a local search seeded by the best candidate samples from the global search phase. Starting with each

of the top matches from the previous phase, we perform a local optimisation to refine the shader parameter vector values in order to produce a texture which best represents the target texture (see Fig. 4–6). The notion here is that the adaptive random sampling from the global search phase has covered the parameter domain sufficiently to *guide* our search so that we can start our optimisation in the most promising areas of the relevant shader's parameter domain, and avoid wasting time searching in areas where the shader is unlikely to produce desirable matches. More specifically, if we assume that each texture region is interpolated approximately by its bounding vertices (as defined by some kind of subdivision technique), then we can see that searching within an area where none of the bounding vertices is similar to the target texture should, in general, not produce a good match.

It is during this phase that our heuristic measure of the *smoothness* of the local neighbourhood surrounding a vertex, $H(v)$, can be employed to influence the search order of samples to be used as starting points for the local optimisation. As mentioned above in Sec. 4.3, it is possible that multiple points in the parameter domain of a particular shader will generate similar textures, yet the local similarity hyper-surfaces surrounding these points can have different characteristics (see Fig. 4–16 for an example). Since searching in smoother spaces is both more efficient and tends to yield better results, we would like to prioritise our search based on the heuristic measure of the local smoothness.

When the local search is no longer able to take a maximising step, the parameter vector which results in the greatest similarity to the target texture determines the final shader and parameter vector returned by the search algorithm. During the entire search phase, the user is presented with visual feedback of the current best match, allowing them to terminate the search at any point if the

current match is to their liking, thus avoiding searching other candidate shaders, or other promising starting points within the same shader.

For the results presented in this thesis, we have used both simplex optimisation, and a gradient-ascent-based optimisation. Each is described below.

4.4.1 Downhill Simplex Method

Definition 4.4.1 (Simplex)

A simplex, sometimes called a hyper-tetrahedron [20] is the generalisation of a tetrahedral region of space to n dimensions. The boundary of a k-simplex has $k+1$ 0-faces (polytope vertices), $\frac{k(k+1)}{2}$ 1-faces (polytope edges), and $\binom{k+1}{i+1}$ i-faces where $\binom{n}{k}$ is a binomial coefficient [80].

The downhill simplex method is a minimisation technique which does not require the computation of derivatives, but rather only function evaluations [60]. For this method, we are only interested in simplices that enclose a finite inner n-dimensional volume, and are hence non-degenerate. If we fix the origin at one of the $n+1$ vertices, the n-dimensional space is spanned by the vectors defined from that origin to the remaining n points.

The downhill simplex method must be started with an initial simplex which can easily be constructed from a given starting point $\mathbf{x}_0$, by adding multiples of each of the unit vectors $\mathbf{e}_i$:

$$\mathbf{x}_i = \mathbf{x}_0 + \alpha \mathbf{e}_i \tag{4.7}$$

where α can either be a constant, or can be tailored to each dimension based on the search problem characteristics. The method entails the use of several operations: reflection, reflection and expansion, contraction, and multiple contraction (shown in Fig. 4–7).

The downhill simplex method first tries to move the point on the simplex where the value is highest *through* the point on the simplex where the value is

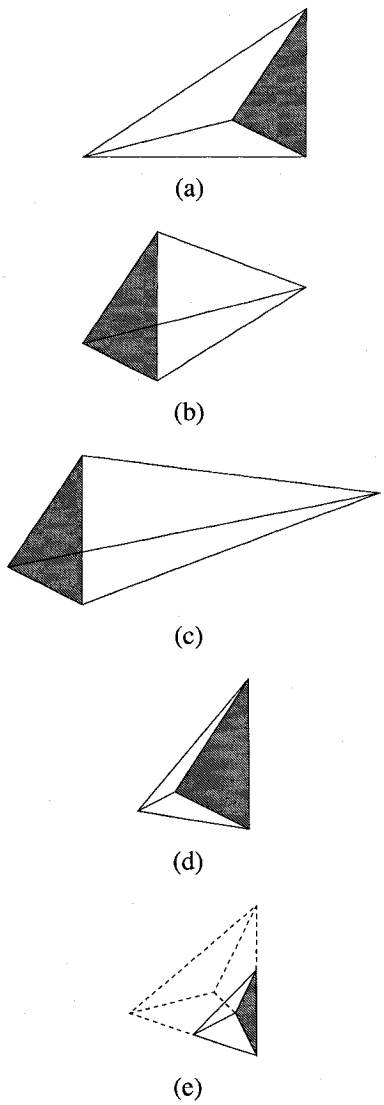


Figure 4–7: Different results after a step in the downhill simplex method. The simplex at the beginning of the step is shown in (a), with the high point on the bottom left of the (tetrahedral) simplex, and the low point on the bottom right. At the end of the step, the simplex can be either (b) due to a reflection away from the high point, (c) due to a reflection and expansion away from the high point, (d) due to a contraction along one dimension from the high point, or (e) due to a contraction along all dimensions towards the low point. This figure is based on Fig. 10.4.1 in [68].

lowest. This volume preserving (and therefore non-degenerate) operation is called a reflection (Fig. 4–7(b)), and is the most common step taken during the optimisation. The notion behind the reflection is that we are attempting to *roll* the simplex down a hill towards the minimum. If possible, the method will attempt to take an even larger step in the direction of the lowest point, in which case it is called a reflection and expansion (Fig. 4–7(c)).

When a reflection results in a point with a higher value, the simplex is instead contracted (Fig. 4–7(d)) so as to facilitate its movement towards the minimum. The idea behind the contraction is to allow the simplex to *squeeze* through a valley on the hyper-surface on its way to the minimum. If this is unsuccessful, the simplex will contract in all directions around the point of minimum value (Fig. 4–7(e)).

The optimisation is stopped when the decrease in the function value is below a given threshold or some maximum number of iterations has been reached. Like many optimisation techniques, downhill simplex optimisation is often restarted from the terminal point in an attempt to narrow in more closely on the function minimum. Since we should already be close to the minimum, multiple restarts generally do not iterate for long before terminating. Note that while we have described downhill simplex optimisation, which finds a minimum, this method can obviously be trivially altered for the purpose of maximising a function instead.

The evaluation of the similarity function $S(p_i(\mathbf{x}_l), T)$ with a new parameter vector $\mathbf{x}'_l$ entails the rendering of a texture sample since $\mathbf{x}'_l$ is not contained in the sample catalogue. Consequently, computing the gradient $\nabla S(p_i(\mathbf{x}_l), T)$ is something we would like to avoid, motivating our desire to use the simplex method when possible as it does not rely on computing derivatives. In particularly unforgiving cases, however, the user can select to use a gradient ascent optimisation method described below.

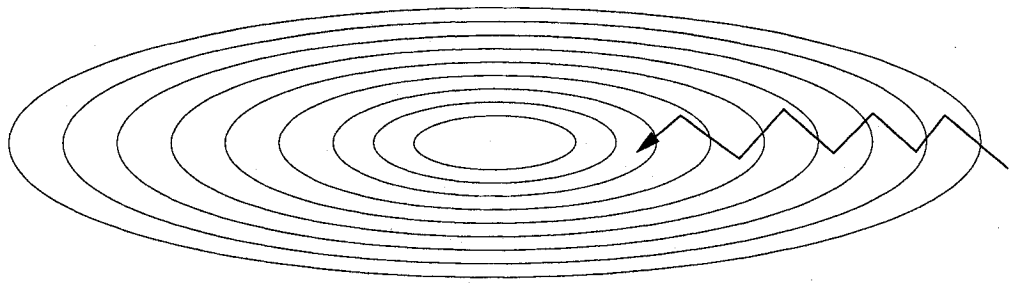


Figure 4–8: Steepest ascent along a narrow crest. Because each change of direction is perpendicular to the last, more steps are taken to reach the maximum than is necessary. This figure is based on Fig. 10.6.1 in [68].

4.4.2 Gradient Ascent Method

The most basic form of gradient ascent is called *steepest ascent.* For this method, we start at the point $\mathbf{x}_0$, and then take steps from $\mathbf{x}_i$ to $\mathbf{x}_{i+1}$ by maximising along the line in the direction of the local uphill gradient $\nabla f(\mathbf{x}_i)$. The main problem with this basic approach is that it is not always best to travel locally in the direction of the gradient. Consider what will happen when trying to ascend a long narrow peak: we would hope that the first line maximisation would take us to the local top of the crest, and that the new line maximisation would then ascend the crest to the true maximum. However, the new gradient at the maximum point of any line maximisation is perpendicular to the direction just traversed (see Fig. 4–8). What is required to minimise the number of steps taken (and hence the number of gradients computed) is to find a direction which is constructed to be *conjugate* to the old gradient, as well as the previous directions taken. These are called *conjugate gradient* optimisation methods, a full discussion of which is beyond the scope of this thesis [67].

4.5 Evaluating Texture Similarity

In order to match a synthetic texture to a target, an important requirement is a distance function to indicate the quality of a candidate match, that is, a texture similarity function that operates on pairs of images. While a naive solution to this

problem might be based on the local pixel intensity differences between images[1] , that would fail to capture the notion of texture fields that *look* the same even when the individual pixels are different. For example, two images of snow falling may have the same apparent textures yet no two pixels may be identical.

Generic image difference metrics are becoming more prevalent with the high demand for content-based retrieval from large image databases. Unfortunately, generic image difference metrics, even when perceptually motivated [61, 59], do not generalise well to the texture domain.

Before delving further into the notion of texture similarity, it is worth considering what we mean by visual texture. Although texture is generally easily recognisable when we see it, it turns out to be very difficult to define as is shown by a sampling of the many definitions found in the literature:

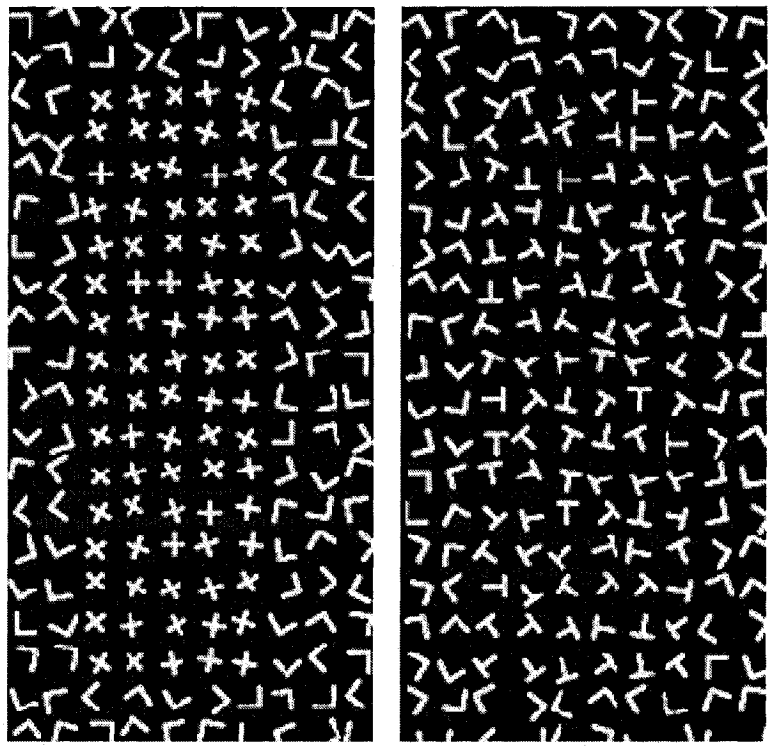
- "We may regard texture as what constitutes a macroscopic region. Its structure is simply attributed to the repetitive patterns in which elements or primitives are arranged according to a placement rule." [75]
- "A region in an image has a constant texture if a set of local statistics or other local properties of the picture function are constant, slowly varying, or approximately periodic." [72]
- "The image texture we consider is non-figurative and cellular... An image texture is described by the number and types of of its (tonal) primitives... A fundamental characteristic of texture: it cannot be analysed without a frame of reference of tonal primitive being stated or implied. For any smooth grey-tone surface, there exists a scale such that when the surface is examined, it

[1] Such as the common method of comparing images using the sum of squared differences (SSD) for the pixel intensity values.

has no texture. Then as resolution increases, it takes on a fine texture and then a coarse texture." [39]

- "Texture is an apparently paradoxical notion. On the one hand, it is commonly used in the early processing of visual information, especially for practical classification purposes. On the other hand, no one has succeeded in producing a commonly accepted definition of texture. The resolution of this paradox, we feel, will depend on a richer, more developed model for early visual information processing, a central aspect of which will be representational systems at many different levels of abstraction. These levels will most probably include actual intensities at the bottom and will progress through edge and orientation descriptors to surface, and perhaps volumetric descriptors. Given these multi-level structures, it seems clear that they should be included in the definition of, and in the computation of, texture descriptors." [85]
- "The notion of texture appears to depend upon three ingredients: (i) some local 'order' is repeated over a region which is large in comparison to the order's size, (ii) the order consists in the nonrandom arrangement of elementary parts, and (iii) the parts are roughly uniform entities having approximately the same dimensions everywhere within the textured region." [41]
- "The character of an object resulting from the arrangement or qualities of its particles or constituent parts." [3]

From the definitions above, it is clear that there is no real consensus on a unified definition of texture. In fact, we can see that some of the definitions are perceptually motivated while others are more application domain specific. The application domains involving some form of textural computation are generally divided into the following four categories:



(a) (b)

Figure 4–9: An example of texture segregation is shown in (a) and two textures which do not segregate are shown in (b). The segregation between the pattern of T's and X's is obvious, whereas the pattern of T's and L's must be examined carefully before the border of the two textures can be identified. This figure originally appeared as Fig. 17.1 in [5, p. 254]. Courtesy of Jim Bergen.

- Texture segmentation - the problem of computing a boundary around the areas consisting of the same texture.
- Texture classification - the problem of identifying the texture classes of the different regions (as normally found using texture segmentation) in a given image.
- Texture synthesis - used mostly in computer graphics to create realistic looking surfaces.
- Shape from texture - one of the general class of vision problems known as *shape from X*. The goal is to use various perspective textural cues to extract the three-dimensional shape information of the textured surfaces.

Psychophysicists are very interested in our ability to separate *figure* from *ground.* Figure-ground separation can be based on many cues, but a classic instance based on textural cues is the tiger-foliage problem: detecting a tiger among the foliage in a forest is a task which carries potentially lethal consequences. The tiger is able to camouflage itself because the observer is unable to discriminate between the two textures, namely the foliage and the tiger's coat. This forms the basis of a prominent open question in the psychophysics community: what are the visual processes which allow a human to separate figure from ground based on textural cues?

An academic example of human texture discrimination can be seen by looking at the two textures shown in Fig. 4–9. There is some disagreement in the community as to what causes us to be able to see the two distinct texture patterns in one of the images, while prohibiting us from seeing that there are two different texture patterns in the other without close inspection. Julesz claims that when the second-order statistics[2] of two textures are similar, they are difficult to segregate [46], while Bergen proposes that when two textures produce a similar response to frequency-selective oriented linear filters they are perceptually similar [4, 3]. Unfortunately, while these theories have merit, counter examples to each theory have been found [45, 55, 54].

Numerous studies have been performed to determine how simple cells in the visual cortex of the macaque monkey respond to various sinusoidal gratings of different frequencies and orientations [69]. These monkeys were chosen because their visual cortex is assumed to be similar to the human brain in its visual

[2] The probability of observing an intensity value at a random location in an image determines its first-order statistics. Second-order statistics are defined as the likelihood of observing a pair of intensity values occurring at the endpoints of a dipole of random length placed in the image at a random location and orientation.

processing, and hence may give us some insight into how our own visual cortex processes texture. The studies concluded that the simple cells are tuned to narrow ranges of frequency and orientation, much along the lines of the claims made by Bergen as described above.

Gurnsey and Fleet applied multidimensional scaling (MDS) to the problem of determining the *texture space* of a set of 20 noise-based texture stimuli [38]. The notion of a texture space is to define a space where similar textures would be near to each other and distinct textures would be far apart. MDS solutions attempt to find a suitable arrangement of objects in an N dimensional space which is most consistent with the measured similarity data. For example, consider an $M \times M$ matrix whose entries (i, j) represent the distance between cities i and j. MDS analysis would yield a most likely arrangement of the objects in a two dimensional space, much like a traditional map. Their experiment consisted of showing triplets of textures from the group of 20 noise based textures (1140 triplets in total) and the subjects had to choose the two textures which were most similar, and the two textures which were least similar. Cumulative similarity scores were stored in the distance matrix as follows: a score of 2 was added each time two textures were deemed similar, 0 for textures deemed distinct, and 1 for the remaining pair in the triplet for a possible distance range of $[0 \ldots 36]$ since each pair was presented a total of 18 times. This experiment was performed on 3 different subjects and it was determined that the subjects' similarity judgements and the MDS solution in 3 dimensions were highly correlated. Although the results from this experiment are promising for this limited set of synthesised textures, this form of texture comparison cannot be computed for generic textures since it is based on processing the data from test subjects in order to find the dimensionality and basis set of the perceptual texture space.

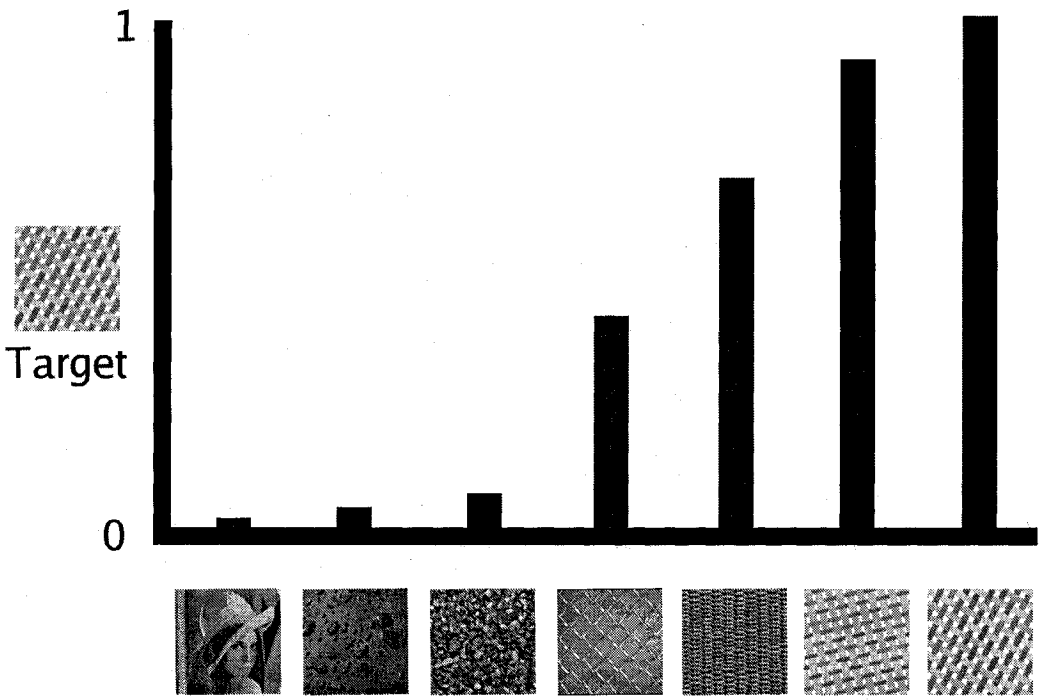


Figure 4–10: An example of the texture similarity measure. The height of the bars indicates the similarity of each texture to the target texture shown on the left. Higher bars imply the texture is more similar to the target.

In order to compare the results of our synthesis process with the target texture, we need a measure of the perceptual similarity of two generic textures, T_1 and T_2:

$$S^*(T_1, T_2) \in (0, 1] \tag{4.8}$$

where a value of 1 indicates that the two textures are indistinguishable, and as values approach 0 the two textures are considered to be increasingly distinct (see Fig. 4–10). As described above, the definition of this ideal measure is still an open problem in the psychophysics community. We therefore need to define computational texture similarity functions $S()$, to approximate our ideal measure S^*.

A common tool used for analysing the second-order statistics in texture images is the co-occurrence matrix [40]. The $G \times G$ grey level co-occurrence

matrix P_d for a given displacement vector $\mathbf{d} = (dx, dy)$, and an image I of size $N \times N$ is defined as follows. The entry (i, j) of P_d is the number of occurrences of the pair of grey levels i and j which are a distance $\mathbf{d}$ apart. More formally:

$$P_d(i,j) = \left|\left\{\left((r,s),(t,v)\right) : I(r,s) = i, I(t,v) = j\right\}\right| \tag{4.9}$$

where $(r, s), (t, v) \in N \times N$, $(t, v) = (r + dx, s + dy)$, and $|.|$ is the set cardinality operator.

Given a co-occurrence matrix, one can compute different texture features such as energy, entropy, contrast, homogeneity, and correlation, however, it can immediately be seen that computing these matrices is a non-trivial matter. For example, consider that there is no easy way to select the displacement vector $\mathbf{d}$, and it is not computationally feasible to construct co-occurrence matrices for all possible values of $\mathbf{d}$.

As previously described, one possible computational model of texture similarity is based on statistical methods particularly in the Fourier domain. While both phase and amplitude play a role in the psychophysics of texture perception [44, 22, 3], the power spectrum alone provides a reasonable approximation to perceptual performance and is computationally expedient. A key observation is that the windowed power spectrum of a texture can be used for distinguishing or segregating textures. The power spectrum describes the mixture of spatial frequencies in an image and it can be obtained readily using a Fourier transform (see Fig. 4–11). As such, one of our computational approximations to the ideal texture similarity function $S^*()$ is:

$$S_F(T_1, T_2) = 1 - ||F_{ps}(T_1) - F_{ps}(T_2)||_2 \tag{4.10}$$

where $F_{ps}(T)$ is the power spectrum of the texture T, computed by using a fast Fourier transform (FFT). Because the power spectrum represents frequency

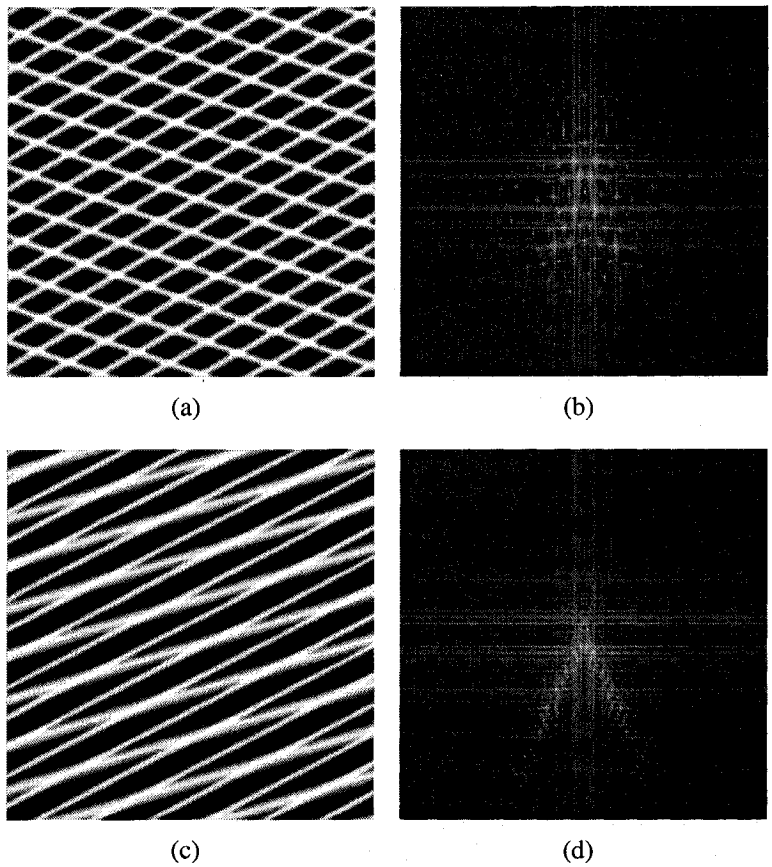


Figure 4–11: An example of the power spectrum for two different images. The images on the left are the originals, and the images on the right are the power spectrum images. Note how the frequency related elements are localised in the power spectrum images.

information as a function of the (inverse) distance from the centre of the image, in practise the differences are weighted radially to favour low frequency (structural) components.

In addition to the power spectrum of the texture sample, we have also considered the use of the histogram of the energy distribution in a Laplacian pyramidal representation of the texture images, as used by several authors for texture analysis and synthesis [1, 42, 27].

An image pyramid is an image representation consisting of multiple copies of the image at various resolutions (see Fig. 4–12). One common type of pyramid is the low-pass pyramid. A low-pass pyramid consists of a full resolution image

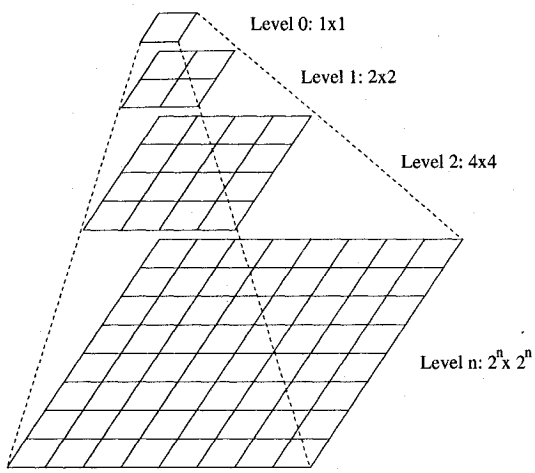


Figure 4–12: An image pyramid has a single pixel at its top level. Each lower level consists of an image at twice the resolution of the previous level. The type of pyramid specifies how the individual pixel values are determined for each level.

at the lowest level, followed by a half resolution image, etc., where each level is formed by an averaging process. The top of the pyramid consists of a single pixel image which is the average intensity of the entire image. In this type of pyramid, each level is independent and can thus be used on its own. Common uses of low-pass pyramids include mip-mapping (Sec. 2.2.3), and image communication where the appropriate level of the pyramid is transmitted according to the available bandwidth.

A Laplacian pyramid is a band-pass pyramid whose top level is again a single pixel which is the average of the entire image. However, the other levels of the pyramid are not independent as in a low-pass pyramid. Each of the other levels consists of the *detail* information necessary to generate an image at the required resolution for level n from the previous resolution, level $n-1$. For example, to reconstruct an image of resolution 4×4, we would first construct an image of resolution 2×2 by assigning the average value from the 1×1 resolution image (top level of the pyramid) to each pixel in a 2×2 resolution image, and then add the detail image from the second level of the pyramid which has a resolution of 2×2. We would then fill a 4×4 resolution image with the corresponding

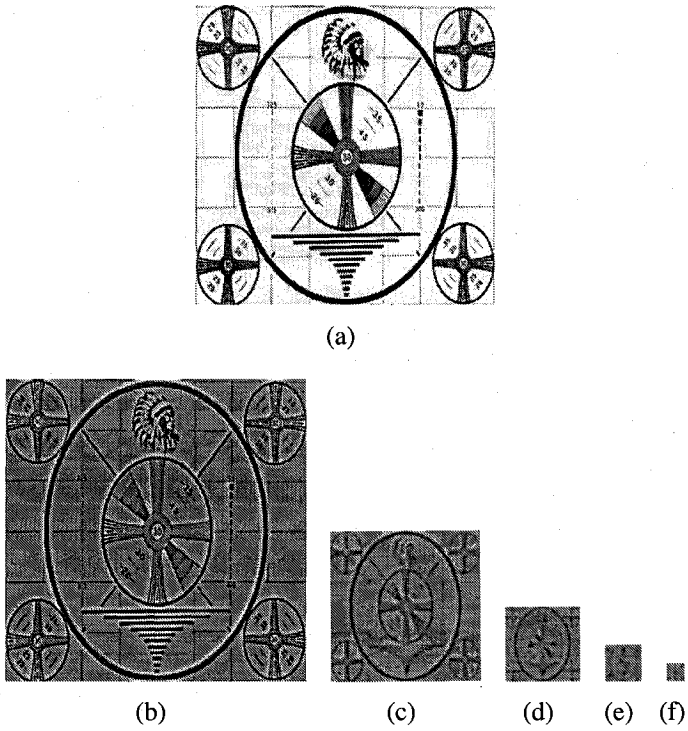


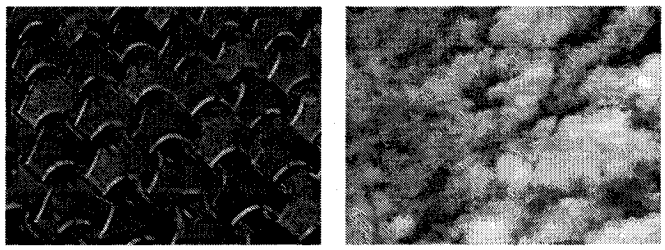
Figure 4–13: An example of a Laplacian image pyramid. The original image is shown in (a), and the *detail* images for the bottom 5 levels are shown in (b) - (f). (The sample image is the "Indian Head" test pattern, which was originated by RCA in 1939. It was non-uniformly scaled to go from the 4:3 NTSC aspect ratio to make it a square image for the purpose of generating the image pyramid.)

average values from the 2×2 image constructed thus far and then finally add the detail from the third level of the pyramid (4×4) to arrive at the desired 4×4 image. In other words, the band-pass pyramid only stores the information at each level required to go from a coarse level n to a finer level $n + 1$. An example of a Laplacian image pyramid can be seen in Fig. 4–13.

Pairs of textures can be compared by computing the histogram difference between the corresponding levels in their Laplacian pyramids:

$$S_L(T_1, T_2) = \sum_i w_i \left| h\left(L_i(T_1)\right) - h\left(L_i(T_2)\right) \right| \tag{4.11}$$

where $h()$ is the histogram of an image, L_i is level i in the Laplacian pyramid of a texture, T_1 and T_2 are the textures being compared, and w_i is a weighting factor.



(a) (b)

Figure 4–14: An example of a deterministic texture (a), and a stochastic texture (b).

The motivation behind comparing the intensity histograms at each level of the Laplacian pyramid stems from the fact that the comparison is computed on a representation that has intrinsic *spatial* structure. In other words, a *spatially* correlated change in the reconstructed image is effected when a pyramid level is modified locally. That is, the first order statistics of the pyramid levels do measure some of the spatial statistics of the original image [42].

In practise, we find that both the Laplacian histograms and the Fourier power spectrum have advantages as the basis for texture similarity functions and allow the user to select the desired comparison function. Because our method is not tied to any particular choice for $S()$, if better computational approximations to texture similarity are discovered, they can easily be incorporated into our system. In the context where the particular choice of texture metric is not significant, we refer to it simply as $S()$.

4.6 Examples

Textures are often divided into two classes, namely, stochastic and deterministic. Stochastic textures generally do not contain any easily identifiable primitives, whereas deterministic textures largely consist of well-defined primitives combined with a set of rules governing their placement (for an example of each see Fig. 4–14). In practise, many textures exhibit some combination of

properties from both classes. Prior work in the field of texture synthesis tends to focus on only one of these texture classes, the predominant methods being based on Markov random fields which assume that the desired texture targets are stationary[3] , local[4] , stochastic textures. The deterministic texture synthesis methods attempt to measure domain specific attributes, and therefore can not be used to synthesise stochastic textures.

In order to demonstrate our method of procedural texture matching, we have chosen many kinds of texture targets. Some of the texture targets are natural, that is, they are from photographs taken in the real world and some are synthetic (rendered). For both of these classes, we show our results on a mixture of deterministic and stochastic textures. As a first example, consider the texture matches shown in Fig. 4–15. These are deterministic synthetic textures which were themselves generated procedurally, and hence have well known parameter values which permits us to quantitatively measure our solution. For both examples, as expected, the exact parameter vector was recovered which was used to render the original texture.

In order to increase the difficulty of the texture match, we performed texture matching experiments on synthetic textures which are stochastic. Example texture matches for a few such textures are shown in Fig. 4–16. These test cases provided an interesting result: although the texture matches are good, the parameter vectors recovered were not the same as those used to render the initial targets. This fact

[3] A *stationary* texture is a texture where samples from various regions will look similar, that is, the local statistics are position invariant.

[4] Each pixel in a texture which exhibits *locality* can be characterised by its local neighbourhood.

is not surprising, especially for stochastic textures, since there are often many different parameter vectors which can produce similar results.

We have also performed our texture matching on texture targets from the Brodatz album [18]. These textures are commonly used as a reference point for texture matching and classification algorithms in the perception community, and are therefore well suited to exemplify our procedural texture matching framework. The first set of examples are shown in Fig. 4–17. These are deterministic textures from photographs of real world phenomena. The brick texture match is very satisfying because the dominant elements present in the target texture such as the height and width of most of the bricks and mortar thickness have been replicated very closely. The weave pattern shown in the same figure is not as close a match, but again the dominant elements (in this case the frequency of the vertical and horizontal lines) have still been replicated quite well. It is worth noting that even though both of the texture samples are quite distinct, the actual shader used to replicate each was the same, showing that this method allows us to find novel uses for existing shaders.

We also performed some experiments using stochastic textures from the Brodatz album as the targets. As noted above, these textures are photographs of real world phenomena. The matching results are shown in Fig. 4–18 with the target texture on the left, and the matched texture on the right. Both cases exhibit successful results, which is especially satisfying given the limited shader library which was used.

The notion of using a shader serendipitously is definitely a positive side-effect of our system, as a graphic artist may not consider some of the shaders which could actually produce positive matches to the given target texture. An example of this is shown in the second match which made use of a shader intended to texture an eyeball. This shader has parameters such as `irissize`,

`veinfreq`, `bloodshot`, etc., which were appropriately specified (removing the iris completely by setting its size to 0) to match the given texture which was in no way related to an eyeball.

In Fig. 4–19, we demonstrate our matching results using two real pictures of the sky – one during the day, and one at night. While, of course, in neither case is an exact match found, the textural characteristics are very similar for both examples. We were unable to produce closer matches by manually tuning the parameters.

In another example shown in Fig. 4–20, we have extracted a texture from an architect's sketch of a house. Again, the match found demonstrates a satisfactory result. Overall, we have shown matches to a large variety of real and synthetic texture targets. These positive results demonstrate the generality of our matching framework.

While all of the examples above show matches which are perceptually very similar to their targets, we have provided an example of a few textures (Fig. 4–21) for which less suitable matches were found. It will not always be the case that we can find a close match if the target texture is not contained in the texture ranges of the procedural textures in the library. In this situation, we can only hope to find a procedural texture p and a parameter vector $\mathbf{x}$ such that $p(\mathbf{x})$ is as similar as possible to the target texture T. This inability of the shader library's texture range to express a given target texture is the case with the top example (Fig. 4–21(b)). It is notable however, that the match which was found is still reasonable given the limited texture range of the shader library. We can see that similar textural elements are present in the match found, and the frequency of these elements seems to be appropriate. The bottom example (Fig. 4–21(d)) fails for a more interesting reason. In this case, the failure to produce a good match is due to the fact that one of the parameters for the shader determines the number of points in

the star which is intended to be an integer value. For this shader, the sampling phase did not produce the desired value of 4 but rather produced floating point values close to 4. This shader gives degenerate results when given non-integer parameters which means that the energy space between integer values is highly non-convex, thus preventing our algorithm from finding the exact result.

Our system is able to detect failed matches when there is a large residual in the similarity measure between the best match and the texture target. Potential approaches for handling this situation are discussed in chapter 6.

The shading language programs as well as the actual recovered parameters for the examples shown in Figs. 4–18(b) and 4–18(d) are presented in appendices B and C respectively. As an illustration of the compactness of the procedural representation of each, the shaders for these two examples are 1.5 and 2.3 kilobytes in size, whereas an uncompressed 8 bit 256×256 texture map occupies 192 kilobytes, giving a size ratio of approximately $105 : 1$ for these small texture samples. Of course, procedural textures can be rendered at arbitrary resolutions, while an equivalent high-resolution texture map can easily occupy a few megabytes leading to size ratios of $2000 : 1$ and beyond.

All of the results shown in this chapter were obtained using a small collection (approx. 100) of publicly available general purpose shaders, none of which were specifically written to replicate the appearance of any of the given natural target textures. The average match time was 12 minutes, with under 100 iterations for all cases.

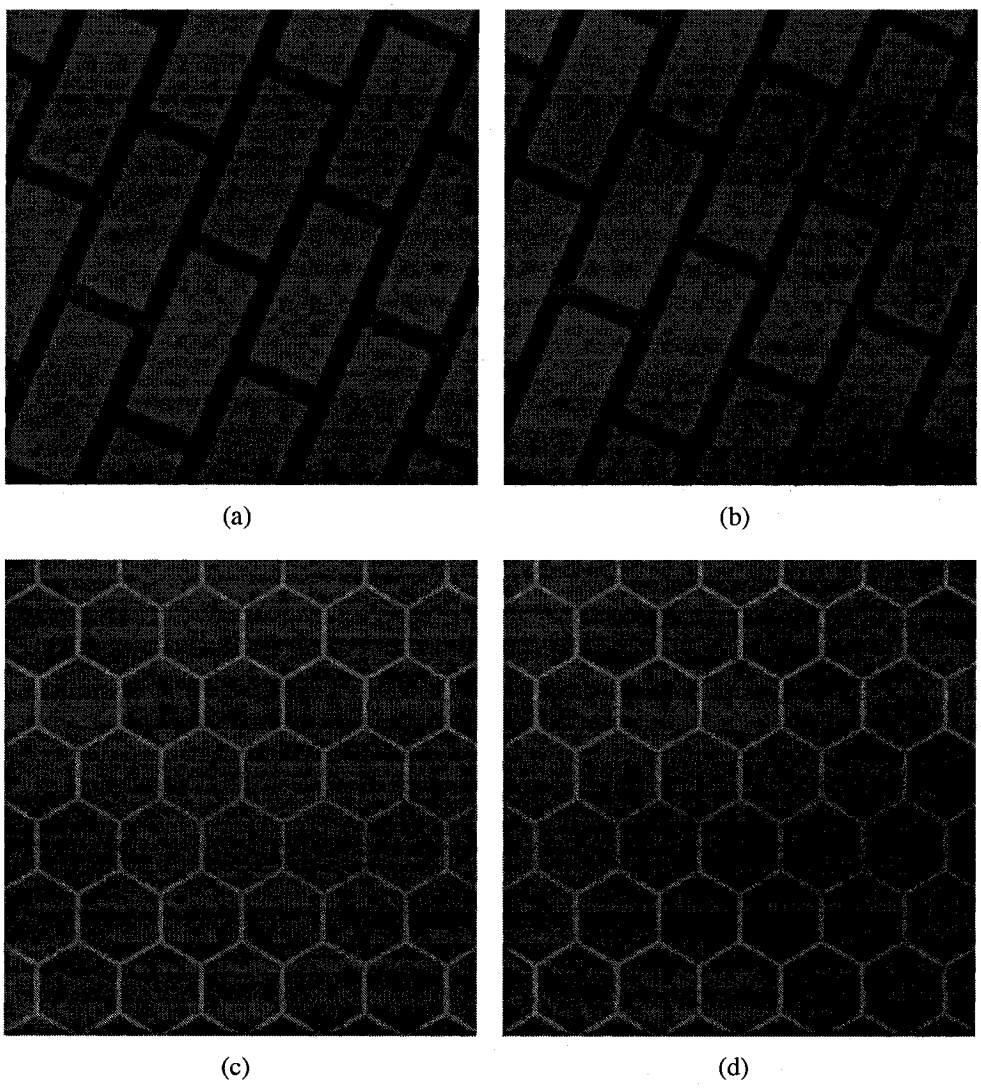


Figure 4–15: Examples of procedural texture matching using deterministic synthetic textures as targets. The target textures are shown on the left, and the procedural texture matches are shown on the right. The use of procedurally generated textures as targets should guarantee a good match since we know beforehand that the desired texture lies within the texture range of the shaders being searched. An example of the texture range of the shader used in (b) is shown in Fig. 2–10. In both cases, as expected, the exact parameters used to produce the target were found for the match.

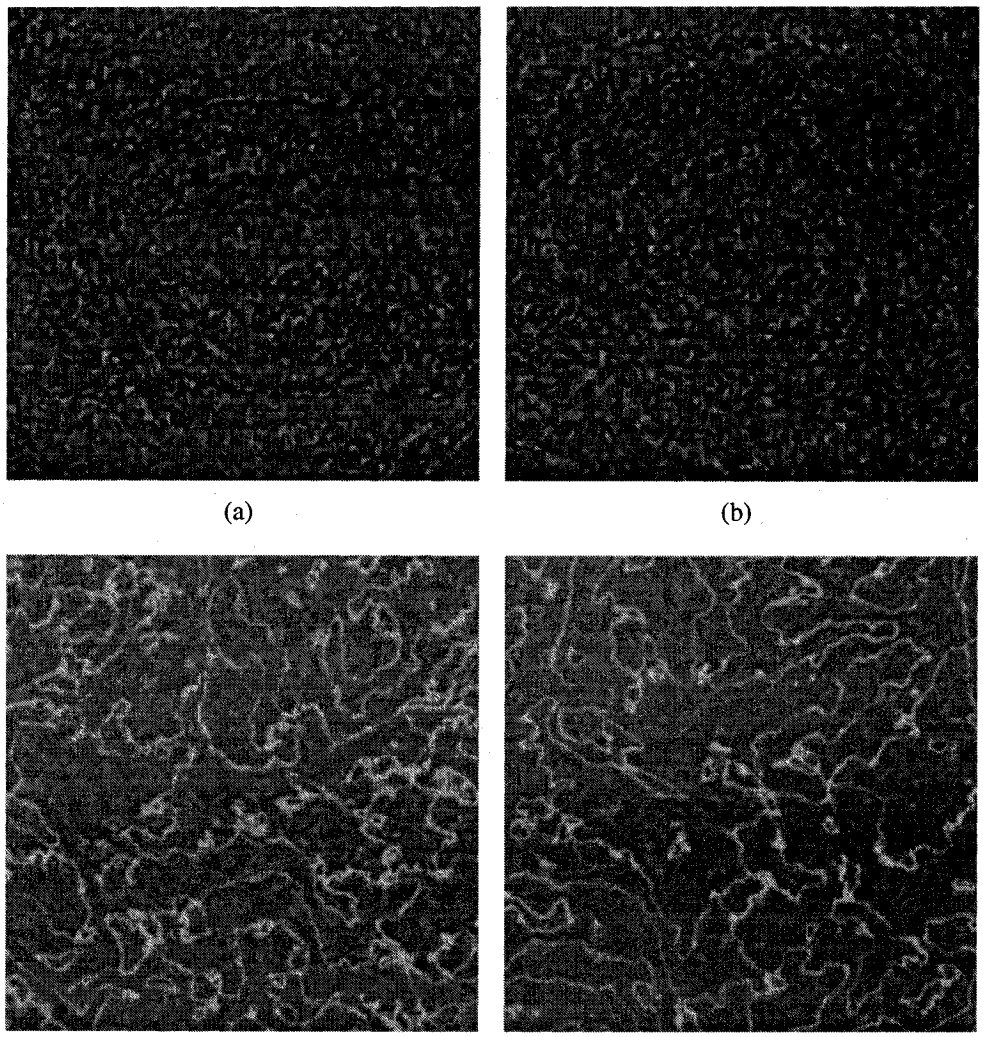


Figure 4–16: Examples of procedural texture matching using stochastic synthetic textures as targets. The target textures are shown on the left, and the procedural texture matches are shown on the right. The use of procedurally generated textures as targets should guarantee a good match since we know beforehand that the desired texture lies within the texture range of the shaders being searched. These examples show close matches, but with entirely different parameter values showing that different points in the parameter domain can map to very similar textures, particularly with stochastic textures.

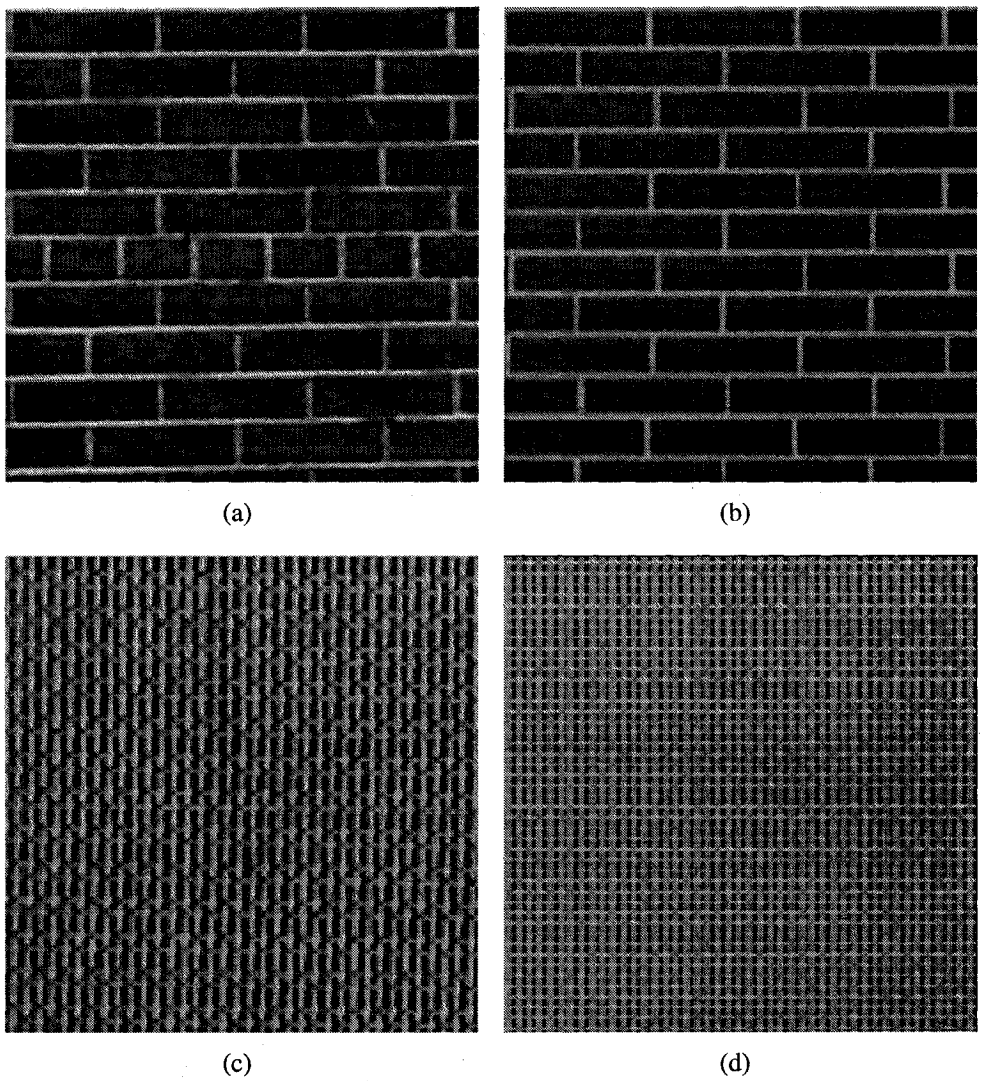


Figure 4–17: Examples of procedural texture matching using deterministic Brodatz textures as targets. The images on the left are the texture targets, and the images on the right are procedurally generated using the automatically recovered shader and parameters.

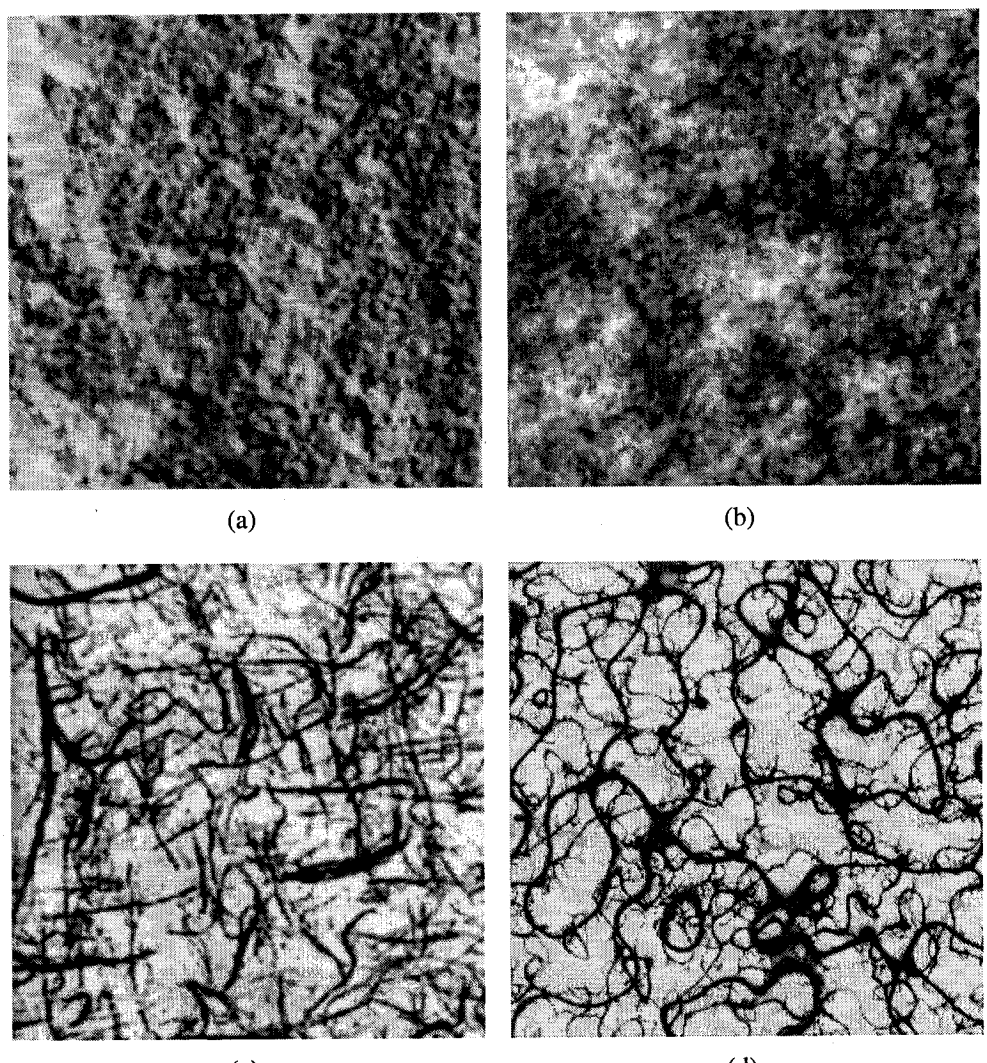


Figure 4–18: Examples of procedural texture matching using stochastic Brodatz textures as targets, and a small repertoire of shaders. The images on the left are the texture targets, and the images on the right are procedurally generated using the automatically recovered shader and parameters. The texture range of the shader used for the match to (c) is illustrated in Fig. 4–5.

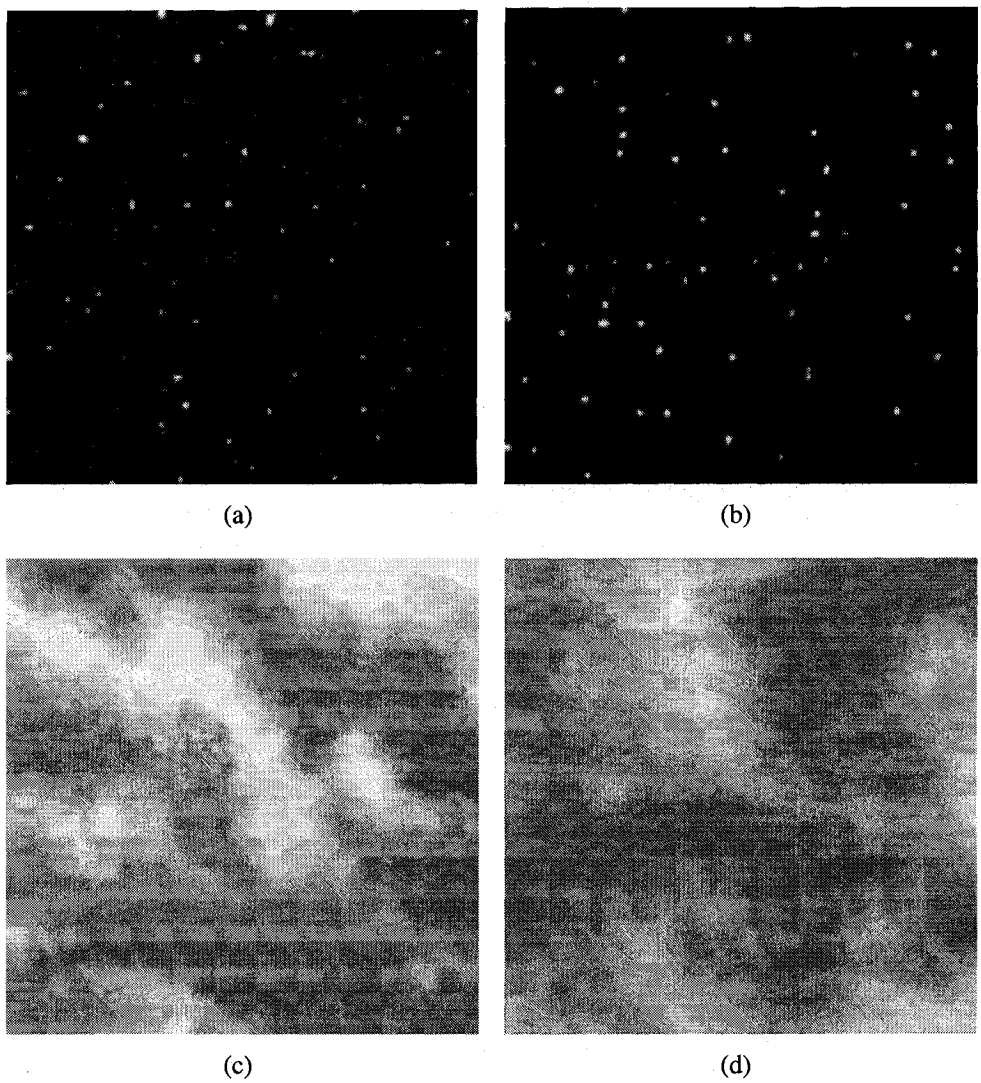


Figure 4–19: Examples of matching real textures from the sky during the day and at night. Again, the target texture is shown on the left, and the texture match is shown on the right. Note that although the image matches are not exact, the textural characteristics are very similar.

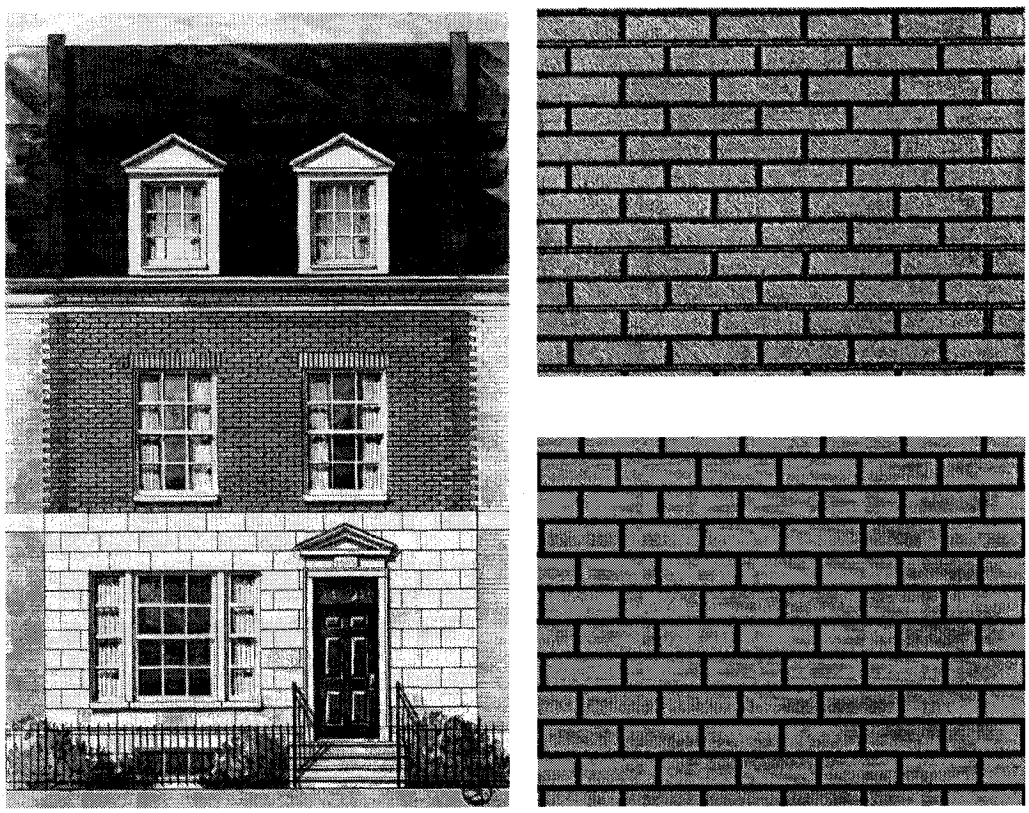


Figure 4–20: An example of texture matching from a sketch. On the left is an architect's sketch of a house, and a zoomed view of the brick texture is shown on the top right. The texture found using our texture matching technique is shown on the bottom right.

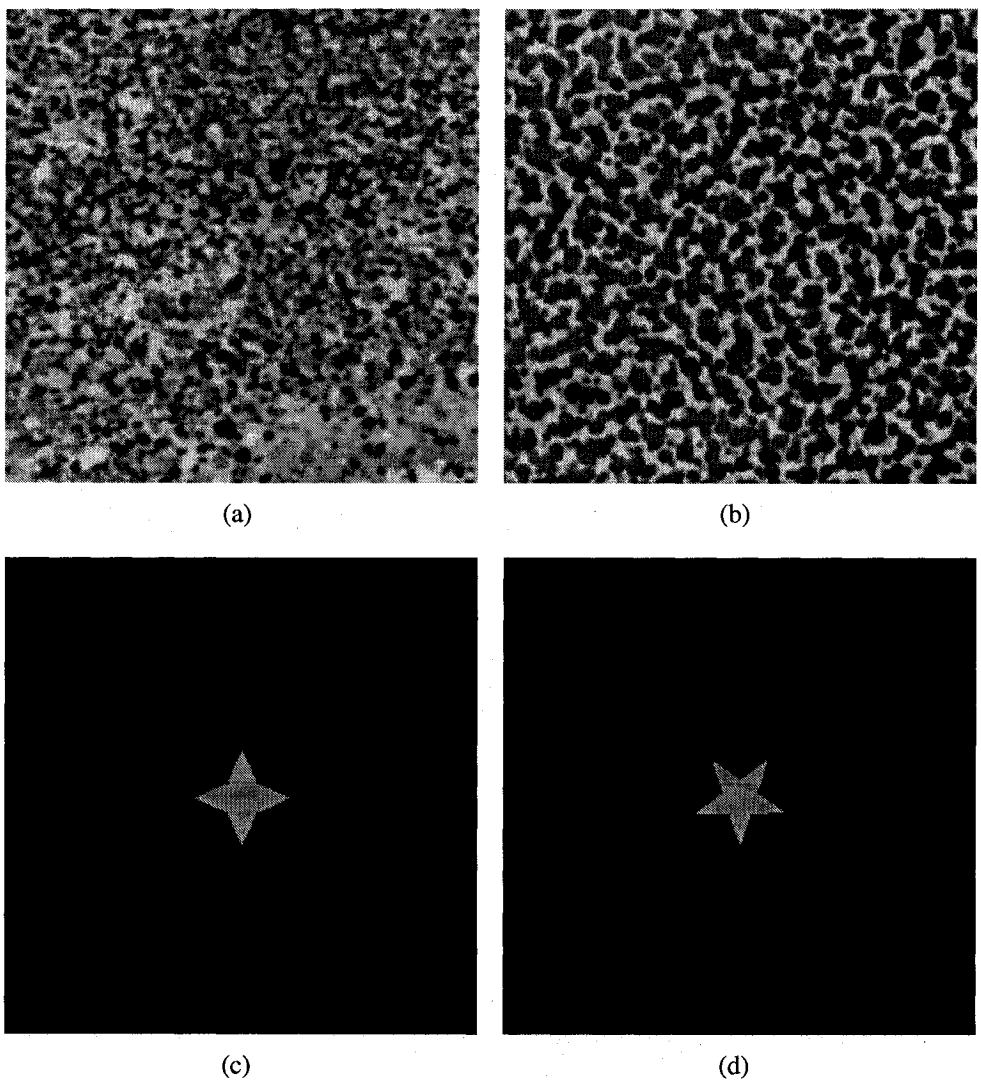


Figure 4–21: Examples of failed matches for both real and synthetic target textures. In (b), the shader library did not contain a shader capable of producing a texture with high similarity to the target (a). The texture shown in (d) produces a failed match to (c) even though it failed for an interesting reason: this particular shader takes only integer valued parameters for the number of points on the star. Non-integer values produced degenerate images (see Fig. 4–22), and hence our local search phase was not started in a reasonable location since the only integer valued sample for the relevant parameter was 5. This is discussed further in Sec. 4.6.

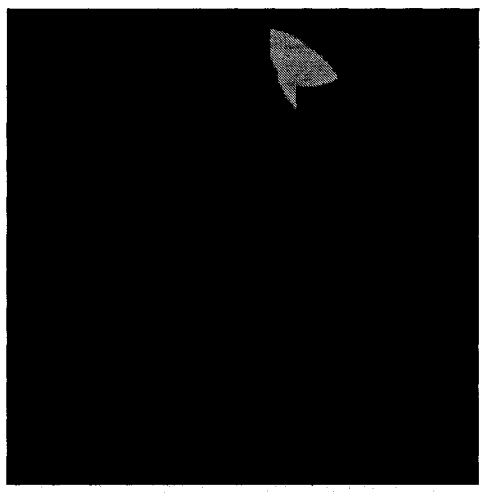


Figure 4–22: An example of a degenerate texture for the star shader discussed in Fig. 4–21.

CHAPTER 5

Procedural Texture Transformation

UNTIL now, we have been thinking of textures as static entities, that is, that textures do not change during the lifetime of an object. For example, a fixed image can be texture mapped onto an object, or a shader can be used with fixed parameters to render the surface of an object. In reality, however, many textures change naturally over time. These changes can occur very slowly, such as is the case with a soda can fading in the sun, a fruit growing mold, pavement cracking, etc., or can be observed to occur more quickly, as in the case of an apple oxidising shortly after one takes a bite.

The field of computer graphics deals not only with the synthesis of static images, but also with *animations*. Animations are image sequences in which the objects, camera, and light sources interact in a certain way in order to convey a particular story. These animations may have a high frame rate[1] , as is the case with motion pictures, or a lower frame rate common to cartoon-like animations.

[1] The frame rate of an animation is the number of images which are shown per second.

Since it is possible for the textural characteristics of many real world objects to change over time, we are interested in how textured objects in an animation could also be made to change automatically. We would like this automatic *texture transformation* to be based on different texture samples in a time-dependant manner.

This notion is a natural extension to the procedural texture matching technique discussed in chapter 4. Because the appearance of each procedural texture is dependant on a set of parameters, it should be possible to vary those parameters in order to achieve a suitable transformation.

We seek a technique which allows the end user to specify the starting and ending textures of the transformation just as in the static case described previously. In addition, the user should be able to specify optional *key-textures*. We define key-textures analogously to *key-frames*: in the field of hand drawn animation, key-frames were the frames drawn by the senior animator, while the apprentice would fill in all the frames in between. In our application, key-textures are specified by the end-user, and our texture transformation technique fills in the remaining frames. These key-textures should be able to be placed arbitrarily between the starting and ending textures.

The main goal of our procedural texture transformation technique is to vary a shader's parameters over time so that the perceived difference between adjacent texture samples is minimised. A framework for creating these kinds of smooth transitions in a controlled environment is desirable for graphical animators. An illustration our user interface for this task can be found in appendix A.

5.1 Approach

The initial step of our texture transformation algorithm consists of identifying the shaders (and parameters) needed to produce textures which are similar to the

texture samples given for the endpoints of the transformation. These are typically recovered from real or synthetic images using the texture matching approach described in chapter 4, although they can also be specified manually.

The following temporal framework is desired: given a series of input textures $\mathcal{T} = \{T_1, \ldots, T_n\}$, we wish to find a corresponding set of procedural shaders and their associated parameter vectors, $\mathcal{P}_\mathcal{T} = \{p_1(\mathbf{x}_1), \ldots, p_n(\mathbf{x}_n)\}$ where $p_1(\mathbf{x}_1) \approx T_1, \ldots, p_n(\mathbf{x}_n) \approx T_n$. Using these shaders $p_i(\mathbf{x}_i)$, we want to produce a continuously changing texture $C(t),\ t \in [0,1]$ such that $C(0) = p_1(\mathbf{x}_1)$, and $C(1) = p_n(\mathbf{x}_n)$. In addition, the remaining texture targets $T_i \in \mathcal{T} - \{T_1, T_n\}$ should be used (in order) as key-textures. Finally, we want the transformation to be smooth, that is, the adjacent frames should be as similar as possible, so we therefore want to maximise $S\left(C(t), C(t+\Delta t)\right)\ \forall\, t \in [0,1)$.

Transformation between two texture samples can be divided into three different cases:

1. Transformation between samples within the same shader.
2. Transformation between samples from two different shaders.
3. Transformation between samples from two different shaders via other *connective* shaders.

These will each be discussed separately below.

5.2 Transformation Within a Shader

In order to transform the appearance of a texture resulting from two different parameter vectors within the same shader, we again use a two-stage algorithm with some pre-computed data. Recall from chapter 4 that a fully connected graph is constructed during the creation of a catalogue of samples when a new shader is added to the library. In this graph, the vertices correspond to the samples from the

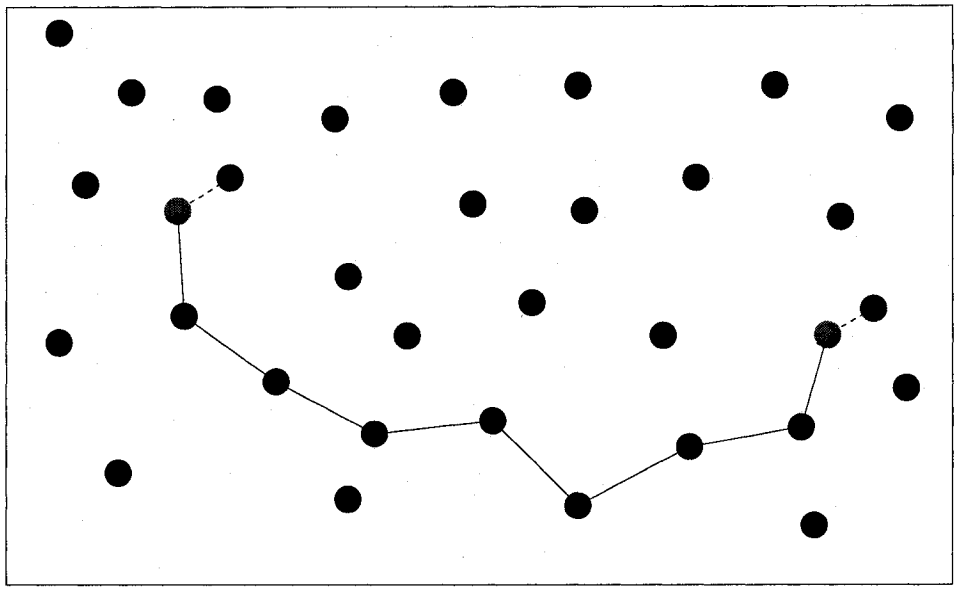


Figure 5–1: Shortest path calculated using the graph from the shader catalogue. The blue vertices represent the desired starting and ending parameter vectors. First the closest points in the graph (shown in red) are identified, and then a single source shortest paths algorithm finds the shortest path between these two points. This path is then refined as described below.

catalogue, and the the similarity measure between those samples determines the edge weights.

In the first stage of the transformation, we find the closest sample from the catalogue for the end-point parameter vectors as described in Sec. 4.3 above. We then compute a path between these two selected vertices using Dijkstra's algorithm as described below. This process is shown in Fig. 5–1.

Dijkstra's algorithm is a single source shortest paths graph algorithm [28]. That is, an algorithm for finding the shortest path from a particular vertex in a graph to all other vertices in the graph. This algorithm has the precondition that all of the edge weights must be non-negative: $e(v_i, v_j) \geq 0 \, \forall \, \{v_i, v_j\} \in V$. For the remainder of this section, we will "invert" our definition of the similarity measure for the sake of tradition and clarity and instead think of it as a distance: smaller values imply that two textures are more similar (closer together), and larger values indicate that textures are more distinct (further apart).

Algorithm 3 Dijkstra's Shortest Path Algorithm, based on the description in [25].

```
for each vertex v ∈ V do
    d[v] ← ∞
    π[v] = nil
end for
d[s] ← 0
S ← ∅
Q ← V
while Q ≠ ∅ do
    v_i ← Extract-Min(Q)
    S ← S ∪ {v_i}
    for each vertex v_j adjacent to v_i do
        if d[j] > d[i] + e(v_i, v_j) then
            d[j] ← d[i] + e(v_i, v_j)
            π[j] ← i
        end if
    end for
end while
```

Dijkstra's algorithm augments the graph structure with two extra values: the distance from the current vertex to the source ($d[]$), and the preceding vertex in the path from the source ($\pi[]$). The graph constructed by following the preceding vertices all the way back to the source (from all vertices) is known as the predecessor subgraph which at the termination of the algorithm is in fact a spanning tree.

As shown in Alg. 3, the algorithm first initialises all of the path distance values to infinity, the predecessor subgraph values to *nil* and the distance to the source s is set to 0. Then all of the vertices are added to a priority queue Q which is keyed by their $d[]$ values. The algorithm then iterates over all of the vertices by removing the vertex i with the lowest distance to the source, and adding it to the set S which contains all of the vertices for which the shortest path has already been determined. The algorithm then proceeds to *relax* all of the edges connected to i (maintained in an adjacency representation). The relaxation step compares the current distance estimate $d[j]$ for each vertex v_j adjacent to v_i to see if it can be

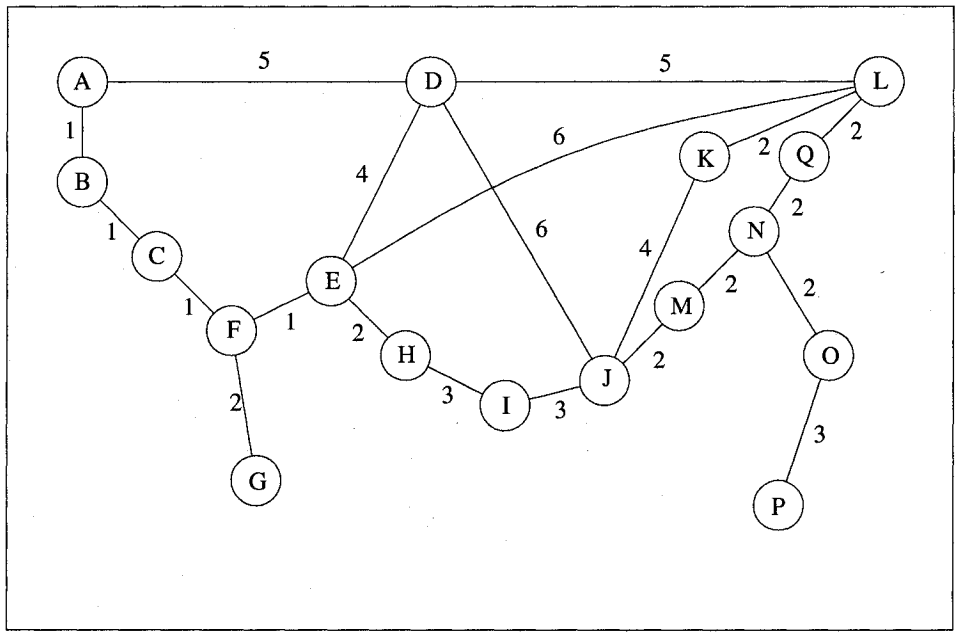


Figure 5–2: An illustration of our path cost function, $\sqrt[n]{e_1^n + \ldots + e_m^n}$, which encourages longer, smoother paths for increasing values of n. For $n = 1$, the chosen path is $\{A, D, L\}$, $n = 2$ gives $\{A, B, C, F, E, L\}$, and $n = 3$ gives the ideal $\{A, B, C, F, E, H, I, J, M, N, Q, L\}$. These paths have edge weight averages of 5, 2, and 1.8 respectively.

improved by connecting directly to v_i. If so, the distance value and predecessor subgraph are updated for v_j to reflect the improved values.

Upon termination, the shortest path from s to any vertex v_i can be determined by following the path backwards in the predecessor subgraph all the way to the source s.

The path cost described so far is dependent on the edge weights, which, as we saw in chapter 4, are simply the similarity measures between the two textures corresponding to the vertices. However, using this path cost is perhaps not appropriate for the properties we seek for our transformation, namely that the transformation should be as smooth as possible. We have chosen to use a path cost function of $\sqrt[n]{e_1^n + \ldots + e_m^n}$ where e_i is the distance measure of the i^{th} edge of the path. This cost function provides finer control over the kinds of paths which are found. Consider the example graph shown in Fig. 5–2. We wish to determine a

path between the starting vertex A and the ending vertex L. If we set $n = 1$, our cost function is $\sqrt[1]{e_1^1 + \ldots + e_m^1}$ or $\sum_i e_i$. This standard path cost returns a path of $\{A, D, L\}$. This path has an average edge weight of 5, as well as a maximum edge weight of 5. A cursory inspection of the graph suggests we could find a *smoother* path, i.e. a path which does not contain any large steps between vertices, even though it will be longer. For $n = 2$, we get a path of $\{A, B, C, F, E, L\}$ which is slightly better: the average edge weight has dropped to 2, however, the largest edge has a weight of 6. For $n = 3$, the path is ideal for our purposes: $\{A, B, C, F, E, H, I, J, M, N, Q, L\}$. This path has an average edge weight of 1.8, and a maximum edge weight of 3. As we can see, for increasing values of n our path cost function discourages large steps and thus promotes longer, smoother paths.

All that remains to be shown is that our cost function can be computed incrementally, and can thus be used in Dijkstra's algorithm. Without loss of generality, we can use the standard cost function of $\sum_i e_i$ and simply raise the cost of each edge to the power of n. Alternatively, instead of computing $d[j] = d[i] + e(v_i, v_j)$, $d[j]$ can be computed as

$$d[j] = \sqrt[n]{d[i]^n + e(v_i, v_j)^n}. \tag{5.1}$$

Although the path through the graph determined above tries to take a series of *small* steps, the selected path most likely will not exhibit the smoothness we are trying to achieve. To refine the path, the second stage of our transformation algorithm employs an adaptive linear subdivision technique[2] . We want to locate the regions in the path where the inter-frame disparity is too high, and insert

[2] This is also a standard refinement strategy for path planning in the field of mobile robotics [50].

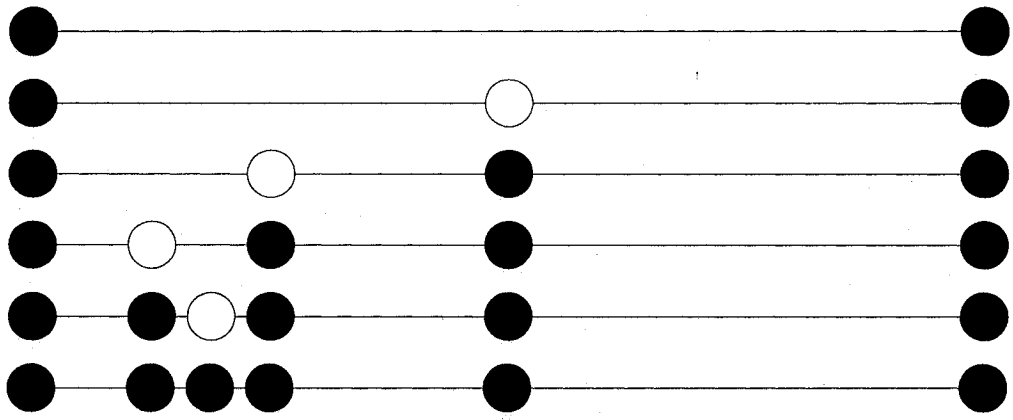


Figure 5–3: An example of the adaptive linear subdivision technique. The sequence of subdivisions occurs from top to bottom, with the white vertices representing the midpoints which were added. This technique ensures that the shader is sampled more densely where it is changing, and less densely where it is more static.

more samples in those regions. To accomplish this, we use the following rule: while the similarity measure between adjacent samples is less than a given threshold, another sample is inserted at the linear midpoint between the two samples in the shader's parameter space. This recursive solution has the effect of sampling more densely where the underlying parameter changes affect more change in the appearance of the shader. In general, this method assures that no two adjacent samples will have a large perceptual disparity. An example of the linear subdivision is shown in Fig. 5–3.

There is, of course, the possibility that with a particularly uncooperative shader, repeated bisection of the parameter values will not result in adjacent samples falling below the perceptual threshold. While this has not been the case with the shaders we have tested thus far, we describe a possible approach for this situation in chapter 6.

With the framework developed so far, key-textures are easily specified. Recall that key-textures are specific frames that the transformation must contain. If it is desirable to use a particular parameter vector at some point for a shader

$p(\mathbf{x}_b)$ while evolving from $p(\mathbf{x}_a)$ to $p(\mathbf{x}_c)$ (with the transformation being denoted by the symbol $\rightsquigarrow$), the approach described above can be used to compute the paths from $p(\mathbf{x}_a) \rightsquigarrow p(\mathbf{x}_b)$, and from $p(\mathbf{x}_b) \rightsquigarrow p(\mathbf{x}_c)$. These paths can then simply be concatenated. This process can be repeated for as many key-textures as necessary.

5.3 Transformation Between Different Shaders

The technique described in Sec. 5.2 is only well suited to transformations *within* a procedural shader since different shaders no longer share a common parameter domain. If we need to transform *between* distinct shaders, we must find some way of connecting the texture transformation, either directly between the two shaders, or possibly by determining a path through some other connective procedural textures.

We will first consider the case of transforming directly between two distinct shaders. Since we have shown above how to transform within a single shader, what remains is to find suitable pairs of points (one in each shader) for which the shaders produce similar texture images. We refer to these transitional points as *jump points* since they determine good locations for switching or "jumping" from one shader to another.

Once suitable jump points between the shaders associated with the starting and ending textures have been determined, we can use the technique described above to first transform each texture to its respective jump point. For example, to transform from the texture $p_k(\mathbf{x}_a)$ to a texture due to another shader $p_l(\mathbf{y}_a)$, we would first find the best jump point between p_k and p_l, that is, the point in each shader's parameter domain ($\mathbf{x}_j$ and $\mathbf{y}_j$) which gives a maximum similarity measure $S(p_k(\mathbf{x}), p_l(\mathbf{y}))\ \forall\ [\mathbf{x}, \mathbf{y}]$. The paths from each shader to their respective jump point can then be concatenated:

$$p_k(\mathbf{x}_a) \rightsquigarrow p_k(\mathbf{x}_j)\ :\ p_l(\mathbf{y}_j) \rightsquigarrow p_l(\mathbf{y}_a) \tag{5.2}$$

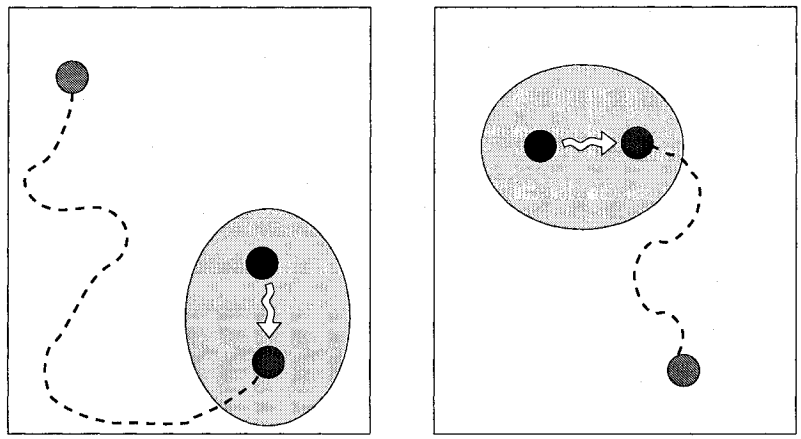


Figure 5–4: Finding jump points between two shaders. The parameter domains of two different shaders are illustrated by the two boxes. The jump regions (shown in yellow) are are identified using a sparse sampling of each shader. The initial matching samples are shown in blue. The actual jump point is then found by performing local optimisations in alternation to best match each point to the other. A path is then determined from the initial starting point (shown in green) to each of the jump points, and these paths are then concatenated.

Since finding these jump points via exhaustive search is prohibitively costly, an approach similar to that described in Sec 4.2 can be used to reduce the computational burden. By adaptively sampling each shader sparsely, and comparing each sample to the (also sparse) samples of the other shader, the best candidate jump *regions* can be found. To narrow these regions to the actual jump points, local optimisations of the similarity between the current candidate and the candidate in the other shader's jump region are performed. This is repeated in alternation until the distance travelled during a step for each candidate is negligible (see Fig. 5–4).

The transformation technique described so far chooses only a single jump point which maximises the similarity between two shaders. It is possible, however, that a better overall transformation exists which does not use this particular transitional point. Consider the case where the top two jump points have only slightly different similarities, but the lesser one (which would normally not be chosen) produces a smoother transformation within one or both of the

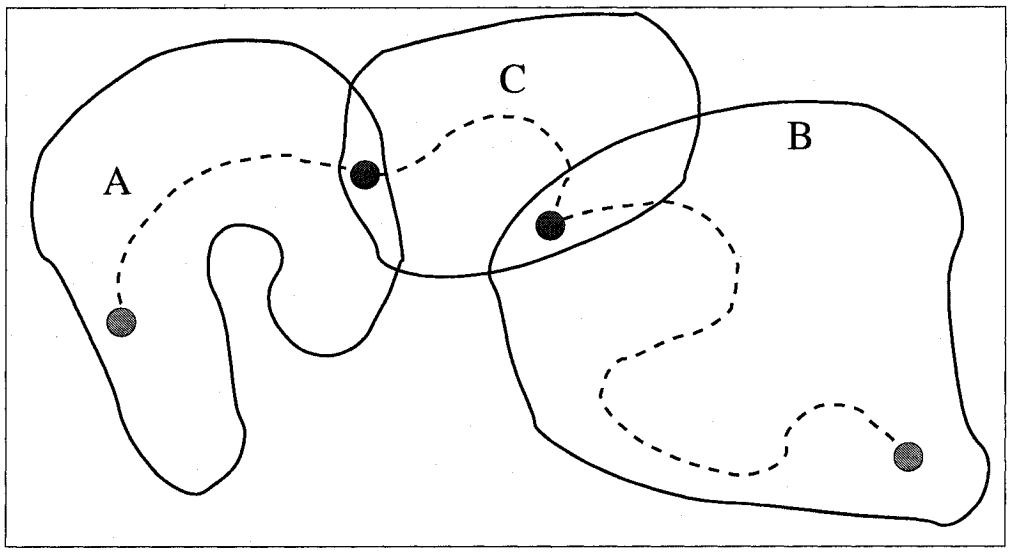


Figure 5–5: An example of a path through a connective shader. The transformation endpoints are shown in green, but unfortunately there are no suitable jump points between the shaders A and B. However, shader C has good jump points to both shaders, and is hence used as a connective shader.

shaders, and hence a smoother transformation overall. We can therefore see that it would be better to consider the jump points as part of the path taken in the transformation. This can be accomplished by linking the graphs for the two shaders by adding edges between the vertices in each graph corresponding to the jump points, with the edge weights set to the similarity measure of the textures produced by each shader. The transformation path can then be found as outlined above with one exception: during the path smoothing phase, the adaptive linear subdivision can not be used on any edges corresponding to jump points since the parameter vectors for each vertex belong to different shaders.

In the two shader transformation case, we have joined the connected components from the sample graphs of each shader using the appropriate jump points. Expanding on this notion, we can connect several different shader graphs to form a larger connected component, thus allowing the path found during the transformation to traverse other *connective* shaders as shown in Fig. 5–5.

In the case where the shaders have limited texture ranges, or share little resemblance, it is possible that the best jump points that connect them may not appear very similar. Various strategies for dealing with this situation are presented in chapter 6.

5.4 Examples

In this section we will demonstrate our method of creating texture transformations. We show examples of the three types of transformations described above in Sec. 5.1.

Our first examples are from the case of transforming within a single shader. Figs. 5–6, 5–7 and 5–8 demonstrate how the transformation technique can be used to create smoothly varying intermediate texture frames. In each of these cases, the end point textures were manually specified, the paths were found using the graph search, and then refined using the adaptive linear subdivision methods described above. Each of these examples exhibits smoothly varying textural characteristics, as is desired for an effective texture transformation.

We next show examples of the second type of texture transformation, namely, transformation between two different shaders. The first example, shown in Fig. 5–9, is an illustration of transforming between two different types of brick shaders. The transitional point between the two shaders can be seen where the adjacent rows of bricks begin show two distinct colours as is the case with the second brick shader.

The next two-shader transformation example, shown in Fig. 5–10, has its endpoints specified by two of the Brodatz texture matching examples shown in Figs. 4–17(a) and 4–17(c). In this case, the two textures for the transitional point have inverted intensities, however, their textural characteristics are very similar.

The last example of transforming between two different shaders, shown in Fig. 5–11, uses the real world texture target images in Figs. 4–19(a) and 4–19(c) as the starting and ending frames. In this particular transformation, the best jump point between the cloud shader and the star field shader results from parameters which produce a simple coloured background. This is actually the best match from a perceptual viewpoint since clouds and stars are distinct textures.

An example of a transformation using a connective shader is shown in Fig. 5–12. In this case, the desired transformation is between a small rectangle and a small circle, each produced by a different shader. For this transformation, there were no satisfactory jump points directly between the two shaders. However, a suitable path was found through a super-ellipse shader since for that shader there were good jump points to both the rectangle and circle shaders given that the super-ellipse shader is capable of producing images similar to both a square and a circle.

All of the examples presented in this section exhibit the smooth perceptual transition desired for an effective texture transformation. The transformations which contained more than one shader also demonstrate very satisfactory transitional points.

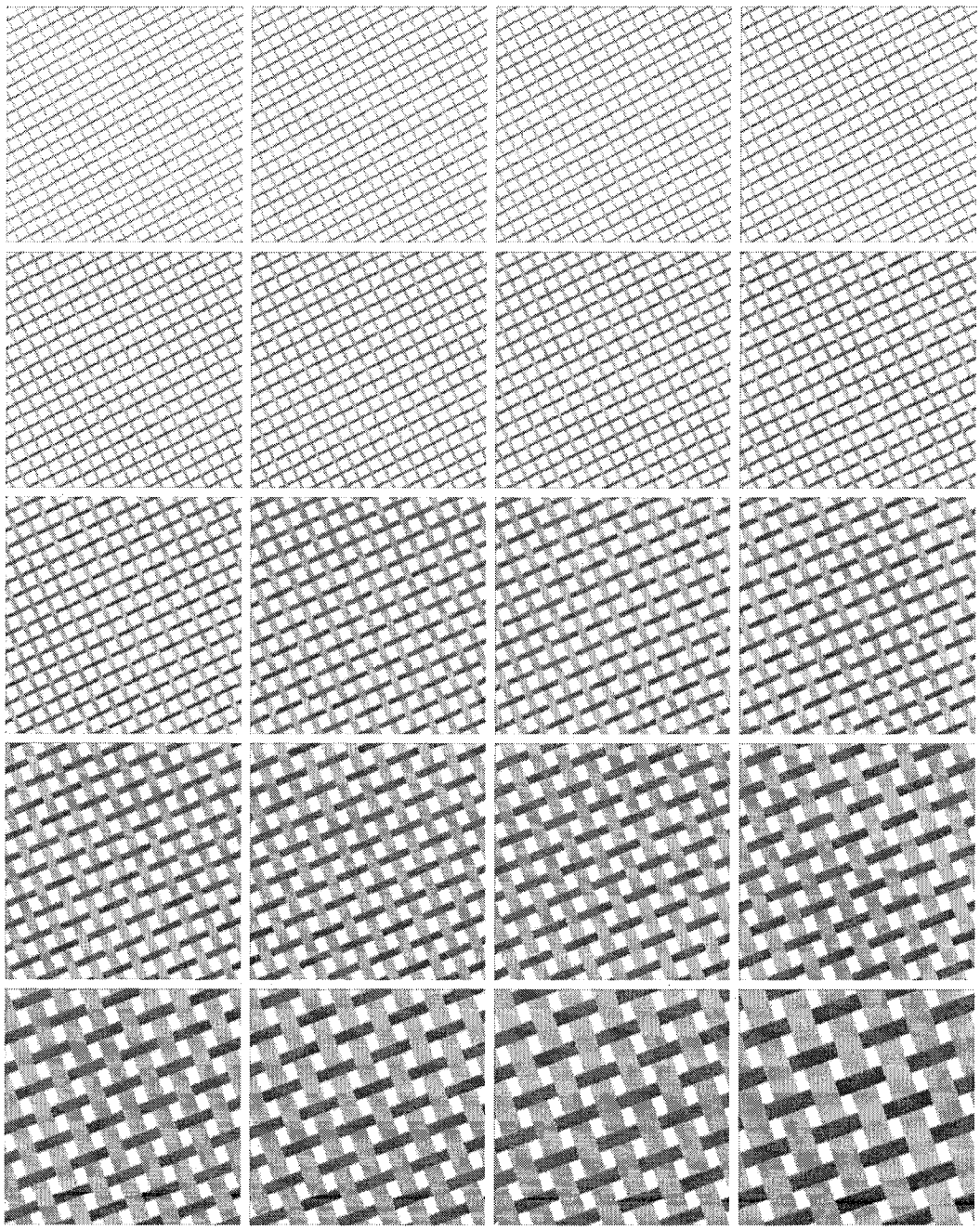


Figure 5–6: An example of a texture transformation within a single shader.

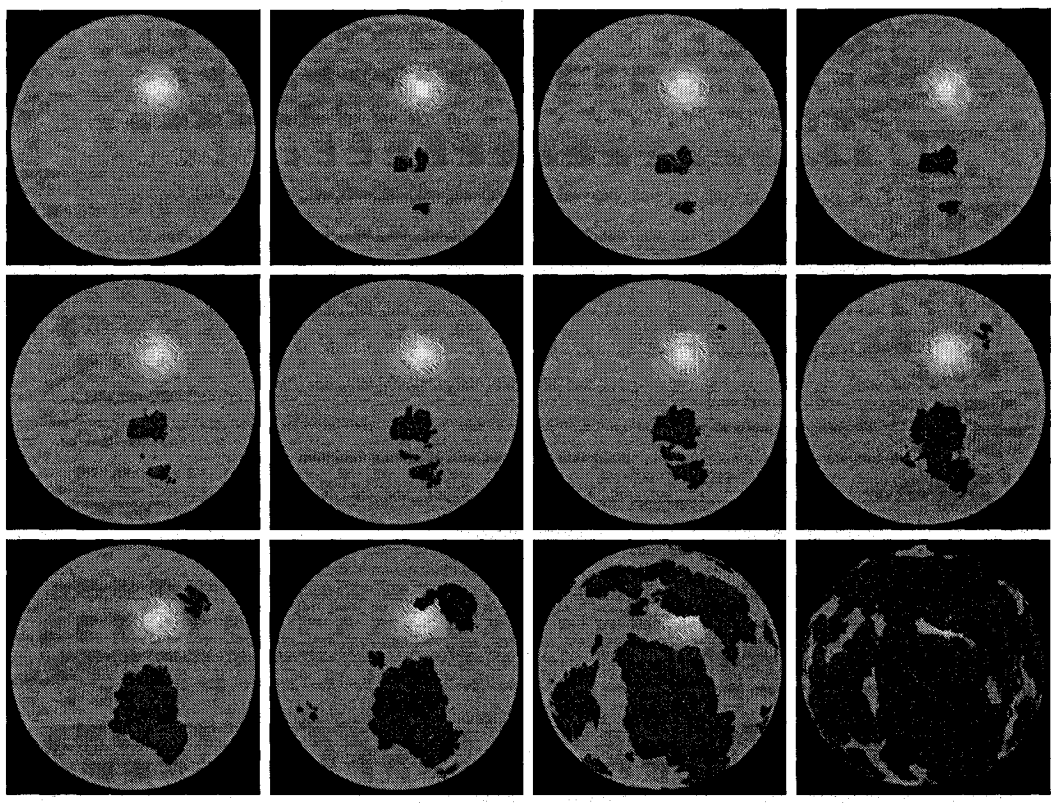


Figure 5–7: Another single shader texture transformation. The texture has been applied to a sphere for the sake of illustration.

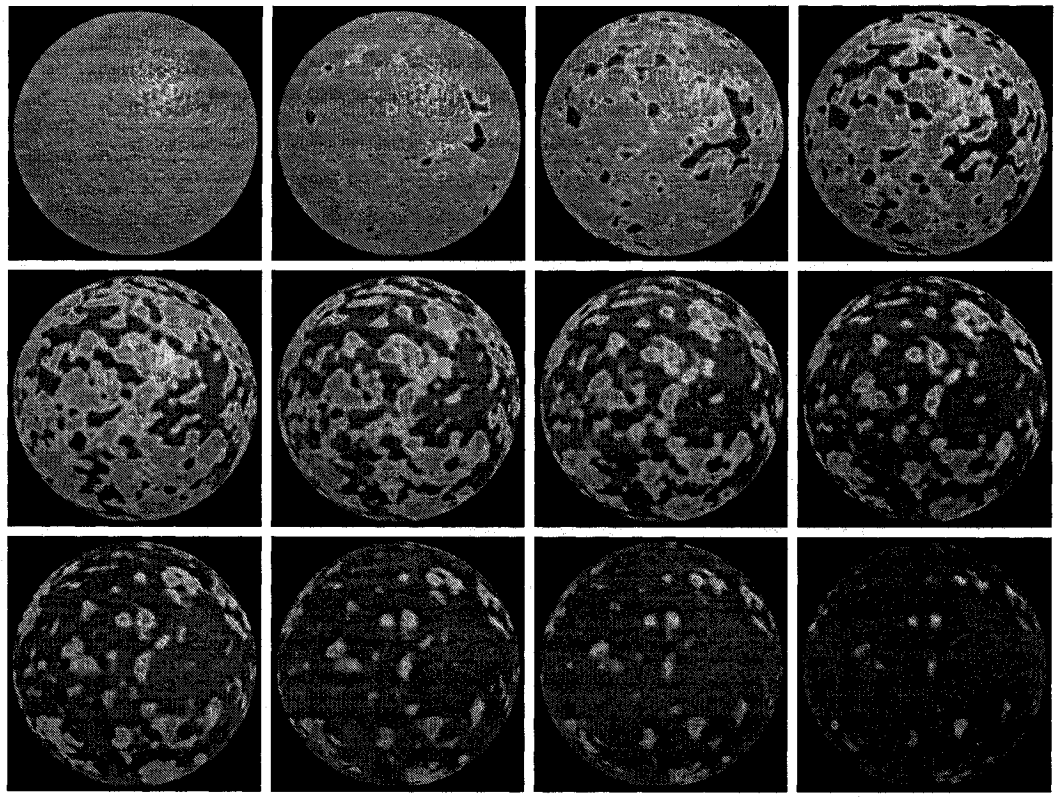


Figure 5–8: Another single shader texture transformation. The texture has been applied to a sphere for the sake of illustration.

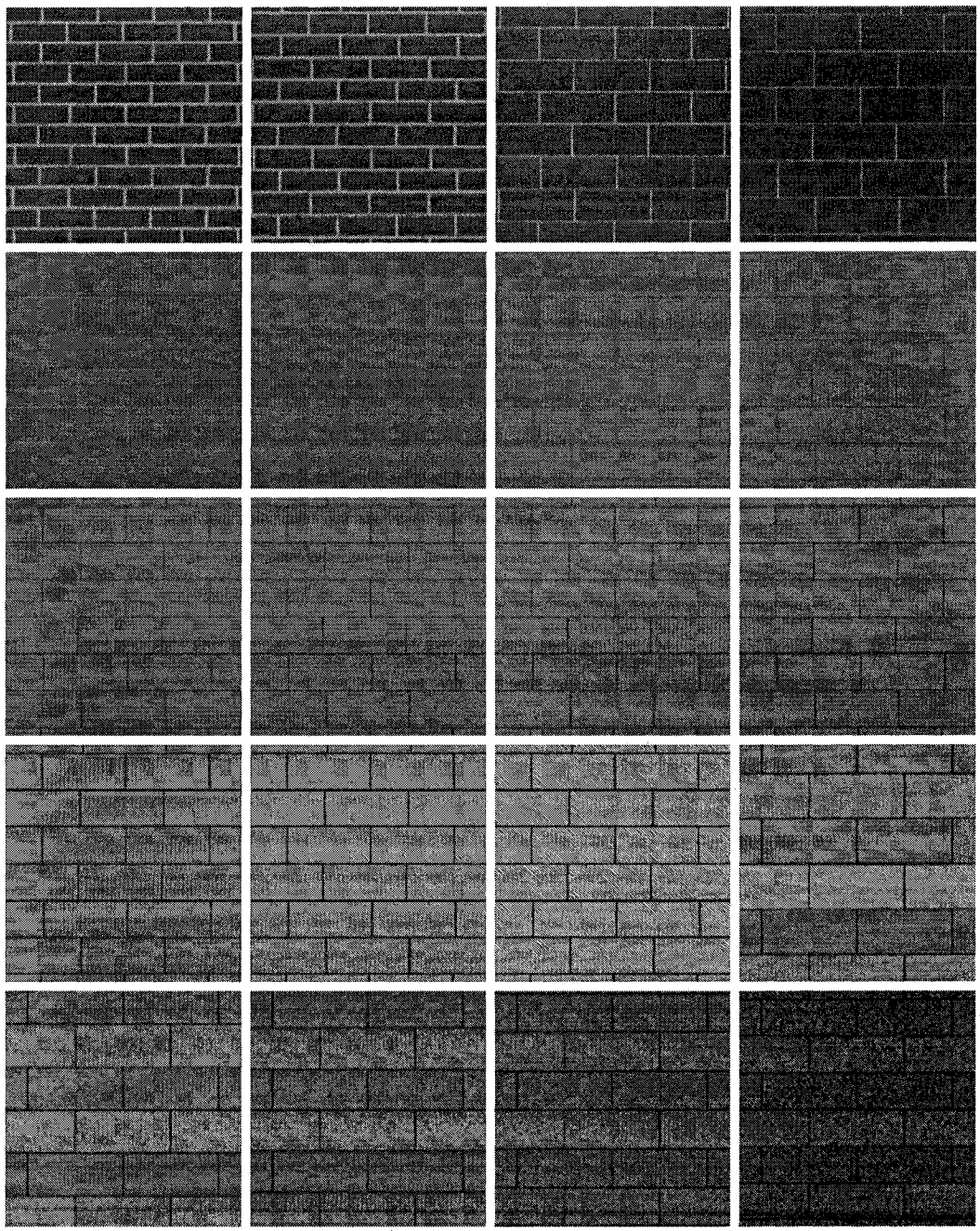


Figure 5–9: A texture transformation between two different shaders. The transitional point is between the third and fourth frames of the fourth row.

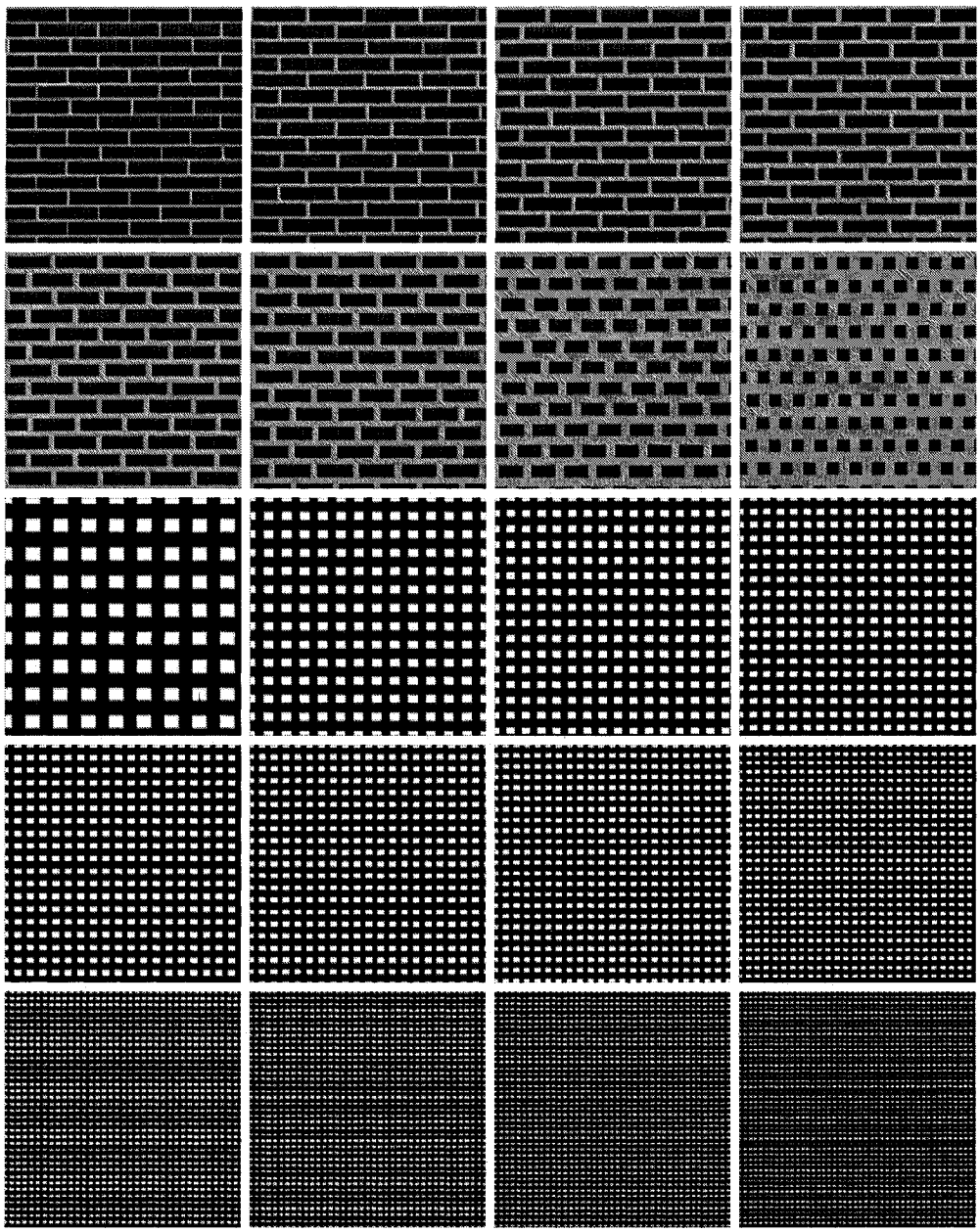


Figure 5–10: Another texture transformation between two different shaders based on the Brodatz texture matches from Figs. 4–17(b) and 4–17(d). The transitional point is between the last frame of the second row and the first frame of the third row. In this case, the two textures for the transitional point have inverted intensities, however their textural characteristics under our similarity metric are very close.

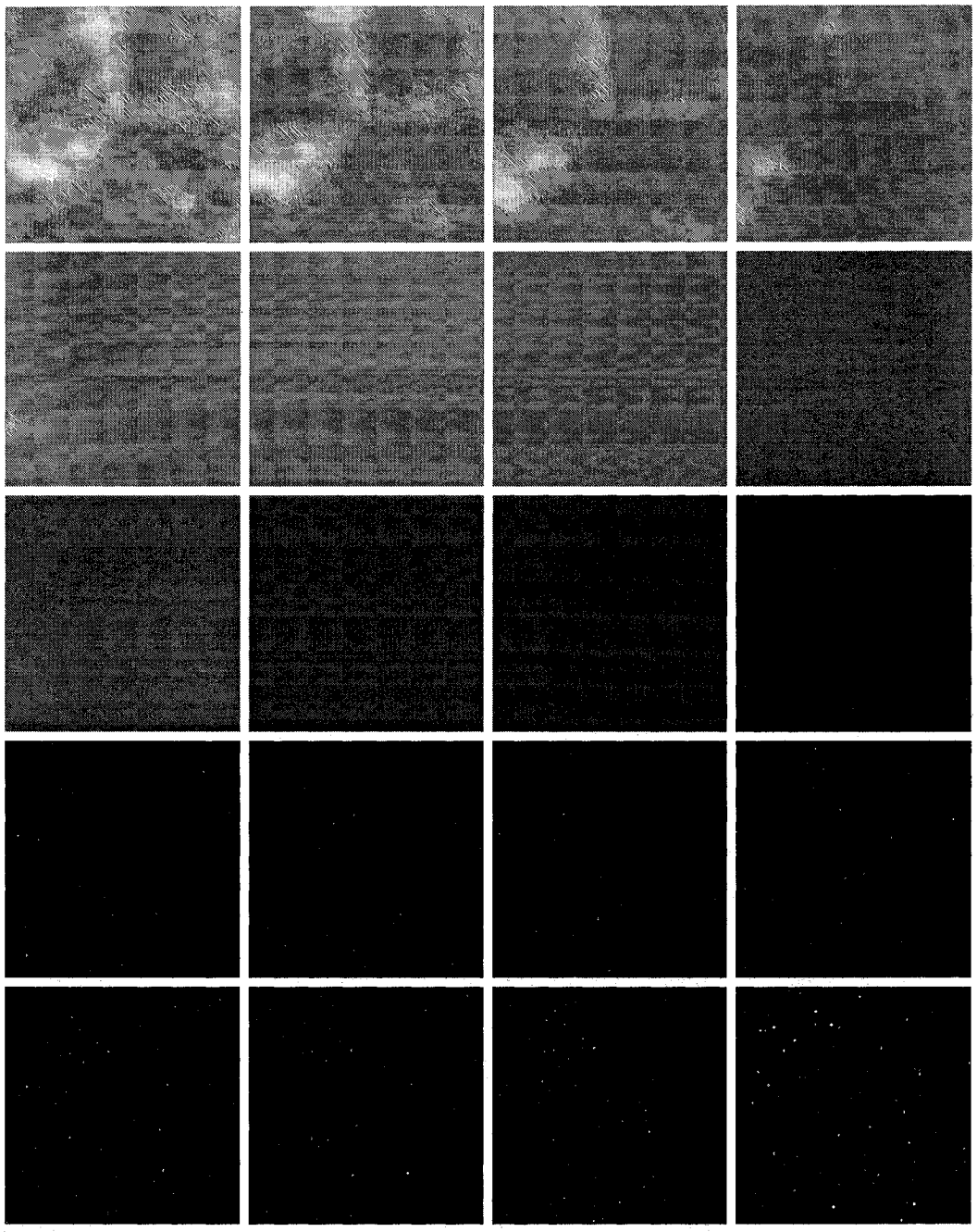


Figure 5–11: A multi-shader transformation from the texture matches corresponding to the real world images shown in Figs. 4–19(a) and 4–19(c). Some reproductions fail to show the detail in the frames containing the star field. Refer to Fig. 6–1 to see an enlarged and intensity inverted version of the last three frames.

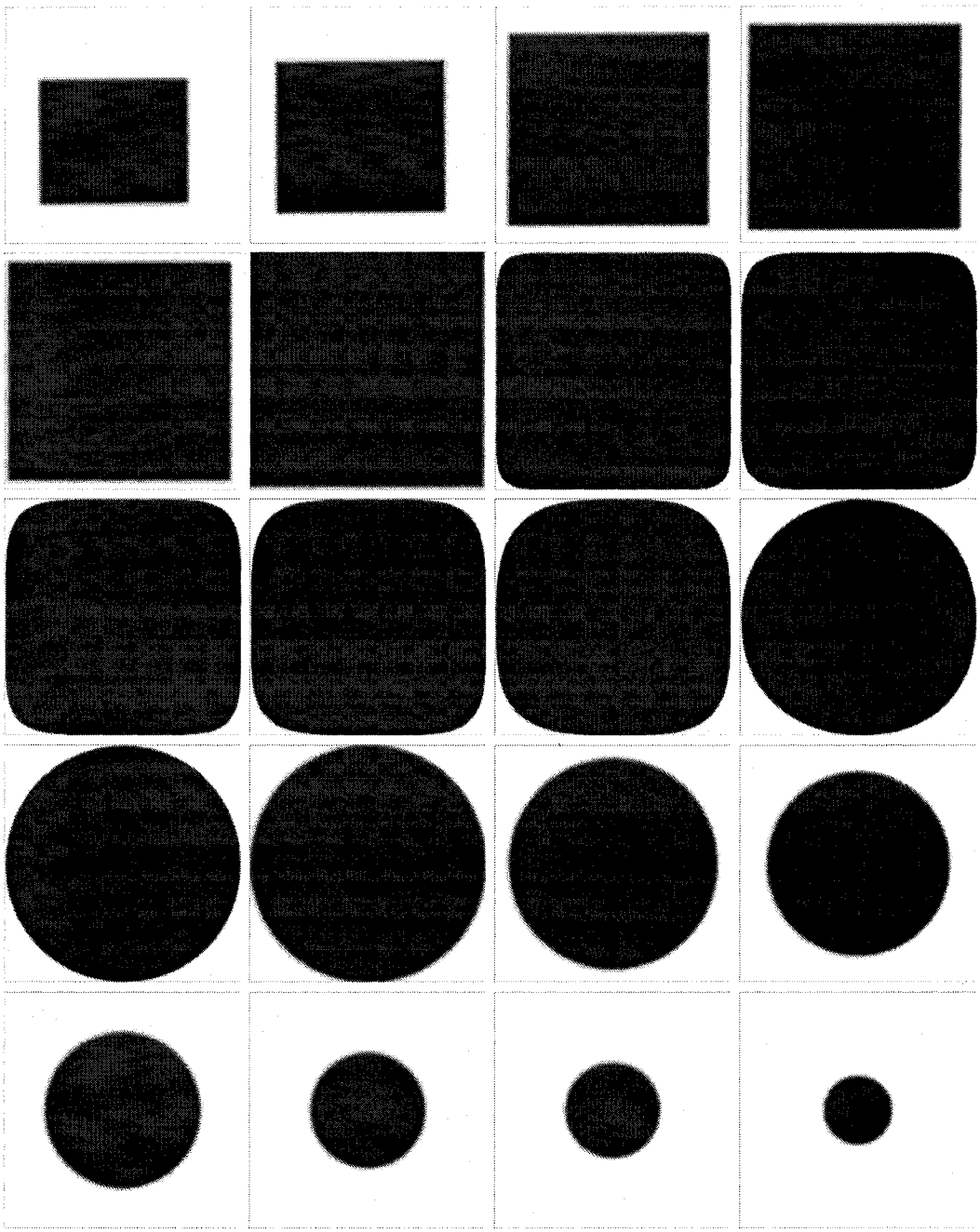


Figure 5–12: An example transformation which uses a connective shader. There were no satisfactory jump points from the rectangle shader to the circle shader, however, a suitable transformation was obtained by passing through a super-ellipse shader. The transition from the rectangle to the super-ellipse occurs between the second and third images in the second row, and the transition from the super-ellipse to the circle occurs between the first two images in the fourth row.

CHAPTER 6

Conclusion

In this thesis, we have presented a technique which allows one to replicate a texture sample procedurally by automatically selecting an appropriate procedural texture from a library. In addition, this technique refines the parameters for the selected procedural texture in order to best match the texture sample based on a perceptually motivated similarity measure. Our solution to this problem involves a two-stage approach, first performing a global search over pre-computed data, followed by a local search stage to refine the quality of the match.

In order for the parameter estimation technique to succeed, the ensemble of procedural shaders must be large enough to approximate the specified texture target. If this is not the case, it will be detected by a large residual error in the similarity measure. In this thesis, we have shown several examples of texture matching using a wide variety of texture samples.

Given the ability to find a procedural specification for a given texture, we then developed a method for creating texture transformations. These transformations are sequences of texture samples in which each adjacent texture is similar to its neighbours, yet an overall transformed appearance between the starting and

ending texture samples is created. This is accomplished by again making use of the pre-computed data from the texture matching stage, combined with a method for finding a path through the texture space of the procedural textures, as well as a technique for smoothing the selected path.

When transformations are created which entail the use of several shaders, then transitional or jump points are needed in order to connect the paths from each individual procedural texture. It is possible, however, that even the best jump points between two shaders will exhibit an unsatisfactory disparity. In these cases it would be possible to use morphing techniques to smooth the transition through these non-ideal jump points. One such method, due to Liu *et al.*, involves morphing between two texture samples using a pattern-based approach [53]. Unfortunately, this method requires the end-user to manually specify many feature correspondence landmarks between the two textures in order to achieve an acceptable morph.

The path planning process in texture space is based on graph search, and a perceptually motivated iterative subdivision procedure to refine the transformation. In some cases this adaptive linear subdivision may not succeed in bringing the similarity of adjacent texture samples above the given threshold, or we may wish to have more precise control over the sequence of textures. For example, we might want to avoid some types of appearance or shader parameter vectors while guaranteeing the texture evolves in a specific fashion. We are currently exploring the use of high-dimensional path planners to enable this type of control [47].

Another issue with the texture transformations is that some kinds of textures can exhibit spatial changes while maintaining perceptual similarity. In an animation, this can produce adjacent texture frames which appear to exhibit a fair amount of motion despite their perceptual similarity. Consider the case of a night sky procedural texture: while two samples of this texture could have

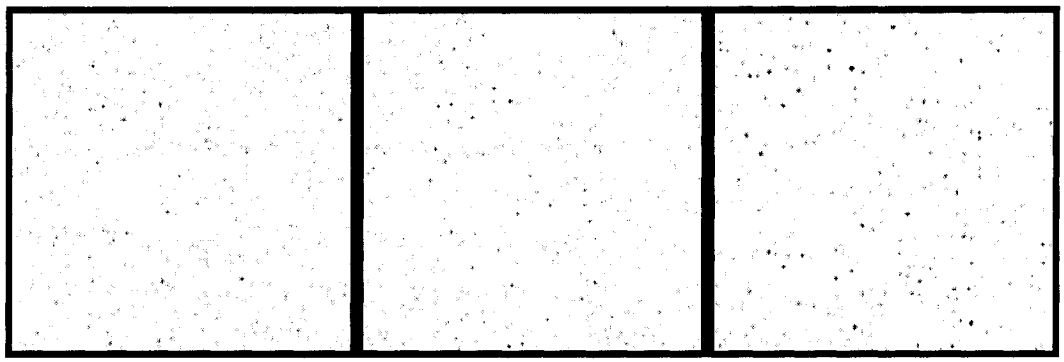


Figure 6–1: An example of adjacent texture frames which show little spatial coherence despite their perceptual similarity. This figure is a reproduction of the last 3 frames of Fig. 5–11, with the intensities inverted for the sake of clarity.

similar distributions of bright stars, small stars, etc., placing two such textures next to each other in an animation would not exhibit the smoothness we seek. An example of this spatial instability can be seen in Fig. 6–1, which is an enlarged version of the last three frames from Fig. 5–11. Close examination of these three adjacent frames reveals that the stars are never in the same location. We are exploring the use of an augmented similarity measure which combines frequency selective content as well as spatially selective content to combat this problem.

6.1 Future Work

Currently, our system assumes that the target texture can in fact be approximated by at least one procedural texture in the given library. An extension would be to augment our system to be able to synthesise new procedural textures when no suitable matches are found. This could be accomplished by measuring certain textural features, and reconstructing them by layering 'procedural building blocks'. These building blocks would be like textural 'DNA' which could be combined, modified and selected using one of several search methods (ranging from Bayesian methods to genetic algorithms). This approach to texturing has been examined non-procedurally, but its application to procedural textures is promising [73, 83]. Because many different solutions would need to be evaluated, and

each solution would be independent, they could be evaluated in parallel thus providing a large computational gain. Our work in the area of cluster computing [12] combined with this new research direction could yield positive results.

An alternative approach to synthesising new procedural textures would be to design *meta-shaders* based on current pyramidal synthesis techniques. The parameters for these shaders would control various operations which could be performed at each level of the synthesis pyramid. We could even make use of the procedural building blocks described above to specify the contents of each level of the pyramid independently, before collapsing the pyramid to determine the final texture.

In this thesis we have presented two different approximations to the ideal similarity measure S^* described in Sec. 4.5. Ongoing research on human texture perception will expand our understanding of how we perceive textures, and thus provide better computational approximations for our framework. It may also be possible to devise a method for using multiple similarity measures in concert to manage a combination of textural features.

Currently the catalogue samples are all stored at a fixed resolution. In order to increase the speed of the texture comparisons during the global search, we could store each sample in an image pyramid which would allow the user to select lower resolutions for the similarity comparisons if desired. In addition, the Laplacian pyramid and power spectrum could also be stored in the image database to avoid computing them at run time.

Our approach to texture matching could also potentially be used for texture classification. Recall from Sec. 4.5 that texture classification is the problem of identifying the class of which a given texture sample is a member. By creating procedural representations to model various texture classes specific to the domain

of the recognition task, one could then classify texture samples based on which procedural texture (model) provides the best match.

This work also has potential applications for sound matching and synthesis. Many of the techniques already developed for procedural texturing might extend well to the acoustic domain. Similar to the texture domain, storing large collections of sound samples for various applications is not feasible given the amount of space which would be required for decent sampling rates (and hence audio quality). By applying our texture matching approach to the acoustic domain, procedural sound generators could be parameterised automatically to match given real-world sounds. In particular, we anticipate that sound matching based on spectrogram analysis will be a close fit to our current perceptually motivated search technique.

Once generic sounds can be replaced by small procedural sound generators, including the latter in an augmented three-dimensional model becomes a realistic goal. This would enable the rendering of a sound track for a scene using procedural sound generation based on the interaction of objects and their material properties as they are animated either directly (eg., live, under user control as in a virtual reality environment) or indirectly (eg., pre-planned as in a motion picture). This line of research would be a natural extension to work done by Dobashi *et al.* [30]. In addition, other attributes of the objects in a scene could be modelled, such as ageing through procedural texture transformation.

The approach described above could also be applied to procedural animation – a technique which allows one to model complex motion behaviours by specifying a set of constraints (eg., permissible joint angles in the human body), and generation patterns (eg., walking gaits). The framework for matching real-world signals to procedural approximations could be useful for extracting behaviours

from motion capture data, thus allowing a graphic artist more control when selecting and tuning the behaviours of digital actors. The compact representation gained by a procedural model is particularly useful in the gaming industry where rapid scene generation is essential in order to release a competitive product.

The work presented in this thesis may also be loosely applicable to other domains, an example of which is the field of proteomics. Proteomics is the study of the structure and function of proteins. One branch of proteomics, namely protein sequence analysis, seeks to discover evolutionary relationships of proteins and it is possible that our high-dimensional transformation framework could be used to find protein sequences, or be used to classify pre-existing sequences.

Overall, the automatic selection and synthesis of procedural textures, texture transformations, models, sounds, and animations exhibits vast research potential in the field of computer graphics. Eventually, it will be possible to render complete procedurally-specified environments in real time, allowing unprecedented complexity and realism thus opening new avenues for computer generated imagery.

APPENDIX A

Software Architecture

A.1 System Design

The software architecture design created in order to perform the texture matching and transformation discussed in this thesis contains many abstract elements, and as such can be used for a number of different combinations of computational problems involving search in high dimensional spaces.

Because procedural textures can come in several forms, and can be tied to specific renderers, we have designed a flexible texture class hierarchy as is illustrated using the unified modelling language (UML) [11] in Fig. A–1. Each texture object contains a renderer object, as well as a 'parameters' object. The renderer class is an abstract class from which specific renderer classes are derived and is itself derived from the threaded class to allow the rendering to be performed in parallel. The 'parameters' object can be used to describe an arbitrary collection of parameters as it is composed of individual objects of the parameter template class. The fact that the parameter class is templated allows our method to search over any type of parameter as long as basic ordering operations are defined for the given type.

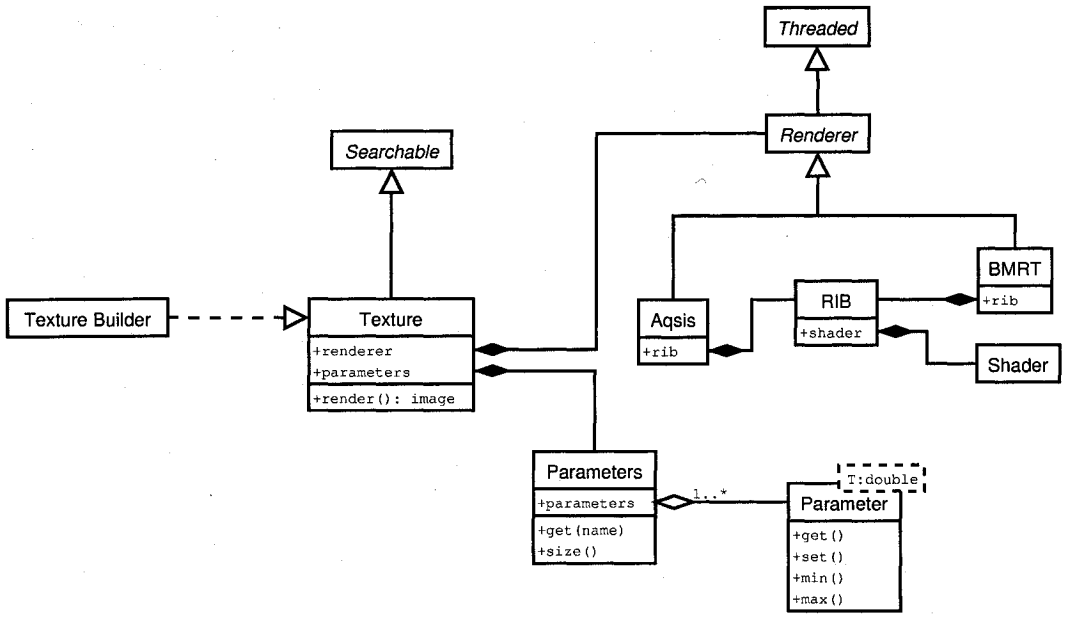


Figure A–1: The class hierarchy for the texture class.

For this work we have used renderers which support the RenderMan® shading language (RSL), with scenes specified using the RenderMan® interface bytestream (RIB). Therefore each of the concrete rendering classes make use of a rib object for specifying scene files, and a shading object which interacts directly with the shading language file for the particular shader being used. The two renderers we have used are *Aqsis*, and *Blue Moon Rendering Tools* (BMRT), each of which is a free implementation of of the RenderMan® specification. This design will allow for easy expansion in order to use other renderers which may have their own shading language, or to even take advantage of some of the newer hardware shading languages.

Each texture object is composed of several parts requiring initialisation, so we therefore use the *builder* creational pattern for the texture builder class [35]. As can be seen in the class hierarchy shown in Fig. A–1, the texture class is derived from a searchable abstract class. This design allows us to apply our parameter sampling and searching algorithms to other application domains, an example of which is discussed in chapter 6.

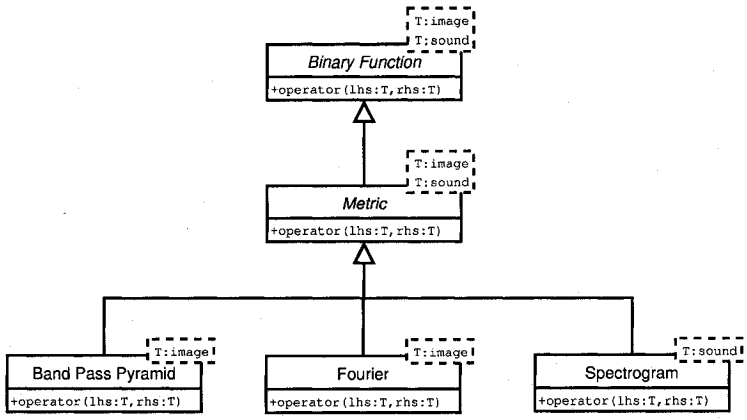


Figure A–2: The class hierarchy for the metric class.

For texture matching, the sampling of each shader is accomplished using a design based on the abstract factory creational pattern, and the global search uses the visitor behavioural pattern to locate the promising regions for search. All of the user specified settings are stored in a preferences class based on the singleton pattern.

The similarity measure classes are derived from the metric abstract class, which is itself derived from the C++ Standard Template Library's "binary function" template class (see Fig. A–2). We can therefore easily plug in different similarity measures which simplifies extensions to other domains as illustrated by the spectrogram class for sound comparison.

Many of the algorithms presented in this thesis exhibit characteristics which enable them to be executed concurrently. This presents a problem of communicating between the various working units. We have implemented a message class which allows objects to communicate using messages formatted in the extensible markup language (XML). We use XML for our message passing so that we can support future changes without modifying the current implementation. The use of XML also simplifies debugging new features as one can easily interpret the messages being passed back and forth.

The message class can be used both to communicate between threads on the same processing unit, as well as to communicate between processes running in a distributed environment. For the distributed case, our message class uses the MPICH implementation of the Message Passing Interface (MPI) standard [37].

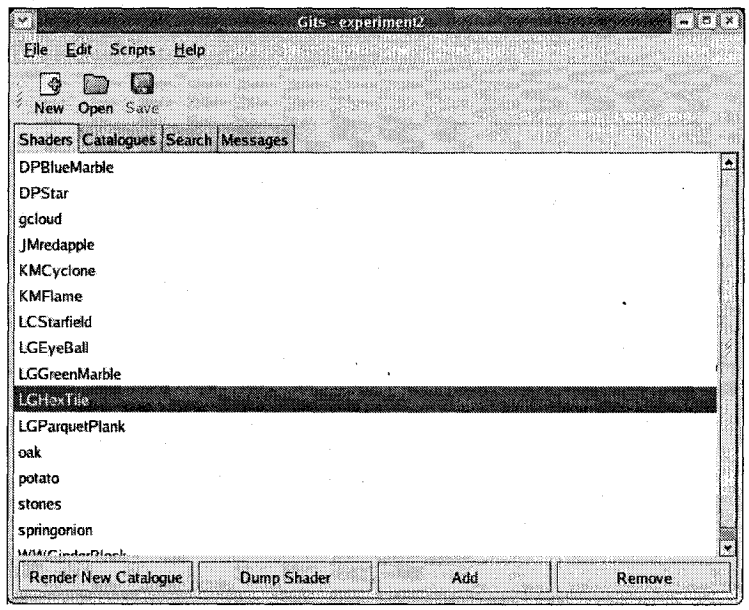
A.2 Graphical User Interface

In order to make the capabilities of this work readily accessible to naïve users, we have also developed a complete graphical user interface (GUI). This interface, examples of which are shown in Figs. A–3 and A–4, has a main view which provides simple access to all the pertinent information concerning the current texture match. The shader view, shown in Fig. A–3(a), contains a list of all the shaders currently in the library. Double clicking on a particular shader in the list presents the parameter view dialog to the end-user, allowing them to manipulate the range of each of the parameters, as well as to choose which parameters will be active during the search (see Fig. A–3(b)). Many shaders contain parameters which do not affect the textural characteristics of the rendered images, such as ambient, diffuse, and specular lighting constants used in the illumination model. The dimensionality of the search can be therefore be reduced by disabling these particular parameters. Any parameters which have been disabled will automatically be set to their default values as specified in the parameter view dialog.

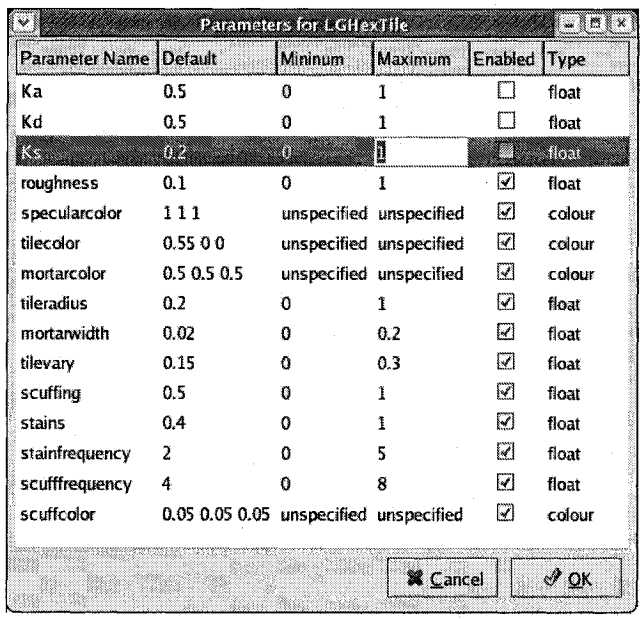
The search criteria can also be specified using the GUI's search view illustrated in Fig. A–4(b). In this view, the end-user is able to specify which similarity measure to use, as well as which local optimisation technique should be employed. This view also allows for the specification of several settings relevant to the chosen search technique. Each application dialog is pre-filled with reasonable defaults so that a novice user can simply point and click, yet

the informed user is afforded the ability to tune the search settings if desired. Finally, an example of the dialogs for rendering the sample catalogues are shown in Figs. A–4(c) and A–4(d).

It can be noted that, through the use of sophisticated abstract design tools as described above, our system is able to exhibit a great degree of flexibility and reconfigurability. It is this flexibility that allows our architecture to readily support both the texture matching and transformation presented in this thesis, as well as our preliminary work on acoustic signal processing.

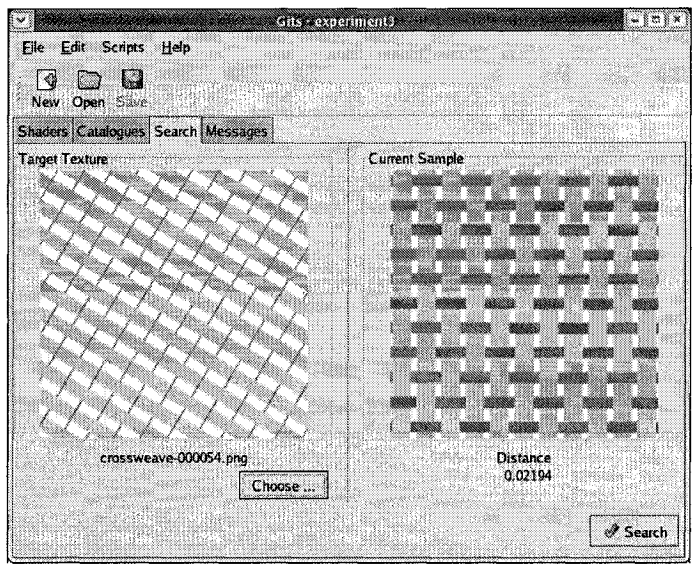


(a)

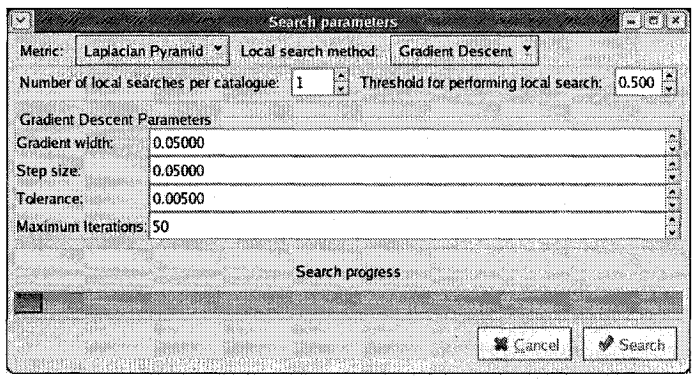


(b)

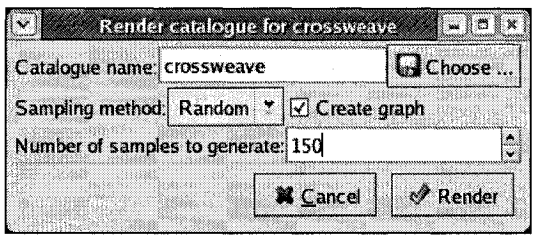
Figure A–3: Two example screen shots of the texture matching application. The shader list view is shown in (a), and the parameter view dialog shown in (b) is displayed when the user double clicks on a shader in the list.



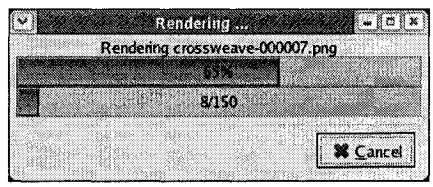
(a)



(b)



(c)



(d)

Figure A–4: More screen shots of the texture matching application. The search view, shown in (a), reflects the texture target as well as the current texture sample being considered. The search parameters dialog is shown in (b) and the dialogs for rendering new catalogues are shown in (c) and (d).

APPENDIX B

Shading Language Code Example 1

Below is the shading language code for the match found in Fig. 4–18(b). This shader was written by F. Kenton Musgrave.

The parameter values for the match were set as follows:

- Ka: 0.5
- Kd: 0.75
- max_radius: 2.28518
- twist: 0.0928873
- scale: 0.956974
- offset: 0.748854
- omega: 0.968523
- octaves: 6.388

```
/*
 * cyclone.sl - surface for a semi-opaque cloud layer to be put on an
 *              earth-like planetary model to model clouds and a cyclone.
 *
 * AUTHOR: Ken Musgrave
 *
```

```
 */

#define TWOPI (2*PI)

/* Use signed Perlin noise */
#define snoise(x) ((2*noise(x))-1)
#define DNoise(p) (2*(point noise(p)) - point(1,1,1))
#define VLNoise(Pt,scale) (snoise(DNoise(Pt)+(scale*Pt)))
#define VERY_SMALL 0.001

surface
   KMCyclone (float Ka = 0.5,
      float Kd = 0.75;
      float max_radius = 2.28518;
      float twist = 0.0928873;
      float scale = 0.956974,
      float offset = 0.748854;
      float omega = 0.968523;
      float octaves = 6.388;)
{
  float radius, dist, angle, sine, cosine, eye_weight, value;
  point Pt;
  point PN;
  point PP;
  float l, o, a, i;

  Pt = transform ("shader", P);

  PN = normalize (Pt);
  radius = sqrt (xcomp(PN)*xcomp(PN) + ycomp(PN)*ycomp(PN));

  if (radius < max_radius) {
      dist = pow (max_radius - radius, 3);
      angle = PI + twist * TWOPI * (max_radius-dist) / max_radius;
      sine = sin (angle);
      cosine = cos (angle);
      PP = point (xcomp(Pt)*cosine - ycomp(Pt)*sine,
  xcomp(Pt)*sine + ycomp(Pt)*cosine,
  zcomp(Pt));

      if (radius < 0.05*max_radius) {
```

```
  eye_weight = (.1*max_radius - radius) * 10;
  eye_weight = pow (1 - eye_weight, 4);
}
      else eye_weight = 1;
    }
  else PP = Pt;

  if (eye_weight > 0) {
      l = 1;  o = 1;  a = 0;
      for (i = 0;  i < octaves  &&  o >= VERY_SMALL;  i += 1) {
  a += o * VLNoise (PP * l, 1);
  l *= 2;
  o *= omega;
}
      value = abs (eye_weight * (offset + scale * a));
    }
  else value = 0;

  Oi = value * Os;

  Ci = Oi * (Ka * ambient() + Kd * diffuse(faceforward(normalize(N),I)));
}
```

APPENDIX C

Shading Language Code Example 2

Below is the shading language code for the match found in Fig. 4–18(d). This shader was written by Larry Gritz.

The parameter values for the match were set as follows:

- Ka: 0.75
- Kd: 0.75
- Ks: 0.4
- roughness: 0.1
- specularcolor = 1
- iriscolor: color (0.135289, 0.084323, 0.372417)
- irisoutercolor: color (0.403882, 0.343944, 0.68276)
- irisinnercolor: color (0.065142, 0.040605, 0.179311)
- eyeballcolor: color(1,1,1)
- bloodcolor: color(0,0,0)
- pupilcolor: 0
- pupilsize: 0.0
- irissize: 0.0

- bloodshot: 0.997141
- veinfreq: 3.54332
- veinlevel = 7.549
- index: 0

```
/*
 * eyeball.sl -- RenderMan compatible shader for an eyeball.
 *
 * AUTHOR: written by Larry Gritz
 *
 */

surface
LGEyeBall (float Ka = .75, Kd = 0.75, Ks = 0.4, roughness = 0.1;
 color specularcolor = 1;
 color iriscolor = color (.135289, .084323, .372417);
 color irisoutercolor = color (.403882, .343944, .68276);
 color irisinnercolor = color (.065142, .040605, .179311);
 color eyeballcolor = color(1,1,1);
 color bloodcolor = color(0,0,0);
 color pupilcolor = 0;
 float pupilsize = 0.0, irissize = 0.0;
 float bloodshot = 0.997141;
 float veinfreq = 3.54332, veinlevel = 7.549;
 float index = 0;
        )
{
#define snoise(P) (2*noise(P)-1)
#define MINFILTERWIDTH 1.0e-7
  color Ct;
  point Nf;
  point PP, PO;
  float i, turb, newturb, freq, f2;
  float displayed, newdisp;
  color Cball, Ciris;
  float irisstat, pupilstat;
  float bloody, tt;
  float ks, rough;
  float twidth, cutoff;
```

```
twidth = max (abs(Du(t)*du) + abs(Dv(t)*dv), MINFILTERWIDTH);
PO = transform ("object", P) + index;

tt = 1-t;
irisstat = smoothstep (irissize, irissize+twidth, tt);
pupilstat = smoothstep (pupilsize, pupilsize+twidth, tt);
bloody = bloodshot * (smoothstep (-irissize, 2.5*irissize, tt));

if (irisstat * bloody > 0.001) {
    turb = bloody;  freq = veinfreq;
    displayed = 0;
    for (i = 1;  (i <= veinlevel) && (turb > 0.1);  i += 1) {
newturb = 1 - abs (snoise(PO*freq + point(0,0,20*freq)));
newdisp = pow (smoothstep (.85, 1, newturb), 10);
displayed += (1-displayed) * newdisp * smoothstep (.1, .85, turb * turb);
turb *= newturb;
freq *= 2;
      }
    Cball = mix (eyeballcolor, bloodcolor, smoothstep(0,.75,displayed));
  }
else Cball = eyeballcolor;

Ciris = mix (iriscolor, irisoutercolor, smoothstep (irissize*.8, irissize, tt));

if (irisstat < 0.9999 && pupilstat > 0.0001) {
    turb = 0;  freq = 1;  f2 = 30;
    for (i = 1;  i <= 4;  i += 1) {
turb += snoise (PO*f2 + point(0,0,20*f2)) / freq;
freq *= 2;  f2 *= 2;
      }
    Ciris *= (1-clamp(turb/2,0,1));
  }

Ct = mix (Ciris, Cball, irisstat);
Ct = mix (pupilcolor, Ct, pupilstat);

ks = Ks * (1+2*(1-irisstat));
rough = roughness * (1-.75*(1-irisstat));

Oi = Os;
```

```
  Nf = faceforward (normalize(N),I);
  Ci = Os * ( Ct * (Ka*ambient() + Kd*diffuse(Nf)) +
       specularcolor * ks*specular(Nf,-normalize(I),rough));
}
```

References

[1] Edward H. Adelson and Eero Simoncelli. Orthogonal pyramid transforms for image coding. In *SPIE Visual Communications and Image Processing II*, volume 845, pages 50–58, 1987.

[2] Aseem Agarwala, Ke Colin Zheng, Chris Pal, Maneesh Agrawala, Michael Cohen, Brian Curless, David Salesin, and Richard Szeliski. Panoramic Video Textures. *ACM Transactions on Graphics*, 24(3):1–8, 2005.

[3] James R. Bergen. *Spatial Vision*, volume 10, chapter 5: Theories of visual texture perception, pages 114–134. CRC Press, 1991.

[4] James R. Bergen and Edward H. Adelson. Early vision and texture perception. *Nature*, 333:363–364, 1988.

[5] James R. Bergen and Michael S. Landy. *Computational Models of Visual Processing*, chapter 17: Computational Modeling of Visual Texture Segregation, pages 253–271. MIT Press, 1991.

[6] M. Bertalmio, G. Sapiro, V. Caselles, and C. Ballester. Image inpainting. In *SIGGRAPH*, pages 417–424, July 2000.

[7] M. Bertalmio, L. Vese, G. Sapiro, and S. Osher. Simultaneous structure and texture image inpainting. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2003.

[8] Eric A. Bier and Kenneth R. Sloan. Two-part texture mapping. *IEEE Computer Graphics and Applications*, 6(9):40–53, 1986.

[9] James F. Blinn. Simulation of wrinkled surfaces. In *SIGGRAPH*, pages 286–292, 1978.

[10] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, October 1976.

[11] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[12] Eric Bourque. Optimizing performance through parallelism. *Linux Journal*, 1(86):100–106, June 2001.

[13] Eric Bourque and Gregory Dudek. Viewpoint selection – An autonomous robotic system for virtual environment creation. In *Proceedings of the IEEE Conference on Intelligent Robotic Systems*, volume 1, pages 526–532, Victoria, Canada, October 1998.

[14] Eric Bourque and Gregory Dudek. On-line construction of iconic maps. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 3, pages 2310–2315, San Francisco, CA, April 2000.

[15] Eric Bourque and Gregory Dudek. On the automated construction of image-based maps. *Autonomous Robots*, 8(2):173–192, April 2000.

[16] Eric Bourque and Gregory Dudek. Procedural texture matching and transformation. *Computer Graphics Forum*, 23(3):461–468, 2004.

[17] Eric Bourque, Gregory Dudek, and Philippe Ciaravola. Robotic sightseeing - A method for automatically creating virtual environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 4, pages 3186–3191, Leuven, Belgium, May 1998.

[18] Phil Brodatz. *Textures – A Photographic Album for Artists and Designers*. Dover, 1966.

[19] Stephen Brooks and Neil A. Dodgson. Self-similarity based texture editing. *ACM Transactions on Graphics*, 21(3):653–656, 2002.

[20] F. Buekenhout and M. Parker. The number of nets of the regular convex polytopes in dimension ≤ 4. *Discrete Mathematics*, 186:69–94, 1998.

[21] Phong Bui-Tuong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[22] F. W. Campbell and J. G. Robson. Application of fourier analysis to the visibility of gratings. *Journal of Physiology*, 197:551–566, 1968.

[23] Edwin Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, Salt Lake City, UT, December 1974.

[24] Yung-Yu Chuang, Dan B Goldman, Ke Colin Zheng, Brian Curless, David Salesin, and Richard Szeliski. Animating Pictures with Stochastic Motion Textures. *ACM Transactions on Graphics*, 24(3):1–8, 2005.

[25] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[26] Kristin J. Dana, Bram Van Ginneken, Shree K. Nayar, and Jan J. Koenderink. Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics*, 18(1):1–34, January 1999.

[27] Jeremy S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. In *SIGGRAPH*, pages 361–368, 1997.

[28] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[29] Jean-Michel Dischler, Karl Maritaud, Bruno Lévy, and Djamchid Ghazanfarpour. Texture particles. *Computer Graphics Forum*, 21(3), 2002.

[30] Yoshinori Dobashi, Tsuyoshi Yamamoto, and Tomoyuki Nishita. Synthesizing sound from turbulent field using sound textures for interactive fluid simulation. *Computer Graphics Forum*, 23(3):539–545, 2004.

[31] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, 2nd edition, 1998.

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

[33] Alexei A. Efros and Thomas K. Leung. Texture synthesis by non-parametric sampling. In *Proceedings of the International Conference on Computer Vision (ICCV)*, volume 2, pages 1033–1038, September 1999.

[34] Stefen Fangmeier. Industrial Light & Magic : The making of "The Perfect Storm". Special Session at SIGGRAPH 2000 (Talk), July 2000.

[35] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[36] Olivier Génevaux, Arash Habibi, and Jean-Michel Dischler. Simulating fluid-solid interaction. In *Graphics Interface*, pages 31–38. A K Peters, June 2003. ISBN 1-56881-207-8, ISSN 0713-5424.

[37] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[38] Rick Gurnsey and David J. Fleet. Texture space. *Vision Research*, 41(6):745–757, 2001.

[39] R. M. Haralick. Statistical and structural approaches to texture. *Proceedings of the IEEE*, 67:786–804, 1979.

[40] R. M. Haralick, K. Shanmugam, and I. Dinstein. Textural features for image classification. *IEEE Transactions on Systems, Man, and Cybernetics*, 3:610–621, 1973.

[41] J. K. Hawkins. *Picture Processing and Psychopictorics*, chapter Textural Properties for Pattern Recognition. Academic Press, New York, 1969.

[42] David J. Heeger and James R. Bergen. Pyramid-based texture analysis/synthesis. In *SIGGRAPH*, pages 229–238, 1995.

[43] Aaron Hertzman, Charles E. Jacobs, Nuria Oliver, Brian Curless, and David H. Salesin. Image analogies. In *SIGGRAPH*, pages 327–340, 2001.

[44] B. Julesz. Visual pattern discrimination. *IRE Transactions on Information Theory*, IT-8:84–92, 1962.

[45] B. Julesz. Textons, the elements of texture perception, and their interations. *Nature*, 290:91–97, 1981.

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

[47] Lydia E. Kavraki, Petr Svestka, Jean-Claude Latombe, and Mark H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.

[48] Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. Texture Optimization for Example-based Synthesis. *ACM Transactions on Graphics*, 24(3):1–8, 2005.

[49] 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):277–286, 2003.

[50] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Norwell, MA, 1991.

[51] Laurent Lefebvre and Pierre Poulin. Analysis and synthesis of structural textures. In *Graphics Interface*, pages 77–86, May 2000.

[52] A. Levin, A. Zomet, and Y. Weiss. Learning how to inpaint from global image statistics. In *Proceedings of the International Conference on Computer Vision (ICCV)*, pages 305–312, 2003.

[53] Ziqiang Liu, Ce Liu, Heung-Yeung Shum, and Yizhou Yu. Pattern-based texture metamorphosis. In *Pacific Graphics*, pages 184–191, 2002.

[54] Jitendra Malik, Serge Belongie, Jianbo Shi, and Thomas Leung. Textons, contours and regions: Cue integration in image segmentation. In *International Conference on Computer Vision (ICCV)*, volume 2, pages 918–925, September 1999.

[55] Jitendra Malik and Pietro Perona. Preattentive texture discrimination with early vision mechanisms. *Journal of the Optical Society of America*, 7(5):923–932, May 1990.

[56] Stephen R. Marschner, Eric P.F. Lafortune, Stephen H. Westin, Kenneth E. Torrance, and Donald P. Greenberg. Image-based brdf measurement. Technical Report PCG-99-1, Cornell University Program for Computer Graphics, January 1999.

[57] Michael D. McCool and Wolfgang Heidrich. Texture shaders. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 117–126, 1999.

[58] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming (revised). In *SIGGRAPH/Eurographics Graphics Hardware Workshop*, pages 57–68, 2002.

[59] Ann McNamara, Alan Chalmers, Tom Troscianko, and Iain Gilchrist. Comparing real & synthetic scenes using human judgements of lightness. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 207–218, 2000.

[60] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[61] L. Neumann, K. Matkovic, and W. Purgathofer. Perception based color image difference. In *Proceedings of Eurographics*, volume 17, pages 233–241, 1998.

[62] F. E. Nicodemus. Reflectance nomenclature and directional reflectance and emissivity. *Applied Optics*, 9:1474–1475, 1970.

[63] Darwyn R. Peachey. Solid texturing of complex surfaces. In *SIGGRAPH*, pages 279–286, 1985.

[64] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In *SIGGRAPH*, pages 425–432, 2000.

[65] Ken Perlin. An image synthesizer. In *SIGGRAPH*, pages 287–296, 1985.

[66] Ken Perlin and Eric M. Hoffert. Hypertexture. In *SIGGRAPH*, pages 253–262, 1989.

[67] E. Polak. *Computational Methods in Optimization.* Academic Press, New York, 1971.

[68] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, New York, NY, USA, 1992.

[69] L. G. Thorell R. L. DeValois, D. G. Albrecht. Spatial-frequency selectivity of cells in macaque visual cortex. *Vision Research*, 22:545–559, 1982.

[70] Arno Schödl, Richard Szeliski, David Salesin, and Irfan Essa. Video textures. In *SIGGRAPH*, pages 489–498, 2000.

[71] R. Siegal and J. Howell. *Thermal Radiation Heat Transfer*. Hemisphere Publishing, Washington, D.C., 3rd edition, 1992.

[72] J. Slansky. Image segmentation and feature extraction. *IEEE Transactions on Systems, Man, and Cybernetics*, 8:237–247, 1978.

[73] Wolfgang Stürzlinger. A generic interface to colors, materials, and textures. In *Compugraphics '96*, pages 192–200, 1996.

[74] Pei-hsiu Suen and Glenn Healey. The analysis and recognition of real-world textures in three dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 22(5):491–503, May 2000.

[75] H. S. Tamura, S. Mori, and Y. Yamawaki. Textural features corresponding to visual perception. *IEEE Transactions on Systems, Man, and Cybernetics*, 8:460–473, 1978.

[76] W.C. Thibault and B.F. Naylor. Set operations on polyhedra using binary space partitioning trees. In *SIGGRAPH*, pages 153–162, 1987.

[77] Greg Turk. Generating textures on arbitrary surfaces using reaction-diffusion. In *SIGGRAPH*, pages 289–298, 1991.

[78] Alan Watt and Fabio Policarpo. *The Computer Image*. Addison Wesley Longman, 1998.

[79] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In *SIGGRAPH*, pages 479–488, 2000.

[80] Eric W. Weisstein. Simplex. From MathWorld–A Wolfram Web Resouonce. `http://mathworld.wolfram.com/Simplex.html`.

[81] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.

[82] L. Williams. Pyramidal parametrics. In *SIGGRAPH*, pages 1–11, 1983.

[83] Andrew Witkin and Michael Kass. Reaction-diffusion textures. In *SIGGRAPH*, pages 299–308, 1991.

[84] Jingdan Zhang, Kun Zhou, Luiz Velho, Baining Guo, and Heung-Yeung Shum. Synthesis of progressively-variant textures on arbitrary surfaces. *ACM Transactions on Graphics*, 22(3):295–302, 2003.

[85] S. W. Zucker and K. Kant. Multiple-level representations for texture discrimination. In *Proceedings of the IEEE Conference on Pattern Recognition and Image Processing*, pages 609–614, 1981.