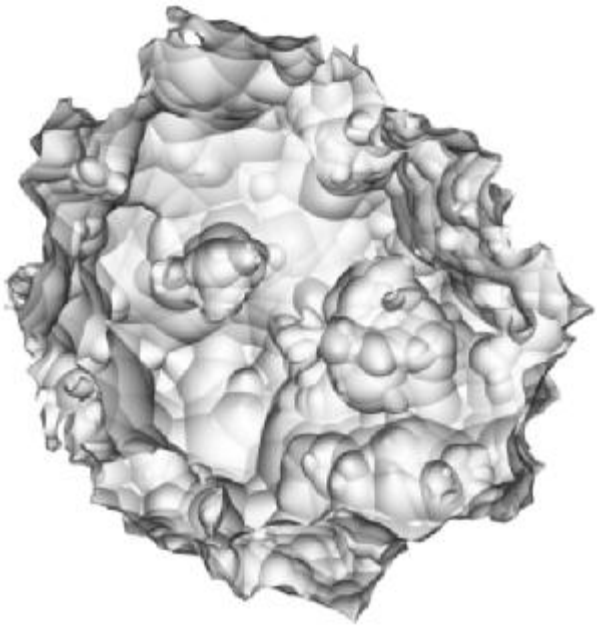
Department of Computer Science

Techniques for Stochastic Implicit Surface Modelling and Rendering

Manuel Noronha Gamito

A thesis submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy



JULY 2009

Para a minha filha Beatriz, com o amor do pai.

ABSTRACT

Implicit surfaces are a powerful shape primitive for computer graphics. This thesis focuses on a shape modelling approach which generates synthetic shapes by the specification of an implicit surface generating function that has random properties. This type of graphic object can be called a *stochastic implicit surface* because the surface is perceived as the realisation of a stochastic process. The main contributions of this thesis are in the form of new and improved modelling and rendering algorithms to deal with stochastic implicit surfaces that can be complex and feature fractal scaling properties.

On the modelling side, a new topological correction algorithm is proposed to detect disconnected surface parts that arise as a consequence of the implicit surface representation. A surface deformation algorithm, based on advection through a vector field, is also presented.

On the rendering side, several algorithms are proposed. First, an improved ray casting method is presented that guarantees correct intersections between view rays and the surface. Second, a new progressive refinement rendering algorithm is proposed that provides a dynamic rendering environment where the image quality steadily increases with time. Third, a distributed rendering mechanism is presented to deal with the long computation times involved in the image synthesis of stochastic implicit surfaces.

An application of the proposed techniques is given in the context of the procedural modelling of a planet. A procedural planet model must be able to generate synthetic planets showing the correct range of geological scales. The planet is generated as a stochastic implicit surface. This represents an improvement over previous models that generated planets as displacement maps over a sphere. Terrain features that were previously difficult to model can be achieved through the implicit surface approach. This approach is a generalisation over those previous models since displacement maps over the sphere can be recast as implicit surfaces.

ACKNOWLEDGEMENTS

I would like, first of all, to acknowledge my supervisor, Steve Maddock, for the constant support he provided during the four years that it took to reach this thesis. I could not have asked for a better supervisor than him with his willingness to listen and to provide advice while still giving me enough independence to follow my own ideas. For all of this and more – thank you.

F. Kenton Musgrave is someone I feel I should also thank. He gave me the opportunity to work with him for a year in the US and I learned a lot about procedural models and procedural terrain models, in particular, through his supervision. Much of the knowledge I gained with him filtered through into this thesis. Our friendship has suffered of late due to a difference of opinion over one of his papers but that does not detract from the fact that I have much to thank him for.

I would also like to remember my colleagues in the Computer Graphics group for their friendship: Örn Gunnarson, Oscar Martinez Lazalde, Michael Meredith, Arturo and Jorge Arroyo Palacios, Paul Richmond, Ahmed bin Subaih, and Miguel Salas Zúñiga. I will always remember with fondness the time I spent with them in the group.

On a more personal note, I would like to thank my wife Jane simply for being my wife despite my peculiar work schedule and my sometimes monomaniac attention to my research. When I decided to leave Portugal to pursue research in the UK, I did not imagine that I would also find a wife and gain a daughter in the process. For those two reasons plus this thesis, I am very fortunate that I made the right decision.

RESEARCH PUBLICATIONS

Book Chapters

Gamito, M.N., Maddock, S.C., A Progressive Refinement Approach for the Visualisation of Implicit Surfaces, volume 4 of *Communications in Computer and Information Science*, chapter 6, pages 93–108, Springer, 2007. (Revised selected paper from the GRAPP 2006 Conference)

Refereed Journal Papers

Gamito, M.N., Maddock, S.C., Topological Correction of Hypertextured Implicit Surfaces for Ray Casting, *The Visual Computer*, 24(6):397–409, June 2008. (Revised selected paper from the SMI '07 Conference)

Gamito, M.N., Maddock, S.C., Progressive Refinement Rendering of Implicit Surfaces, *Computers & Graphics*, 31(5):698–709, October 2007.

Gamito, M.N., Maddock, S.C., Ray Casting Implicit Fractal Surfaces with Reduced Affine Arithmetic, *The Visual Computer*, 23(3):155–165, March 2007.

Gamito, M.N., Maddock, S.C., Anti-aliasing with Stratified B-spline Filters of Arbitrary Degree, *Computer Graphics Forum*, 25(2):163–172, June 2006.

Refereed Conference Papers

Gamito, M.N., Maddock, S.C., Localised Topology Correction for Hypertextured Terrains. In I. Lim and W. Tang, editors, *Theory and Practice of Computer Graphics 2008*, Eurographics UK Chapter Proceedings, pages 91–98. The Eurographics Association, 9th–11th June 2008, Manchester, U.K. (Best student paper of the conference for technical content)

Gamito, M.N., Maddock, S.C., Topological Correction of Hypertextured Implicit Surfaces for Ray Casting. In E. Galin, M. Pauly, B. Wyvill, editors, *IEEE International Conference on Shape Modelling and Applications (Proc. SMI '07)*, pages 103–112. IEEE Press, 13th–15th June 2007, Lyon, France.

Gamito, M.N., Maddock, S.C., A Progressive Refinement Approach for the Visualisation of Implicit Surfaces. In J. Braz, J.A. Jorge, M.S. Dias, A. Marcos, editors, *International Conference on Computer Graphics Theory and Applications (GRAPP 2006 Proceedings)*, pages 26–33, INSTICC - Institute for Systems and Technologies of Information, Control and Communication, 25th–28th February 2006, Setúbal, Portugal.

Technical Reports

Gamito, M.N., Maddock, S.C., Grid Computing Techniques for Synthetic Procedural Planets, Dept. of Computer Science, The University of Sheffield, Research Memorandum CS–07–01.

Gamito, M.N., Maddock, S.C., A Progressive Refinement Approach for the Visualisation of Implicit Surfaces, Dept. of Computer Science, The University of Sheffield, Research Memorandum CS–05–11.

Gamito, M.N., Maddock, S.C., Ray Casting Implicit Procedural Noises with Reduced Affine Arithmetic, Dept. of Computer Science, The University of Sheffield, Research Memorandum CS–05–04.

Gamito, M.N., Maddock, S.C., Anti-aliasing with Stratified B-spline Filters of Arbitrary Order, Dept. of Computer Science, The University of Sheffield, Research Memorandum CS–05–02.

Other Publications

Gamito, M.N., Maddock, S.C, Photorealistic Rendering of Synthetic Procedural Planets, The White Rose Grid e-Science Centre brochure, pages 24–25. Universities of Leeds, Sheffield and York, April 2008.

Gamito, M.N., Photorealistic Rendering of Synthetic Procedural Planets. Poster presented at the 5th UK e-Science All Hands Meeting, 18th–21st September 2006, Nottingham, U.K.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

THE long standing approach of using large polygonal models to represent surfaces of arbitrary complexity is increasingly being challenged within the computer graphics community. Polygonal models constitute piecewise linear surfaces. The approximate representation of a smooth but otherwise very detailed surface with such models requires that a large number of polygons be used. This not only leads to memory storage problems but also implies the cumbersome task of dealing with the connectivity information between the faces and edges of a polygonal mesh. One alternative to polygonal models is parametric patches that have a compact representation and are intrinsically smooth [Farin, 2001]. Such parametric patch models find much application in the area of computer-aided design. However, their parametric nature makes their application difficult to topologically complex surfaces, e.g. surfaces with a high genus. Another, more recent, alternative to polygonal models is point-based models [Gross and Pfister, 2007]. These models do not need to store any connectivity information but still require that a very large number of point primitives be used to represent a surface such that no visual gaps between points are perceived. Yet another surface representation model, and the main focus of this thesis, is implicit surfaces, which can easily represent sets of surfaces with complex topology that may undergo blending and separation over time [Bloomenthal et al., 1997]. Implicit surfaces were chosen for the topic of this thesis because of their simplicity and compactness when dealing with irregular surfaces that may even have a fractal dimension.

Although the implicit surface model can generate surfaces with any kind of topology, it does not offer explicit control over the resulting surface topology. This is in contrast with polygonal meshes or parametric patches, for example, where the topology is entirely dictated by the way the polygons or patches are sewn together. This lack of topology control may be desirable or not, depending on the context where implicit surfaces are used. As an example, when implicit surfaces are used as part of the level set method for the simulation of fluids, the surface is allowed to separate into different parts to simulate the splashing behaviour of the free surface of the fluid [Foster and Fedkiw, 2001]. In contrast, when implicit surfaces are used to model solid objects, the existence of separate surface parts is not desirable as it leads to the unrealistic situation of an object with clouds of smaller objects surrounding it. In practical solid modelling situations where implicit surfaces are used, this often places constraints on the amount of detail that is possible to introduce into the surface of a solid without it breaking apart into separate pieces. Other types of topology control, such as constraining the genus of the surface, may also be desirable in situations where the surface may not be allowed to contain any holes[1].

[1] One example could be the modelling of a rugged container that must hold water.

Another drawback of modelling complex surfaces with an implicit representation is the rendering cost that is incurred for their correct visualisation. The possibility exists of first converting the surfaces into a polygonal mesh but this is not pursued here because of the already mentioned problems of polygonal meshes. The main alternative is to directly render implicit surfaces with a ray casting approach. Ray casting is a subset of the more general ray tracing algorithm and involves casting rays from the viewpoint and finding their intersections with a surface in order to obtain its image projection [Roth, 1982]. The cost of ray casting implicit surfaces comes from the need to find the correct intersection point between every view ray and the surface and this cost tends to increase when the complexity of the surface also increases. In the end, the computational cost of visualising complex implicit surfaces with ray casting is offset by the high quality of the rendering results, which allow the surface to be viewed from any viewpoint and with any level of detail without any objectionable rendering artifacts being present. With the help of ray casting algorithms, implicit surfaces with fractal detail can be visualised in the context of photorealistic rendering methods such as ray tracing or other more advanced global illumination methods [Glassner, 1989; Dutré et al., 2006].

1.1 Aims

The main aim of this thesis is to investigate techniques that allow stochastic surfaces to be modelled and rendered with an implicit surface representation. Stochastic surfaces are governed by a probabilistic model and typically contain very high levels of detail. Stochastic surfaces are ideal to represent many natural objects due to the ease with which it is possible to generate intricate surface detail out of simple probabilistic rules [Fournier, 1980]. This principle has been termed *data amplification* by Smith [1984] – the probabilistic production rules are used to amplify a small initial set of parameters into a seemingly infinite amount of detail. Stochastic surfaces can be used to model materials such as water, glass, wood, eroded rock and, on a larger scale, entire planets with oceans and continents.

The representation of stochastic surfaces in implicit form is the most flexible for generating surface detail. This representation leads to what may be called *stochastic implicit surfaces*. The first known example of the synthesis of stochastic implicit surfaces are the "fractal flakes" of Voss [1983]. Voss rendered an iso-surface of a fractional Brownian motion scalar field, generated with a spectral synthesis method. The continuous scalar field was interpolated from values at the vertices of a cubic grid. Generating an implicit surface based only on values stored on a discrete grid has many limitations, most of which have to do with lack of detail at small scales. Lewis [1989] improved on the modelling of stochastic implicit surfaces by proposing the use of *procedural noise functions* to introduce the stochastic detail. Procedural noise functions are one example of *procedural modelling*. A procedural model is generated by a procedure in a computer program. This procedure implements a mathematical function, which is responsible for describing the shape of the surface, but may also include additional programming constructs such as loops, recursive calls, or lookup tables. A procedural surface model is therefore a generalisation of a functional surface model.

The implicit surface representation does not impose any constraint on the topology of the surface. The only topological constraint that should be retained is that of *connectedness*, which allows solid shapes to be modelled in a plausible way. The flexibility provided by stochastic

implicit surfaces is in contrast to earlier stochastic models based on solid texturing or bump mapping that were not capable of introducing any geometric detail into the surface at all, creating only the illusion that such detail exists [Perlin, 1985]. The application of the displacement mapping technique to stochastic surfaces allows the creation of true surface detail but imposes limitations on the shape and the topology of the resulting surface [Musgrave, 2003d]. The technique through which implicit surfaces can become stochastic surfaces is called *hypertexturing* and forms the core of the algorithms proposed in this thesis [Perlin and Hoffert, 1989].

To summarise, this thesis investigates algorithms that are directed towards the following four general issues in the area of computer graphics modelling and rendering:

- The application of stochastic models to surfaces in order to generate shapes that resemble natural solid objects.
- The use of the implicit surface representation, combined with the hypertexturing technique, to generate stochastic surface details in the most flexible way.
- The generation of stochastic implicit surfaces that are simply connected.
- The efficient rendering of stochastic implicit surfaces with photorealistic quality.

The modelling of planets will be used throughout this thesis as an example of an advanced application of stochastic implicit surfaces, motivating many of the research results that have been obtained. Because of the extreme range of scales involved in modelling the entire surface of a planet, a procedural definition of the terrain is the most feasible. The distance to the camera can be passed into the terrain-generating procedure so that the appropriate level of detail is created for every point on the terrain.

An alternative technique to modelling planet terrains that has recently gained attention is the *geometry clipmap* [Losasso and Hoppe, 2004]. A geometry clipmap, as the name implies, is a clipped version of a mipmap that stores the terrain as a texture [Williams, 1983; Tanner et al., 1998]. A clipmap clips out portions of the mipmapped texture in such a way that distant parts of the terrain are represented with a smoothed low resolution polygonal mesh. This results in a dynamic level of detail mechanism for terrain visualisation and also reduces the memory size of the texture map that needs to be kept in memory. Geometry clipmaps can be implemented on the GPU and provide real time terrain visualisation [Asirvatham and Hoppe, 2005]. The geometry clipmap technique has been extended to spherical surfaces, allowing the representation of planets [Clasen and Hege, 2006]. One disadvantage of clipmaps is that relatively complex memory paging strategies need to be implemented to dynamically update the memory resident clipmap as the camera travels across the terrain. Predictive algorithms, based on the motion of the camera, are used to fetch memory pages before they are actually needed. Current implementations of clipmaps also impose a maximum terrain resolution and can only be used for terrains represented in the form of heightfields. Procedural terrain models, by comparison, are conceptually simple, are resolution independent, can represent terrains with overhanging features, require simple algorithms for ray casting and can incorporate dynamic level of detail. Although the ray casting of procedural terrains can be expensive, it is possible to implement it easily on GPUs, multi-core CPUs and multi-processor machines.

1.2 Main Contributions

This thesis presents contributions in both the modelling and the rendering aspects of stochastic implicit surfaces. The contributions to implicit surface modelling seek to address the topology control problem of implicit surfaces when used to model solid objects, guaranteeing that a surface remains simply connected, i.e. it does not contain more than one separate part. The two contributions are:

- A new topology correction method, integrated in a ray casting algorithm, that can detect and remove all disconnected surface parts other than the main desired surface [Gamito and Maddock, 2007e, 2008b].
- An implicit surface deformation method based on the advection through vector fields. This method is guaranteed not to change the topology of the surface. The surface will remain simply connected if it was already so before the deformation was applied [Gamito and Musgrave, 2001].

The contributions to implicit surface rendering seek to address issues related to the photorealistic quality and the high computational cost of rendering with the ray casting algorithm. They are:

- An improved ray-surface intersection algorithm for ray casting that can handle any type of implicit surface and that is shown to be faster than any previous intersection algorithms for general implicit surfaces [Gamito and Maddock, 2007d].
- An improved anti-aliasing and motion blur technique for ray casting that employs a B-spline basis function of any desired degree as the anti-aliasing filter [Gamito and Maddock, 2006a].
- A new progressive refinement rendering method for implicit surfaces that can provide an early estimate of the final surface visualisation without having to wait for the full rendering to complete. This progressive refinement method does not offer photorealistic quality but can be used during the design stage of a new surface [Gamito and Maddock, 2006b, 2007b,c].
- A distributed rendering tool to offload the computational cost involved in the photorealistic rendering of implicit surfaces onto a network of high performance computing platforms [Gamito and Maddock, 2007a].

These four contributions in the rendering area provide a set of tools that reduce the time required to design and subsequently render stochastic implicit surfaces without sacrificing surface complexity or photorealistic image quality.

1.3 Chapter Overview

This thesis contains ten chapters and three appendices. Chapters 2, 3 and 4 provide an overview of essential material. The main research results are described in the subsequent chapters.

Most of the main research chapters expand upon papers that have been presented by the author in peer reviewed conferences and journals. The only exception appears after the main chapters, in Appendix C, where a slightly expanded version of a departmental research memo is given. Because this thesis deals with implicit surfaces that are directly rendered, rather than being approximated by a polygonal mesh, the modelling aspects that have been researched are inextricably linked to the rendering aspects that were also the focus of research. This has led to a layout for this thesis where the modelling related chapters are presented after the chapters dealing with rendering issues.

Chapter 2 gives an introduction to implicit surfaces. It also explains how implicit surfaces can be used to model terrain covering an entire planet. The representation of terrain with implicit surfaces allows the modelling of such terrain features as overhangs and caves in a simple manner. This type of terrain features were difficult to model with previous terrain representations based on displacement maps.

Chapter 3 provides a literature review of stochastic surface modelling techniques. Many of these techniques can be defined procedurally and applied within the framework of implicit surfaces, leading to the development of *stochastic implicit surfaces*.

Procedural noise functions are the topic of Chapter 4. These functions are a basic building block for stochastic modelling when attempting to recreate natural features such as terrain or clouds. A unified procedural noise function model is presented that encompasses all the noise functions that are used in this thesis.

Robust ray-surface intersection algorithms for ray casting are presented in Chapter 5. An intersection algorithm is said to be robust if it is guaranteed to always find the correct intersection point for every view ray. An efficient robust intersection algorithm is presented that works for any type of implicit surface [Gamito and Maddock, 2007d]. It is also shown that, if some extra conditions are imposed on the implicit surfaces to be rendered, it is possible to use even more efficient intersection algorithms.

Chapter 6 discusses anti-aliasing and motion blur techniques for ray casting. Anti-aliasing is especially important when rendering fractal implicit surfaces due to the large amount of surface details that are present. Without such anti-aliasing techniques, a computer animation of fractal implicit surfaces would be too noisy. An anti-aliasing and motion blur technique is presented that uses a B-spline basis function of any desired degree as the anti-aliasing filter [Gamito and Maddock, 2006a].

The progressive refinement rendering of implicit surfaces forms the topic of Chapter 7. Progressive refinement is an answer to the long rendering times necessary for ray casting complex surfaces as long as photorealistic rendering effects are not required. An interactive progressive refinement rendering method is presented where the user can steer the rendering by specifying image hot spots that should receive priority. This method can be used while building a new stochastic surface and provides a visualisation whose quality steadily improves with time [Gamito and Maddock, 2006b, 2007b,c].

Chapter 8 describes how Morse theory can be used to completely characterise the topology of an implicit surface that is twice differentiable. The location and interconnection of the surface critical points, as described by this theory, allows disconnected surface parts to be identified [Gamito and Maddock, 2007e, 2008b]. This is an important step when using implicit surfaces

to model planets as the terrain should be made of only one surface part. The ray casting algorithm can ignore intersection points that belong to disconnected parts and find subsequent intersections further along a view ray.

Due to efficiency reasons, topology correction with the help of Morse theory should be performed on surfaces that are relatively simple. Fractal surfaces, for example, have a very high probability of having surface parts break away due to the small size of the surface details. This makes topology correction on fractal surfaces impractical since it needs to examine a large number of possible disconnection points. Chapter 9 explains how fractal detail can be added on top of a relatively simple surface that has previously been topologically corrected. This is achieved by deforming the surface over an externally supplied vector field, which adds additional surface detail without changing the results of the topology correction procedure [Gamito and Musgrave, 2001].

Chapter 10 concludes this thesis by summarising the main research results that have been achieved and by describing future research directions in the areas of stochastic shape modelling with implicit surfaces and of photorealistic outdoor rendering.

Appendix A presents the derivation of the recursive equations for the integral of a B-spline of any order – the integral of a B-spline of order m can be expressed as a linear combination of the integrals of two B-splines of order $m - 1$. This is at the heart of the anti-aliasing algorithm of Chapter 6. The derivation is only shown in the Appendix because it is mathematically oriented and does not add useful detail to Chapter 6.

Appendix B lists the complete state equations for the dynamic systems that model the flow field surface deformation technique of Chapter 9. There are two such systems, represented as two sets of ordinary differential equations – the first describes the ray domain flow field deformation and the second gives the normal vectors of the deformed surface.

The ray casting algorithm is easily parallelisable due to the fact that each ray is cast independently. Given an image rendering task, this allows the straightforward distribution of the ray casting operations across the CPUs of a high performance computing (HPC) node. Algorithms such as ray casting are informally known as *embarrassingly parallel problems* due to facility with which they can be parallelised [Wikipedia, 2008]. Appendix C describes the development of a grid rendering tool that distributes the load of a computer animation of an implicit surface over a collection of geographically distant HPC nodes [Gamito and Maddock, 2007a]. The animation frames are distributed across the nodes and each frame is further partitioned across the CPUs of a node.

CHAPTER 2

Implicit Surface Models

An implicit surface is a set in the three-dimensional space of real numbers $\mathbb{R}^3$ defined by all points that are roots of some given function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$. The function f is the implicit surface generating function or, more compactly, the *implicit function*. In mathematical terms, the implicit surface is a set S such that:

$$S = \{\mathbf{x} \in \mathbb{R}^3 : f(\mathbf{x}) = 0\}. \tag{2.1}$$

Implicit surfaces can be known alternatively by the name of *level sets* in the computational physics community [Sethian, 1999]. This name has recently become popular in the computer graphics community also, due to their application in the modelling of fluids [Osher and Fedkiw, 2003]. The name *level set* is usually employed to signify an implicit surface whose shape changes over time in response to some dynamic system. Throughout this thesis the more general term of *implicit surface* will continue to be employed to name surfaces described by (2.1).

An implicit surface is one member of the larger family of *iso-surfaces* of the function f (see Figure 2.1 for an example of one such family of surfaces). An iso-surface of f is another set S_c defined relative to some constant parameter $c \in \mathbb{R}$ such that:

$$S_c = \{\mathbf{x} \in \mathbb{R}^3 : f(\mathbf{x}) = c\}. \tag{2.2}$$

By definition, $S = S_0$ and the implicit surface is simply the iso-surface of f for the case $c = 0$. Every iso-surface is also an implicit surface in its own right relative to the function $f - c$. Without loss of generality, therefore, one can concentrate on surfaces S that follow the definition given in (2.1). A more compact way of referring to an implicit surface is achieved by writing the inverse relation $f^{-1}(0)$ instead of using definition (2.1). The function f is non-injective because it generally maps an infinity of points in $\mathbb{R}^3$ to a single value of 0. Since f is non-injective, the inverse f^{-1} is not a function. It can be understood, however, in a set theoretic sense as the set of all points $\mathbf{x}$ for which $f(\mathbf{x}) = 0$. Similarly for iso-surfaces, it is possible to write $f^{-1}(c)$ as a more compact notation than the definition (2.2) to describe the iso-surface S_c.

This chapter provides a general overview of implicit surfaces. Section 2.1 begins by analysing some of the mathematical characteristics of implicit surfaces and how they relate to the implicit functions that generate such surfaces. Section 2.2 gives a classification of the several types of implicit surfaces (and implicit functions) that can be developed for computer graphics. Section 2.3 describes several common additional modelling techniques that can be applied on

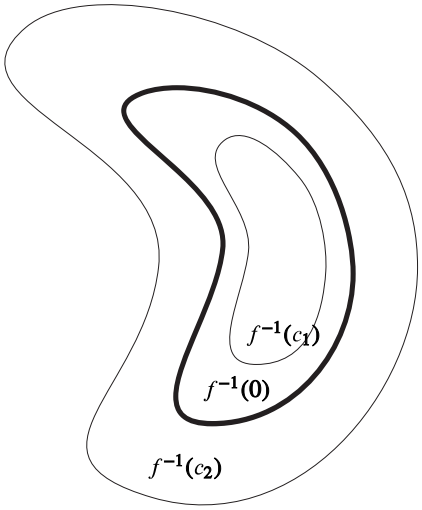


Figure 2.1: An implicit surface and two of the iso-surfaces with $c_1 < 0$ and $c_2 > 0$.

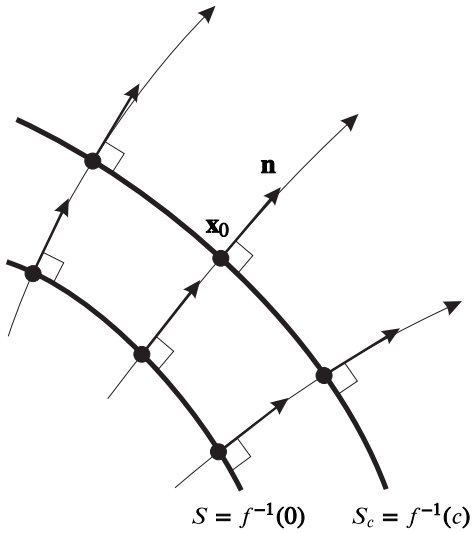


Figure 2.2: The normal vector field is everywhere orthogonal to the family of iso-surfaces.

implicit surfaces. Finally, Section 2.4 explains how hypertexturing (one of the modelling techniques presented in Section 2.3) can be used to generate procedural fractal terrains that take the form of implicit surfaces.

2.1 Continuity, Differentiability and Regularity

In this section an investigation is made of the requirements the implicit function f must obey in order to generate implicit surfaces that can be used for solid modelling in computer graphics. The first important requirement is that of *continuity*. The implicit surface is continuous if the implicit function is also continuous. Lack of continuity of f could sometimes result in surface cracks for S. Cracked surfaces or, more correctly stated, surfaces with boundaries can be visualised in computer graphics but they are regarded as degenerate surfaces since they cannot be used to enclose a volume. For surfaces with boundaries, it is not possible to define an "inside" or an "outside" of the surface, which precludes their use for solid modelling applications. Although not all discontinuous functions f will generate surfaces with boundaries, it is better to work with continuous functions as these are always guaranteed to generate correct volume enclosing surfaces.

The issue of *differentiability* of f is related to the definition of a surface normal vector for S. Such a normal vector is essential for the application of a lighting model to the surface. Consider the problem of finding the normal vector $\mathbf{n}(\mathbf{x}_0)$ for some point $\mathbf{x}_0$ on the surface. Consider also that the surface is differentiable in a neighbourhood $V(\mathbf{x}_0) \subset S$ of $\mathbf{x}_0$. To find $\mathbf{n}(\mathbf{x}_0)$, an arbitrary path $\mathbf{x}(t)$ is chosen such that $\mathbf{x}(t) \in V(\mathbf{x}_0)$ for every t and $\lim_{t\to\infty} \mathbf{x}(t) = \mathbf{x}_0$. The total derivative of f along the path is given by $Df(\mathbf{x}(t)) = \nabla f(\mathbf{x}(t)) \cdot \dot{\mathbf{x}}(t)$, where the derivative $\dot{\mathbf{x}}(t)$ is the tangent vector to the path at t. Since the path is on the surface, one has that $f(\mathbf{x}(t)) \equiv 0$ and, consequently, $Df(\mathbf{x}(t)) \equiv 0$. The gradient $\nabla f(\mathbf{x}(t))$, therefore, is orthogonal to the tangent

vector everywhere along the path and, in particular, in the limit when $\mathbf{x}(t) \to \mathbf{x}_0$. Because the path was chosen arbitrarily, every possible tangent vector to the surface at $\mathbf{x}_0$ is orthogonal to $\nabla f(\mathbf{x}_0)$ and this gradient vector, finally, is orthogonal to the implicit surface itself. The normal vector $\mathbf{n}$ is the unit length version of the gradient vector of f:

$$\mathbf{n} = \frac{\nabla f}{\|\nabla f\|}. \tag{2.3}$$

If the function is not differentiable at $\mathbf{x}_0$ then it is not possible to define a normal vector there since different paths $\mathbf{x}(t)$ will converge to different values of $\nabla f(\mathbf{x}_0)$ when $t \to \infty$. This situation occurs along ridges or over cusps of the implicit surface. A unique feature of implicit surfaces, when compared to other computer graphics primitives, is that it is possible to define normal vectors with equation (2.3) for points that are not on the surface. Figure 2.2 illustrates this property. The normal $\mathbf{n}$ for a point $\mathbf{x}_0 \notin S$ is the normal to the iso-surface S_c that passes through that point, which obeys $f(\mathbf{x}) = f(\mathbf{x}_0) = c$ and is given by $f^{-1}(f(\mathbf{x}_0))$. The normal vectors, calculated with (2.3), constitute a vector field whose streamlines are everywhere orthogonal to the iso-surfaces of f.

There remains the issue of what to do with such surface points $\mathbf{x}_0$ where f is differentiable but for which $\nabla f(\mathbf{x}_0) = \mathbf{0}$. If the gradient is equal to the null vector at $\mathbf{x}_0$, it is not possible to define a unit surface normal at that point, according to (2.3). A point $\mathbf{x}_0 \in S$ is said to be *regular* if the gradient $\nabla f(\mathbf{x}_0)$ is not null. If this happens for every point of $f^{-1}(0)$ then the implicit surface itself is said to be regular. The concept of regularity can be generalised to the family of iso-surfaces of f: a constant c is a *regular value* of f if the iso-surface $f^{-1}(c)$ is regular, with the same implied meaning as before that every point of $f^{-1}(c)$ is a regular point.

If f is smooth (meaning that it is continuous and infinitely differentiable) and furthermore $f^{-1}(0)$ is regular then it is possible to apply the Implicit Function Theorem and show that the implicit surface is a two-dimensional manifold or 2-manifold [Bruce and Giblin, 1992]. A 2-manifold is a surface without boundaries where, for every surface point, it is possible to find a sufficiently small neighbourhood in which the surface is locally approximated by a plane. In topological terms, the surface is locally homeomorphic to a plane for every $\mathbf{x}_0 \in S$ in the sense that it is possible to find a one-on-one mapping, inside a small vicinity of $\mathbf{x}_0$, between points on the surface and points on the plane. The Jourdan-Brower Separation Theorem then states that the surface separates the space into two open and disjoint spaces whose common boundary is the surface[1] [Montiel and Ros, 2005]. Furthermore, if the surface is bounded, the Brower-Samuelson Theorem states that the surface is also *orientable* [Montiel and Ros, 2005].

An orientable implicit surface has two well defined sides, each one associated with one of the two open spaces given by the Jourdan-Brower Separation Theorem. The normal vector $\mathbf{n}$ of an orientable implicit surface identifies in a consistent way one of the two sides of the surface. The gradient vector ∇f, by construction, always points in the direction of increasing values of f. Given that the surface is placed on the points for which $f(\mathbf{x}) = 0$, the gradient vector, and consequently the normal vector also, points into the region where $f(\mathbf{x}) > 0$. If one follows the computer graphics convention of having surface normals that point towards the outside of a surface, the space $f > 0$ is therefore associated with the outside of the implicit surface. This

[1]The Jourdan-Brower Separation Theorem applies to connected surfaces only but it is possible to apply it iteratively to every connected part of a surface if the surface is disconnected.

also means that all the iso-surfaces $f^{-1}(c)$ for which $c > 0$ are placed outside of the implicit surface. Similarly, the iso-surfaces for which $c < 0$ are all geometrically contained inside the implicit surface. In a more general way, given two contants $c_1 < c_2$, the iso-surface $f^{-1}(c_1)$ is contained inside the iso-surface $f^{-1}(c_2)$. Figure 2.1 shows how all the iso-surfaces, including the implicit surface, form a nested sequence for increasing values of c.

All the previous topological results about implicit surfaces were derived on the assumption that the surface $f^{-1}(0)$ is regular. This may seem like it could be a restrictive condition but fortunately it is not. Take any smooth implicit function f and consider the family $f^{-1}(c)$ of iso-surfaces generated from it. Sard's Theorem states that the set of constants $c \in \mathbb{R}$ for which the iso-surfaces $f^{-1}(c)$ are not regular is a null set of $\mathbb{R}$, meaning that it has a measure of zero in $\mathbb{R}$ [Bruce and Giblin, 1992]. Sard's Theorem implies that the probability of $f^{-1}(0)$ being a regular implicit surface is very high. Even if the implicit surface is not regular, it is always possible to find an iso-surface $f^{-1}(\epsilon)$, with ϵ arbitrarily small, that is regular and that can be used to approximate $f^{-1}(0)$ as closely as desired.

To summarise the results of this section, it has been seen that infinite differentiability of the implicit function (corresponding to a function that is C^∞ continuous) is required for the implicit surface to be a 2-manifold, separating an inside space from an outside space. For the purposes of this thesis this condition on f will be relaxed. Specifically, the functions that will be of interested must obey the following condition:

> The implicit function f must be continuous and differentiable. The first derivatives of f must also be continuous *almost everywhere*, i.e. the set of points over which the first derivatives of f are discontinuous must be a null set of $\mathbb{R}^3$.

It is easy to see that this continuity condition constitutes a significant relaxation over the C^∞ continuity condition. The first derivatives of the implicit function are allowed to be discontinuous provided that the set of points where that happens has a zero measure in $\mathbb{R}^3$. This includes countable sets of isolated points, lines and surfaces that are embedded in $\mathbb{R}^3$. The more relaxed continuity condition on f is necessary to create implicit surfaces with creases. Creased surfaces are generally considered as undesirable for shape modelling but they are very useful for the purposes of this thesis. When using this type of implicit surfaces to generate terrain, it becomes possible to model sharp terrain ridges. In practice, the relaxation of the C^∞ continuity condition does not appear to lead to incorrect results – the implicit surfaces that were generated for this thesis are still 2-manifolds. Most studies of surfaces in mathematical textbooks consider the case of smooth surfaces only, for which C^∞ continuity automatically holds, because the subsequent treatment of differential geometry concepts applies well over such surfaces[2]. Several variations of the Implicit Function Theorem, which relates the continuity of f to the existence of a 2-manifold, exist that relax the C^∞ condition (e.g. [Kumagai, 1980]). No attempt was made, however, to find a variation of the Implicit Function Theorem that would validate the continuity conditions used in this thesis.

[2]Differential geometry can also be applied to non-smooth surfaces, such as piecewise linear surfaces, but requires a more specialised treatment. For an example, see [Eastlick, 2006].

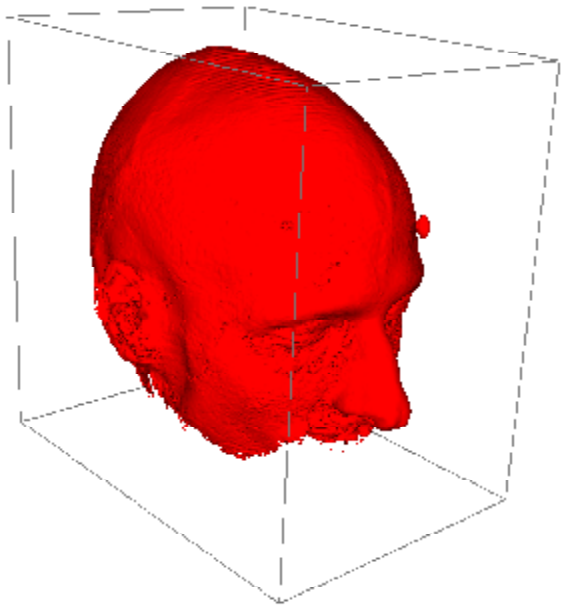


Figure 2.3: A head dataset visualised with the Marching Cubes algorithm (from the Wikimedia Commons, a repository of public domain multimedia content).

2.2 An Implicit Surface Classification

The following sections provide a classification of implicit surfaces. What essentially distinguishes different types of implicit surfaces is the way the implicit function f is built. Several characteristics can be taken into account when designing f such as surface smoothness, amount of detail generated, degree of control over the resulting surface or computational complexity. Only six approaches are presented, which are considered to be fundamental when building an implicit function. Many variations have been proposed relative to these fundamental techniques but those will not be covered here. To give one example, the case of interpolating implicit surfaces made with sums of radial basis functions can be seen as a variation of the blobby surfaces that are presented in Section 2.2.4 [Morse et al., 2001]. For a more in-depth introduction to implicit surfaces the reader is referred to Bloomenthal et al. [1997].

2.2.1 Discrete Surfaces

Discrete implicit surfaces result when the implicit function is defined by a set of samples that are placed over a regular cubic grid. Rather than a continuous three-dimensional function $f(\mathbf{x})$, one deals with a set $\{f_{ijk}\}$ where the triple index gives the coordinates of a sample in the grid. Since the function is only defined at a discrete set of samples, it is necessary to specify how the function behaves away from the samples. For each cube in the grid, the implicit function can be trilinearly interpolated from the sample values at the eight corners of the cube. Higher order interpolation schemes are possible inside a cube by taking into account other more distant samples, besides the ones at the cube's corners, but this is rarely used.

Discrete implicit functions arise naturally in the area of medical imaging, where CT and MRI scans result in sequences of two-dimensional discrete slices of the human body that, when stacked up, form a cubic grid. Identifying features such as bone or fat tissue in these datasets is possible either by direct volume rendering of the data or by the extraction of iso-surfaces. Many fields of science also produce scalar datasets defined over cubic grids, either through measurement or as the result of numerical simulations, which can be visualised as implicit surfaces. The image in Figure 2.3 shows an example of an implicit surface extracted from a medical dataset with the Marching Cubes algorithm [Lorensen and Cline, 1987]. It is possible to perceive, especially at the crown of the head, some artifacts in the polygonisation of the surface as a result of the discrete nature of the cubic grid.

An implicit surface can be modelled by the direct specification of implicit function samples over the points of a grid. This technique is often called *sculpting* since it has a strong analogy with the act of carving or sculpting an object out of a raw block of material. This idea was pioneered by Galyean and Hughes [1991], who presented an extension of the familiar two-dimensional digital paint techniques to a three-dimensional grid setting. Wang and Kaufman [1995] considered additional sculpting operations such as the act of scooping out a piece of material from the volume with a predefined tool or sawing along a user specified line. Sculpting an implicit surface out of a cubic grid is a memory expensive operation since a fine resolution is necessary, if the surface is to have a sufficient amount of detail. Ferley et al. [2001] introduced a dynamic adaptive level of detail mechanism to counteract this excessive memory usage. The adaptive mechanism allows the grid to refine itself in areas where detail is being introduced into the surface and, conversely, allows the grid to coarsen in areas where the surface is being smoothed. More recently, Bærentzen and Jørgen [2002] have proposed tools based on the level set method to introduce curvature dependent modifications on the gridded implicit surface.

2.2.2 Algebraic Surfaces

When the implicit function is a polynomial of some degree in $\mathbb{R}^3$, the resulting implicit surface is algebraic. The simplest case is that of a first degree polynomial:

$$f(\mathbf{x}) = \mathbf{n} \cdot \mathbf{x} + d = n_1 x_1 + n_2 x_2 + n_3 x_3 + d. \tag{2.4}$$

This is the equation for a plane with a uniform surface normal $\mathbf{n} = (n_1, n_2, n_3)$ and a signed distance to the origin that is equal to d. Second degree polynomials lead to quadric surfaces that include ellipsoids, paraboloids and hyperboloids among others. As an example, the implicit function for a sphere centred at the origin and with a unit radius is:

$$f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{x} - 1 = x_1^2 + x_2^2 + x_3^2 - 1. \tag{2.5}$$

Quadric surfaces can be generalised to *superquadrics* by raising the terms of the quadratic polynomial to fractional powers [Barr, 1981]. The previous example of a sphere generalises to a specific type of superquadrid given by the implicit function:

$$f(\mathbf{x}) = \left(x_1^{2/\varepsilon_2} + x_2^{2/\varepsilon_2}\right)^{\varepsilon_2/\varepsilon_1} + x_3^{2/\varepsilon_1} - 1, \tag{2.6}$$

where ε_1 and ε_2 are additional parameters that deform the surface of the superquadric relative to the surface of the sphere. The function (2.6) reduces to (2.5) when $\varepsilon_1 = \varepsilon_2 = 1$.

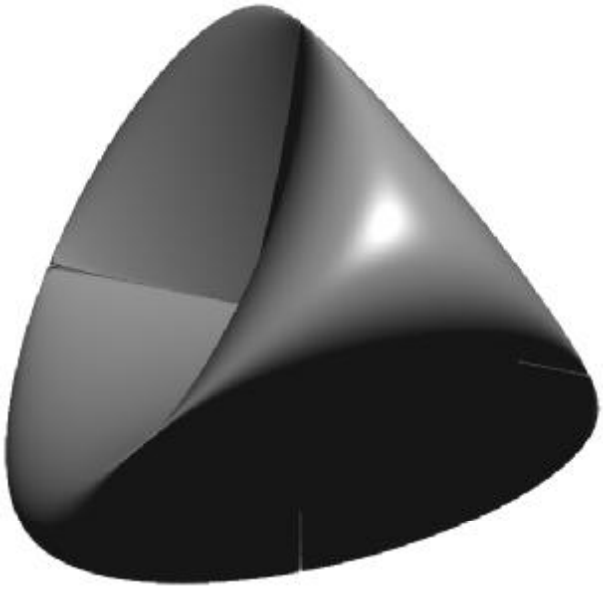


Figure 2.4: The Steiner Roman Surface (from the Wikimedia Commons, a repository of public domain multimedia content).

Algebraic surfaces provide simple recipes for generating basic graphics shapes such as spheres or cylinders but can also be used to generate complex shapes, some of which are not even manifolds. Figure 2.4 shows the Steiner Roman Surface, generated with the fourth degree polynomial:

$$f(\mathbf{x}) = x_1^2x_2^2 + x_2^2x_3^2 + x_3^2x_1^2 + x_1x_2x_3. \tag{2.7}$$

The surface is self-intersecting along the three coordinate axes and the origin is a triple point. Because of these self-intersections, the surface is not a 2-manifold since it cannot be parameterised as a plane in the vicinity of the coordinate axes. For the same reason, the surface is also not orientable – when following the surface from the outside one suddenly goes inside after crossing one of the lines of self-intersection. The Steiner Roman Surface is an example of an implicit surface that is not regular and for which the topological results of Section 2.1 do not apply. Rendering imperfections are visible in Figure 2.4 along the coordinate axes due to the fact that these are the places where the surface is irregular and where $\nabla f = \mathbf{0}$ occurs. Sard's Theorem, however, guarantees that any iso-surface $f^{-1}(\epsilon)$ of the function (2.7) with a very small ϵ will be a proper 2-manifold.

2.2.3 Piecewise Algebraic Surfaces

Piecewise algebraic surfaces were introduced by Sederberg [1985]. A tetrahedral partition of three-dimensional space is performed and each tetrahedron is assigned with its own implicit function. The implicit functions that are valid inside each tetrahedron are instances of a Bezier-Bernstein polynomial of some degree, being assigned with different polynomial coefficients, and the resulting surface is locally algebraic. The implicit function of degree d for one tetrahedron is given by:

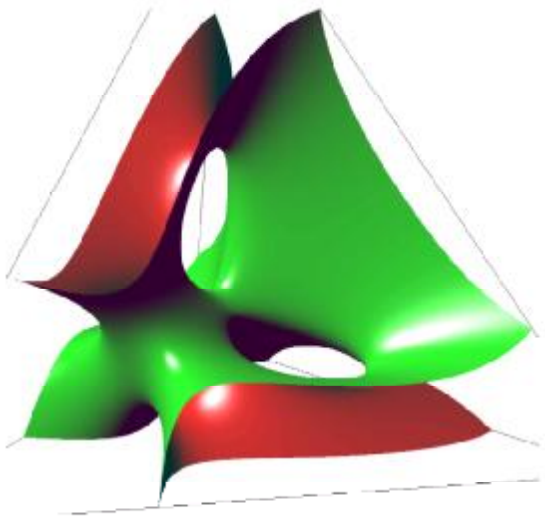


Figure 2.5: The Clebsch Cubic Surface. The edges of the tetrahedron that encloses the surface are displayed as grey lines (image courtesy of Charles Loop [Loop and Blinn, 2006]).

$$f(\mathbf{x}) = \sum_{i+j+k+l=d} b_{ijkl} \binom{d}{ijkl} r^i s^j t^k u^l, \tag{2.8}$$

where b_{ijkl} are the Bezier polynomial coefficients and r, s, t and u are the barycentric coordinates of the point $\mathbf{x}$ inside the tetrahedron. Surface continuity across tetrahedral patches is guaranteed by enforcing linear dependencies between the coefficients of adjacent tetrahedra, similar to what is done for parametric Bezier patches. A surface with C^1 continuity is achieved with cubic Bezier polynomials and requires 20 Bezier coefficients per tetrahedron [Bajaj et al., 1995b]. A surface with C^2 continuity is achieved with quintic Bezier polynomials and requires 56 Bezier coefficients per tetrahedron [Bajaj et al., 1995a].

Figure 2.5 shows the Clebsch Cubic Surface - a manifold with a complex topology expressed in piecewise algebraic form. The representation of this surface with parametric patches would have been cumbersome due to the difficulty of establishing a piecewise two-dimensional parametrisation of the surface. Piecewise algebraic implicit surfaces provide a much more flexible approach for the modelling of surfaces with arbitrary genus and, like their parametric counterparts, also benefit from the compact representation of a surface by a small set of Bezier coefficients. These characteristics make piecewise algebraic surfaces a useful tool for CAD.

2.2.4 Blobby Surfaces

Implicit surfaces have the ability to create surface blends between different objects. This is an important ability as it allows complex surfaces to be built by blending many simpler ones. Blending is basically achieved by summing together the contributions of the implicit functions from all the object primitives that comprise the surface. The simplest object primitives are points and blobby surfaces are the result of the additive blend of a set of points $\{\mathbf{x}_i\}$:

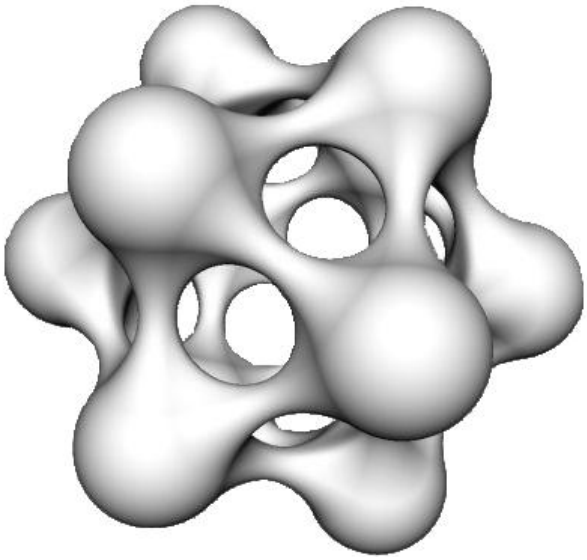


Figure 2.6: A soft object made by placing point primitives at the vertices of an icosahedron (image courtesy of Steve Hill).

$$f(\mathbf{x}) \;=\; T \;-\; \sum_i a_i h\big(\|\mathbf{x}-\mathbf{x}_i\|/r_i\big). \tag{2.9}$$

In the above expression, T is an overall threshold, responsible for making $f(\mathbf{x})$ positive on the outside of the blobby surface and negative on the inside. The function h is the blending function. The weight a_i expresses the contribution of the i-th point to the final surface. This weight can be negative, with the effect that the point carves out a piece from the surface. Finally, r_i is a dilation factor that can be used to enlarge or to shrink the area of influence of the point. To avoid having to use a square root when computing $\|\mathbf{x}-\mathbf{x}_i\|$, the blend function is commonly replaced according to $h(\|\mathbf{x}-\mathbf{x}_i\|/r_i) = g(\|\mathbf{x}-\mathbf{x}_i\|^2/r_i^2)$, where g is an equivalent blend function that works with squared distances.

Several authors have proposed blobby surfaces and only the definition of the blend functions h or g changes between proposals. Blinn [1982a] proposed *blobby models* that use an exponentially decaying blend function for g with infinite support. Nishimura et al. [1985] proposed *metaballs*, which use piecewise quadratic polynomials for h that are compactly supported. Wyvill et al. [1986] proposed *soft objects* where g is also compactly supported and takes the form of a cubic Hermite polynomial. Murakami and Ichihara [1987] use a compactly supported quadratic polynomial for g. Figure 2.6 shows one example of a soft object.

Compactly supported blend functions make the evaluation of (2.9) efficient because only a small number of point primitives needs to be considered in the neighbourhood of every point – any other primitives contribute with a value of zero to the summation in (2.9). The blobby models of Blinn with their exponential blends of infinite support, by contrast, require the summation in (2.9) to be carried out over all the primitives. If the number of primitives in the model is large, this can be a time consuming process. In practice, one must then establish a

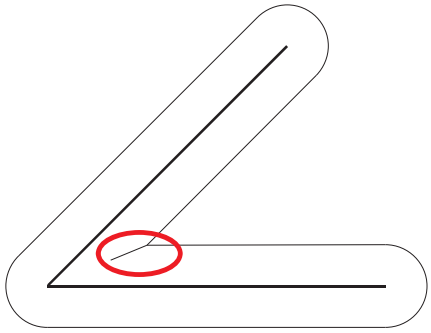


Figure 2.7: A schematic of a distance surface with a crease shown inside the red oval.

maximum distance for the primitives to be included in the summation, with the consequence that the surface representation will not be accurate. Blobby models, however, still have one advantage. Because the exponential blend function is C^∞ continuous, the resulting surface is guaranteed to be smooth. In the case of soft objects and metaballs, the blending functions are either cubic or quadratic polynomials that can only provide C^1 continuity between primitives. The consequence of this C^1 continuity is that, after the surface has been rendered, discontinuities in the gradient of the image intensity exist, which give rise to the *Mach banding* optical illusion – the perception of illusory dark bands centred around the intensity gradient discontinuities. Mach banding is present in Figure 2.6 in the form of dark bands along the seams between the blobs and the connecting arches. Hart et al. [1998] have proposed a compactly supported degree five polynomial to be used as a blending function that provides C^2 surface continuity and eliminates any shading artifacts[3].

2.2.5 Distance Surfaces

Distance surfaces are a generalisation of blobby surfaces that consider graphics primitives more complex than points [Bloomenthal and Wyvill, 1990]. The implicit function for a distance surface is:

$$f(\mathbf{x}) \;=\; T \;-\; \sum_i a_i h\big(d_i(\mathbf{x})/r_i\big). \tag{2.10}$$

All the parameters in the above expression have the same meaning as in (2.9). The novelty is the introduction of the distance function $d_i(\mathbf{x})$, which gives the distance from point $\mathbf{x}$ to the i-th primitive. This can be the standard Euclidean distance or any other function that has the semantics of a metric in $\mathbb{R}^3$. The type of primitives that are used in the blend (2.10) can be any graphics primitives for which distance functions are easy to compute. This includes straight line segments and polygons. The case of curvilinear primitives can also be considered, leading to the definition of *generalised cylinders*. The primitives in this case are either piecewise linear curves or parametric curves such as B-splines [Agin and Binford, 1976; Grimm, 1999].

[3]The blending function of Wyvill et al. [1986] is actually a polynomial of degree six (a cubic polynomial of a squared distance). However, because of the specific way this polynomial is constructed, it still cannot provide C^2 surface continuity.

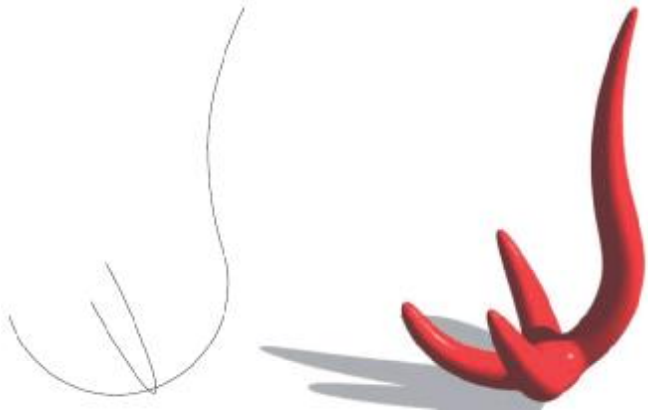


Figure 2.8: An example of a convolution surface generated from two intersecting curve primitives. The original primitives are shown on the left. (image courtesy of Xiaogang Jin [Jin and Tai, 2002])

Distance surfaces should not to be confused with *offset surfaces*, which are CAD surfaces that are offset by a fixed amount along their normal vectors. The two are equivalent only in the case when one single primitive is used so that no blending exists between primitives. One problem with distance surfaces is that they can show creases in areas of high curvature of a primitive. This is because the distance function is not C^1 continuous over points $\mathbf{x}$ that are equidistant to two or more near points on the primitive. Figure 2.7 illustrates this effect for the case of a generalised cylinder made from two joined line segments.

2.2.6 Convolution Surfaces

Convolution surfaces result from a different generalisation of blobby surfaces and solve the creasing problem of displacement surfaces. Irrespective of whether primitive i is a line, a curve, or a polygon, it is also a set of points S_i in three-dimensional space. A convolution surface is the result of the convolution of a blend function h over all the S_i sets:

$$f(\mathbf{x}) \;=\; T \;-\; \sum_i a_i \int_{S_i} h(\|\mathbf{x}-\mathbf{u}\|/r_i)\, d\mathbf{u}. \tag{2.11}$$

In a straightforward implementation of convolution surfaces, the integral in (2.11) must be approximated with numerical techniques. These techniques replace the integral by a finite sum over a sufficiently large number of points taken from the set S_i of each primitive. For this reason, convolution surfaces have traditionally been regarded as an expensive surface model. Figure 2.8 shows an example of a convolution surface, created by sweeping and accumulating the effect of a blending function along two curve primitives. The curve primitives themselves are made from a superposition of circular arcs. The varying thickness of the surface was obtained by modulating the blend function along the arclength of the curves. If this surface had been created as a distance surface instead, a bulging would have been visible around the point where the two curves intersect. Convolution surfaces can avoid this undesirable bulging effect [Bloomenthal, 1997].

In the initial proposal of convolution surfaces by Bloomenthal and Shoemake [1991], planar primitives were discretised onto a frame buffer in a pre-computation step. The convolution was then performed by applying a discrete convolution filter over the buffer with the blending function h as the convolution kernel. This makes the computation of convolution surfaces more efficient but also introduces some aliasing artifacts due to the fact that the primitives are being sampled over a regular grid of points. Sealy and Wyvill [1996] extended the technique of discrete convolution to three dimensions for non-planar primitives.

Sherstyuk [1999b] has been able to obtain closed-form analytic solutions to the convolution integral, for the case of primitives made with points, lines, triangles and circular arcs, by considering a specific type of blending function. Due to the distributive property of the convolution operator, a convolution surface made from linear combinations of these simple primitives can be obtained by linearly combining the convolution surfaces of each individual primitive. More recently, Jin and Tai [2002] also obtained closed-form solutions for surfaces made from line segments, circular arcs and quadratic B-spline curve primitives, using the compactly supported quartic polynomial of Murakami and Ichihara [1987] for the blend function h.

2.3 Additional Modelling Techniques

This section describes additional techniques for modelling implicit surfaces. These techniques work over surfaces that have been defined already with any one of the methods presented in Section 2.2. Additional techniques for colour texturing or animating implicit surfaces are not considered, however, as they fall outside the scope of this thesis. The techniques of constructive solid geometry (Section 2.3.2) and domain deformation (Section 2.3.3) can be combined in a hierarchical fashion with blended implicit surfaces (Sections 2.2.4 to 2.2.6) to form *Blob-Trees* [Wyvill et al., 1998]. A Blob-Tree is, therefore, a data structure that encodes an association between a set of implicit surfaces and additional modelling techniques to produce a final implicit surface. Of especial importance for this thesis is the technique of *hypertexturing* (Section 2.3.1), which adds additional small-scale detail to an otherwise smooth implicit surface. Hypertexturing forms the basis for the representation of implicit surfaces with fractal detail and, in particular, for the representation of procedural terrain.

2.3.1 Hypertexturing

Hypertexturing is a procedural technique proposed by Perlin and Hoffert [1989] that is used to add three-dimensional small-scale detail to the surface of smooth objects. It can model a large collection of materials such as fur, fire, glass, fluids and eroded rock. As a procedural modelling and texturing tool, hypertexturing can be regarded as an improvement over solid texturing [Perlin, 1985]. Using hypertextures, it becomes possible to actually deform the surface of an object instead of merely modifying its material shading properties. Hypertextures were initially presented as a technique to model *fuzzy objects* rather than implicit surfaces, i.e. objects such as a cloud that are represented by a density function and which do not have a well-defined boundary. Hypertextures, however, can also be applied to implicit surfaces.

In the original definition of hypertexture, an object is defined in three-dimensional space with

an *object density function* $f_0 : \mathbb{R}^3 \to [0, 1]$, which associates a density value $f_0(\mathbf{x})$ with every point $\mathbf{x}$ in space. A density of 0 means total transparency while a density of 1 means total opacity. The shape of the object can then be deformed through the composition of f_0 with one or more *density modulation functions* $f_i : [0, 1] \times \mathbb{R}^3 \to [0, 1],\ i = 1, \ldots n$ such that the final object density f is given by:

$$f(f_0(\mathbf{x}), \mathbf{x}) = f_n(\ldots f_2(f_1(f_0(\mathbf{x}), \mathbf{x}), \mathbf{x}), \ldots, \mathbf{x}). \tag{2.12}$$

The visualisation of a hypertextured fuzzy object is a volume rendering task. For every pixel, a ray must be marched by taking a sequence of small steps and accumulating density and opacity values along the way [Levoy, 1988]. This rendering method shows objects with a fuzzy appearance, hence the name of fuzzy objects.

It is possible to apply (2.12) similarly to implicit surfaces by considering f_0 to be an implicit function with the definition $f_0 : \mathbb{R}^3 \to \mathbb{R}$. After this change, $f(f_0(\mathbf{x}), \mathbf{x})$ also becomes an implicit function that generates a *hypertextured implicit surface*. The restriction that density modulation functions return opacity values in the range $[0, 1]$ is no longer necessary and one can define them as functions $f_i : \mathbb{R} \times \mathbb{R}^3 \to \mathbb{R},\ i = 1, \ldots n$. As long as all the modulation functions f_1 to f_n obey the continuity conditions set forth in Section 2.1, the surface is a proper 2-manifold where $f < 0$ is verified for points inside the surface and, correspondingly, $f > 0$ for points outside. Figure 2.9 shows an example of a hypertexture where an initial egg shape has been carved out with modulation functions.

2.3.2 Constructive Solid Geometry

Constructive solid geometry (CSG) is a solid modelling technique that builds large objects by performing a sequence of boolean operations on simpler primitives [Requicha and Voelcker, 1982]. Implicit surfaces can be used for CSG because, despite their name, they actually define solid volumes instead of surfaces (provided, of course, one deals with regular surfaces that generate manifolds). For CSG, it is only necessary to define the boolean operations of union and intersection – any other boolean operation can be expressed as a combination of these two. For two surfaces generated by the implicit functions f_1 and f_2, union and intersection can be expressed as new implicit functions $f_\cup$ and $f_\cap$, respectively, given by:

$$f_\cup(\mathbf{x}) = \min\big(f_1(\mathbf{x}), f_2(\mathbf{x})\big), \tag{2.13a}$$
$$f_\cap(\mathbf{x}) = \max\big(f_1(\mathbf{x}), f_2(\mathbf{x})\big). \tag{2.13b}$$

The implicit functions (2.13) introduce creases along the seams between the two original surfaces, which is generally considered as an undesirable effect. Ricci [1973] proposed new expressions for the two boolean operators that are analytic and that, therefore, do not generate creases:

$$f_\cup(\mathbf{x}) = \big(f_1^{-a}(\mathbf{x}) + f_2^{-a}(\mathbf{x})\big)^{-1/a}, \tag{2.14a}$$
$$f_\cap(\mathbf{x}) = \big(f_1^{a}(\mathbf{x}) + f_2^{a}(\mathbf{x})\big)^{1/a}. \tag{2.14b}$$

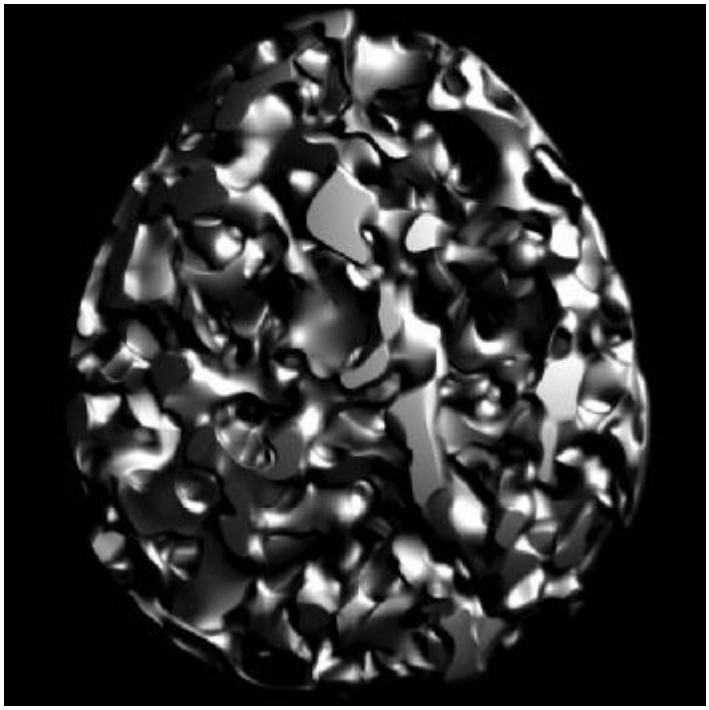


Figure 2.9: A hypertextured implicit surface (image courtesy of Prof. Ken Perlin [Ebert et al., 2003]).

The parameter $a > 0$ in (2.14) controls the amount of blending between the original surfaces. It is possible to show that, when $a \to \infty$, the functions (2.14) converge towards the functions (2.13). The CSG functions (2.14) can be straightforwardly generalised for use with three or more input implicit functions. The blends (2.14) still have the problem that they affect the entire surface of the primitives to a variable extent. Localised blends can be used instead, which only affect the primitives in a small neighbourhood around their intersections [Middleditch and Sears, 1985; Hoffmann and Hopcroft, 1987; Rockwood and Owen, 1987]. Barthe et al. [2004] have provided yet more powerful CSG operators where a user-defined curve is used to control the blend between objects. Figure 2.10 shows the use of the CSG operators of Barthe et al. to define an implicit surface from a set of simple surface primitives.

2.3.3 Domain Deformation

Domain deformation is used to introduce deformations in a surface by deforming its domain rather than the surface itself. Consider a domain mapping function $\mathbf{d} : \mathbb{R}^3 \to \mathbb{R}^3$. This function takes a point in space and maps it to a different point in the same space. The implicit function can now be evaluated after this domain mapping has been performed:

$$f(\mathbf{x}) = f_0(\mathbf{d}(\mathbf{x})). \tag{2.15}$$

Figure 2.10: A CSG implicit surface built from simple primitives (image courtesy of Loïc Barthe [Barthe et al., 2004]).

where f_0 is the implicit function of the original undeformed surface. The normal vector of the deformed surface is given by $\mathbf{n}(\mathbf{x}) = \mathcal{J}\{\mathbf{d}\}^{-T}\mathbf{n}_0(\mathbf{x})$ followed by vector normalisation, where $\mathcal{J}\{\mathbf{d}\}$ is the Jacobian matrix of the domain mapping function $\mathbf{d}$ evaluated at $\mathbf{x}$ and $\mathbf{n}_0(\mathbf{x})$ is the normal of the undeformed surface. For rendering purposes, it is preferable if the domain mapping function can be easily invertible. The reason why this is so will be explained in Chapter 9. It also helps if a simple analytic expression exists for the inverse transpose Jacobian matrix. Barr [1984] proposed three deformations, named *taper*, *twist* and *bend*, that obey these requisites. Figure 2.11 shows three blended implicit cylinders that have been deformed with the twist deformation. The technique of free-form deformation can also be applied to implicit surfaces with little difficulty [Sederberg and Parry, 1986]. In the case of a free-form deformation, the domain mapping function d is expressed as a trivariate Bezier-Bernstein polynomial, defined by a network of control points in space.

2.3.4 Metamorphosis

Metamorphosis, also called *morphing*, generates one surface that is intermediate between two other surfaces. In the simplest case, an in-between surface can be achieved by linearly interpolating between the implicit functions f_1 and f_2 of two surfaces:

$$f(\mathbf{x}, t) = (1-t)f_0(\mathbf{x}) + tf_1(\mathbf{x}). \tag{2.16}$$

The new surface is given by the implicit function $f(\mathbf{x}, t)$, where $0 \leqslant t \leqslant 1$ is a blending factor that controls the percentage of each of the original surfaces that is present in the final surface.

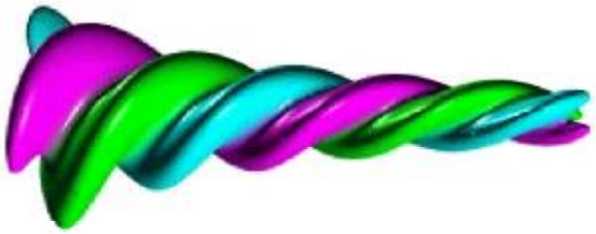


Figure 2.11: Three blended and twisted cylinders (image courtesy of Prof. Brian Wyvill [Wyvill et al., 1999]).

The interpolation expressed by (2.16) is rather simplistic and more sophisticated interpolation schemes have been presented.

For discrete surfaces (see Section 2.2.1), Hughes [1992] proposed doing metamorphosis in frequency space, by computing the Fourier transforms of the discretised implicit functions and interpolating their Fourier coefficients. This is followed by an inverse Fourier transform to obtain the interpolated surface. The technique was later improved by replacing the Fourier transform with the wavelet transform [He et al., 1994]. Interpolation between gridded implicit functions can also be performed according to a dynamic system that is governed by a level set equation [Breen and Whitaker, 2001].

For surfaces that are generated from geometric primitives (see Sections 2.2.4 to 2.2.6) a matching can be established between primitives of the first surface that should interpolate with primitives of the second surface [Wyvill, 1993]. Often, however, a precise matching cannot be established such as, for example, when the two surfaces have a different number of primitives. When the primitives consist of convex polygons, Minkowsky sums can be used to generate a family of intermediate polygons after polygon matching has been performed (matching polygons don't need to have the same number of sides and can be degenerate polygons in one and two dimensions, i.e. points and line segments) [Galin and Akkouche, 1996a]. The intermediate polygons then generate the intermediate surfaces. This technique was improved by introducing a blend graph and controlling the trajectory of the interpolated polygons, between starting and ending polygon, through user-specified curves [Galin and Akkouche, 1996b]. Figure 2.12 shows an example of a sphere transforming into a propeller shape.

As already mentioned, the primitive matching problem is very often ill-defined and several attempts to solve it have been proposed. An automatic matching technique was presented by Pasko and Savchenko [1995] for implicit surfaces generated through the combination of R-functions with CSG operators. Galin et al. [1999] presented semi-automatic procedures for matching between pairs of Blob-Trees. For blobby surfaces (Section 2.2.4), the matching problem can be solved by creating a single implicit surface in a hyper-dimensional space [Turk and O'Brien, 1999]. This space is made from the three spatial coordinates plus an extra coordinate for the blending factor. The first and last surfaces are specified with constraint points through which the hyper-surface must pass. By sweeping an $\mathbb{R}^3$ volume through the four-dimensional implicit surface, one obtains a sequence of implicit surfaces that interpolate between the first and the last.

Figure 2.12: Metamorphosis between two implicit surfaces (image courtesy of Eric Galin [Galin et al., 1999]).

2.4 An Implicit Surface Representation for Terrain

Terrain in computer graphics has traditionally been represented as a heightfield. The heightfield is a function $x_3 = f(x_1, x_2)$, where the coordinate x_3 is assumed to be taken along the vertical direction [Marshall et al., 1980]. The heightfield can either be a procedural model or it can be discretised over a grid as a digital terrain map – a triangular mesh being used to connect the elevation samples. The use of a heightfield imposes restrictions on the shape of the terrain. Specifically, the terrain cannot feature overhangs, arches or caves because for each (x_1, x_2) pair only one elevation value x_3 can be specified. Although the majority of real terrains can be expressed as heightfields, terrain overhangs do exist. Figure 2.13 shows a photograph of a real terrain with an overhang. The ability to model terrain elements such as overhangs and arches in a procedural way is a feature that a flexible terrain modelling system should have.

Hypertexturing can add overhangs, arches and caves to a terrain that is expressed as an implicit surface. Figure 2.14 compares the ability of the heightfield and the hypertexturing techniques for adding detail to an otherwise flat terrain. Hypertexturing can significantly increase the complexity of a surface's topology, giving it an arbitrarily high genus. In simple terms, this means that hypertexturing can carve an arbitrarily high number of holes in the surface. The ability of hypertexturing to carve holes through a surface is evident in the hypertextured surface of Figure 2.9. Hypertexturing, therefore, is a flexible way of specifying terrain shapes and it can also specify heightfields as a particular case. One drawback of this flexibility is that a surface can also become fragmented into several disconnected parts (this is illustrated in Figure 2.14(b)). A disconnected surface part leads to non-sensical effects for terrain modelling as it causes floating pieces of rock to appear. The problem of disconnected surface parts will be ignored until Chapter 8, where a solution is presented. Another drawback of hypertextured implicit surfaces is that they can take a long time to render, especially if the hypertextured detail is very small. This issue will be addressed in Chapters 5 and 7.

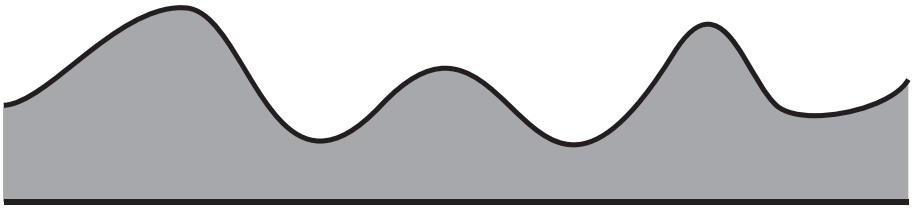
The idea of adding additional detail to an implicit surface was also considered by Sclaroff and Pentland [1991]. In their method, surface detail is added in the form of a displacement

Figure 2.13: A real terrain with an overhang caused by a slow process of wind erosion.

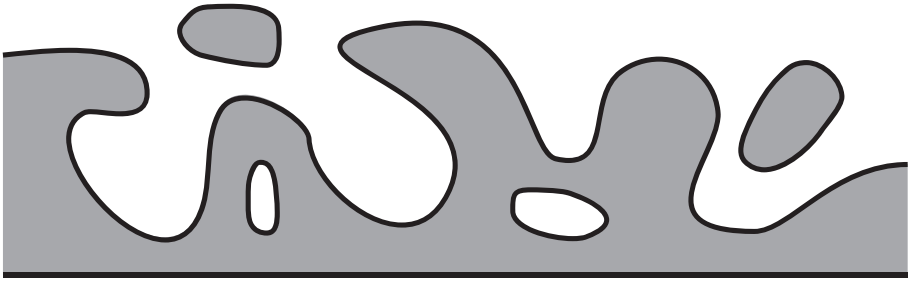
map. Their technique requires that the initial implicit surface be first parameterised so that a two-dimensional map, containing the displacement data, can be indexed. This poses restrictions on the implicit surfaces that can be used since they need to be easily parameterised. Furthermore, the method of Sclaroff and Pentland cannot generate overhangs, given that it takes the form of a displacement map.

Pedersen [1994] generalised the method of Sclaroff and Pentland by immersing the surface in a vector field, which the author calls a *flow map*, and letting this vector field do the deformation. The method of Pedersen, however, still works over surfaces that must be parameterised because the amount of vector deformation is governed by a two-dimensional map, defined over the original surface. The vector field deformation establishes a diffeomorphism between the modified surface and the original surface, meaning that the two surfaces are topologically equivalent. If the original surface has no holes, for example, the modified surface will not have any holes either. The method of Pedersen, therefore, can generate overhangs but cannot generate arches because the latter imply a topology change in the surface. The issue of vector field deformation will be considered in Chapter 9 where a ray casting technique for surfaces that underwent vector field deformation is presented. The vector field deformation technique does not require surface parametrisation and can be used to add additional surface detail on top of the surface detail that is created with hypertexturing[4].

[4]In fairness, the requirement that implicit surfaces be parameterisable in both the methods of Sclaroff and Pentland and of Pedersen is not strictly necessary. Rather than define a two-dimensional map over the surface of an



(a) Heightfield



(b) Hypertexture

Figure 2.14: Comparison between the heightfield and hypertexturing techniques.

2.4.1 Evaluating Hypertexture Functions, Gradients and Hessians

The hypertexturing technique of Perlin and Hoffert [1989] that was presented in Section 2.3.1 is reformulated here in a different way. Rather than use a chain of modulation functions as in (2.12), hypertexturing of an implicit function f_0 is obtained through the evaluation of a hierarchy of modulation functions. Figure 2.15 illustrates a simple hypertexture hierarchy. The function at the root of the hierarchy produces the final value for the implicit function $f(f_0(\mathbf{x}), \mathbf{x})$ of the hypertextured surface. The functions at the leaves of the hierarchy are the *source functions*, being defined as $f_i : \mathbb{R}^3 \rightarrow \mathbb{R}$ and written as $f_i(\mathbf{x})$. These functions depends only on the position $\mathbf{x}$ in space where the hypertexture is evaluated. A special source function is the *seed function* f_0, which describes the shape of the original implicit surface before any hypertexturing is applied. In Chapter 4, useful source functions that can be used for hypertexturing will be studied. All other functions in the upper levels of the hierarchy are the modulation functions, which can take an arbitrary number of arguments from lower functions. The definition of a modulation function is $f_i : \mathbb{R}^{m_i} \rightarrow \mathbb{R}$ where m_i is the number of arguments for function f_i. The function is written as $f_i(y_1, y_2, \ldots, y_{m_i})$. Notice that modulation functions do not have an explicit dependence on the position $\mathbf{x}$ in space.

Finding the normal vector of a hypertextured surface requires computing the gradient of the implicit function $f(f_0(\mathbf{x}), \mathbf{x})$ by application of the chain rule of differentiation. The gradient of the implicit function is best expressed in recursive form:

object, one can define a three-dimensional map, which is meant to be evaluated only on the surface of the object, much like a solid texture. This removes the need for surface parametrisation. Some additional coding would be required in both methods to remove the parametrisation restriction, which none of the authors presented.

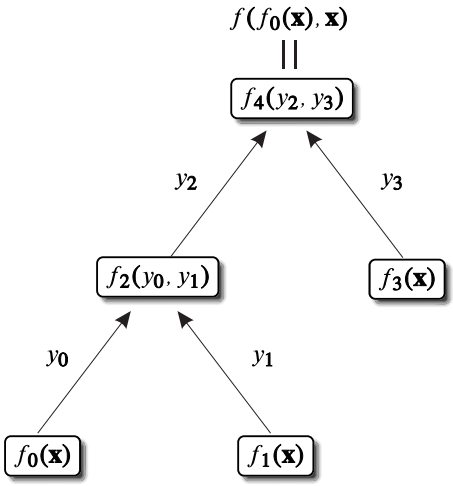


Figure 2.15: A simple hypertexture hierarchy.

$$\nabla f_i(y_1, y_2, \ldots, y_{m_i}) = \sum_{j \in C_i} \frac{\partial f_i}{\partial y_j} \nabla f_j, \tag{2.17}$$

where C_i is the set of children of function i. The final gradient is $\nabla f(f_0(\mathbf{x}), \mathbf{x}) = \nabla f_n$, where f_n is the modulation function at the root of the hierarchy. The surface normal vector is then obtained by normalising the gradient, according to (2.3). One can avoid computing the recurrence (2.17) for some point $\mathbf{x}$ by estimating the normal vector with finite differences. This requires that $f(f_0(\mathbf{x}), \mathbf{x})$ be computed for three other points on the axes of a local coordinate frame centred on $\mathbf{x}$. The computation of a normal vector is the last step in a ray-surface intersection procedure – it is evaluated once an intersection point has been found. The cost of computing the normal with (2.17) is negligible compared with the cost of finding the intersection point along the ray. For this reason, it is better to compute the normal exactly, with (2.17), rather than estimate it with finite differences, especially when the surface is very irregular. Procedures need to be implemented not only to evaluate the functions f_i but also to evaluate their derivatives.

In Chapter 8, the Hessian matrix of $f(f_0(\mathbf{x}), \mathbf{x})$ is required. The Hessian matrix $\mathcal{H}\{f\} = [\partial^2 f / \partial x_i \partial x_j]$, with $i, j = 1, 2, 3$, gathers all the second partial derivatives of the implicit function. The Hessian matrix of any function f_i in the hypertexture hierarchy is obtained by applying the chain rule of differentiation a second time on (2.17):

$$\mathcal{H}\{f_i\}(y_1, y_2, \ldots, y_{m_i}) = \sum_{j \in C_i} \frac{\partial^2 f_i}{\partial y_j^2} \left(\nabla f_j \cdot \nabla f_j^T \right) + \sum_{j \in C_i} \frac{\partial f_i}{\partial y_j} \mathcal{H}\{f_j\}. \tag{2.18}$$

As Perlin and Hoffert [1989] already suggested, the hypertexture model can be enhanced by considering for each modulation function not only a dependence on the outputs y of other functions but also a dependence on the gradient ∇y of those outputs. This will allow the modelling of additional hypertexture features that are based on the local slope introduced by

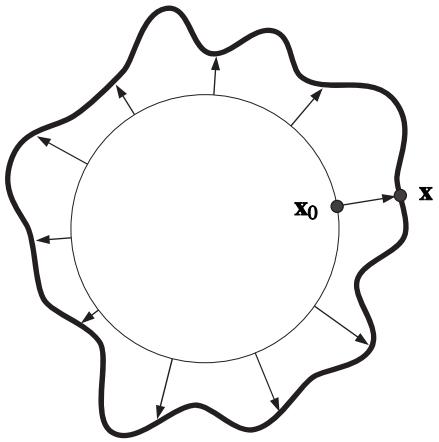


Figure 2.16: A surface defined with displacement mapping over a sphere.

previous features. Higher derivatives of y, such as the Hessian matrix $\mathcal{H}\{y\}$, could also be included as arguments to modulation functions to introduce curvature-dependent hypertexture features. The recursive analytical expressions (2.17) and (2.18) for the surface gradient vector and Hessian matrix, unfortunately, become increasingly more complex as dependencies of progressively higher derivatives of the y outputs are introduced. The hypertexture model that was implemented for this thesis considers only modulation functions of the type $f_i(y_1, y_2, \ldots, y_{m_i})$. There is no obstacle in the current implementation, however, to the introduction of differential hypertexturing effects. The study of modulation functions with differential dependencies will be left for a future research opportunity.

2.4.2 Basic Terrain Shape

The seed function f_0 defines the terrain seed shape out of which the hypertextured surface is made. For modelling terrain, one is interested in simple seed shapes that can be described as algebraic surfaces (Section 2.2.2). A terrain defined over a horizontal plane is a particular case of (2.4). Considering again that the x_3 coordinate is taken along the vertical direction, the implicit function and the gradient vector are given by:

$$f_0(\mathbf{x}) = f_0(x_1, x_2, x_3) = x_3, \tag{2.19a}$$

$$\nabla f_0(\mathbf{x}) = (0, 0, 1). \tag{2.19b}$$

Traditionally, terrains were modelled with polygon meshes and their definition as heightfields over a limited area of a horizontal plane was natural. If the modulation functions for the hypertexture can be evaluated in a procedural way, there is no space limitation to observe and a terrain that is modelled as a hypertexture can easily be defined over an infinite plane. The quality of this terrain will depend on the ability of the modulation functions to generate detail over an infinite area without creating objectionable repetitions.

A more interesting shape out of which one can generate a terrain through hypertexturing is that of a sphere. A sphere represents a significant shift in the way terrains can be modelled.

Combined with the power of procedural modulation functions, a sphere can be used to model an entire planet in a more realistic way than a horizontal plane could do. This new approach to terrain modelling was pioneered by Musgrave [2003c] who built the first truly fractal planets that could be visualised in a realistic way from any altitude above the ground. The terrains created by Musgrave were specified as procedural displacement maps over the sphere. Figure 2.16 exemplifies how such displacement maps work. The altitude used to perform displacements is defined as the length along the normal vector to any point on the sphere. The realistic visualisation of a planetary landscape results from the combination of convincing procedural models of terrain with a realistic rendering of the atmosphere above the planet [Nishita et al., 1993].

An algebraic surface that represents a sphere can be generated from the implicit function given in (2.5). A sphere of unit radius is considered without loss of generality since a sphere of any size can be reduced to this case following a uniform scaling operation. To model surface features the size of Mount Everest on Earth, for example, one needs to create features on the unit sphere that have a size of $8,848m/6,371 \times 10^3 m = 1.38 \times 10^{-3}$ in dimensionless units. The implicit function and the gradient for the unit sphere can take an additional parameter n and are given by:

$$f_0(\mathbf{x}) = (\mathbf{x} \cdot \mathbf{x})^n - 1, \tag{2.20a}$$

$$\nabla f_0(\mathbf{x}) = 2n(\mathbf{x} \cdot \mathbf{x})^{n-1}\mathbf{x}, \tag{2.20b}$$

Equation (2.20a) is a generalisation of (2.5) for the unit sphere since (2.5) results from (2.20a) when $n = 1$. The parameter $n > 0$ is related to the rate of change of f_0 away from the surface of the sphere. Evaluating the modulus of the gradient over the surface of the unit sphere (where $\mathbf{x} \cdot \mathbf{x} = 1$) gives $\|\nabla f_0(\mathbf{x})\| = 2n$. The parameter n has an influence on the blending between the initial spherical surface and the hypertextured detail. If n is large, the function f_0 will be dominant in the hypertexturing hierarchy and the surface detail will be significantly compressed towards the sphere. If n is small, the other functions will dominate in the hypertexturing hierarchy and the surface detail will extend much beyond the sphere. The optimal case occurs when a unit rate of change of f_0 is verified across the spherical surface. In this case, the interaction between the hypertexture detail and the sphere has a linear behaviour. The optimal case is given by $\|\nabla f_0(\mathbf{x})\| = 1$ along the original surface, which means that $n = 1/2$ so that the functions (2.20) for the sphere now become:

$$f_0(\mathbf{x}) = (\mathbf{x} \cdot \mathbf{x})^{1/2} - 1 = \|\mathbf{x}\| - 1, \tag{2.21a}$$

$$\nabla f_0(\mathbf{x}) = (\mathbf{x} \cdot \mathbf{x})^{-1/2}\mathbf{x} = \mathbf{x}/\|\mathbf{x}\|. \tag{2.21b}$$

The functions (2.21) are used throughout this thesis to generate images of procedural terrain defined over entire planets. This occurs even for viewpoints that are at a sufficiently low altitude for the curvature of the planet not to be discernible. For these viewpoints the planar approximation (2.19) could also have been used. The spherical functions (2.21) are clearly more expensive than the functions (2.19) for a planar terrain but their computational cost remains insignificant when compared with the cost of evaluating the procedural noise functions of Chapter 4, which are required to model the terrain shape. For that reason, the modelling of terrain over a sphere has little overhead over the modelling of terrain over a plane. By modelling spherical terrains, it is possible to have camera zooming sequences away from the surface of the planet so that the planet's curvature gradually becomes visible.

2.4.3 Incorporating Displacement Maps

The conventional method of modelling terrain as a displacement map (which includes height-fields above a plane) can be incorporated into a hypertextured implicit surface representation. Consider the case previously shown in Figure 2.16. Given a point $\mathbf{x}_0$ on the original surface, displacement mapping generates a displaced point $\mathbf{x}$ according to:

$$\mathbf{x} = \mathbf{x}_0 + h(\mathbf{x}_0)\mathbf{n}(\mathbf{x}_0), \tag{2.22}$$

where $\mathbf{n}(\mathbf{x}_0)$ is the surface normal at point $\mathbf{x}_0$ and $h(\mathbf{x}_0)$ is a three-dimensional map evaluated at $\mathbf{x}_0$. The surface generated by displacement mapping is equivalent to an implicit surface generated through domain deformation (Section 2.3.3) whose implicit function is:

$$f(f_0(\mathbf{x}), \mathbf{x}) = f_0(\mathbf{x}) - h(\mathbf{d}(\mathbf{x})). \tag{2.23}$$

A domain mapping function $\mathbf{d}$ is sought such that the deformed surface $f(f_0(\mathbf{x}), \mathbf{x})$ is the same as the surface generated from (2.22). This will hold provided that $\mathbf{d}(\mathbf{x}) = \mathbf{x}_0$, i.e. provided the domain mapping function maps any point $\mathbf{x}$ to its nearest point $\mathbf{x}_0$ on the original surface. As long as the original surface is convex (which is true for both planes and spheres), the nearest surface point for any point $\mathbf{x}$ away from the surface is always well defined. For a general convex surface, the distance function $\mathbf{d}(\mathbf{x})$ may be difficult to specify but for planes and spheres it has very simple expressions. In the case of a horizontal plane, the domain mapping function is:

$$\mathbf{d}_{PLANE}(\mathbf{x}) = \mathbf{d}_{PLANE}(x_1, x_2, x_3) = (x_1, x_2, 0). \tag{2.24}$$

In the case of a sphere with unit radius, the domain mapping function is:

$$\mathbf{d}_{SPHERE}(\mathbf{x}) = \frac{\mathbf{x}}{\|\mathbf{x}\|}. \tag{2.25}$$

One can check that the implicit function f is able generate a displacement mapped surface over the sphere by plugging (2.22) in (2.23) and using the definition (2.21) for f_0. One also uses the fact that, for a unit radius sphere, $\mathbf{x}_0 = \mathbf{n}(\mathbf{x}_0)$ and that $\|\mathbf{x}_0\| = \|\mathbf{n}(\mathbf{x}_0)\| = 1$:

$$\begin{aligned} f(f_0(\mathbf{x}), \mathbf{x}) &= f_0(\mathbf{x}) - h(\mathbf{d}(\mathbf{x})) \\ &= f_0\big(\mathbf{x}_0 + h(\mathbf{x}_0)\mathbf{n}(\mathbf{x}_0)\big) - h(\mathbf{x}_0) \\ &= \|\mathbf{x}_0 + h(\mathbf{x}_0)\mathbf{n}(\mathbf{x}_0)\| - 1 - h(\mathbf{x}_0) \\ &= 1 + h(\mathbf{x}_0) - 1 - h(\mathbf{x}_0) \\ &= 0. \end{aligned} \tag{2.26}$$

This result shows that the implicit surface passes through the point $\mathbf{x}$ of the displacement mapped surface. Because $\mathbf{x}$ is chosen arbitrarily, the two surfaces are equal. A similar result can be obtained for a displacement mapped surface over a plane.

Displacement mapping can be inserted into a hypertexture hierarchy by plugging the domain mapping function into the desired source functions. These are the only functions in the hierarchy that have an explicit dependence on the spatial variable $\mathbf{x}$ and for which the introduction of a domain deformation makes sense. The use of displacement mapping within a hypertexture

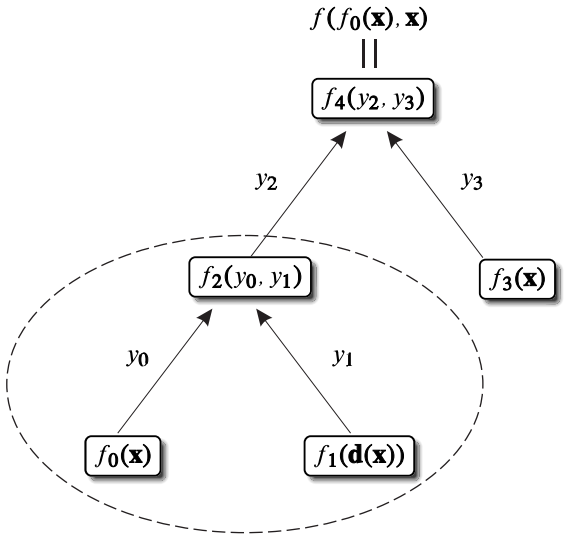
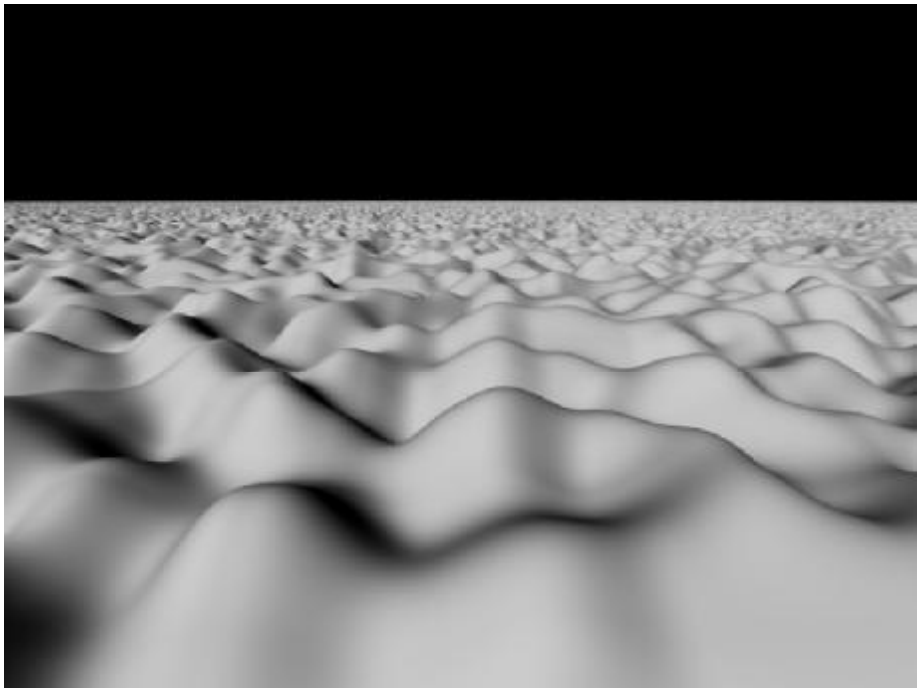


Figure 2.17: A hypertexture hierarchy with displacement mapping effects. The sub-hierarchy inside the oval contains only displacement mapped surfaces.

hierarchy is illustrated in Figure 2.17, which shows the same hierarchy of Figure 2.15 with displacement mapping effects added. The source function f_1 is displacement mapped but the source function f_3 remains unchanged. Any hierarchy that contains only displacement mapped source functions (not counting the original seed function f_0) is guaranteed to generate only displacement mapped surfaces. If the implicit surface of Figure 2.17 was taken directly from the output of function f_2, the result would be a displacement mapped surface. The insertion of at least one source function into the hierarchy without displacement mapping creates the possibility that the final implicit surface may have overhangs. The source function f_3 of Figure 2.17 has the role of introducing overhanging features into the displacement mapped surface that is output by modulation function f_2. Figure 2.18 shows an example of a hypertexture that adds overhangs to a terrain generated with displacement mapping. The displacement mapped terrain is shown in isolation in Figure 2.18(a) and with the additional hypertextured features in Figure 2.18(b).



(a) Displacement map

(b) Displacement map plus hypertexture

Figure 2.18: A hypertexture applied to a displacement mapped surface.

CHAPTER 3

Stochastic Surface Models

STOCHASTIC surface models introduce randomness into the description of an object and, in doing so, allow complex and natural looking surfaces to be generated. All stochastic surface models resort to a random number generator that introduces the required element of irregularity. Stochastic surfaces were first proposed by Fournier and Fussell [1980] as a technique to represent natural objects. Fournier and Fussell demonstrated the usefulness of stochastic surfaces by using them to model fractal terrains. A stochastic surface is generated as the outcome of a *stochastic process* – a mathematical model responsible for describing some quantity that varies randomly in space or time, based on a predefined set of statistical parameters. Given a stochastic surface model, associated to some specific stochastic process, an infinite number of surfaces can be generated that correspond to different realisations of the same process. One can then apply statistical concepts to surfaces and consider, for example, the average surface shape or the variance of the generated surfaces relative to this average shape.

Stochastic surface models must obey the principles of *internal consistency* and *external consistency* [Fournier et al., 1982]. Internal consistency means that independent generations of the same surface should lead to the same geometry. Imaging a stochastic surface from different viewpoints, for example, should produce different renderings of the same surface. In practice, one achieves internal consistency by seeding the random number generator with keys that are univocally assigned during the construction of the surface. Each key or set of keys, depending on the model that is used, identify unambiguously a surface among the infinite number of surfaces that a stochastic process can generate. A stochastic surface model has external consistency if it is possible to control the surface boundaries so that continuity is achieved when placing several surfaces together in a piecewise fashion. External consistency is generally difficult to achieve but the issue is not relevant for this thesis since each of the surfaces considered here is generated by a single stochastic model.

Fournier [1989] created a taxonomy of algorithms for the modelling of natural phenomena. According to this taxonomy, the representation of natural objects with stochastic surfaces can be considered as an example of a *morphological* approach. In a morphological approach, one does not care about the exact physical mechanisms that may have given rise to some natural shape but, instead, tries to develop a model that will approximate that shape through a small set of parameters. This is an important point as all the surfaces that are treated in this thesis can be seen as examples of morphological models. Often, natural shapes are governed by physical processes that are significantly complex and costly to simulate. A morphological approach attempts to replace accurate physical models with convincing stochastic model approximations

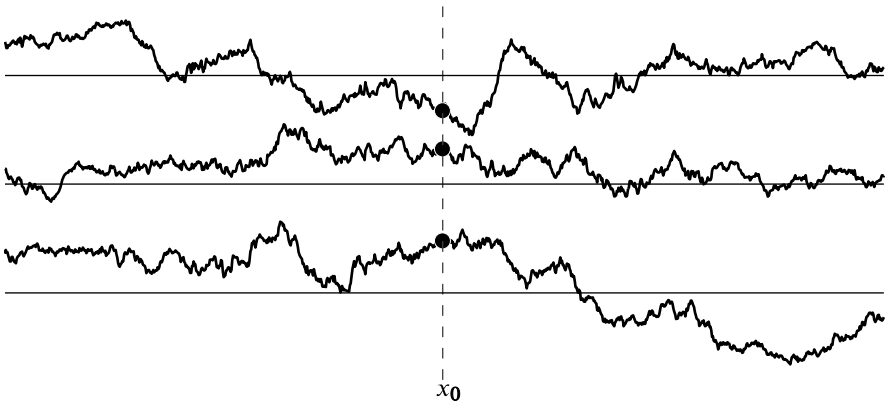


Figure 3.1: Three outcomes of a one-dimensional stochastic process $f(x)$. The point $f(x_0)$ is a random variable whose outcomes are also shown.

that, by comparison, are much easier to compute. For many computer graphics applications, this resemblance between an appropriate morphological model and a physically accurate model is often all that is required. The same morphological principle was also identified by Musgrave [2003a], who called it *ontogenetic modelling*.

This chapter presents an overview of the established theory of stochastic processes, fractal stochastic models for Computer Graphics and terrain synthesis models. These topics have been extensively described in other works but, in this chapter, an attempt is made to explain them in a logical sequence that leads to the research results presented in subsequent chapters. Section 3.1 gives a general introduction to the theory of stochastic processes, which is behind all stochastic surface models. Section 3.2 then considers the more specific case of fractal surface models. These are stochastic surface models with well defined statistical scaling properties. Section 3.3 applies the concepts from the previous sections to the generation of synthetic terrain models. Finally, Section 3.4 considers synthetic terrains that are modelled as hypertextured implicit surfaces. In Sections 3.2 to 3.4, special attention is given to procedurally defined stochastic surfaces. Procedural surfaces have many advantages over polygonal mesh surfaces such as a potentially infinite domain, compact memory storage and built-in level of detail.

3.1 Stochastic Processes

An n-dimensional stochastic process is designated by $f(\mathbf{x}, s)$, where $\mathbf{x} \in \mathbb{R}^n$ is a vector indicating the spatial extent of the process and s is a sample from the space of possible outcomes [Papoulis and Pillai, 2001]. For the purposes of surface modelling, the process is assumed to have only a spatial dependency. Under this assumption, a stochastic process can also be called a *stochastic field*. If one sets $s = s_0$ then $f(\mathbf{x}, s_0)$ is a field representing the outcome of the process for the particular case s_0. If, on the other hand, one sets $\mathbf{x} = \mathbf{x}_0$ then $f(\mathbf{x}_0, s)$ is a random variable, giving the value of the process at the point $\mathbf{x}_0$. Finally, if both $\mathbf{x}$ and s are set then all randomness is removed from the process and $f(\mathbf{x}_0, s_0)$ is the deterministic value of point $\mathbf{x}_0$ for outcome s_0. For simplicity, the s parameter is dropped and the stochastic field

is written as $f(\mathbf{x})$. It should be clear from the context whether a deterministic or a stochastic field is being mentioned. Figure 3.1 shows three independent outcomes of a one-dimensional stochastic field $f(x)$. The point x_0 in Figure 3.1 is a random variable, like every other point, and the three outcomes of this variable are part of the graphs of the stochastic field.

3.1.1 Preliminaries

Some concepts and mathematical notations should be clarified before proceeding with the discussion. Stochastic fields will be defined on the space of square integrable functions. A given field $f : \mathbb{R}^n \to \mathbb{R}$ is said to be square integrable if it has a finite norm $\|f\|$:

$$\|f\| = \int_{\mathbb{R}^n} |f(\mathbf{x})|^2 \, \mathbf{dx} < \infty. \tag{3.1}$$

The norm above corresponds to the energy of the stochastic field so that we are only interested in studying fields with finite energy. The Fourier transform $\mathcal{F}\{f\}$ of a square integrable field f is defined as:

$$\hat{f}(\mathbf{k}) = \mathcal{F}\{f(\mathbf{x})\} = \left(\frac{1}{\sqrt{2\pi}}\right)^n \int_{\mathbb{R}^n} f(\mathbf{x}) e^{-2\pi i \, \mathbf{x}\cdot\mathbf{k}} \, \mathbf{dx}. \tag{3.2}$$

The variable $\mathbf{k}$ is called the *wave vector* and is used to describe the spatial frequency content of the field. The modulus $k = |\mathbf{k}|$ of the wave vector is the spatial frequency or *wavenumber*, which has SI units of reciprocal metres (m^{-1}). The inverse Fourier transform can similarly be defined as an operator $\mathcal{F}^{-1}$ such that:

$$f(\mathbf{x}) = \mathcal{F}^{-1}\{\hat{f}(\mathbf{k})\} = \left(\frac{1}{\sqrt{2\pi}}\right)^n \int_{\mathbb{R}^n} \hat{f}(\mathbf{k}) e^{2\pi i \, \mathbf{x}\cdot\mathbf{k}} \, \mathbf{dk}. \tag{3.3}$$

One naturally has that $\mathcal{F}^{-1}\{\mathcal{F}\{f(\mathbf{x})\}\} = f(\mathbf{x})$. With this property, a field can be unambiguously recovered from its Fourier transform. The important Plancherel theorem states that the energy of a field can be measured equally in Euclidean space or in wavenumber space:

$$\|f\| = \|\hat{f}\|. \tag{3.4}$$

Equation (3.4) characterises the Fourier transform as an *isometry* relative to the norm $\|\cdot\|$ as a metric of both the Euclidean and wavenumber spaces.

3.1.2 Distributions and Statistical Measures

A stochastic field is completely determined by a hierarchy of joint probability distribution functions. At the bottom level there are the independent distributions of every point in the field:

$$F_{\mathbf{x}_0}(y) = P(f(\mathbf{x}_0) < y). \tag{3.5}$$

These distributions tell us how each particular point $f(\mathbf{x}_0)$ will behave in a statistical sense. At the next higher level, the joint distributions are defined between any pair of points:

$$F_{\mathbf{x}_1,\mathbf{x}_2}(y_1, y_2) = P(f(\mathbf{x}_1) < y_1, f(\mathbf{x}_2) < y_2). \tag{3.6}$$

One can continue in this fashion up to any number m of points and specify all the joint distributions $F_{\mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_m}(y_1, y_2, \ldots, y_m)$. If two stochastic processes f and g can be shown to have the same set of joint distribution functions then they are *statistically identical*. Two processes can be identical even if the underlying mechanisms that generate them are different. One can be the evolution of a market share, to give an example of a one-dimensional time-varying process, and the other can be the voltage noise on a transmission line. If the two are plotted side by side they will look identical. It is not meant by "identical" that the two graphs will be exact copies of each other but rather that they will have the same statistical signature, as happens with the graphs of Figure 3.1.

Clearly, the distributions $F_{\mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_m}(y_1, y_2, \ldots, y_m)$ for every possible combination of points represent an infinite amount of information. Some simplifying assumptions must be considered in order to reduce the complexity of representing a stochastic field. The most widely used simplification characterises f as a *Gaussian field*. In a Gaussian field all the previous distributions are jointly Gaussian and the field is completely determined from (3.5) and (3.6) alone [Doob, 1953]. The information contained in these two distributions is normally represented in a different format, based on the statistical measures of *mean*, *variance* and *autocorrelation*. The mean is given by:

$$\mu_f(\mathbf{x}) = E\{f(\mathbf{x})\} = \int_{\mathbb{R}} y f_{\mathbf{x}}(y)\, dy, \tag{3.7}$$

where $f_{\mathbf{x}}(y)$ is the Gaussian probability density at point $\mathbf{x}$ and $E\{\cdot\}$ is the expected value operator. The variance is given by:

$$\sigma_f^2(\mathbf{x}) = E\{(f(\mathbf{x}) - \mu_f(\mathbf{x}))^2\} = \int_{\mathbb{R}} y^2 f_{\mathbf{x}}(y)\, dy - \mu_f^2(\mathbf{x}). \tag{3.8}$$

Only fields of zero mean will be of concern[1], in which case the previous equation becomes:

$$\sigma_f^2(\mathbf{x}) = E\{f^2(\mathbf{x})\} = \int_{\mathbb{R}} y^2 f_{\mathbf{x}}(y)\, dy. \tag{3.9}$$

Finally, the autocorrelation is given by:

$$R_f(\mathbf{x}_1, \mathbf{x}_2) = E\{f(\mathbf{x}_1) f(\mathbf{x}_2)\} = \int_{\mathbb{R}^2} y_1 y_2 f_{\mathbf{x}_1\mathbf{x}_2}(y_1, y_2)\, dy_1 dy_2. \tag{3.10}$$

Note that $R_f(\mathbf{x}_1, \mathbf{x}_1) = E\{f(\mathbf{x}_1) f(\mathbf{x}_1)\} = \sigma_f^2(\mathbf{x}_1)$, under the assumption of zero mean fields. The autocorrelation expresses the amount of coherence between different points of the field. It is intuitive that such coherence must get arbitrarily small as the distance $\|\mathbf{x}_1 - \mathbf{x}_2\|$ between points increases. An autocorrelation must not be a completely arbitrary function. In fact, it must obey the relations:

$$R_f(\mathbf{x}_1, \mathbf{x}_2) = R_f(\mathbf{x}_2, \mathbf{x}_1), \tag{3.11a}$$

$$R_f(\mathbf{x}_1, \mathbf{x}_1) \geqslant |R_f(\mathbf{x}_1, \mathbf{x}_2)|. \tag{3.11b}$$

The first relation (3.11a) means that the correlation depends only on the relative position between two points while (3.11b) means that each point in the field has always a maximal

[1] A general stochastic field can always be obtained through the addition of a deterministic field with a zero mean stochastic field.

correlation with itself. A corollary of (3.11b) is that $R_f(\mathbf{x}_1, \mathbf{x}_1) \geqslant 0$, which was to be expected since $R_f(\mathbf{x}_1, \mathbf{x}_1) = \sigma_f^2(\mathbf{x}_1)$.

Gaussian processes are extremely useful to characterise random natural phenomena. Some processes, like the thermal noise between the ends of a resistor, are intrinsically Gaussian. Others, like most noises studied in Communications Theory, can be considered Gaussian to a good level of accuracy. This happens when a given noise comes from different and independent sources. The addition of a large number of independent and identically distributed noises converges to a Gaussian noise, according to the Central Limit Theorem, whatever the nature of the initial noises. The mathematical treatment of Gaussian fields is substantially simplified by another important property: a Gaussian field injected through a linear, space-invariant, filter will always remain Gaussian. Gaussian fields thus form a closed subset of the broader class of stochastic fields under filtering operations.

3.1.3 Stationarity and Isotropy

A stochastic field is said to be strictly *stationary* (also called *homogeneous*) if it is statistically invariant relative to displacements of the coordinate system. This means that all the joint distribution functions do not change under translation, i.e. $F_{\mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_m}(y_1, y_2, \ldots, y_m) = F_{\mathbf{x}_1+\mathbf{c},\mathbf{x}_2+\mathbf{c},\ldots,\mathbf{x}_m+\mathbf{c}}(y_1, y_2, \ldots, y_m)$ for any vector $\mathbf{c}$. A stochastic field is said to be stationary in the wide sense if the mean is constant, i.e. $\mu_f(\mathbf{x}) \equiv \mu_f$, and, furthermore, the autocorrelation depends only on the displacement vector $\tau = \mathbf{x}_1 - \mathbf{x}_2$ between points:

$$R_f(\mathbf{x}_1, \mathbf{x}_2) = R_f(\mathbf{x}_1 - \mathbf{x}_2) = R_f(\tau). \tag{3.12}$$

Note that (3.12) also implies that the variance is constant: $\sigma_f^2(\mathbf{x}) = R_f(\mathbf{0}) = \sigma_f^2$. Strict stationarity is a stronger statement about the statistics of a field than wide sense stationarity. Strict stationarity implies wide sense stationarity but the opposite is not necessarily true. It is true, however, for Gaussian stochastic fields so that one does not need to worry about these distinctions and consider Gaussian fields that are simply stationary. The admissibility conditions (3.11) for the case of stationary autocorrelations become:

$$R_f(\tau) = R_f(-\tau), \tag{3.13a}$$

$$R_f(\mathbf{0}) \geqslant |R_f(\tau)|. \tag{3.13b}$$

The autocorrelation of stationary signals is related to the important concept of *power spectrum distribution function* (PSDF for short). The autocorrelation $R_f(\tau)$ and the PSDF $G_f(\mathbf{k})$ of a field f constitute a Fourier transform pair, as stated by the Wiener-Khintchine theorem [Carlson et al., 2001]:

$$G_f(\mathbf{k}) = \mathcal{F}\{R_f(\tau)\}. \tag{3.14}$$

The PSDF expresses the distribution of power among the several frequencies of the stochastic field. For instance, in a one-dimensional process, the power contained in the frequencies between k_0 and $k_0 + dk$ is given by $dG_f(k_0) = G_f(k_0)dk$. Since $R_f(\tau)$ and $G_f(\mathbf{k})$ are a Fourier pair, it follows that both these functions must be in $L^2(\mathbb{R}^n)$. In fact, the space of admissible PSDFs is tighter than that. A PSDF must both belong to $L^2(\mathbb{R}^n)$ and have an associated autocorrelation obeying (3.13). Other properties of the power spectrum distribution include:

$$G_f(\mathbf{k}) \geqslant 0, \tag{3.15a}$$
$$G_f(-\mathbf{k}) = G_f(\mathbf{k}), \tag{3.15b}$$

which can be deduced from the properties of the autocorrelation function. The variance of a stationary field can also be obtained from its PSDF according to:

$$\sigma_f^2 = R_f(\mathbf{0}) = \int_{\mathbb{R}^n} G_f(\mathbf{k})\mathbf{dk}. \tag{3.16}$$

The discussion so far has been centred on 'power spectrum distribution functions'. It is also possible to work with the concept of 'energy spectrum distribution functions' except that it does not apply well to the case of stationary fields. Recall that a field in $L^2(\mathbb{R}^n)$ has an energy given by:

$$E_f = \|f\|_2^2 = \int_{\mathbb{R}^n} |f(\mathbf{x})|^2\mathbf{dx}. \tag{3.17}$$

If E_f is finite then $\lim_{\|\mathbf{x}\|\to\infty} f(\mathbf{x}) = 0$ must hold. This means the statistical measures taken from the field will be different depending on whether they are taken near the origin or far away from it. Displacement invariance becomes impossible to hold, with the conclusion that energy is not the appropriate unit of measure for stationary stochastic fields. A better measuring unit is the *expected power* of the field, given by:

$$P_f = \lim_{r\to\infty} \frac{1}{V(r)} \int_{\|\mathbf{x}\|\leqslant r} E\{f^2(\mathbf{x})\}\mathbf{dx}, \tag{3.18}$$

where $V(r)$ is the volume of a n-dimensional hypersphere centred at the origin and with radius r. For a stationary field, the expected power is also the variance of the field since (3.18) becomes $P_f = E\{f^2\} = \sigma_f^2$. The PSDF of a field quantifies the distribution in wave space of this power σ_f^2, according to (3.16).

A second invariance property of stochastic fields is called *isotropy*. While stationarity deals with displacement invariance, isotropy deals with rotational invariance. The statistical properties of an isotropic field must remain the same under a rotation of the coordinate axes around the origin. The autocorrelation function is now further simplified:

$$R_f(\boldsymbol{\tau}) = R_f(\|\boldsymbol{\tau}\|) = R_f(\tau), \tag{3.19}$$

with τ being the norm of $\boldsymbol{\tau}$. The correlation between a pair of points depends exclusively on their distance and is free from directional information. If $R_f(\tau)$ is a radially symmetric function then the same must happen to $G_f(\mathbf{k})$ since these are a Fourier pair. The relation between $R_f(\tau)$ and $G_f(k) = G_f(\|\mathbf{k}\|)$ is a particular case of the Fourier transform in several dimensions and is called the *Hankel transform*. For a n-dimensional field, this takes the form:

$$G_f(k) = \frac{2\pi}{k^{n/2-1}} \int_0^\infty R_f(\tau) J_{n/2-1}(2\pi k\tau)\tau^{n/2} d\tau, \tag{3.20}$$

where $J_{n/2-1}$ is the Bessel function of the first kind of order $n/2-1$ [Bracewell, 1999].

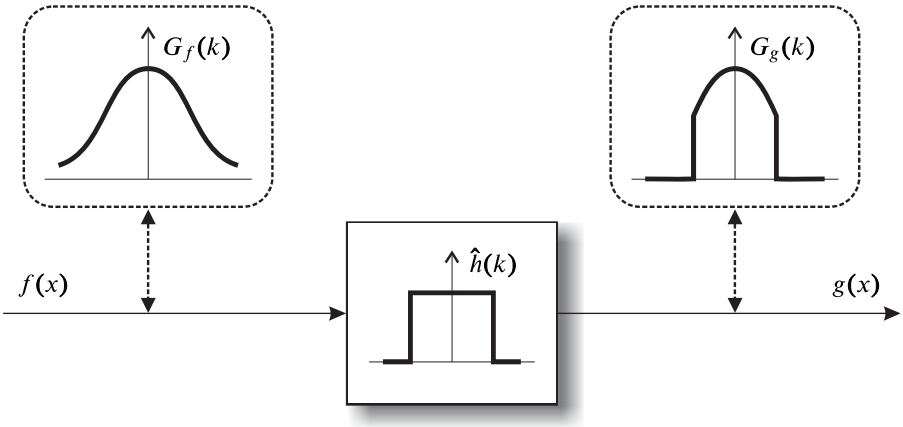


Figure 3.2: A one-dimensional field f passes through an idealised low-pass filter with impulse response h and becomes field g.

It has already been mentioned in Section 3.1.2 that Gaussian fields remain Gaussian after being filtered through a linear space-invariant filter. If the frequency response of the filter is known, it is possible to relate exactly the statistics of the input field with those of the output field. More specifically, if g is the filtered version of f then their respective PSDFs will be given by:

$$G_g(\mathbf{k}) = G_f(\mathbf{k})|\hat{h}(\mathbf{k})|^2, \tag{3.21}$$

where $\hat{h}(\mathbf{k})$ is the frequency response of the filter and $h(\mathbf{x}) = \mathcal{F}^{-1}\{\hat{h}(\mathbf{k})\}$ is the corresponding impulse response. The PSDF of the original field comes out modulated by the squared frequency response of the filter. Figure 3.2 illustrates this filtering operation with an idealised low-pass filter. The high frequency content of the original field is removed from the PSDF of the filtered field. As a consequence, the filtered field is going to be smoother and more regular than the original since it will not have any power in the high frequencies. The relation (3.21) becomes even simpler if f is a *Gaussian white noise*. Each point on this type of noise field is a random variable with a Gaussian distribution and is completely independent from every other point. The PSDF of a white noise is $G_f(\mathbf{k}) \equiv \eta/2$, where η is the power density (power per wavenumber) of the noise. The filtered PSDF becomes:

$$G_g(\mathbf{k}) = \frac{\eta}{2}|\hat{h}(\mathbf{k})|^2. \tag{3.22}$$

Equation (3.22) shows how a stationary stochastic field with any desired PSDF can be obtained by injecting white noise through a filter with the appropriate frequency response. Because of this, the output noise g is sometimes called *coloured noise*. Equation (3.22) forms the basis of the spectral synthesis method that is explained in Section 3.2.2.

3.2 Fractal Surface Models

Stochastic models for computer graphics rely, to a large extent, on an intriguing branch of mathematics called *fractal geometry*. The concept of a fractal process was initially proposed

by Benoit Mandelbrot to model the irregular bursts of noise that occur along an electrical transmission line but he soon discovered that it could also describe a very large range of natural phenomena including terrains, coastlines, clouds and fluid turbulence [Mandelbrot, 1983]. Mandelbrot showed that all these seemingly disparate phenomena occur over a very wide range of scales and have a property of *self-similarity* in common.

If a self-similar function is plotted on a graph then one can zoom in or zoom out on the function as much as desired and never run out of details to look at. One can also pan left or right and keep seeing ridges, valleys and plateaus in endless succession. According to the data amplification principle of Smith [1984], this infinite amount of information does not require an infinite amount of space to be stored. Rather, it only takes a few parameters that one might interpret visually as "raggedness" or "wiggliness" in order to completely specify a fractal model.

Fractal geometry can be broadly divided between deterministic fractals and random fractals. Examples of deterministic fractals are the famous Mandelbrot and Julia sets. These sets are obtained through successive iteration of an algebraic function that is evaluated on the plane of the complex numbers. Deterministic fractal sets have self-similarity. By zooming in on the Mandelbrot set, for example, one encounters an endless number of smaller copies embedded within the original set. Stunning images of deterministic fractals have been produced but the deterministic nature of these fractals imposes a rigid geometric structure that cannot easily mimic the randomness found in natural phenomena [Peitgen et al., 1992].

Random fractals also have a property of self-similarity but, unlike determinist fractals, they are realisations of a stochastic field. A stochastic field is fractal if it is statistically identical to scaled copies of itself. Methods for computing random fractal surfaces can be classified into three categories, depending on the technique that is used to enforce the fractal scaling law for the surface:

- The spectral synthesis method.
- Polygonal subdivision methods.
- Procedural methods.

These are presented in sequence. Of more interest to this thesis are the procedural methods since they lead naturally to procedural fractal models for surfaces. Before this is done, however, *fractional Brownian motion* is presented as the underlying stochastic fractal process that all methods attempt to generate.

3.2.1 Fractional Brownian Motion

Fractional Brownian motion (fBm), as discovered by Mandelbrot and van Ness [1968], is a stochastic field f that has a statistical characteristic of similarity across all scales. For fBm, there is the following statistical identity:

$$f(\mathbf{x}_0 + \mathbf{x}) - f(\mathbf{x}_0) \equiv \frac{1}{r^H}\left(f(\mathbf{x}_0 + r\mathbf{x}) - f(\mathbf{x}_0)\right), \tag{3.23}$$

where $r > 0$ is an arbitrary scaling factor and $0 < H < 1$ is the *Hurst coefficient.* Fractional Brownian motion is a generalisation of Brownian motion, which corresponds to the case $H = 0.5$. The equivalence sign in (3.23) is used to signify that the probability distributions are the same for the fields on both sides of the equation. To be more precise, it is not the fBm field f that is fractal but rather its increments relative to some reference point $\mathbf{x}_0$ that are fractal, as the previous equation shows. One can, however, chose $\mathbf{x}_0 = \mathbf{0}$ and $f(\mathbf{x}_0) = 0$ so that (3.23) becomes:

$$f(\mathbf{x}) \equiv \frac{1}{r^H} f(r\mathbf{x}), \tag{3.24}$$

The similarity law (3.24) is normally used to study the statistical properties of fBm with the understanding that this equation applies to the increments of the fBm field relative to its value at the origin and not to the fBm field itself.

Any fractal, be it deterministic or random, has an associated *fractal dimension.* This dimension can take fractional values (as opposed to the integer valued euclidian dimension) and expresses the amount of irregularity contained in that fractal[2]. Fractal dimension is usually measured with a box-counting algorithm [Falconer, 1990]. The fractal object is covered with a grid of square boxes with sides of length r. The number N of boxes that intersect with the object is then counted. The fractal dimension is obtained in the limit of $\log N / \log r^{-1}$ as the size of the boxes goes to zero. For fBm, however, this process is not required because the fractal dimension D of a n-dimensional field is a straightforward function of the Hurst coefficient H:

$$D = n + 1 - H. \tag{3.25}$$

Given that $0 < H < 1$, the fractal dimension of a fBm field is always in-between the dimension n of its parameter space, also called the *topological dimension*, and the next higher dimension $n+1$. The proximity to each of these two dimensions will depend on the value of H and hence on the irregularity of the field. Consider, for simplicity, the case of a one-dimensional field $f(x)$. For values of H that are close to 1.0, D will also be close to 1.0 and the graph of f in these circumstances will show the most regularity. It is a common misconception to think that, as $H \to 1$ and consequently $D \to 1$, f will lose all detail and converge to a horizontal line. The fact that the fractal dimension in this case will approach the topological dimension of a line does not mean that the graph of f will become a line. What $D = 1$ does mean is that, if one zooms in close enough, the graph of f will *locally* flatten out and converge to a line of constant slope. For any other fractal dimension $D > 1$, the graph of f will never become flat at small scales, no matter how close one zooms in. In the opposite case, for values of H close to 0.0, D will be close to 2.0. This means that, as H is decreased, the graph of f becomes increasingly irregular. In the limit of $H = 0.0$ and $D = 2.0$, f becomes a *space filling curve*, i.e. a curve so irregular that it fills all of space.

Figure 3.3 shows the graphs of a one-dimensional fBm field with H equal to 0.8, 0.5 and 0.2. As H is decreased, the curve for f becomes increasingly erratic. It is easy to see that if $H = 0.0$ the curve becomes a continuous blotch of ink on the paper. The most interesting of the three curves for f is the one with $H = 0.8$ and $D = 1.2$. This curve could easily pass for the profile of a mountain range seen from afar. It is this single observation that has sparked the

[2]The name "fractal" was coined by Benoit Mandelbrot precisely because of the fact that dimensional measures of self-similar objects have a fractional component.

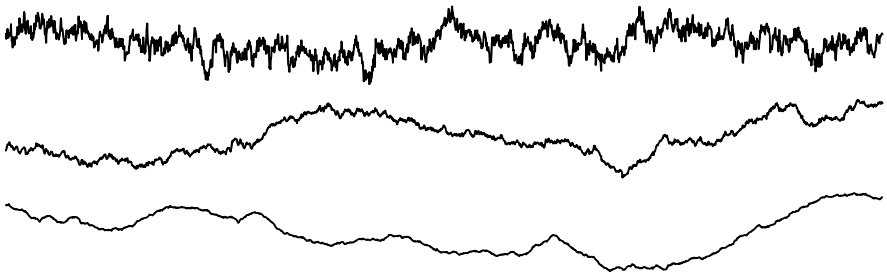


Figure 3.3: Three graphs of a fractional Brownian motion stochastic field. The parameters are $H = 0.2$ and $D = 1.8$ (top), $H = 0.5$ and $D = 1.5$ (middle), $H = 0.8$ and $D = 1.2$ (bottom).

research into all of the fractal terrain generation methods used for computer graphics. As Voss [1988] pointed out, the fractal objects whose dimension D has a fractional component in the vicinity of 0.2 seem to be the most successful in replicating many phenomena in nature, like terrain or clouds.

It can be shown that the power spectrum distribution function for a n-dimensional fBm field f must scale with wavenumber according to:

$$G_f(\mathbf{k}) \propto \|\mathbf{k}\|^{-\beta}, \tag{3.26}$$

where the spectral falloff exponent is given by $\beta = 2H + n$. Equation (3.26) can be obtained by deriving a scaling law similar to (3.24) for the autocorrelation R_f and then applying a Fourier transform. A different derivation based on the power of f is presented in Saupe [1988]. A stochastic field whose PSDF obeys (3.26) can be categorised as fractal. There are two problems with such a PSDF, however. The first problem is related to the fact that $G_f(\mathbf{0}) = +\infty$. Since $G_f(\mathbf{k})$ and $R_f(\tau)$ constitute a Fourier transform pair, one then has that $G_f(\mathbf{0}) = \int_{\mathbb{R}^n} R_f(\tau)\,\mathbf{d}\tau = +\infty$. This is not possible as it implies that either $\lim_{\|\tau\|\to\infty} R_f(\tau) > 0$ or the limit doesn't exist. Any meaningful autocorrelation function must decay to zero eventually, signifying that points sufficiently far apart in the field are totally uncorrelated. The second problem is the reverse of the first and is related to the fact that $\int_{\mathbb{R}^n} G_f(\mathbf{k})\,\mathbf{dk} = +\infty$. It was already seen that the integral of G_f is the power of the field and this will mean that a fBm field has infinite power. No stochastic processes in nature, however, can have such a characteristic[3].

The considerations in the previous paragraph lead to the conclusion that the fractal PSDF of (3.26) is only an approximation to the PSDF of actual stochastic fields. In particular, the PSDF of a stochastic field must deviate from a fractal scaling law for sufficiently small wavenumbers to avoid the singularity at the origin. Physically, this means that for sufficiently large scales (correspondingly, small wavenumbers) the field is no longer self-similar. Many fields also stop being self-similar at very small scales (correspondingly, high wavenumbers), which suggests that their PSDFs decay asymptotically faster than (3.26). Figure 3.4 shows the graph of a fractal PSDF with a falloff exponent $\beta = 2.0$, for which $H = 0.5$ and $D = 1.5$. The same

[3]The white noise process does have infinite power but it is a mathematical abstraction. Any real noise is band-limited by the channel through which it is transmitted and that limits its available power.

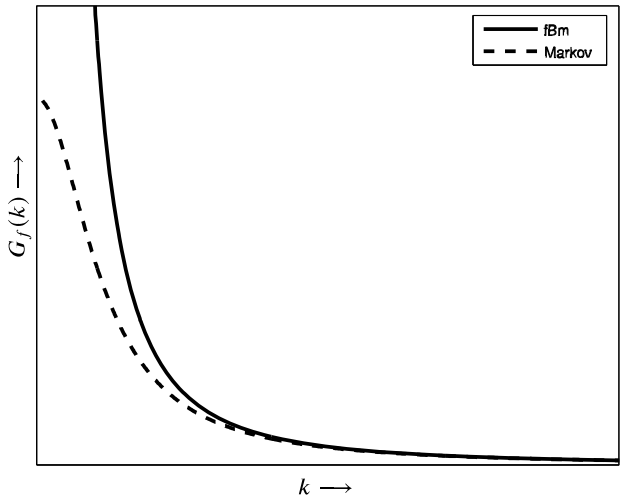


Figure 3.4: A fractal spectrum with $\beta = 2.0$ and a Markov spectrum with $\alpha = 0.1$.

figure shows, in a dotted line, an example of a more realistic PSDF. This is the PSDF for a Markov field, given by $G_f(\mathbf{k}) = (\|\mathbf{k}\|^2 + \alpha^2)^{-1}$ [Lewis, 1987]. The range of scales over which a real stochastic field can be approximated to a reasonable degree with a fractal scaling law is called the *fractal range*. The lowest wavenumber in this range must always be specified when providing a fractal approximation of a stochastic field. The highest wavenumber need not be specified, in which case the field assumes the asymptotic behaviour of the fractal PSDF of equation (3.26) for the high wavenumbers.

It it restated that the statistics derived for the fBm field are not valid for the actual field but instead for the increments of the field relative to some reference point, as (3.23) dictates. For any dimension n, the statistics for the increments of f remain invariant under translation and the increments are stationary. For $n > 1$, there is also a statistical invariance under a rotation of the coordinate system and the increments are isotropic, as can be seen from the fact that the PSDF in equation (3.26) only depends on the magnitude $\|\mathbf{k}\|$ of the wave vector. The actual fBm field is neither stationary nor isotropic and a PSDF cannot be derived for it under those circumstances. The only exception occurs for $H = 0.5$ when fBm becomes the simpler Brownian motion, which is indeed stationary and isotropic. Flandrin [1987] has shown that the stationary PSDF (3.26) can be recovered from a non-stationary fBm field through an averaging mechanism. The value of $G_f(k)$ at any wavenumber $k = \|\mathbf{k}\|$ can be obtained by averaging over distances $\|\mathbf{x}\| \gg 1/k$.

3.2.2 The Spectral Synthesis Method

With the spectral synthesis method, a filtering operation is performed over a white Gaussian noise stochastic field f to produce the desired surface g. The transfer function of the filter is chosen so that its output has the statistics of a fBm field, according to the filtering equation

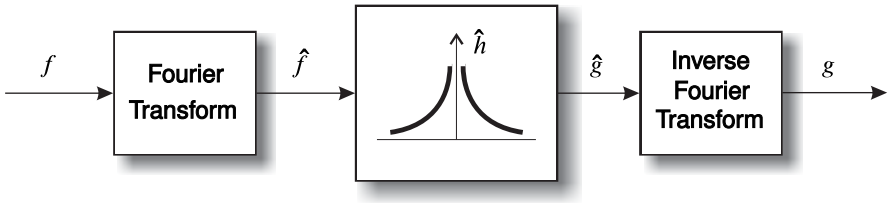


Figure 3.5: The sequence of operations for the spectral synthesis of fBm fields.

(3.22). The intention is to generate a field whose PSDF $G_g(\mathbf{k})$ is given by equation (3.26), and, therefore, the appropriate transfer function for the filter is $\hat{h}(\mathbf{k}) = \sqrt{G_g(\mathbf{k})} = \|\mathbf{k}\|^{-\beta/2}$. Figure 3.5 shows the conceptual operations performed by the synthesis method for the case of a two-dimensional surface. A white noise field $f(x_1, x_2)$ is passed through a Fourier transform to obtain the noise spectrum $\hat{f}(k_1, k_2)$. The spectrum is then multiplied by the filter's response to become the spectrum $\hat{g}(k_1, k_2)$ of the surface. A final inverse Fourier transform results in the desired surface $g(x_1, x_2)$. For any wavenumber pair, $\hat{f}(k_1, k_2)$ is a complex number whose real and imaginary components are independent Gaussian random variables with variance $\eta/2$. The output spectrum $\hat{g}(k_1, k_2)$, for the same set of wavenumbers, has similar characteristics but with variance $\eta|\hat{h}(k_1, k_2)|^2/2$. In practice, one does not need to perform the actual filtering operation. One can directly build $\hat{g}(k_1, k_2)$ with the desired statistics as if it had just passed through the filter. In this way, the initial Fourier transform and the multiplication in wave space that corresponds to the filtering operation can be skipped.

The use of a spectral synthesis method to generate fBm surfaces was first presented in Voss [1983]. The surface $g(x_1, x_2)$ is discretised on a regular grid with a size of $N \times N$ samples and represented as an inverse discrete Fourier transform given by:

$$g(x_1, x_2) = \sum_{k_1=0}^{N-1} \sum_{k_2=0}^{N-1} g_{k_1,k_2} \, e^{\frac{2\pi i}{N}(k_1 x_1 + k_2 x_2)}, \tag{3.27}$$

with integer coordinates $0 \leqslant x_1, x_2 < N$. The complex Fourier coefficients g_{k_1,k_2} have real and imaginary random components obeying a Gaussian distribution with zero mean and variance $\eta(k_1^2 + k_2^2)^{-\beta/2}/2$. Of the total N^2 coefficients, slightly less than half need not be generated. These latter coefficients are obtained through symmetry considerations so that $g(x_1, x_2)$ can be a real valued process. More specifically, the coefficients must obey $g_{k_1,k_2} = g^*_{N-k_1,N-k_2}$, where the asterisk signifies the complex conjugate, for the outcome of (3.27) to be strictly real (see Pratt [1978] for more details). Equation (3.27) is solved for $g(x_1, x_2)$ with the help of an inverse Fast Fourier Transform (iFFT). For maximum efficiency in computing the iFFT, N should be a power of two although iFFT algorithms for arbitrary N also exist [Brigham, 1988].

For a grid with a total of $N_T = N \times N$ samples, the complexity of the spectral synthesis method is $O(N_T \log N_T)$, which is the complexity for performing the iFFT. The polygon subdivision methods, to be presented in the next section, are faster, with a linear complexity $O(N_T)$. The spectral synthesis method, however, still retains some advantages. It is quite general and can be used to generate any stochastic field whose PSDF is known [Anjyo, 1988, 1991]. An example is shown in Figure 3.6 where a non-fractal ocean surface has been synthesised based on the Pierson-Moskowitz spectrum for fully developed wind-driven gravity waves [Mastin et al., 1987]. The spectral synthesis method also extends easily to three-dimensional fields. The

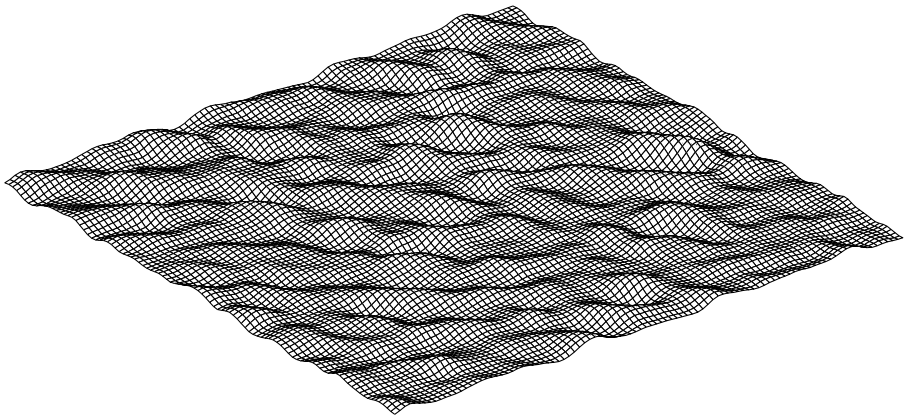


Figure 3.6: A water surface of fully developed wind-driven gravity waves generated through spectral synthesis.

surfaces generated with an iFFT are naturally periodic and obey $g(x_1+iN, x_2+jN) = g(x_1, x_2)$ for any integers i and j. An infinite surface can be obtained by tiling copies of $g(x_1, x_2)$ although the periodic artifacts resulting from the tiling are easily noticed.

3.2.3 Polygonal Subdivision Methods

The generation of one-dimensional fBm by *midpoint subdivision* uses a discrete representation of the process $f(x)$ and inserts new samples in-between already existing ones in an iterative fashion. At each iteration i, the process is approximated by the sequence of samples f_j^i, where $j = 0, 1, \ldots, 2^i$. The issue with midpoint subdivision methods is how to compute iteration f_j^i based on the results of the previous iteration f_j^{i-1} so that, in the limit, the statistics of fBm will be replicated.

Consider Figure 3.7. There are two consecutive samples f_j^{i-1} and f_{j+1}^{i-1} after iteration $i-1$. These samples become f_{2j}^i and f_{2j+2}^i in the next iteration. A new sample f_{2j+1}^i now has to be introduced, thereby doubling the resolution of the discrete approximation. Given the knowledge of the two original samples, it is expected that, on average, the new sample will be exactly in-between them. The new sample, however, will suffer a random displacement relative to this midpoint position, which will depend on the fractal dimension of the fBm process. The iteration rule for a fBm subdivision method can, therefore, be summarised as follows:

$$f_{2j}^i = f_j^{i-1}, \tag{3.28a}$$

$$f_{2j+1}^i = (f_j^{i-1} + f_{j+1}^{i-1})/2 + d^i, \tag{3.28b}$$

where d^i is a Gaussian random variable with zero mean and a variance $(\sigma^2)^i$ that is yet to be determined. The self-similarity law (3.23) for a fBm process can be rewritten, using the two endpoint samples:

$$f_{j+1}^{i-1} - f_j^{i-1} \equiv 2^H (f_{2j+1}^i - f_j^{i-1}). \tag{3.29}$$

The random increments on the left and right hand sides of (3.29) are statistically identical. By

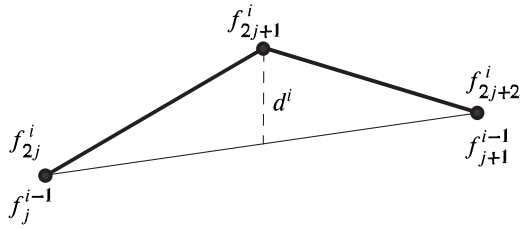


Figure 3.7: Midpoint subdivision of one segment of a fBm process.

implication, the variances of those increments must be equal. Plugging (3.28b) into (3.29) and rearranging, one obtains:

$$(\sigma^2)^i = \frac{1 - 2^{2H-2}}{2^{2H}} E\{(f^{i-1}_{j+1} - f^{i-1}_j)^2\}, \tag{3.30}$$

According to Mandelbrot and van Ness [1968], increments of fBm have a variance that obeys $E\{(f(x_0 + x) - f(x_0))^2\} \propto |x|^{2H}$. Using this result in (3.30), one finally obtains:

$$(\sigma^2)^i = \frac{1 - 2^{2H-2}}{2^{2H}} \frac{\sigma^2}{2^{2H(i-1)}} = \frac{1 - 2^{2H-2}}{2^{2Hi}} \sigma^2, \tag{3.31}$$

where σ^2 is an arbitrarily chosen global variance for the process.

Midpoint subdivision was first presented by Fournier et al. [1982]. In their paper, the authors present two methods by which the one-dimensional subdivision technique just described can be used to generate fractal surfaces. The first method uses a polygonal mesh, with midpoint subdivision being used to split polygons into smaller ones. The second method uses the f^i_j samples as offsets over an initially smooth bicubic parametric surface[4]. The parametric surface approach generates excessively smooth terrains and, because of that, the polygonal mesh approach was given prominence in subsequent research. Many computer graphics textbooks, such as Watt [1989] or Foley et al. [1990], only mention the polygonal mesh version of the midpoint subdivision method.

Midpoint subdivision, in the case of polygonal meshes, can be performed with either triangles or quadrilaterals. With triangles, one starts with an initial triangle and subdivides its three edges. The three resulting midpoints are joined with new edges, thereby forming four smaller triangles. The procedure is then applied recursively to each of the smaller triangles. In order to ensure internal consistency, the random number generator is seeded with a key that is unique to every edge. In this way, the application of midpoint subdivision on each of the two triangles sharing a common edge will generate the same midpoint vertex along that edge. Figure 3.8 shows the connectivity of a triangular mesh as it undergoes one step of midpoint subdivision. With quadrilaterals, one generates the centre point of each quadrilateral through bilinear interpolation, followed by the addition of the Gaussian random component. As Figure 3.9 shows, the mesh alternates between two grids with an orientation of 45 degrees relative to each other

[4]The polygonal mesh and parametric surface approaches were simultaneously and independently proposed during the Siggraph '80 conference by Carpenter [1980] and Fournier and Fussell [1980], respectively. The proceedings of this conference, however, only contain a one page abstract of each presentation. The authors were then invited to submit a joint paper to the journal Communications of the ACM, which resulted in Fournier et al. [1982]

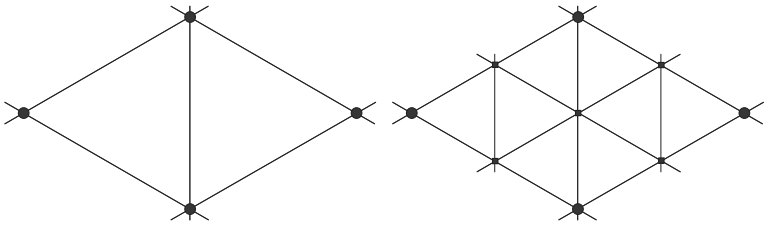


Figure 3.8: From left to right: two stages in the subdivision of a triangular mesh. Circles represent old samples and squares represent newly inserted samples.

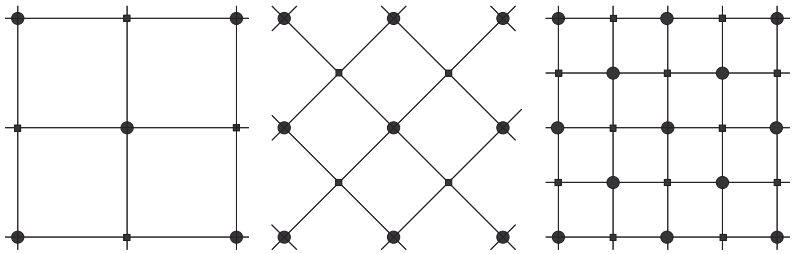


Figure 3.9: From left to right, three stages in the subdivision of a quadrilateral mesh. Circles represent old samples and squares represent newly inserted samples.

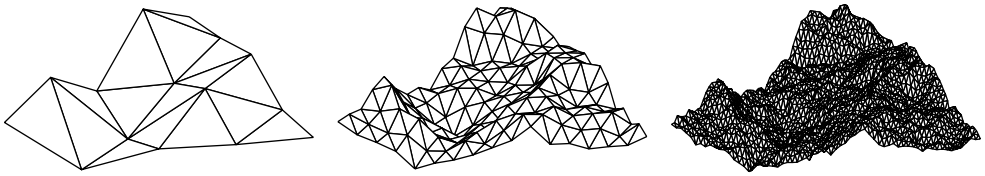


Figure 3.10: Three stages in the triangular subdivision method for the generation of fBm surfaces. The stages correspond, from left to right, to 2, 4 and 6 subdivision levels.

after each iteration is performed. Figure 3.10 shows an example of triangular subdivision after 2, 4 and 6 iterations, respectively.

The total number of iterations for polygonal subdivision is chosen so that a desired level of detail is achieved. The number of iterations can also change dynamically. For example, one can steadily increase the iterations as the camera zooms in on a surface. Cracks may appear, however, if different parts of the polygonal mesh receive different levels of subdivision. One may wish to use such different levels of subdivision to account for distance from the camera. Distant portions of the surface may not be as deeply subdivided as portions that are nearer to the camera. Some additional processing is then required to cover any holes in the mesh. Usually, a special triangulation is used over the boundary between two areas of the mesh with different subdivision levels.

Miller [1986] showed that both the triangle and quadrilateral subdivision strategies of Fournier et al. [1982] are statistically biased. As a result of this bias, the large edges resulting from the initial stages of subdivision remain conspicuously visible in the final surface. He proposed

a biquadratic interpolation method, applied to quadrilateral meshes only, that alleviates these unwanted artifacts. Previously, Mandelbrot [1982] had already shown that midpoint subdivision methods have a deeper flaw. Except for the case $H = 0.5$, the approximation of fBm by midpoint subdivision leads to increments that are neither stationary nor isotropic. The loss of these properties also contributes to the terrain artifacts detected by Miller [1986].

The stationarity and isotropy of discrete realisations of a fBm process were improved with the introduction of the method of *successive random additions* [Voss, 1985]. With successive random additions, a random component d^i is added to all samples f_j^i of a fBm process, not just the midpoint samples, after iteration i is performed:

$$f_{2j}^i = f_j^{i-1} + d_0^i, \tag{3.32a}$$

$$f_{2j+1}^i = (f_j^{i-1} + f_{j+1}^{i-1})/2 + d_1^i. \tag{3.32b}$$

The random variables d_0^i and d_1^i are Gaussian with zero mean and variance $(\sigma^2)^i$. The randomness that was previously assigned to the f_{2j+1}^i sample is now shared between this and the f_{2j}^i sample. For that reason, the variance $(\sigma^2)^i$ is half of what is was in (3.31):

$$(\sigma^2)^i = \frac{1 - 2^{2H-2}}{2^{2Hi+1}} \sigma^2. \tag{3.33}$$

The method of successive random additions improves the statistics of a discrete fBm process because it keeps adding new detail at every iteration. In the midpoint subdivision method, by contrast, once a sample is defined, it remains fixed during all subsequent iterations. Mandelbrot [1988] has also considered other possibilities of extending the midpoint subdivision method. Random variables d^i with a distribution other than the Gaussian distribution were studied together with two new mesh refinement schemes based on either equilateral triangles or hexagons.

A more general form of polygon subdivision is the *generalised stochastic subdivision* method of Lewis [1987]. It can be used to generate a two-dimensional field $f(x_1, x_2)$ whose autocorrelation function $R_f(\tau_1, \tau_2)$ is known. With generalised stochastic subdivision, the new samples f_j^i are interpolated from the old samples f_j^{i-1} through the solution of a system of linear equations. The system of equations involves an autocorrelation matrix that expresses the statistical correlation between the new and the old samples. As with successive random additions, a random component with an appropriate variance is added to the interpolated values for f_j^i.

Traditionally, the generation of fractal surfaces with polygon subdivision had the disadvantage of producing bounded surfaces since these are represented by polygonal meshes. Dixon et al. [1999], however, have been able to apply polygon subdivision in the context of a whole planet. The planet starts out as an octahedron whose eight triangular faces are then recursively subdivided. The newly inserted vertices, resulting from face subdivisions, are projected onto the surface of an enclosing sphere to preserve the spherical shape of the mesh. To economise memory resources, subdivision is carried out only for triangles that intersect with the camera's viewing frustum. Dixon et al. report problems during zoom-in animations, where small triangular islands suddenly pop out of the sea after a new level of subdivision is triggered.

Velho et al. [2001] have employed stochastic polygonal subdivision techniques in the context of subdivision surfaces. A subdivision surface is a polygonal mesh which uses specific subdivision rules such that the mesh converges to a smooth surface in the limit of an infinite number

of subdivisions [Catmull and Clark, 1978; Loop, 1987]. Velho et al. added a random displacement component to the subdivision rules, applied along the normals to the mesh. The user can specify displacements procedurally as a function of position on the mesh and subdivision level to achieve various stochastic effects besides fBm surfaces.

The extension of polygon subdivision methods to three-dimensional stochastic fields is possible but the handling of the data structures becomes more cumbersome. In three dimensions, one can either use a tetrahedral grid or a cubic grid and extend the respective triangular or quadrilateral versions of the midpoint subdivision algorithm. The complexity of such spatial data structures, however, means that procedural methods are to be preferred when generating high dimensional fractal fields.

3.2.4 Procedural Methods

The greatest difference of procedural methods when compared with the previous methods is that the field $f(\mathbf{x})$ can be evaluated independently for any desired value of $\mathbf{x}$. In this way, f behaves as any normal function would and can be implemented as a procedure in a computer program. The first procedural method was the Weierstrass-Mandelbrot function [Mandelbrot, 1977]. This function takes a sine curve as the basis and sums the contributions from a discrete sequence of wavenumbers according to a geometric progression. In one dimension, the function is written in the following way:

$$f_{WM}(x) = \sum_{i=-\infty}^{+\infty} \frac{a_i}{r^{iH}} \cos\left(2\pi r^i x + \phi_i\right). \tag{3.34}$$

The a_i are zero mean Gaussian random variables, all with a variance of σ^2, and the ϕ_i are uniform random variables in the interval $[0, 2\pi]$. A comparison of (3.34) with the Fourier series is useful. In the Fourier series, several sine basis functions (more precisely, complex exponentials) are added together with wavenumbers that follow an arithmetic progression $k_i = ir$, where r is the spacing between successive wavenumbers. In the Weierstrass-Mandelbrot function, the discrete wavenumbers follow the geometric progression $k_i = r^i$. The parameter $r > 1$, in this context, is called *lacunarity*. It expresses how tightly the wavenumbers are packed together along the progression.

The most commonly used value for r is $r = 2.0$, which causes the r^i wavenumbers to be successive *octaves* along the spatial frequency spectrum. Worley [2003], however, advises that r should be slightly offset from 2.0 to avoid the artificial alignment of surface features across scales. In the same spirit, the random phases ϕ_i are introduced to destroy correlations between the sine curves so that f_{WM} can exhibit a better behaviour as a stochastic function.

The PSDF of the Weierstrass-Mandelbrot function can be easily obtained by computing the autocorrelation function for f_{WM} and applying the Wiener-Khintchine theorem:

$$G_{f_{WM}}(k) = \frac{\sigma^2}{4} \sum_{i=-\infty}^{+\infty} \frac{\delta(k - r^i)}{r^{2Hi}}, \tag{3.35}$$

where the $\delta(k)$ are Dirac delta functions. This is a discrete PSDF that only takes discrete values at the wavenumbers r^i. One can approximate $G_{f_{WM}}$ as a continuous PSDF by performing a

moving average over consecutive wavenumbers (see Berry and Lewis [1980] for details):

$$G_{f_{WM}}(k) \approx \frac{\sigma^2}{4 \ln r} \frac{1}{k^{2H+1}}. \tag{3.36}$$

The approximation becomes exact in the limit when $r \to 1$. The PSDF in (3.36) is clearly fractal since it obeys (3.26) and the fractal dimension is given, as usual, by $D = n+1-H = 2-H$ in one dimension. The Weierstrass-Mandelbrot function is, therefore, a good representation of a fBm field. Computation of f_{WM} with (3.34) requires that the series be truncated to a finite number of terms since, by definition, i would have to range over all integers. A discussion of how to estimate appropriate minimum and maximum values of i to use in procedural methods, like the present one, is deferred until the end of this section.

The Weierstrass-Mandelbrot function extends to three dimensions by performing the summation of separable sine waves along the three coordinate axes simultaneously:

$$f_{WM}(x_1, x_2, x_3) = \sum_{i_1=-\infty}^{+\infty} \sum_{i_2=-\infty}^{+\infty} \sum_{i_3=-\infty}^{+\infty} \frac{a_{i_1,i_2,i_3}}{r^{(i_1+i_2+i_3)H}} \times$$
$$\times \cos(2\pi r^{i_1} x_1 + \phi_{i_1}) \cos(2\pi r^{i_2} x_2 + \phi_{i_2}) \cos(2\pi r^{i_3} x_3 + \phi_{i_3}). \tag{3.37}$$

If the summation along each coordinate axis is carried out for m terms, the complexity of (3.37) becomes $O(m^3)$. It is evident from this result that the Weierstrass-Mandelbrot function does not scale well with increasing dimensions. For three dimensions, in particular, f_{WM} is already quite expensive to compute. Ausloos and Berman [1985] proposed a different multivariate extension of the Weierstrass-Mandelbrot function:

$$f_{WM}(\mathbf{x}) = \sum_{i=-\infty}^{+\infty} \sum_{j=1}^{J} \frac{a_{i,j}}{r^{iH}} \cos(2\pi r^i (\mathbf{d}_j \cdot \mathbf{x}) + \phi_{i,j}), \tag{3.38}$$

where $\mathbf{d}_j$ is a random unit vector. This method works by summing together, for each wavenumber r^i, the contributions of J one-dimensional sine functions that point along uniformly distributed random directions on a hyper-dimensional sphere. The complexity of (3.38) in the three-dimensional case, however, does not improve significantly on the complexity of (3.37). To obtain a stochastic function that is fairly isotropic one has to add a large number J of sine functions for each wavenumber. In conclusion, a more efficient functional approximation to an fBm field must be sought even if it is found not to model the statistics of fBm as well as f_{WM} does. A first hint at such an efficient approximation was given by Perlin [1985] when he proposed the following function:

$$f_P(\mathbf{x}) = \sum_{i=-\infty}^{+\infty} \frac{1}{2^i} f_N(2^i \mathbf{x}). \tag{3.39}$$

The stochastic function f_N is a procedural noise function The function (3.39) is independent of the dimension n. The dependence on n is hidden inside the evaluation of the noise function f_N. Assuming f_N can be evaluated efficiently for $n = 3$ dimensions then f_P becomes a useful stochastic function for computer graphics.

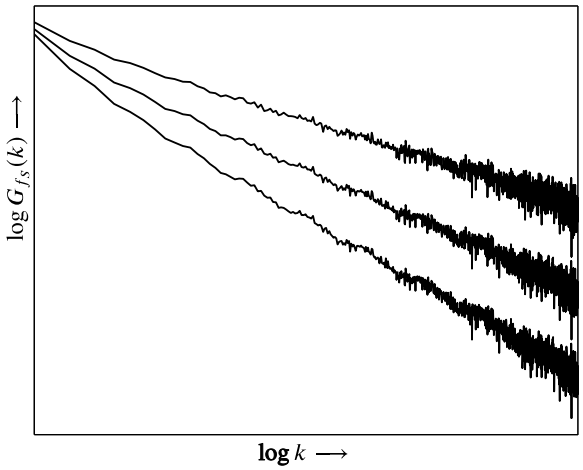


Figure 3.11: The PSDF of the rescale-and-add method with Hurst coefficients $H = 0.2$ (top), $H = 0.5$ (middle) and $H = 0.8$ (bottom). The lacunarity is $r = 2.0$.

Perlin suggested that his function could be used to simulate fields with fractal dimension $D = 1.5$. In reality, f_P has dimension $D = 1.0$. This is a borderline fractal (refer to the discussion on page 40 about this issue). Saupe [1989] extended the stochastic function f_P in order to accommodate any desired fractal dimension by inserting the lacunarity r and the Hurst coefficient H into (3.39):

$$f_S(\mathbf{x}) = \sum_{i=-\infty}^{+\infty} \frac{1}{r^{iH}} f_N(r^i \mathbf{x}). \tag{3.40}$$

Comparing (3.40) with (3.39), it is seen that Perlin implicitly used a lacunarity $r = 2.0$ and a Hurst coefficient $H = 1.0$, hence the fractal dimension $D = 1.0$ that was mentioned before. Figure 3.11 shows the PSDF of a one-dimensional field $f_S(x)$, in a log-log graph, for several values of the Hurst coefficient. The plots of $G_{f_S}(k)$ are approximations to lines of constant slope $\beta = -2H - 1$. The reason why G_{f_S} deviates from the PSDF of a perfect fBm field is related to the lack of statistical stationarity of f_S. This lack of stationarity is, in turn, related to the way the noise function f_N is built. Saupe called his method the *rescale-and-add* method due to the way it repeatedly scales and adds several copies of a same noise function. Examples generated with this method can be seen in the one-dimensional graphs of Figure 3.3 and the surface of Figure 3.14(a).

The complexity of the rescale-and-add method is $O(m)$, if only m terms are used in the series, which compares favourably with the $O(m^3)$ complexity of the Weierstrass-Mandelbrot function. Table 3.1, from Saupe [1989], shows the number of floating point operations per point required to compute a fBm field on a 1024^3 cubic grid with successive random additions (Section 3.2.3), the Weierstrass-Mandelbrot function or the rescale-and-add method. The method of successive random additions is the fastest but can only be used to compute the field over *all* grid points simultaneously. The other methods are local and can compute any point independently. Clearly, the rescale-and-add method is the most efficient for point-wise evaluation.

Method	**flops / point**
Successive random additions	48
Weierstrass-Mandelbrot function	3000
Rescale-and-add	650

Table 3.1: The cost of computing fBm on a three-dimensional grid (adapted from Saupe [1989]).

The computation of three-dimensional fBm with (3.40) leads to a field that is not isotropic. The reason is, again, related to the nature of the procedural noise function. It is possible to achieve near isotropy by employing a method similar to (3.38) whereby one-dimensional noise functions are summed together, with directions randomly chosen around a sphere, but this increases costs considerably as many such noise functions have to be considered. Worley [2003] suggests a more practical compromise that consists of introducing a random rotation in each term of the series (3.40). This will not make the field isotropic but at least it will prevent grid artifacts at different scales from adding up coherently.

For the evaluation of the rescale-and-add method to be computationally feasible the infinite series of terms must be replaced with a finite series where $i_0 \leqslant i \leqslant i_1$. The lower bound i_0 gives the scale of the largest features in the stochastic surface. It was already seen in Section 3.2.1 that fractal fields are only valid within an appropriate range of scales. It seems natural, therefore, to choose a scale L above which the field ceases to be fractal. One must then choose i_0 so that r^{-i_0} is smaller than L or, equivalently, $i_0 > -\log L / \log r$.

The higher bound i_1 gives the scale of the smallest features, which is to say that it gives the largest wavenumber r^{i_1} that is present in the surface. Considering that the surface is being point sampled, it is important to limit this higher frequency so that aliasing is avoided. And because the surface is being seen through a perspective transformation, with samples initially placed at the pixel positions and then projected onto the terrain, the highest wavenumber will also depend on the distance to the camera. Distant portions of the surface must be severely restricted in their spectral content because samples there are placed very far apart. Portions of the surface close to the camera, on the other hand, can afford to have a few more frequency terms added to them because samples are now packed much closer together. Musgrave [2003e] gives the following estimate for the maximum term i_1 as a function of distance d to the camera:

$$i_1(d) = -\log(\tan fov/res)/\log r - \log d/\log r - 3.5, \tag{3.41}$$

where fov is the camera's field of view in radians and res is the resolution across the image plane. The constant 3.5, according to Musgrave, is related to the procedural noise function.

With (3.41), an effective control over the frequency content of the model is achieved. This amounts to performing an anti-aliasing filtering operation with a spatially varying filter. Such a sophisticated filtering operation would be very costly to perform with a polygonal model but fits easily into a procedural model. One just has to replace (3.41) for the upper bound of the series in (3.40). The estimate i_1 is first split into an integer part $\lfloor i_1 \rfloor$ and a fractional part $i_1 - \lfloor i_1 \rfloor$. These are then replaced in (3.40) in the following way:

$$f_S(\mathbf{x}) = \sum_{i=i_0}^{\lfloor i_1 \rfloor} \frac{1}{r^{iH}} f_N(r^i \mathbf{x}) + \frac{i_1 - \lfloor i_1 \rfloor}{r^{(\lfloor i_1 \rfloor + 1)H}} f_N(r^{\lfloor i_1 \rfloor + 1} \mathbf{x}). \tag{3.42}$$

The way the rescale-and-add method is specified above avoids the popping artifact that would occur if a new term was suddenly included in the series, which could be caused, for example, by zooming in on the surface. Any new frequency component is gradually eased in into the series by way of the term on the right hand side of (3.42). It should be said, however, that the estimate (3.41) is much too conservative. This causes the surface to become noticeably smooth as one gazes towards the horizon.

3.3 Terrain Models

Fractals have a limited validity in modelling natural phenomena. As previously seen, when applied to real stochastic processes, fractal laws of self-similarity are only valid within an appropriate range of scales. In the case of landscape modelling, fractal geometry is most appropriate to generate geologically new terrains, like alpine mountain ranges, while older types of terrain, where erosion processes have played a dominant role, are not as amenable to a fractal modelling approach. This has led many researchers to experiment with other techniques for terrain modelling. Such techniques either introduce modifications to the fBm model or add additional effects on top of it or avoid fBm altogether. Examples are presented here of terrain modelling techniques that follow each of these three different strategies.

3.3.1 Terrain Synthesis by Example

Techniques that synthesise new terrains from a set of terrain examples are an extension of texture synthesis by example techniques [Kwatra and Wei, 2007]. A heightfield terrain can be considered as a form of image texture where the terrain height is equivalent to the gray scale intensity of an image pixel. This analogy makes it possible to use simple image painting applications as a way to create new terrains. Painting applications, however, are labour intensive and the realism of the resulting painted terrain depends on the skill of the user to create realistic terrain features. Terrain synthesis by example techniques, while still requiring some input from the user, derive most of their surface features from digital elevation models of real terrains. Texture synthesis by example methods cannot be used for terrain synthesis without modification for several reasons. Texture synthesis methods, for example, do not generally create large scale coherent features in textures and can also generate contours with a sharp variation in image intensity. These contours are commonly found in many natural textures but create unnatural cliffs in terrains. For these reasons, synthesis by example methods specially adapted to terrains have been developed. All these methods work with gridded terrain representations because examples of real terrain data are usually stored in that form.

Ong et al. [2005] apply genetic algorithms to synthesise new terrain. The horizontal plane is partitioned into several domains by the user and different types of terrain are assigned to each domain. A chromosome gives a candidate solution for the terrain within a domain. A chromosome is further subdivided into genes. A gene is a small circular area that is blended with other nearby genes. Each gene encodes a pair of scaling and rotation transformations that are applied to the terrain within its area. The terrain contained in the area of a gene is assigned randomly from the example terrains that were selected by the user for the domain where the gene is located. A genetic algorithm then introduces random mutations and cross-

overs into the genes in an attempt to minimise a cost function that measures the quality of the resulting terrain. The cost function seeks to avoid excessive discontinuities between domains and to obtain a good match with the example terrains. The genetic modifications are expressed as modifications to the sequence of genes in a chromosome and to the pair of rotations and scalings of each gene. This genetic approach is conceptually interesting but in practice it has a very slow convergence towards the final terrain. The slow convergence is characteristic of genetic algorithms when used to solve optimisation problems. Furthermore, it is possible for the algorithm to become locked in a local minimum of the cost function.

Brosz et al. [2006] use Multi-Resolution Analysis (MRA) to combine the large scale detail from one terrain with the small scale details from another terrain. A MRA is a cascade of wavelet analysis filters that, when applied to a terrain, extract successively lower frequency (larger scale) detail maps, leaving only the lowest frequency (largest scale) terrain features [Strang and Nguyen, 1996]. The corresponding MRA synthesis filters then take the sequence of detail maps and combine them progressively with the low-pass filtered terrain to reconstruct the full scale terrain. Brosz et al. propose that by feeding the sequence of detail maps from a terrain into the large scale features from another terrain as part of a MRA synthesis process, it is possible to generate a new terrain that combines characteristics from both the original terrains. Instead of simply superimposing the detail maps on top of the large scale terrain, a modified image quilting algorithm is used to create a randomised distribution of details over the terrain [Efros and Freeman, 2001]. The image quilting algorithm partitions an input texture into blocks and creates a new randomised texture by combining blocks from the input texture in raster scan order. At each step, a block is placed in the output texture by randomly selecting from blocks in the input texture. The blocks that are considered for selection are the ones that have the greatest similarity with the neighbouring blocks in the output texture (relative to the new output block) that have already been placed.

Zhou et al. [2007] have proposed a terrain by example method that is derived from geomorphology studies. The user first sketches the ridge lines that he wishes the resulting terrain to have and then selects an example terrain. The method constructs a terrain that conforms to the ridges indicated by the user and with the surface features of the example terrain. A geomorphology algorithm is used to extract two acyclic graphs of ridge lines from both the user's sketch and the example terrain [Chang et al., 1998]. A matching is then made by traversing the graph of the user's sketch in a breadth-first manner and finding similar ridge features in the graph of the terrain. The final terrain is created by assembling patches from the example terrain based on the matches that resulted from the graph traversal. The graphcut algorithm is used to join all the patches while minimising the discrepancies at the seams between patches [Kwatra et al., 2003]. Any remaining discrepancies are smoothed by solving a Poisson equation that is derived from the terrain height in the neighbourhood of each seam. The user, rather than drawing a network of ridge lines, can instead draw a network of valleys and the method will work similarly by matching with valleys from the example terrain. The results obtained with this method are impressive. Figure 3.12 shows a 2000×2000 resolution terrain generated with digital elevation data of the Grand Canyon as the example terrain. The valleys of the resulting terrain have been constrained to follow roughly the outline of the greek letter 'λ', placed within a circle.

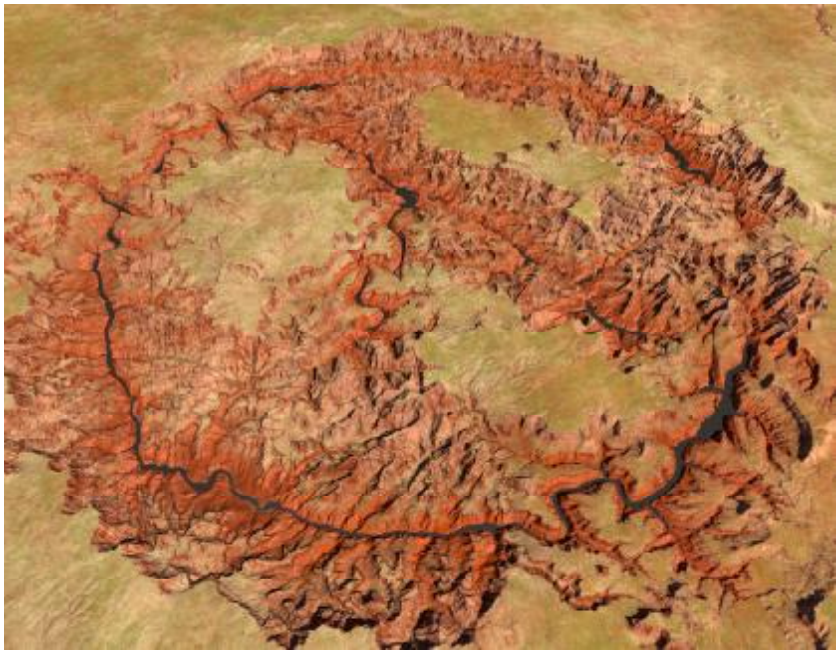


Figure 3.12: Terrain synthesised from an elevation map of the Grand Canyon and constrained to follow lines sketched by the user (image courtesy of Howard Zhou [Zhou et al., 2007]).

3.3.2 Terrain Erosion and River Network Models

Water has an important role in the shaping of natural terrain. Several terrain models have been published that attempt to create more realistic landscapes by simulating the erosion processes caused by water. Such erosion processes are global in nature. They rely on the passing of information between different areas of the terrain. Erosion processes also have memory. Any particular portion of terrain records the history of all erosion events that occurred in it. In the case of terrains with river networks, for example, there are interdependencies that can be established across very large distances on the terrain. Terrains with erosion and river networks, therefore, are always represented as data grids. In this way, it becomes possible to establish rules for the dependency due to erosion between any pair of points on the grid. Figure 3.13 shows an example of a terrain after the application of an erosion and water transport model, resulting in a network of rivers and lakes.

There are two distinct approaches to the modelling of eroded terrains. One approach incorporates the generation of erosion features and river networks directly into the synthesis of the terrain. The terrain comes out of the synthesis algorithm complete with river beds and alluvial plains. The second approach decouples the generation of terrain from the formation of erosion features. In a first step, the terrain is generated without erosion, using any of the terrain synthesis models that have been presented in Section 3.2. In some cases, the initial terrain can just be a completely flat plain. In a second step, erosion is introduced into the model by simulating the action of rain. The rain will carve out the terrain erosion features and form rivers through which it flows out of the model.

Figure 3.13: A terrain after the application of an erosion and water transport model (image courtesy of John Beale).

Kelley et al. [1988] build a terrain with rivers by first defining the river network as a structural model. They draw extensively from geomorphological studies to derive probabilistic laws that describe the formation of river tributaries. The terrain is then created according to a model of a surface under tension with the river network functioning as a set of constraint points. The terrain is forced to interpolate these constraint points and, in doing so, is able to integrate the river network into its final shape. Chiba et al. [1991] describe a similar method but using ridge lines instead. A hierarchy of ridges is created through recursive subdivision and the terrain is interpolated from them. Prusinkiewicz and Hammel [1993] extend the triangular mesh version of the fractal midpoint subdivision method by Fournier et al. [1982] to account for rivers. Special subdivision rules are introduced for triangles that contain sections of a river in order to maintain its proper connectivity after subdivision. As the subdivision progresses into increasingly smaller triangles, the river becomes a fractal curve that is seamlessly integrated into the landscape. For completeness, it should be said that Mandelbrot [1988] had already attempted a similar river subdivision mechanism based on hexagonal meshes. His method. however, did not include randomness and generated river networks that were far too regular, these networks being examples of deterministic fractals on a par with Koch curves, for example.

Erosion models that carve out the terrain after it has been built use a computational approach based on cellular automata. The terrain is decomposed into many cells. Each cell stores an internal state and simple update rules are defined that allow a cell to change its state by receiving messages from its neighbour cells. Musgrave et al. [1989] describe the state of a cell by the following internal variables:

- The altitude relative to sea level.
- The amount of water that is currently above the cell.

- The amount of sediment contained in the water.

Rules are now defined that allow the above quantities to change dynamically. For example, a percentage of the amount of water over a cell can be transferred to neighbouring cells but only to those that are situated at lower altitudes. When water is transferred to another cell, part of the sediment goes with it but another part stays permanently in the original cell and is used to increment its altitude value. Conversely, the altitude of a cell can sometimes be decremented and converted into sediment that is then transported to other cells. Musgrave et al. also include rules that allow another erosion mechanism called *thermal weathering* to be integrated with water erosion. Thermal weathering is responsible for the erosion of very steep slopes caused by the continuous changes of temperature induced by the cycles of day and night. The sediments that fall off these steep slopes accumulate below in the form of *talus slopes*. Later, Roudier et al. [1993] extended the work of Musgrave et al. to include the effect of rock strata with different geological properties. These strata can be folded or fractured as a pre-processing step to simulate the effect of internal geological stresses on the terrain. The erosion mechanism is then invoked, using a set of rules that attempt to model the effects caused by gravity creep, transport of silt, chemical dissolution and alluvial deposition.

Nagashima [1997] defines the initial terrain to be a volume composed of several horizontally stacked rock strata with different properties, similar to the approach of Roudier et al. [1993] for the specification of the initial terrain. Only the upper part of the rock strata is initially visible. A river, constructed by midpoint subdivision of a fractal curve, is then placed over the terrain and allowed to slowly dig into the strata by converting terrain into sediment that is then removed from the system. Due to the different geological properties of the strata, Nagashima is able to simulate terrain structures that are similar to the Grand Canyon, with a succession of terrain steps that progress towards the river bed at the bottom. Chiba et al. [1998] model the movement of water over the terrain as a particle system. The acceleration of each water particle is based on the local slope of the terrain underneath it. As the water particles flow over the terrain, they take sediment away and deposit it further downstream. Ito et al. [2003] model the initial terrain as a volumetric array of voxel elements. Each voxel is connected to its neighbours by a network of joints. Ito et al. then introduce geological faults into these joints and run stability analyses to determine which voxels have become unstable and must be removed from the terrain. They are able to model rocky cliffs that have been shaped by gravity and by geological faulting processes.

Terrain erosion models based on cellular automata employ a physically based approach to the simulation of erosion. The rules that update the state of each cell are over-simplified but they nevertheless follow a physical reasoning. These models work very slowly by iteratively eroding the terrain away. A decision must be made at some point to stop the iterations, otherwise the terrain would completely disappear due to sediment transport. The study of terrain erosion models continues to be an active area of research. Just to give an example, during the Eurographics 2005 Workshop on Natural Phenomena no less than three papers were presented on this topic. In light of the fact that this thesis is mostly interested in procedural models for terrain, as opposed to heightfield terrains defined over grids, an exhaustive explanation of all recent developments in terrain erosion will not be pursued here.

3.3.3 Heterogeneous Procedural Models

It is common practice in computer graphics to take some formal mathematical model and introduce variations that destroy its formalism but that, at the same time, expand the range of visual results that it can produce. The case of fractional Brownian motion is no exception. All techniques that have been presented in Section 3.2 have attempted, with varying degrees of success, to model a fBm field as accurately as possible. Fractional Brownian motion, however, does not provide all the answers for the realistic modelling of natural terrains. The greatest shortcoming of a fBm process is that it generates surfaces that look the same everywhere. Stated more rigourously, the fractal dimension of fBm is equal at all points and the outcome of a fBm process is said to be a *mono-fractal*. All the techniques that are described in this section attempt to introduce new features into the generation of a fBm process. In doing so, many of them can no longer claim to produce results that obey a model of fractional Brownian motion and, what is more, cannot even be formally classified as fractal. These new techniques, however, are still examples of stochastic fields since randomness continues to play an important part in the synthesis process.

Perhaps the earliest variation on a fBm field was presented by Haruyama and Barsky [1984] in the context of procedural texture synthesis. Haruyama and Barsky employ a midpoint subdivision method to generate two-dimensional textures that are subsequently applied over parametric patches. In an attempt to have additional control over the shape of the resulting textures, they propose that different Hurst coefficients H_i be used for each stage i of the subdivision. This amounts to having a process with a PSDF whose spectral falloff exponent β changes across the spectrum. The same authors also remove the restriction $0 < H < 1$ for greater flexibility. Although these modifications were applied to a polygonal midpoint subdivision method, their application to a procedural method is straightforward. Saupe [1989] and Musgrave et al. [1989] suggest that the Hurst coefficient can be changed across the domain by supplying some function $H(\mathbf{x})$. This allows the generation of terrain with spatially varying roughness by having different areas with different values of H, with smooth interpolation of the coefficient occurring between these areas. Both techniques for controlling H can easily be combined to produce a sequence $H_i(r^i\mathbf{x})$ of Hurst functions, where each function is applied to the corresponding i-th term of the series shown in (3.40)[5]. These modifications to the Hurst parameter H generate terrains that are *multi-fractal*, i.e. whose fractal dimension changes across the terrain and with different scales.

Besides changing the characteristics of the terrain across scales and across the domain, heterogeneous terrain models also attempt to model erosion processes in a limited way. The original fBm formulation produces peaks and valleys with the same characteristics. If one inverts the terrain along the vertical direction, the peaks become perfect valleys and vice-versa. This uniformity of fBm can be perceived in Figure 3.14(a). One would expect, however, that valleys should be much smoother than peaks because of the transport and accumulation of sediments. Voss [1985] suggests using a simple power scaling $(f(\mathbf{x}))^\alpha$ to change the statistics of f with altitude. With $\alpha > 1$, peaks become exaggerated and valleys become flatter. With $0 < \alpha < 1$, one obtains the impression of deep valleys carved by river erosion on an otherwise flat plain.

[5]For simplicity, procedural methods are considered in this section that employ an infinite series of frequencies. Bandwidth control through the choice of the i_0 and i_1 frequency bounds, as explained in Section 3.2.4, can easily be incorporated into these methods at a later stage.

Musgrave [2003d] goes further and presents two hybrid additive-multiplicative frequency cascades that allow a finer control of the terrain statistics with altitude.

To better understand Musgrave's statistics-by-altitude models, it is helpful to first recast the rescale-and-add method (3.40) in an iterative form:

$$f_S^i(\mathbf{x}) = f_S^{i-1}(\mathbf{x}) + f_N(r^i\mathbf{x})/r^{iH}, \tag{3.43}$$

where f_S^i is the output of the process after the i-th iteration. For the models by Musgrave to work correctly it is also necessary that the noise function f_N return only positive values. This can be accomplished by adding a constant offset ξ to the result returned by $f_N(\mathbf{x})$. The first of the hybrid models takes the current global height of the terrain as a weighting factor when adding new frequency contributions and uses the positive noise function:

$$\begin{aligned} f_{M1}^i(\mathbf{x}) &= f_{M1}^{i-1}(\mathbf{x}) + f_{M1}^{i-1}(\mathbf{x})\,(f_N(r^i\mathbf{x}) + \xi)/r^{iH} = \\ &= f_{M1}^{i-1}(\mathbf{x})\big(1 + (f_N(r^i\mathbf{x}) + \xi)/r^{iH}\big). \end{aligned} \tag{3.44}$$

If the terrain, at iteration $i-1$, has an altitude that is close to zero for some point $\mathbf{x}$, new frequency contributions will be heavily damped and the terrain will tend to remain smooth at that point during subsequent iterations. This will create the effect of rugged mountains surrounded by flat low lying valleys as Figure 3.14(b) shows. Valleys, however, should be smooth at all altitudes, not just at sea level. Musgrave proceeds to present the second of his hybrid models in an attempt to answer this observation. The new model includes a weighting factor $w^i(\mathbf{x})$ that is iterated simultaneously with the terrain:

$$w^i(\mathbf{x}) = w^{i-1}(\mathbf{x})(f_N(r^i\mathbf{x}) + \xi)/r^{iH}, \tag{3.45a}$$

$$f_{M2}^i(\mathbf{x}) = f_{M2}^{i-1}(\mathbf{x}) + w^i(\mathbf{x}). \tag{3.45b}$$

The weighting factor is initialised with $w^0(\mathbf{x}) = 1.0$ and is clamped to the range $0 \leqslant w^i(\mathbf{x}) \leqslant 1$ after the computation of (3.45a). Figure 3.14(c) shows the application of this procedural model. Smooth valleys are now more prevalent in comparison with Figure 3.14(b) and exist at several altitudes. The statistics-by-altitude methods (3.44) and (3.45) can only be loosely classified as multi-fractals because, unlike the rescale-and-add method, there is no direct control over the fractal dimension. Multi-fractals with a solid mathematical grounding are based on multiplicative cascades of frequencies [Evertsz and Mandelbrot, 1992; Mannersalo et al., 2002]. Musgrave [2003d] attempted to use these multiplicative models for computer graphics and found them to be too unwieldy. Multiplicative cascades are prone to numerical divergence and produce terrains with an excessively large dynamic range. Musgrave considers his hybrid additive-multiplicative cascades to be better behaved and presents them as a replacement for pure multiplicative cascade models.

Another possibility for enriching the set of features generated by procedural models is to change the shape of the noise function. This was first suggested by Perlin [1985] when he presented a *turbulence function* that is similar to (3.39) but uses the modulus $|f_N(\mathbf{x})|$ of a procedural noise. The name "turbulence" was chosen because the function produces results that resemble gases in a state of fully developed turbulence[6]. Musgrave [2003d] gives a different

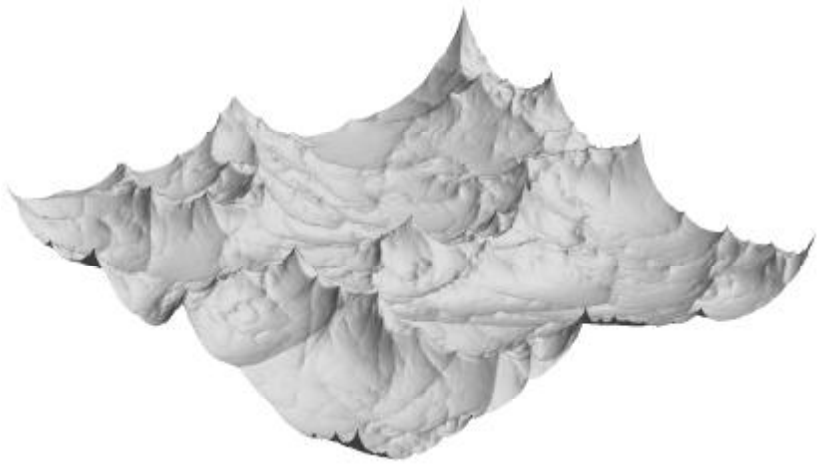
[6]The choice of name is not entirely coincidental, considering that fluid turbulence is one of the best examples of fractals in nature (see Mandelbrot [1975]).

(a) The rescale-and-add method (3.40).

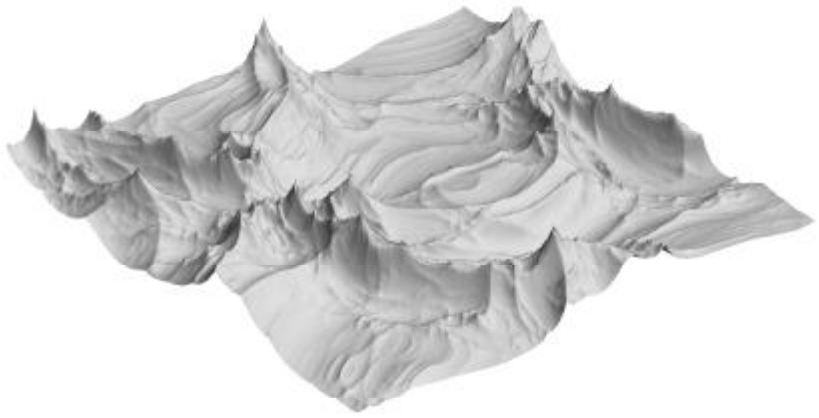
(b) The first statistics-by-altitude method (3.44).

(c) The second statistics-by-altitude method (3.45).

Figure 3.14: Comparison of three procedural terrain synthesis methods that use the same Hurst parameter $H = 0.8$. The statistics-by-altitude methods have a noise offset $\xi = 0.6$.



(a) Without domain deformation.



(b) With domain deformation.

Figure 3.15: The second statistics-by-altitude method (3.45) with a ridged noise function.

version of a modified noise function: $(1 - |f_N(\mathbf{x})|)^2$. This new noise function creates the appearance of a ridged terrain due to the presence of derivative discontinuities at isolated points in the domain. When used in conjunction with any statistics-by-altitude method, Musgrave's basis function is quite successful in creating a terrain with a cascade of ridges over many scales as Figure 3.15(a) shows.

Yet another possibility of creating more general terrain surfaces is to apply domain deformation, similar to what was described in Section 2.3.3 for implicit surfaces. In the case of a two-dimensional heightfield terrain, one can write a terrain function $f(\mathbf{d}(\mathbf{x}))$ where $\mathbf{d} : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ is a domain mapping function. The initial terrain shape generated by $f(\mathbf{x})$ is deformed through this mapping because for a given point $\mathbf{x}$ the terrain height is associated with another point $\mathbf{d}(\mathbf{x})$, different from $\mathbf{x}$. The domain mapping function is usually a procedural noise function in vector form, constructed by assembling two scalar noise functions into a vector. One can also make $\mathbf{d}$ be equal to the gradient $\nabla\phi(\mathbf{x})$ of some scalar noise function ϕ. This creates

Advantages	
Flexibility	Functions can generate many different shapes.
Infinite Domain	A function can be evaluated everywhere in space.
Minimal Storage Size	Only the storage for the function's parameters is required.
Built-in Level of Detail	A self-similar function can choose a range of scales that is appropriate for any given viewing distance.
Disadvantages	
Computational Cost	Useful functions for natural surfaces are expensive to compute.
Large parameter space	A user must control a large set of parameters, many of which may not be intuitive.

Table 3.2: The characteristics of procedural surface models.

flow effects that can mimic some geomorphological deformations of real terrains. A gradient flow deformation is used in Figure 3.15(b), which shows an example of domain deformation applied to the ridged terrain of Figure 3.15(a).

3.4 Stochastic Implicit Surfaces

The application of the stochastic models described in this chapter as implicit surface generating functions leads to a class of *stochastic implicit surfaces*. Any model of a three-dimensional stochastic field can be used as an implicit function although, as previously seen, some models are more easily extensible to three dimensions than others. Given a three-dimensional stochastic field f, one wants to find the implicit surface made of those points for which $f(\mathbf{x}) = 0$ is true. The fundamental choice is whether to use models that generate discrete data or procedural models. Discrete models, such as the ones that implement spectral synthesis or the ones that simulate terrain erosion, give rise to the discrete implicit surfaces that were the subject of Section 2.2.1. In this thesis, procedural models are preferred for several reasons. Procedural models can be evaluated at any point in space, which makes it easy to generate infinite planar landscapes or landscapes over the surface of an arbitrarily large sphere. Procedural models are quite compact and only need to store a set of function parameters – there is no need to store potentially large data grids or polygonal meshes. Procedural models can have a simple built-in mechanism to provide adaptive level of detail as was demonstrated with equation (3.42) for the case of the rescale-and-add method. This level of detail mechanism makes it possible to zoom in on a stochastic implicit surface and never run out of details to look at since new details are created on the fly in response to the decreasing viewing distance. Even though procedural models for natural surfaces are not physically accurate, it is possible to simulate many different types of surface, according to the morphological modelling principle of Fournier [1989], by experimenting with different combinations of surface-generating functions.

Stochastic implicit surface generation through procedural models also has its disadvantages. The computational expense associated with the functions that are useful for stochastic modelling is an important factor. Another problem is the complexity involved in realistically modelling the surface of a natural object by having to specify a possibly large set of control paramet-

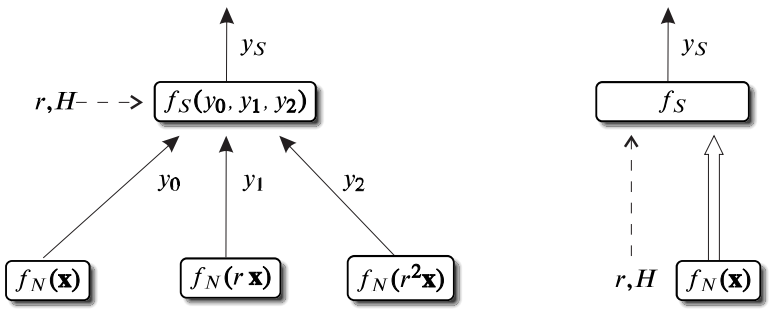


Figure 3.16: On the left, a hypertexture modulation function implementing a rescale-and-add method that uses three layers of noise. On the right, a simplified representation of the same function. The dotted arrows show the input of parameters.

ers. Standard fractal surface generating functions are specified with only a few parameters, like the Hurst parameter H or the granularity r, but these functions are seldom used in isolation. A realistic looking surface will often use several procedures, combined through functional addition, multiplication and composition. The total number of parameters that are required for setting the final surface may be considerable. Table 3.2 summarises the advantages and disadvantages of using procedural models.

The generation of implicit surface terrains on the sphere with the hypertexturing model of Section 2.4 combines the seed function f_0 of an implicit sphere with one or more procedural stochastic models f_i, expressed as modulation functions in the hypertexture hierarchy that is exemplified in Figure 2.15. Figure 3.16 shows how the rescale-and-add method of Saupe [1989] would be implemented as a hypertexture for the case where only three octaves of procedural noise are used. The output of this rescale-and-add method is then fed into the remainder of the hypertexture hierarchy that is not shown in the figure. Other procedural methods, like the statistics-by-altitude methods of Musgrave [2003d], can be implemented in a similar manner. The three instances of procedural noise $f_N(\mathbf{x})$, $f_N(r\,\mathbf{x})$ and $f_N(r^2\mathbf{x})$, shown in Figure 3.16, are source functions and constitute leaves of the hypertexture hierarchy. The expression $f_N(r^i\mathbf{x})$, for $i > 0$, corresponds to a basic form of domain deformation that uniformly shrinks the noise function across its domain by a factor of r^i. The output of the modulation function that implements the rescale-and-add method is given by $f_S(y_0, y_1, y_2) = y_0 + r^{-H} y_1 + r^{-2H} y_2$. The frequency cascade methods, such as the rescale-and-add method, can be more succinctly expressed as macro-functions in the hypertexture hierarchy, as shown on the right of Figure 3.16. A macro-function invokes internally a noise function f_N several times and receives a set of parameters, which in this example are just the H and r parameters. The inclusion of an additional parameter that gives the distance of point $\mathbf{x}$ to the viewpoint can be used by the macro-function to determine how many instances of f_N should be invoked in order to provide adaptive level of detail.

Figure 3.17 shown an example of a procedural hypertextured terrain on the surface of the sphere, which uses the rescale-and-add method applied to the ridged noise function $(1 - |f_N(\mathbf{x})|)^2$ proposed by Musgrave [2003d]. Several images are shown in this figure, corresponding to an increasing number of noise octaves added to the surface. The domain deformation

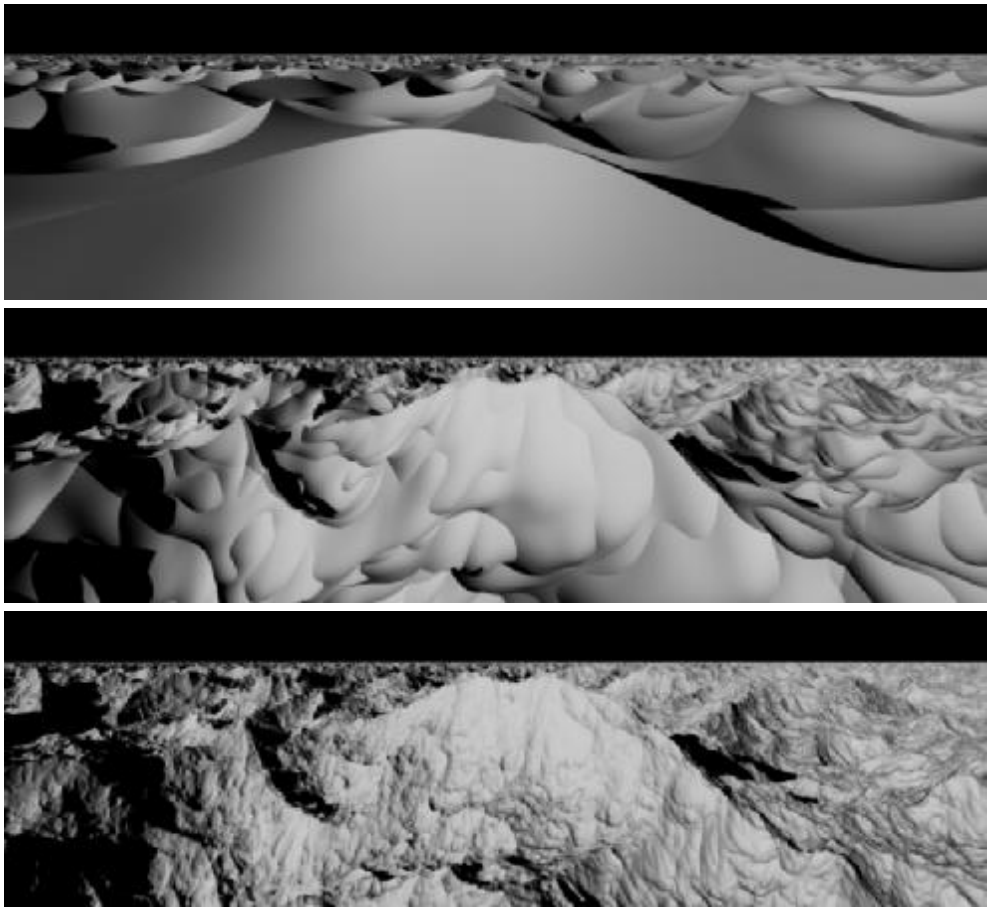


Figure 3.17: From top to bottom, a ridged hypertexture terrain with two, five and eight layers of ridged noise, respectively.

method explained in Section 2.4.3 was used to create a hypertextured surface that is equivalent to a displacement map. The ridged nature of this terrain justifies the initial requirement, presented in Section 2.1, that implicit surfaces should be allowed to have discontinuous derivatives at some points. The surface continuity requirements will have to be restricted to C^2 in Chapter 8 in order to apply connectivity testing to implicit surfaces. The vector field surface deformation technique, subsequently explained in Chapter 9, will allow ridges to be reintroduced into the terrain.

CHAPTER 4

Procedural Noise Functions

PROCEDURAL noise functions constitute the basic building block out of which procedural stochastic surfaces can be built. A procedural noise function generates a stochastic field with random fluctuations that have an approximately constant spatial frequency. As previously seen in Section 3.2.4, procedural noise functions form part of both additive cascade models and hybrid additive-multiplicative cascade models that generate fractal surfaces. Procedural noise functions can also be used for surface texturing and for modelling gases [Ebert et al., 2003]. Peachey [2003] gives a list of the basic requirements that a procedural noise function should obey. These requirements are acknowledged in the literature as being necessary to procedurally build stochastic models from the sum of several procedural noise functions:

Procedural evaluation. As their name implies, procedural noise functions should have the ability to evaluate the noise at a point $\mathbf{x}$, independently from any other points. Evaluation should also be possible for any point in space without limitations.

Internal consistency. The requirement of internal consistency that was defined by Fournier et al. [1982] for stochastic models must also apply to noise functions – evaluating a noise function $f_N(\mathbf{x})$ should yield the same value every time the same point $\mathbf{x}$ is invoked.

Zero mean value. A procedural noise function should generate fluctuations around the zero value so that, on average, it does not introduce any bias into a stochastic procedural model. Most noise functions are specifically designed to fulfill this requirement.

Bandlimited spectrum. The power spectrum distribution (PSDF) of a procedural noise function should be bandlimited in nature with most of the spectral content concentrated around a single wavenumber. Without loss of generality, this wavenumber can be $k = 1m^{-1}$. More complex spectra can then be built using additive combinations of scaled and shifted copies of one or more procedural noise functions.

Stationarity and Isotropy. Procedural noise functions should generate stochastic fields that are both stationary and isotropic. Any intended non-stationarity or anisotropy should be explicitly built in by the user by applying transformations to noise functions. Effects of non-stationarity or anisotropy intrinsic to a noise function are regarded as unwanted artifacts and should be as small as possible.

Closed Form Formulae for the Derivatives. The first derivatives of a procedural noise function should have a closed form expression so that the gradient vector can be readily computed. This is then used when evaluating the normal vector of hypertextured surfaces.

For Morse theory to be applied (refer to Chapter 8), it is also important that expressions for second derivatives be available.

Computational Efficiency. The evaluation of a procedural noise function should not be a computationally intensive task, considering that it will have to be evaluated many times and for different points in the course of an image rendering. This requirement often conflicts with previous requirements concerning the spectral quality of the noise function. Noise functions with a good spectrum are often more expensive to compute.

This chapter presents the three most important noise functions from the literature that are used in this thesis. Section 4.1 begins by presenting a new procedural noise model that was developed as part of the research for this thesis and which unifies the three noise functions under consideration [Gamito and Maddock, 2007d]. Section 4.2 explains the methods that are used in this chapter to study and to characterise the spectral behaviour of procedural noise functions. Sections 4.3 to 4.5 then present the three noise functions. For each of the noise functions, a characterisation of its spectral properties is given and some possible variations on the basic function are also presented. Another noise function from the literature that has not been used in this thesis is briefly described in Section 4.6. Finally, Section 4.7 concludes the chapter by presenting a summary of the advantages and disadvantages of each noise function.

4.1 A Unified Model for Procedural Noise Functions

Most procedural noise functions that have been developed in the literature follow a common pattern for generating samples $f_N(\mathbf{x})$ of a stochastic field in a procedural manner. This common pattern is explained here in the form of a unified model for procedural noise functions. This noise function model will be used throughout the thesis and has implications in the way that stochastic implicit surfaces are ray cast in a robust manner, as explained in Chapter 5. The description of the unified noise model is given in general terms. The examples of specific noise functions that are built out of this model are then presented in Sections 4.3 to 4.5.

The value of a procedural noise function f_N at some point $\mathbf{x}$ in $\mathbb{R}^3$ depends on the position of $\mathbf{x}$ relative to a discrete but infinite set $S = \{\mathbf{x}_i \in \mathbb{R}^3 : i = 0, 1, 2, \ldots\}$ of node points $\mathbf{x}_i$ that are distributed throughout space. Because S has an infinite number of node points, the evaluation of $f_N(\mathbf{x})$ is feasible when $f_N(\mathbf{x})$ is made to depend only on a small subset $S(\mathbf{x})$ of S. At each location $\mathbf{x}$, the subset $S(\mathbf{x})$ is the finite set of node points in S that surround $\mathbf{x}$ according to some specified criterion. Figure 4.1 exemplifies this for the case where the infinite set S consists of a random distribution of node points.

It is possible to define the value of a procedural noise function f_N at $\mathbf{x}$ as a sum of kernel functions ϕ_l that depend on the displacement vectors between $\mathbf{x}$ and the node points in $S(\mathbf{x})$:

$$f_N(\mathbf{x}) = \sum_{l=1}^{L} \phi_l(\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_N), \tag{4.1}$$

where $\mathbf{d}_i = \mathbf{x} - \mathbf{x}_i$ and $\mathbf{x}_i$, with $i = 0, 1, \ldots, N$, belongs to $S(\mathbf{x})$. The characteristics of each particular noise function come from the choice of several factors, namely:

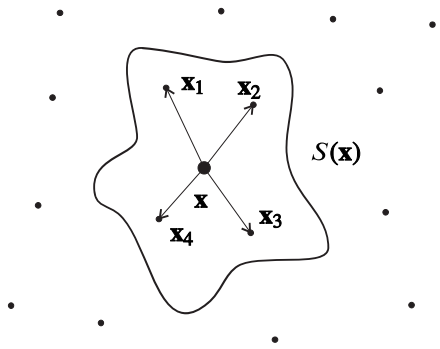


Figure 4.1: The value of the noise function $f_N(\mathbf{x})$ only depends on a finite subset $S(\mathbf{x})$ of node points around the point $\mathbf{x}$. The definition of $S(\mathbf{x})$ varies with noise functions.

- The shape of the kernels.
- The number L of kernels used.
- The criterion used to define $S(\mathbf{x})$.
- The distribution of the $\mathbf{x}_i$ in space to form S.

The random fluctuations of procedural noise result from the introduction of stochastic components into some of the previous factors. In some cases of procedural noise functions, the distribution of node points through space follows a desired probability density. Random variables are also often included in the definition of the kernel functions. Internal consistency is achieved by tying the generation of random numbers to the node points. Repeated invocations of the same node points and their associated kernels for different evaluations of a noise function will always produce the same random values thereby producing consistent results.

The set of first partial derivatives of f_N is grouped in the gradient vector ∇f_N. This vector is the sum of the gradient vectors of all the displaced kernels:

$$\nabla f_N(\mathbf{x}) = \sum_{l=1}^{L} \nabla \phi_l(\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_N). \tag{4.2}$$

The gradient vector may not be defined at some points for noise functions that introduce surface creases. Therefore, there is no requirement that all noise functions be C^1 or even C^2 continuous. C^2 continuity is important for those noise functions that are designed to generate smooth surfaces and where any lower degree of continuity gives rise to unwanted effects.

Similar to the gradient vector, the Hessian matrix $\mathcal{H}\{f_N\}$ of a noise function is also a sum of the Hessian matrices of all the noise's kernels:

$$\mathcal{H}\{f_N\}(\mathbf{x}) = \sum_{l=1}^{L} \mathcal{H}\{\phi_l\}(\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_N). \tag{4.3}$$

The Hessian is only well defined for noise functions that are C^2 continuous. Morse theory, therefore, can only be applied to smooth surfaces hypertextured with C^2 continuous noise functions. Under these conditions, the Hessian is also known to be a symmetric matrix.

The existence of closed form formulae for the gradient and Hessian depends on the nature of the kernel functions. If kernels are formed from simple combinations of analytic functions and polynomials, as will be seen to be the case, closed form expressions for the derivatives become readily available. For any given noise function, only the gradient and Hessian of its respective kernels need to be provided. These are then inserted in equations (4.2) and (4.3) to produce the gradient and Hessian of the noise function.

4.2 Statistical and Spectral Measures

Each noise function is analysed in terms of its statistical stationarity and its spectral characteristics. For simplicity, only one-dimensional and two-dimensional versions of the noise functions are considered. The noise functions are discretised into arrays $\{f_N(x_i, s) : i \in \mathbb{Z}\}$ for one dimension and $\{f_N(x_i, y_j, s) : i, j \in \mathbb{Z}\}$ for two dimensions, where s is the parameter that identifies each particular realisation of the stochastic noise field. In practice, s is used to control the random number generators and hashing tables that each of the noise functions utilise. To obtain accurate measures, it is necessary to average statistical results over a large number of realisations of the stochastic field generated by f_N. One hundred realisations of the discrete stochastic field are used to obtain statistical averages.

The wide sense stationarity of the discretised noise functions is investigated by computing their autocovariance tensors [Jain, 1989]. An autocovariance tensor is the discrete equivalent of the autocorrelation function (Section 3.1.3). For a realisation of a two-dimensional discrete noise field, it takes the form:

$$R_{f_N}(x_i, y_j, x'_i, y'_j, s) = E\{\big(f_N(x_i, y_j, s) - \mu(x_i, y_j)\big)\,\big(f_N(x'_i, y'_j, s) - \mu(x'_i, y'_j)\big)\}, \quad (4.4)$$

where $\mu(x_i, y_j)$ is the mean value of the field at (x_i, y_j). An estimate of the ensemble autocovariance tensor is then obtained by averaging over several realisations of the field:

$$R_{f_N}(x_i, y_j, x'_i, y'_j) = \frac{1}{N}\sum_{l=1}^{N} R_{f_N}(x_i, y_j, x'_i, y'_j, s_l). \quad (4.5)$$

For the discrete field to be stationary in the wide sense it must have a constant mean value and an autocovariance tensor that depends only on the distance between the samples:

$$R_{f_N}(x_i, y_j, x'_i, y'_j) = R_{f_N}(x_i - x'_i, y_j - y'_j). \quad (4.6)$$

The left hand side of (4.6) is a tensor of rank four for two-dimensional noise fields. For simplicity, only one-dimensional fields are considered when studying stationarity. This leads to a rank two tensor for the general autocovariance, which is equivalent to a matrix. Because none of the noise functions studied has an explicit dependence on dimensionality, the statistical results apply for any number of dimensions. The stationarity condition $R_{f_N}(x_i, x'_i) = R_{f_N}(x_i - x'_i)$ for autocovariance matrices means that the matrix $R_{f_N}(x_i, x'_i)$ is symmetric and Toeplitz. A Toeplitz matrix is a symmetric matrix where the elements along each diagonal are equal. Some small scale fluctuations in the matrix elements are expected due to the fact that $R_{f_N}(x_i, x'_i)$ is estimated by averaging a finite number of noise realisations. The matrix elements close to the

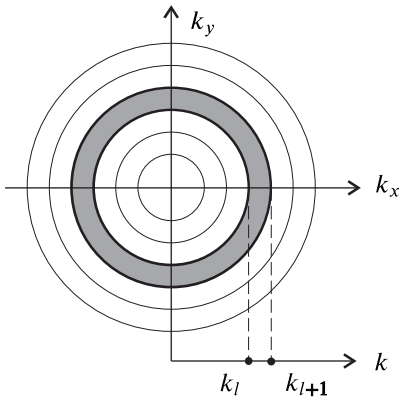


Figure 4.2: The radial power and anisotropy curves are obtained by integrating the power spectrum inside concentric rings in frequency space.

main diagonal should be strongly correlated since they correspond to noise samples that are a small distance apart. The elements close to the main diagonal corners of the matrix should have an almost zero correlation because they correspond to samples that are farther away.

Computing the spectral characteristics of noise functions requires the computation of the discrete two-dimensional Fourier transforms $\hat{f}_N(k_x, k_y, s)$ for each noise function realisation that is parameterised by the s coordinate. The PSDF of f_N (Section 3.1.3) is then estimated with:

$$G_{f_N}(k_x, k_y) = \frac{1}{N} \sum_{l=1}^{N} |\hat{f}_N(k_x, k_y, s_l)|^2. \tag{4.7}$$

Spectral power is measured in decibels: $G_{f_N}(dB) = 10 \log G_{f_N}(k_x, k_y)$. The positive values of $G_{f_N}(dB)$ correspond to amplification while the negative values correspond to attenuation relative to a reference white noise stochastic field with unit power density. The PSDF can be further decomposed into a radial power curve and an anisotropy curve. The radial power curve expresses the power of the noise function after all anisotropic effects have been removed and depends only on the magnitude $k = (k_x^2 + k_y^2)^{1/2}$ of the two-dimensional wave vector. The anisotropy curve expresses the amount of anisotropy that is present for each radial wavenumber k. These two curves have been proposed in computer graphics to study the spectral characteristics of stochastic sample distributions [Ulichney, 1988; McCool and Fiume, 1992]. They can also be applied to the study of continuous stochastic fields. Obtaining the radial power and anisotropy curves requires averaging spectral power values inside concentric rings centred around the origin of the frequency space. Figure 4.2 shows rings with uniformly spaced radii $k_{l+1} = k_l + \Delta k$. The expressions for the two curves are:

$$P_{f_N}(k_l) = \frac{1}{A_l} \int_0^{2\pi} \int_{k_l}^{k_{l+1}} G_{f_N}(k \cos\theta, k \sin\theta)\, k\, dk\, d\theta, \tag{4.8a}$$

$$A_{f_N}(k_l) = \left\{ \frac{1}{A_l} \int_0^{2\pi} \int_{k_l}^{k_{l+1}} \left(G_{f_N}(k \cos\theta, k \sin\theta) - P_{f_N}(k_l) \right)^2 k\, dk\, d\theta \right\} / P_{f_N}^2(k_l). \tag{4.8b}$$

where A_l is the area of the ring with inner radius k_l and outer radius k_{l+1}. The curves are discrete in nature since they only take values for the wavenumbers k_l. With a sufficiently fine

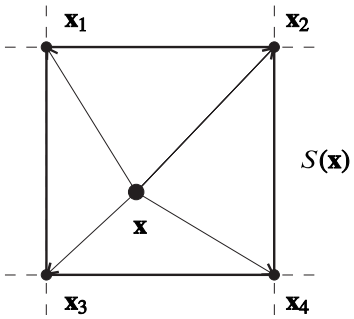


Figure 4.3: The node points of the set $S(\mathbf{x})$ for gradient noise in two dimensions.

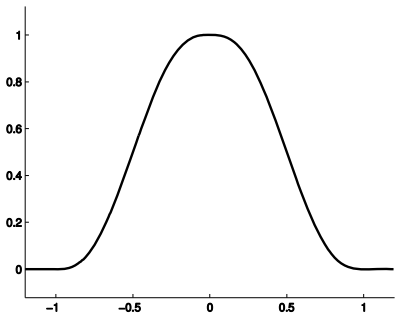


Figure 4.4: The profile of the interpolating function h for gradient noise.

discretisation of the radial wavenumber into k_l samples, it is possible to recover radial power and anisotropy curves that are almost continuous. This also requires that a dense sampling of the PSDF be available so that a sufficient number of power samples can fall within each ring.

4.3 Gradient Noise

Gradient noise was the first procedural noise function to be proposed in the literature [Perlin, 1985]. The description given here corresponds to the newer and improved version of gradient noise as proposed by Perlin [2002]. In this noise function, the node points $\mathbf{x}_i$ coincide with the vertices of a regular lattice placed at integer coordinate positions: $S = \{(u, v, w) : u, v, w \in \mathbb{Z}\}$. For each location $\mathbf{x}$, the set $S(\mathbf{x})$ is made of the eight node points at the vertices of the lattice cell in which $\mathbf{x}$ resides (Figure 4.3 shows the equivalent setting in two dimensions, for which $S(\mathbf{x})$ consists of the four vertices of the square lattice cell where $\mathbf{x}$ resides). There are eight kernels and each one depends on a single node point from $S(\mathbf{x})$. Each kernel depends on the displacement $\mathbf{d}_i = (x_i, y_i, z_i)$, relative to point $\mathbf{x}_i$ and is written as:

$$\phi(\mathbf{d}_i) \;=\; (\xi_1 x_i + \xi_2 y_i + \xi_3 z_i)\, h(x_i) h(y_i) h(z_i). \tag{4.9}$$

The function h, shown in Figure 4.4, is a quintic polynomial with support in the interval $[-1, +1]$. The coefficients of this polynomial are chosen such that the first and second derivatives at the points ± 1 are zero. This ensures that the resulting noise function will be C^2 continuous. The vector (ξ_1, ξ_2, ξ_3) can take any of twelve possible values, chosen randomly from the set:

$$\begin{gathered}(+1,+1,0),\ (-1,+1,0),\ (+1,-1,0),\ (-1,-1,0),\\ (+1,0,+1),\ (-1,0,+1),\ (+1,0,-1),\ (-1,0,-1),\\ (0,+1,+1),\ (0,-1,+1),\ (0,+1,-1),\ (0,-1,-1).\end{gathered}$$

The average of these twelve vectors is $(0, 0, 0)$, which ensures that the function has zero mean. The random vector for each node point is selected from the above set with a 12-way hashing

function that depends on the integer coordinates (u, v, w) of the point. This also ensures that the function is consistent. The name "gradient noise" comes from the fact that the random vector (ξ_1, ξ_2, ξ_3) specifies directly the gradient of the noise function over the node points, i.e. $\nabla f_N(u, v, w) = (\xi_1, \xi_2, \xi_3)$ for $(u, v, w) \in \mathbb{Z}^3$.

For compactness, when writing out the expression for the gradient and the Hessian of the kernels, the polynomial for coordinate x_i is written as $h_x = h(x_i)$, the first derivative as $h'_x = h'(x_i)$ and the second derivative as $h''_x = h''(x_i)$ and similarly for y_i and z_i. The gradient vector and Hessian matrix of the eight kernels for gradient noise are then given by:

$$\nabla\phi(x_i, y_i, z_i) = h_x h_y h_z \begin{bmatrix}\xi_1\\ \xi_2\\ \xi_3\end{bmatrix} + (\xi_1 x_i + \xi_2 y_i + \xi_3 z_i) \begin{bmatrix}h'_x h_y h_z\\ h_x h'_y h_z\\ h_x h_y h'_z\end{bmatrix}, \tag{4.10a}$$

$$\begin{aligned}\mathcal{H}\{\phi\}(x_i, y_i, z_i) &= \\ &= \begin{bmatrix} 2\xi_1 h'_x h_y h_z & (\xi_1 h_x h'_y + \xi_2 h'_x h_y) h_z & (\xi_1 h_x h'_z + \xi_3 h'_x h_z) h_y \\ (\xi_2 h'_x h_y + \xi_1 h_x h'_y) h_z & 2\xi_2 h_x h'_y h_z & (\xi_2 h_y h'_z + \xi_3 h'_y h_z) h_x \\ (\xi_3 h'_x h_z + \xi_1 h_x h'_z) h_y & (\xi_3 h'_y h_z + \xi_2 h_y h'_z) h_x & 2\xi_3 h_x h_y h'_z \end{bmatrix} \\ &\quad + (\xi_1 x_i + \xi_2 y_i + \xi_3 z_i) \begin{bmatrix} h''_x h_y h_z & h'_x h'_y h_z & h'_x h_y h'_z \\ h'_x h'_y h_z & h_x h''_y h_z & h_x h'_y h'_z \\ h'_x h_y h'_z & h_x h'_y h'_z & h_x h_y h''_z \end{bmatrix}.\end{aligned} \tag{4.10b}$$

The Hessian in equation (4.10b) is the sum of two symmetric matrices and is itself symmetric.

4.3.1 Properties of Gradient Noise

Figure 4.5 shows samples of gradient noise in both two and three dimensions. The sample in two dimensions has a spatial extent that covers 32 × 32 lattice cells. The sample in three dimensions fills a 32 × 32 × 32 cubic lattice and is used to hypertexture a sphere. Figure 4.6 shows the autocovariance matrix and the PSDF of gradient noise. The fact that gradient noise is not stationary is immediately clear when one considers that the value of the noise function over the node points is always zero, i.e. $f_N(u, v, w) = 0$. This also becomes evident in the autocovariance matrix where dark spots can be seen regularly spaced on either side of the main matrix diagonal. The dark spots cause the main diagonal and the nearby diagonals to have an oscillatory nature and prevent the matrix from being Toeplitz. The PSDF has a strong central ring with a radius $k = 1m^{-1}$ that roughly corresponds to the desired spectral behaviour but also exhibits multiple aliases of this ring distributed throughout the higher wavenumbers. These aliases are caused by both the regular structure of the integer lattice and the small number of possible directions for the gradient vectors over the nodes.

The radial power curve, shown on the left of Figure 4.7, gives the power spectrum of gradient noise after all anisotropy artifacts have been averaged out. The curve has a dominant wavenumber $k = 1m^{-1}$ and an attenuation that increases for wavenumbers that move progressively away from $1m^{-1}$. The anisotropy curve shows little anisotropy for small wavenumbers ($k < 2m^{-1}$), which roughly correspond to the location of the ring at the centre of the PSDF. For higher wavenumbers, the anisotropy becomes increasingly more significant.

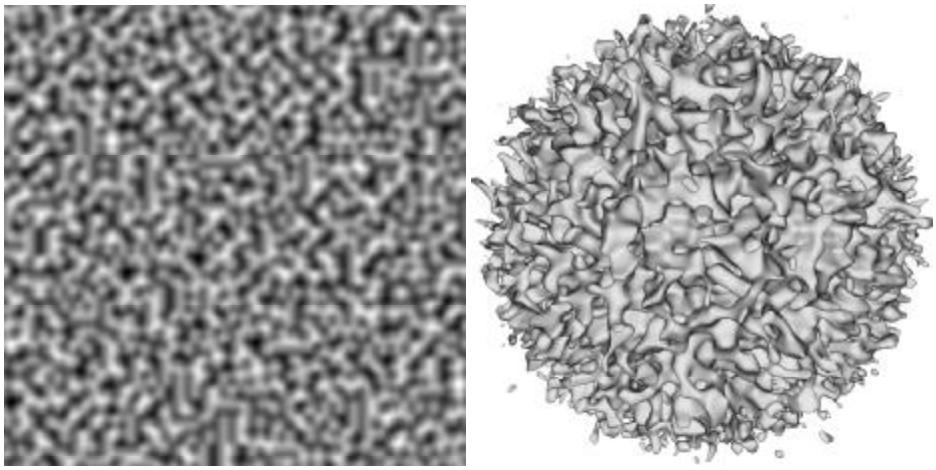


Figure 4.5: A sample of gradient noise in two dimensions (left) and three dimensions (right).

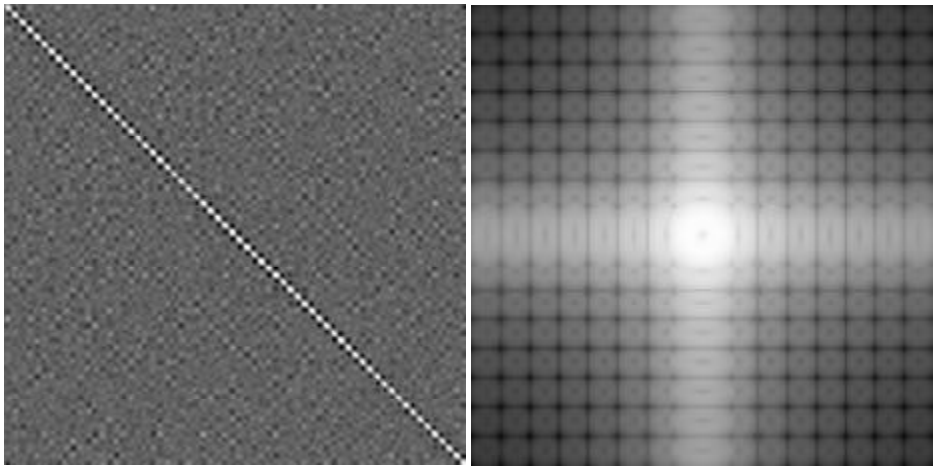


Figure 4.6: The autocovariance matrix (left) and PSDF (right) of gradient noise.

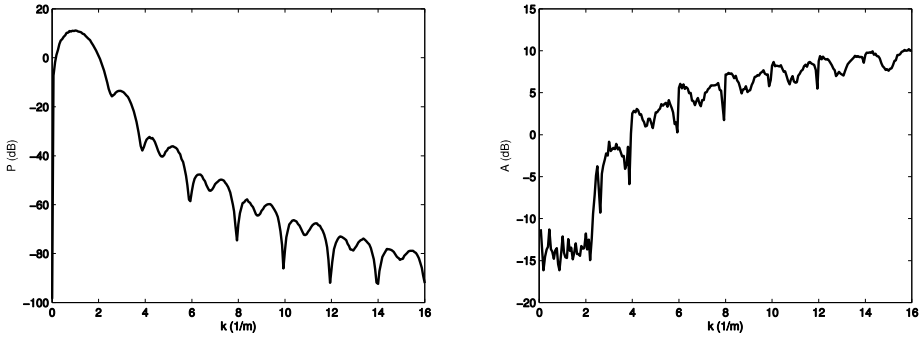


Figure 4.7: The radial power (left) and anisotropy (right) of gradient noise.

4.3.2 Variations on Gradient Noise

Two possible variations on gradient noise are *value noise* and *value-gradient noise* [Peachey, 2003]. Value noise specifies directly the value of the noise function at the node points. The kernel for this noise function is:

$$\phi(\mathbf{d}_i) = \xi_0 h(x_i)h(y_i)h(z_i), \tag{4.11}$$

where ξ_0 is a random variable. Value-gradient noise combines the features of value noise and gradient noise, making it possible to specify both the values and the gradients at the node points:

$$\phi(\mathbf{d}_i) = (\xi_0 + \xi_1 x_i + \xi_2 y_i + \xi_3 z_i)\, h(x_i)h(y_i)h(z_i), \tag{4.12}$$

Value noise has a problem similar to that of gradient noise, which causes the local maxima and minima of the noise function to always be placed over the node points. This regularity can easily become noticeable. Value-gradient noise is the best of the three noise functions. Nevertheless, gradient noise is more widely used in computer graphics than the other two variations because it was the noise function initially proposed by Perlin in his seminal paper [Perlin, 1985]. The `noise` function in the Renderman language, for example, is an implementation of gradient noise [Upstill, 1990].

The attempt to implement gradient noise on GPUs has led to the development of *simplex noise* [Perlin, 2001]. Simplex noise is very similar to gradient noise except that the space is decomposed into a simplicial complex rather than a cubic integer lattice. In two dimensions, the simplicial complex is a mesh of equilateral triangles that partition the plane. In three dimensions, it is a decomposition of space into equilateral tetrahedra. The advantage of using simplices is that, in n dimensions, a simplex has $n+1$ node points placed at its vertices whereas a hypercube has 2^n such points. This makes the implementation of a gradient noise function in three or even higher dimensions much more efficient. Simple comparison rules can be written that identify the simplex where any given point $\mathbf{x}$ resides for a n-dimensional noise function.

4.4 Sparse Convolution Noise

The sparse convolution noise function was proposed by Lewis [1989]. As with the gradient noise function, a regular lattice placed at integer positions is used. Inside each cell in this lattice, K node points are uniformly distributed. This scheme approximates a Poisson sample process. The value of f_N at each location $\mathbf{x}$ depends on the node points of the cell that contains $\mathbf{x}$ plus the node points in the twenty six surrounding cells. The set $S(\mathbf{x})$, therefore, always contains $27K$ node points. There is an equal number of kernels, one for each node point. Figure 4.8 shows a two-dimensional equivalent of $S(\mathbf{x})$ for the case $K = 1$. A kernel ϕ depends only on the distance $d_i = \|\mathbf{d}_i\|$ to its corresponding node point:

$$\phi(\mathbf{d}_i) = \xi h(d_i). \tag{4.13}$$

The scalar ξ is a Gaussian random variable with zero mean and unit variance. The function h needs to be compactly supported on the interval $[0, 1]$. This requirement is necessary to guarantee that only the $27K$ node points in the cell that contains $\mathbf{x}$ and in its immediate neighbours

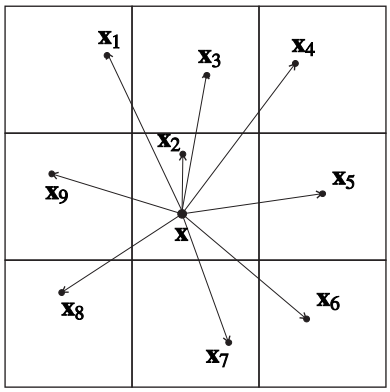


Figure 4.8: The node points of $S(\mathbf{x})$ for sparse convolution noise in 2D with $K = 1$.

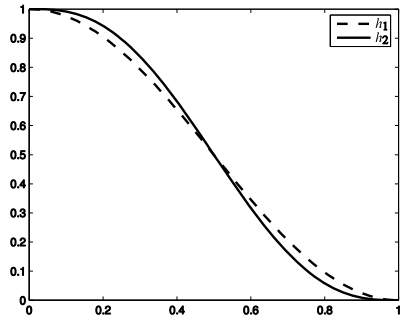


Figure 4.9: The profile of the Lewis (h_1) and quintic (h_2) kernels for sparse convolution noise.

can possibly influence the value of $f_N(\mathbf{x})$. Lewis proposed the function $h_1(d) = (1+\cos \pi d)/2$, restricted to the $[0, 1]$ interval (Figure 4.9). This function, however, has the problem of generating a second derivative at $d = 1$ that is different from zero: $h_1''(1) = \pi^2/2$. As a consequence, an implicit surface hypertextured with a sparse convolution noise that uses h_1 as the kernel will only be C_1 continuous. This leads to the same Mach banding artifacts that were mentioned in Section 2.2.4 for blobby surfaces. To use sparse convolution for hypertexturing smooth surfaces, C^2 continuity needs to be enforced and this, in turn, requires the kernel function to be C^2 continuous and to obey $h(1) = h'(1) = h''(1) = 0$. The latter conditions will not cause any value or derivative jumps at the outer border of the kernel's volume of support. In this thesis, the same quintic kernel that is used for gradient noise is also used for sparse convolution noise with a domain now restricted to the unit interval. It is referred to here as $h_2 : [0, 1] \rightarrow [0, 1]$ to differentiate it from the h_1 kernel previously mentioned and it is also shown in Figure 4.9.

Sparse convolution has the desirable property of allowing very precise control over the spectral behaviour of the noise function. The noise function can be seen as the result of the convolution between the kernel (4.13) and a random distribution of Dirac impulses in space:

$$p(\mathbf{x}) = \sum_i \delta(\mathbf{x} - \mathbf{x}_i). \tag{4.14}$$

The name "sparse convolution noise" is taken from the observation that the noise results from this convolution between a sparsely distributed set of impulses and a kernel. Because the distribution of impulses is Poisson, it has a constant PSDF $G_p(k) = \eta$, where η is related to the density of impulses per unit volume. The convolution of h with p in space corresponds to a filtering operation and the outcome has a PSDF that, as explained in Section 3.1.3, is given by:

$$G_{f_N}(k) = \eta |\hat{h}(k)|^2. \tag{4.15}$$

Sparse convolution noise is guaranteed to be stationary provided that a Poisson sampling process can be correctly simulated. It is also guaranteed to be isotropic (as seen from the fact that (4.15) only depends on the magnitude of the wave vector) because the kernel is radially symmetric. Any radial profile for the PSDF of sparse convolution noise can be selected by

choosing the shape of the kernel such that the squared magnitude of its Fourier transform will have the desired profile. The only restrictions are that the kernel must be C^2 continuous, as previously explained, and with compact support in $[0, 1]$. This defines a subset of all possible kernel shapes that have a Fourier transform in $L^2(\mathbb{R}^3)$.

Sparse convolution noise has zero mean value because the random variable ξ of (4.13) also has zero mean value. The noise function is consistent because the generation of the random node points inside each lattice cell is seeded with a hash value taken from the (u, v, w) integer coordinates of the cell. The number of different node point distributions for the cells depends on the quality of the hash function. If two different cells are attributed with the same random seed, they will have the same distribution of points. The occurrence of cells with the same seed value is inevitable, considering that a hash function can only generate a finite number of seeds. This causes some degradation in the spectral quality of the noise, especially if two cells with identical node point distributions happen to be close to each other.

The gradient vector and Hessian matrix of the kernels for sparse convolution noise depend on both the displacement vector $\mathbf{d}_i$ and its magnitude d_i relative to node point $\mathbf{x}_i$:

$$\nabla\phi(\mathbf{d}_i) = \xi \frac{h'(d_i)}{d_i}\mathbf{d}_i, \tag{4.16a}$$

$$\mathcal{H}\{\phi\}(\mathbf{d}_i) = \xi\left(\frac{h''(d_i)}{d_i^2} - \frac{h'(d_i)}{d_i^3}\right)\left(\mathbf{d}_i \cdot \mathbf{d_i}^T\right) + \xi\frac{h'(d_i)}{d_i}\mathbf{I}. \tag{4.16b}$$

The matrix $\mathbf{I}$ is a 3×3 identity matrix. The Hessian matrix is symmetric because the matrix $\mathbf{d}_i \cdot \mathbf{d}_i^T$ is also symmetric by construction.

4.4.1 Properties of Sparse Convolution Noise

Figure 4.10 shows samples of sparse convolution noise in two dimensions and three dimensions. The evaluation of the noise functions in these samples falls within the same number of lattice cells that was used for gradient noise, specifically a square of 32^2 cells and a cube of 32^3 cells. The hashing function was implemented so that no two cells within the square or cube have the same random seed. This removes any spectral artifacts from the results obtained in this section. The evaluation of the noise function for points outside of either the square or cube, however, may still lead to repeated seed values.

The estimated autocovariance of the noise function, shown on the left of Figure 4.11, exhibits strong correlation along the main diagonal, as expected. Apart from that there are low amplitude background fluctuations that result from the finite averaging procedure. The absence of coherent structures in these fluctuations indicates that the noise function is indeed stationary. The PSDF, on the right of Figure 4.11, is clearly isotropic with a spectrum power decaying away from the origin.

The radial spectrum is shown on the left of Figure 4.12 and corresponds to a lowpass filter with attenuation starting for wavenumbers above $2m^{-1}$. A similar situation would have occurred if the Lewis kernel h_1 had been used. Since this latter kernel has a simple expression, it is possible to obtain its Fourier transform by hand and verify that it also gives rise to a lowpass filter. The anisotropy curve on the right of Figure 4.12 oscillates between -20dB and 0dB. The anisotropy

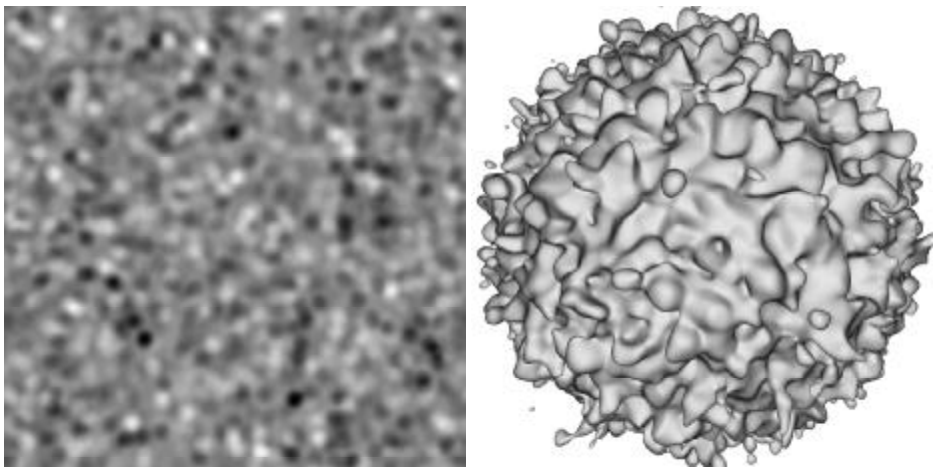


Figure 4.10: A sample of sparse convolution noise in two dimensions (left) and three dimensions (right).

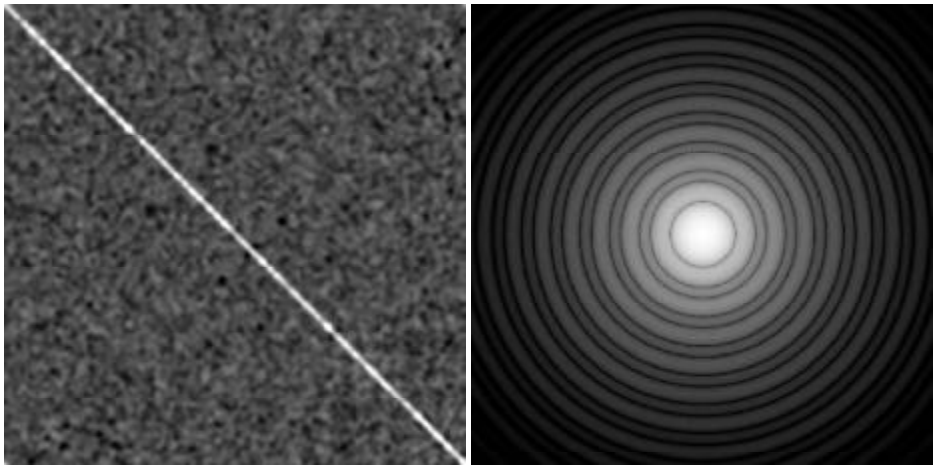


Figure 4.11: The autocovariance matrix (left) and PSDF (right) of sparse convolution noise.

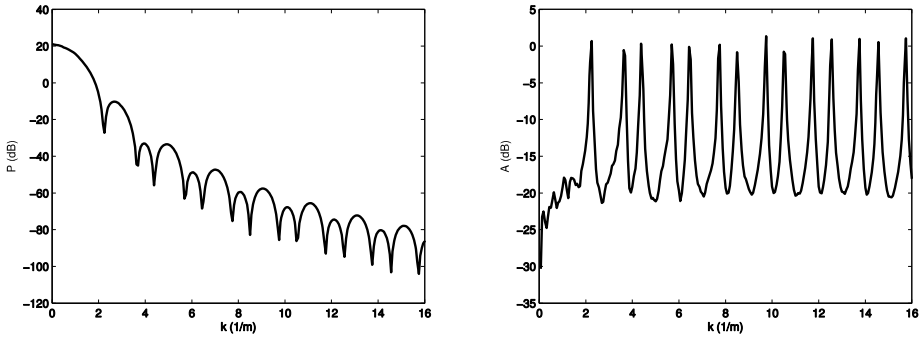


Figure 4.12: The radial power (left) and anisotropy (right) of sparse convolution noise.

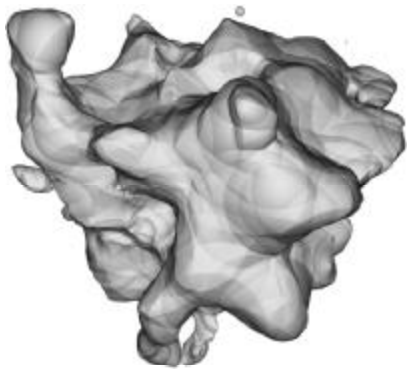


Figure 4.13: A hypertexture with a sparse convolution noise that uses a linear kernel.

peaks at 0dB are placed over spatial frequencies where the PSDF is mostly devoid of power. These frequencies correspond to the dark rings of the PSDF on the right of Figure 4.11 and the local minima of the radial power curve on the left of Figure 4.12. Because the radial power over these discrete frequencies is almost zero, the slightest deviation from an isotropic PSDF leads to a relatively large increase in the anisotropy estimate.

4.4.2 Variations on Sparse Convolution Noise

A generalisation of sparse convolution noise is called *spot noise* [van Wijk, 1991]. Spot noise removes any restrictions on the shape or the size of the kernel used for sparse convolution. The expression for spot noise is:

$$f_N(\mathbf{x}) = \sum_i \xi_i h(\mathbf{x} - \mathbf{x}_i, \mathbf{x}_i), \tag{4.17}$$

where the node points $\mathbf{x}_i$ are, as before, samples of a Poisson distribution process and the ξ_i are zero mean Gaussian random variables with possibly different variances from node point to node point. Different node points can have different kernels, which is expressed in (4.17) by the dependence of h on the position $\mathbf{x}_i$ and not only on the displacement vector $\mathbf{x} - \mathbf{x}_i$ towards the evaluation point of the function. The contribution of each individual kernel centred around each $\mathbf{x}_i$ node point is called a *spot*. The extra flexibility of spot noise allows it to generate a large variety of stochastic textures, which can be intentionally non-stationary and anisotropic. The main drawback of spot noise is that it cannot easily be used as a procedural noise function – it is mainly used as a texture synthesis method. The spatial variation that spot noise introduces into the size and shape of the spots makes it cumbersome to evaluate $f_N(\mathbf{x})$ for any arbitrary point $\mathbf{x}$. Given some point $\mathbf{x}$ and depending on the local size of the spots, the function may need to look at an arbitrarily large neighbourhood of lattice cells. Spot noise is usually calculated by successively dropping and accumulating the contribution of spots onto a texture buffer. Node points over the buffer are randomly selected with a uniform probability distribution and the process is terminated when a sufficient coverage of the texture with spots is achieved.

It is possible to introduce variations into sparse convolution noise by changing the shape of the kernel as long as the basic requirement that kernels be supported in the $[0, 1]$ interval is

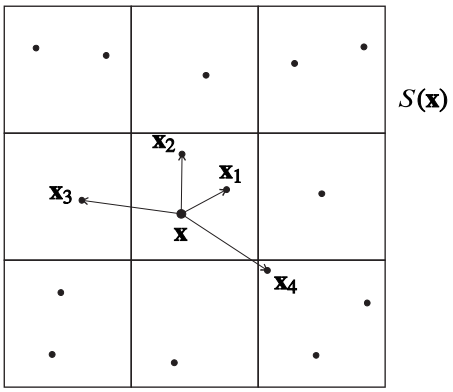


Figure 4.14: The node points of the set $S(\mathbf{x})$ for cellular texture noise.

maintained. The kernel need not even be C^2 continuous provided the designer of the noise function is aware of the implications that this may have for surface continuity. Using a kernel that is only C^0 continuous, for example, will produce a hypertextured implicit surface with creases and cusps. Figure 4.13 exemplifies this with a kernel given by $h(d) = 1 - d$, which decreases linearly from $h(0) = 1$ to $h(1) = 0$. The relative size of the hypertexture over the sphere has been reduced, when compared to the hypertextures shown in Figures 4.5 and 4.10, to make the creases more visible.

4.5 Cellular Texture Noise

Cellular texture functions, proposed by Worley [1996], rely on a Voronoi decomposition of space based on the location of the $\mathbf{x}_i$ node points. As with sparse convolution functions, an approximation to a Poisson sample distribution of node points is generated inside an integer lattice, although the technique used to achieve this effect is slightly different from the one employed by Lewis [1989]. The kernels for cellular texture noise functions consist of the ordered set of increasing distances between any location $\mathbf{x}$ and the node points. For some location $\mathbf{x}$, let $D(\mathbf{x}) = \{\|\mathbf{d}_i\| : i = 0, 1, 2 \ldots\}$ be the set of distances from $\mathbf{x}$ to all node points in S. Let also $p : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ be a permutation of the indices in $D(\mathbf{x})$ so that the new set $D(\mathbf{x}) = \{\|\mathbf{d}_l\| : l = p(i), l = 0, 1, 2 \ldots\}$ is ordered by increasing distances. The l-th kernel function is then taken as the l-th element in the ordered set $D(\mathbf{x})$:

$$\phi_l(\mathbf{d}_l) = a_l \|\mathbf{d}_l\|, \tag{4.18}$$

where a_l is some chosen scaling constant. Worley points out that only the kernel functions ϕ_1 to ϕ_4 are useful for texture synthesis. The ϕ_i with $i > 4$ resemble ϕ_4 and do not add significant new details. Kernel functions can also be turned off by setting $a_l = 0$.

Worley constructs a table of 256 possible cells containing Poisson distributed node points. A hashing function is then used to index into this table from any lattice cell in space. The number of node points per cell is given as the outcome of a Poisson distributed discrete random variable. This constitutes a more accurate modelling of a Poisson sample process than the one done by Lewis for sparse convolution noise. In a Poisson sampling process, the number of samples

falling inside some subset of $\mathbb{R}^3$ should always follow a Poisson distribution (hence the name of the process). Lewis, by comparison, always attributes a constant number K of node points per cell. Worley, however, introduces additional samples into some of the less populated cells in the table to guarantee that the fourth smallest distance to any point $\mathbf{x}$ (corresponding to the ϕ_4 kernel function) can always be found in the cell of $\mathbf{x}$ or in any of its immediate neighbours. In this way, the set $S(\mathbf{x})$ for cellular texture functions is the same $3 \times 3 \times 3$ cube of cells used for sparse convolution. Figure 4.14 shows this set. Even though there can be many node points within $S(\mathbf{x})$, only the four nearest ones relative to some evaluation point $\mathbf{x}$ are used for the computation of the noise function.

Because all four kernels of cellular texture noise are distances to node points, the gradient of those kernels is simply the gradient of their respective distances:

$$\nabla\phi_l(\mathbf{d}_l) \;=\; a_l \nabla\|\mathbf{d}_l\| \;=\; a_l \frac{\mathbf{d}_l}{\|\mathbf{d}_l\|} \quad \text{for} \quad l = 1, 2, 3, 4. \tag{4.19}$$

Cellular texture noise is inherently C^0 continuous and has derivative discontinuities over points that are equidistant to two or more node points. This turns cellular texture noise into a useful noise function for generating ridged terrains. On the other hand, the lack of smoothness of cellular texture noise makes it unnecessary to derive the expression for the Hessian matrices of the kernels since Morse theory cannot be applied to C^0 continuous functions.

4.5.1 Properties of Cellular Texture Noise

Figure 4.15 shows two-dimensional and three-dimensional examples of cellular texture noise generated with only the first kernel function, i.e. with $a_1 = 1$ and $a_2 = a_3 = a_4 = 0$. For this setting, the two-dimensional sample on the left of Figure 4.15 corresponds to a Voronoi partitioning of the plane with the node points corresponding to the centroids of each Voronoi cell. A special table containing 1024 pre-computed two-dimensional cells was used, instead of the usual table with 256 entries, so that each of the 32×32 lattice cells forming the sample are unique. This eliminates any artifacts caused by repetition of cells when computing the spectral characteristics of the noise function. The name for this noise function is due to the fact that the spatial Voronoi partitioning creates an impression of a cellular tissue. For this reason, cellular texture noise is quite successful at synthesising textures of an organic nature. Figure 4.18 shows another example in three-dimensions where both the first and second kernel functions are used according to the weights $a_1 = a_2 = 1.0$ and $a_3 = a_4 = 0$. The overall intensity of the hypertexture over the sphere has been decreased, relative to the hypertexture in Figure 4.15. This low intensity of the hypertexture makes the features of the cellular texture noise easier to distinguish but also causes the surface overhangs to disappear. A surface similar to the one in Figure 4.18 could have been obtained by performing a simple spherical displacement map based on cellular texture noise. Many more combinations of the weights a_1 to a_4 can be found in Worley's original paper.

The statistics and spectral estimates shown in Figures 4.16 and 4.17 correspond to a cellular texture noise made with the single kernel ϕ_1 and correspond to the samples shown in Figure 4.15. The spatial autocovariance matrix for cellular texture noise, shown on the left of Figure 4.16, indicates that this noise function is, for all purposes, statistically stationary. The PSDF, on the

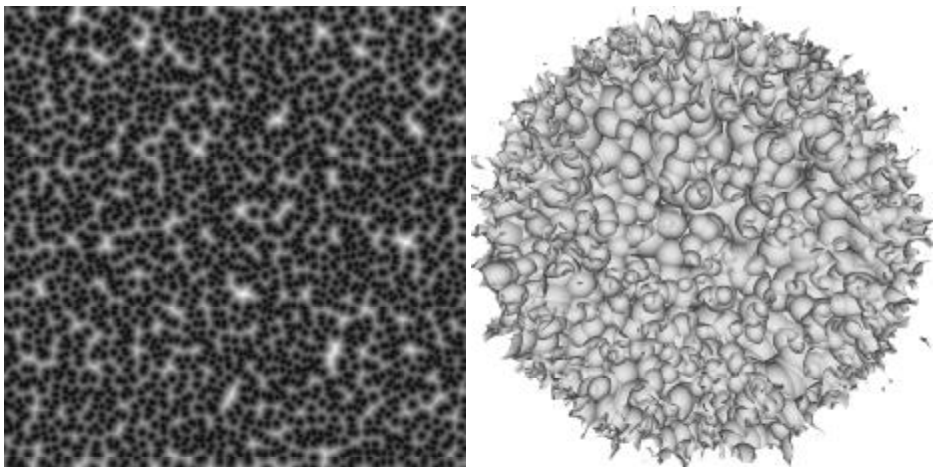


Figure 4.15: A sample of cellular texture noise in two dimensions (left) and three dimensions (right).

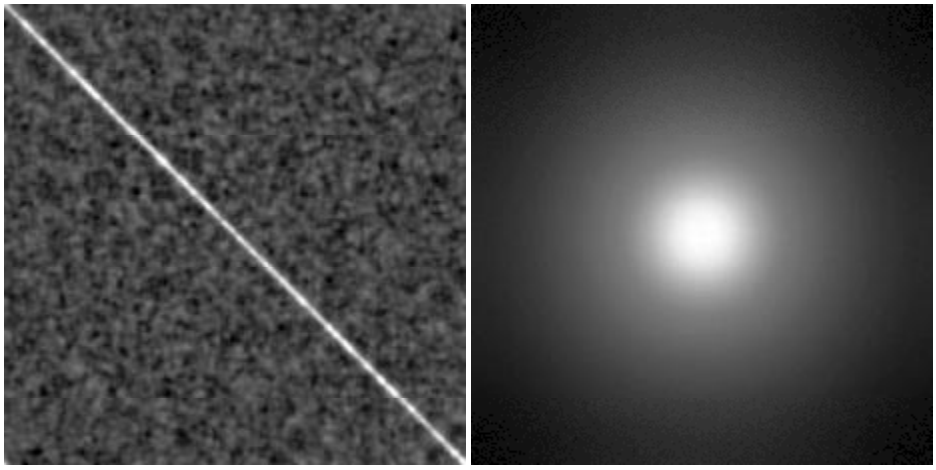


Figure 4.16: The autocovariance matrix (left) and PSDF (right) of cellular texture noise.

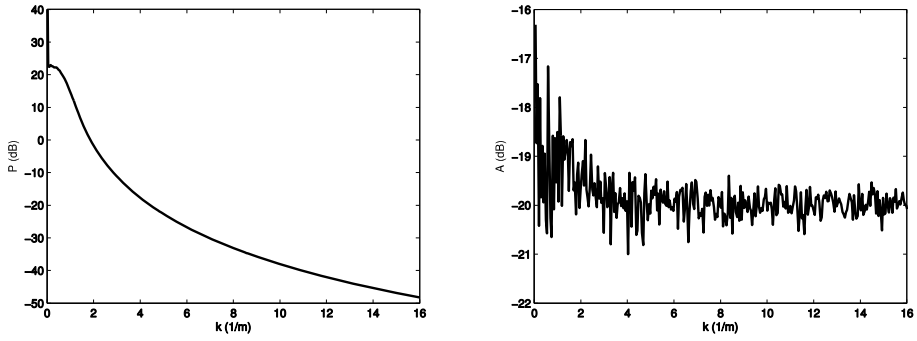


Figure 4.17: The radial power (left) and anisotropy (right) of cellular texture noise.

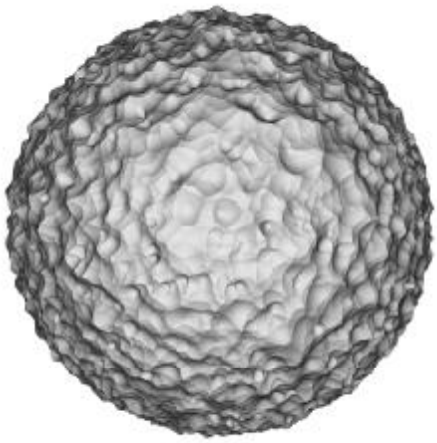


Figure 4.18: A hypertexture with a cellular texture noise that uses the ϕ_1 and ϕ_2 kernels with equal weights.

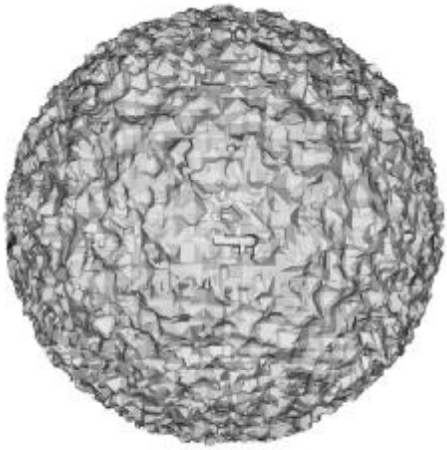


Figure 4.19: A hypertexture with a cellular texture noise that uses the Manhattan distance.

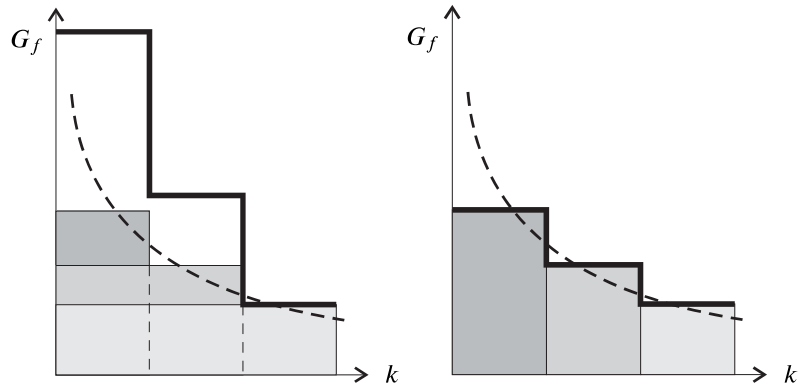
right of 4.16, also shows that it is isotropic. Stationarity and isotropy are a consequence of the Poisson sample distribution of node points – a property that this noise function shares with sparse convolution noise. The radial PSDF is shown on the left of Figure 4.17 and corresponds to a radial slice through the full PSDF of Figure 4.16. It contains a DC spike and has a lowpass behaviour with attenuation beginning for spatial frequencies over $2m^{-1}$. The spike of the PSDF over the null spatial frequency indicates that the mean value of cellular texture noise made from the ϕ_1 kernel is higher than zero. This is also true for any other combination of weights used. The mean value of each kernel ϕ_1 to ϕ_4 is the mean first to fourth smallest distance towards the Poisson distributed node points. These mean distance values depend on the density of node points over the space. It is possible to remove the bias from cellular texture noise by working with the kernels $\phi_l(\|\mathbf{d}_l\|) - \mu_l$, where μ_l is the mean value of the l-th kernel. Finally, the curve on the right of Figure 4.17 shows that cellular texture noise has negligible anisotropy centred around the value of −20dB.

4.5.2 Variations on Cellular Texture Noise

Beyond changing the weights of the four kernel functions, variations to cellular texture noise can be made by changing the definition of the distance metric. In the standard definition of cellular texture noise, the distance from a point $\mathbf{x}$ to another point $\mathbf{y}$ is the usual Euclidean distance. This can be replaced with the p-norm distance in $\mathbb{R}^3$:

$$\|\mathbf{x}-\mathbf{y}\|_p = \begin{cases} \left(\sum_{i=1}^{3} |x_i - y_i|^p\right)^{1/p} & \text{for } 1 \leqslant p < \infty, \\ \max\left(|x_1 - y_1|, |x_2 - y_2|, |x_3 - y_3|\right) & \text{for } p = \infty. \end{cases} \tag{4.20}$$

Euclidean distance corresponds to the 2-norm. The 1-norm is also known as the *Manhattan distance* or the *taxicab norm* because it measures distance in analogy to the shortest distance



(a) Lowpass noise function (b) Bandpass noise function

Figure 4.20: A PSDF can be built approximately by assembling frequency shifted copies of a single procedural noise function. The thick broken line shows the desired PSDF. The thick continuous line shows the resulting PSDF.

travelled by a car (without reversing direction) through a city laid out in square blocks. Figure 4.19 shows the same hypertexture of Figure 4.18 with the 2-norm replaced by the 1-norm when computing the distances $\mathbf{d}_i$ between any point $\mathbf{x}$ and the Poisson distributed node points $\mathbf{x}_i$. The resulting hypertexture has a more angular nature that the one in Figure 4.18. Other possible variations make the cellular texture function depend on the square of the p-norm distance to the node points or, more generally, on some function of distance $\psi(\|\mathbf{d}_i\|_p)$, where $\psi : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ is a monotonically increasing function.

4.6 Wavelet Noise

The noise functions considered so far have a spectral behaviour that corresponds to that of a lowpass filter. The ideal situation, however, would be to have a noise function with a bandpass behaviour. This is because a stochastic field with some desired power spectrum can be built by accumulating frequency shifted copies of a procedural noise function. One example is the rescale-and-add method of Saupe [1989] (Section 3.2.4) that is used to build a fractal PSDF. Figure 4.20 illustrates the problem of using lowpass noise functions, when compared with bandpass noise functions, for building general spectra. The use of lowpass filters introduces a significant crosstalk between the different noise copies for the low spatial frequencies, which causes the resulting PSDF to be different from the intended PSDF. A bandpass noise does not have this behaviour because there is no overlap between the frequency bands. Of course, Figure 4.20 shows idealised lowpass and bandpass noise functions. In a real situation there will be some inevitable crosstalk between adjacent frequency bands but a well designed bandpass noise function should be able to minimise this.

Wavelet noise was specifically designed to have a good bandpass spectral behaviour [Cook and DeRose, 2005]. It is based on the concept of *multiresolution analysis* (MRA) [Chui, 1992]. Consider the space S_0 of real functions $f \in L^2(\mathbb{R})$ that are obtained by adding shifted copies of a single function φ:

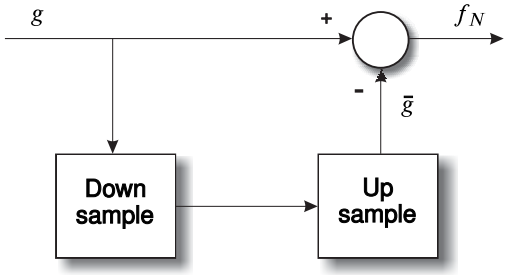


Figure 4.21: The generation of a bandlimited wavelet noise is done by taking a noise field g in the S_1 space and subtracting its image $\bar{g}$ in the lower S_0 space.

$$f(x) = \sum_{i \in \mathbb{Z}} a_i \varphi(x - i). \tag{4.21}$$

In the space S_0, f is expressed as the sequence of coefficients a_i. If φ is the *scaling function* of the MRA, it is equally possible to express f in a more detailed space S_1 made from shifted copies of $\varphi(2x)$. The function $\varphi(2x)$ corresponds to $\varphi(x)$ after having been shrunk by half along its domain. In the space S_1, f is expressed with a different sequence b_i of coefficients:

$$f(x) = \sum_{i \in \mathbb{Z}} b_i \varphi(2x - i). \tag{4.22}$$

The space S_0 is nested in S_1 since every function $f \in S_0$ is also a function of S_1. A discrete upsampling filter can be designed that transforms the coefficients a_i into the coefficients b_i in the higher space. This upsampling is represented as $f \uparrow$. The converse is not true, however. Some general function $g \in S_1$ cannot exist in S_0 without losing some of its detail. The downsampling filter $g \downarrow$ is a lowpass filter that smooths g. The difference $g - g \downarrow$ is a function that isolates all the detail of g that had to be removed in order to express it in the lower space S_0. This detail function is bandlimited because it contains those frequencies that can exist in the space S_1 but cannot exist in the space S_0. All the detail functions belong to a *wavelet space* W_0 such that $S_1 = S_0 + W_0$, where the plus sign is used to indicate the Cartesian sum of sets.

Cook and DeRose construct wavelet noise by initially specifying a Gaussian noise field $g \in S_1$. The coefficients b_i for g are uncorrelated Gaussian distributed random variables. The noise field is downsampled to S_0 and subsequently upsampled to S_1 again, resulting in a new stochastic field $\bar{g} = g \downarrow\uparrow$ with a set of coefficients $\bar{b}_i$. In the process, all the details of g that are in W_0 will have been lost for $\bar{g}$. The wavelet noise function in S_1 is given by $g - \bar{g}$ and is able to recover the bandpass details by comparing g with $\bar{g}$. The coefficients of the wavelet noise function correspond to the difference between the original coefficients b_i of g and the coefficients $\bar{b}_i$ of $\bar{g} = g \downarrow\uparrow$. Figure 4.21 shows the synthesis process. A quadratic B-spline basis function is used for the scaling function φ of the MRA. The wavelet noise function is then given by:

$$f_N(2x) = \sum_{i \in \mathbb{Z}} \left(b_i - \bar{b}_i\right) \varphi(2x - i). \tag{4.23}$$

The synthesis of wavelet noise can be extended to two or three dimensions by considering tensor products of B-spline basis functions. The expression (4.23) fits easily in the unified

representation for noise functions given in Section 4.1. The node points are placed over the vertices of a lattice. The kernels are tensors of B-spline bases and the weight of each kernel is suitably calculated, as previously explained, to achieve a bandpass behaviour. The authors compute several three-dimensional tiles of wavelet noise coefficients. For any point $\mathbf{x} \in \mathbb{R}^3$, a tile is selected with a hashing function and the noise value is obtained by evaluating (4.23) with the coefficients of the tile. The tiles are made to overlap slightly so that no artifacts result when evaluating $f_N(\mathbf{x})$ for a point $\mathbf{x}$ that is close to the border of its assigned tile.

4.7 Summary

Four procedural noise functions have been presented in this chapter. All of them can be expressed with a unified representation that sums kernels placed over node points. The first three of these noise functions have been implemented as part of the research performed for this thesis. Gradient noise is the most efficient of the noise functions considered but is also the one that produces noise with the least quality. For gradient noise in three dimensions, only eight kernels need to be evaluated. Gradient noise, however, is neither stationary nor isotropic.

Sparse convolution noise can be regarded as a replacement to gradient noise that has optimal spectral properties. It offers precise control over the shape of the PSDF by choosing the shape of the kernel used, although some restrictions need to be enforced. Specifically, the kernel cannot extend beyond a unit distance away from each node point and C^2 continuity may be necessary. The disadvantage of sparse convolution noise is its computational expense. Even if only one random node point is placed inside each lattice cell, the sparse convolution algorithm needs to query $3 \times 3 \times 3 = 27$ kernels in order to evaluate the noise function correctly. For K node points per lattice cell, the complexity of the function increases to $27K$ kernel queries.

Cellular texture noise has unique characteristics stemming from its use of distance functions to node points. The computational complexity of cellular texture noise is in-between that of gradient noise and sparse convolution noise. Like sparse convolution noise, it also uses a Poisson sample distribution of node points in space. Cellular texture noise, however, does not need to look exhaustively at all the node points in the 27 lattice cells that surround some evaluation point $\mathbf{x}$. Consider the case of evaluating the kernel ϕ_1 for some point $\mathbf{x}$. If the distance to the nearest node point in the lattice cell that contains $\mathbf{x}$ is d then any neighbouring cell that is at a distance to $\mathbf{x}$ larger than d need not be considered. This is because the consideration of the node points in the neighbouring cell cannot possibly reduce the nearest distance to any value smaller than d.

Wavelet noise is a recent contribution to the field of procedural texture modelling. It offers good control over the maximum and minimum frequencies of the PSDF, which, in turn, minimises the interferences when adding together several copies of a wavelet noise function. This makes it possible to build with greater accuracy stochastic fields that have some desired PSDF. It also gives effective control over the anti-aliasing of surfaces textured with wavelet noise. Because the minimum and maximum frequencies of wavelet noise are known to a good approximation, it is possible to clamp those copies of wavelet noise whose frequencies are above the Nyquist limit thereby largely eliminating aliasing artifacts. Despite the good qualities of wavelet noise, it has not been used in this thesis due to its greater implementational complexity. The other

three noise functions provide the necessary support for the rest of the research. Wavelet noise can be easily added to the general framework in the future.

CHAPTER 5

Ray Casting

RAY casting is a rendering and hidden surface removal algorithm that works by casting rays from the viewpoint and through the image pixels into the scene [Roth, 1982]. For each ray, an intersection point is found with the nearest surface. The evaluation of a shading model at the intersection point then gives the colour intensity for the pixel through which the ray was cast. Ray casting is a conceptually simple algorithm when compared with other hidden surface removal algorithms that are based on scanline rasterisation [Sutherland et al., 1974; Lane et al., 1980]. Unlike those scanline algorithms, ray casting does not impose any particular order for rendering the image pixels, which makes it easily parallelisable by scheduling pixels to different processors. Another advantage of ray casting is that it is not restricted to rendering polygonal meshes or parametric patches but can be used to render any surface for which a ray-surface intersection algorithm can be written. The main disadvantage of ray casting is that finding the intersection point between a ray and the nearest surface can often be an expensive procedure. Ray casting is a specialisation of the more general *ray tracing algorithm* [Glassner, 1989]. Ray tracing includes the possibility of spawning new rays at a surface intersection point through reflection and refraction.

A ray can be parameterised as a semi-infinite line $\mathbf{r}(t)$, with $t \geqslant 0$, starting from an origin $\mathbf{o}$ and going along a direction $\mathbf{d}$:

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}. \tag{5.1}$$

For ray casting, the origin $\mathbf{o}$ is the viewpoint position and the direction $\mathbf{d}$ is calculated to make the ray pass through the centre of some pixel on the image plane. The direction vector is usually normalised so that the parameter t expresses the Euclidean distance along the ray. Figure 5.1 shows the geometry of ray casting. When ray casting an implicit surface, one needs to find a point $\mathbf{r}(t)$ along the ray for which the surface's implicit function $f(f_0(\mathbf{x}), \mathbf{x})$ returns a value of zero. This amounts to finding a root t of the equation:

$$f(f_0(\mathbf{r}(t)), \mathbf{r}(t)) = 0. \tag{5.2}$$

For simplicity of notation, the auxiliary function $g = f \circ \mathbf{r}$ is introduced, being different for every ray since it results from the composition of f with the ray parametrisation $\mathbf{r}$. Based on g, the equation (5.2) can instead be written as:

$$g(t) = 0. \tag{5.3}$$

Equation (5.3) makes it clear that finding the intersection point between a ray and an implicit surface amounts to finding the root of a one-dimensional arbitrary function, subject to the

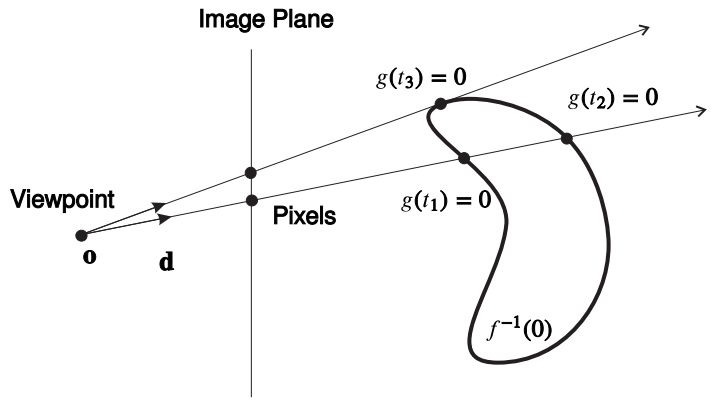


Figure 5.1: A ray intersects an implicit surface $f^{-1}(0)$ at two points characterised by the roots t_1 and t_2 of the equation $g(t) = 0$. Every root corresponding to an entry point, such as t_1, is followed by a root corresponding to an exit point, such as t_2. An exception occurs for multiple roots such as t_3 that result from tangent rays.

continuity constraints defined in Section 2.1. If no such root exists, the ray does not intersect with the surface. If more than one root exists, one is usually interested in the smallest root since it corresponds to the intersection point along the ray that is directly visible from the viewpoint. The case of a multiple root can also occur for rays that are tangent to the surface as shown in Figure 5.1. Multiple roots, however, are not of much concern because the probability of a ray shot through some arbitrary pixel being exactly tangent to the surface is very small. When dealing with topology correction for implicit surfaces in Chapter 8, it will be seen that locating the nearest intersection point along a ray is not enough and that sometimes further intersection points also need to be computed by increasing order of distance along the ray. Traditional numerical methods for one-dimensional root finding are not adequate by themselves because, in the case where there is more than one root, they do not have control over which root is returned [Corliss, 1977][1]. This has motivated research into numerical root finding methods that are appropriate for ray-surface intersection problems.

Section 5.1 begins by presenting previous ray-surface intersection methods. Attention is given exclusively to methods that are robust. These methods can always find the correct intersection point and are limited only by the floating point precision of the machine. A survey of such methods is given by Hart [1993]. Intersection methods that can work with arbitrary implicit surfaces either assume the surface is *Lipschitz continuous* or rely on interval range estimates. A specific Lipschitz based ray intersection algorithm, proposed by Hart [1996], is described in Section 5.2. This particular intersection method, called *sphere tracing*, is given some prominence in this chapter because it can be significantly more efficient than other general intersection algorithms under certain assumptions. Section 5.3 then presents the theory of range estimation techniques, which leads to Section 5.4 where a novel ray casting method based on range estimation is presented [Gamito and Maddock, 2007d]. This novel method is well suited for implicit surfaces that are hypertextured with the procedural noise functions presented in the previous chapter. The method is shown to be superior to other algorithms based on interval range estim-

[1]One exception occurs when $g(t)$ is a polynomial, in which case traditional techniques for isolating single roots of polynomials can be successfully applied.

ates. Section 5.5 concludes the chapter by presenting the advantages and disadvantages of the proposed ray casting method when compared with the sphere tracing method of Section 5.2.

5.1 Previous Work

Robust implicit surface intersection methods were initially developed for surfaces with a simple and well known shape. If the implicit surface is algebraic (Section 2.2.2) the function f is a polynomial and the intersection points can be obtained with polynomial root finders [Hanrahan, 1983]. Algorithms for surfaces generated by sweeping a sphere along a curve (Section 2.2.5) and surfaces that are subject to non-linear deformations (Section 2.3.3) have also been considered [van Wijk, 1985; Barr, 1986]. Blobby surfaces (Section 2.2.4) are popular because of their ability to model objects with complex topology. Many authors who have worked with this type of surface have also developed ray intersection algorithms for them. Such authors include Blinn [1982a] with his blobby model, Nishimura et al. [1985] with metaballs and Wyvill and Trotman [1990] with soft objects. Sherstyuk [1999a] has developed a general intersection method for surfaces generated from sums of compactly supported radial basis functions. His method approximates any basis function with piecewise Hermite polynomials, the roots of which can then be found with analytical formulas.

Generic intersection algorithms that rely on the assumption of Lipschitz continuity require that a *Lipschitz bound* be supplied for the surface that is to be rendered. Lipchitz bounds impose a limit on the maximum rate of change that the implicit function f can take inside some region of space. Kalra and Barr [1989] successfully rendered so called *LG-surfaces* by advancing rays inside an octree structure. For a LG-surface, each cell of the octree defines a Lipchitz bound L for f and another Lipchitz bound G for the derivative of f along the ray direction. Hart [1996] also uses Lipschitz bounds in his *sphere tracing* method. Unlike Kalra and Barr, it is not necessary to employ Lipschitz bounds for the derivatives of f. The method works by marching along a ray with steps that are guaranteed not to cause intersection with the surface. In both the LG-surface method and in sphere tracing it is necessary to specify *a priori* Lipchitz bounds related to the function f that one wishes to use. That can be difficult in a general case although Kalra and Barr and also Hart present bounds for some commonly used functions. If the Lipschitz bounds are not optimal, these methods will converge more slowly. The sphere tracing method of Hart [1996] is explained in more detail in the Section 5.2. Worley and Hart [1996] introduced several optimisations in the sphere tracing method for the case of implicit surfaces generated from hypertextures. The improved sphere tracing method takes into account the fact that hypertextured objects are often generated from the sums of many procedural noise functions. Other optimisations include a spatial coherence technique to reduce the number of function evaluations and image coherence and overshooting techniques to increase the stepping size along the rays.

In the category of interval range estimation methods, Mitchell [1990] computes ray-surface intersections with interval arithmetic. Interval arithmetic (IA) is a framework that replaces arithmetic operators and function evaluations on real numbers with equivalent operators and functions that are evaluated on intervals [Moore, 1966]. With IA it is possible to obtain interval bounds for the variation of f along some arbitrary span along a ray. The method by Mitchell performs a recursive binary subdivision along the length of a ray, computing interval bounds

for the function and its derivative inside each ray span. Newton's method is used to find the root once the interval bounds indicate the function has become monotonic inside some ray span. The method by Mitchell was later extended to use affine arithmetic (AA), instead of IA [de Cusatis Jr. et al., 1999]. Ray casting with AA produces interval bounds that are much tighter than those obtained with IA, therefore increasing the efficiency of the intersection algorithm. Flóres et al. [2006] use IA to estimate intersections across regions of the image plane and not only along the length of individual rays. This allows regions of empty space on the image to be quickly skipped, causing the algorithm to concentrate on image regions where the implicit surface is projected. One advantage of interval methods over Lipschitz methods is that interval bounds are computed automatically and on the fly. It is not necessary to supply an initial parameter, in the form of a conservative estimate for the Lipchitz bound, that will ultimately determine the efficiency of the algorithm. The disadvantage of interval methods is that they can be much slower than Lipschitz based methods when an optimal Lipschitz bound can be found for the implicit surface.

Range estimation methods such as IA and AA can be applied to general implicit functions but they take on particularly simple shapes when the function is a polynomial, which, as previously mentioned, occurs in the case of algebraic surfaces (Section 2.2.2). This has led to the development of many variants of the IA and AA methods, valid only for polynomial functions, that are more accurate than their general counterparts. A survey of such methods is presented by Martin et al. [2002], in the context of the plotting of algebraic curves. These methods can also be used for ray casting algebraic implicit surfaces simply by increasing the dimensionality of the implicit function from two to three dimensions. A novel ray casting method, based on the recursive evaluation of a Taylor series expansion of the implicit function, has been shown to be more efficient for algebraic surfaces than the IA and AA variants [Shou et al., 2006].

There is currently much research interest in the parallelisation of ray casting algorithms for implicit surfaces. Knoll et al. [2007] use SIMD language extensions to compute the bounds of IA estimates in parallel for ray-surface intersections. They also exploit spatial coherence by casting several rays in parallel into the scene. GPU hardware is able to successfully exploit the fact that different pixels can be rendered independently. This allows pixels to be distributed in parallel among the several hardware shaders of the GPU. Loop and Blinn [2006] use Bezier clipping to render piecewise algebraic surfaces (Section 2.2.3). de Toledo et al. [2007] render cubic and quartic algebraic surfaces with several iterative methods. Knoll et al. [2009] use reduced affine arithmetic (refer to Section 5.3.3) to ray cast general implicit surfaces. Kanamori et al. [2008] also use Bezier clipping to render blobby surfaces. Reimers and Seland [2008] modify the implicit function of algebraic surfaces to account for the perspective effects, transforming the view frustum into a unit cube, and then solve in parallel for the ray intersection points.

5.2 Lipschitz Continuous Implicit Surfaces

The continuity of an implicit surface is directly related to the continuity of its implicit function. The standard definition of continuity is due to Cauchy. In its three-dimensional form, it states that a function $f : D \subseteq \mathbb{R}^3 \rightarrow \mathbb{R}$ is continuous at some point $\mathbf{x}_0 \in D$ when:

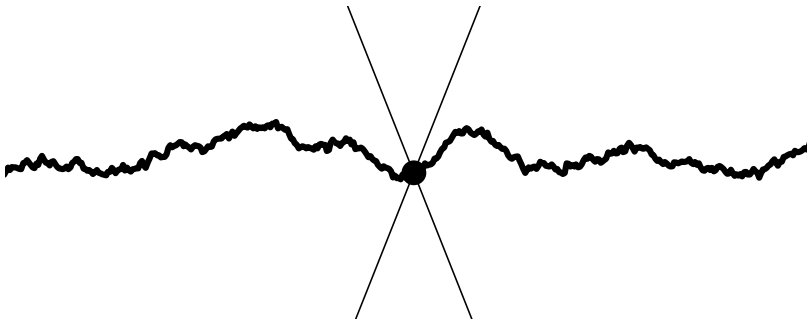


Figure 5.2: A geometrical interpretation of the Lipschitz constant. For every point, the graph must be bounded by the two lines with slopes of $+\lambda$ and $-\lambda$.

$$\forall \varepsilon > 0,\ \exists \delta > 0,\ \forall \mathbf{x}_1 \in D:\ \|\mathbf{x}_1 - \mathbf{x}_0\| < \delta \ \Rightarrow\ |f(\mathbf{x}_1) - f(\mathbf{x}_0)| < \varepsilon. \tag{5.4}$$

The Cauchy definition tells us that the function is continuous at $\mathbf{x}_0$ if, given any value f_1 of the function arbitrarily close to $f(\mathbf{x}_0)$, it is possible to find a point $\mathbf{x}_1$ that is also arbitrarily close to $\mathbf{x}_0$ such that $f(\mathbf{x}_1) = f_1$. The Cauchy definition of continuity is used throughout this thesis unless otherwise stated.

Lipschitz continuity represents a stronger statement about function continuity than the Cauchy definition. It relies on a parameter λ called a *Lipschitz bound.* A function is Lipschitz continuous on its domain D when:

$$\exists \lambda > 0,\ \forall \mathbf{x}_1, \mathbf{x}_2 \in D:\ |f(\mathbf{x}_1) - f(\mathbf{x}_2)| < \lambda \|\mathbf{x}_1 - \mathbf{x}_2\|. \tag{5.5}$$

If a value λ exists such that (5.5) is verified then any other value greater than λ is also a Lipchitz bound. The smallest of all possible values for λ is the *Lipschitz constant* of f on its domain D. A function may have different Lipschitz constants for different domains $D \subseteq \mathbb{R}^3$. It is also possible that a function is not Lipschitz continuous over the entire space $\mathbb{R}^3$ but becomes so for a smaller domain $D \subset \mathbb{R}^3$. Every Lipschitz continuous function is also continuous but the converse may not be true. One such example is the function $1/\|\mathbf{x}\|$ in the domain $D = \mathbb{R}^3 \setminus \{\mathbf{0}\}$. The function is continuous over its entire domain but not Lipschitz continuous. This is because the function becomes increasingly steeper close to the origin so that it is not possible to find a finite λ that will verify (5.5) everywhere on D.

Figure 5.2 exemplifies the concept of Lipschitz continuity for a one-dimensional stochastic process. For any point $f(x_0)$, the graph of the process must be delimited within the lines of slopes $\pm\lambda$ passing through $f(x_0)$. This example makes it clear that Lipschitz continuity imposes a restriction on the maximum rate of change of the function. Specifically, the function cannot change at a rate larger than λ at any point. If, besides being Lipschitz continuous, a function f is also differentiable then the Lipschitz constant of f is given by:

$$\lambda = \sup_{\mathbf{x} \in D} \|\nabla f(\mathbf{x})\|. \tag{5.6}$$

The choice of a stochastic process for the example of Figure 5.2 is intentional. As explained in Chapter 3, a stochastic process can be approximated in a procedural manner by summing procedural noise functions. Although it is not demonstrated here, all four types of noise functions

presented in Chapter 4 are Lipschitz continuous. If, furthermore, all the modulation functions that are part of a hypertexturing hierarchy (Section 2.4.1) are also Lipschitz continuous then the implicit surface itself is Lipschitz continuous. Ray casting algorithms that rely on this type of continuity can then be used successfully on hypertextured implicit functions.

5.2.1 Sphere Tracing

Lipschitz continuity provides a bound for the intersection distance between a ray and an implicit surface. Consider a point $\mathbf{x}_0$ that is placed outside the surface so that $f(f_0(\mathbf{x}_0), \mathbf{x}_0) > 0$. Consider also another point $\mathbf{x}_1$ on the surface so that $f(f_0(\mathbf{x}_1), \mathbf{x}_1) = 0$. Application of (5.5) to $\mathbf{x}_0$ and $\mathbf{x}_1$ leads to:

$$|f(f_0(\mathbf{x}_1), \mathbf{x}_1) - f(f_0(\mathbf{x}_0), \mathbf{x}_0)| = f(f_0(\mathbf{x}_0), \mathbf{x}_0) < \lambda \|\mathbf{x}_1 - \mathbf{x}_0\| \Rightarrow$$
$$\Rightarrow \|\mathbf{x}_1 - \mathbf{x}_0\| > f(f_0(\mathbf{x}_0), \mathbf{x}_0)/\lambda. \quad (5.7)$$

Because the point $\mathbf{x}_1$ on the surface was chosen arbitrarily, the distance from $\mathbf{x}_0$ to the surface is guaranteed to be larger than $f(f_0(\mathbf{x}_0), \mathbf{x}_0)/\lambda$. In other words, any point at a distance to $\mathbf{x}_0$ smaller than $f(f_0(\mathbf{x}_0), \mathbf{x}_0)/\lambda$ is guaranteed to be outside the surface. There is an *unbounding sphere* centred on $\mathbf{x}_0$ and with radius $f(f_0(\mathbf{x}_0), \mathbf{x}_0)/\lambda$ that does not contain any part of the surface within itself. Ultimately, for Lipschitz continuous implicit surfaces, the implicit function $f(f_0(\mathbf{x}), \mathbf{x})$ represents an *algebraic distance* from point $\mathbf{x}$ to the surface. This is not the same as the standard Euclidean distance from a point to a surface but verifies the requirements of a distance metric nevertheless.

The previous results can be applied to points that are placed along a ray, i.e. points parameterised as $\mathbf{x} = \mathbf{r}(t)$. For a point $\mathbf{r}(t_i)$, it is possible to go a further distance $f(f_0(\mathbf{r}(t_i)), \mathbf{r}(t_i))/\lambda$ along the ray without intersecting the surface. Considering that $f(f_0(\mathbf{r}(t_i)), \mathbf{r}(t_i)) = g(t_i)$, where g is the auxiliary function of equation (5.3), this can be expressed as an iterative equation for the distance t_i:

$$t_{i+1} = t_i + g(t_i)/\lambda. \quad (5.8)$$

There are two possible outcomes to the successive application of (5.8). Either the sequence t_i, with $i = 0, 1, 2, \ldots$, converges to a finite value or it diverges. The first case corresponds to the nearest intersection along the ray being found at the distance $t = \lim_{i \to \infty} t_i$. The second case corresponds to the situation where the ray does not intersect with the surface. Figure 5.3 shows the case of an intersecting ray. The sequence of unbounding spheres along the ray is also shown. The spheres gradually decrease in size as the current point $\mathbf{r}(t_i)$ along the ray approaches the surface. This is the reason behind the name of "sphere tracing", given by Hart [1996] when he first proposed the algorithm.

Sphere tracing is a robust ray-surface intersection algorithm with the potential to be very efficient because it only requires the point-wise evaluation of the implicit function. It is robust because of equation (5.7), which guarantees that the iterations (5.8) never pass over a root. Whether the algorithm is actually efficient or not relies on the complexity of the implicit function and quality of the Lipschitz bound λ used in (5.8). If only a very conservative Lipschitz bound can be found for f, meaning that λ has an unnecessarily high value, the algorithm becomes quite slow because λ appears as a denominator to $g(t_i)/\lambda$. Many small steps need to

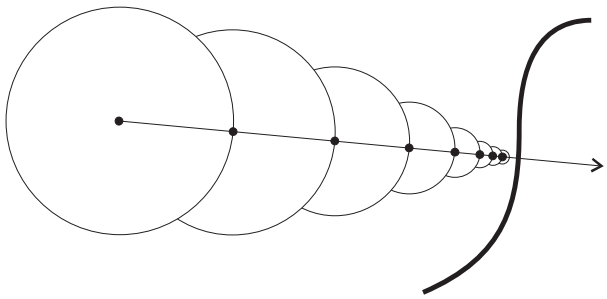


Figure 5.3: Sphere tracing a ray towards an implicit surface. The algorithm takes steps along a ray that converge towards the nearest intersection point.

be taken in this situation. The optimal case occurs when the Lipschitz constant of f is known (the smallest value of λ that still verifies equation (5.5)). In this case, sphere tracing can be significantly faster than any other robust intersection algorithm for general implicit surfaces. It is also possible to use a value for λ smaller than the Lipschitz constant to further speedup the intersection calculations but the algorithm is no longer guaranteed to be robust as some intersection points may occasionally be missed.

One drawback of the sphere tracing algorithm is that it can only be used to find the first intersection point along a ray. The iterations (5.8) converge towards the distance to the first intersection point and don't go any further. In a naïve approach, it could be possible to add a small offset Δt to t after the first intersection is found and restart the algorithm, hoping that it would converge to the next intersection point[2]. The problem is what value to give to Δt. If Δt is too small, it may not be enough to skip over the current intersection point and restarting the algorithm will have no effect. If Δt is too large, it may skip over more than one intersection point and the algorithm will no longer be robust.

5.2.2 Speculative Sphere Tracing

Worley and Hart [1996] proposed a speedup technique for sphere tracing that takes steps along the ray larger that those allowed by equation (5.8). There is an *overshoot factor* $\eta > 1$ that is used to increase the step sizes:

$$t_{i+1} = t_i + \eta g(t_i)/\lambda. \tag{5.9}$$

This can be called a speculative sphere tracing technique because the algorithm tries to gamble with the probability that the greater step size will still be valid, i.e. it will not be so large as to pass into the inside of the surface, skipping over the intersection point. By using larger step sizes, it is possible to accelerate the convergence of the algorithm. The validity of the increased step sizes is tested by checking the unbounding spheres at the beginning and end of each step. At the beginning of a step there is a sphere centred at point $\mathbf{r}(t_i)$ and with radius $r_i = g(t_i)/\lambda$ and at the end of the same step there is a similar sphere for distance t_{i+1} along the ray with radius $r_{i+1} = g(t_{i+1})/\lambda$ (Figure 5.4). The step from t_i to t_{i+1} is valid provided that the corresponding segment along the ray is completely covered by the union of the two spheres. As

[2]This would also require that the step lengths be computed as $|g(t_i)|/\lambda$ (note the modulus) to account for the fact that $\mathbf{r}(t_i)$ could now be inside the surface, where $f < 0$.

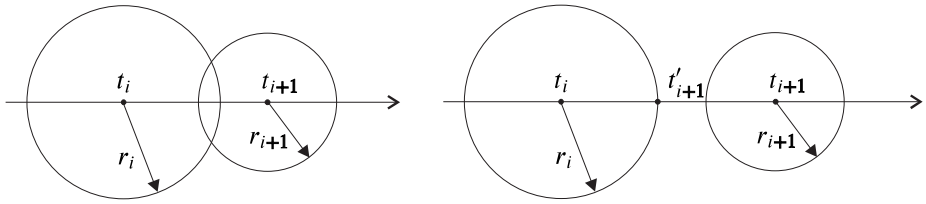


Figure 5.4: The two cases of speculative tracing. A valid overshooting step is shown on the left and a step that requires backtracking is shown on the right.

previously explained, the spheres represent regions of empty space and a segment of ray that goes through them is known not to intersect the surface. This is expressed by the condition:

$$t_{i+1} - t_i \leqslant r_i + r_{i+1}. \tag{5.10}$$

If the above condition is not satisfied the spheres are disjoint and the algorithm needs to backtrack to the best known good position. This is the position $t'_{i+1} = t_i + r_i$ that corresponds to the usual sphere tracing step given by equation (5.8). Figure 5.4 shows, on the left, an example of a valid speculative step and, on the right, an example of a step that requires backtracking.

The algorithm is shown in pseudo-code in Figure 5.5. It returns the distance along the ray towards the nearest intersection point. The iterations start from an initial distance t_{MIN} and proceed up to a maximum distance t_{MAX}. If t_{MAX} is reached, the algorithm considers that no intersection was found and returns $+\infty$. The values for t_{MIN} and t_{MAX} are found by initially intersecting the ray with a simple bounding geometry, such as a box or a sphere, that encircles the implicit surface. Convergence is assumed to have been achieved when the step length goes below a supplied threshold value ϵ. Worley and Hart suggest that an overshooting factor of 46% ($\eta = 1.46$) provides the best speedups for general surfaces. Larger factors begin to introduce too many backtracking steps, which end up slowing down the algorithm because extra radii $r'_{i+1} = g(t'_{i+1})/\lambda$ need to be computed.

5.2.3 Results

The sphere tracing algorithm is evaluated by rendering one hypertexture for each of the three procedural noise functions, described in Chapter 4, that have been implemented. The results from this section are then used to compare the performance of the sphere tracing algorithm against the performance of the new reduced affine arithmetic algorithm that is presented in Section 5.4. Such a comparison is done in Section 5.5 of this chapter. The hypertexture is performed on a unit radius implicit sphere, given by $f_0(\mathbf{x}) = \|\mathbf{x}\| - 1$. Three layers of procedural noise are added according to:

$$f(f_0(\mathbf{x}), \mathbf{x}) = f_0(\mathbf{x}) + 0.3 f_N(4\mathbf{x}) + 0.15 f_N(8\mathbf{x}) + 0.05 f_N(16\mathbf{x}). \tag{5.11}$$

A Lipschitz bound for the function (5.11) depends on the available Lipschitz bound λ_N for the procedural noise function used. It becomes a Lipschitz constant if λ_N is also a Lipschitz constant. Note that $f_0(\mathbf{x})$ is a Euclidian distance function and, therefore, has a Lipschitz constant of unity.

$$\lambda = 1 + 0.3 \times 4 \times \lambda_N + 0.15 \times 8 \times \lambda_N + 0.05 \times 16 \times \lambda_N = 1 + 3.2\lambda_N. \tag{5.12}$$

```
let t_i = t_MIN;                    // Initial position
let r_i = g(t_i)/λ;                 // Initial radius
while t_i < t_MAX
   let t_{i+1} = t_i + η r_i;       // New position
   let r_{i+1} = g(t_{i+1})/λ;      // New radius
   if t_{i+1} − t_i > r_i + r_{i+1}
      let t_{i+1} = t_i + r_i;      // Backtrack
      let r_{i+1} = g(t_{i+1})/λ;   // Adjust new radius
   if t_{i+1} − t_i < ε
      return t_{i+1};               // Found intersection
   let t_i = t_{i+1};               // Take step
   let r_i = r_{i+1};
return +∞;                          // No intersection found
```

Figure 5.5: The speculative sphere tracing algorithm with an overshooting factor η.

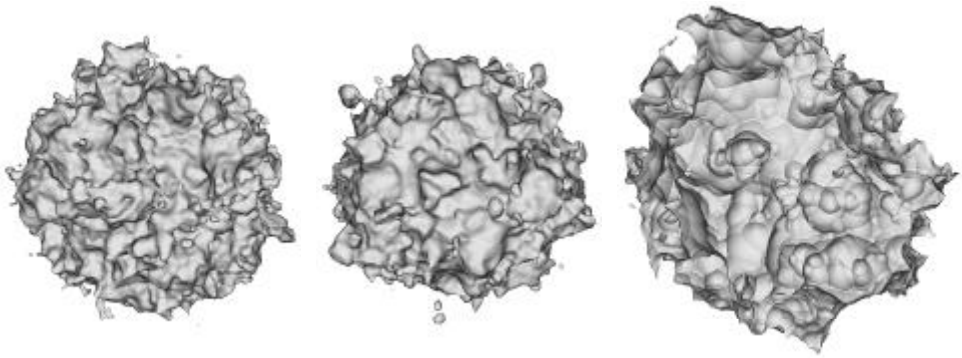


Figure 5.6: Hypertextures generated with, from left to right, gradient noise, sparse convolution noise and cellular texture noise. These hypertextures are used to test the sphere tracing algorithm.

Figure 5.6 shows 800 × 800 resolution images of the hypertextures that were tested. Table 5.1 gives the timing results obtained on a dual Athlon 2.1GHz processor. Results are given for gradient noise, sparse convolution noise, with $K = 2$ node points per lattice cell, and cellular texture noise, with $a_1 = 1$ and $a_2 = a_3 = a_4 = 0$ (refer to Chapter 4).

The values used for λ_N are shown in Table 5.1. In the case of gradient noise and cellular texture noise, these are actually Lipschitz constants. For cellular texture noise, in particular, λ_N is always equal to $|a_1| + |a_2| + |a_3| + |a_4|$ because all four of the noise kernels are distance functions and have a unitary Lipschitz constant. The Lipschitz constant for sparse convolution noise is equal to $15K$. This constant accounts for the longest possible gradient vector of the noise function, which occurs when $8K$ node points happen to be exactly coincident over a lattice corner point that is shared by eight lattice cells[3]. Given that all node points are randomly

[3]The constant is equal to $1.875 \times 8K = 15K$, where 1.875 is the maximum slope of the quintic polynomial used for the kernel.

	λ_N	Avg. Steps p/ray		Time		Speedup
		Sph.T.	Spec.Sph.T.	Sph.T.	Spec.Sph.T.	
Gradient Noise	2.75	78.19	55.37 (0.78)	1m 26s	1m 00s	42%
Sparse Convolution Noise	2.00	77.71	54.27 (0.71)	13m 43s	9m 37s	43%
Cellular Texture Noise	1.00	30.77	26.51 (4.05)	10m 07s	8m 45s	16%

Table 5.1: Statistics for the two sphere tracing algorithms. Refer to the text for an explanation of these statistics. "Sph.T." refers to sphere tracing while "Spec.Sph.T" refers to speculative sphere tracing. The quantities in parenthesis are the average number of backtracking steps.

distributed inside each lattice cell, the probability that $8K$ node points will all fall very close to the same lattice corner is infinitesimally small. For this reason, a value of $\lambda_N = 2$ was attempted for sphere tracing the hypertexture based on sparse convolution noise. No wrong ray-surface intersections were detected in the resulting image but because $\lambda_N < 15K$ it is not possible to guarantee that intersections will always be correctly computed. This issue with sparse convolution noise could be avoided if the Poisson sample distribution of node points per lattice cell was replaced with a Poisson-disc sample distribution [Lagae and Dutré, 2007]. The Poisson-disc distribution enforces a minimum distance constraint between samples. This would eliminate the possibility of node points being coincident and, in turn, would lower the Lipschitz constant considerably, making it usable for robust sphere tracing. The generation of sparse convolution noise based on a Poisson-disc distribution of node points is a topic for future research.

Timing results in Table 5.1 are given for the original sphere tracing algorithm and for the speculative sphere tracing algorithm. In both cases, the average number of steps taken along the rays is shown. For speculative sphere tracing, the average number of backtracking steps per ray is also shown in parenthesis next to the number of forward steps. Speculative sphere tracing can generate the same results faster than the original sphere tracing algorithm by reducing the number of steps taken per ray. Because each sphere tracing step requires one evaluation of the implicit function, reducing the number of steps leads to a reduction in the rendering time. It can be seen from these results that the rendering time for each noise function varies linearly with the number of steps. For gradient noise and sparse convolution noise, the speedup is equal to 42% and 43%, respectively (the speedup being the rendering time for the original sphere tracing algorithm divided by the rendering time for speculative sphere tracing minus one). This value is closely related to the overshooting factor $\eta = 46\%$ that was used – if no backtracking ever occurred, the speedup would be exactly equal to η. In the case of cellular texture noise, the speedup is only 16%. This is related to the much larger average of backtracking steps taken per ray (4.05) when compared with the other two noise functions with averages of approximately 0.7 backtracking steps per ray. The larger percentage of backtracking steps for cellular texture noise is due to this function not being smooth, which makes it possible for a ray to overshoot over a surface crease and have to backtrack as a consequence.

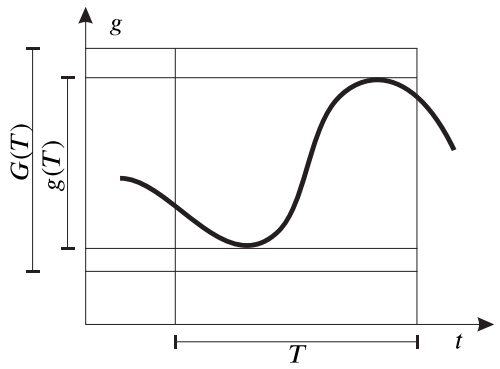


Figure 5.7: The image $g(T)$ and one possible interval extension $G(T)$ of a function g over T.

5.3 Range Estimation Techniques

Range estimation techniques form the basis for another type of ray-surface intersection methods that, unlike sphere tracing, do not require the knowledge of a Lipschitz bound in advance. The general idea is that, given an interval $T = [t_{MIN}, t_{MAX}]$ corresponding to the distance along some span of the ray, a range estimation technique will produce an interval for the variation of the auxiliary function $g(t)$ (recall the discussion concerning equation (5.3)). One wishes to find the *image* $g(T)$ of the function $g(t)$ as t takes values from T:

$$g(T) = \{ g(t) : t \in T \}. \tag{5.13}$$

Range estimation techniques attempt to guess the image $g(T)$ for some interval T. Very often, however, an exact determination of $g(T)$ is difficult to perform, especially when the function g that is being analysed is complex. Range estimation techniques produce instead an *interval extension* $G(T)$ of $g(T)$. An interval extension is allowed to overestimate the size of the image but it must always verify an *inclusion property*, which states that $G(T) \supseteq g(T)$. Figure 5.7 exemplifies this. The quality of a particular range estimation depends on how snugly does $g(T)$ fit inside the interval extension. If the estimated interval does include $g(T)$ but is much larger than the latter, the quality of the estimate is very poor to the point of possibly giving little useful information about the function. Despite the fact that range estimations may often suffer from over-conservativeness, it is still possible to make decisions about the presence of a root $g(t) = 0$ for some $t \in T$. If $0 \notin G(T)$ then it is known with certainty that no root can be present in T. If, on the other hand, $0 \in G(T)$, a root may or may not exist in T. This is a consequence of the interval $G(T)$ being potentially larger than $g(T)$. The fact that $0 \in G(T)$ does not necessarily mean that $0 \in g(T)$. The best strategy in this case is to split T into smaller intervals and compute the interval extensions for each of them.

The next three sections present three range estimation techniques that can be used to compute interval extensions for any function g. These techniques vary in their computational complexity and in their accuracy when estimating the true image of g inside some interval T. Generally, the greater the accuracy of a range estimation technique, the greater its computational complexity.

5.3.1 Interval Arithmetic

An interval arithmetic (IA) representation $X = [x_a, x_b]$ expresses the limited knowledge about the value of the quantity x. All that can be said with this representation is that $x_a \leqslant x \leqslant x_b$. It is possible to derive an arithmetic for intervals, similar to the arithmetic of the real numbers, by replacing all real operations with equivalent operations working on interval quantities [Moore, 1966]. Interval arithmetic finds application in many computer graphics problems [Snyder, 1992]. When performing an arithmetic operation on intervals, the extremes x_a and x_b of the result are computed so that all possible outcomes of the real operation are bounded by the interval $[x_a, x_b]$. Several examples of IA operations are given below. These consist of the addition and subtraction of two IA quantities $X = [x_a, x_b]$ and $Y = [y_a, y_b]$, the addition, subtraction and multiplication of a single IA quantity with a scalar α and the multiplication of two IA quantities:

$$X + Y = [x_a + y_a, x_b + y_b], \tag{5.14a}$$

$$X - Y = [x_a - y_b, x_b - y_a], \tag{5.14b}$$

$$\alpha \pm X = [x_a \pm \alpha, x_b \pm \alpha], \tag{5.14c}$$

$$\alpha X = \begin{cases} [\alpha x_a, \alpha x_b] & \text{for } \alpha \geqslant 0, \\ [\alpha x_b, \alpha x_a] & \text{for } \alpha < 0, \end{cases} \tag{5.14d}$$

$$XY = [\min(x_a y_a, x_a y_b, x_b y_a, x_b y_b), \max(x_a y_a, x_a y_b, x_b y_a, x_b y_b)]. \tag{5.14e}$$

The transcendental functions in $\mathbb{R}$ can also be generalised to receive intervals as arguments and return intervals. It is valid, for example, to write an IA expression like $Y = \sin X$. To guarantee that the inclusion property is verified with IA, it is necessary to control the rounding mode of the floating point hardware. When computing the upper value of an interval, the rounding mode must be set to round up and the opposite must be true when computing the lower interval value. In practice, the continual setting of the hardware rounding mode makes computations very slow as the floating point pipeline is repeatedly flushed. Most IA implementations ignore rounding control and assume a floating point hardware with infinite precision. Although it is no longer possible to claim that the inclusion property is verified, this simplification does not produce any undesired effects in most applications.

Interval arithmetic is a very fast range estimation technique because for each IA operation only the computation of two numbers is required. The main problem of IA, however, is over-conservativeness. The radius $|X| = (x_b - x_a)/2$ of an interval X can be used as a measure of this conservativeness in IA computations. Consider the subtraction $X - X$ of an IA quantity with itself. Naturally, the result should be zero. Strict application of the IA rules for subtraction, however, lead to the result $[x_a - x_b, x_b - x_a]$. The correct result of zero is indeed contained in the interval but its radius $|X - X| = x_b - x_a$ is twice the radius $|X|$ of the original quantity. This is because IA was not able to recognise that the two operands to the subtraction are correlated (in fact, they are the same in this example) and had to apply conservative rules to ensure the resulting interval would enclose every possible answer. The over-conservativeness problem of IA becomes worse as progressively longer sequences of IA operations are performed. As a sequence of computations progresses, the radii of the computed intervals continually increase, making the estimates ever more conservative.

5.3.2 Affine Arithmetic

Affine arithmetic is a technique proposed by de Figueiredo and Stolfi [2004]. This technique can provide accurate estimates for the interval extension function $G(T)$ featured in the ray-surface intersection problem. Affine arithmetic represents an improvement over the previous interval arithmetic technique. The representation of some quantity with affine arithmetic (AA) tries to model the uncertainties about that quantity so that it is always bounded inside a known interval. The advantage over the simpler interval arithmetic framework is that AA tries to keep correlations between quantities, calculated along some arbitrarily long chain of computations. AA keeps correlations between similar quantities through the use of *error symbols*. A quantity $\hat{x}$ in AA is represented as a central value x_0 plus a sequence of error symbols e_i, each with its associated error coefficient x_i:

$$\hat{x} = x_0 + x_1 e_1 + x_2 e_2 + \cdots + x_n e_n. \tag{5.15}$$

The error symbols lie in the interval $[-1, +1]$ but are otherwise unknown and the coefficients x_i express the contribution of each symbol to the AA quantity. Error symbols can be shared among several AA quantities and that is how correlation information can be kept among related quantities. The computation of affine operations on AA quantities does not result in the creation of any new error symbols. For two AA quantities $\hat{x}$, $\hat{y}$ and a scalar α, the affine operations are:

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) + (x_1 \pm y_1)e_1 + \cdots + (x_n \pm y_n)e_n, \tag{5.16a}$$

$$\alpha\hat{x} = (\alpha x_0) + (\alpha x_1)e_1 + \cdots + (\alpha x_n)e_n, \tag{5.16b}$$

$$\alpha \pm \hat{x} = (\alpha \pm x_0) + x_1 e_1 + \cdots + x_n e_n. \tag{5.16c}$$

The previous equations assume that rounding errors are ignored. Otherwise, affine operations, just like their non-affine counterparts, would require the insertion of a new error symbol that accounts for the rounding error of the operation. This would imply control of the hardware rounding mode and, similar to interval arithmetic, it is often avoided. It is easy to check, however, that, unlike with interval arithmetic, the equation $\hat{x} - \hat{x} = 0$ is always verified with affine arithmetic. This result is a consequence of the application of the expression (5.16a) for the difference between two AA quantities. If the two quantities are the same, in particular, all the common error symbols cancel out, giving the correct result of zero.

For non-affine operations, like multiplication or square root, a new error symbol must be introduced to express the non-linearity of the operator. The result of some non-affine operator is a new AA quantity $\hat{z} = z_0 + z_1 e_1 + \cdots + z_n e_n + z_k e_k$, where the extra error symbol e_k has been added to the representation. For example, if $\hat{z} = \hat{x}\hat{y}$, the coefficients of $\hat{z}$ are given by:

$$\begin{aligned} z_0 &= x_0 y_0, \\ z_i &= x_0 y_i + y_0 x_i \qquad \text{for } i = 1, \ldots, n, \\ z_k &= \sum_{i=1}^{n} |x_i| \cdot \sum_{i=1}^{n} |y_i|. \end{aligned} \tag{5.17}$$

The coefficient z_k in (5.17), associated with the newly inserted error symbol e_k, is positive and represents the magnitude of the error introduced by the linearisation of $\hat{x}\hat{y}$ into an affine form. The same property of z_k holds in the case of all the other non-affine operations.

As a sequence of AA operations progresses, quantities have an increasingly larger number of error symbols, slowing down subsequent AA computations and increasing the memory requirements. This is because, if the uncertainty associated with some error symbol e_i of an AA quantity is not shared with any other AA quantities, the latter must all have a null coefficient for e_i. When implementing AA, cumbersome book-keeping routines are required to manage the large but sparse sequences of error coefficients[4]. This inefficiency associated with AA representations has been acknowledged by Stolfi and de Figueiredo [1997]. They recommend that a procedure called *condensation* be periodically applied on an AA quantity when its sequence of error symbols grows too large. An AA quantity $\hat{x}$, with m error symbols, can be condensed to form another quantity $\hat{y}$, with $n < m$ error symbols, according to:

$$\begin{aligned} y_i &= x_i \qquad \text{for } i = 0, \ldots, n-1, \\ y_n &= \sum_{i=n}^{m} |x_i|. \end{aligned} \tag{5.18}$$

Condensation brings some large AA quantity $\hat{x}$ down to a more manageable size but it also destroys the correlation information that was kept in the error symbols e_i, with $n < i \leqslant m$. This is not a problem if the aforementioned error symbols were unique to $\hat{x}$. The accuracy of subsequent computations is affected, however, if the e_i were being shared with other AA quantities that are involved in those computations.

5.3.3 Reduced Affine Arithmetic

The problem of having to deal with ever increasing sets of error symbols in standard AA has motivated the use of a reduced AA form for ray casting implicit surfaces. Reduced affine arithmetic keeps a small and fixed number of error symbols per AA quantity. Any new error symbols that need to be introduced due to non-affine operations are automatically condensed, preventing the sequence of error symbols from growing. Reduced affine arithmetic was proposed by Messine as the first of several possible extensions to affine arithmetic [Messine, 2002]. In his work, this first extension is called Affine Form 1 (AF1).

As Section 4.1 explained, procedural noise functions are built from sums of independent kernel functions. No correlations exist between the sequence of computations that are performed for any two kernel functions during the computation of f_N. Correlations during the evaluation of a procedural noise function have a very localised nature and are isolated inside the sequence of computations for each individual kernel. The only global correlation that is expected to exist throughout the computation of f_N is related to the uncertainty with the position of the root t along a ray. This happens when f_N is embedded in the equation $g(t) = 0$ that must be solved by the ray caster.

The reduced AA representation for the distance along a ray $\hat{t}$, used in ray casting, is equivalent to an AF1 representation which considers only two error symbols: the symbol e_1, expressing

[4]A simpler alternative would be to store in full all the coefficients of an AA quantity. This would be wasteful of memory, considering that many of those coefficients are zero. It would also be wasteful of CPU cycles. As example (5.17) shows, the computation of AA operations involves a loop over all the e_i error symbols. If the sequence of coefficients is sparse, many of the loop iterations become unnecessary.

the uncertainty along the ray, and the symbol e_2, which is always non-negative and expresses uncertainties involved in the computation of $\hat{t}$ alone. The error symbol e_1 is the only symbol that is shared between $\hat{t}$ and other AA quantities. The expression for $\hat{t}$ is:

$$\hat{t} = t_0 + t_1 e_1 + t_2 e_2. \tag{5.19}$$

Reduced AA operations are always followed by a condensation step to remove any extra error symbols that would have been introduced otherwise. Reduced AA can, therefore, be seen as a modification of affine arithmetic that employs an aggressive form of condensation. The case of the multiplication between two reduced AA quantities $\hat{x}$ and $\hat{y}$ is presented as an example. Originally, the result of this multiplication would give:

$$\hat{x}\hat{y} = x_0 y_0 + (x_0 y_1 + y_0 x_1)\, e_1 + x_0 y_2 e_{2y} + y_0 x_2 e_{2x} + (|x_1| + x_2) \cdot (|y_1| + y_2)\, e_k, \tag{5.20}$$

where the second error symbols of $\hat{x}$ and $\hat{y}$ have been written as e_{2x} and e_{2y}, respectively, to make it clear that they are not correlated. The last three terms of (5.20) are condensed into a new symbol e_2 that is unique to the $\hat{x}\hat{y}$ quantity, giving:

$$\hat{x}\hat{y} = x_0 y_0 + (x_0 y_1 + y_0 x_1)\, e_1 + \big(|x_0| y_2 + |y_0| x_2 + (|x_1| + x_2) \cdot (|y_1| + y_2)\big) e_2. \tag{5.21}$$

In practice, all operations in reduced AA are modified so that a condensation step is automatically built into them. The affine operators (5.16) now become:

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) + (x_1 \pm y_1) e_1 + (x_2 + y_2) e_2, \tag{5.22a}$$

$$\alpha \hat{x} = (\alpha x_0) + (\alpha x_1) e_1 + (|\alpha| x_2) e_2, \tag{5.22b}$$

$$\alpha \pm \hat{x} = (\alpha \pm x_0) + x_1 e_1 + x_2 e_2. \tag{5.22c}$$

Reduced affine arithmetic is only slightly more expensive to compute than interval arithmetic. It requires the computation of three numbers instead of two for each quantity. Depending on the nature of the correlations that exist when computing the interval extension $G(T)$ for ray casting, reduced affine arithmetic can be as accurate as standard affine arithmetic.

5.4 Ray Casting with Reduced Affine Arithmetic

Figure 5.8 lists a robust algorithm that is used to find the first intersection point between a ray and the implicit surface [Gamito and Maddock, 2007d]. The algorithm relies on the subdivision of an initial interval T_0 and the information returned by the interval extension function $G(T)$, which is evaluated with reduced affine arithmetic. The algorithm is robust because, as proven by Comba and Stolfi [1993], the interval extension $G(T)$ can be reliably computed with affine arithmetic. Reduced affine arithmetic, being a particularisation of the more general affine arithmetic framework, does not change this property of the interval extension estimates $G(T)$. A stack is used to store the subdivided intervals that are waiting to be tested for the existence of roots. The algorithm terminates either when a small enough interval bounding a root has been found or when the stack becomes empty. The latter scenario occurs in situations where a ray does not intersect the surface. Each interval taken from the stack is subdivided if there is

```
push T_0 = [t_MIN, t_MAX] onto stack;
while stack not empty
  pop T = [t_a, t_b] from the stack;
  if 0 ∈ G(T)              // Evaluate G(T) with reduced affine arithmetic
    if t_b − t_a < ε       // Converged to a root
      return t_a;          // t_a is outside the surface and t_b is inside
    let t_i = (t_b + t_a)/2; // Compute midpoint of interval
    let T_l = [t_a, t_i];  // Nearest half of interval T
    let T_r = [t_i, t_b];  // Farthest half of interval T
    push T_r onto stack;   // Push farthest half onto stack first
    push T_l onto stack;
return +∞;                 // No intersection found
```

Figure 5.8: The ray-surface intersection algorithm that uses reduced affine arithmetic.

the possibility that it may contain a root. The order with which the two subintervals are then pushed onto the stack is not arbitrary. By pushing T_r first and then T_l, the nearest intersection is guaranteed to be found.

The algorithm of Figure 5.8 is a simplified version of the interval algorithm by Mitchell [1990]. In Mitchell's original algorithm, an interval extension $G'(T)$ of the derivative of g along the ray was also computed. When $0 \in G(T)$ and $0 \notin G'(T)$ were both verified, the function was known to have an isolated root inside the interval T and Newton's method could then be used to provide quadratic convergence. In the case of combinations of procedural noise functions, however, g varies erratically and only for very small intervals do the conditions for monoticity exist that enable us to isolate a single root. It was found that the use of $G'(T)$ does not provide any speedup while ray casting fractal implicit surfaces and, in fact, slows down the algorithm since two interval extensions have to be computed instead of one. de Cusatis Jr. et al. [1999] reached the same conclusion for the implicit surfaces that they were interested in rendering.

Although the algorithm as shown in Figure 5.8 only computes the nearest intersection point, it is easy to introduce modifications that will enable all intersection points to be found. The distance t_a that is returned on the sixth line of the algorithm can instead be appended to a list of distances to intersection points. The algorithm then continues along the ray until there are no more intervals on the stack to be tested. At completion, the list will have the distances to all intersections between the ray and the surface, sorted by increasing depth along the ray.

The starting point for the computation of the interval extension $G(T)$, as part of the intersection algorithm, is the conversion of the interval $T = [t_a, t_b]$ into the three coefficients that constitute the reduced AA form $\hat{t}$ for the interval, according to equation (5.19):

$$\begin{aligned} t_0 &= (t_b + t_a)/2, \\ t_1 &= (t_b - t_a)/2, \\ t_2 &= 0. \end{aligned} \tag{5.23}$$

The value t_0 is the central value of the interval. The value t_1 is its radius and, because there are no sources of error at the start, $t_2 = 0$. The expression for $g(\hat{t})$ is then evaluated with the

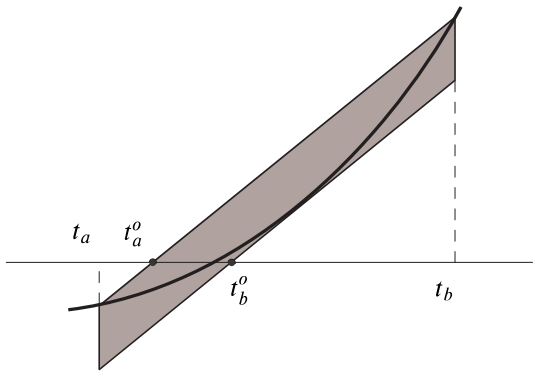


Figure 5.9: The information conveyed by reduced affine arithmetic for the behaviour of the function g inside an interval $T = [t_a, t_b]$.

application of the reduced affine arithmetic operators. The last stage in the computation of an interval extension is the conversion of the reduced AA form $g(\hat{t}) = g_0 + g_1 e_1 + g_2 e_2$ into the interval $G(T) = [g_a, g_b]$, which allows the test $0 \in G(T)$ to be performed trivially:

$$g_a = g_0 - |g_1| - g_2, \tag{5.24a}$$

$$g_b = g_0 + |g_1| + g_2, \tag{5.24b}$$

$$0 \in G(T) \Leftrightarrow g_a \leqslant 0 \leqslant g_b. \tag{5.24c}$$

5.4.1 Interval Optimisation

The idea of optimising the size of the interval bounding the first root of (5.3) was initially presented by de Cusatis Jr. et al. [1999]. It is presented here again in the framework of reduced affine arithmetic. When implementing the ray-surface intersection algorithm with affine arithmetic, it is possible to reduce the size of the interval T being tested at the start of each iteration, and prior to its subdivision, by taking advantage of the extra information provided by reduced affine arithmetic.

Figure 5.9 shows an example of the information conveyed by a reduced AA representation of the function g, evaluated inside some interval $T = [t_a, t_b]$ along the ray, for a situation where g increases smoothly. The purpose of the ray casting algorithm is to find the point where the graph of g crosses the horizontal axis. The reduced AA representation $g(\hat{t})$, where $\hat{t}$ encodes T in reduced AA form according to (5.23), is geometrically equivalent to a parallelogram that encloses the graph of g for the interval T. The bounding interval can be optimised by reducing it to $T^o = [t_a^o, t_b^o]$ prior to subdivision. It is clear from the figure that significant convergence towards the root is achieved with just a single evaluation of $g(\hat{t})$ in reduced AA form.

When computing $\hat{g} = g(\hat{t})$, it occurs that $\hat{g} = g_0 + g_1 e_1 + g_2 e_2$ and $\hat{t} = t_0 + t_1 e_1$, where the error symbols e_1 and e_2 are unknown but vary in the interval $[-1, +1]$. Putting the e_1 error symbol in evidence in the expression for $\hat{t}$ and replacing this in the expression for $\hat{g}$:

$$\hat{g} = \frac{g_1}{t_1}\left(\hat{t} - t_0\right) + g_0 + g_2 e_2. \tag{5.25}$$

Equation (5.25) is the equation for a line in the $\hat{g}$-$\hat{t}$ plane with a slope of g_1/t_1. By letting e_2 vary in $[-1, +1]$ and keeping $t_a \leqslant \hat{t} \leqslant t_b$ the parallelogram that is shown in Figure 5.9 is swept. The upper and lower edges of this parallelogram are obtained from (5.25) when $e_2 = \pm 1$. The intersections of these two edges with the horizontal axis give the new and optimised limits for the interval. Setting (5.25) equal to zero, with $e_2 = \pm 1$, and rearranging, leads to:

$$\hat{t} = t_0 - \frac{g_0}{g_1} t_1 \pm \frac{g_2}{g_1} t_1. \tag{5.26}$$

Independently of the sign of g_1 (g_2 is always positive and t_1 is also always positive because of the way it is constructed in equation (5.23)), the left and right solutions to (5.26) along the horizontal axis are, respectively:

$$t_a^o = \max\left(t_0 - \frac{g_0}{g_1} t_1 - \frac{g_2}{|g_1|} t_1, t_a\right), \tag{5.27a}$$

$$t_b^o = \min\left(t_0 - \frac{g_0}{g_1} t_1 + \frac{g_2}{|g_1|} t_1, t_b\right). \tag{5.27b}$$

The solutions to equation (5.26) are only used if they lead to a tighter interval than the original $[t_a, t_b]$, hence the min and max functions in equations (5.27). To summarise, the steps necessary to compute the interval extension $G(T)$ in the ray-surface intersection algorithm of Figure 5.8 are presented here, after the interval T has been removed from the stack:

1. Compute the reduced AA variable $\hat{t}$ from $T = [t_a, t_b]$, written as (5.19) and with coefficients given by (5.23).
2. Compute the reduced AA estimate $\hat{g} = g(\hat{t})$, using reduced affine arithmetic operators.
3. Compute the interval extension $G(T)$ from $\hat{g}$, using (5.24).

If, after step 3, it is found than $0 \in G(T)$, the optimised interval $T^o = [t_a^o, t_b^o]$ is obtained from the initial interval T, its reduced AA representation $\hat{t}$, and the estimate $\hat{g}$, using (5.27). It is T^o, rather than T, which is then subdivided and its subintervals pushed back onto the stack for further processing during subsequent iterations of the algorithm[5].

5.4.2 Application to Hypertexturing with Procedural Noise Functions

An analysis of the localised nature of the correlations during AA computations, as part of the evaluation of a procedural noise function, requires that the kernels for each individual noise function be examined in turn. In the case of the gradient noise function (Section 4.3), the kernel (4.9) has three AA multiplications, each of which would introduce new error symbols in a standard AA representation. However, once these three multiplications are performed,

[5]When $g_1 \to 0$, the enclosing parallelogram in Figure 5.9 tends toward an axis aligned rectangle. In the limit, no optimisation is possible and the original interval T must be subdivided.

the evaluation of $\phi(\mathbf{d}_i)$ is complete and the new error symbols can be safely condensed. At the same time, the AA evaluation of the cubic hermite polynomial h is performed through a direct process of Chebyshev affine approximation rather than applying all the usual algebraic operations [Stolfi and de Figueiredo, 1997; Heidrich et al., 1998]. This means that no internal correlations have to be considered during the evaluation of h because the reduced AA result is computed in one single step. In the end, one can say that the evaluation of $\phi(\mathbf{d}_i)$ with reduced AA does not lose any correlation information and has the same accuracy as standard AA.

In the case of the sparse convolution noise function (Section 4.4), the kernel (4.13) depends only on the distance $\|\mathbf{d}_i\|$ to some node point $\mathbf{x}_i$. This distance computation features four non-affine operations, namely three squares and one square root operation. The distance $\|\mathbf{d}_i\|$, however, is involved in the computation of ϕ_i alone and does not influence the other ϕ_l kernels that are required for the evaluation of f_N. The condensation of the new error symbols from the evaluation of $\|\mathbf{d}_i\|$ does not, therefore, lead to any loss of accuracy. The evaluation of the function h is performed directly by Chebyshev approximation and, again, one can say that a reduced AA computation of ϕ_i is as accurate as a standard AA computation.

In the case of the cellular texture noise function (Section 4.5), the kernel is evaluated by iteratively applying a binary minimum operator $\min(d_i, d_j)$ on all the pairs of distances $d_k = \|\mathbf{d}_k\|$ from $\mathbf{x}$ to the node points $\mathbf{x}_k$ that belong to the set $S(\mathbf{x})$. The minimum operator is evaluated with affine arithmetic according to the expression:

$$\min(d_i, d_j) = \frac{d_i + d_j}{2} - \frac{|d_i - d_j|}{2}. \tag{5.28}$$

For example, if $d_i > d_j$, then $\min(d_i, d_j) = (d_i + d_j)/2 - (d_i - d_j)/2 = d_j$. This exact cancellation effect can only be achieved if all correlations between $d_i = \|\mathbf{d}_i\|$ and $d_j = \|\mathbf{d}_j\|$ are maintained. However, the distance computations with reduced AA involve the condensation of error symbols, as previously seen for the case of the sparse convolution noise function, and an exact cancellation cannot be obtained. For this reason, the application of reduced AA to cellular texture noise functions incurs some loss of accuracy when compared to standard AA.

5.4.3 Results

The reduced AA ray caster was tested with the same set of hypertextures that were previously used to test the sphere tracing algorithm (Section 5.2.3). A library of reduced AA operations was implemented and the implicit functions for the three hypertextures were recast in reduced AA format based on this library. A reduced AA model of the cellular texture noise function was implemented that can use linear combinations of the ϕ_0 and ϕ_1 kernels only. It was found that the ϕ_2 and ϕ_3 kernels are too complex to implement when using affine arithmetic or even interval arithmetic. This is due to the difficulty in determining the third and fourth smallest distances in the set $S(\mathbf{x})$ when all the $\|\mathbf{d}_i\|$ have an arbitrary degree of uncertainty. For this reason, the current reduced AA implementation of cellular texture noise must enforce the restriction $a_2 = a_3 = 0$.

Tables 5.2, 5.3 and 5.4 show some statistics that enable a comparison between all the range estimation techniques for the procedural noise functions under consideration. The performance

	Avg. Evals. p/ray	Time
IA	116.57	7m 35s
Standard AA	61.69	40m 41s
Reduced AA	61.69	6m 43s
Reduced AA + Int. Opt.	46.39	5m 04s

Table 5.2: Statistics for the hypertexture based on gradient noise functions.

	Avg. Evals. p/ray	Time
IA	87.65	26m 46s
Standard AA	47.99	38m 07s
Reduced AA	47.99	13m 00s
Reduced AA + Int. Opt.	30.92	7m 58s

Table 5.3: Statistics for the hypertexture based on sparse convolution noise functions.

	Avg. Evals. p/ray	Time
IA	45.80	30m 51s
Standard AA	34.16	2h 50m 57s
Reduced AA	42.48	31m 16s
Reduced AA + Int. Opt.	30.03	24m 01s

Table 5.4: Statistics for the hypertexture based on cellular texture noise functions.

of interval arithmetic (IA), standard AA, reduced AA and reduced AA with interval optimisation is compared. The average number of function evaluations per ray indicates how often an interval extension $G(T)$ had to be computed as part of the ray casting algorithm. This statistic is a measure of the accuracy of each particular range estimation technique. A more accurate technique causes the ray casting algorithm to converge to the intersection point with fewer iterations since the computed interval extensions $G(T)$ will be closer to the true variation of the function g inside any interval T.

As expected, the IA intersection algorithm needs a large number of function evaluations due to the excessive conservativeness of IA estimates. Interval arithmetic, however, compensates for this lack of accuracy by being quite fast, which makes it competitive with some of the more advanced algorithms. Straightforward replacement of the IA operations with AA equivalents leads to a more inefficient algorithm, due to the need to compute sequences of error symbol coefficients that grow progressively larger. Nevertheless, standard AA is able to reduce the average number of function evaluations, which shows that AA does have the potential to optimise ray-surface intersection algorithms, if only it can be implemented in a more efficient manner.

The better performance of IA over standard AA for the evaluation of procedural noise functions was acknowledged implicitly by Heidrich et al. [1998]. In their work, IA was used for computing the interval estimates of a gradient noise function. These interval estimates were then converted into AA form for use in the rest of the application. No reason is given for preferring IA over AA when computing a gradient noise function but it is symptomatic that such a decision was taken in a paper whose purpose was to propose AA as a better alternative to IA.

Efficiency with AA is obtained in the reduced AA representation. As predicted in Section 5.4.2, no accuracy is lost by the use of reduced AA for the gradient noise function and the sparse

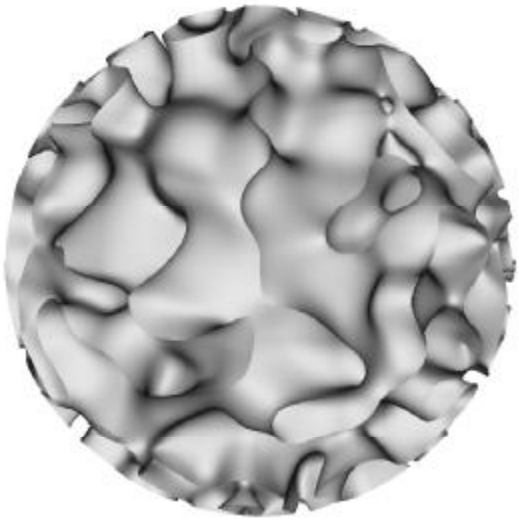


Figure 5.10: A hypertexture that cannot be rendered with sphere tracing.

convolution noise function. There is a loss of accuracy in the case of the cellular texture noise, which is compensated by its increased computation speed so that, overall, reduced AA performs much better than standard AA for all three procedural noise functions. The final improvement comes from optimising the size of the intervals, as explained in Section 5.4.1. Reduced AA combined with interval optimisation gives the lowest rendering statistics of all interval estimation techniques.

Figure 5.10 shows a hypertexture rendered with the reduced AA ray caster, which could not have been rendered with sphere tracing. The expression for this hypertexture is:

$$f(f_0(\mathbf{x}), \mathbf{x}) = f_0(\mathbf{x}) + 0.3\sqrt{|f_N(6\mathbf{x})|}. \tag{5.29}$$

The presence of the square root causes the implicit function f not to be Lipschitz continuous, preventing the use of the sphere tracing algorithm. The square root of the modulus of the procedural noise function is responsible for the very tall and thin ridge lines that are seen on the surface, always at a constant height above the sphere. The surface gradient vector grows without limits as one gets progressively closer to one of the ridges and that is why it is not possible to find a finite Lipschitz bound that will be valid throughout the surface[6]. Implementing the square root with reduced AA, on the other hand, is easily done. Stolfi and de Figueiredo [1997] give a standard affine arithmetic implementation of the square root. The same implementation also applies to reduced affine arithmetic by considering quantities that have only two error symbols.

[6]Although the surface gradient gets infinitely large as one gets closer to one of the ridge lines, the normalisation of the gradient still leads to a well-behaved unit length normal vector. The normal only becomes undefined exactly over the ridges.

	Spec.Sph.T	RAA+Int.Opt.
Gradient Noise	1m 00s	5m 04s
Sparse Convolution Noise	9m 37s	7m 58s
Cellular Texture Noise	8m 45s	24m 01s

Table 5.5: Timing comparison between the speculative sphere tracing algorithm and the ray casting algorithm with reduced AA and interval optimisation.

5.5 Summary

Sphere tracing can often outperform ray casting methods that rely on range estimation techniques when rendering general implicit surfaces. Table 5.5 compares the rendering times of the three hypertextures tested in this chapter with speculative sphere tracing and with ray casting with reduced affine arithmetic and interval optimisation. Except in the case of sparse convolution noise, sphere tracing has the best times. Sphere tracing, however, can only be applied to surfaces that are Lipschitz continuous and it also requires that the Lipschitz constant for the surface, or at least a Lipschitz bound, be known in advance. The requirement of Lipschitz continuity is not overly demanding as most approximations of natural stochastic shapes based on procedural noise functions obey this type of continuity. The Lipschitz constants for gradient noise and cellular texture noise are easily determined. In the case of sparse convolution noise, the Lipschitz constant is overly pessimistic as it is determined from a worst case scenario that is unlikely to ever occur in practice. A value for λ lower than the Lipschitz constant of sparse convolution noise is used, which makes the algorithm efficient but does not have a theoretical guarantee of always finding the correct intersection point. A second drawback of sphere tracing is that it cannot be used to find multiple roots along a ray. This eliminates the use of sphere tracing for the topology correction algorithm of Chapter 8. It is still the method of choice when rendering terrains that were modelled as displacement maps (Section 2.4.3) since these do not have any overhangs, making topology correction unnecessary.

Ray casting implicit fractal surfaces with affine arithmetic becomes efficient only with the introduction of a reduced representation for uncertain quantities. The representation of an uncertain quantity with reduced affine arithmetic uses a maximum of two error symbols. It has been shown that without this reduced representation affine arithmetic would not be able to compete against a simpler interval arithmetic representation. By maintaining only the correlation related to the uncertainty in the position of the root along the ray, reduced affine arithmetic can achieve the same results as standard affine arithmetic while being more efficient. Ray casting with reduced affine arithmetic has recently been ported to the GPU [Knoll et al., 2009]. A reduced AA quantity can easily be encoded as a `float3` variable in the Cg language [Fernando and Kilgard, 2003]. Implicit surfaces can then be rendered in real time provided that their implicit functions can be implemented in Cg.

As predicted in Section 5.3.3 and subsequently confirmed in Section 5.4.3, the application of reduced AA to cellular texture functions incurs a loss of accuracy. This is ultimately due to the destruction of important correlation information through the condensation of error symbols. The loss of accuracy, however, is compensated for by the greatly increased efficiency that comes from dealing with only two error symbols for each AA quantity, with a rendering time

that drops from almost 3 hours with standard AA to only 24 minutes with reduced AA. It is possible, however, that for some other procedural noise functions, with kernels that have not been tested, the loss of accuracy may be more significant. In such a case, standard AA can be used for the calculation of the some specific kernel and a switch to reduced AA can be done for the remainder of the calculations in the unified noise model (4.1).

Standard and reduced AA quantities can easily be interchanged. Any reduced AA quantity is also a valid standard AA quantity that happens to have only two error symbols. The second error symbol e_2 has to be given a new and unique index number, after conversion to standard AA, to express the fact that it is not shared with any other standard AA quantities. A standard AA quantity can be transformed to a reduced AA quantity through condensation. For ray casting purposes, it is required that both AA representations agree that the common error symbol e_1 is used to express uncertainty relative to the position of the root along the ray. This is so that the interval optimisation procedure of Section 5.4.1 can be properly implemented. In a situation where several procedural noise functions are used, the majority of the noise functions would be entirely computed with reduced AA while only the more problematic ones would use standard AA internally to compute their kernel functions.

The two ray casting techniques presented in this chapter allow the efficient evaluation of the surface visibility for any point on the image plane by casting a ray through that point into the scene. The execution of a single ray casting operation for every image pixel is possible but leads to aliasing artifacts. The next chapter discusses how ray casting can be used as part of an anti-aliasing technique for rendering images of implicit surfaces.

CHAPTER 6

Anti-aliasing and Motion Blur

ANTI-ALIASING is the process of removing unwanted frequencies from a signal due to its discretisation into a regularly spaced set of samples. In the context of computer graphics, the need for anti-aliasing arises from the rendering of images as two dimensional arrays of samples. An image is a two-dimensional field $I(\mathbf{x}) : \mathbf{x} \in \mathbb{R}^2$ of luminous intensities, which needs to be sampled onto an array of pixels with coordinates:

$$\mathbf{x}_{i,j} = \Delta_x(i, j) \quad \text{with} \quad i, j \in \mathbb{Z}. \tag{6.1}$$

It is assumed for simplicity that the image pixels are square in shape with a size of $\Delta_x \times \Delta_x$. The fundamental result from information theory that concerns the representation of images with a discrete set of pixels is the *Nyquist-Shannon sampling theorem* [Jain, 1989]. This theorem, in its two-dimensional form, states that given an image $I(\mathbf{x})$, whose spectrum $\hat{I}(\mathbf{k})$ is completely contained inside the square wavenumber domain $D = [-K, +K] \times [-K, +K]$ for some $K > 0$, can be represented without any loss of information by a set of equi-spaced pixels placed at $\mathbf{x}_{i,j}$ *provided that*:

$$\Delta_x < 1/(2K). \tag{6.2}$$

A sampling distance of Δ_x corresponds to a wavenumber $k_\Delta = 1/\Delta_x$. This is the equivalent of the sampling frequency in time-varying systems and is here expressed as a spatial sampling rate instead. Substituting k_Δ in the previous equation and rearranging, one obtains:

$$K < k_\Delta/2. \tag{6.3}$$

Equation (6.3) states that the maximum wavenumber K present in the image spectrum must be less than half of the sampling wavenumber k_Δ for no information to be lost due to sampling. The sampling condition (6.3) can be seen either as a constraint on the image or on the sampling rate. If the sampling rate is given then only images with spectral content below K can be successfully discretised. If an arbitrary image is given instead then the sampling rate, and hence the number of pixels, must be chosen such that (6.3) is verified.

When all the conditions for the sampling theorem are met, it becomes possible to recover exactly the image $I(\mathbf{x})$ given the set of pixels $I(\mathbf{x}_{i,j})$. This is done with the help of a low-pass reconstruction filter with impulse response $g(\mathbf{x})$. The reconstruction filter must be an ideal low-pass filter whose bandwidth matches the maximum wavenumber K in the domain D of the image spectrum. The transfer function of the filter $\hat{g}(\mathbf{k}) = \mathcal{F}\{g(\mathbf{x})\}$ must then verify $\hat{g}(\mathbf{k}) = 1$ for $\mathbf{k} \in D$ and $\hat{g}(\mathbf{k}) = 0$ otherwise. Figure 6.1 illustrates the sequence of operations involved in

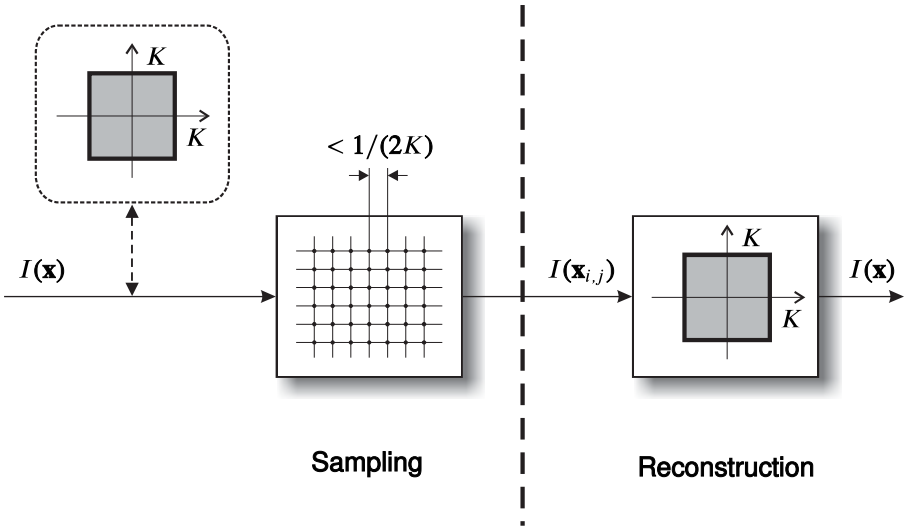


Figure 6.1: Illustration of the image sampling theorem under perfect conditions. The input image is bandlimited inside a square of length $2K$. The sampler has an inter-pixel distance of less than $1/(2K)$. The reconstruction filter is an ideal low-pass filter with a cutoff wavenumber K.

first sampling an image and then filtering it with a reconstruction filter. In computer graphics, the first part (Sampling) corresponds to the discrete process of computing the image intensities over a set of pixels in the image plane, according to some rendering algorithm. The second part (Reconstruction) corresponds to the process of creating a continuous image representation for viewing. Very often this consists in the visualisation of the discrete pixel array on a CRT monitor, in which case the reconstruction filter has the impulse response of the CRT phosphors. If all conditions are verified, the image seen on the screen will be the image that was intended for visualisation.

The sampling theorem describes an idealised situation where a continuous image can be recovered exactly from its discrete set of pixels. In practice, three common situations occur that invalidate the theorem:

- Most images of interest have infinite bandwidth. It is not possible to find some wavenumber K such that the frequency content of the image beyond K is null.
- Ideal low-pass filters do not exist. Low-pass filter transfer functions have a smooth fall-off from $\hat{g}(\mathbf{0}) = 1$ at the origin and converge asymptotically to $\hat{g}(+\infty) = 0$.
- The image is not only sampled but also quantised. Each pixel $I(\mathbf{x}_{i,j})$ is converted from a real number to an integer with a finite number of bits. This is an irreversible operation that destroys information.

Any of these situations implies that exact reconstruction of the image is impossible. As a consequence, the spectrum of the reconstructed image has some spurious frequencies that were not present in the original image. These frequencies are called *aliases* and introduce undesirable effects such as the common staircasing patterns along the silhouettes of rendered objects or the

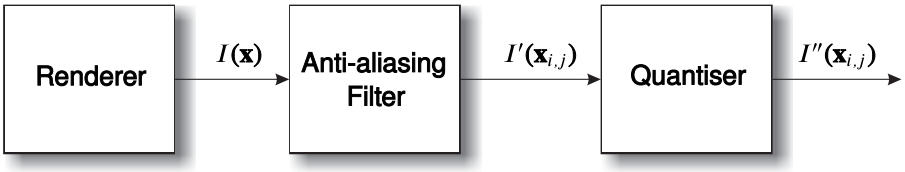


Figure 6.2: The conceptual image generation pipeline. An image renderer feeds a continuous image $I(\mathbf{x})$ into an anti-aliasing filter that outputs low-pass filtered pixels $I'(\mathbf{x}_{i,j})$. These are then quantised into a given number of bits to generate a digital image.

Moiré patterns seen on regular textures that are too small compared with the image resolution. An explanation of how aliases contaminate the spectrum of an image can be found in Blinn [1989b] together with graphs illustrating all the mechanisms as they occur in image space and wavenumber space.

Even though exact reconstruction of images is impossible in practice, the sampling theorem still forms the foundation for all image anti-aliasing algorithms. The idea is to introduce an additional low-pass image filter to reduce the aliasing artifacts at the end of the image rendering algorithm and before sending the image to the viewing device. Figure 6.2 shows the position of this anti-aliasing filter in relation to the other components in the image generation pipeline. The anti-aliasing filter, together with the image rendering algorithm and the quantiser, form the first part of the pipeline. The reconstruction filter of the viewing device, not shown in the drawing, is largely outside the control of the rendering application and forms the second part of the pipeline[1]. The anti-aliasing filter, with impulse response $h(\mathbf{x})$ and transfer function $\hat{h}(\mathbf{k})$, introduces a low-pass pre-filtering prior to the final low-pass filtering performed by the viewing device, and attempts to verify the sampling theorem as much as possible. Because image rendering is a synthesis process that maps three-dimensional geometric information into two-dimensional light intensity information, it can conceptually generate a continuous image $I(\mathbf{x})$. The anti-aliasing filter then converts this continuous image field into a discrete set of filtered pixels $I'(\mathbf{x}_{i,j})$. In rendering applications where no anti-aliasing is performed, an implicit filter exists that simply executes the assignment $I'(\mathbf{x}_{i,j}) = I(\mathbf{x}_{i,j})$. More clever anti-aliasing filters perform a low-pass operation where the image intensity $I(\mathbf{x})$ in a neighbourhood of some pixel $\mathbf{x}_{i,j}$ is averaged in order to form the filtered pixel $I'(\mathbf{x}_{i,j})$.

The concept of anti-aliasing extends naturally to time-varying images $I(\mathbf{x}, t)$ in which case it is known as *motion blur*. Computer generated animated sequences introduce a sampling $t_k = \Delta_t k$ in time with $k \in \mathbb{Z}$. Aliased frequencies in the time domain can translate into objectionable flickering or popping effects during the playback of an animation. Anti-aliasing in space and time is done with a three-dimensional low-pass filter with impulse response $h(\mathbf{x}, t)$ and transfer function $\hat{h}(\mathbf{k}, \omega)$. Usually, the space-time filter is separated into the product of two filters in space and in time, i.e. $h(\mathbf{x}, t) = h_1(\mathbf{x})h_2(t)$. This allows independent control of the spatial and temporal filtering characteristics by the choice of the filter functions h_1 and h_2.

Anti-aliasing and motion blur are an essential part of a rendering application for stochastic surfaces, especially when those surfaces have fractal characteristics. A fractal surface has many

[1] A common attempt at accounting for the characteristics of the display device as part of the rendering application is *gamma correction* [Blinn, 1989a]. Gamma correction is a mapping of the image intensities that approximately cancels the non-linearity in the intensity response of the display device.

small scale details, which generate high wavenumber content in the image plane. These high wavenumbers can easily violate condition (6.3) given a sampling wavenumber k_Δ and, if not correctly treated with an anti-aliasing filter, can reappear as aliases. As explained in Chapter 5, the rendering of implicit stochastic surfaces is performed with ray casting. This makes the computation of an image intensity $I(\mathbf{x})$ at any arbitrary point $\mathbf{x}$ particularly easy for the purpose of computing the filtered image pixels $I'(\mathbf{x}_{i,j})$. Section 6.1 explains the fundamental methods that have been developed for anti-aliasing image rendering algorithms. Section 6.2 presents the anti-aliasing method that was developed for this thesis. It can also be found in [Gamito and Maddock, 2006a]. Although this method was developed with an application to implicit stochastic surfaces in mind, it is general enough to be applied as part of any ray tracing algorithm. The same method is extended to perform motion blur in Section 6.3. Section 6.4 shows results and Section 6.5 then gives a summary of this chapter.

6.1 Anti-aliasing in Computer Graphics

Anti-aliasing techniques are usually related to the rendering algorithms for which they are employed. Scanline rendering algorithms impose a rigid sequence for the rendering of pixels and this, in turn, places some constraints on the type of anti-aliasing techniques that can be used. Ray casting or ray tracing algorithms, on the other hand, can compute the light intensity at any arbitrary point on the image plane and this allows more sophisticated anti-aliasing techniques to be implemented.

One basic anti-aliasing technique that can be used for both scanline rendering and ray tracing algorithms is *supersampling*. Supersampling simply considers a virtual image with a resolution higher than the true resolution of the image that is to be generated. With supersampling, one pixel in the image is subdivided into $n \times n$ subpixels, given some supersampling factor n. Any rendering algorithm can then be applied over the increased resolution image that consists of the collection of all subpixels. The final intensity of a pixel is obtained by performing a weighted average of the subpixel intensities that surround it, using a discrete filter mask such as a Bartlett window [Crow, 1981]. The idea behind supersampling is to increase the sampling wavenumber, which now becomes $k_\Delta = n/\Delta$. With a higher k_Δ, it becomes possible to increase the maximum wavenumber K in the spectrum of an image without violating the sampling condition (6.3). Supersampling, although trivial, is not a popular anti-aliasing technique because the rendering complexity grows with $O(n^2)$ while the allowed image bandwidth only grows with $O(n)$. One other problem is that computer graphics images are never truly bandwidth limited. This has the consequence that supersampling makes aliasing artifacts ever smaller with increasing n but cannot eliminate them completely.

Anti-aliasing techniques that have been developed for scanline rendering algorithms consider every square pixel in the image as a box filter and average all the image intensities that fall within the pixel. No information from outside the pixel square is used when computing the final pixel intensity. Catmull [1978] developed a scanline rendering algorithm where polygons are clipped against the square boundaries of every pixel. Algorithms exist for clipping all scene polygons in an incremental fashion by taking advantage of the fact that rendering is done in raster scan order. The contribution of a polygon to a pixel is weighted by the visible area of the clipped polygon fragment divided by the total area of the pixel. Clipping the scene polygons

against every pixel is an expensive procedure, even when done incrementally, and this area weighting mechanism was made more efficient, although less accurate, by Carpenter [1984] with the introduction of the *A-buffer*. The A-buffer is an extension of the Z-buffer [Catmull, 1974]. It stores partial visibility in the form of an alpha mask together with distance in a $m \times n$ subpixel buffer for every pixel. Standard image compositing operations are then applied so that anti-aliasing is expressed in the form of a transparency alpha mask along the edges of the polygons [Porter and Duff, 1984].

In the case of the ray casting or ray tracing algorithms, anti-aliasing can be done by averaging several samples of the image intensity $I(\mathbf{x})$ in order to form the anti-aliased pixels $I'(\mathbf{x}_{i,j})$. Any pattern of samples can be used *a priori* and, unlike the scanline rendering algorithms, there is no restriction to taking samples only inside the square shape of each pixel at $\mathbf{x}_{i,j}$. A new approach for computer graphics was presented by Cook [1986] and was called *distributed ray tracing*. In distributed ray tracing, intensities are calculated over samples placed randomly around the centre of every pixel and not over regularly spaced samples as is done when supersampling. The contribution of the intensity over each sample is then weighted by the anti-aliasing filter placed at the centre of the pixel. One alternative that does not require weighting is to perform *importance sampling* around each pixel based on the filter function. This approach is explained in more detail in Section 6.2.1 as it also forms the basis for the anti-aliasing method developed for this thesis. The numerical techniques behind distributed ray tracing involve stochastic sampling and Monte Carlo integration techniques. Cook showed that these techniques can be used not only to perform anti-aliasing but also to generate several other effects such as soft shadows, glossy reflections and translucencies, motion blur and depth of field.

The advantage of stochastic sampling techniques over the simpler supersampling technique is that any aliasing artifacts are converted into noise. Since aliases can never be completely removed from an image, given the previously explained limitations of the sampling theorem, their conversion into low power noise constitutes a better strategy for practical anti-aliasing applications. The stochastic placement of samples mimics the placement of the photoreceptors on the human retina [Yellot, 1983]. This random distribution of photoreceptors trades coherent aliasing artifacts for noise, most of which is later masked out by the perception mechanisms in the brain. Stochastic sampling becomes a very effective anti-aliasing technique by taking advantage of this characteristic in the human visual system.

6.2 Anti-aliasing with Stratified B-spline Filters

A method is presented for performing anti-aliasing on ray traced images with stochastic sample placement and a low-pass filter chosen from a family of B-spline basis functions. A similar approach was presented by Stark et al. [2005]. Stark et al. presented the solution for B-spline filters of up to degree four (cubic B-splines) whereas the proposed method can work with any filter degree. The evaluation of the basis functions uses an elegant recursive formulation that is valid for any filter degree. This method of evaluating B-spline basis functions was already considered by Thévenaz and Unser [2001] for the generation of implicit surfaces from gridded data points. By choosing the degree of the B-spline filter, one can have different filter behaviours for anti-aliasing, starting with the box filter and proceeding through the tent filter,

the cubic filter, and beyond. It is this flexibility of B-spline functions that makes them very popular as anti-aliasing filters for Computer Graphics.

6.2.1 Stratified Monte Carlo Anti-aliasing

The goal is to compute the filtered luminous intensity $I'(\mathbf{x}_{i,j})$ for the pixels placed at the positions $\mathbf{x}_{i,j}$ in image space, given the original image $I(\mathbf{x})$, in such a way that high frequencies are disguised as noise and do not cause aliasing. Removal of high frequencies from a signal is possible by convolving it with some appropriate low-pass filter $h(\mathbf{x})$. The convolution operator for the pixel at $\mathbf{x}_{i,j}$ takes the form:

$$I'(\mathbf{x}_{i,j}) = \int I(\mathbf{x})h(\mathbf{x} - \mathbf{x}_{i,j})\mathbf{dx}, \tag{6.4}$$

where the filter is displaced so as to be located over the centre of the pixel with coordinates given by (6.1). Many different numerical techniques can be used to approximate the integral above. The one that is most useful in this context is *Monte Carlo integration* [Glassner, 1995]. Monte Carlo integration approximates the integral (6.4) by evaluating $I(\mathbf{x})$ over $N \times N$ randomly placed samples $\mathbf{u}_i$:

$$I'(\mathbf{x}_{i,j}) \approx \frac{1}{N^2} \sum_{u,v} I(\mathbf{x}_{u,v}). \tag{6.5}$$

The low-pass filter appears implicitly in the above equation. The $N \times N$ positions $\mathbf{x}_{u,v}$ are taken to be the outcome of a two-dimensional random variable whose probability density function (PDF) is $h(\mathbf{x})$. One must, therefore, deal with the issue of how to randomly place samples in image space according to some arbitrary probability density $h(\mathbf{x})$.

First, the filter function $h(\mathbf{x})$ itself must be considered. It must obey certain constraints if it is to be a valid PDF. The most obvious constraint is that $h(\mathbf{x})$ must be strictly positive, since it is not meaningful to consider negative probabilities. Secondly, the integral $\int_\infty h(\mathbf{x})\mathbf{dx}$ must be unity. This means that there is a probability of one that a sample will be located somewhere in image space. With a valid PDF, it is possible to obtain another important measure for a random process: the cumulative density function (CDF). If $h(\mathbf{x})$ is a PDF, then its CDF is given by:

$$H(\mathbf{x}) = \int_{-\infty}^{\mathbf{x}} h(\mathbf{t})\,\mathbf{dt}. \tag{6.6}$$

The CDF $H(\mathbf{x})$ gives the probability that the random variable will take values below $\mathbf{x}$. In the case of two-dimensional random variables, it gives the probability that an image sample will be inside the rectangle that has a lower left vertex at minus infinity, in both horizontal and vertical coordinates, and an upper right vertex at point $\mathbf{x}$. If $h(\mathbf{x})$ is a proper PDF, then it occurs that $H(+\infty) = 1$, meaning that every sample must be located somewhere on the image plane.

To numerically compute a sample from a random process, knowing its CDF, the method of *function inversion* is used [Press et al., 1992]. One begins by generating a uniform random variable $\mathbf{y}$ that takes values in the unit rectangle $[0, 1) \times [0, 1)$. The sample having the desired CDF, $H(\mathbf{x})$, and PDF, $h(\mathbf{x})$, is now obtained by finding the solution to the equation:

$$\mathbf{y} = H(\mathbf{x}). \tag{6.7}$$

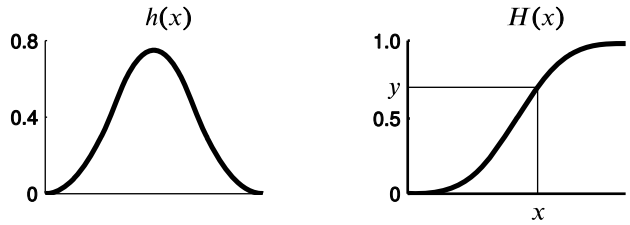


Figure 6.3: A probability density function (left) and its corresponding cumulative density function (right). To obtain a sample x with this PDF, sample uniformly with $y \in [0, 1)$ and obtain x by inverting the CDF.

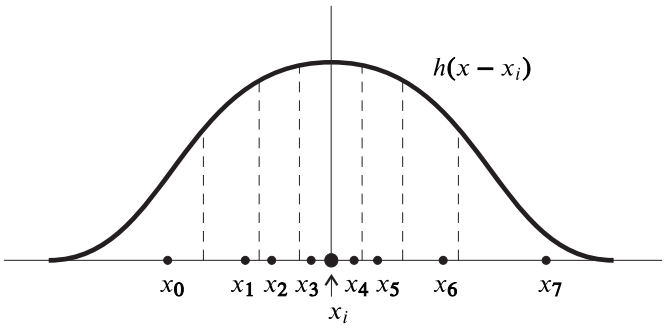


Figure 6.4: Importance sampling of a PDF in one dimension. The domain is split into sectors with equal area under the PDF. There is one random sample per sector. The dotted lines are the boundaries between sectors.

Figure 6.3 exemplifies this process for a typical low-pass filter in one dimension. It is possible to see that $H(\mathbf{x})$ is a monotonically increasing function from 0 to 1. This is true of any CDF and it is a property that will be used to advantage in Section 6.2.3.

To distribute sample points around the point $\mathbf{x}_{i,j}$ for Monte Carlo integration, a regular grid of samples is created, not in image space, but in the unit square space of the $\mathbf{y}$ variable. These samples become stratified by the addition of a random component ξ that breaks up the regularity of the grid:

$$\mathbf{y}_{u,v} = \Delta_y(u, v) + \xi \quad \text{with} \quad i, j = 0, \ldots, N-1. \tag{6.8}$$

If N samples are used, along both the horizontal and vertical coordinates, then $\Delta_y = 1/N$ and the random vector ξ has components taking uniform random values in the interval $[0, 1/N)$. A stratified distribution of samples $\mathbf{x}_{u,v}$ is thus obtained for use in the Monte Carlo approximation (6.5) to the anti-aliasing integral. The samples obey a PDF equal to the filter $h(\mathbf{x})$ by having them be the solution of $\mathbf{y}_{u,v} = H(\mathbf{x}_{u,v} - \mathbf{x}_{i,j})$ for all u, v and given $\mathbf{x}_{i,j}$.

Figure 6.4 shows a one-dimensional example where $N = 8$ random samples x_u are placed around a pixel at position x_i. The samples were initially generated in the interval $[0, 1)$ and then placed around the pixel by inverting the CDF $H(x - x_i)$. The graph of the PDF $h(x - x_i)$ is shown in the figure, centred over the pixel. The stratification of samples in the $[0, 1)$ space leads to a partition of the PDF into sectors such that the area of all sectors is the same. The samples x_u are then randomly placed inside each one of the sectors. This sample placement strategy is known as *importance sampling*. One can see that there is a higher density of samples

in the locations where the PDF has higher values. Sampling a one-dimensional light intensity profile $I(x)$ with such a set of samples will concentrate computational effort in the places the PDF indicates are more important. There is no need, therefore, to weight the $I(x_u)$ samples with the filter function $h(x - x_i)$ because it is already taken into account when choosing the x_u.

6.2.2 A Family of Uniform B-splines

A family of B-spline basis functions $n_m(x)$, with knots placed at integer positions, can be obtained by performing consecutive convolutions with the characteristic function on the interval $[0, 1)$ [Chui, 1992]. The sequence starts off with the B-spline of degree one, which is the aforementioned characteristic function itself:

$$n_1(x) = \begin{cases} 1 & \text{if } x \in [0, 1), \\ 0 & \text{otherwise.} \end{cases} \tag{6.9}$$

The convolution of a B-spline of degree $m - 1$ with the characteristic function gives the next B-spline in the sequence:

$$n_m(x) = (n_{m-1} * n_1)(x) = \int_0^1 n_{m-1}(x - t)\, dt, \quad \text{for } m > 1. \tag{6.10}$$

Because of the consecutive convolutions, the B-spline curves become progressively smoother as m increases. From Chui [1992], the properties of the family of B-spline functions are stated that make them particularly attractive for use as low-pass filters in anti-aliasing:

$$n_m(x) > 0, \quad \text{for } 0 < x < m. \tag{6.11a}$$

$$\int_{-\infty}^{+\infty} n_m(x)\, dx = 1, \quad \text{for all } m. \tag{6.11b}$$

$$\operatorname{supp} n_m = [0, m]. \tag{6.11c}$$

$$n_m(x) = \frac{x}{m-1} n_{m-1}(x) + \frac{m-x}{m-1} n_{m-1}(x-1), \quad \text{for } m > 1. \tag{6.11d}$$

Properties (6.11a) and (6.11b) together say that any $n_m(x)$ is a valid probability density function for Monte Carlo anti-aliasing. The support of a B-spline $n_m(x)$, when defined according to (6.10), is the interval $[0, m]$. For anti-aliasing, however, it is required that random samples be centred around some desired pixel position. This can be achieved for B-spline filters by generating the samples, as explained in the previous section, and performing a simple offset of $-m/2$ along the horizontal and vertical coordinates. Property (6.11d) is the most important. It gives an algebraic relation between the B-spline of degree m and the B-spline of degree $m - 1$. With this knowledge and with knowledge of the shape of $n_1(x)$ from equation (6.9) it is possible to compute $n_m(x)$ recursively, at any point x and for any degree m.

It is known from the above that B-splines can be used as filters for anti-aliasing and that a simple recursive procedure exists to evaluate them. To study the behaviour of B-splines as low-pass filters, it is necessary to also study their spectra $\hat{n}_m(k)$, as given by the application of the Fourier transform to $n_m(x)$:

$$\hat{n}_m(k) = \mathcal{F}\{n_m(x)\} = \left(\frac{\sin \pi k}{\pi k}\right)^m e^{-i\pi k m/2}. \tag{6.12}$$

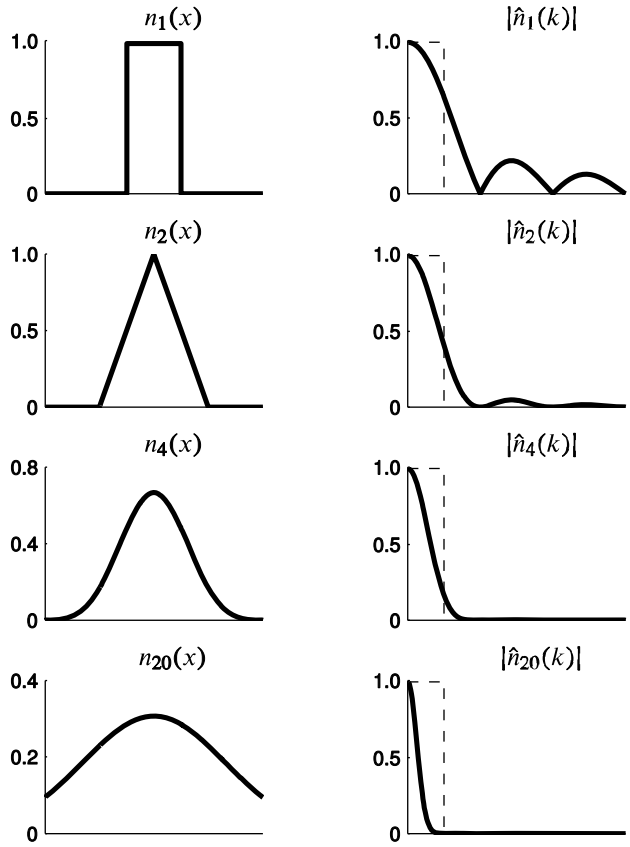


Figure 6.5: B-spline basis functions (on the left), for m equal to 1, 2, 4 and 20, and their respective spectra (on the right). The spectrum for the ideal anti-aliasing filter is superimposed as a dashed rectangle on the latter.

If, for simplicity, one admits a unit distance between pixels so that $\Delta_x = 1$, the sampling wavenumber becomes $k_\Delta = 1m^{-1}$ along the horizontal and vertical directions. The sampling theorem then says that it is necessary to filter out all wavenumbers above $0.5m^{-1}$ if no aliasing is to occur. Ideally, a perfect low-pass filter with a sharp wavenumber cutoff at $0.5m^{-1}$ should be implemented. Such an ideal filter, however, would have an infinite support and would be computationally intractable. The family of B-spline basis functions provides a sequence of approximations to this ideal low-pass filter.

Figure 6.5 shows the shape of four B-splines with degrees of 1, 2, 4 and 20, on the left, together with the modulus of their respective spectra, on the right. These spectra are symmetric about the origin, since they are the transform of real functions, and only the positive wavenumbers are shown. The spectrum of the ideal low-pass filter for anti-aliasing has been superimposed as a dashed rectangle. The basis function $n_1(x)$ gives the well-known box filter for anti-aliasing. It is quite simple to implement but it allows too many high wavenumbers to pass through, as evidenced by the significant lobes that fall outside of the spectrum for the perfect filter. The basis function $n_2(x)$ is also known as the tent or triangle filter because of its shape. It has a better

spectral behaviour than the box filter, since the lateral spectral lobes are now more attenuated. However, the trend of increasing m in order to block more of the high wavenumbers cannot continue indefinitely. For too large a value of m, the low wavenumbers also become excessively attenuated. This is exemplified by the B-spline filter of degree 20 in Figure 6.5. Visually, this distortion in the low wavenumbers translates into a blurry image, where the amount of blurring is far greater than necessary to eliminate aliasing. It is generally considered that a good compromise between blocking the high wavenumbers and not distorting the low wavenumbers is achieved with the cubic B-spline filter $n_4(x)$, also shown in Figure 6.5.

Once a particular degree for the B-spline is chosen, a two-dimensional anti-aliasing filter is made from the cartesian product of the $n_m(x)$ kernel with itself:

$$h(\mathbf{x}) \;=\; h(x_1, x_2) \;=\; n_m(x_1 + m/2)\, n_m(x_2 + m/2). \tag{6.13}$$

As previously explained, the offset by $-m/2$ properly centres the kernels around the origin so that they can later be positioned over the pixel positions at $\mathbf{x}_{i,j}$. It would be equally as simple to have different degrees for the horizontal and vertical kernels but there does not seem to be any significant advantage in doing so.

6.2.3 Stratified Anti-Aliasing with B-splines

Stratified Monte Carlo anti-aliasing with B-spline low-pass filters requires that random image samples be computed with a PDF given by $n_m(x)$. This, in turn, requires the following CDF to be known:

$$N_m(x) \;=\; \int_0^x n_m(t)\, dt. \tag{6.14}$$

Based on Chui [1992], four properties about the integral $N_m(x)$ of a B-spline basis function can be derived that are important when writing an implementation of anti-aliasing:

$$\operatorname{supp} N_m(x) \;=\; [0, +\infty[. \tag{6.15a}$$

$$N_m(x) \text{ increases from } N_m(0) = 0 \text{ to } N_m(+\infty) = 1. \tag{6.15b}$$

$$n_m(x) \;=\; N_{m-1}(x) - N_{m-1}(x-1), \quad \text{for } m > 1. \tag{6.15c}$$

$$N_m(x) \;=\; \frac{x}{m} N_{m-1}(x) + \left(1 - \frac{x}{m}\right) N_{m-1}(x-1), \quad \text{for } m > 1. \tag{6.15d}$$

Properties (6.15a) and (6.15b) are a direct consequence of (6.14), together with properties (6.11a), (6.11b) and (6.11c). These two former properties mean that $N_m(x)$ is a valid CDF. Properties (6.15c) and (6.15d) are derived in Appendix A. Property (6.15c) gives an algebraic relationship between the spline basis function and its integral at the next lower degree. Property (6.15d) presents a numerical recipe for computing any value of $N_m(x)$ in a recursive way. At the end of the recursion lies the $N_1(x)$ function, whose shape has a trivial expression:

$$N_1(x) \;=\; \begin{cases} 0 & \text{if } x < 0, \\ x & \text{if } x \in [0, 1[, \\ 1 & \text{if } x \geq 1. \end{cases} \tag{6.16}$$

The generation of random samples for Monte Carlo integration, with a B-spline acting as the PDF, now requires solving the following equation for a sample x in either horizontal or vertical screen coordinates, where y is a stratified uniform random variable in the $[0, 1)$ interval:

$$y = N_m(x). \tag{6.17}$$

The solution is immediate when $m = 1$ but, unfortunately, becomes rather involved for higher degrees. Rather than try to solve (6.17) analytically, the best approach is to use a numerical iterative method like Newton to find the zero of the auxiliary function $f(x) = N_m(x) - y$ (see [Press et al., 1992] for such a method).

Newton-Raphson is a powerful root finder since it has quadratic convergence, but some care must usually be taken before its application. The root must first be bracketed inside a suitable interval. Then, $f(x)$ must be monotonic inside that interval with a non-vanishing derivative everywhere. All these conditions are naturally met in the case of a B-spline CDF. Clearly, there is one and only one root x inside the interval between $y = 0$ and $y = 1$. The function is monotonically increasing inside that interval, as assured by property (6.15b), and the only points where the derivative vanishes are the two interval extremes. If the boundary case $y = 0 \Rightarrow x = 0$ is first cleared (because $y \in [0, 1)$, the boundary $y = 1$ need not be considered) then a series of Newton iterations can be started, with full confidence that it will converge to the solution:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{N_m(x_i) - y}{N'_m(x_i)} = x_i - \frac{N_m(x_i) - y}{n_m(x_i)}. \tag{6.18}$$

The cost of performing (6.18) is $2^m - 1$ recursive calls of $N_m(x_i)$ and $2^m - 1$ recursive calls of $n_m(x_i)$, giving a total complexity of $O(2^{m+1})$ per Newton iteration. It is possible to do better by inserting (6.15c) and (6.15d) in (6.18):

$$x_{i+1} = x_i - \frac{\frac{x_i}{m} N_{m-1}(x_i) + \left(1 - \frac{x_i}{m}\right) N_{m-1}(x_i - 1) - y}{N_{m-1}(x_i) - N_{m-1}(x_i - 1)} \quad \text{for } m > 1. \tag{6.19}$$

The cost is now $2^{m-1} - 1$ recursive calls to compute $N_{m-1}(x_i)$ and similarly for $N_{m-1}(x_i - 1)$. The complexity is $O(2^m)$, half of what it was before. The Newton iterations are started off with $x_0 = m/2$, which is at the centre of the interval (6.11c) where the random variable x is bound to lie. When the iterations finish, the solution is $x_{i+1} - m/2$.

Figure 6.6 shows an example of a sampling pattern generated by this method for the case of a cubic B-spline filter and using 10×10 samples per pixel. The boundaries between the pixels are marked with horizontal and vertical lines. The diamond shape shows the centre of the pixel for which the sampling pattern applies. The samples are preferentially located in areas where the filter has higher importance, which is to say, closer to the centre of the pixel.

6.3 Motion Blur

The anti-aliasing method can be extended to the extra dimension of time. Anti-aliasing in the time domain is commonly referred to as *motion blur*. To obtain a combined space and

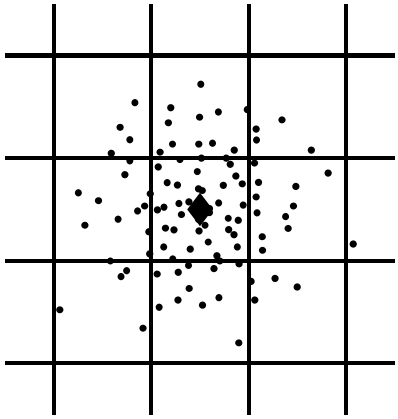


Figure 6.6: A sampling pattern for a cubic B-spline filter.

time anti-aliasing for an image intensity $I(\mathbf{x}, t)$, a three-dimensional Monte Carlo integration is performed:

$$I'(\mathbf{x}, t) \approx \frac{1}{N^2} \sum_{u,v,w} I(\mathbf{x}_{u,v}, t_w). \tag{6.20}$$

The random spatial samples $\mathbf{x}_{u,v}$ obey the PDF (6.13), as before. The random time sample t_w obeys a PDF given by a B-spline filter $n_l(t - l/2)$ in time. The degree l of the time filter need not be the same as the degree m of the spatial filter. In equation (6.20), a total of N^2 samples is still being used instead of the N^3 samples that might seem more obvious for a three-dimensional Monte Carlo integration. A table of N^2 random permutations is created before starting the anti-aliasing algorithm. Then, for each evaluation of equation (6.20), N random time samples are generated and accessed through this permutation table, based on the index (u, v) of the $\mathbf{x}_{u,v}$ samples. This technique, which was originally presented by Cook et al. [1984], keeps the cost of the integration (6.20) constant at N^2 evaluations of $I(\mathbf{x}, t)$ without introducing significant correlation between the spatial and the time domains.

The common example of motion blur generated by a camera with a finite exposure time can be easily modelled with the anti-aliasing method. Each frame in a movie runs from time t_i to time $t_i + \Delta_t$, where Δ_t is the inverse of the frame rate. The camera shutter is only open during a smaller interval of time, from t_i to $t_i + s$, where $s < \Delta_t$ is called the ***shutter speed*** (even though it is really a time quantity). During the final portion of the frame, the shutter must be closed while the camera's motor advances the film into the next position. The shutter, therefore, behaves as a box filter with a shape given by $n_1(t/s)/s$. The $1/s$ factor outside n_1 gives a constant unit area to the filter and ensures it is a valid PDF for any value of s.

If B-splines with $l > 1$ are used, a smoother motion will be obtained even though it will not be the same as what a real camera would record. A stronger objection to using $n_l(t - l/2)$ with $l > 1$ is that such a filter, being symmetrical about the origin, would equally weight past and future values of $I(\mathbf{x}, t)$ relative to some instant t_i. No physical imaging device could possibly have such a behaviour. It would be relatively straightforward to have $n_l(t)$ obey a principle of

causality (which basically states that any value of $n_l(t)$ for $t < 0$ must be zero) by manipulating the position of the spline knots and thus generating a new family of non-uniform B-splines. It was decided, however, not to pursue this since there is still much speculation about how the human retina processes time-varying information [Anderson and Anderson, 1993; VanRullen and Koch, 2003]. Without more research from the psychophysical sciences it is difficult to decide what time filter best matches the behaviour of our own visual system.

6.4 Results

A good test of the anti-aliasing properties of the method is shown in Figure 6.7. The images in this figure were generated by directly sampling the function:

$$I(x, y) = \frac{1}{2}\left(\sin\left(\frac{1000}{x^2 + y^2}\right) + 1\right), \tag{6.21}$$

with a resolution of 320 × 320. The origin $(x, y) = (0, 0)$ is at the lower left corner of each image. This function was chosen to highlight the problems caused by aliasing and to show how B-spline filtering can solve them. It has a frequency content that increases without bounds for pixels progressively closer to the origin. The top image in Figure 6.7 does not use anti-aliasing and takes only one sample at the centre of each pixel. In this image, the outer rings are correctly rendered, the middle rings show Moiré patterns caused by aliasing and the inner rings have become totally masked by severe white noise. The image in the middle of Figure 6.7 uses a box filter and shows how the noise of the inner rings has been replaced by a constant grey value. The middle rings still show some Moiré patterns. The image at the bottom of Figure 6.7 uses a cubic B-spline filter. The uniform gray area is larger than in the previous image and no Moiré patterns remain.

The timings for the generation of anti-aliased samples is shown in Table 6.1, where the method presented in this chapter is compared against the analytic and the numerical versions of the anti-aliasing method by Stark et al. [2005]. The surface used for the testing is the gradient noise hypertexture shown on the left of Figure 5.6 and the rendering algorithm used is sphere tracing (Section 5.2). The image has a resolution of 300×300 pixels and uses cubic B-spline filters with 10×10 samples per pixel. Each line in Table 6.1 shows, for each anti-aliasing method, the time spent in generating the samples for the Monte Carlo integration and the corresponding percentage of the total rendering time. Clearly, the method presented in this chapter is slower than those of Stark et al. since the evaluation of $N_4(x)$ involves several recursive function calls. The time required to generate the samples, however, remains insignificant when compared to the time required to solve the ray casting problem. In the unlikely event that sample generation becomes a bottleneck for some particular rendering task, one can always pre-compute $N_m(x)$ into a lookup table. Given that $N_m(x)$ changes quite smoothly, this should not introduce significant numerical errors.

Figure 6.8 shows a small area of the implicit surface and illustrates the effect of changing the number of samples per pixel. The background has been switched to black, relative to the rendering of Figure 5.6, to make the contours of the surface easier to distinguish. From left to right the sequence shows no anti-aliasing, and anti-aliasing with a cubic B-spline filter with N^2 equal to 4, 9 and 100 samples per pixel. Performing anti-aliasing by taking only four random

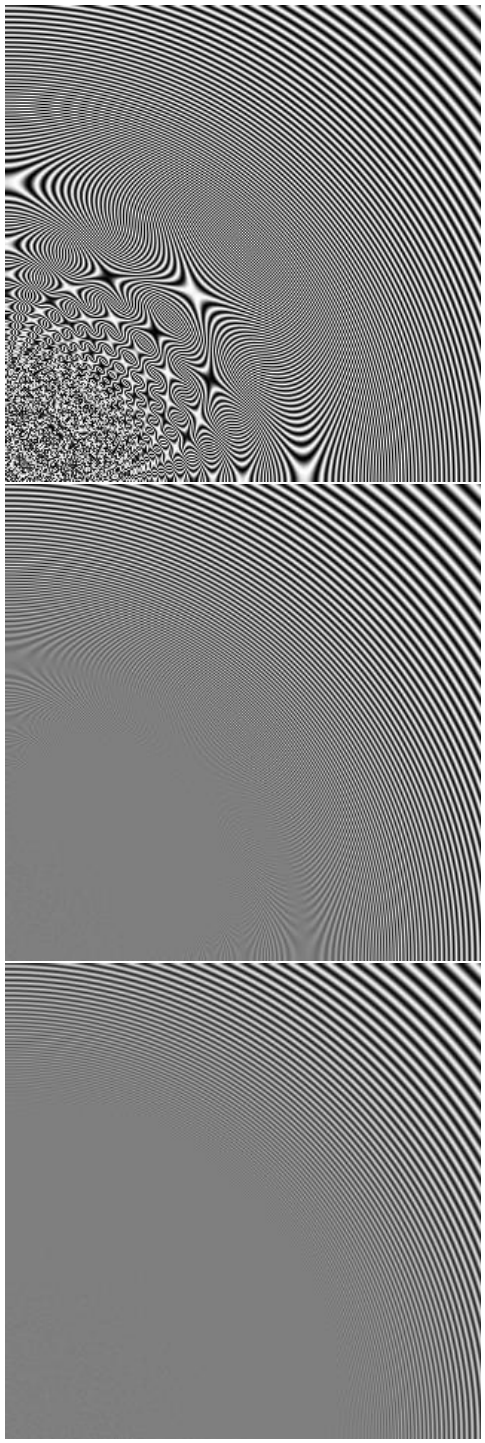


Figure 6.7: Anti-aliasing test pattern. From top to bottom: no anti-aliasing, anti-aliasing with a box filter and anti-aliasing with a cubic B-spline filter.

Method	Time	%
Gamito and Maddock	24.61s	2.81
Stark et al. (analytical)	8.82s	1.02
Stark et al. (numerical)	6.96s	0.81

Table 6.1: Time and percentage of total rendering time for the generation of samples in Monte Carlo anti-aliasing with cubic B-splines.

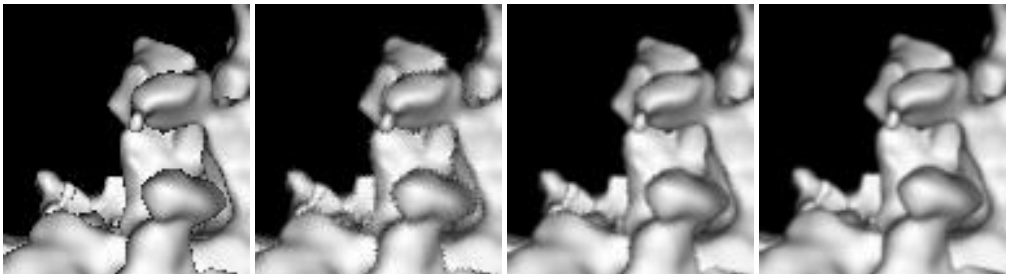


Figure 6.8: From left to right: no anti-aliasing, anti-aliasing with a cubic B-spline filter with N equal to 2, 3 and 10 samples along each coordinate direction.

Figure 6.9: From left to right: no anti-aliasing, anti-aliasing with a B-spline filter of degree 1, 4, and 15. $N = 10$ samples were used along each coordinate direction for the anti-aliased images.

.

samples (two samples along each coordinate direction) is not very effective but it does show that coherent aliasing artifacts are converted into noise by the stratification of sample points. When N^2 increases to 9 and then to 100, results become progressively better. This increase in sampling density, however, must be balanced against the increased computational cost of tracing the additional rays for those samples. It must also be considered that after a sufficient number of samples per pixel has been computed, computing additional samples does not bring any further improvement. This result occurs when the Monte Carlo integral has converged successfully. For the case of Figure 6.8, a density of 100 samples per pixel is enough to achieve convergence of the integral although other images with a higher frequency content (caused by the use of surface textures, for example) may require higher densities.

Figure 6.9 shows another small area of the surface. This time, the number of samples per pixel was kept constant and the degree of the B-spline filter was increased. From left to right, the

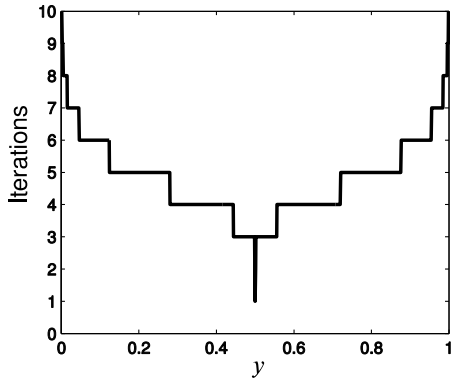


Figure 6.10: Number of Newton iterations required to invert $y = N_4(x)$ for $y \in [0, 1)$.

sequence shows no anti-aliasing and anti-aliasing with a B-spline filter of degree m equal to 1, 4 and 15. In the cases where anti-aliasing was used, $N^2 = 100$ samples per pixel were computed. It can be seen that as the filter degree is increased the image becomes progressively blurred. This is because, for higher degrees, the filter has a larger area of support and therefore spreads the influence of the bright pixels farther into the dark areas. The optimal degree is $m = 4$, as explained in Section 6.2.2.

The number of Newton iterations necessary to invert $y = N_4(x)$ is shown in Figure 6.10. The iterations were stopped once the condition $|x_{i+1} - x_i| < 10^{-8}$ became true. If $y = 0.5$ then $x = x_0 = m/2$ and the algorithm converges after exactly one iteration. If, on the other hand, y moves away towards one of the extremities of the $[0, 1[$ interval, a larger number of iterations will be required for x_i to converge to the solution. Nevertheless, the number of iterations always remains small due to the quadratic convergence properties of Newton root finding. From the results in Figure 6.10, one can say that an average of 4.82 iterations is necessary to invert the CDF.

Figure 6.11 shows, on the left, two frames from an animation of a rotating wheel at 25 frames per second ($\Delta t = 0.04$) as seen by a camera with a shutter speed $s = 0.033$. This illustrates the famous *wagon wheel illusion* [Purves et al., 1996]. In the lower left image, the spokes are turning counter-clockwise at high speed but the blurred features in the image are slowly turning clockwise. If the box filter, which models the behaviour of the camera shutter, is replaced by a cubic B-spline filter, the images on the right of Figure 6.11 result. In the lower right image, the spokes are no longer discernible and a uniform circular blur results, showing no indication of wheel rotation. There is some controversy as to whether or not this is what the human eye would see under constant lighting conditions since it has been reported that the wagon wheel illusion is sometimes discernible under these circumstances [VanRullen et al., 2005].

6.5 Summary

An approach to anti-alias ray traced images using B-spline basis functions has been developed that improves upon the approach by Stark et al. [2005]. The proposed method is simpler to

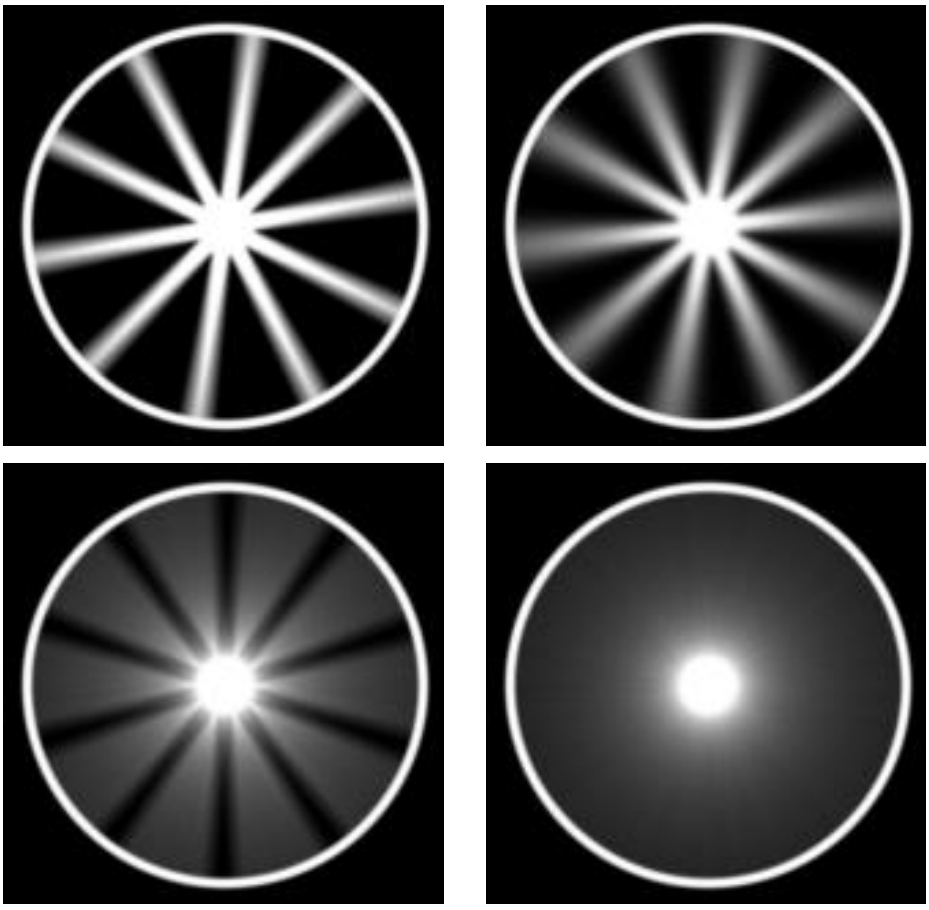


Figure 6.11: A rotating wheel as seen by a camera (on the left) and filtered with a cubic B-spline (on the right). The images at the top show the wheel rotating slowly and the images on the bottom show the wheel rotating at high speed.

implement and can be used to generate B-spline filters of any degree. The application of this anti-aliasing method in both the space and the time domains was demonstrated.

The proposed approach can generate any B-spline filter through a simple and elegant recursive procedure. This recursive procedure means it is not necessary to deal with the actual piecewise polynomials that describe the shape of the B-splines. Explicitly working with the polynomial representation for some B-spline $n_m(x)$ can become quite an involved procedure, even for moderate values of m. The cost of recursively evaluating $n_m(x)$ grows geometrically with m. This is not, however, a serious constraint for the method since filters of too high a degree introduce excessive blurring and should be avoided. A filter of degree $m = 4$ (a cubic B-spline filter) provides the best compromise between blocking the undesirable high frequencies and not distorting the low frequencies, thereby introducing minimal blurring.

Generation of random samples for Monte Carlo anti-aliasing, having $n_m(x)$ as their probability density function, requires inversion of $y = N_m(x)$ where $N_m(x)$ is the cumulative density

function associated with the B-spline filter. This can be accomplished numerically with Newton root finding in a way that is always guaranteed to converge. Experiments have shown that the number of required iterations is always less than or equal to ten.

The method was developed to perform anti-aliasing in space but it extends to anti-aliasing in time in a simple way. The motion blur behaviour of a camera shutter was shown to be equivalent to a B-spline of degree $m = 1$. B-spline filters of higher degree are non-causal. This is not a problem for pre-scripted animations, whose behaviour in time is known *a priori*, but it does raise the objection of physical implausibility since a filtered intensity value will depend equally on samples from the past and from the future.

Performing anti-aliasing on computer generated images is an expensive proposition. By using N^2 samples per pixel one is doing the same amount of work as though a virtual image with a resolution N^2 higher than the actual image was being rendered. The present method does not take into account local image information when performing anti-aliasing. In the example presented in the previous section, it is quite wasteful to be casting N^2 rays for a pixel located on the inside of the implicit surface and far from any edge. A single ray would have sufficed, as the non-antialiased image on the left of Figure 6.8 shows. In this image, aliasing is only evident at the edges, while the internal surface features remain smooth. Techniques for turning the present anti-aliasing method with B-splines into an adaptive procedure, similar to the work by Painter and Sloan [1989] should be further investigated. One other improvement would be to use an isotropic B-spline filter $n_m(l + m/2)$ where $l = (x_1^2 + x_2^2)^{1/2}$ instead of the current separable filter (6.13). An isotropic filter would treat all wave vectors with equal wavenumber k in the same way, unlike the current filter that introduces a slight anisotropy for horizontal and vertical wave vectors.

CHAPTER 7

Progressive Refinement Rendering

THE visualisation of hypertextured implicit surfaces can be an inefficient task when such surfaces are complex and highly detailed. Visualising a surface by first converting it to a polygon mesh may lead to an excessive polygon count. Visualising a surface by direct ray casting is often a slow procedure. In this chapter, a progressive refinement renderer for implicit surfaces is presented. The renderer first displays a low resolution estimate of what the final image is going to be and, as the computation progresses, increases the quality of this estimate at an interactive frame rate. This renderer provides a quick previewing facility that can reduce the design cycle of a new and complex implicit surface. The renderer is also capable of completing an image faster than the ray casting algorithms of Chapter 5.

The idea of progressive refinement for image rendering was first formalised in 1986 [Bergman et al., 1986]. Progressive refinement rendering has received much attention in the fields of radiosity and global illumination [Cohen et al., 1988; Farrugia and Peroche, 2004]. Progressive refinement approaches to volume rendering have also been developed [Laur and Hanrahan, 1991; Lippert and Gross, 1995]. This chapter's implicit surface renderer uses progressive refinement to visualise an increasingly better approximation to the final implicit surface. It allows the user to make quick editing decisions without having to wait for a full ray casting solution to be computed. Because the algorithm is progressive, the rendering can be terminated as soon as the user is satisfied or not with the look of the surface. The renderer can also be interactively controlled by the user through the specification of image regions. Image refinement will only occur inside a region, once the region becomes active, while the remainder of the image is kept on hold. In this way, the user can steer the application into rendering image regions that he considers to be more interesting or troublesome.

Section 7.1 presents previous solutions that have been proposed to preview implicit surfaces. Progressive refinement techniques that converge to a correct ray traced image are also presented although these are not specifically tuned to implicit surfaces and compute refinement based on image space information only. Two versions of the progressive refinement renderer have been developed. One of the versions, presented in Section 7.2, works for Lipschitz continuous surfaces and is an extension of the sphere tracing algorithm of Section 5.2 [Gamito and Maddock, 2007c]. The other version, presented in Section 7.3, works for generic implicit surfaces and extends the reduced affine arithmetic ray casting algorithm of Section 5.4 [Gamito and Maddock, 2006b, 2007b]. Section 7.4 explains how regions of interest can be drawn over the image, providing an interactive feedback mechanism through which the user can direct the renderer. Section 7.5 explains implementation details and Section 7.6 summarises this chapter.

7.1 Previous Work

One of the best known techniques for previewing implicit surfaces at interactive frame rates is based on the dynamic placement of discs that are tangent to the surface [Witkin and Heckbert, 1994; Hart et al., 2002]. The discs are kept apart by the application of repulsion forces and are constrained to remain on the surface. Each disc is also made tangent to the surface by sharing the surface normal at the point where it is located. This previewing system relies on a characteristic of our visual system whereby we are able to infer the existence of an object based solely on the distribution of a small number of features on the surface of that object [Wolfe et al., 2005]. This visual trait only works, however, when the surface of the object is simple and fairly smooth. If the surface is irregular, as in the case of a fractal hypertexture, a random distribution of discs is visible and the object becomes difficult to perceive.

An approximate representation of an implicit surface can be generated by subdividing the space in which the surface is embedded into progressively smaller voxels and using a surface classification technique to identify which voxels are potentially intersecting with the surface. One such spatial subdivision method employs interval arithmetic to perform the surface classification step [Duff, 1992]. The subdivision strategy of this method is adapted from an earlier work and is not suitable for interactive previewing [Woodwark and Quinlan, 1982]. One must wait for the subdivision to finish before any surface approximation can be visualised unless some additional data processing is added, which will tend to slow down the algorithm. Another spatial subdivision method employs affine arithmetic to perform surface classification and subdivides space with an octree data structure [de Figueiredo and Stolfi, 1996]. The octree voxels are rendered from back to front, relative to the viewpoint, with a painter's algorithm. This subdivision strategy is wasteful as it tracks the entire surface through subdivision, including parts that are occluded and that could be safely discarded for a given viewing configuration.

Instead of performing object space subdivision, one can also perform image space subdivision in order to obtain a progressive rendering mechanism. Image space subdivision provides a general approach to progressive refinement rendering and can be used for any rendering problem, not only for the visualisation of implicit surfaces. tile subdivision in image space was originally proposed as an anti-aliasing method for ray tracing [Whitted, 1980]. Four rays are shot at the corners of each rectangular tile. If the computed colours for these rays differ by more than some specified amount, the tile is subdivided into four smaller tiles and more rays are shot through the corners of the new tiles. This type of image space subdivision can also be used for progressive refinement previewing by Gouraud shading the interior of each tile based on the colours at its corners. As the tiles become progressively smaller, the image converges to the correct rendering solution. Such a previewing tool was implemented as part of the `Rayshade` public domain ray tracer [Kolb, 1992]. Hollasch [1992] presented another simple image tile subdivision mechanism for ray tracers.

Rather than simply comparing the colours at the corners of a tile, more sophisticated approaches to image space subdivision have been proposed for ray tracing [Painter and Sloan, 1989; Maillot et al., 1992]. These approaches feature a stochastic distribution of rays that are shot around each image tile. By performing a statistical analysis on the colours returned by these rays, it is possible to make a decision with a desired degree of confidence on whether the tile should be subdivided or not. These probabilistic subdivision methods have also been

extended to distributed ray tracers running on massively parallel computers [Notkin and Gotsman, 1997]. There are two problems associated with statistical image subdivision methods. One problem is that the decision to subdivide a tile is entirely dependent on the information returned by a discrete set of rays. Because this discrete set is only an approximation to a continuous image distribution, wrong subdivision decisions can sometimes occur. This often leads to small objects being missed by the progressive ray tracer. The other problem is that subdivision techniques in image space do not take into account any information about the surface that is being rendered. They only look at colour information on the image plane. The subdivision of a tile causes new rays to be shot that do not take advantage of any surface information that may have been gathered whilst tracing rays that originated earlier in the subdivision process.

The progressive refinement previewer that is described in this chapter also follows an image space subdivision principle but incorporates object space information. Spatial information about the surface, as seen through some area of the image, is retrieved and used to guide the subdivision of that area. The retrieval of spatial information about the surface is done with either Lipschitz bounds or reduced affine arithmetic, depending on which of the two versions of the progressive refinement renderer is used. This combination of image space subdivision with an object space classification leads to much faster convergence rates for the progressive rendering of implicit surfaces. Progressive refinement rendering can be applied to any implicit function $f(\mathbf{x})$ and not only to implicit functions $f(f_0(\mathbf{x}), \mathbf{x})$ that result from hypertexturing a simpler function $f_0(\mathbf{x})$.

7.2 Progressive Refinement for Lipschitz Continuous Surfaces

Progressive refinement rendering of implicit surfaces that are Lipschitz continuous proceeds by subdividing the image space into increasingly smaller tiles. We use the term "tile" in this context to refer to square partitions of the image space with arbitrary size. A tile always corresponds to a $n \times n$ set of pixels on the screen so that the boundary of the tile coincides with the boundary of that square array of pixels. As tiles are subdivided, the size n in pixels of newer tiles progressively decreases. The smallest tiles used by the progressive renderer are sized 1×1 and correspond to the screen extent of just one pixel.

The portion of the scene that is visible through a tile defines a quadrilateral pyramid with the apex located at the viewpoint. Shooting such pyramidal rays through each image pixel has previously been proposed as a technique to perform anti-aliasing and to compute fuzzy reflections and shadows [Genetti et al., 1998]. For simplicity of implementation, however, cones rather than pyramids are used to enclose the space visible through each tile. In this respect, the algorithm has similarities with cone tracing as cones are shot into the scene and checked for intersection with the surface [Amanatides, 1984]. The difference with the original cone tracing algorithm is that, rather than compute a visibility coverage mask that is used to perform anti-aliasing, the tiles are subdivided and therefore the cones get progressively thinner as they get closer to the surface. This is so that one can guarantee that no surface intersection will occur inside each cone up to some maximum distance. As the distance travelled along a cone approaches the intersection distance to the surface, this guarantee can only be provided if the aperture angle of the cone is made smaller. Having stressed this difference, the process of shooting a cone through a square tile and into the scene is still referred to as "cone tracing".

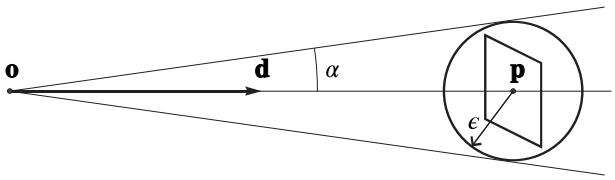


Figure 7.1: The geometry of a cone that encloses the space visible through a tile.

A queue data structure is used to hold tiles waiting to be processed. The rendering algorithm is initialised by placing a set of large tiles that covers the whole visible portion of the image space onto the queue. If the image to be rendered is square, in particular, only one tile needs to be used to initialise the queue. The algorithm then iterates by removing the tile from the top of the queue and performing the following steps:

1. Trace cone through the tile up to a maximum distance.
2. Render the tile by evaluating the shading model at the point of maximum distance along the axis of the cone.
3. If the tile is larger than a pixel, subdivide it and append the children to the end of the queue.

Speculative sphere tracing, rather than the cone tracing algorithm, is used for tiles that have reached pixel size (Section 5.2.2). The parts of the image that are covered by these tiles receive their final colour in step (2), as explained above, and are no longer affected by subsequent iterations. The progressive renderer finishes when the queue becomes empty. Once this happens the image will have reached its best quality. The final image is exactly equal to that which would have been obtained by conventional sphere tracing. This is because the final pixel colours are obtained by ray casting through the centre of each pixel along the distance that still needs to be travelled toward the surface.

7.2.1 Cone Tracing

Consider a tile with dimensions $n\Delta l \times n\Delta l$ where $n \times n$ is, as before, the number of pixels contained by the tile and Δl is the lateral size of a pixel, measured along the image plane. The tile is centred at the point $\mathbf{p}$ on the image plane and a cone is traced through it with the apex placed at the camera's viewpoint $\mathbf{o}$. Figure 7.1 exemplifies this geometry. The axis of the cone is a parametric ray defined as:

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}, \tag{7.1}$$

where the normalised direction vector is $\mathbf{d} = (\mathbf{p} - \mathbf{o})/\|\mathbf{p} - \mathbf{o}\|$. The aperture of the cone is an angle α large enough to encompass all the scene visible through the tile. For the purpose of finding this angle, the smallest bounding sphere that encloses the tile is used. The radius of the bounding sphere is given by $\epsilon = 0.5\sqrt{2}n\Delta l$ and the aperture angle is:

$$\alpha = \tan^{-1} \frac{\epsilon}{\|\mathbf{p} - \mathbf{o}\|}. \tag{7.2}$$

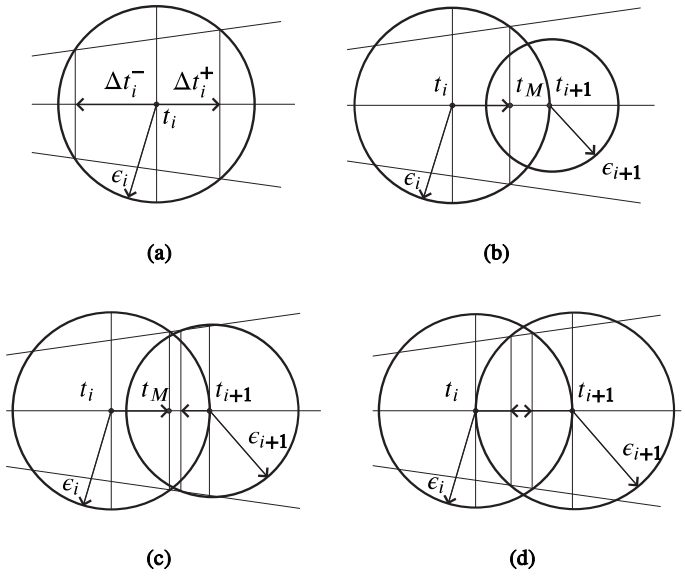


Figure 7.2: Several configurations that may arise while performing cone tracing.

The cone is traced through the scene by stepping along the axis $\mathbf{r}$ with step lengths that are guaranteed not to cause intersection with the implicit surface. Such a guarantee is provided by the Lipschitz bound λ of the implicit function f. Application of the sphere tracing algorithm (Section 5.2.1) along the axis of the cone leads to a sequence of steps, taken according to the iterative equation $t_{i+1} = t_i + f(\mathbf{r}(t_i)))/\lambda$. Similar to what occurrs when sphere tracing is used for ray casting, there is, for every iteration of the cone tracing procedure, a sphere centred at the point $\mathbf{r}(t_i)$ and with radius $f(\mathbf{r}(t_i)/\lambda$ that encloses a region of space known to be completely outside the surface. As the tracing procedure converges towards the intersection point, these spheres become smaller and more densely packed.

Tracing along the axis $\mathbf{r}(t)$ can only proceed while the sequence of unbounding spheres encloses the cone. There is a maximum distance t_M after which parts of the cone begin to fall outside the union of all the spheres. It is known with certainty that for $t_i \leqslant t_M$ no intersection between the cone and the implicit surface has occurred. For $t_i > t_M$ this certainty no longer exists and cone tracing for the current tile must be terminated. The tracing will proceed for distances larger than t_M after the tile has been subdivided and cones with smaller angles of aperture have been generated. The smaller cones that result from the subdivision will then start tracing from t_M.

Figure 7.2 shows several situations that occur as part of the cone tracing procedure. Let $\epsilon_i = f(\mathbf{r}(t_i))/\lambda$ be the radius of the sphere that is centred on the point $\mathbf{r}(t_i)$ for the distance t_i along the cone axis. Figure 7.2(a) shows that any individual sphere must be large enough to bound the cone in the neighbourhood of its centre at $\mathbf{r}(t_i)$. This is true if the radius of the sphere is larger than the radius of the section of the cone at t_i:

$$\epsilon_i > t_i \tan \alpha. \tag{7.3}$$

Once condition (7.3) is verified, it is possible to define two positive offsets Δt_i^- and Δt_i^+, relative to t_i, that give the neighbourhood $t_i - \Delta t_i^- \leqslant t_i \leqslant t_i + \Delta t_i^+$ where the sphere is known to bound the cone. Inside this range of distances, it is guaranteed that no intersections with the implicit surface will exist. The two offsets are given by:

$$\Delta t_i^- = \delta + t_i \sin^2 \alpha \quad \text{and} \tag{7.4a}$$

$$\Delta t_i^+ = \delta - t_i \sin^2 \alpha, \quad \text{with} \tag{7.4b}$$

$$\delta = \sqrt{\epsilon_i^2 \cos^2 \alpha - t_i^2 \sin^2 \alpha}. \tag{7.4c}$$

It is a trivial consequence of (7.3) that the square root used to compute the factor δ is well behaved in the sense that it never returns complex values.

Figures 7.2(b) to 7.2(d) illustrate the three situations that can arise when marching from the sphere at t_i to the next sphere at t_{i+1}. In Figure 7.2(b), the next sphere is not large enough to verify (7.3). In this case, cone tracing must stop at the current sphere with radius ϵ_i and the maximum distance that can be travelled along the cone is $t_M = t_i + \Delta t_i^+$. In Figure 7.2(c), the sphere at t_{i+1} satisfies (7.3) but there is still a small part of the cone that falls outside the union of the two spheres. This situation can be detected by the condition:

$$t_{i+1} - t_i < \Delta t_{i+1}^- + \Delta t_i^+. \tag{7.5}$$

If (7.5) does not hold then again the sphere at t_{i+1} must be ignored, with the distance t_M being given as in the previous case. Finally, in Figure 7.2(d), cone tracing can proceed from t_i to t_{i+1} with the testing of subsequent spheres.

Figure 7.3 shows the cone tracing algorithm in pseudo-code form. The cone is traced from a starting distance t_0 up to a large enough distance t_∞ so that any valid intersections are known to be inside the interval $[t_0, t_\infty]$. The distance t_∞ is usually found by performing a quick intersection test between the cone and a bounding sphere that surrounds the whole implicit surface. The algorithm returns the maximum distance t_M that can be traced along the cone. Within the interval $[t_0, t_M]$ there are no intersections with the surface. If it happens that the value t_M returned by the algorithm is larger than t_∞ then the surface is not intersected along the whole extent of the cone. One necessary prerequisite for the cone tracing algorithm to work is that condition (7.3) must hold for the initial sphere placed at t_0. If the condition is not verified, the tile to which the cone belongs will have to be subdivided, with no cone tracing actually occurring. This is handled as part of the tile subdivision process, as explained in Section 7.2.3. The offsets Δt_{i+1}^- and Δt_{i+1}^+ are computed simultaneously with equations (7.4) even though Δt_{i+1}^+ will only be required during the subsequent iteration, when it becomes Δt_i^+. This is more efficient than computing Δt_i^+ and Δt_{i+1}^- independently for every iteration.

7.2.2 Cone Rendering

Once cone tracing terminates, with the computation of the distance t_M, the tile through which the cone was shot is painted on the screen. The colour for this tile is obtained by evaluating some appropriate shading model, which, from the point of view of the progressive refinement algorithm, is regarded as a generic function $s(\mathbf{x}, \mathbf{n}, \mathbf{v})$ that returns a colour at the point $\mathbf{x}$ with

```
let t_i = t_0;                                  // initial distance
let ε_i = |f(r(t_0))|/λ;                        // initial radius
compute Δt_i^+;                                 // initial offset
while t_i < t_∞ do
    let t_{i+1} = t_i + ε_i;                    // next distance
    let ε_{i+1} = |f(r(t_{i+1}))|/λ;            // next radius
    if ε_{i+1} < t_{i+1} tan α                  // condition (7.3)
        return t_i + Δt_i^+;
    compute Δt_{i+1}^- and Δt_{i+1}^+;          // equations (7.4)
    if t_{i+1} − t_i < Δt_{i+1}^- + Δt_i^+      // condition (7.5)
        return t_i + Δt_i^+;
    let t_i = t_{i+1};                          // take step
    let ε_i = ε_{i+1};
    let Δt_i^+ = Δt_{i+1}^+;
return t_i;
```

Figure 7.3: The cone tracing algorithm in pseudo-code format.

surface normal $\mathbf{n}$ and view vector $\mathbf{v}$. Any other shading parameters such as light sources or surface properties are queried from inside the shading function. The point where the shading model is applied is $\mathbf{x} = \mathbf{r}(t_M)$. The surface normal vector is the gradient $\nabla f(\mathbf{r}(t_M))$ of the implicit function, followed by normalisation. The vector $\mathbf{v} = -\mathbf{d}$ is the view vector (recall (7.1) for the definition of $\mathbf{d}$).

The shading model can be evaluated at any point in space and not only for points on the implicit surface. During the cone tracing process, the rays that constitute the axes of the cones have not yet reached the surface and so shading values are being computed that do not correspond to the correct shading or geometry of the surface. For any given cone, the evaluation of the shading model $s(\mathbf{x}, \mathbf{n}, \mathbf{v})$ corresponds to shading an implicit surface given by $\{\mathbf{x} \in \mathbb{R}^3 : \tilde{f}(\mathbf{x}, \mathbf{r}(t_M)) = 0\}$, where the new implicit function is $\tilde{f}(\mathbf{x}, \mathbf{r}(t_M)) = f(\mathbf{x}) - f(\mathbf{r}(t_M))$. The function $\tilde{f}$ generates a surface that is larger than the one generated by f. Recalling the introduction to Chapter 2 and the families of iso-surfaces that enclose any given implicit surface, it is possible to show that the correct implicit surface is completely enclosed by the surface generated from $\tilde{f}$, given that $f(\mathbf{x})$ and $f(\mathbf{r}(t_M))$ have the same sign when evaluated at points $\mathbf{x}$ and $\mathbf{r}(t_M)$ outside the surface. As cone tracing progresses and newer cones get closer to the surface, the term $f(\mathbf{r}(t_M))$ vanishes with the consequence that $\tilde{f} \to f$ and the shading values converge towards the correct shading of the surface. This progression, in visual terms, corresponds to a gradual shrinking of the perceived surface, with the surface finally settling on its correct shape at the completion of the progressive rendering algorithm.

The painting of a tile on the screen is done with a uniform colour that corresponds to the shading value of the point at the centre of the tile, since it is through this point that the axis of the cone passes. While painting the square region of the screen that corresponds to a tile, the

previous colour that was stored in that region is overwritten by the new colour. The previous colour was obtained when painting the parent tile. Through this procedure, the screen buffer is constantly refreshed with shading data as subdivision of the tiles progresses.

7.2.3 Tile Subdivision

The subdivision of tiles requires that the width and height of the image in pixels be first decomposed into a product of prime numbers. In this way, it becomes possible to know, at each level of subdivision, how a given tile should be split so that the newer tiles still correspond to an integer number of pixels on the screen. For an image with a resolution of $m \times n$ pixels, the prime number factorisation results in:

$$m = u\, p_1^{k_1} p_2^{k_2} \ldots p_n^{k_n}, \tag{7.6a}$$

$$n = v\, p_1^{k_1} p_2^{k_2} \ldots p_n^{k_n}, \tag{7.6b}$$

where $u = m/\gcd(m,n)$, $v = n/\gcd(m,n)$ and the function $\gcd(m,n)$ returns the greatest common divisor between m and n. The sequence of prime factors $p_1, p_2, \ldots, p_n$ in (7.6) is ordered by decreasing values. The pair (u,v) indicates how the image should be initially subdivided into a set of top level square tiles of the same size. If $m = n$, in particular, then $u = v = 1$ and the image corresponds to a single top level tile. For a 800×600 image, to give an example of the more general rectangular image case, we have $m = 4 \times 5^2 \times 2^3$ and $n = 3 \times 5^2 \times 2^3$. The queue of tiles used by the progressive refinement algorithm, in this case, is initialised with 4×3 top level tiles with a resolution of 200×200 pixels each. At the first level of subdivision, every top level tile is then subdivided into 5×5 tiles with a resolution of 40×40 pixels each. The optimal image subdivision scenario occurs with square images that have a resolution of $2^k \times 2^k$, for some $k > 0$, where every tile is always subdivided into 2×2 smaller tiles. The worst scenario occurs when either m or n is a prime number, in which case it becomes impossible to perform any further prime factorisation. The progressive refinement algorithm is then initialised with $m \times n$ top level tiles with the consequence that the subdivision stage is skipped and the algorithm goes to the final ray casting stage that occurs at the pixel level.

The generation of new cones is part of the tile subdivision process. Figure 7.4 shows how one of the $p_i \times p_i$ child cones is generated after its parent cone was traced, where p_i is the prime factor at subdivision level i. The unit direction vector $\mathbf{d}_i$ of the child cone is the one that goes from the camera's viewpoint through the centre point of the child tile. The initial distance t_{0_i} along the child cone is:

$$t_{0_i} = t_{M_{i-1}}/(\mathbf{d}_i \cdot \mathbf{d}_{i-1}), \tag{7.7}$$

where $\mathbf{d}_{i-1}$ is the unit direction vector of the parent cone. Equation (7.7) makes the child cone start off from a point that lies in the plane orthogonal to the parent cone at $t_{M_{i-1}}$. The aperture angle of the child cone is given by (7.2) and is always smaller than the aperture angle of the parent cone. If the child cone does not verify condition (7.3), it is further subdivided into $p_{i+1} \times p_{i+1}$ children. The initial distance $t_{0_{i+1}}$ for each new child is computed with (7.7), using again the parameters $t_{M_{i-1}}$ and $\mathbf{d}_{i-1}$ from the original parent cone. Tile subdivision stops once all the nearest descendants of the parent cone for which condition (7.3) holds have been reached.

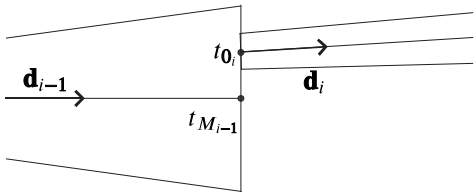


Figure 7.4: A child cone shown in relation to its parent cone as part of the tile subdivision process.

7.2.4 Results

Figures 7.5, 7.6 and 7.7 show four snapshots each, taken during the progressive refinement rendering of three different types of implicit surfaces: algebraic surfaces (Section 2.2.2), blobby surfaces (Section 2.2.4) and hypertextures (Section 2.3.1). All snapshots were rendered with a resolution of 800 × 800 pixels. The snapshots were obtained at the transition point between two levels of subdivision in the image, i.e. when all the tiles at one level have been processed and before processing any of the tiles at the next lower levels.

The algebraic surface shown in Figure 7.5 was first presented by Mitchell to demonstrate his robust ray casting algorithm based on interval arithmetic [Mitchell, 1990]. The implicit function for Mitchell's surface is:

$$f(x, y, z) = 4\left(x^4 + (y^2 + z^2)^2\right) + 17x^2(y^2 + z^2) - 20(x^2 + y^2 + z^2) + 17. \qquad (7.8)$$

The Lipschitz constant of (7.8), when considered over $\mathbb{R}^3$, is infinite. The surface, however, is known to exist inside a cube with dimensions $[-2, 2] \times [-2, 2] \times [-2, 2]$. The Lipschitz constant of $f(x, y, z)$ becomes finite when evaluated over this subdomain only.

Figure 7.6 shows a blobby surface that was made to resemble a figure in a painting by Dali. The implicit surface for this model is a sum of sixty radial basis functions:

$$f(\mathbf{x}) = 0.5 - \sum_{i=1}^{60} s_i\, p(\|\mathbf{x} - \mathbf{x}_i\| / R_i), \qquad (7.9)$$

where s_i is the strength, R_i is the radius and $\mathbf{x}_i$ is the position of each basis function. The factor 0.5 is the surface threshold. The basis function p is a polynomial with a compact support in the range $[0, 1]$, verifying $p(0) = 1$ and $p(1) = 0$. The evaluation of the implicit function for any point $\mathbf{x}$ that is outside the support of all the basis functions returns the constant value $f(\mathbf{x}) = 0.5$. Shading computations cannot be properly performed in this outside region because $\nabla f(\mathbf{x}) = \mathbf{0}$. To overcome this problem, the implicit function (7.9) was modified to return $f(\mathbf{x}) = 0.5 + \|\mathbf{x} - \mathbf{x}_i\| - R_i$ for points where $f(\mathbf{x})$ would have been equal to 0.5 otherwise and where $\mathbf{x}_i$ and R_i are related to the basis function that is closest to $\mathbf{x}$. With this modification, the surface appears in the early stages of the progressive visualisation as a union of spheres of different radii, where each sphere is centred at one of the $\mathbf{x}_i$ points. It must be mentioned that this modification is only required for models that are generated from sums of compactly supported basis functions. Surface models that use basis functions of infinite support, e.g. Gaussian functions, have a well defined surface normal at every stage of progressive refinement.

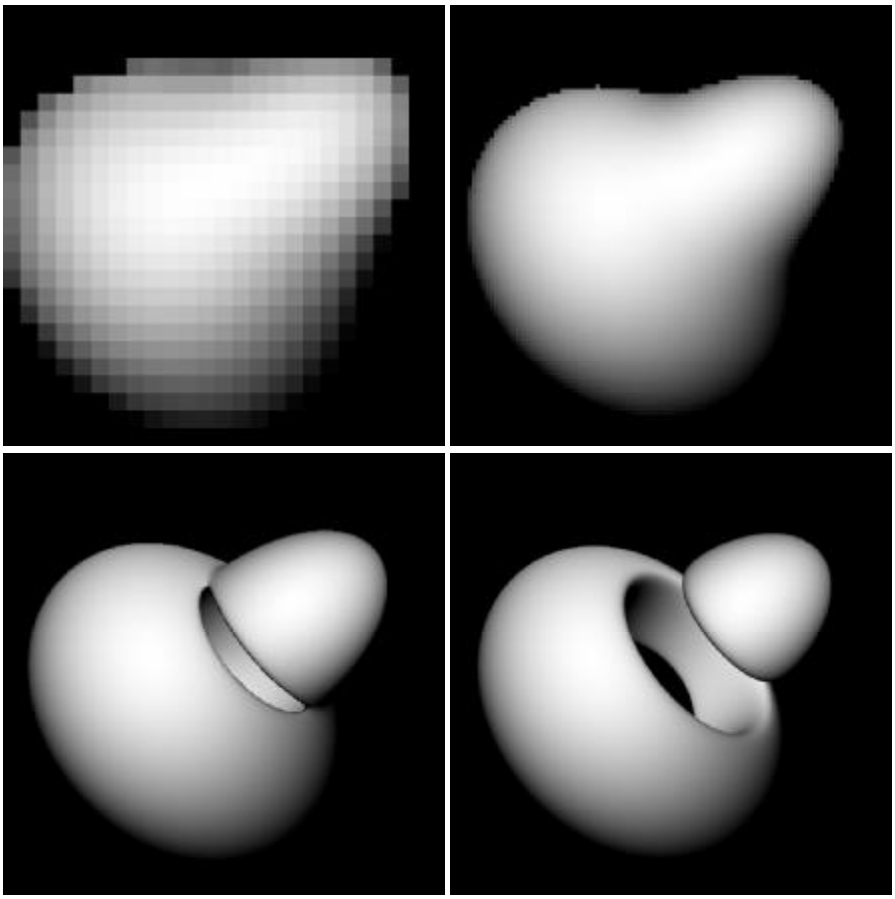


Figure 7.5: From left to right, top to bottom, snapshots taken during the progressive refinement rendering of an algebraic surface. The snapshots were taken after 626, 8178, 107 862 and 410 590 iterations, respectively. The wall clock times at each snapshot are 0.35s, 0.71s, 5.71s and 19.18s, respectively.

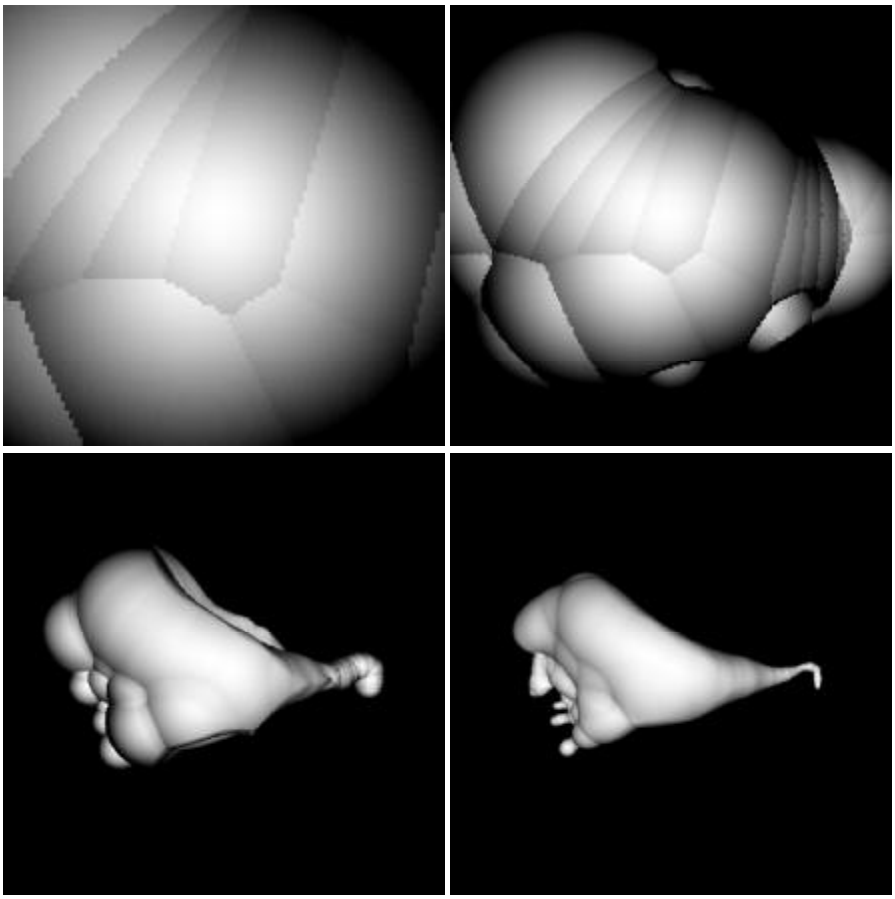


Figure 7.6: From left to right, top to bottom, snapshots taken during the progressive refinement rendering of a blobby surface. The snapshots were taken after 13 151, 53 151, 163 851 and 298 467 iterations, respectively. The wall clock times at each snapshot are 2.56s, 18.41s, 1m 17.11s and 2m 39.48s, respectively.

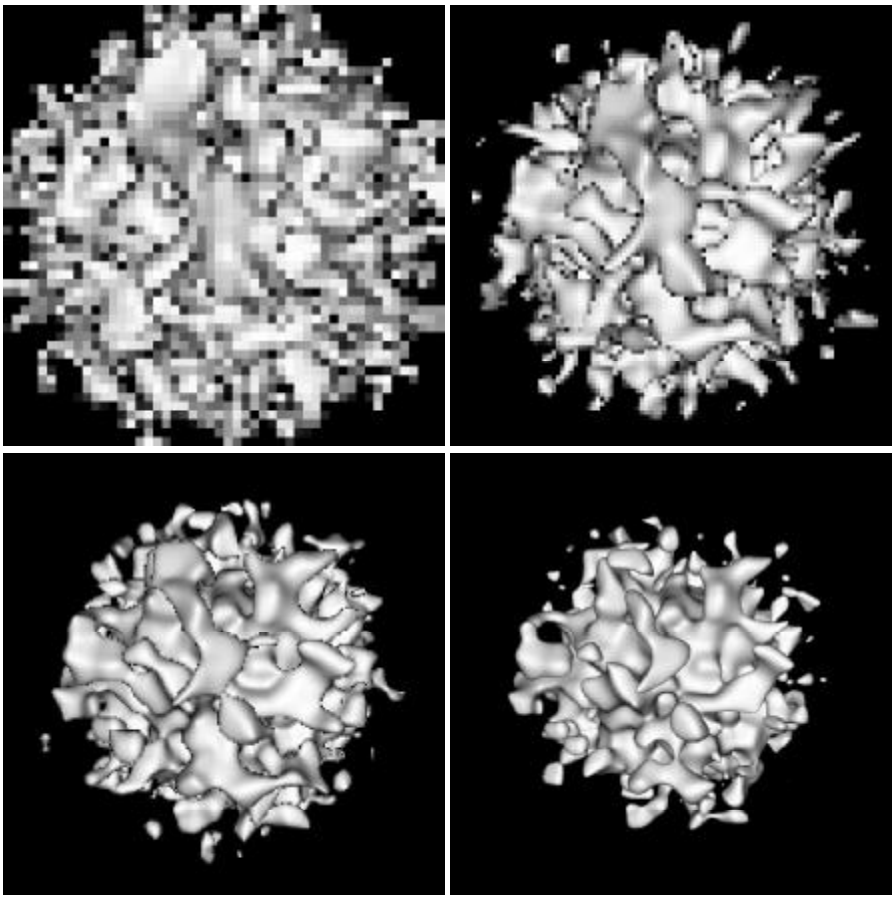


Figure 7.7: From left to right, top to bottom, snapshots taken during the progressive refinement rendering of a hypertextured surface. The snapshots were taken after 3103, 10 755, 31 279 and 327 295 iterations, respectively. The wall clock times at each snapshot are 0.45s, 0.90s, 2.10s and 23.22s, respectively.

	Previewer		Speculative Sphere Tracing		Sphere Tracing	
	Time	# Evals	Time	# Evals	Time	# Evals
Fig. 7.5	19.18s	26 803 413	14.03s	30 701 240	19.87s	44 895 932
Fig. 7.6	2m39.48s	30 563 596	4m47.56s	58 132 753	6m59.27s	84 760 569
Fig. 7.7	23.22s	12 641 154	29.69s	21 425 128	36.69s	27 504 354

Table 7.1: Statistics for Figures 7.5, 7.6 and 7.7 with three different rendering methods.

It is possible to improve the rendering time for the model of Figure 7.6 by first clipping every cone against the bounding spheres of radii R_i that represent the support of each basis function. As a consequence, the distance along the cone is partitioned into disjoint intervals where only a small set of basis functions influence each particular interval. This optimisation was first proposed to speed up the ray tracing of soft objects and can be extended to our cone tracing technique with little difficulty [Wyvill and Trotman, 1990]. It makes the evaluation of (7.9) much more efficient since it is not necessary to iterate over all the sixty basis functions for every function call. Localised Lipschitz bounds can also be used inside each interval, reflecting only the basis functions that contribute to that interval and providing further improvement to the convergence rate of the renderer.

Figure 7.7 shows the progressive refinement rendering of a stochastic implicit surface generated through hypertexturing. The implicit function is:

$$f(f_0(\mathbf{x}), \mathbf{x}) \;=\; f_0(\mathbf{x}) + 0.8\, f_N(4\mathbf{x}), \tag{7.10}$$

where $f_0(\mathbf{x}) = 1 - \|\mathbf{x}\|$ defines an implicit sphere of unit radius and f_N is Perlin's improved gradient noise function [Perlin, 2002]. The basic features of the stochastic surface become discernible early in the rendering process, stabilising as the rendering progresses and the surface shrinks towards its correct shape.

Table 7.1 shows the rendering times, obtained on a Pentium 4 1.8GHz laptop machine, and the number of evaluations of the implicit function for the three previous implicit surface models. A comparison is made between the progressive rendering method, the speculative sphere tracing algorithm of Section 5.2.2 and the original sphere tracing algorithm of Section 5.2.1. The progressive refinement algorithm also uses speculative sphere tracing for rendering terminal tiles, i.e. tiles that have reached pixel size.

To test the performance of the progressive refinement renderer with increasingly complex surfaces, the hypertexture example of Figure 7.7 was modified to sum the contribution of several layers of procedural noise, based on the rescale-and-add method of Section 3.2.4. Examples using two to five layers of procedural noise are shown in Figure 7.8. The bar chart shown in Figure 7.9 compares the total rendering times for these surfaces.

Progressive refinement previewing incurs several costs that a ray casting algorithm does not have. These consist of the subdivision of tiles, the invocation of the shading model during the rendering of non-terminal tiles for previewing purposes and the periodical updating of the hardware frame buffer. The previewer compensates for these extra costs by being able to quickly eliminate large areas of empty space as part of the cone tracing procedure, thereby reducing the number of evaluations of the implicit function. For most cases, this leads to a

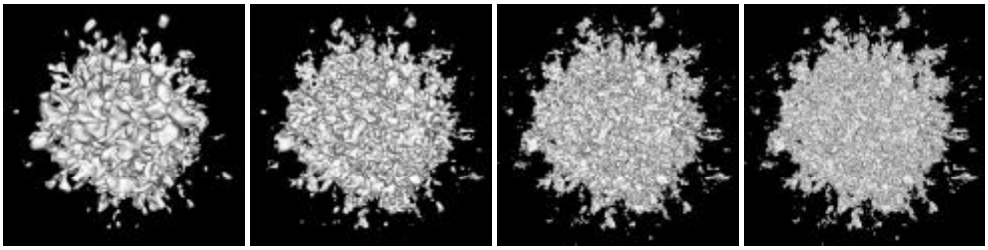


Figure 7.8: A hypertexture with (from left to right) two to five layers of procedural gradient noise.

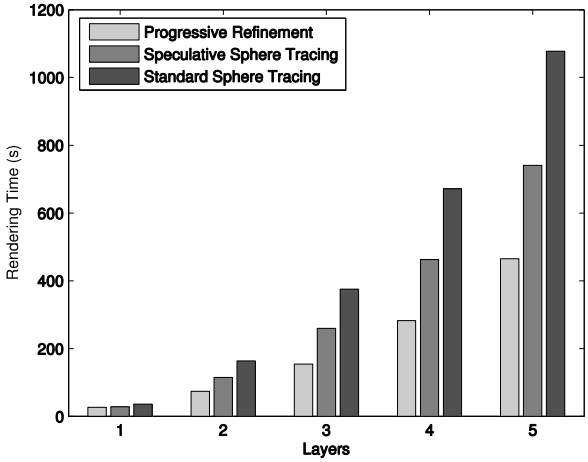


Figure 7.9: Comparison between the total rendering times between the progressive refinement previewer, sphere tracing and speculative sphere tracing for the surfaces shown in Figure 7.8.

reduction of the total rendering time as shown by the previous results. If, however, the implicit function is very simple and can be evaluated quickly, the previewer can no longer compensate for the extra progressive refinement costs compared to a ray casting algorithm. This is the case of the algebraic surface of Figure 7.5, as shown in Table 7.1, where it takes longer to do progressive refinement even though the number of evaluations of the implicit function with previewing is still the smallest for the three rendering methods.

7.3 Progressive Refinement for General Surfaces

The progressive refinement rendering algorithm can be generalised to surfaces that do not have Lipschitz continuity. This is done by replacing the unbounding spheres, obtained with Lipschitz bounds, with reduced affine arithmetic (see Section 5.3.3). Instead of shooting cones through each tile on the screen, the viewing frustum is subdivided into smaller elements called *cells*. The interval extension of the implicit function f inside a cell is obtained through its evaluation with reduced affine arithmetic. The main stage of the method consists in the binary subdivision of the space, visible from the camera, into progressively smaller cells that are known to straddle

the boundary of the surface. The subdivision mechanism stops as soon as the projected size of a cell on the screen becomes smaller than the size of a pixel. The procedure for rendering general implicit surfaces with progressive refinement can be broken down into the following steps:

1. Build an initial cell coincident with the camera's viewing frustum. The near and far clipping planes are determined so as to bound the implicit surface.
2. Recursively subdivide this cell into smaller cells. Discard cells that do not intersect with the implicit surface. Stop subdivision if the size of the cell's projection on the image plane falls below the size of a pixel.
3. Assign the shading value of a cell to all pixels that are contained inside its projection on the image plane. The shading value for a cell is taken from the evaluation of the shading model at the centre point of the cell.

The evaluation of the implicit function f inside a cell requires a reduced affine arithmetic representation with four error symbols since a cell represents a portion of three-dimensional space. Three of the error symbols represent the three spatial degrees of freedom. These are the horizontal distance u along the image plane, the vertical distance v along the same image plane and the distance t along the ray that passes through the point at (u, v). A reduced AA variable $\hat{a}$ has, therefore, the following representation:

$$\hat{a} = a_0 + a_u e_u + a_v e_v + a_t e_t + a_k e_k. \tag{7.11}$$

The noise symbols e_u, e_v and e_t are shared between all reduced AA variables in the system, which allows for the representation of correlation information between reduced AA variables relative to the u, v and t degrees of freedom. The extra noise symbol e_k is included to account for uncertainties in the variable $\hat{a}$ that are not shared with any other variable.

7.3.1 The Anatomy of a Cell

A cell is a portion of the camera's viewing frustum that results from a recursive subdivision along the u, v and t parameters. Figure 7.10 depicts the geometry of a cell. It has the shape of a truncated pyramid of quadrangular cross-section, similar to the shape of the viewing frustum itself. Four vectors, taken from the camera's viewing system, are used to define the spatial extent of a cell. These vectors are:

The vector o This is the location of the camera in the world coordinate system.

The vectors $\mathbf{p}_u$ and $\mathbf{p}_v$ They represent the horizontal and vertical direction along the image plane. The length of these vectors gives the width and height, respectively, of a pixel in the image plane.

The vector $\mathbf{p}_t$ It is the vector from the camera's viewpoint and is always defined orthogonally to the image plane. The length of this vector gives the distance from the viewpoint to the image plane.

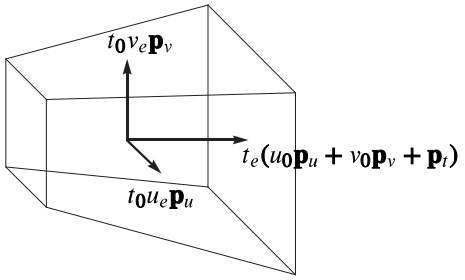


Figure 7.10: The geometry of a cell. The vectors show the three medial axes of the cell.

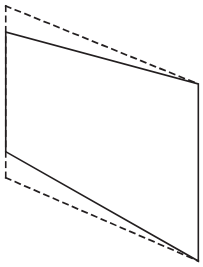


Figure 7.11: The outline of a cell (solid line) and the outline of its reduced AA representation (dashed line) shown in profile. The reduced AA representation is a prism that forms a tight enclosure of the cell.

The vectors $\mathbf{p}_u$, $\mathbf{p}_v$ and $\mathbf{p}_t$ define a left-handed perspective viewing system. The position of any point $\mathbf{x}$ inside the cell is given by the following inverse perspective transformation:

$$\mathbf{x} = \mathbf{o} + (u\mathbf{p}_u + v\mathbf{p}_v + \mathbf{p}_t)t = \mathbf{o} + ut\mathbf{p}_u + vt\mathbf{p}_v + t\mathbf{p}_t. \tag{7.12}$$

The spatial extent of a cell is obtained from the above by having the u, v and t parameters vary over appropriate intervals $U = [u_a, u_b]$, $V = [v_a, v_b]$ and $T = [t_a, t_b]$. Knowledge about the presence of the surface inside a cell can be obtained by testing the condition $0 \in F(U, V, T)$, where F is an interval extension of f (see Section 5.3). A feasible interval extension is obtained by evaluating $F(U, V, T) = f(\hat{\mathbf{x}}(U, V, T))$ with reduced affine arithmetic, where $\hat{\mathbf{x}}(U, V, T)$ is the reduced AA representation of the spatial extent of a cell. For simplicity, this variable will be referred to only as $\hat{\mathbf{x}}$ henceforth. To compute $\hat{\mathbf{x}}$, a change of variables must first be performed. The reduced AA variable $\hat{u} = u_0 + u_e e_u$ will span the same interval $[u_a, u_b]$ as u does if we have:

$$u_0 = (u_b + u_a)/2, \tag{7.13a}$$
$$u_e = (u_b - u_a)/2. \tag{7.13b}$$

Similar results apply for the v and t parameters. Substituting $\hat{u}$, $\hat{v}$ and $\hat{t}$ in (7.12) for u, v and t, one obtains:

$$\begin{aligned} \mathbf{x} = \mathbf{o} &+ t_0 u_0 \mathbf{p}_u + t_0 v_0 \mathbf{p}_v + t_0 \mathbf{p}_t \\ &+ t_0 u_e e_u \mathbf{p}_u + t_0 v_e e_v \mathbf{p}_v \\ &+ u_0 t_e e_t \mathbf{p}_u + v_0 t_e e_t \mathbf{p}_v + t_e e_t \mathbf{p}_t \\ &+ t_e u_e e_u e_t \mathbf{p}_u + t_e v_e e_v e_t \mathbf{p}_v. \end{aligned} \tag{7.14}$$

The first line of (7.14) contains only constant terms. The second and third lines contain linear terms of the noise symbols e_u, e_v and e_t. The fourth line contains two non-linear terms $e_u e_t$ and $e_v e_t$, which are a consequence of the non-linearity of the perspective transformation. Since

a reduced AA representation cannot accommodate such non-linear terms they are replaced by the independent noise terms e_{k_1}, e_{k_2} and e_{k_3} for each of the three cartesian coordinates of $\hat{\mathbf{x}}$. The reduced AA vector $\hat{\mathbf{x}}$ is finally given by:

$$\begin{aligned}\hat{\mathbf{x}} = \mathbf{o} &+ t_0(u_0\mathbf{p}_u + v_0\mathbf{p}_v + \mathbf{p}_t) \\ &+ t_0 u_e \mathbf{p}_u e_u + t_0 v_e \mathbf{p}_v e_v \\ &+ t_e(u_0\mathbf{p}_u + v_0\mathbf{p}_v + \mathbf{p}_t)e_t \\ &+ \begin{bmatrix} x_{k_1}e_{k_1} & x_{k_2}e_{k_2} & x_{k_3}e_{k_3} \end{bmatrix}^T,\end{aligned} \tag{7.15}$$

with

$$x_{k_i} = |t_e u_e p_{u_i}| + |t_e v_e p_{v_i}|, \quad i = 1, 2, 3. \tag{7.16}$$

A consequence of the non-linearity of the perspective projection and its subsequent approximation with reduced AA is that the region spanned by $\hat{\mathbf{x}}$ is going to be larger than the spatial extent of the cell. Figure 7.11 shows the geometry of a cell and the region spanned by its reduced AA representation in profile. Because the reduced AA representation has been linearised, its spatial extent is a prism rather than a truncated pyramid. This has further consequences in that the evaluation of $f(\hat{\mathbf{x}})$ is going to include information from the regions of the prism outside the cell and will, therefore, lead to range estimates that are larger than necessary. The linearisation error is more pronounced for cells that exist early in the subdivision process. As subdivision continues and the cells become progressively smaller, their geometry becomes more like that of a prism and the discrepancy with the geometry of $\hat{\mathbf{x}}$ decreases[1].

The subdivision of a cell proceeds by first choosing one of the three perspective projection parameters u, v or t and splitting the cell in half along that parameter. This scheme leads to a *k-d* tree of cells where the sequence of dimensional splits is only determined at run time. The choice of which parameter to split along is based on the average width, height and depth of the cell:

$$\bar{w}_u = 2t_0 u_e \|\mathbf{p}_u\|, \tag{7.17a}$$

$$\bar{w}_v = 2t_0 v_e \|\mathbf{p}_v\|, \tag{7.17b}$$

$$\bar{w}_t = 2t_e \|u_0\mathbf{p}_u + v_0\mathbf{p}_v + \mathbf{p}_t\|. \tag{7.17c}$$

If, say, $\bar{w}_u$ is the largest of these three measures, the cell is split along the u parameter. The two child cells will have their u parameters ranging inside the intervals $[u_a, u_0]$ and $[u_0, u_b]$, where $[u_a, u_b]$ was the interval spanned by u in the parent cell. In practice, the factors of 2 in (7.17) can be ignored without changing the outcome of the subdivision. This subdivision strategy ensures that, after a few iterations, all the cells will have an evenly distributed shape, even when the initial cell is very long and thin.

7.3.2 The Process of Cell Subdivision

Cell subdivision is implemented in an iterative manner rather than using a recursive procedure. The cells are kept sorted in a priority queue based on their level of subdivision. A cell has

[1]This can be demonstrated by the fact that the terms $t_e u_e$ and $t_e v_e$ in (7.14) decrease more rapidly than any of the linear terms u_e, v_e and t_e of the same equation as the latter converge to zero.

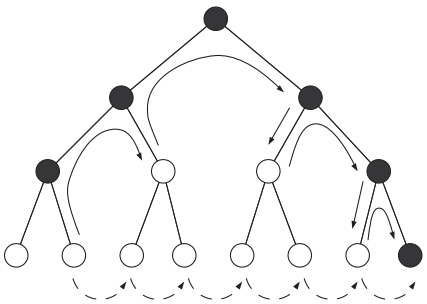


Figure 7.12: Scanning along the depth subdivision tree. Cells represented by black nodes may intersect with the surface. Cells represented by white nodes do not. The solid arrows show progression by depth-first order. The dotted arrows show progression by breadth-first order.

priority over another if it has undergone less subdivision. For cells at the same subdivision level, the one that is closer to the camera will have priority. The algorithm starts by placing the initial cell, which corresponds to the complete viewing frustum, on the priority queue. At the start of every new iteration, a cell is removed from the head of the queue. If the extent of the cell's projection on the image plane is larger than the extent of a pixel, the cell is subdivided and its two children are examined. In the opposite case, the cell is considered a leaf cell and is discarded after being rendered. The two conditions that indicate whether a cell should be subdivided are:

$$u_b - u_a > 1, \tag{7.18a}$$

$$v_b - v_a > 1. \tag{7.18b}$$

The values on the right hand sides of (7.18) are a consequence of the definition of $\mathbf{p}_u$ and $\mathbf{p}_v$ in Section 7.3.1, which cause all pixels to have a unit width and height.

The sequence of events after a cell has been subdivided depends on which of the parameters u, v or t was used to perform the subdivision. If the subdivision occurred along t, there will be two child cells with one in front of the other and totally occluding it. The front cell is first checked for the condition $0 \in F(U, V, T)$. If the condition holds, the cell is pushed into the priority queue and the back cell is ignored. If the condition does not hold, the back cell is also checked for the same condition. The difference now is that, if $0 \notin F(U, V, T)$ for the back cell, a new cell must be searched by marching along the t direction. The first cell scanned, at the same subdivision level of the front and back cells, for which $0 \in F(U, V, T)$ holds is the one that is pushed into the priority queue. On the other hand, if the subdivision occurred along the u or v directions, there will be two child cells that sit side by side relative to the camera without occluding each other. Both cells are processed in the same way. If, for any of the two cells, $0 \in F(U, V, T)$ holds, that cell is placed on the priority queue, otherwise a farther cell must be searched by marching forward in depth.

The process of marching forward from a cell along the depth direction t tries to find a new cell that has a possibility of intersecting the implicit surface by verifying the condition $0 \in F(U, V, T)$. The process is invoked when the starting cell has been determined not to verify the same condition. The reason for having this scanning in depth is because cells that do not intersect with the surface must be discarded. Only cells that verify $0 \in F(U, V, T)$ are allowed

into the priority queue for further processing. Figure 7.12 shows an example of this marching process. The scanning is performed by following a depth-first ordering relative to the tree that results from subdividing in t. The scanning sequence skips over the children of cells for which $0 \notin F(U,V,T)$. The possibility of scanning in breadth-first order, by marching along all the cells at the same level of subdivision, is not recommended because in deeply subdivided trees a very high number of cells would have to be tested.

As mentioned before, when subdivision is performed along t, the back cell is ignored whenever the front cell verifies $0 \in F(U,V,T)$. This does not mean, however, that the volume occupied by this back cell will be totally discarded from further consideration. The front cell may happen to be subdivided during subsequent iterations of the algorithm and portions of the volume occupied by the back cell may then be revisited by the depth marching procedure.

7.3.3 Results

Figure 7.13 shows the progressive refinement renderer for general implicit surfaces applied to the hypertexture that is discussed in Section 5.4.3. This surface is not Lipschitz continuous and could not have been previewed with the cone tracing technique of Section 7.2. The sharp ridges caused by the square root that is present in the implicit function for this surface appear as black lines in the early stages of refinement. When rendering a cell, the surface normal is taken at the centre of the front face, as seen from the viewpoint. For those cells that are too close to the ridges, the normals may sometimes point away from the viewpoint, causing the shading function to interpret this as a back facing surface and render it in black accordingly. The rendering of the ridge lines improves as the algorithm progresses and the cells track the position of the surface with increasing accuracy although some black pixels may still remain in the final rendering.

The image in Figure 7.13 took $1m\ 17.12s$ to complete, compared with $2m\ 45.57s$ to render the same image with the reduced affine arithmetic ray caster of Section 5.4. The Lipschitz continuous surfaces shown in Section 7.2, which were rendered with the cone tracing previewer, can also be rendered with the reduced AA previewer. For Lipschitz continuous surfaces, however, cone tracing is generally much more efficient than reduced AA previewing. When performing cone tracing, only a point-wise evaluation of the implicit function is necessary. This, together with the knowledge of a Lipschitz bound for the function, allows an unbounding sphere to be computed outside the surface. In contrast, the computation of an interval extension for the implicit function inside a cell requires several affine arithmetic computations that use four error symbols, as exemplified by equation (7.15).

7.4 Regions of Interest

A user can interactively influence the rendering algorithms by drawing a rectangular region of interest (ROI) over the image. The image will then be refined only inside the specified region. This is accomplished by creating a secondary queue that stores the image tiles that are relevant to the ROI. When the user finishes drawing the region, the primary queue is scanned and all tiles that intersect with the rectangle corresponding to that ROI are transferred to the

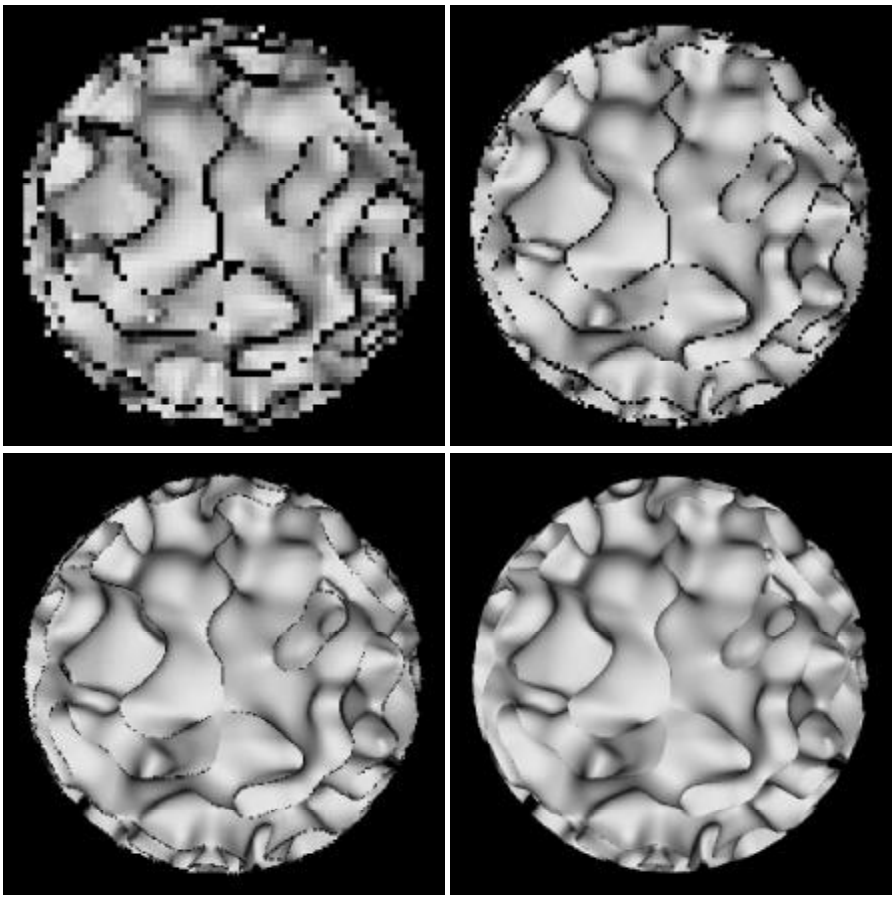


Figure 7.13: From left to right, top to bottom, snapshots taken during the progressive refinement rendering of a hypertextured surface that is not Lipschitz continuous. The snapshots were taken after 6936, 25 961, 99 627 and 810 904 iterations, respectively. The wall clock times at each snapshot are 1.95s, 4.11s, 11.43s and 77.12s, respectively.

secondary queue. Both versions of the progressive refinement renderer can then proceed as explained before with the difference that the secondary queue is now being used. Once this queue becomes empty, the portion of the image inside the ROI is fully rendered and both algorithms returns to subdividing the tiles that were left in the primary queue. It is also possible to cancel the ROI at any time by flushing any tiles still in the secondary queue back to the primary queue.

Figure 7.14 shows three images, with a resolution of 800 × 400 pixels, taken during the rendering of a procedural landscape. This surface is Lipschitz continuous and the progressive refinement algorithm of Section 7.2 was used. A region of interest, shown as a white rectangle, was used to focus the rendering on one of the terrain features. The top image shows the image at the moment when the ROI was specified. The rendering time for this image is 2m 15.09s. The middle image shows the surface at the moment when rendering inside the ROI was completed. The rendering time is now 4m 0.93s. The portion of the terrain inside the ROI has become fully resolved while the remaining terrain has not suffered any change since the previous image. The bottom image shows the final rendering, which is achieved after 14m 42.47s. The same landscape took 17m 13.26s to render with speculative sphere tracing. The top image demonstrates how an approximate and quite acceptable rendering of the surface is available early during the rendering, in this case after 15% of the total rendering time has elapsed. During the course of some editing task, the user may be interested in resolving only the portion of the terrain that is contained by the ROI. In that case, he will achieve the most accurate results with only 27% of the total rendering time. The time necessary to resolve the ROI depends on its size and also on its placement over the image. Portions of the image that are closer to the horizon take longer to resolve since cones must be traced over a greater distance. The example of Figure 7.14 also demonstrates the shrinking effect of progressive refinement for implicit surfaces that is explained in Section 7.2.2. The middle image illustrates how the horizon gradually lowers as the refinement progresses and the surface reaches its final shape.

7.5 Threaded Implementation

The best implementation strategy for the rendering methods presented in this chapter is to have several threads running concurrently: one refinement thread for each core of a multi-core CPU and a display thread. The threads communicate through an image buffer. The refinement threads require write access to the buffer as they continuously draw coloured squares onto it that correspond to the tiles that have just been updated. The display thread only requires read access to the same buffer. No mutual exclusion mechanisms need to be enforced because the rendering threads are working on disjoint tiles of the image so that no conflict exists between them. It may happen that the display thread will read some part of the image buffer in an incoherent state because of a simultaneous write by a refinement thread but any display errors that may occur will be erased during the next display refresh.

The display thread is controlled by a timer that ensures a constant frame rate. The thread remains in a sleep state except for the periodical invocation of the timer handler routine. The main function of this timer handler is to invoke a graphics library call that transfers the content of the image buffer, handled by the application, to the hardware frame buffer, handled by the machine's GPU. The display thread is also responsible for the interactive editing of regions

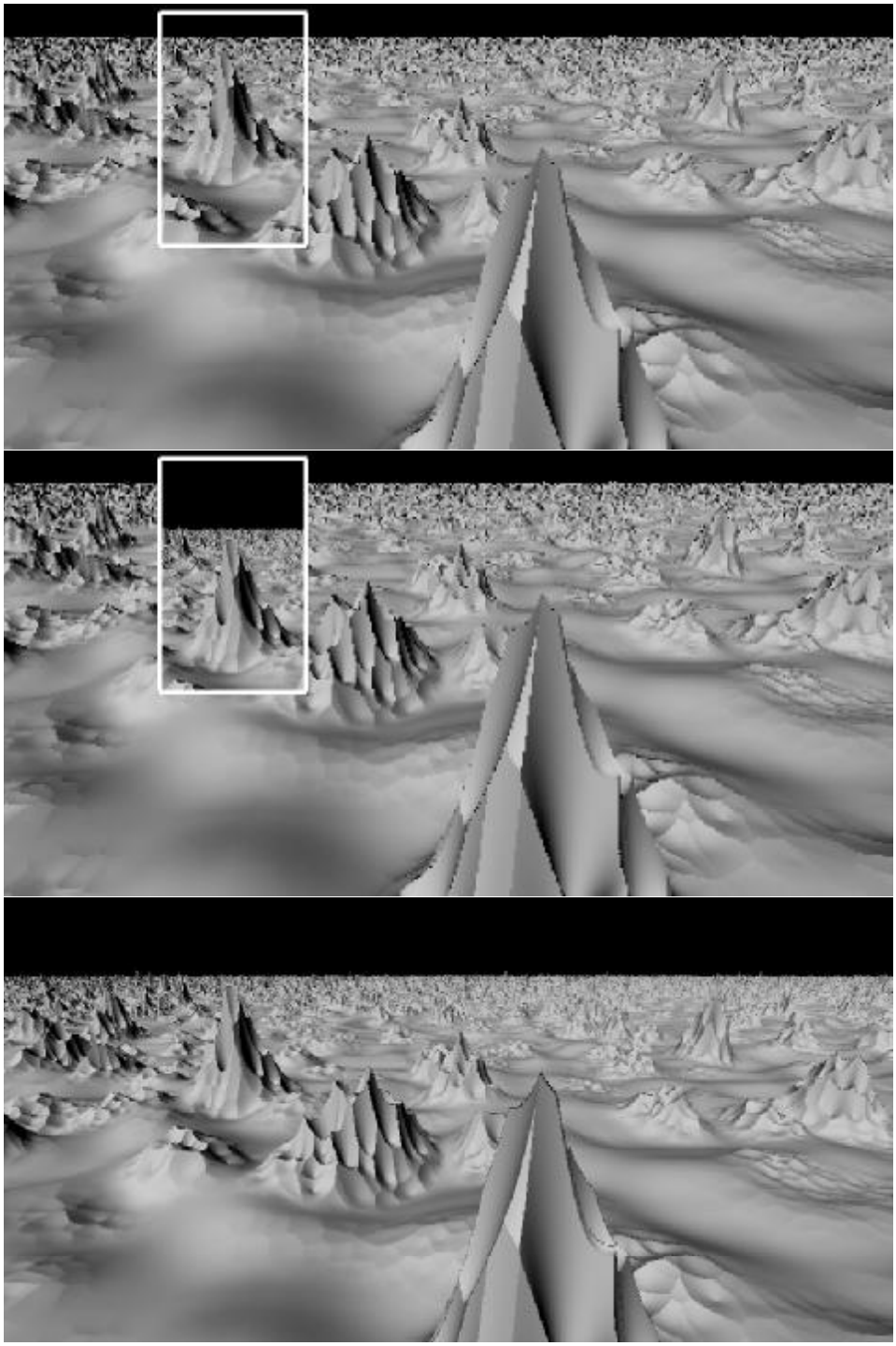


Figure 7.14: A progressive refinement rendering of a procedural landscape, illustrating the use of a region of interest.

of interest and notifying the refinement threads to the existence of such regions. The timing results reported in this chapter were obtained on an old single-core processor so that only one refinement thread was used. The reported rendering times should decrease linearly with the use of a dual-core or a quad-core processor. The rendering times for the ray casting algorithms of Chapter 5 also decrease linearly with the number of available cores, if a multi-threaded implementation is used, so that the comparative performance of the two algorithms, reported in this chapter, is maintained irrespective of the number of cores.

The contents of the application image buffer are transferred to a file at the completion of the progressive refinement algorithm. This happens either if the algorithm completed normally or was terminated early by the user. In the case of early termination, the most up to date results of the surface visualisation are stored in the file. This allows an approximate rendering of the surface to be stored for later analysis even in the case where not all of the surface features have been completely resolved.

7.6 Summary

The visualisation of implicit surfaces with progressive refinement offers the possibility of previewing the surface, as it is being rendered, with increasing accuracy. The user can exercise control over the renderer by specifying image regions that should receive priority. This rendering strategy offers significant advantages over conventional ray casting during the editing stages of a new and potentially complex implicit surface. Ray casting usually renders an image in scan line order, which constitutes a rigid rendering progression. The more flexible rendering approach offered by a progressive refinement previewer allows the user to make quick editing decisions about the surface. The previewer runs faster than straightforward ray casting for surfaces that are generated from complex implicit functions. It becomes slower than ray casting if the implicit function is simple and can be evaluated efficiently because of the overheads required for progressive refinement. This is not a limitation since simple functions that can be rendered quickly with ray casting do not really need a previewing facility.

Two progressive refinement algorithms have been devised. One of the algorithms works only with Lipschitz continuous surfaces and proceeds by tracing a sequence of unbounding spheres along the cone defined by a tile in the image. The other algorithm works by subdividing the view frustum into progressively smaller cells. A classification of the surface inside each cell is performed by computing an interval extension based on reduced affine arithmetic. The first algorithm is faster and should be used whenever possible. Most hypertextured procedural terrains are Lipschitz continuous and can be previewed with this algorithm. The second algorithm can be used for previewing implicit surfaces where the gradient vector may become infinite at some places. Besides the example shown in Section 7.3.3, examples of this type of surfaces include algebraic surfaces with long and thin spikes [Balsys et al., 2008].

The use of guaranteed Lipschitz bounds or reduced affine arithmetic estimates as part of the progressive refinement algorithm means that no surface details with sizes that are above the size of a pixel can possibly be missed. This is an improvement over more general image space progressive refinement algorithms where small surfaces can sometimes escape detection. Surface components that are smaller than a pixel may still be rendered incorrectly, considering

that image subdivision does not proceed below the pixel level. This problem can be solved with the implementation of an anti-aliasing technique.

Because the aim is to provide an interactive and progressive refinement previewer for implicit surfaces, anti-aliasing is not an issue. Anti-aliasing, however, can be easily added to the rendering algorithms that have been developed. Anti-aliasing is achieved by the process of filtering rectangular image tiles that have constant luminous intensity [Painter and Sloan, 1989]. All tiles must be subdivided down to a specified minimum size, which can be several times smaller than the size of a pixel. Let the area of a terminal tile s in image space be $[x_a, x_b] \times [y_a, y_b]$. Let also $(i + 0.5, j + 0.5)$ be the image coordinates of some pixel $\{i, j\}$. Given some appropriate anti-aliasing filter $h(x, y)$, the intensity $I_{i,j}$ of the pixel is:

$$I_{i,j} = \sum_{s \in S_{i,j}} I_s \int_{x_a}^{x_b} \int_{y_a}^{y_b} h(x - i - 0.5, y - j - 0.5)\, dydx, \tag{7.19}$$

where I_s is the intensity of tile s, assumed constant throughout its area, and $S_{i,j}$ is the set of all tiles whose areas overlap with the area of support of the anti-aliasing filter centred at the coordinates of pixel $\{i, j\}$. The double integral of the filter function h can be precalculated into a lookup table for increased efficiency. With this anti-aliasing technique, progressive refinement can be extended into much deeper levels of subdivision than before. This provides the opportunity for further performance gains relative to conventional ray casting, where many independent rays need to be shot for each pixel in order to implement anti-aliasing. If one is only interested in the final anti-aliased image, the evaluation of the shading function can be applied exclusively to the smallest image tiles, giving an additional speed up to the refinement.

CHAPTER 8

Topology Correction

COMPARED to displacement mapping or flow mapping, hypertexturing is a more flexible technique for adding geometric detail to an otherwise smooth implicit surface [Cook, 1984; Sclaroff and Pentland, 1991; Gamito and Musgrave, 2001]. Hypertextures can generate surface overhangs, which can also be produced with flow mapping but not with displacement mapping, and arches, which neither mapping technique can produce (refer to the discussion in Section 2.4). One drawback of the flexibility provided by hypertextures is that a surface can also become fragmented into several disconnected components. Depending on the context, this surface splitting effect may be desirable or not. If one is using implicit surfaces to model splashing fluids (in the form of level sets), for example, then the surface splitting effect is actually beneficial. If, on the other hand, one is trying to model a solid object with a complex surface structure, e.g. a rock, the disconnected state of the surface leads to physically implausible results.

This chapter presents a solution to the surface splitting effect of hypertextures that are defined with C^2 continuous functions. Disconnected surface components are detected in an initial connectivity analysis step and then removed during rendering. To achieve this goal, Morse theory is used to analyse the topology of the surface. The surface connectivity, in particular, can be completely determined by studying the critical points of the implicit function f and the way they are joined together. In this thesis, the process of removing disconnected components form a surface is named "topology correction". The technique is applied as part of the reduced affine arithmetic ray casting algorithm for hypertextured implicit surfaces, presented in Section 5.4.

A distinction is made between *global methods* and *local methods* for performing topology correction. Global methods must analyse the entire surface as a pre-processing step before they can determine the connectivity state of any arbitrary surface point. Local methods, by contrast, can perform a localised connectivity analysis for every point. A global method is easier to implement but can only work with relatively simple surfaces where the number of critical points is not too large [Gamito and Maddock, 2007e, 2008b]. A local method, on the other hand, can be applied to very large and complex surfaces [Gamito and Maddock, 2008a]. One example is a procedurally defined hypertextured terrain on the scale of an entire planet.

Section 8.1 demonstrates the surface splitting effect and shows why it is hard to control in a general way. Section 8.2 presents two simple methods that can be used to solve the surface connectivity problem and explains their limitations. Section 8.3 explains the necessary concepts from Morse theory that are required in Sections 8.4 and 8.5 where the global and local topology correction algorithms are presented, respectively. Section 8.6 summarises the chapter.

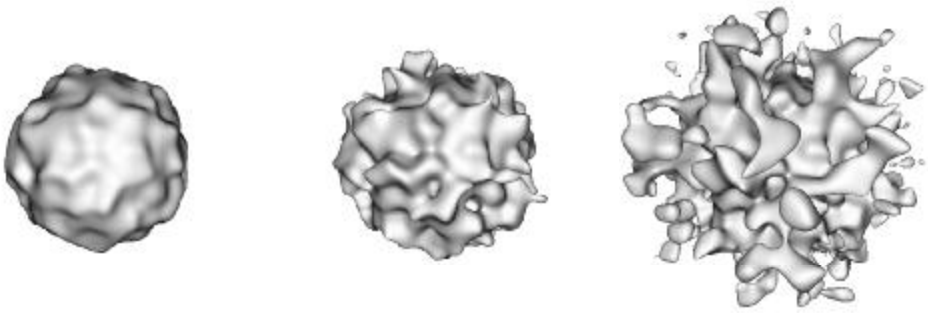


Figure 8.1: A sphere rendered with increasing amounts of hypertexture ($\epsilon = 0.1, 0.3$ and 0.8).

8.1 The Surface Splitting Effect

The splitting effect of hypertextured surfaces is illustrated with an example. Figure 8.1 shows three implicit surfaces that have been generated by adding increasing amounts of hypertexture. The function that generates these surfaces is:

$$f(f_0(\mathbf{x}), \mathbf{x}) \;=\; f_0(\mathbf{x}) \;+\; \varepsilon\, f_N(4\mathbf{x}), \tag{8.1}$$

which is similar to (7.10) with the exception that the amplitude ε of the hypertexture is variable. This amplitude takes values of 0.1, 0.3 and 0.8 for the three surfaces in Figure 8.1. The case $\varepsilon = 0.1$ shows a surface with a small amount of perturbation relative to the initially smooth sphere. This type of surface could more easily have been modelled as a procedural displacement map [Heidrich and Seidel, 1998]. The case $\varepsilon = 0.3$ generates an object with more pronounced surface features but which, from a topological point of view, is still homeomorphic to a sphere. The case $\varepsilon = 0.8$ generates an object with the interesting overhanging and arching features that only the implicit surface approach can give. At the same time, it also causes the surface to split, generating a cloud of small objects that are seen floating at fixed locations around the main object in the centre.

The splitting effect places an upper bound on the amount of hypertexture that can be added to an object while keeping it as a topologically connected set. It can occur for any hypertexture and not just the additive hypertexture given by function (8.1). The only exceptions consist of hypertextures that, by construction, are equivalent to displacement maps. The maximum amount of hypertexture depends on the particular function f that generates the surface and, without recourse to the Morse theory used in this chapter, can only be found by trial and error. For solid modelling purposes, one would often like to use stronger hypertexturing effects than those allowed if a surface is to remain connected.

8.2 Alternative Approaches

A simple but inaccurate way to perform topological correction on hypertextured surfaces is to employ a voxel grid, where the function $f(f_0(\mathbf{x}), \mathbf{x})$ is sampled at the corners of the voxels. A voxel is known to straddle the surface when the function changes sign at some of the voxel's

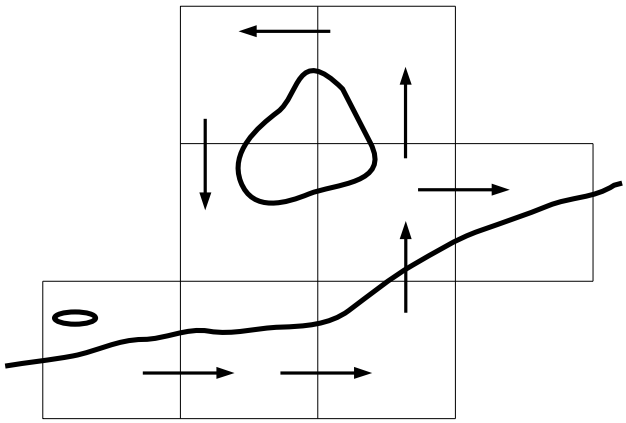


Figure 8.2: Two surface components incorrectly determined to be part of the main surface. The arrows show the region growing sequence, starting from the voxel on the bottom left.

eight corners. One can perform a discrete three dimensional region growing process to segment the voxel space into disjoint volumes, each enclosing a particular disconnected component of the surface. If the original data is already discrete, e.g. a series of MRI scans, then this is probably the best approach to take. When generating an isosurface from the volume data, this voxel-based method can be used to remove outlying surface components that may be the result of measurement error. This thesis is concerned with the topological correction of *procedurally defined surfaces*. Sampling the function $f(f_0(\mathbf{x}), \mathbf{x})$ onto a grid implies loss of information unless the function happens to be bandlimited and the sampling frequency is above the Nyquist limit. This loss of information leads to incorrect connectivity results, as shown in Figure 8.2, which can occur for surface components that are too small or too close to the main surface, relative to the sampling distance. Although the results of Figure 8.2 can be considered correct if one looks only at the function values stored on the grid nodes, they are clearly incorrect when one considers the implicit surface, generated by the continuous function $f(f_0(\mathbf{x}), \mathbf{x})$, before it was discretised onto the grid points.

Another possible approach is to first convert the implicit surface into a polygonal mesh before performing any topology correction. Stander and Hart [1997] have presented a meshing algorithm for implicit surfaces that is guaranteed to preserve surface topology. Once the polygonal mesh has been generated, one can perform region growing by jumping across the edges shared by neighbouring polygons to obtain a set of disjoint polygonal objects. One of these objects approximates the main implicit surface and the others represent the outliers that should be eliminated. One objection against this approach is that it cannot be used for direct rendering of implicit surfaces with ray casting – it is only meaningful for applications where implicit surfaces are converted to polygonal meshes and subsequently rendered on a GPU. Another objection is that it is wasteful of CPU cycles since it takes time to correctly polygonise surface components that are later found to be disconnected and which must then be removed. Topology correction should occur before the meshing process rather than after.

λ_1	λ_2	λ_3	**Type**
−	−	−	Maximum
−	−	+	2-saddle
−	+	+	1-saddle
+	+	+	Minimum

Table 8.1: Distinct types of critical points are determined by combinations of the signs of the eigenvalues of the Hessian matrix $\mathcal{H}\{f\}$.

8.3 Morse Theory

Morse theory studies the behaviour of functions over a manifold [Milnor, 1963]. The theory was first introduced to computer graphics by Shinagawa et al. [1991] and was later shown by Hart [1998] to be relevant for the topological study of implicit surfaces. When the theory is applied to implicit surfaces, the manifold becomes the entire $\mathbb{R}^3$ space and the function defined over this space is the function f that generates the surface. Central to the Morse theory is the notion of a *critical point* of f. A critical point $\mathbf{x}_C$ is such that:

$$\nabla f(f_0(\mathbf{x}_C), \mathbf{x}_C) = 0. \tag{8.2}$$

A critical point can be further classified by studying the eigenvalues of the Hessian matrix of f at $\mathbf{x}_C$. The Hessian matrix $\mathcal{H}\{f\}$ collects all the second partial derivatives of the function f:

$$\mathcal{H}\{f\} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_1 \partial x_3} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \frac{\partial^2 f}{\partial x_2 \partial x_3} \\ \frac{\partial^2 f}{\partial x_3 \partial x_1} & \frac{\partial^2 f}{\partial x_3 \partial x_2} & \frac{\partial^2 f}{\partial x_3^2} \end{bmatrix}. \tag{8.3}$$

If f is C^2 continuous then it is known that $\partial f^2/\partial x_i \partial x_j = \partial f^2/\partial x_j \partial x_i$ and the Hessian is symmetric. The spectral theorem of Hermitian matrices then guarantees that $\mathcal{H}\{f\}$ is diagonalisable and that all its three eigenvalues are real[1]. Depending on the signs of the eigenvalues λ_1, λ_2 and λ_3, sorted in increasing order, a critical point can be classified as shown in Table 8.1. The type of a critical point gives an indication of the topology of the surface around that point. For example, the minima occur near the local centroids of the surface while the 1-saddles occur at points where two surface components are joined together. In this chapter, attention needs to be given only to the minima and the 1-saddles in order to characterise the connectivity of the surface.

The case where one or more of the eigenvalues is zero leads to a *degenerate critical point*. Morse theory breaks down in these circumstances. However, degenerate critical points are unstable and can easily be removed by introducing a small perturbation in the parameters defining

[1]A real symmetric matrix is a special case of a Hermitian matrix.

the function. A function f that contains no degenerate critical points is then said to be a *Morse function*. Morse functions need to be C^2 continuous, considering that both first and second partial derivatives of f are required by the Morse theory. It is possible to relax this restriction and work with C^1 functions, provided that second derivatives are continuous at least over the critical points [Hart et al., 1998].

By taking the gradient ∇f, one obtains a vector flow field whose structure is intimately related to the topology of the implicit surface. From equation (8.2), the critical points of the surface are also the stagnation points of the flow field. A streamline of this field is a path that is obtained by descending along the local gradient vector, according to the ordinary differential equation:

$$\frac{d\mathbf{x}}{dt} = -\nabla f(f_0(\mathbf{x}), \mathbf{x}). \tag{8.4}$$

The minus sign in equation (8.4) causes the path of any streamline to descend towards a minimum. The gradient vector ∇f points towards increasing values of f, making the minus sign necessary. A streamline, starting at any point $\mathbf{x}$, can be followed by integrating (8.4) for $t \to +\infty$. A streamline is called a *separatrix* if it separates two regions of the flow with different characteristics [Helman and Hesselink, 1991]. Separatrices are important as they also give information about the topology of the surface. All the separatrices originate and terminate at minima of f. For every separatrix there is a 1-saddle somewhere along its path. The separatrix is locally tangent to the $\mathbf{v}_1$ eigenvector (associated with the λ_1 eigenvalue) at the 1-saddle.

Figure 8.3 shows a simple case of two implicit blobs connected as a single surface. There are two minima close to the centroids of each blob and a 1-saddle at the junction of the two blobs. The separatrix, in this simple case, is a straight line segment joining the two minima and passing through the 1-saddle. In a more general situation the separatrix would be curvilinear. Knowing the position $\mathbf{x}_S$ of the 1-saddle, it is possible to locate the two minima sharing this critical point by integrating equation (8.4) backwards and forwards from $\mathbf{x}_S$, following a direction that is initially coincident with the $\mathbf{v}_1$ eigenvector of the 1-saddle. It is also possible to determine the connectivity of the two blobs by checking the sign of $f(f_0(\mathbf{x}_S), \mathbf{x}_S)$. If this sign is negative, the blobs are connected and the separatrix is known to travel exclusively through the interior of the surface. If the sign is positive, the two blobs are disconnected and the separatrix must exit and enter the surface again at some points.

The separatrices defined by f form a network of lines that partition $\mathbb{R}^3$ into a topological entity called the *CW-complex* [Hart, 1999]. The CW-complex is a data structure that encodes all the topology of the implicit surface. It consists of a disjoint partitioning of the space into curved cells. The minima are located at the corners of these cells and the separatrices form the edges of the same cells. Connectivity information can be obtained by following only the network of separatrices that are interior to the surface. This process will partition the minima into a number of separate sets, which reflects the number of disconnected components of the surface.

Morse theory is increasingly finding use in computer graphics. Stander and Hart [1997] used it to convert implicit surfaces into polygonal meshes whilst guaranteeing that the topological features of the original surface were correctly represented in the mesh. Much work has also been done to use Morse theory in the simplification of polygonal meshes and three-dimensional volume data in a way that provides a graceful degradation of the topological features in those

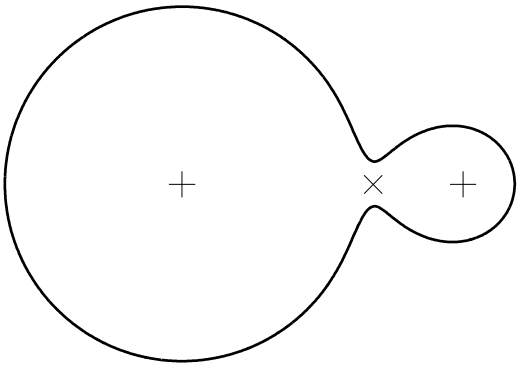


Figure 8.3: An implicit surface formed from two blobs. The "+" signs mark the two minima and the "×" sign marks the 1-saddle.

geometric representations. Morse theory is adapted to work with functions f that have a piecewise linear behaviour between the vertices of a triangular mesh (for polygonal meshes) or the vertices of a cubic or tetrahedral grid cell (for volumetric datasets). The topological analysis is somewhat simplified relative to the general case of a C^2 continuous Morse function because it is already known in advance that the critical points lie at the vertices of the 2D or 3D domain decomposition. Examples of this type of approach can be found in [Gyulassy et al., 2006; Danovaro et al., 2007] and references therein.

8.4 Global Topology Correction

The global method for correcting the topology of hypertextured implicit surfaces proceeds by identifying all disconnected components of the surface. Of all the components detected, the larger one is considered to be the main surface, which is rendered as part of the ray casting algorithm. The remaining surface components are ignored during ray-surface intersection tests. The detection of disconnected surface components proceeds in two steps:

1. Build a set of all minima and 1-saddles that are located inside the surface.
2. Segment the previous set into disjoint subsets by following the separatrices from the 1-saddles towards the minima.

Steps 1 and 2 are performed before any surface rendering occurs. The outcome of step 2 is a sequence of sets S_i, with $i = 1, 2, \ldots, N$, where each set contains all the minima that exist inside some particular component. The number N of sets is equal to the total number of surface components. One of these sets is the main set, corresponding to the main surface to be rendered. During a ray-surface intersection test, the set S_i that corresponds to the surface component to which the intersection point belongs is identified. If this is not the main set, the intersection point is ignored and another point is searched for further along the ray. The following sections describe the relevant steps of the topology correction method.

```
push bounding box V_0 onto stack;
while stack not empty
  pop voxel V from stack;
  let X_V = interval extent of V;
  if f(f_0(X_V), X_V) > 0
    continue;
  if ∇f(f_0(X_V), X_V) ∌ 0
    continue;
  let r = radius of bounding sphere for V;
  if r < ε
    Test V;
    continue;
  subdivide V;
  push children onto stack;
```

Figure 8.4: The `Subdivision` algorithm.

8.4.1 Locating Critical Points

Location of critical points is made by recursive subdivision of an initial bounding box that surrounds the surface. A technique is employed that was first proposed by Stander and Hart [1997] and later improved by Hart et al. [1998]. For every cubical voxel resulting from the subdivision, a series of tests is made to determine, first, if the voxel contains part of the surface and, second, if a critical point may be contained within it. If these tests pass, the voxel is subdivided and the children are tested in turn, down to a minimum specified voxel size.

Interval arithmetic is used to check if a voxel is part of any of the components of the surface [Moore, 1966; Snyder, 1992]. An interval estimate for the variation of f inside the voxel is used in the following test to determine if the voxel lies completely outside the surface:

$$f(f_0(\mathbf{X}_V), \mathbf{X}_V) \; > \; 0, \tag{8.5}$$

where $\mathbf{X}_V$ is an interval vector that spans the spatial extent of the voxel. Because interval arithmetic is a conservative range estimation technique, this test is always guaranteed to return a correct result for outside voxels.

A voxel is checked for the existence of critical points once it is known from test (8.5) that it may be either inside or straddling the surface. The test for the existence of critical points is achieved by obtaining an interval vector estimate of the function gradient, using again the $\mathbf{X}_V$ interval extent:

$$\nabla f(f_0(\mathbf{X}_V), \mathbf{X}_V) \; \ni \; \mathbf{0}. \tag{8.6}$$

If the null vector $\mathbf{0}$ is contained inside the interval vector for ∇f, there is the possibility that one or more critical points may be contained in the voxel. The voxel is then either subdivided or an explicit test is made for the presence of minima and 1-saddles, once a minimum voxel size has

been reached. Figure 8.4 shows in pseudo-code the `Subdivision` algorithm that implements the sequence of tests for each voxel. The voxels are kept in a stack, which is initialised with the bounding box V_0 for the object.

Due to the conservative properties of interval arithmetic, it often happens that voxels neighbouring a voxel that contains critical points are also incorrectly flagged by the interval arithmetic tests to contain such points. The `Test` routine, that is invoked in the listing of Figure 8.4, performs the final stage in the search for critical points, weeding out the false positives output by the interval tests. It is assumed at this stage that a voxel is small enough to contain only one critical point. In general, it is not possible to guarantee that the interval subdivision technique will isolate all individual critical points. Therefore, the minimum voxel size ϵ, in the algorithm of Figure 8.4, should be chosen as small as possible. In the implementation that was used for this thesis, a value $\epsilon = 10^{-8}$ was used. If an implicit function has two critical points that are nearer than 10^{-8}, it can be considered as almost degenerate and Morse Theory is very close to not yielding any useful results for the implicit surface.

Starting from the voxel centre $\mathbf{x}_V$, the following iteration sequence of a multi-dimensional Newton root finder is performed towards the critical point:

$$\begin{aligned} \mathcal{H}\{f\}\delta\mathbf{x}_i &= -\nabla f, \\ \mathbf{x}_{i+1} &= \mathbf{x}_i + \delta\mathbf{x}_i. \end{aligned} \tag{8.7}$$

Both the Hessian matrix $\mathcal{H}\{f\}$ and the gradient ∇f are evaluated at the point $\mathbf{x}_i$ to solve for $\delta\mathbf{x}_i$. The iteration is stopped if the sequence of points $\mathbf{x}_i$ goes outside the voxel. Otherwise, the sequence will converge to some point $\mathbf{x}_C$ inside the voxel where a critical point is known to exist. If the critical point is inside the surface such that $f(f_0(\mathbf{x}_C), \mathbf{x}_C) < 0$, and if it is a minimum or a 1-saddle (which is found after the eigenvalues of $\mathcal{H}\{f\}$ at $\mathbf{x}_C$ have been computed), the point is added to a set S of critical points interior to the surface. Each element in S stores the following information regarding a critical point:

- The position $\mathbf{x}_C$.
- The value $f(f_0(\mathbf{x}_C), \mathbf{x}_C)$, which must be negative.
- A flag indicating if $\mathbf{x}_C$ is a minimum or a 1-saddle.
- The eigenvector $\mathbf{v}_1$ if $\mathbf{x}_C$ is a 1-saddle.

When the `Subdivision` algorithm completes, the set S will contain the minima and the 1-saddles that were found inside every disconnected component of the surface.

8.4.2 Locating Disconnected Components

The set S is segmented into the sequence S_i, where each set S_i contains the minima for one surface component. The algorithm `Segmentation` is shown in Figure 8.5. Each critical point $\mathbf{x}_C$ of S is considered at a time, by increasing order of $f(f_0(\mathbf{x}_C), \mathbf{x}_C)$. If $\mathbf{x}_C$ is a minimum then a new set $S_i = \{\mathbf{x}_C\}$ is created. If, on the other hand, $\mathbf{x}_C$ is a 1-saddle, the two minima $\mathbf{x}_i$ and $\mathbf{x}_j$ connected to it are determined by integrating the separatrix backwards and forwards with

```
for every x_C ∈ S by increasing order of f(f_0(x_C), x_C)
  if x_C is a 1-saddle
    let x_i, x_j be the minima reached from x_C;
    let S_i ∋ x_i and S_j ∋ x_j;
    if S_i ≠ S_j
      create S_k = S_i ∪ S_j;
  else
    create S_i = {x_C};
```

Figure 8.5: The `Segmentation` algorithm.

equation (8.4), starting from $\mathbf{x}_C$ and going initially along the direction of the $\mathbf{v}_1$ eigenvector for the 1-saddle. Because the critical points are evaluated by increasing order of f, it is certain that by the time a 1-saddle is considered, the two minima $\mathbf{x}_i$ and $\mathbf{x}_j$ to which it connects will already have been processed by the algorithm. The sets S_i and S_j that contain $\mathbf{x}_i$ and $\mathbf{x}_j$, respectively, are then joined together to form a new set. The 1-saddle is ignored, however, if both $\mathbf{x}_i$ and $\mathbf{x}_j$ are found to be part of the same set already.

Once `Segmentation` completes, all disconnected surface components will have been identified through the S_i sets. The main surface is identified by the set S_m that contains the largest number of minima, where the index m is:

$$m = \max_i \#S_i. \tag{8.8}$$

This criterion for selecting the main surface that is to be rendered may fail for objects with an excessively large amount of hypertexture. If there is too much hypertexture, the object will break into a cloud of many smaller objects of approximately equal size. It is not clear in these conditions which of these smaller objects should be selected for rendering. The purpose in this thesis is to study hypertextured functions $f(f_0(\mathbf{x}), \mathbf{x})$ where the geometry of the original object $f_0(\mathbf{x})$ is still discernible after the hypertexture has been applied. The criterion (8.8) will then identify the correct surface component for rendering since the majority of the minima will be contained inside the main surface – only a smaller number of minima will exist outside the main surface, being responsible for the disconnected components.

8.4.3 Computing Ray Intersections

The computation of ray intersections with the implicit surface is performed with the affine arithmetic range estimation algorithm of Section 5.4. This affine arithmetic intersection algorithm, like all interval based intersection algorithms, is capable of finding every intersection point between the ray and the surface, sorted by increasing distance along the ray. The sphere tracing method of Section 5.2, by comparison, can only find the first intersection point with reliability. Once an intersection point $\mathbf{x}_I$ has been found along a ray, a test is performed to determine if it belongs to the main surface or not. To that effect, a streamline is followed with equation (8.4), starting from $\mathbf{x}_I$, which will converge towards some minimum $\mathbf{x}_M$ interior to the surface.

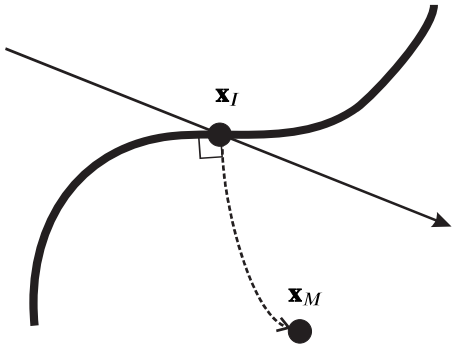


Figure 8.6: The intersection between a ray and the surface. The streamline originating at the intersection point is shown as a dotted line.

Figure 8.6 shows an example. The streamline starts off along a direction that is initially orthogonal to the implicit surface and converges towards the point $\mathbf{x}_M$. Having found the minimum $\mathbf{x}_M$, the set S_i to which it belongs is retrieved. If this is the main set S_m, the intersection point $\mathbf{x}_I$ is rendered, otherwise intersection testing continues along the ray to try to find another intersection point further along. Following every intersection that is found not to be part of the main surface, the connectivity test need not be performed again for the next intersection point, given that this will be the exit point of the ray from a disconnected component.

8.4.4 Tracking Streamlines

The path of a streamline needs to be tracked as part of the ray-surface intersection procedure of Section 8.4.3 and as part of the `Segmentation` algorithm of Section 8.4.2 where, in the latter case, the streamline is also a separatrix of the surface. Special care needs to be taken when performing this path tracking procedure because the endpoint of the streamline (and also the starting point, in the case of a separatrix) is a critical point where $\nabla f = 0$ occurs.

When tracking a separatrix, the path originates from a 1-saddle located at some point $\mathbf{x}_S$. If one were to integrate equation (8.4) with the initial condition $\mathbf{x}(0) = \mathbf{x}_S$, the path would never leave $\mathbf{x}_S$ since this is a stagnation point of the flow. To start off the integration from a 1-saddle, the following initial condition must be used instead:

$$\mathbf{x}(0) = \mathbf{x}_S \pm \epsilon \mathbf{v}_1, \tag{8.9}$$

where ϵ is a small displacement. The displacements of $\pm\epsilon$ along the $\mathbf{v}_1$ eigenvector will enable the integrator to move away from $\mathbf{x}_S$ and to converge towards the two minima that connect with the 1-saddle through the separatrix. The minima, however, are also stagnation points and path tracking would have to proceed from $t = 0$ up to $t = \pm\infty$ if the two minima were to be reached exactly. In practice, one proceeds with the integration for as long as possible and then finds the minima that are nearest to the points where the integrator left off.

The `lsodar` ordinary differential equation solver from the ODEPACK Fortran package is used to perform path integration [Hindmarsh, 1983]. The `lsodar` solver is able to select between a stiff and a non-stiff integration method, depending on the local conditions of the flow. When

given an upper limit of $+\infty$ or $-\infty$, `lsodar` inevitably finishes with an error status as it tries to get close to one of the minima. It also returns the farthest point $\mathbf{x}(t)$ that could be computed along the path. By controlling the numerical precision requested from `lsodar`, it is possible for $\mathbf{x}(t)$ to be as close to the correct minimum point as desired. A search is then performed among all the minima of all the S_i sets for the one that is closest to $\mathbf{x}(t)$, thus identifying the particular set S_i to which the separatrix has converged. The procedure is similar when tracking streamlines as part of the ray-surface intersection tests except that one is now only interested in following the path from $t = 0$ to $t = +\infty$ and the starting condition $\mathbf{x}(0) = \mathbf{x}_I$ is used, instead of equation (8.9).

Currently, the search for the minimum point nearest to $\mathbf{x}(t)$ is performed exhaustively by computing the squared distance to every possible minimum. This search method has linear time complexity and can become slow for a surface with a large number of minima inside. Although it has not been implemented for this thesis, it is possible to perform the search for a minimum in average logarithmic time with the help of a *k-d tree* [Friedman et al., 1977; Sproull, 1991].

8.4.5 Results

The application of the topology correction algorithm is demonstrated with hypertextures that are generated from scaled sums of a basis procedural noise function. The hypertexture function is:

$$f(f_0(\mathbf{x}), \mathbf{x}) \;=\; f_0(\mathbf{x}) \;+\; 0.8 \sum_{i=0}^{L-1} 2^{-0.8i} f_N(2^{i+2}\mathbf{x}). \tag{8.10}$$

The function f_0 generates a sphere of unit radius, as in the example of Figure 8.1, and f_N is a sparse convolution noise function (Section 4.4). The summation in (8.10) models a fractional Brownian motion process with a Hurst parameter given by $H = 0.8$ (Section 3.2.4). The number of layers of noise that are added to the sphere is given by L. As this number increases, the surface of the sphere becomes increasingly more irregular and, in the limit, attains a fractal dimension of $3 - H = 2.2$.

Figure 8.7 shows the network of separatrices for a hypertextured object computed from equation (8.10), with $L = 1$, after the `Subdivision` and `Segmentation` algorithms have been applied. The network is shown superimposed over an image of the object. This network represents a partial visualisation of the CW-complex for the object's surface since only the separatrices that are inside the surface are shown. Minimum points are also shown as dots and are located at the endpoints of one or more separatrices. Several of these points, however, are isolated and correspond to small disconnected surface components that can be seen surrounding the main surface.

Table 8.2 lists the number of minima, 1-saddles and disconnected components of the surface as the number of noise layers increases. These numbers follow a roughly geometrical progression with L, which causes the `Subdivision` algorithm to become increasingly less efficient as it needs to identify an ever denser cloud of critical points. The application of the topology correction method to a fractal hypertexture is, therefore, impractical since a surface needs to have five or more layers of noise to become recognisably fractal. These results are confirmed by the computation times that are shown in Table 8.3 for the same values of $L = 1, 2, 3$. The

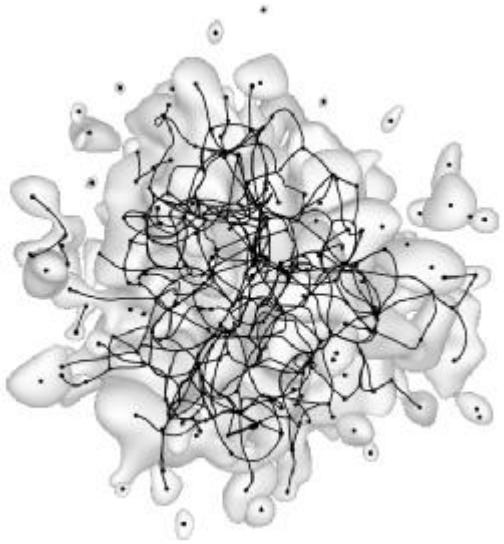


Figure 8.7: The network of separatrices and minima interior to a hypertextured surface.

L	Minima	1-saddles	Components
1	214	304	58
2	1006	1585	182
3	4567	8408	488

Table 8.2: Statistics for a hypertextured sphere with an increasing number of layers of noise.

L	`Subdivision`	`Segmentation`	Ray Casting
1	4m 12s	1s	3m 00s
2	88m 51s	13s	9m 21s
3	1016m 21s	1m 31s	17m 49s

Table 8.3: Times for global topology correction and rendering of a hypertextured sphere with an increasing number of layers of noise.

table discriminates between the time taken to isolate the critical points (corresponding to the `Subdivision` algorithm), the time taken to link the critical points (corresponding to the `Segmentation` algorithm) and the time taken to ray cast the surfaces. It is clear that the time required to locate critical points quickly dominates as L increases. Figure 8.8 shows the cases $L = 1$ and $L = 3$ of the hypertexture generated from equation (8.10). The original surface is first shown, without any topological correction. The disconnected components are then identified and visualised in red. Finally, the same disconnected components are ignored during the ray-surface intersection procedure.

A more efficient method than spatial subdivision for the localisation of critical points was proposed for implicit surfaces that are made from sums of radial basis functions by Wu and de Gomensoro Malheiros [1999]. With their method, simple heuristics are used to estimate the position of the critical points. The application of several relaxation steps then causes the critical points to converge towards their correct positions. Sparse convolution noise is an ex-

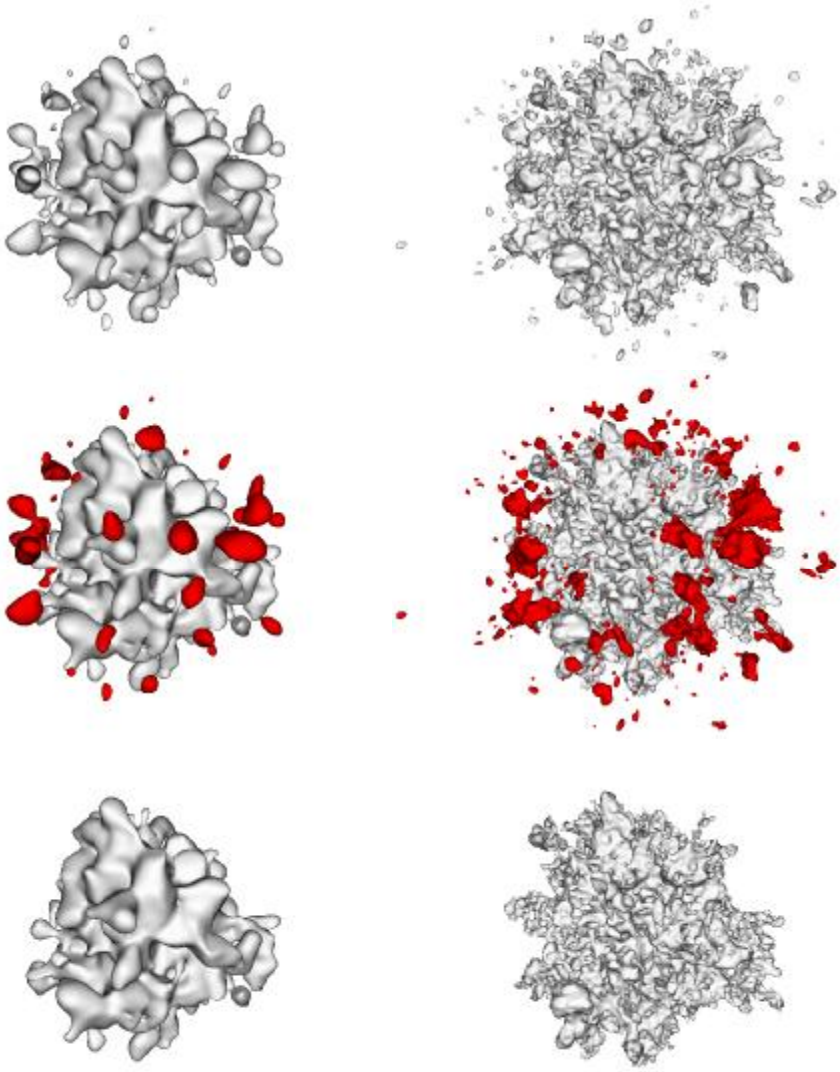


Figure 8.8: A hypertextured sphere with one layer (left) and three layers (right) of a sparse convolution noise function. Top row shows original surfaces. Middle row shows disconnected components in red. Bottom row shows surfaces after topological correction.

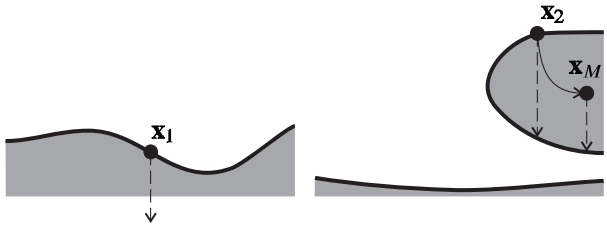


Figure 8.9: A quick connectivity test. The point $\mathbf{x}_1$ is connected. The point $\mathbf{x}_2$ has unknown connectivity.

ample of a hypertexturing function that could use the improved localisation method by Wu and de Gomensoro Malheiros since it consists of the sum of an infinite number of radial basis functions that follow a Poisson distribution in space. The same method, however, cannot be applied to Perlin noise functions. For that reason, a critical point localisation method based on spatial subdivision has been adopted, which, although being less efficient, is quite general and can be applied to any C^2 or even C^1 function. Spatial subdivision is also an easily parallelisable algorithm where disjoint regions of space can be assigned to different processors.

As mentioned in Section 8.4.1, a minimum voxel size $\epsilon = 10^{-8}$ was used as part of the `Subdivision` algorithm to obtain the results shown in Figure 8.8. The iterations (8.7) for the multi-dimensional Newton root finder were stopped when $\|\mathbf{x}_{i+1} - \mathbf{x}_i\| < 10^{-12}$. The numerical precision requested from the `lsodar` ODE solver was also equal to 10^{-12}. After determining the connectivity information, the component sets S_i, with $i = 1, \ldots N$, were stored to a file so that they could be reused for different renderings of the same surface. This is especially helpful when performing computer animation as the `Subdivision` and the `Segmentation` algorithms need to be run only once for each surface.

8.5 Local Topology Correction

The local topology correction method, by contrast with the method of Section 8.4, integrates the finding and linking of critical points into the ray casting procedure. Critical points are found on demand and only inside a small neighbourhood centred at the current ray-surface intersection point. The size of the neighbourhood is progressively enlarged, and more critical points are located, until a definite answer can be given about the connectivity state of the intersection point. Critical points are then cached and reused for nearby ray intersection points on the surface. Local topology correction was developed to handle procedural terrains over very large distances. In what follows, the local topology correction method is explained in the context of procedural terrains, defined by hypertexturing either a plane (in the case of infinite planar terrains) or a sphere (in the case of a procedural planet).

The connectivity testing algorithm is a boolean function that receives an arbitrary point $\mathbf{x}$, assumed to be on the surface of the terrain, and returns a result of true if $\mathbf{x}$ is part of the ground or false if it is part of a disconnected component. The algorithm begins by performing a quick connectivity test, as illustrated in Figure 8.9, which consists of tracing a ray from $\mathbf{x}$ towards the interior of the terrain and checking if it results in any intersection. For a terrain defined over a

```
if x is connected return true;
find minimum x_M reached from x;
if x_M is connected return true;
initialise list of active minima L_M with x_M;
initialise list of active voxels L_V with voxel V(x_M);
while L_V not empty
   remove current voxel V with the lowest altitude;
   find all minima and 1-saddles of V;
   while any 1-saddles of V link with active minima
      remove a 1-saddle linking with an active minimum
      find minimum x_M at opposite end of link;
      if x_M ∉ L_M
         if x_M is connected return true;
         add x_M to L_M;
         add voxel V(x_M) to L_V if V(x_M) ≠ V_0;
   for every voxel V_i ∉ L_V that is a neighbour of V
      if terrain component continues into V_i
         add V_i to L_V;
return false;
```

Figure 8.10: The local connectivity testing algorithm. The input is an arbitrary point $\mathbf{x}$ on the surface of the terrain.

plane, the ray follows a vertical direction and, for a terrain defined over a sphere, it follows the direction towards the centre of the sphere. If no intersection is found, the point $\mathbf{x}$ is known to be part of the ground and the algorithm returns, otherwise more expensive connectivity tests need to be performed. In preparation for those tests, the minimum $\mathbf{x}_M$ that is closest to $\mathbf{x}$ is found by integrating (8.4). A minimum is tagged as connected or disconnected by tracing another interior ray, as shown on the right side of Figure 8.9. If there is an intersection, the minimum is tentatively classified as disconnected although this status may change in the course of the algorithm, otherwise the minimum is connected and the algorithm again returns early with a connectivity status of true.

The three-dimensional space is decomposed into a partition of cubic voxels and the search for critical points is performed within each voxel, as explained in Section 8.4.1. The lateral size of the voxels is given by the characteristic length of the hypertexture. This characteristic length is the average distance between the smallest features of the hypertexture. The algorithm keeps a list of active voxels and a list of active minima. At this point, the list of minima is initialised to the single minimum $\mathbf{x}_M$ and the list of voxels is initialised to the voxel that contains this minimum. Figure 8.10 shows the main loop of the algorithm in pseudo-code.

The algorithm proceeds by removing one voxel at a time from the list of active voxels. The active voxels are kept sorted by their vertical coordinates, for planar terrains, or by their distance to the centre of a spherical terrain, for planets. The voxel with the lowest altitude is removed

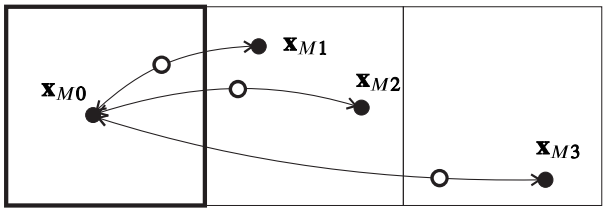


Figure 8.11: Examples of linking. The current voxel is on the left. Black circles are minima. White circles are 1-saddles.

at the start of each iteration and all the critical points contained in the voxel are located. The algorithm then iterates repeatedly over the 1-saddles found inside the voxel. When a 1-saddle links to an active minimum, the minimum at the opposite end of the link, if new, is added to the list of active minima and the 1-saddle is discarded (this is illustrated in Figure 8.11 for minimum $\mathbf{x}_{M1}$). The voxel that contains the new minimum is added to the list of active voxels if it is different from the current voxel. The algorithm keeps iterating over the 1-saddles for the current voxel until no more 1-saddles can be linked. Any remaining 1-saddles of the voxel are then ignored since they do not provide any further links to active minima. If, at any time, a 1-saddle links to a new minimum that is tagged as connected, the algorithm terminates by classifying all the active minima as connected and returning a connectivity status of true.

One difficulty of the localised topology correction algorithm is that a 1-saddle, linking an active minimum to a new minimum, may reside in a voxel different from the one that is currently being examined. This is illustrated in Figure 8.11 for the minima $\mathbf{x}_{M2}$ and $\mathbf{x}_{M3}$. Iterating over the 1-saddles of the current voxel, as previously explained, will not be able to find these new minima. If no other 1-saddles inside any of the active voxels link with $\mathbf{x}_{M2}$ and $\mathbf{x}_{M3}$ those critical points will not be considered, leading to possible incorrect results. The problem is solved by checking all the neighbours of the current voxel that are not already in the active list. If the terrain component that is being examined continues into any of the non-active neighbouring voxels (see Figure 8.12) those voxels are added to the list of active voxels. This is further explained in Section 8.5.1.

The search is exhausted when no more active voxels remain in the list. All the active minima are then known to belong to a disconnected component and the algorithm finally returns with a connectivity status of false.

8.5.1 Checking Neighbouring Voxels

Before completing the processing of the current voxel in the algorithm of Figure 8.10, it is necessary to add those neighbouring voxels that share part of the terrain into the list of active voxels. It is only necessary to consider voxels that share the same terrain component of point $\mathbf{x}$, for which the connectivity test was invoked. Figure 8.12 shows an example – the voxel on the left needs to be added to the list but the voxel on the right needn't. One can detect if the terrain component continues into any of the 26 neighbouring voxels by checking for the existence of minima of f restricted to the subdomains consisting of the 2D faces, 1D edges and 0D corner points of the current voxel.

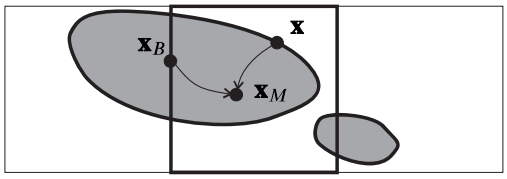


Figure 8.12: Example of edge neighbour checking. The current voxel is in the centre. The algorithm is determining the connectivity of point **x** on the larger component.

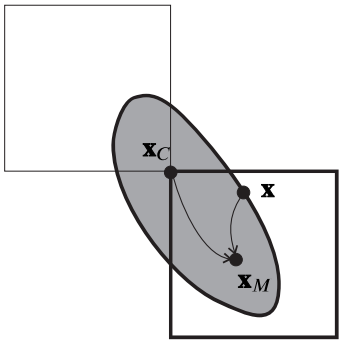


Figure 8.13: Example of corner neighbour checking. The current voxel is in the lower right. The other voxels sharing the corner point $\mathbf{x}_C$ are treated as in Figure 8.12.

In the case of face neighbours, a recursive subdivision of the boundary face is performed, followed by a two-dimensional Newton root finder to locate the minima. This is analogous to what is described in Section 8.4.1 for the three-dimensional case. The Hessian matrix of f restricted to the face of a voxel has a dimension of 2×2 and a minimum is a critical point where the two eigenvalues λ_1 and λ_2 of the Hessian are both positive. The situation is similar for edge neighbours with subdivision and root finding working over one dimension. Along the edge of a voxel, the Hessian matrix degenerates into a single partial derivative $\partial^2 f / \partial x_i^2$ for i equal to 1, 2 or 3, depending on which coordinate axis the edge is parallel to. A minimum along the edge is then characterised by $\partial f / \partial x_i = 0$, which is detected by the Newton root finder, and $\partial^2 f / \partial x_i^2 > 0$. For corner neighbours one only needs to verify that the corner point $\mathbf{x}_C$ is inside the terrain by checking the condition $f(f_0(\mathbf{x}_C), \mathbf{x}_C) < 0$. This is illustrated in Figure 8.13. Corner neighbours whose corner point, relative to the current voxel, is outside the terrain are not considered.

For each minimum that is found along one of the borders between the current voxel and a neighbour, it is necessary to check if this boundary minimum belongs to the terrain component being examined. For that purpose, one tracks a streamline starting from the boundary minimum and checks if it converges towards one of the active minima. Figure 8.12 shows an example where $\mathbf{x}_B$ is a boundary minimum relative to the left neighbour voxel and $\mathbf{x}_M$ is one of the active minima for the terrain component. The path of the streamline is followed by integrating (8.4). If the streamline terminates at one of the active minima of the component then the neighbour voxel also contains the component. At this point, no further minima along the common border between the two voxels needs to be considered and the neighbour can be added to the list of active voxels. If no boundary minimum links with active minima then the terrain that is shared

between the two voxels is not part of the terrain component being examined for connectivity. The situation is similar for corner neighbours (Figure 8.13) where a streamline is followed from the corner point $\mathbf{x}_C$ to see if it terminates at an active minimum.

When going from the current voxel V to a neighbouring voxel V_i through the boundary minimum $\mathbf{x}_B$, it is necessary to record the triple $(V, V_i, \mathbf{x}_B)$. If V is revisited during a later iteration of the algorithm (by following new links that may have since been discovered), the transition to V_i via $\mathbf{x}_B$ will not be performed again if it has been done before. This is to prevent the possibility of the algorithm becoming locked in an infinite loop. A transition from V to V_i is still possible if done through some other boundary minimum different from $\mathbf{x}_B$. In the case of corner neighbours, it is only necessary to record the pair (V, V_i) since the corner point $\mathbf{x}_C$ is unique. The transition from a voxel to one of its corner neighbours, therefore, can only be done once for each invocation of the connectivity test algorithm.

8.5.2 Caching

Caching is essential for the efficiency of the local topology correction algorithm. Without caching many of the operations previously described would be needlessly repeated for points $\mathbf{x}$ on the surface of the terrain that happened to be in close proximity. There are several levels of cache that can be implemented: caching of minima, caching of critical points per voxel and caching of boundary minima per face or edge.

Active minima are sent to the cache whenever their connectivity status becomes known. This happens when the algorithm of Figure 8.10 returns with a connectivity result of either true or false. When the algorithm links to a new minimum, it always checks first to see if the new minimum is in the cache and returns early if true. The return value is the connectivity status of the new minimum in the cache.

All the voxels that have been processed in previous invocations of the algorithm are also kept in another cache together with their minima and 1-saddles. This avoids having to locate again the critical points for those voxels with recursive subdivision. The algorithm merely retrieves from the cache the list of minima and 1-saddles of a voxel that was previously computed as part of the connectivity test for a different point on the terrain. A similar caching mechanism is implemented for the boundary minima of the faces and the edges of the voxel space decomposition.

Caching is also useful for computer animations where the camera moves over the terrain. The caches can be kept when rendering the consecutive frames of an animation. To prevent the caches from growing without bounds, however, it is necessary to remove those cache elements that have not been used for some specified number of frames.

8.5.3 Results

The local topology correction algorithm was validated with the implicit function (8.10) that was previously used to produce the results of Section 8.4.5, concerning the global topology correction algorithm. The same rendering results of Figure 8.8 were replicated by the local algorithm. Table 8.4 compares the total rendering times for the implicit function with L =

L	Global	Local
1	7m 15s	30m 51s
2	98m 25s	137m 59s
3	1035m 41s	576m 22s

Table 8.4: Comparison of the times for rendering with global and local topology correction of a hypertextured sphere with an increasing number of layers of noise.

1, 2, 3. The rendering times for the global algorithm are the total of the times, from Table 8.3, for critical point location and linking and for ray casting. The local algorithm is less efficient than the global one for relatively simple surfaces (the cases $L = 1$ and $L = 2$) because it needs to perform additional tests, such as the location of boundary minima on the faces and edges of the voxels, which the global algorithm does not require. The local algorithm, on the other hand, scales better with increasing surface complexity, as exemplified by the case $L = 3$. This is ultimately because, for the local algorithm, only a small percentage of the total number of critical points is necessary to determine the connectivity of the visible surface points. The global algorithm, by contrast, requires that all critical points be computed beforehand, whether they are necessary to determine surface connectivity or not.

Figure 8.14 is a rendering of the surface of a procedural planet with overhangs and arches, modelled as a hypertextured implicit surface. Although the terrain appears to be defined over a flat surface, it is actually a sphere seen from a very close range. The atmosphere is generated with a model of Rayleigh scattering that considers only single-scattering events between photons and air molecules [Nishita et al., 1993]. The hypertexture combines two procedural noise functions:

$$f(f_0(\mathbf{x}), \mathbf{x}) = f_0(\mathbf{x}) + 2 \cdot 10^{-4} f_G(4 \cdot 10^4 \mathbf{x}) f_S^2(10^4 \mathbf{x}/\|\mathbf{x}\|). \tag{8.11}$$

A gradient noise function $f_G(\mathbf{x})$ (Section 4.3) provides the basic terrain pattern and is then modulated by a squared sparse convolution noise function $f_S(\mathbf{x})$ (Section 4.4) to create the appearance of rocky outcrops over an otherwise flat terrain. The modulating noise is expressed in the form $f_S(\mathbf{x}/\|\mathbf{x}\|)$ so that it is made to depend only on the position over the surface of the planet but not on the height above the same surface.

The detection of surface connectivity is shown in the middle image of Figure 8.14 with the disconnected surface components coloured in green. The topology correction is then shown in the bottom image. It is possible to see that the shadows cast on the ground by disconnected components, which are visible in the lower left corner of the top and middle images, have disappeared in the bottom image due to those surface components having been removed. This effect is achieved by performing connectivity testing for shadow rays, similar to what is done for view rays. Disconnected components are ignored for shadow rays and a point is only in shadow if its shadow ray intersects with the ground terrain. The images in Figure 8.14 have a resolution of 1368×828 and took, from top to bottom, 354 minutes, 516 minutes and 685 minutes to render on a Pentium4 2.8GHz. These numbers give an overhead of 46% for connectivity testing relative to pure ray casting and a 94% overhead for full topology correction. It should be remarked that the results of Figure 8.14 could never have been obtained with the global topology correction algorithm of Section 8.4 because the computation of the entire set of critical points inside the procedural planet is simply intractable.

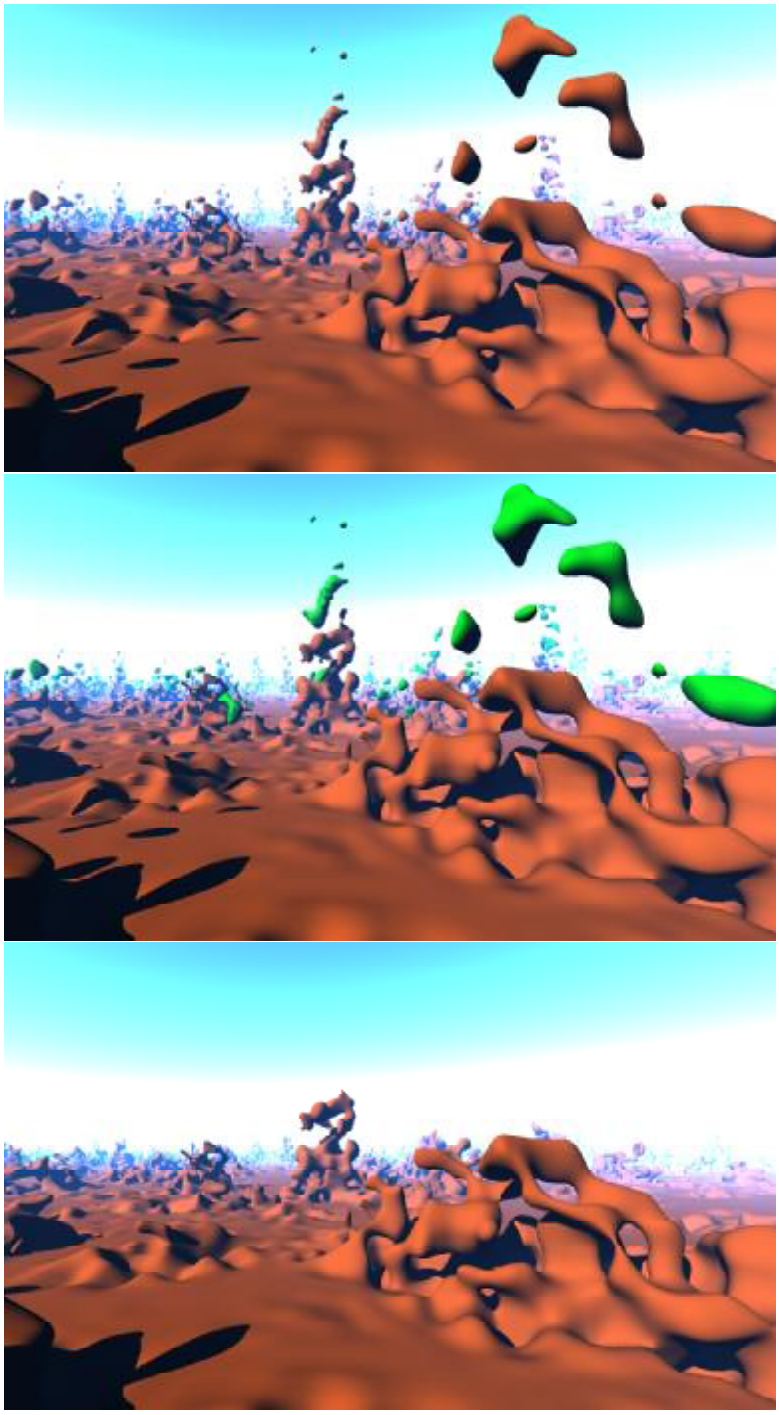


Figure 8.14: A hypertextured planet featuring terrain overhangs and arches. The top image shows the original surface. The middle image shows disconnected components in green. The bottom image shows the topological correction.

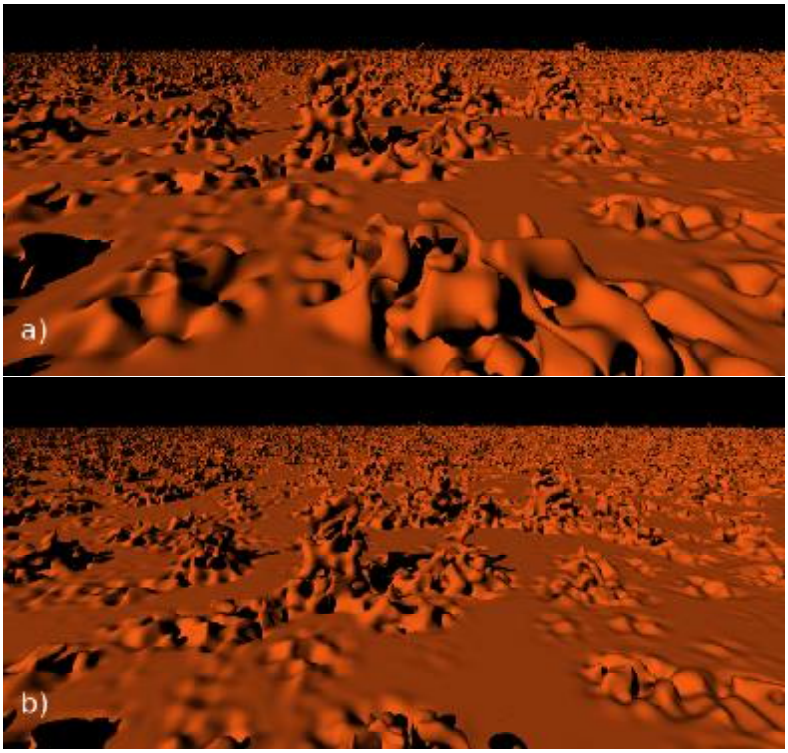


Figure 8.15: The same terrain of Figure 8.14 without atmosphere and seen from two higher altitudes.

Figure 8.15 shows the same terrain as Figure 8.14 with the viewpoint placed at two successively higher altitudes. The atmosphere has been removed so as to see more of the terrain over long distances. Table 8.5 gives some statistics relative to the images of Figures 8.14 and 8.15. The statistics shown are the number of minima (# Minima) and 1-saddles (# Saddles) located, the number of minima placed in the cache (Cache Size), the cache hit rate for the minima (Cache Hits), the percentage of success of the initial quick connectivity test (Quick Test), the percentage of time spent in ray tracing (Tracing), which includes the time spent in tracing shadow and connectivity test rays, the percentage of time spent in integrating streamlines (Streaming) and the percentage of time spent locating critical points with recursive subdivision (Locating).

Table 8.5 shows that the statistics of the local topology correction algorithm generally degrade as the altitude of the viewpoint increases and a larger area of the terrain becomes visible. The number of critical points located by the algorithm increases accordingly. The number of cached minima is generally smaller than the number of located minima because, when locating critical points inside a voxel, some minima belong to terrain components that are never tested. These terrain components are missed by the ray caster due to the finite sampling density. The performance of the cache also decreases as more disconnected components need to be removed. The performance of the quick connectivity test, used at the beginning of the algorithm, is the only statistic that improves as there is more flat terrain visible on the lower part of the images with increasing altitude.

	# Minima	# Saddles	Cache Size	Cache Hits
Fig. 8.14	5901	5742	5423	98.67%
Fig. 8.15a)	20534	21242	18650	95.38%
Fig. 8.15b)	31607	33440	28611	90.57%
	Quick Test	Tracing	Streaming	Locating
Fig. 8.14	58.24%	38.12%	25.03%	36.85%
Fig. 8.15a)	59.30%	20.57%	14.02%	65.42%
Fig. 8.15b)	68.34%	14.57%	9.12%	76.31%

Table 8.5: Several statistics for the images of Figures 8.14 and 8.15. Refer to the text for the meaning of these statistics.

Even with the high hit rates for the cache, shown in Table 8.5, the location of critical points can easily become the bottleneck of the local topology correction algorithm. As the number of critical points increases, the time spent in this operation becomes larger than the time spent in either tracing rays or following streamlines. Similarly to the other statistics, this imbalance becomes more pronounced as a larger area of the terrain becomes visible.

8.6 Summary

Morse theory provides all the connectivity information about an implicit surface that is necessary to determine how many components it is split into. This property of Morse theory finds application in the hypertexturing of implicit surfaces as it enables disconnected components other than the desired main surface to be detected and removed during rendering. In this way, one can add much greater amounts of hypertexture than previously possible to a solid object without the inconvenience of fracturing it into many smaller objects. The topology correction technique can be applied to C^2 continuous hypertextured surfaces. In the most general situation, the technique can be applied to any C^2 continuous implicit surface whenever it may be desirable to identify and isolate disconnected components of the surface.

The topological correction method is robust and will detect any disconnected component, no matter how small or how close it may be to the main surface. This robustness is again a consequence of the application of Morse theory. The accuracy of the method is only limited by the numerical tolerance factors and threshold values that are chosen for the algorithms described in this chapter. The values used are equal to or smaller than 10^{-8}, giving the topology correction method an overall accuracy similar to that of single precision floating point arithmetic.

Global and local versions of the topology correction algorithm have been presented. The global version can be used for hypertextured implicit surfaces where the number of critical points is not too large. Otherwise, the local version of the topology correction algorithm can be used instead, as this has a greater ability to handle surfaces with a very large number of critical points. An efficient caching scheme is also used by the local topology correction algorithm to minimise the number of connectivity analysis steps that need to be taken.

The main bottleneck of the algorithm, in both its global and local versions, is the time required to locate all the surface's critical points. Unlike the case of Morse theory applied over

polygonal meshes where the location of the critical points is easy to determine, the location of critical points inside a procedurally defined C^2 continuous implicit surface requires an expensive recursive subdivision algorithm, coupled with an interval arithmetic range estimation procedure. This algorithmic bottleneck becomes more evident for procedurally defined terrains when visualised from a very high altitude. For a high altitude camera, the visible area of the terrain is quite large, making the number of critical points grow considerably and also decreasing the efficiency of the cache. More research is needed to overcome this problem in the future. One possibility is to replace the interval arithmetic range estimation with a reduced affine arithmetic range estimation (Section 5.3.3), as part of the search for critical points, to see if it generates any speed improvements. Another possibility is to investigate critical point location methods that are tailored to specific types of implicit functions, along the lines of [Wu and de Gomensoro Malheiros, 1999]. Nevertheless, the topology correction methods presented in this chapter are general and robust. They also have the ability to exploit the growing parallelism that exists in modern processor architectures, allowing the critical point location stage of the algorithms to be performed in parallel over several spatially distinct volumes.

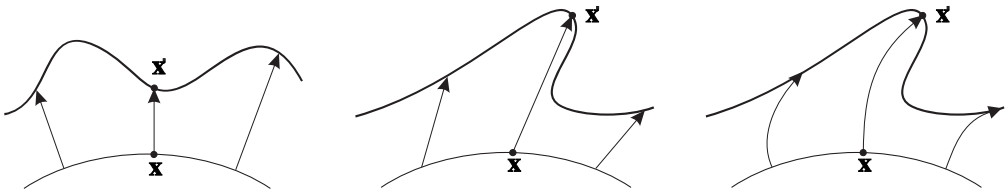
CHAPTER 9

Surface Deformation with Flow Mapping

As seen in the previous chapter, the number of critical points that need to be detected during the topology correction stage imposes a practical limit on the amount of detail of a surface. A finely detailed surface can have many potential points of connection between different surface components. A topology correction algorithm then needs to examine exhaustively all possible connections, by detecting minimum-saddle-minimum links, in order to give a conclusive answer about the connectivity of any point on the surface. The results of Table 8.2 indicate that the number of critical points (which include minima and saddles) grows geometrically with the number of layers of procedural noise that are added to the hypertextured surface.

To make the generation of topologically correct surfaces with high level of detail more feasible, a surface deformation technique is developed that allows small scale detail to be added on top of an implicit surface. The initial surface should be sufficiently smooth for the topology correction algorithm to complete in a reasonable amount of time. An example of such an initial surface is the procedural terrain shown in Figures 8.14 and 8.15. The surface deformation technique then adds extra detail to compensate for the smoothness of the initial surface. Both the surface deformation and the topology correction algorithms are integrated in a ray casting procedure for rendering implicit surfaces. The surface deformation results from the advection of the implicit surface through a procedurally defined vector flow field, according to a technique called *flow mapping*. The results of the topology correction algorithm are not invalidated by the application of surface deformation for any continuous flow field that may be specified. In other words, if the surface is connected, it will remain so after additional detail has been introduced through flow mapping.

The flow mapping technique explained in this chapter follows from an earlier study where it was suggested that flow fields could be used to introduce overhangs into procedurally defined terrains [Gamito and Musgrave, 2001]. This chapter improves on the technique presented by Gamito and Musgrave [2001] with the development of a better intersection algorithm between rays and deformed surfaces. This chapter also gives the proof, which was missing in Gamito and Musgrave [2001], that the proposed technique does not induce topology changes. Section 9.1 presents techniques that have been developed for surface deformation and puts them in context with flow mapping. Section 9.2 presents flow mapping as an example of a domain deformation modelling technique for implicit surfaces (Section 2.3.3). It also explains that flow mapping can be used to define a homeomorphism between two implicit surfaces, thereby guaranteeing that they are topologically equivalent. Section 9.3 develops a ray casting algorithm for implicit surfaces that takes into account the effects of flow mapping and shows results. Unlike



(a) Displacement map (b) General displacement map (c) Flow map

Figure 9.1: Three deformation techniques for parametric surfaces.

the ray casting algorithms that were presented in Chapter 5, this algorithm is not robust and may fail to find the correct ray-surface intersection point when an excessive amount of surface deformation is introduced. Finally, Section 9.4 summarises the chapter.

9.1 Previous Work

The first attempt at adding additional detail to otherwise smooth surfaces was done by Blinn [1978] with the introduction of the *bump mapping* technique. Bump mapping introduces slight perturbations into the surface normal, which, when passed on to the surface shading model, create the illusion of surface detail. Bump mapping, however, does not really create geometric detail as the surface continues to be smooth. This becomes apparent along the silhouette edges of the surface, which are not affected by the perturbation of the normals.

Cook [1984] introduced true geometric detail onto surfaces with the development of the *displacement mapping* technique. He proposed displacement mapping as one example of the shaders that could be included in a *shade tree*. His work formed the background for the modern concept of *shader* that is used in languages such as Renderman and Cg [Upstill, 1990; Fernando and Kilgard, 2003]. Displacement mapping can be expressed as a scalar map $d(u, v) \in \mathbb{R}$ where the (u, v) pair gives the parametric coordinates over the surface of some object. This can be used for either parametric patches or polygonal meshes with (u, v), in the latter case, being interpolated from texture coordinates specified at the vertices. For a point $\mathbf{x}(u, v)$ on the surface, with unit normal vector $\mathbf{n}(u, v)$, the displaced surface point $\mathbf{x}'(u, v)$ is:

$$\mathbf{x}'(u, v) = \mathbf{x}(u, v) + d(u, v)\mathbf{n}(u, v). \tag{9.1}$$

Figure 9.1(a) illustrates how displacement mapping operates. Displacement mapping can be rendered with recursive mesh subdivision [Cook et al., 1987], ray-domain deformation [Logie and Patterson, 1995] (see also Section 9.3), geometry caching [Pharr and Hanrahan, 1996], range estimation [Heidrich and Seidel, 1998], image space warping [Schaufler and Priglinger, 1999] and grid-based traversal [Smits et al., 2000].

Although displacement mapping has traditionally been applied to parametric patches and polygonal meshes, Sclaroff and Pentland [1991] have also applied the technique to implicit surfaces. They combine domain deformation and displacement mapping on top of an initial implicit surface in order to generate a more complex one. Their technique, however, only works for initial surfaces such as superquadrics that can be easily parameterised. This is because there must

exist a projection operator that for every point $\mathbf{x}$ in space finds the (u, v) coordinates of its normal projection on the initial surface. This type of operator is not generally available unless the initial surface happens to be simple. One example of such an operator (without the surface parameterisation aspect) is $\mathbf{x}/\|\mathbf{x}\|$, which projects $\mathbf{x}$ onto the surface of a unit radius sphere and was used in Section 2.4.3 to combine displacement maps with hypertextures for procedural planets.

A more general displacement mapping technique was proposed by Lewis [1989] and uses a vector displacement map $\mathbf{d}(u, v) \in \mathbb{R}^3$ instead of the simpler scalar map $d(u, v)$[1]. Lewis used procedural noise functions with this general displacement map to deform stochastic surfaces. The general displacement mapping operation can be expressed as:

$$\mathbf{x}'(u, v) = \mathbf{x}(u, v) + \mathbf{d}(u, v). \tag{9.2}$$

This is illustrated in Figure 9.1(b). It is clear that displacement mapping is a particular case of (9.2), which can be obtained with $\mathbf{d}(u, v) = d(u, v)\mathbf{n}(u, v)$. The general displacement mapping technique is more flexible at introducing surface detail and, in particular, can introduce surface overhangs, something that the original displacement mapping method cannot do.

Flow mapping was introduced by Pedersen [1994] and deforms a surface by advecting it along the streamlines of a specified vector field $\mathbf{v} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$. A flow mapped parametric surface has parameterisation $\mathbf{x}'(u, v, t)$ where (u, v), as before, are the parametric coordinates of the original surface and the extra parameter t expresses the amount of deformation that the surface undergoes. The expression for $\mathbf{x}'(u, v, t)$ is:

$$\mathbf{x}'(u, v, t) = \mathbf{x}(u, v) + \int_0^t \mathbf{v}(\mathbf{x}'(u, v, \epsilon))d\epsilon. \tag{9.3}$$

Figure 9.1(c) illustrates flow mapping. An alternative way of looking at flow mapping is to consider it as the solution of an ordinary differential equation. The surface $\mathbf{x}'(u, v, t)$ is obtained by solving:

$$\frac{\partial \mathbf{x}'}{\partial \epsilon} = \mathbf{v}(\mathbf{x}'(u, v, \epsilon)), \tag{9.4}$$

subject to the initial condition $\mathbf{x}'(u, v, 0) = \mathbf{x}(u, v)$ and where the solution is obtained for $\epsilon = t$. The deformed surface is generated when the equation (9.4) is solved for every (u, v) pair. A problem with displacement mapping, both in its original form (9.1) and its more general form (9.2), is that the deformed surface may no longer be a manifold if the original surface happens to have concavities (see a two-dimensional example in Figure 9.2). The same problem does not occur with flow mapping and the deformed surface is always guaranteed to be a proper orientable manifold for any continuous flow field $\mathbf{v}$. The reason for this will be explained in the following section where flow mapping is further studied.

Flow maps are a subset of the class of general displacement maps, which is clear when equation (9.2) is compared with equation (9.3). The general displacement map to which any given flow map is equivalent to is given by $\mathbf{d}(u, v) = \int_0^t \mathbf{v}(\mathbf{x}')d\epsilon$. For every (u, v) pair, it is a vector that connects the starting point to the ending point of the streamline generated by the differential equation (9.4) (compare Figures 9.1(b) and 9.1(c)). If, for a given flow field $\mathbf{v}$, the integral

[1]This should not be confused with the *generalised displacement mapping* technique of Wang et al. [2004], which is, in fact, a hypertexture discretised as a five dimensional texture buffer to facilitate ray casting operations.

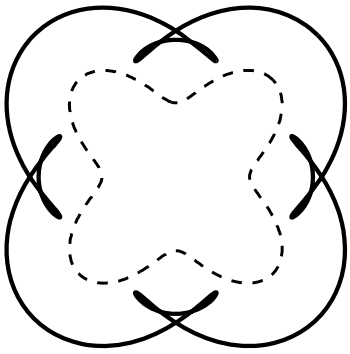


Figure 9.2: Displacement mapping does not always generate manifolds. The dotted line shows the initial manifold surface. The continuous line shows the displaced surface.

in (9.3) can be solved analytically then the flow map can be directly implemented as a general displacement map without having to resort to a numerical integration routine. A general displacement map, on the other hand, may not always be equivalent to a flow map. This is again illustrated with Figure 9.2 where the displacement map shown could never be the result of advecting the surface through a flow field.

There has been considerable interest in recent years in implementing displacement mapping on the GPU. In most cases, displacement mapping is implemented at the fragment shader level by performing localised ray casting. Some GPU methods are not robust and may sometimes miss the correct intersection points with the deformed surface. Others use either range estimation techniques or Lipschitz bounds to perform robust ray-surface intersections. Szirmay-Kalos and Umenhoffer [2008] give a comprehensive survey of GPU displacement mapping methods. The displacement map is stored as a texture in the GPU's video memory, which limits its application to relatively small surfaces, but it is certainly possible to use a procedural displacement map that is evaluated on the fly. Current GPU methods work over polygonal meshes and are not directly applicable to implicit surfaces.

9.2 Domain Deformation of Hypertextured Implicit Surfaces

The deformation of implicit surfaces can be achieved with the domain deformation technique of Section 2.3.3. Flow mapping was initially developed for parametric surfaces but it can also be applied to implicit surfaces as a case of domain deformation. Before presenting flow mapping for implicit surfaces, the domain deformation technique is briefly reviewed here. Let $f(f_0(\mathbf{x}), \mathbf{x})$ be an original hypertextured implicit surface relative to the seed surface $f_0(\mathbf{x})$. A deformed implicit surface $f_D(f_0(\mathbf{x}), \mathbf{x})$ can be obtained in the following way:

$$f_D(f_0(\mathbf{x}), \mathbf{x}) \;=\; f(f_0(\mathbf{d}(\mathbf{x})), \mathbf{d}(\mathbf{x})), \tag{9.5}$$

where $\mathbf{d} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ is the domain mapping function. This vector function takes a point $\mathbf{x}$ in space and maps it onto a different point $\mathbf{d}(\mathbf{x})$ before evaluating the implicit function f, thereby

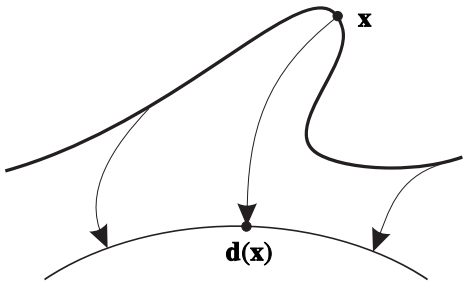


Figure 9.3: Flow mapping for implicit surfaces takes points from the deformed surface and converts them into points on the original surface.

causing the implicit surface to become deformed. Figure 9.3 shows an example in two dimensions. The domain mapping function takes all the points $\mathbf{x}$ on the deformed surface and converts them into points $\mathbf{d}(\mathbf{x})$ on the original surface. This mechanism is opposite to the one used by the surface deformation techniques of Section 9.1 where the displacement functions mapped points from the original surface to the deformed surface (compare Figure 9.3 with Figure 9.1(c)). Note also that in (9.5) the domain mapping function is applied simultaneously to the seed surface generated by f_0 and the hypertextured component of f. The seed surface and the hypertexture must undergo deformation together in order to avoid any changes in surface topology. If deformation was applied to one but not the other, the interaction between the two could give rise to new disconnected surface components.

The normal of the deformed surface is required when performing shading calculations. The normal is simply the gradient vector ∇f_D followed by normalisation. The gradient, in turn, can be computed by applying the chain rule of differentiation to each of of the three vector components of ∇f_D relative to the functions f, f_0 and $\mathbf{d} = [d_1\ d_2\ d_3]^T$:

$$\frac{\partial f_D}{\partial x_i} = \frac{\partial f}{\partial f_0}\left(\frac{\partial f_0}{\partial d_1}\frac{\partial d_1}{\partial x_i} + \frac{\partial f_0}{\partial d_2}\frac{\partial d_2}{\partial x_i} + \frac{\partial f_0}{\partial d_3}\frac{\partial d_3}{\partial x_i}\right) + \\ + \frac{\partial f}{\partial d_1}\frac{\partial d_1}{\partial x_i} + \frac{\partial f}{\partial d_2}\frac{\partial d_2}{\partial x_i} + \frac{\partial f}{\partial d_3}\frac{\partial d_3}{\partial x_i} \quad \text{with } i = 1, 2, 3. \tag{9.6}$$

This can be written more succinctly in matrix form as:

$$\nabla f_D = \mathcal{J}^T\{\mathbf{d}\}\left(\frac{\partial f}{\partial f_0}\nabla f_0 + \nabla_{\mathbf{x}} f\right) = \mathcal{J}^T\{\mathbf{d}\}\nabla f, \tag{9.7}$$

where $\nabla_{\mathbf{x}} f$ is the gradient of f relative to its second argument and $\mathcal{J}^T\{\mathbf{d}\} = \left[\partial d_i/\partial x_j\right]$, with $i, j = 1, 2, 3$, is the Jacobian matrix of the vector function $\mathbf{d}$. The difference between (9.7) and the gradient of the undeformed hypertextured surface is the presence of the Jacobian $\mathcal{J}\{\mathbf{d}\}$, which needs to be computed for every point on the surface.

9.2.1 Flow Mapping

Flow mapping for implicit surfaces is achieved with a domain deformation function $\mathbf{d}$ where points in space are made to follow the streamlines of some supplied vector field $\mathbf{v}$. Each point

follows a trajectory $\mathbf{x}(t)$ where t is the desired amount of deformation. The starting point $\mathbf{x}(0)$ of each trajectory is the original point in the undeformed space. The expression for the deformation $\mathbf{d}(\mathbf{x})$ at a point $\mathbf{x}$ in undeformed space is then:

$$\mathbf{d}(\mathbf{x}) = \mathbf{x} + \int_0^t \mathbf{v}(\mathbf{x}(\epsilon))d\epsilon. \tag{9.8}$$

If one looks at $\mathbf{v}$ as a velocity field then t is the amount of time that the points are allowed to follow the streamlines. The higher the value of t, therefore, the higher the amount of deformation that will be imposed on the surface. With a change of variable $\xi = \epsilon/t$, the deformation (9.8) can also be written as:

$$\mathbf{d}(\mathbf{x}) = \mathbf{x} + \int_0^1 t\mathbf{v}(\mathbf{x}(t\xi))d\xi. \tag{9.9}$$

The flow field $\mathbf{v}(\mathbf{x})$ can be replaced with a new flow field $\mathbf{u}(\mathbf{x}) = t\mathbf{v}(\mathbf{x})$, which is similar to $\mathbf{v}(\mathbf{x})$ except that the vectors have an increased length. The trajectory $\mathbf{x}(\epsilon)$ can be replaced with a new trajectory $\mathbf{y}(\xi) = \mathbf{x}(t\xi)$ that has the same spatial configuration of $\mathbf{x}(\epsilon)$, the only difference being that the original point follows this trajectory at an increased speed. With these replacements, the domain deformation function can also be written as:

$$\mathbf{d}(\mathbf{x}) = \mathbf{x} + \int_0^1 \mathbf{u}(\mathbf{y}(\xi))d\xi, \tag{9.10}$$

where it is understood, as before, that $\mathbf{y}(0) = \mathbf{x}$. There is no loss in generality when using the simpler definition (9.10) as opposed to (9.8). The amount of deformation can be controlled by increasing the length of the vectors of the flow field $\mathbf{u}(\mathbf{x})$ that is supplied. The trajectory $\mathbf{y}(\xi)$ is calculated by expressing (9.10) in differential form and corresponds to solving the ordinary differential equation (ODE):

$$\frac{d\mathbf{y}}{d\xi} = \mathbf{u}(\mathbf{y}(\xi)). \tag{9.11}$$

The same FORTRAN routine `lsodar` that was used in Chapter 8 for tracking separatrices is used in this context for finding the deformation of each domain point $\mathbf{x}$ [Hindmarsh, 1983]. The desired solution is $\mathbf{d}(\mathbf{x}) = \mathbf{y}(1)$.

There are three ways in which the flow field $\mathbf{u}$ can be specified. The first one consists of specifying the three components $\mathbf{u} = [u_1\ u_2\ u_3]^T$ directly. The second one consists of specifying $\mathbf{u}$ as the gradient of some scalar field $\phi : \mathbb{R}^3 \to \mathbb{R}$:

$$\mathbf{u}(\mathbf{x}) = \nabla\phi. \tag{9.12}$$

This second option for $\mathbf{u}$ is simpler to specify as it only requires that one scalar function be designed rather than three. It has the property that the minima of $\phi(\mathbf{x})$ act as attractors for the deformed surface while the maxima act as repulsers. This is because the streamlines of the gradient field $\nabla\phi$ originate in minima and converge towards the maxima (recall from Figure 9.3 that the streamlines of the flow perform the inverse deformation). One disadvantage of the flow field (9.12) is that excessive surface deformation can easily be introduced, due to the attraction/repulsion effect caused by the stationary points of $\nabla\phi$, and this can cause problems for the ray-surface intersection routine that is explained in Section 9.3. The third and final

option for specifying $\mathbf{u}$, which was not implemented for this thesis, is to make it equal to the curl of some other vector field $\psi : \mathbb{R}^3 \to \mathbb{R}^3$:

$$\mathbf{u}(\mathbf{x}) = \nabla \times \psi. \tag{9.13}$$

This type of flow field has the interesting property of preserving the volume enclosed by the initial surface independently of the amount of deformation that is introduced. If the surface is stretched in some areas, for example, it must be compressed in others so that the total volume enclosed remains constant. This result follows from the study of fluid dynamics and is a consequence of the fact that the divergence of (9.13) is always zero, which corresponds to a property of mass conservation [Batchelor, 2000].

The Jacobian matrix $\mathcal{J}\{\mathbf{d}\}$ expresses the local deformation imposed by the flow map $\mathbf{d}(\mathbf{x})$ at every point $\mathbf{x}$. As previously explained, this matrix is required to compute the gradient and hence the normal of the deformed surface. The Jacobian is obtained by taking the three partial derivatives of the deformation (9.10) with respect to the x_1, x_2 and x_3 spatial coordinates. This leads to a set of new ODEs that must be solved simultaneously with (9.11):

$$\frac{d}{d\xi}\left(\frac{\partial \mathbf{y}}{\partial x_i}\right) = \mathcal{J}\{\mathbf{u}\}\frac{\partial \mathbf{y}}{\partial x_i}(\xi) \quad \text{with } i = 1, 2, 3. \tag{9.14}$$

The Jacobian matrix of the flow field $\mathcal{J}\{\mathbf{u}\}$ is evaluated at the point $\mathbf{y}(\xi)$, whose trajectory is tracked by (9.11). In the case where a flow field of the type given in (9.12) is used, the Jacobian $\mathcal{J}\{\mathbf{u}\}$ becomes the Hessian matrix $\mathcal{H}\{\phi\}$. The initial conditions for the ODEs are the vectors $\partial\mathbf{y}/\partial x_1(0) = [1\ 0\ 0]^T$, $\partial\mathbf{y}/\partial x_2(0) = [0\ 1\ 0]^T$ and $\partial\mathbf{y}/\partial x_3(0) = [0\ 0\ 1]^T$. The solutions of the ODEs (9.14), obtained for $\xi = 1$, become the columns of the final 3×3 Jacobian matrix $\mathcal{J}\{\mathbf{d}\}$ for the flow map:

$$\mathcal{J}\{\mathbf{d}\} = \left[\frac{\partial \mathbf{y}}{\partial x_1}(1) \;\middle|\; \frac{\partial \mathbf{y}}{\partial x_2}(1) \;\middle|\; \frac{\partial \mathbf{y}}{\partial x_3}(1)\right]. \tag{9.15}$$

Appendix B gives a complete description of the system of 12 ODEs that need to be solved (three for $\mathbf{y}$, three for $\partial\mathbf{y}/\partial x_1$, three for $\partial\mathbf{y}/\partial x_2$ and three more for $\partial\mathbf{y}/\partial x_3$).

9.2.2 Flow Mapping as a Homeomorphism

A homeomorphism establishes a topological equivalence between two surfaces. It is a function that continuously deforms one surface into the other while maintaining the same topological properties. A sphere is homeomorphic to a cube, for example. It is easy to imagine a transformation that gradually deforms the sphere into the shape of a cube. A sphere is not homeomorphic to a torus, on the other hand. Any deformation of the sphere towards a torus would have to tear the surface at some point in order to create the hole of the torus. If a surface is deformed through flow mapping, the original surface and the deformed surface are guaranteed to be homeomorphic.

A function $\mathbf{d} : X \to Y$ establishes a homeomorphism between the topological spaces X and Y if it verifies the following three conditions:

- The function $\mathbf{d}$ is continuous.

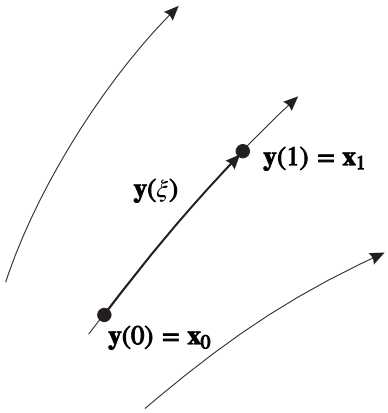


Figure 9.4: The trajectory of a point undergoing flow mapping follows one of the streamlines of the flow. The direction from $\mathbf{x}_0$ to $\mathbf{x}_1$ can be reversed by integrating backwards or reversing the direction of the flow field.

- The function $\mathbf{d}$ is bijective.
- The inverse function $\mathbf{d}^{-1}$ is continuous.

The topological space of any implicit surface is the set of three-dimensional points that form the surface. For two surfaces related through flow mapping, therefore, the two topological spaces obey $X \subset \mathbb{R}^3$ and $Y \subset \mathbb{R}^3$. If one can find a function $\mathbf{d} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ that obeys the previous three properties then it can be used as a homeomorphism between any subset X of $\mathbb{R}^3$ and the subset $Y = \mathbf{d}(X)$ that is the image of X after the application of $\mathbf{d}$.

The flow mapping operator $\mathbf{d}$ defined in equation (9.10) is continuous if the flow field $\mathbf{u}$ is also continuous. To see if $\mathbf{d}$ is bijective it is sufficient to show that there exists another operator $\mathbf{o} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ such that $\mathbf{o}(\mathbf{d}(\mathbf{x})) = \mathbf{x}$ for every point $\mathbf{x}$. If such an operator exists then $\mathbf{d}$ is invertible and one can identify $\mathbf{o}$ as $\mathbf{o} = \mathbf{d}^{-1}$. Consider a point $\mathbf{x}_0 \in \mathbb{R}^3$ that is mapped onto another point $\mathbf{x}_1 \in \mathbb{R}^3$ by flow mapping. There exists a trajectory $\mathbf{y}(\xi)$, following a streamline of the flow field, such that $\mathbf{y}(0) = \mathbf{x}_0$ and $\mathbf{y}(1) = \mathbf{x}_1$ (see Figure 9.4). The points $\mathbf{x}_1$ and $\mathbf{x}_0$ are related by:

$$\mathbf{x}_1 = \mathbf{d}(\mathbf{x}_0) = \mathbf{x}_0 + \int_0^1 \mathbf{u}(\mathbf{y}(\xi))d\xi. \tag{9.16}$$

As the time parameter ξ evolves from 0 to 1, the point travels along the trajectory from $\mathbf{x}_0$ to $\mathbf{x}_1$. It is possible to have the point return from $\mathbf{x}_1$ back to $\mathbf{x}_0$ along the same trajectory by simply causing the time to run backwards:

$$\mathbf{x}_1 + \int_1^0 \mathbf{u}(\mathbf{y}(\xi))d\xi = \mathbf{x}_1 - \int_0^1 \mathbf{u}(\mathbf{y}(\xi))d\xi = \\ = \mathbf{x}_0 + \int_0^1 \mathbf{u}(\mathbf{y}(\xi))d\xi - \int_0^1 \mathbf{u}(\mathbf{y}(\xi))d\xi = \mathbf{x}_0. \tag{9.17}$$

Since the point $\mathbf{x}_0$ is arbitrary, this result ensures that $\mathbf{d}$ is bijective. The inverse of $\mathbf{d}$ is:

$$\mathbf{d}^{-1}(\mathbf{x}) = \mathbf{x} - \int_0^1 \mathbf{u}(\mathbf{y}(\xi))d\xi, \tag{9.18}$$

where the trajectory obeys $\mathbf{y}(1) = \mathbf{x}$. This is an intuitive result, which states that the flow mapping can be inverted by making the streamline integration run backwards or, equivalently, by replacing the flow field $\mathbf{u}$ with the flow field $-\mathbf{u}$. The streamlines of $\mathbf{u}$ and $-\mathbf{u}$ are the same, the difference being that points travel along these streamlines in opposite directions. The inverse flow map (9.18) is continuous because $\mathbf{u}$ had previously been required to be continuous, which makes $-\mathbf{u}$ continuous also. This is enough to characterise the flow map $\mathbf{d}$, defined by equation (9.10), as a homeomorphism between an original implicit surface and the deformed implicit surface resulting from the application of $\mathbf{d}$.

The homeomorphism between two implicit surfaces related by flow mapping has two important consequences in the context of this thesis:

1. If the original surface is a manifold, the deformed surface is also a manifold.
2. If the original surface is connected, the deformed surface is also connected.

The first result guarantees that deformed surfaces will not suffer from self-intersections or any other topological artifacts that might make it impossible to differentiate between an outside space and an inside space. The second result guarantees that the outcome of the topology correction algorithms of Chapter 8 will not be invalidated by the application of flow mapping – deforming an implicit surface with flow mapping will not cause any new pieces of surface to become disconnected.

9.3 Ray Casting Flow Mapped Implicit Surfaces

As explained in Chapter 5, ray casting an implicit surface considers the problem of finding the intersection point between the surface and a ray $\mathbf{r}(t)$ defined parametrically by the distance t, according to equation (5.1). Plugging the definition of the ray into the implicit function (9.5) that defines the deformed implicit surface, one must find the smallest value of t verifying the equation:

$$f(f_0(\mathbf{d}(\mathbf{r}(t))), \mathbf{d}(\mathbf{r}(t))) = 0. \tag{9.19}$$

From one perspective, this equation requires that the values of the deformed implicit function (deformed due to the presence of $\mathbf{d}$) be examined along the points of the rectilinear ray $\mathbf{r}(t)$ until the first zero is found. From a different perspective, one can consider instead the parametric curve defined by $\mathbf{s}(t) = \mathbf{d}(\mathbf{r}(t))$, which results from applying the domain deformation to the ray rather than the surface. The problem is now one of finding the first intersection point between the *original* implicit surface and an arbitrary parametric curve $\mathbf{s}(t)$:

$$f(f_0(\mathbf{s}(t)), \mathbf{s}(t)) = 0. \tag{9.20}$$

This is illustrated in Figure 9.5. The technique of deforming rays in order to simplify intersection calculations was first proposed by Kajiya [1983] to ray trace surfaces of revolution. The same technique was used by Barr [1984] to ray trace superquadrics that were subjected to his taper, twist and bend deformations. Logie and Patterson [1995] again used ray-domain deformation to ray trace simple displacement mapped solids. The underlying idea behind these ray deformation techniques is that it is often easier to compute the intersection between a curve

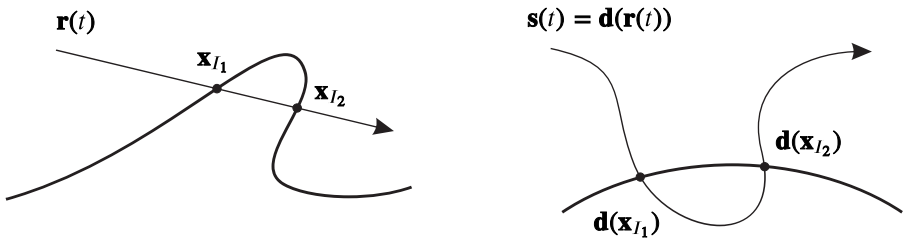


Figure 9.5: Ray-domain deformation for intersection testing. The two intersection points $\mathbf{x}_{I_1}$ and $\mathbf{x}_{I_2}$ are mapped by the deformation function $\mathbf{d}$ onto the original implicit surface.

and a simple surface than between a rectilinear ray and a complex surface. This is not necessarily the case for the flow mapped implicit surfaces of this section because the original surface, defined by $f(f_0(\mathbf{x}), \mathbf{x})$, may already be quite complex. Nevertheless, the ray-domain deformation perspective (9.20) of the intersection problem is adopted because it facilitates the development of a ray intersection algorithm for flow mapped implicit surfaces.

9.3.1 Ray Casting Algorithm

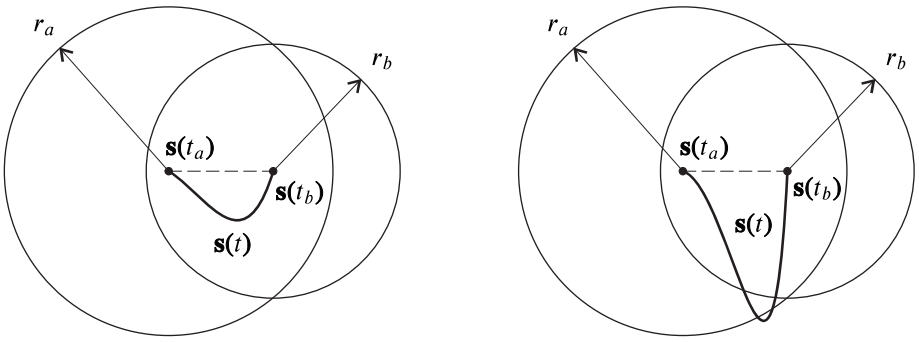
The location of a ray-surface intersection point amounts to finding the smallest root t of the one-dimensional equation:

$$g(t) = 0, \tag{9.21}$$

where $g(t) = f(f_0(\mathbf{s}(t)), \mathbf{s}(t))$ is an auxiliary function that is specific to every ray. In line with the ray casting algorithm of Section 5.4, the ray casting of flow mapped surfaces could proceed by computing an interval extension $G(T)$ with reduced affine arithmetic, where the interval $T = [t_{MIN}, t_{MAX}]$ is the complete extent of the ray. The interval T would then be recursively subdivided whenever the test $0 \ni G(T)$ indicated the possible presence of a root. The problem with this approach is that it requires the computation of a reduced affine arithmetic estimate $\mathbf{s}(\hat{t})$ for the trajectory of the deformed ray, where $\hat{t}$ expresses the interval T in affine arithmetic form, according to (5.23). The computation of a point $\mathbf{s}(t)$ involves the solution of the ODE (9.11). The computation of $\mathbf{s}(\hat{t})$, therefore, would require that an ODE solver be implemented in reduced affine arithmetic form for the entire extent $\hat{t}$ along the ray. Consider the computation of any individual point $\mathbf{s}(t)$, with $t \in \hat{t}$, on the ray. The simplest possible ODE solver for (9.11) is *Euler's method.* It computes successive iterations of the point:

$$\mathbf{s}_{i+1}(t) = \mathbf{s}_i(t) + \mathbf{u}(\mathbf{s}_i(t))\Delta\xi, \tag{9.22}$$

where $\Delta\xi$ is a small increment along the streamline of $\mathbf{u}$ that transports the point $\mathbf{s}(t)$. The iteration (9.22) has a straightforward representation $\mathbf{s}_{i+1}(\hat{t})$ with reduced affine arithmetic but it needs to be evaluated many times because $\Delta\xi$ is small and ξ has to evolve from 0 to 1. For every evaluation of (9.22), the estimate $\mathbf{s}_i(\hat{t})$ becomes a bit more conservative. When the solver finishes, after some given number N of iterations, the final estimate $\mathbf{s}_N(\hat{t})$ is going to be so conservative that it ceases to give any useful information about the evolution of the ray. Euler's method is notoriously inaccurate and any serious implementation would have to use a better method such as a fourth order Runge-Kutta, which is also amenable to an affine



(a) Correct case. (b) Incorrect case.

Figure 9.6: Ray casting a flow mapped surface. The section of the deformed ray $\mathbf{s}(t)$, for $t_a < t < t_b$, does not intersect the surface if it remains inside both unbounding spheres.

arithmetic implementation [Press et al., 1992]. The problem remains, however, that numerical ODE solvers are iterative in nature and typically require a large number of iterations to achieve a reasonable accuracy. Unless the flow field $\mathbf{u}$ is simple and the integration can be solved analytically, the use of affine arithmetic estimates taken from a numerical ODE solver are going to be over-conservative. This is true for standard affine arithmetic and more so for reduced affine arithmetic that systematically condenses error symbols.

Considering that it is difficult to obtain an affine arithmetic estimate for $\mathbf{s}(t)$ that is simultaneously robust and accurate, it becomes unfeasible to have a robust intersection algorithm. The algorithm would have to subdivide the ray into very small intervals to compensate for the over-conservativeness of the affine arithmetic estimates, slowing down considerably the progression along the ray in the search for a root. The ray-surface intersection algorithm that has been implemented is not robust and may fail if the deformed ray undergoes a very sharp transition at some point. The algorithm is still based on recursive subdivision along the length of the ray. Lipschitz bounds are used to detect spans of the ray that are not likely to intersect with the surface. Figure 9.6 illustrates how this works for some section $[t_a, t_b] \subset T$ of the ray.

An unbounding sphere can be placed at the point $\mathbf{s}(t_a)$ with a radius that is given by the expression $r_a = f(f_0(\mathbf{s}(t_a)), \mathbf{s}(t_a))/\lambda$ (recall Section 5.2.1 for the definition of unbounding spheres). The parameter λ is the Lipschitz bound for the implicit function f (see Section 5.2). A similar unbounding sphere with radius r_b can be placed at the point $\mathbf{s}(t_b)$. If the trajectory of the deformed ray from t_a to t_b is entirely contained inside the union of the two unbounding spheres then that section of the ray is known to be outside the surface, otherwise the interval $[t_a, t_b]$ needs to be subdivided and the same test repeated for the smaller subintervals. Since it is not possible to know with certainty the evolution of $\mathbf{s}(t)$ inside the interval (given the previously mentioned limitations of affine arithmetic estimates), one considers that the ray does not intersect with the surface if the following condition holds:

$$\| \mathbf{s}(t_b) - \mathbf{s}(t_a) \| < \min(r_a, r_b). \tag{9.23}$$

This means that the displacement of the trajectory from t_a to t_b is contained inside the smallest of the two unbounding spheres. By implication, it is also contained inside the largest of the two spheres (Figure 9.6(a)). If the section of the deformed ray, between t_a and t_b, does not vary

```
push T_0 = [t_MIN, t_MAX] onto stack;
while stack not empty
  pop T = [t_a, t_b] from the stack;
  if t_b − t_a < ε                    // If small enough interval
    if g(t_a)g(t_b) < 0               // Converged to a root
      return t_a;                     // t_a is outside the surface and t_b is inside
    continue;                         // Skip to next interval otherwise
  let r_a = radius of unbounding sphere at s(t_a);
  let r_b = radius of unbounding sphere at s(t_b);
  if ‖ s(t_b) − s(t_a)‖ < min(r_a, r_b)
    continue;                         // Skip ray section for T
                                      // as it is likely outside the surface
  let t_i = (t_b + t_a)/2;            // Compute midpoint of interval
  let T_l = [t_a, t_i];               // Nearest half of interval T
  let T_r = [t_i, t_b];               // Farthest half of interval T
  push T_r onto stack;                // Push farthest half onto stack first
  push T_l onto stack;
return +∞;                            // No intersection found
```

Figure 9.7: The ray-surface intersection algorithm for flow mapped surfaces.

excessively then the test (9.23) will give a correct result. A situation that may give an incorrect answer is depicted in Figure 9.6(b), where a sharp corner of the ray juts outside the two spheres. The test gives an incorrect answer if an intersection with the surface happens to occur at this corner. In practice, this intersection testing method performs well except for large amounts of deformation, expressed through a flow field $\mathbf{u}$ where the magnitude of the flow vectors may change very rapidly over a small region. The pseudo-code for the intersection algorithm is shown in Figure 9.7. A stack is used to store the intervals $[t_a, t_b]$ that have a possibility of intersecting with the ray. For each such interval, the corresponding pair $\{\mathbf{s}(t_a), \mathbf{s}(t_b)\}$ of deformed coordinates is also stored. This additional storage prevents the deformation $\mathbf{s}(t_i)$ of an interval intermediate point from being computed twice – once for the interval $[t_a, t_i]$ and once for the interval $[t_i, t_b]$.

The intersection algorithm for flow mapped surfaces has similarities with the algorithm of Figure 5.8 that is based on reduced affine arithmetic. The interval along the ray is recursively subdivided in the search for a root. If a small enough interval $[t_a, t_b]$ is found, one sees if it contains a root by checking if $g(t_a)$ and $g(t_b)$ have opposite signs. A return is made if that is the case, otherwise the interval is ignored. If the interval $[t_a, t_b]$ is still large, one checks condition (9.23) to see if the section of the ray generated by the interval is likely outside the surface. The interval is again ignored if that is true. If all else fails, the interval is subdivided and the halves are pushed onto the stack for subsequent analysis. The algorithm returns no intersection if the interval stack becomes empty with no root having been detected. Hidden in the evaluation of $\mathbf{s}(t_a)$ and $\mathbf{s}(t_b)$ is the numerical solution of the differential equation for the deformation of the ray.

The intersection algorithm relies entirely on information taken from discrete samples along the ray, which is the cause for its lack of robustness since it is not able to examine a continuous stretch of deformed ray. Condition (9.23), however, provides an adaptive sampling mechanism because the radii r_a and r_b decrease as the ray gets progressively closer to the surface. This, in turn, causes an increase in sampling density along the ray when it approaches the surface, making it likely that an intersection will not be missed. In a similarity with the reduced affine arithmetic algorithm of Figure 5.8, the algorithm of Figure 9.7 only finds the nearest intersection point along the ray. Small modifications to the code, which are not shown for clarity, allow the algorithm to detect all surface intersections by increasing distance along the ray. For every intersection that is found, the connectivity testing algorithms of Chapter 8 can then be used, allowing flow mapping to integrate seamlessly with topology correction.

9.3.2 Results

Figure 9.8 shows a sphere that has been deformed with a flow field made of three cellular texture noise function components. The expression for the flow field is:

$$\mathbf{u}(\mathbf{x}) = \alpha \begin{bmatrix} f_N(4(\mathbf{x}+\Delta_1)) \\ f_N(4(\mathbf{x}+\Delta_2)) \\ f_N(4(\mathbf{x}+\Delta_3)) \end{bmatrix}. \tag{9.24}$$

The factor 4 gives the overall scale of the noise relative to the sphere, making the noise details roughly four times smaller than the sphere radius. The three delta factors are translations intended to decorrelate the three noise components, given that a single noise function f_N is used for all three components. The factor α gives the intensity of the flow field. From top to bottom in Figure 9.8, α is equal to 0.1, 0.5 and 1.0, respectively. The rendering times are equal to 19m 38s, 100m 20s and 197m 25s, respectively. This figure shows how surface features are created over the initially smooth sphere that resemble the Voronoi patterns typical of cellular texture noise (Section 4.5). As the intensity of the flow is increased, these features are progressively dragged along the streamlines of the flow. At no time does the surface of the sphere break into multiple components or different parts of the surface intersect with each other. The deformed sphere is also seen to increase in size since the flow field chosen does not have volume preserving properties.

Figure 9.9 shows a sphere that has been deformed with a gradient flow field of the type (9.12). The scalar function used for this field is based on a gradient noise function (Section 4.3) and has the form:

$$\phi(\mathbf{x}) = \alpha f_N(4\mathbf{x}). \tag{9.25}$$

The intensity α grows from top to bottom in Figure 9.9 with the values 0.01, 0.025 and 0.04, respectively. The rendering times are equal to 9m 58s, 18m 8s and 25m 59s, respectively. Unlike the surface of Figure 9.8, this surface does not grow appreciably in size. This is because of the minima of $\phi(\mathbf{x})$, which act as sinks for the flow, as previously explained, preventing the surface from expanding past them. The peaks on this surface indicate the nearby presence of the minima of $\phi(\mathbf{x})$. The deep valleys that resemble inverted peaks also indicate the presence of minima. Figure 9.10 illustrates rendering errors that can occur during the visualisation of flow mapped surfaces. It is again a gradient flow (9.12), this time using a sparse convolution

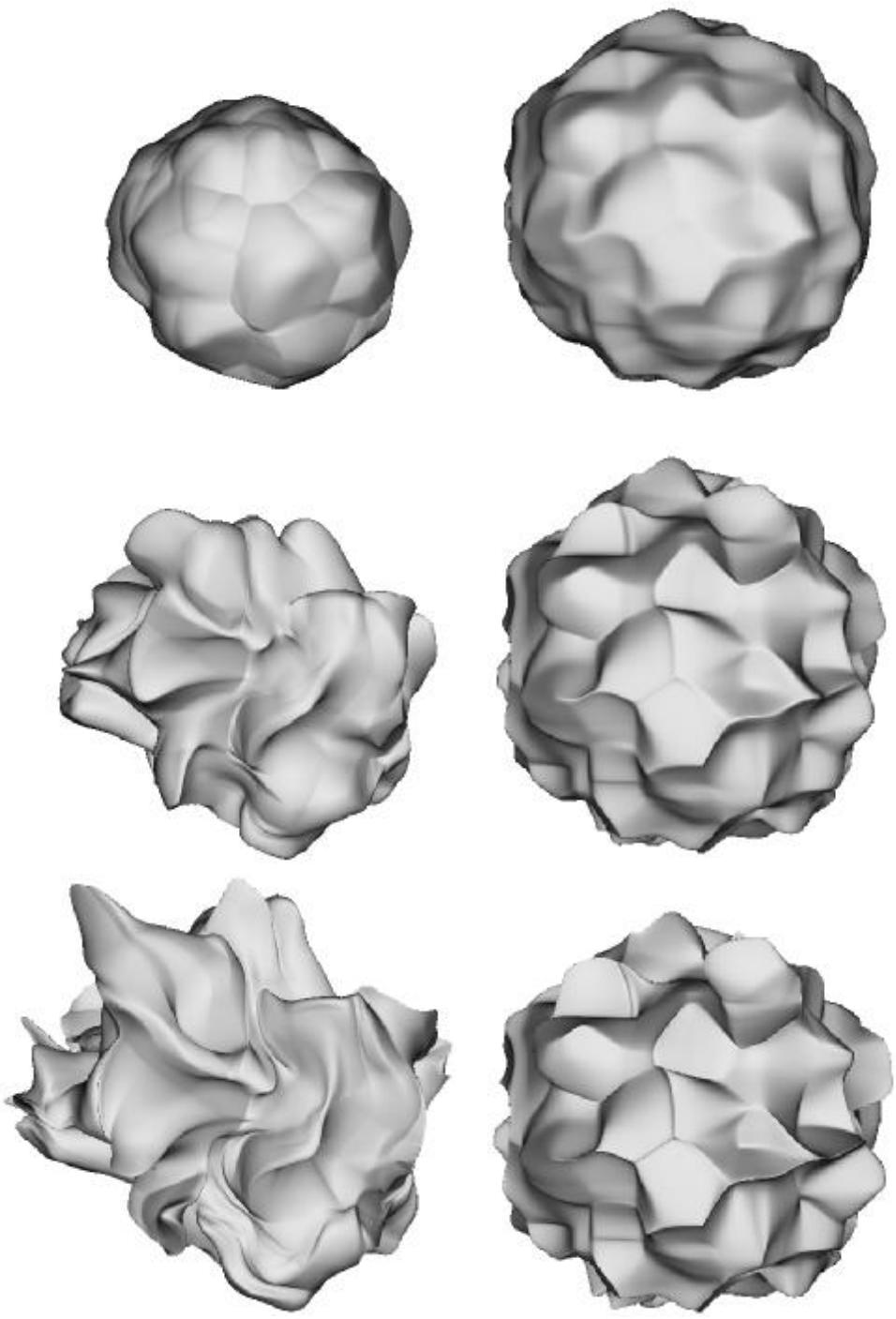


Figure 9.8: A sphere flow mapped with cellular texture noise. From top to bottom, the value of α is 0.1, 0.5 and 1.0, respectively.

Figure 9.9: A sphere flow mapped with gradient noise. From to to bottom, the value of α is 0.01, 0.025 and 0.04, respectively.

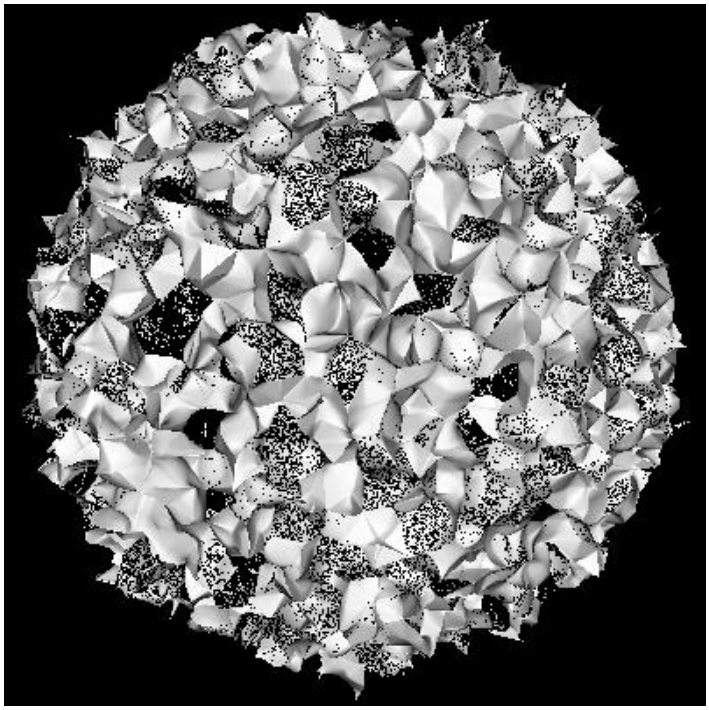


Figure 9.10: A flow mapped sphere incorrectly rendered.

noise function (Section 4.4) with the form $\phi(\mathbf{x}) = 0.1 f_N(16\mathbf{x})$. The surface is shown against a dark background to make more apparent those pixels for which surface intersection errors occurred. The intensity of the flow was increased relative to the flow (9.25) and the size of the flow features was also made four times smaller. This increase in the range of variation of the flow field leads to deformed rays with trajectories that change much more rapidly and, in turn, leads to wrong intersection calculations. The errors in Figure 9.10 could be corrected by introducing a modification in condition (9.23) so that it becomes:

$$\| \mathbf{s}(t_b) - \mathbf{s}(t_a) \| \; < \; \varepsilon \min(r_a, r_b), \tag{9.26}$$

where $0 < \varepsilon \leqslant 1$ is a sufficiently small tolerance factor. This modification was not done because errors such as those shown in Figure 9.10 were only observed in the case of extreme deformations and condition (9.26), with small ε, would make the algorithm too inefficient.

Figure 9.11 shows the four possible combinations between the flow mapping and the topology correction algorithms. It uses one of the surfaces that was shown in Figure 8.8. The localised version of the topology correction algorithm was employed (Section 8.5). The flow has an expression similar to (9.24) with a gradient noise function f_N and an intensity $\alpha = 0.1$. The noise features are 16 times smaller than the sphere radius. When flow mapping is applied, the rays are deformed and the surface on the top left corner of Figure 9.11 is transformed into the surface on the top right corner. Flow mapping can be applied both with or without topology correction. In the case where it is applied with topology correction, the connectivity testing algorithm works over the undeformed surface and proceeds in exactly the same way as it did for

the example of Figure 8.8, resulting in the surface on the bottom left corner of Figure 9.11. This surface was obtained by ray casting with deformed rays, followed by topology correction. All the surface transformations take place inside the ray-surface intersection algorithm for every pixel. Because the intersection algorithm returns the distance t to each intersection point along the trajectory of the deformed ray, the surface on the bottom right corner of Figure 9.11 is displayed once the rays have been straightened out back to their original shape.

Figure 9.12 shows two versions of the topologically corrected landscape of Figure 8.14 with flow mapping. The top image uses the same cellular texture flow that was used in Figure 9.8 with an intensity $\alpha = 3 \times 10^{-6}$. The bottom image uses a fractional Brownian motion flow, obtained by adding five layers of gradient noise with the procedural rescale-and-add method (Section 3.2.4). The Hurst exponent is $H = 0.8$, corresponding to a fractal dimension of 2.2. The intensity of the flow is $\alpha = 1 \times 10^{-5}$. Both flows were modulated by altitude with an auxiliary function h:

$$\mathbf{u}(\mathbf{x}) = \alpha h(\|\mathbf{x}\|) \begin{bmatrix} f_N(\beta(\mathbf{x} + \Delta_1)) \\ f_N(\beta(\mathbf{x} + \Delta_2)) \\ f_N(\beta(\mathbf{x} + \Delta_3)) \end{bmatrix}. \tag{9.27}$$

The function h switches smoothly from 1.0 to 0.0 as the altitude decreases, allowing the bottom part of the landscape to remain flat and unaffected by the flow mapping. The spatial scaling parameter β is 32×10^4 for the top image and 8×10^4 for the bottom image. In the case of the bottom image, the five layers of noise for the flow field add up with the base layer of the terrain that has been topologically corrected, combining into a procedural terrain that has a total of six layers of noise. Performing topology correction on a large terrain such as this with so many layers of noise would have been impractical due to the large number of critical points present. By performing topology correction on the first layer only and allowing the subsequent layers to deform the surface, one obtains a more practical rendering algorithm. The top image in Figure 9.12 took fourteen hours to render and the bottom image took one day to render with a distributed multi-processor implementation of the ray tracer.

Figure 9.13 shows a planetary terrain landscape featuring a single isolated overhang at the centre of the image. This figure illustrates the possibility of exercising additional control over the terrain so that overhanging features are present only in desired areas. The terrains of Figure 9.12, by contrast, exhibit an extreme variety of overhangs everywhere, which is good to test the algorithms that have been developed but also adds a certain degree of surrealism to the scene. The same atmospheric lighting model that was used in Figure 9.12 is also used in Figure 9.13 but with different parameters to make the atmosphere look thinner [Nishita et al., 1993]. Additional lighting effects have also been inserted to increase the realism of the image, namely, indirect diffuse illumination, atmospheric shadows and soft surface shadows. The moon on the upper right is generated with a procedurally displacement mapped surface, using the same techniques that are used to model displacement mapped planetary terrains. The moon's terrain also has a procedural colour texture that is made to depend on terrain altitude so that low lying areas have a brownish colour while higher areas become progressively whiter. The reflectance model applied to the moon's surface simulates the reflective behaviour of materials that are covered by a thick layer of dust [Blinn, 1982b].

Figure 9.14 illustrates how the overhang of Figure 9.13 was generated. The terrain is the same displacement map of Figure 3.17. A cylindrical volume of influence, shown with a

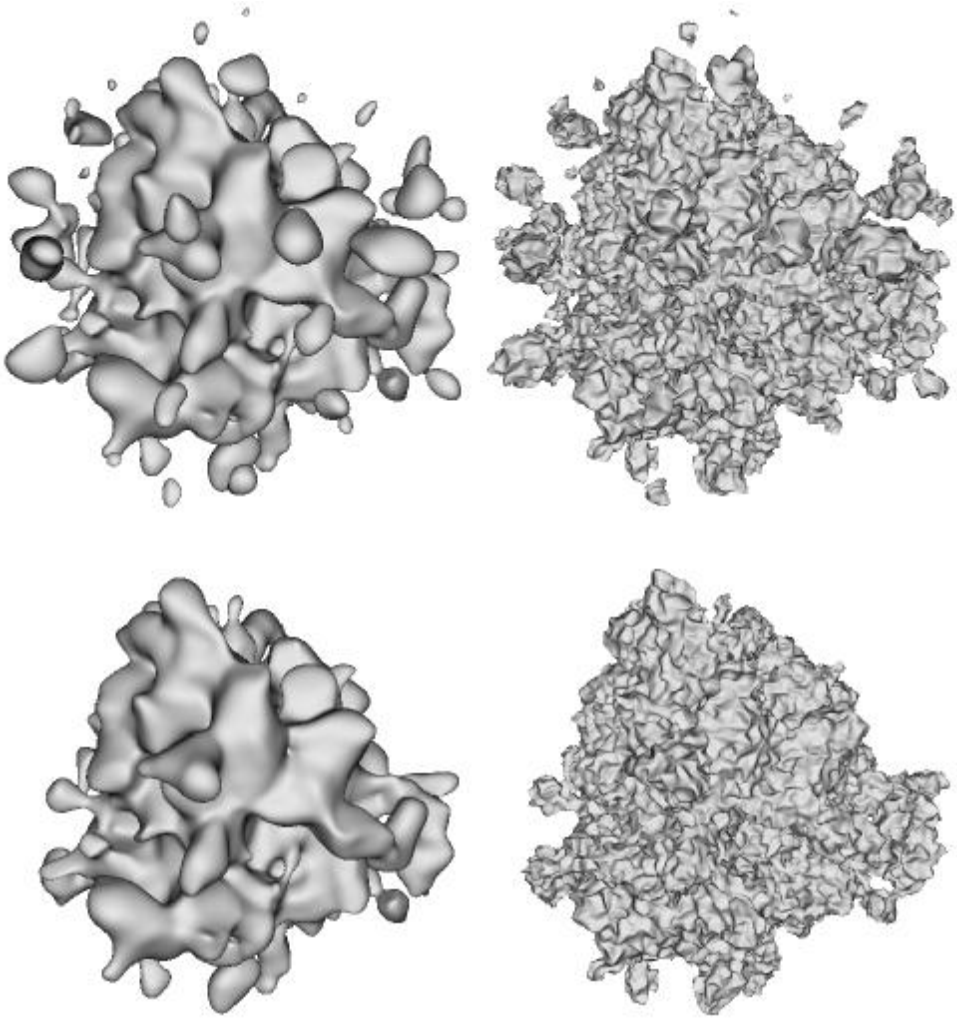


Figure 9.11: Flow mapping combined with topology correction. Flow mapping is shown on the surfaces of the right column only. Topology correction is shown on the surfaces of the bottom row only. From left to right, top to bottom, the rendering times are 31s, 84m 50s, 30m 51s and 192m 20s, respectively.

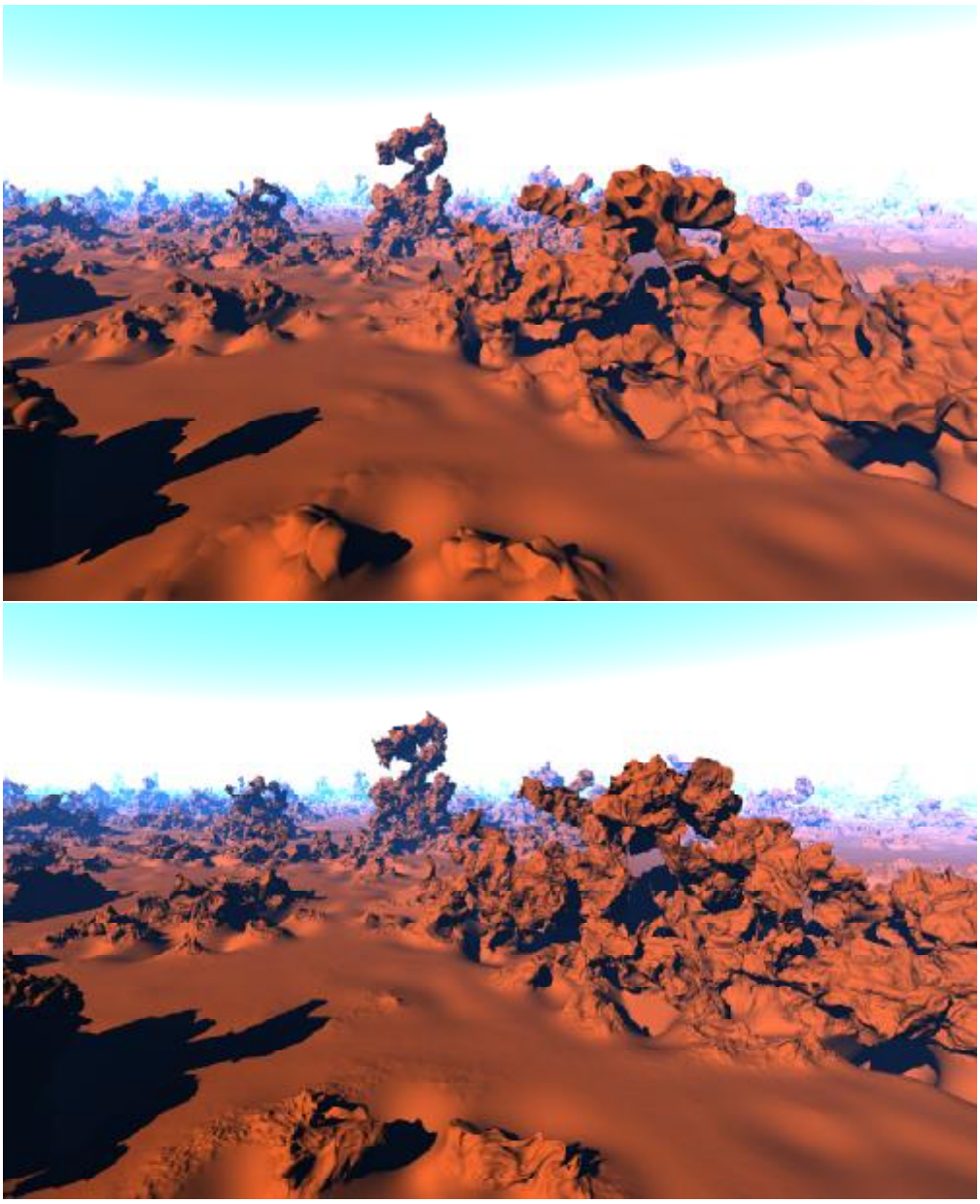


Figure 9.12: Two versions of a topologically corrected terrain, flow mapped with two different flow fields. Note that flow mapping also affects the shadows.

semi-transparent yellow colour, was placed vertically over a chosen area of the terrain and an implicit function was created inside of this cylinder. The expression for $f(f_0(\mathbf{x}), \mathbf{x})$ is:

$$f(f_0(\mathbf{x}), \mathbf{x}) = f_0(\mathbf{x}) + f_D(\mathbf{x}/\|\mathbf{x}\|) - \\ - 1.2 \times 10^{-5} h^2\left(0.66 \times 10^6\left(\frac{\mathbf{x}}{\|\mathbf{x}\|} - \mathbf{x}_0\right)\right) f_N^2(0.66 \times 10^6 \mathbf{x}), \quad (9.28)$$

where f_0 is, as usual, the function for a sphere of unit radius, f_D is the function generating the displacement mapped terrain of Figure 3.17 and f_N is a Perlin noise. The cylinder was placed vertically over a chosen point $\mathbf{x}_0$ located on the surface of the unit sphere. This chosen point corresponds in Figures 9.14(a) and 9.14(b) to the intersection of the cylinder's axis (shown as a white line) with the unit sphere, which is invisible under the terrain. In fact, the transparent solid shown in yellow is not really a cylinder but a cone with origin at the centre of the planet and with a very small angle of aperture. It is only because the terrain is seen from a very close distance to the ground that the solid appears to be a cylinder. The volume of the cone is modulated by a squared radial function $h^2(r)$ that is equal to 1 for $r = 0$ and decays smoothly to 0 at $r = 1$. This functional modulation effect creates a maximum intensity of 1 for the squared function f_N^2 that generates the overhang, which then gradually decreases to 0 towards the outer surface of the cone. The squaring of h was intended to create a faster transition from 1 to 0 while the squaring of f_N was intended to make the latter function always positive, thereby guaranteeing, together with the minus sign in equation (9.28), that any overhanging feature will protrude out of the ground rather than grow towards the interior of the planet.

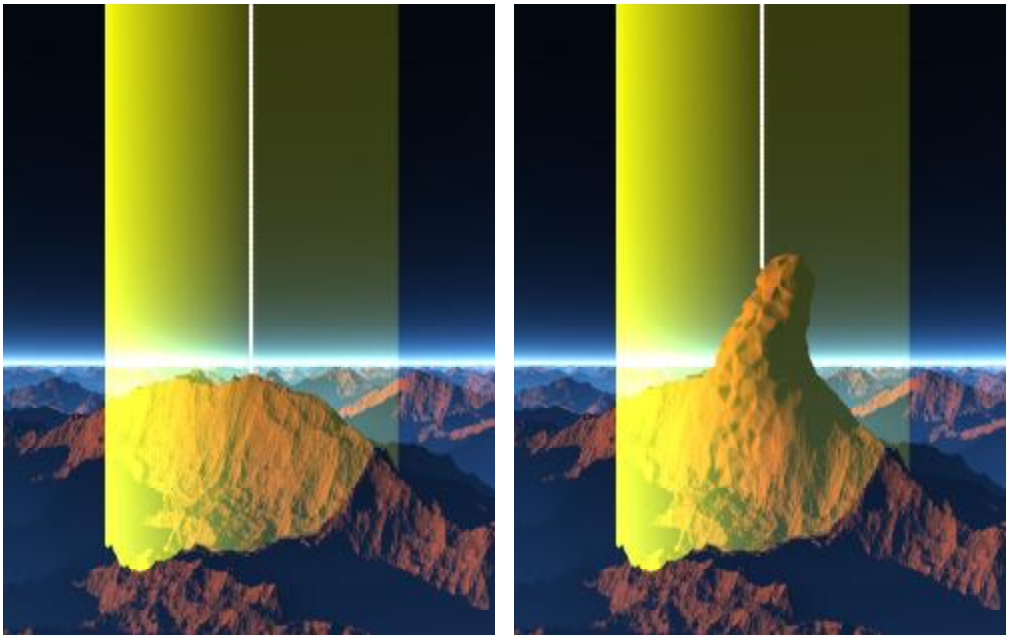
The flow mapping of the overhang was generated with a vector flow field that is always horizontal relative to the surface of the sphere:

$$\mathbf{u}(\mathbf{x}) = 10^{-7} h^2\left(0.66 \times 10^6\left(\frac{\mathbf{x}}{\|\mathbf{x}\|} - \mathbf{x}_0\right)\right)\left(\mathbf{f}_M(10^7 \mathbf{x}) - \left(\mathbf{f}_M(10^7 \mathbf{x}) \cdot \frac{\mathbf{x}}{\|\mathbf{x}\|}\right)\frac{\mathbf{x}}{\|\mathbf{x}\|}\right), \quad (9.29)$$

where the h^2 function is readily seen to be the same as the one used in equation (9.28). This means that the modulation effect influencing the generation of the overhang also influences its deformation by the flow field in the same way. The flow field is given by the function $\mathbf{f}_M$, which is a vector assembled from three independent cellular texture noise functions. The flow field is made parallel to the surface of the sphere by subtracting from $\mathbf{f}_M$ its component that is orthogonal to the sphere. The latter component is obtained from the dot product $\mathbf{f}_M \cdot \mathbf{x}/\|\mathbf{x}\|$, where $\mathbf{x}/\|\mathbf{x}\|$ is the normal to the sphere passing through point $\mathbf{x}$. The intensity of the flow is 10^{-7} and its spatial scaling is 10^7.

Figure 9.13 took four days to render with the same distributed implementation of the ray tracer that was used for Figure 9.12. The rendering time is dominated by two factors. First, there is the computational expense of solving an ordinary differential equation based on the vector field (9.29) for many points along every ray. Second, the full range of atmospheric lighting effects that was considered is computationally very intensive, with no optimisation techniques having been employed. For example, the computation of indirect diffuse lighting on the surface of the terrain was obtained with a brute force method that consists of solving a Monte Carlo integral for every ray-terrain intersection point. The implementation of an irradiance cache would have made this computation significantly more efficient [Ward et al., 1988].

Figure 9.13: An isolated overhang generated above a displacement mapped terrain. The moon is another displacement map with a colour texture that is procedurally defined based on height.



(a) Without overhang (b) With overhang

Figure 9.14: The terrain of Figure 9.13 with and without the overhang. The yellow transparent cylinder bounds the volume where the implicit function that generates the overhang is active.

9.4 Summary

Flow mapping is a conceptually simple technique that allows additional detail to be added on top of a topologically corrected surface. A surface can be flow mapped by supplying the expression for a procedural vector field, which may resort to procedural noise functions in vector form. These vector noise functions provide the stochastic details that are used to deform the surface. Unlike displacement mapping, flow mapping has good topological properties that allow it to deform a surface whilst maintaining its topological structure. The surface is deformed by following the streamlines of the supplied vector field. This generally requires the numerical solution of differential equations for every surface point that is rendered.

A flow mapped surface is ray cast by applying the flow map to the ray itself, transforming it into a parametric curve and finding the intersection of the curve with the original surface that existed before the deformation was applied. This has two important consequences. The first consequence is that it is difficult to know the exact trajectory of a ray that has been flow mapped, given an arbitrary flow field. Range estimation techniques are not useful due to the large number of computations that go into numerically solving the equation for the deformed ray. Devising a robust ray casting algorithm, therefore, becomes difficult and there is the possibility that some rays may return wrong surface intersections. The results obtained in this chapter indicate that gradient type flows (flows obeying equation (9.12)) are particularly sensitive to rendering errors since they can significantly stretch the surface over a short space. The intensity of gradient flow fields must be kept low to avoid rendering artifacts. In the case of flows where the three vector components are directly specified, no rendering errors were detected in this chapter's results. The flow fields used for these results were made from procedural noise functions that change smoothly, allowing the ray caster to locate intersections reliably. It would be possible, however, to specify other procedural vector fields that are ill-behaved in the sense that they can contain regions with sharp transitions, which would again bring problems to the ray casting algorithm.

A second consequence of performing ray-domain deformation is that every point sampled along the ray needs to have its position computed with a differential equation. Considering that many points need to be sampled whilst finding the intersection point between a ray and the surface, this makes the intersection algorithm computationally intensive. The computational cost is directly related to the amount of deformation that is introduced. The surfaces shown on the top of Figures 9.8 and 9.9, for example, take much less time to render than the surfaces on the bottom of the same Figures. Appendix C presents a distributed ray tracing algorithm that alleviates the rendering time for flow mapped and topology corrected surfaces by distributing the rendering load over many processors. It takes advantage of the fact that rays can be traced independently into the scene. The algorithm can work with several geographically distant high performance computing facilities by employing grid computing techniques.

There are several possible areas for future research concerning the flow mapping algorithm. Beyond the stochastic deformations already presented that act globally over the whole surface, it would be useful to introduce additional localised surface deformations, allowing users to fine tune particular areas of the surface to their liking. Milliron et al. [2002] give a survey of localised geometric deformations and present a unified framework capable of applying them over parametric patches, polygonal meshes and subdivision surfaces. The deformations presented

by Milliron et al. are all topology preserving, just like flow mapping, and it is likely they could be applied to implicit surfaces. Another area of research concerns the implementation of flow mapping on the GPU. The intersection algorithm could easily be implemented as a fragment shader program, allowing the rendering to proceed in parallel. The numerical ODE solver could be coded in a high level shading language. Current GPUs only have single precision arithmetic, which limits the accuracy of the ray caster but this would be greatly offset by the increased efficiency stemming from the parallelism. However, it is expected that future GPUs will have double precision arithmetic thus removing this handicap.

CHAPTER 10

Conclusions

THE main goal of this thesis was to investigate a series of techniques to advance the modelling and rendering of stochastic implicit surfaces. The use of an implicit surface representation, as opposed to the more traditional polygonal mesh representation, leads to a simple and elegant solution to the problem of modelling stochastic surfaces. A first advantage is that there is no need to deal with large and cumbersome lists of vertices and edges. For an implicit stochastic surface, only an appropriate implicit function needs to be defined. A second advantage is that stochastic surfaces with complex topologies arise naturally from the implicit surface representation. Yet another advantage is that no complex algorithms are required to implement adaptive level of detail. The implicit function can receive the distance to the camera as an additional input, allowing a smooth but finely detailed surface to be visualised from any viewpoint.

Tightly coupled with the implicit surface representation is the use of procedural models to define stochastic surfaces. Lewis [1989] was the first to propose a procedural model for the synthesis of stochastic implicit surfaces (the proposed model consisted of a single procedural noise function). More complex procedural models can be obtained with the hypertexturing technique of Perlin and Hoffert [1989]. Hypertexturing was initially applied to fuzzy objects with no defined boundary and required expensive volume rendering algorithms for visualisation. In this thesis, the procedural hypertexturing technique is modified to work over implicit surfaces so that a well defined stochastic object is created with a boundary that separates an interior space from an exterior space.

10.1 Discussion of Results

At the modelling level, a solution to the splitting problem of stochastic implicit surfaces has been presented, which detects and eliminates any disconnected components of the surface. The presence of unwanted disconnected surface components has long been a stumbling block in the generation of stochastic solids, such as rocks or tree bark, with implicit surfaces. The proposed topology correction technique works as a type of surface filter, turning a given initial surface into a simply connected surface. This is a better approach when compared with the possible alternative of introducing constraints into the implicit function so that no surface disconnections occur. By constraining the way the implicit function is built, one is effectively limiting the class of stochastic surfaces that can be generated. With the proposed method, any stochastic surface can be built initially and the topology correction algorithm then takes the responsibility of

removing all the components that may have become disconnected. There are, however, two limitations to the proposed algorithm. One is that it can only be used with implicit functions that are C^2 continuous. The other is that it cannot be used with implicit functions that generate significant amounts of small scale detail due to efficiency concerns.

To alleviate the drawbacks of the topology correction method, a procedural surface deformation technique is introduced to add extra detail to the topology corrected surfaces. The simplest surface deformation techniques, namely displacement mapping and general displacement mapping, cannot be used because they have the potential to introduce topology changes, thereby invalidating the topology correction results. Flow mapping can safely be used as it always defines a surface homeomorphism for any arbitrary but continuous flow field. Flow mapping is more expensive to apply when compared with the two displacement mapping techniques because it requires the solution of ordinary differential equations. The performance gain that is obtained by introducing small surface detail through flow mapping depends on the amount of deformation that is used. Consider the case of a surface made from the superposition of five layers of procedural noise with increasing level of detail. If the amount of surface deformation is small, it is more efficient to perform topology correction on the first layer only, followed by a flow mapping operation with the remaining four layers. As the amount of deformation increases, the performance gain that stems from avoiding a full topology correction decreases. The C^2 continuity requirement for topology correction can also be circumvented with flow mapping by using procedural flow fields that have less than C^2 continuity. The topology correction stage still needs to work with C^2 continuous surfaces but surface features such as creases can then be introduced during the flow mapping stage.

Both the topology correction and the surface deformation algorithms are invoked as part of the ray casting operation that consists of finding the intersection between any view ray and the surface. Although the ray casting of implicit surfaces cannot be considered an efficient operation when compared with the rendering of an approximated polygonal mesh, a reduced affine arithmetic ray casting method was presented that was shown to be faster than any previous ray casting techniques with the same characteristic of robustness. The reduced affine arithmetic ray caster can be applied to any implicit surface by implementing its implicit function in reduced affine arithmetic form. This can be done by invoking calls into a library of common reduced affine arithmetic operations. In the case of ray casting combined with surface deformation, the requirement of robustness, i.e. the guarantee that the correct intersection point is always found, had to be relinquished. The development of a range estimation technique that can withstand the numerical solution of a differential equation and still produce useful range estimates would solve this problem. A range estimation technique based on truncated Taylor series expansions may be a possibility [Nedialkov et al., 2004]. Accuracy will increase as more terms are included in the series but the computational expense and the implementation complexity will increase accordingly.

A Monte Carlo anti-aliasing algorithm was presented that builds upon the ray casting algorithm for implicit surfaces. The anti-aliasing algorithm uses an elegant recursive formulation that allows any member from a family of B-spline basis curves to be used as the anti-aliasing filter. The algorithm relies only on the ray casting of random samples through the image plane and so does not impose any restrictions on the type of surfaces that can be visualised. The algorithm also generalises to sampling in the time domain (implementing motion blur) in a trivial manner.

Two progressive refinement rendering algorithms were developed in order to provide a preview of a stochastic implicit surface in real time with increasing quality. Both algorithms perform sample subdivision in the image plane and rely on object space surface classification to speed up the convergence of the rendering. The more efficient of the two algorithms is applied to surfaces that are Lipschitz continuous. The other can be applied to any surface and relies on a reduced affine arithmetic representation in three dimensions. The two algorithms are meant to be used as previewers during the design stage of a new implicit function, to check if the resulting implicit surface is taking on the desired shape. This is a better alternative to ray casting the surface and waiting for the rendering to complete in order to evaluate the result. It would be interesting to see if the two progressive refinement algorithms could be extended to provide production quality rendering results. The surface classification that is performed in object space as part of the two algorithms solves the hidden surface aspect of the rendering problem more quickly than ray casting, which needs to shoot rays independently for every pixel. The rendering of lighting effects such as shadows, reflections and refractions, could then be deferred until the hidden surface problem had been solved. This line of research was not pursued in the thesis because the topology correction algorithm and especially the surface deformation algorithm do not fit easily in a progressive refinement rendering scheme. The two proposed algorithms are, therefore, used to preview an approximation of a surface where the topology correction and surface deformation algorithms have been turned off. In the case of the Lipschitz based progressive refinement algorithm, topology correction and surface deformation can be delayed until the final ray casting step that is applied for every pixel.

The overriding concern while performing the research for this thesis was one of generality. Algorithms were chosen for each particular task that could be used for as large a class of stochastic implicit surfaces as possible. Whenever a choice had to be made between a general algorithm and a more efficient algorithm that only works for a specific subset of surfaces the general algorithm was chosen so as not to constrain the design of new surfaces. Sometimes both algorithms were developed as in the case of the two progressive refinement rendering algorithms (where one of them is only valid for the common subset of Lipschitz continuous surfaces). The increased computational cost of the more general algorithms is offset through a distributed rendering strategy, as explained in Appendix C. The only major constraint that could not be avoided was the C^2 continuity for topologically corrected surfaces. This was, however, alleviated with the use of the surface deformation algorithm.

10.2 Assessment of the Procedural Hypertexturing Approach

Hypertexturing based on a procedural model is a powerful tool for the generation of stochastic implicit surfaces. The number of surfaces that can be generated is as large as the number of different hypertexturing functions that can be designed, which is to say that it is infinite. Not only can individual functions be composited in different ways to generate new hypertexturing functions but each hypertexturing function can also have many variations on the values of its parameters. This extreme flexibility comes at a cost. As already mentioned in Section 3.4, the two drawbacks of the hypertexturing approach are the difficulty in setting up the right function that will generate some desired surface shape and the computational expense of rendering a hypertextured surface.

The design problem for hypertextures is derived from its purely procedural approach. This often requires a mathematically minded designer that has some intuition for what each particular function is going to create in terms of surface shapes. Previous experience working with procedural models is also advantageous even if it was only obtained in designing procedural surface textures such as marble or wood. Users can sometimes find the experience of experimenting with different functions daunting. Even when a hypertexturing function is provided and only the function's parameters need to be adjusted, the exploration of a large parameter space can often be frustrating or counter-intuitive. At the same time, the rendering complexity of hypertextures creates long design cycles where the iterative adjustment of a hypertexturing function tends to proceed slowly. For these two reasons, hypertexturing has not been a popular modelling tool. Since its introduction by Perlin and Hoffert [1989], there have been only a handful of papers published on the topic of surface hypertexturing.

This thesis has addressed the problem of reducing the rendering complexity for hypertextured surfaces. This was done in two ways: by reducing the time to ray cast a surface robustly and by providing a quick real time previewer for hypertextured implicit surfaces. The problem of hypertexture design, however, was not addressed. This problem can potentially be solved in three different ways. By increasing level of complexity, these are:

- A pre-defined library of useful hypertexturing functions. The user is then free to modify or mix the functions together to generate new surfaces.
- A genetic algorithm that randomly mutates functions and combines pairs of functions. The user selects the new functions that seem more interesting, causing the algorithm to evolve a hypertexturing function towards some design goal. Attempts in this direction have been done by Sims [1993] and Musgrave [2003b].
- An example-based algorithm that given a stochastic surface as an example could synthesise hypertexturing functions that generate a class of surfaces similar to the given example. Although this type of example-based algorithms has received considerable attention for the synthesis of digital textures, its application to procedural models has not been undertaken so far.

10.3 Future Developments

Many suggestions for possible future improvements have already been given at the end of the preceding chapters. These include the use of isotropic and adaptive anti-aliasing filters in Chapter 6, the use of the progressive rendering mechanisms of Chapter 7 as part of a high quality anti-aliased renderer, the use of the reduced affine arithmetic technique of Chapter 5 for faster location of Morse critical points in Chapter 8 and the use of localised and user-controlled flows for surface deformation in Chapter 9.

This section concentrates on future developments that have a broader scope. An interesting possibility is the use of GPUs to speed up some of the more expensive aspects of the algorithms that have been developed. The research for this thesis was developed over a time of great change in the way that GPUs are used for computer graphics, with an increasing number of algorithms being implemented on a GPU. The ray casting algorithm is a prime candidate for

GPU implementation because the rays through each pixel can be traced independently. This applies to the sphere tracing algorithm of Section 5.2, the ray casting algorithm with reduced affine arithmetic of Section 5.4 and the ray casting algorithm of Section 9.3 that incorporates surface deformation. It should be noted that some GPU implementations of sphere tracing and reduced affine arithmetic ray casting have already been presented [Liktor, 2008; Knoll et al., 2009]. The location of Morse critical points for topology correction, based on recursive spatial subdivision, is also a candidate for GPU implementation (Section 8.4.1). All the spatial cells at the same level of subdivision can be analysed in parallel on the GPU. The algorithm would then subdivide space in a breadth-first manner, one level of subdivision at a time. This could be applied in the case of the global topology correction algorithm of Section 8.4, where the location of critical points is done before ray casting begins. Considering that the location of critical points is the major bottleneck of the topology correction algorithm, a GPU implementation should provide significant speed ups.

Another possibility is to extend current global illumination algorithms so that they can interact with stochastic surfaces over very large scales. The application that comes to mind in this context is the visualisation of planetary terrains by taking into account the correct transmission and scattering of light in the atmosphere. Most global illumination algorithms tend to work over small spaces and preferably in enclosed environments. This facilitates the use of illumination data structures such as irradiance caches and photon maps that can only be applied over limited volumes due to memory concerns. An extension of global illumination algorithms could be made, perhaps using spatially adaptive illumination data structures, that would not have to be artificially limited to small spaces. Again, a GPU implementation would be a valuable feature of such illumination algorithms, given that ray casting would be the main rendering method for large terrains. A promising development in this direction has recently been presented by Bruneton and Neyret [2008].

Finally, it would be desirable to explore new ways of generating stochastic surfaces, either in implicit form or otherwise. The method of spectral synthesis that was explained in Section 3.2.2 can create an instance of a stochastic surface that obeys a given power spectrum density function (PSDF). This method is conceptually quite powerful because it allows new stochastic surfaces to be developed simply by designing a PSDF curve if the surface is isotropic. Anisotropy can be included by the multiplication of the PSDF with another curve that describes the angular dependency of the power spectrum. Spectral synthesis, however, relies on the FFT algorithm and this traditionally meant that spectral synthesis could only be performed on gridded surface data. Using yet again the new power of GPUs, it is possible to have a procedural spectral synthesis model where any arbitrary point on the surface can be obtained as the result of a local evaluation of a discrete Fourier transform operator. Implementations of FFTs on GPUs have now been developed and this should allow a seemingly expensive point-wise FFT evaluation to become feasible [Moreland and Angel, 2003; Govindaraju et al., 2008]. Spectral synthesis generates statistically stationary surfaces. In the case of a procedural terrain this can lead to unrealistic results since the terrain looks essentially the same everywhere. The problem can be avoided by making the PSDF vary in space, effectively changing the statistical characteristics of the terrain across different regions. The relationship between this spatially-varying PSDF and the stochastic terrain is no longer given by a Fourier transform and requires mathematical tools from the field of Time-Frequency Analysis [Cohen, 1995].

APPENDIX A

Recursive B-Spline Filters

We begin by repeating the equations (6.15) from Chapter 6 that describe the behaviour of the integral $N_m(x)$ of a B-spline basis function $n_m(x)$ of order $m > 1$:

$$\text{supp}\, N_m(x) = [0, +\infty[. \tag{A.1a}$$

$$N_m(x) \text{ increases from } N_m(0) = 0 \text{ to } N_m(+\infty) = 1. \tag{A.1b}$$

$$n_m(x) = N_{m-1}(x) - N_{m-1}(x-1), \quad \text{for } m > 1. \tag{A.1c}$$

$$N_m(x) = \frac{x}{m} N_{m-1}(x) + \left(1 - \frac{x}{m}\right) N_{m-1}(x-1), \quad \text{for } m > 1. \tag{A.1d}$$

Equations (A.1a) and (A.1b) have a straightforward derivation from the fact that $N_m(x) = \int n_m(\xi) d\xi$, together with the properties for $n_m(x)$ given in equations (6.11). The relation (A.1c) between $n_m(x)$, a B-spline of degree m, and $N_{m-1}(x)$, the integral of a B-spline of degree $m-1$ is now derived. To do so, $n_m(x)$ is expressed as an mth-degree finite difference of powers of x. Take the notation $x_+ = \max(x, 0)$ to represent the restriction of x to positive values only. Similarly, $x_+^m = (x_+)^m$. Consider also the finite differences of degree m for some function $f(x)$. These are defined with the following recurrence relation:

$$(\Delta f)(x) = f(x) - f(x-1), \tag{A.2}$$

$$\left(\Delta^m f\right)(x) = \left(\Delta^{m-1}(\Delta f)\right)(x). \tag{A.3}$$

Armed with this notation, it is possible to write a B-spline $n_m(x)$ as:

$$n_m(x) = \frac{1}{(m-1)!} \Delta^m x_+^{m-1}. \tag{A.4}$$

For more details of how this equality can be reached, refer to Chui [1992].

Taking the integral of (A.4), we arrive at a similar equation for $N_m(x)$:

$$\begin{aligned} N_m(x) &= \int_0^x n_m(t)\,dt = \frac{1}{(m-1)!}\int_0^x \Delta^m t_+^{m-1}\,dt \\ &= \frac{1}{(m-1)!}\int_0^x \Delta^{m-1}\left\{t_+^{m-1} - (t-1)_+^{m-1}\right\}\,dt \\ &= \frac{1}{(m-1)!}\,\Delta^{m-1}\left\{\int_0^x t_+^{m-1}\,dt - \int_0^x (t-1)_+^{m-1}\,dt\right\} \\ &= \frac{1}{(m-1)!}\,\Delta^{m-1}\left\{\int_0^x t_+^{m-1}\,dt - \int_{-1}^{x-1} t_+^{m-1}\,dt\right\} \\ &= \frac{1}{(m-1)!}\,\Delta^{m-1}\left\{\int_0^x t_+^{m-1}\,dt - \int_0^{x-1} t_+^{m-1}\,dt\right\} \\ &= \frac{1}{(m-1)!}\,\Delta^m \int_0^x t_+^{m-1}\,dt = \frac{1}{m!}\,\Delta^m x_+^m. \end{aligned} \tag{A.5}$$

Using (A.5) twice, we have:

$$\begin{aligned} N_{m-1}(x) - N_{m-1}(x-1) &= \frac{1}{(m-1)!}\,\Delta^{m-1}x_+^{m-1} - \frac{1}{(m-1)!}\,\Delta^{m-1}(x-1)_+^{m-1} \\ &= \frac{1}{(m-1)!}\,\Delta^{m-1}\left\{x_+^{m-1} - (x-1)_+^{m-1}\right\} \\ &= \frac{1}{(m-1)!}\,\Delta^m x_+^{m-1} = n_m(x). \end{aligned} \tag{A.6}$$

This completes the derivation of (A.1c). We now take the recurrence relation that exists for $n_m(x)$ and arrive at a similar relation for $N_m(x)$, expressed in (A.1d). The recurrence relation for $n_m(x)$, copied from (6.11d), is:

$$n_m(x) = \frac{x}{m-1}\,n_{m-1}(x) + \frac{m-x}{m-1}\,n_{m-1}(x-1). \tag{A.7}$$

An integral is placed on both sides of (A.7) and an integration is performed by parts:

$$\begin{aligned} N_m(x) &= \int_0^x n_m(t)\,dt = \int_0^x \frac{t}{m-1}\,n_{m-1}(t)\,dt + \int_0^x \frac{m-t}{m-1}\,n_{m-1}(t-1)\,dt \\ &= \frac{t}{m-1}\,N_{m-1}(t)\Big|_0^x + \frac{m-t}{m-1}\,N_{m-1}(t-1)\Big|_0^x \\ &\quad - \frac{1}{m-1}\int_0^x N_{m-1}(t)\,dt + \frac{1}{m-1}\int_0^x N_{m-1}(t-1)\,dt \\ &= \frac{x}{m-1}\,N_{m-1}(x) + \frac{m-x}{m-1}\,N_{m-1}(x-1) \\ &\quad - \frac{1}{m-1}\int_0^x \{N_{m-1}(t) - N_{m-1}(t-1)\}\,dt. \end{aligned} \tag{A.8}$$

Replacing (A.1c) in (A.8), we get:

$$\begin{aligned} N_m(x) &= \frac{x}{m-1}\,N_{m-1}(x) + \frac{m-x}{m-1}\,N_{m-1}(x-1) - \frac{1}{m-1}\int_0^x n_m(t)\,dt = \\ &= \frac{x}{m-1}\,N_{m-1}(x) + \frac{m-x}{m-1}\,N_{m-1}(x-1) - \frac{1}{m-1}\,N_m(x). \end{aligned} \tag{A.9}$$

Sending the $N_m(x)$ term on the right to the left hand side of equation (A.9) and rearranging terms, we finally arrive at:

$$N_m(x) = \frac{x}{m} N_{m-1}(x) + \left(1 - \frac{x}{m}\right) N_{m-1}(x-1). \qquad \text{(A.10)}$$

This completes the derivation of (A.1d).

APPENDIX B

State Equations for Flow Mapping

THE deformation of an implicit surface through flow mapping is performed by solving a differential equation in three dimensions for every point on the surface. The canonical expression for any set of ordinary differential equations (ODEs) is $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, where the n-dimensional state vector $\mathbf{y}$ groups all the scalar variables that need to be solved and where $\dot{\mathbf{y}}$ represents the time derivative of the state vector. The vector function $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^n$ expresses the dynamics of each particular set of ODEs that is being solved. In the case of surface flow mapping, one needs to solve an ODE for a large collection of points along a ray when trying to find a surface intersection. Each point corresponds to a three-dimensional state vector $\mathbf{y} = (y_1, y_2, y_3)$. If the flow field $\mathbf{u} = (u_1, u_2, u_3)$ is directly specified in terms of its three components then the state equations for flow mapping are simply:

$$\begin{aligned} \dot{y}_1 &= u_1(y_1), \\ \dot{y}_2 &= u_2(y_2), \\ \dot{y}_3 &= u_3(y_3). \end{aligned} \tag{B.1}$$

If the flow field is the gradient of a scalar function $\phi : \mathbb{R}^3 \rightarrow \mathbb{R}$, as specified in equation (9.12), the state equations are:

$$\begin{aligned} \dot{y}_1 &= \frac{\partial \phi}{\partial x_1}(y_1), \\ \dot{y}_2 &= \frac{\partial \phi}{\partial x_2}(y_2), \\ \dot{y}_3 &= \frac{\partial \phi}{\partial x_3}(y_3). \end{aligned} \tag{B.2}$$

Note that although partial derivatives appear in (B.2), it is still an ODE because the state variables y_1, y_2 and y_3 are only differentiated relative to time. The system of ODEs is integrated, starting from the initial point $\mathbf{y}(0) = \mathbf{x}$, and the solution is obtained for $\mathbf{d}(\mathbf{x}) = \mathbf{y}(1)$, where $\mathbf{d}(\mathbf{x})$ is the flow mapped deformation of point $\mathbf{x}$.

Once an intersection point between a ray and the flow mapped surface has been found, the surface normal needs to be computed. As explained in Section 9.2, this requires computing the Jacobian matrix $\mathcal{J}\{\mathbf{d}\}$ of the flow map. The system of ODEs needs to be augmented to include the 3×3 entries of the Jacobian matrix leading to a total of 12 state variables y_1 to y_{12}. The variables y_1 to y_3 represent, as before, the positional coordinates of a point undergoing

deformation. The remaining variables represent the entries in the Jacobian matrix in row major order:

$$\mathcal{J}\{\mathbf{d}\} = \begin{bmatrix} y_4 & y_5 & y_6 \\ y_7 & y_8 & y_9 \\ y_{10} & y_{11} & y_{12} \end{bmatrix} \tag{B.3}$$

The state equations for a general flow field are:

$$\begin{aligned}
\dot{y}_1 &= u_1(y_1), \\
\dot{y}_2 &= u_2(y_2), \\
\dot{y}_3 &= u_3(y_3), \\
\dot{y}_4 &= \frac{\partial u_1}{\partial x_1} y_4 + \frac{\partial u_1}{\partial x_2} y_5 + \frac{\partial u_1}{\partial x_3} y_6, \\
\dot{y}_5 &= \frac{\partial u_2}{\partial x_1} y_4 + \frac{\partial u_2}{\partial x_2} y_5 + \frac{\partial u_2}{\partial x_3} y_6, \\
\dot{y}_6 &= \frac{\partial u_3}{\partial x_1} y_4 + \frac{\partial u_3}{\partial x_2} y_5 + \frac{\partial u_3}{\partial x_3} y_6, \\
\dot{y}_7 &= \frac{\partial u_1}{\partial x_1} y_7 + \frac{\partial u_1}{\partial x_2} y_8 + \frac{\partial u_1}{\partial x_3} y_9, \\
\dot{y}_8 &= \frac{\partial u_2}{\partial x_1} y_7 + \frac{\partial u_2}{\partial x_2} y_8 + \frac{\partial u_2}{\partial x_3} y_9, \\
\dot{y}_9 &= \frac{\partial u_3}{\partial x_1} y_7 + \frac{\partial u_3}{\partial x_2} y_8 + \frac{\partial u_3}{\partial x_3} y_9, \\
\dot{y}_{10} &= \frac{\partial u_1}{\partial x_1} y_{10} + \frac{\partial u_1}{\partial x_2} y_{11} + \frac{\partial u_1}{\partial x_3} y_{12}, \\
\dot{y}_{11} &= \frac{\partial u_2}{\partial x_1} y_{10} + \frac{\partial u_2}{\partial x_2} y_{11} + \frac{\partial u_2}{\partial x_3} y_{12}, \\
\dot{y}_{12} &= \frac{\partial u_3}{\partial x_1} y_{10} + \frac{\partial u_3}{\partial x_2} y_{11} + \frac{\partial u_3}{\partial x_3} y_{12}.
\end{aligned} \tag{B.4}$$

The partial derivatives of **u** in (B.4) are evaluated at the point (y_1, y_2, y_3).

If the flow mapping employs a gradient flow field $\nabla\phi$, the Hessian matrix $\mathcal{H}\{\phi\}$ of the field needs to be evaluated. The Hessian matrix is symmetric and contains all the second order partial derivatives $\partial^2\phi/\partial x_i \partial x_j$ of ϕ for $i, j = 1, 2, 3$. The state equations become:

$$
\begin{aligned}
\dot{y}_1 &= \frac{\partial \phi}{\partial x_1}(y_1), \\
\dot{y}_2 &= \frac{\partial \phi}{\partial x_2}(y_2), \\
\dot{y}_3 &= \frac{\partial \phi}{\partial x_3}(y_3), \\
\dot{y}_4 &= \frac{\partial^2 \phi}{\partial x_1^2} y_4 + \frac{\partial^2 \phi}{\partial x_1 \partial x_2} y_5 + \frac{\partial^2 \phi}{\partial x_1 \partial x_3} y_6, \\
\dot{y}_5 &= \frac{\partial^2 \phi}{\partial x_1 \partial x_2} y_4 + \frac{\partial^2 \phi}{\partial x_2^2} y_5 + \frac{\partial^2 \phi}{\partial x_2 \partial x_3} y_6, \\
\dot{y}_6 &= \frac{\partial^2 \phi}{\partial x_1 \partial x_3} y_4 + \frac{\partial^2 \phi}{\partial x_2 \partial x_3} y_5 + \frac{\partial^2 \phi}{\partial x_3^2} y_6, \\
\dot{y}_7 &= \frac{\partial^2 \phi}{\partial x_1^2} y_7 + \frac{\partial^2 \phi}{\partial x_1 \partial x_2} y_8 + \frac{\partial^2 \phi}{\partial x_1 \partial x_3} y_9, \\
\dot{y}_8 &= \frac{\partial^2 \phi}{\partial x_1 \partial x_2} y_7 + \frac{\partial^2 \phi}{\partial x_2^2} y_8 + \frac{\partial^2 \phi}{\partial x_2 \partial x_3} y_9, \\
\dot{y}_9 &= \frac{\partial^2 \phi}{\partial x_1 \partial x_3} y_7 + \frac{\partial^2 \phi}{\partial x_2 \partial x_3} y_8 + \frac{\partial^2 \phi}{\partial x_3^2} y_9, \\
\dot{y}_{10} &= \frac{\partial^2 \phi}{\partial x_1^2} y_{10} + \frac{\partial^2 \phi}{\partial x_1 \partial x_2} y_{11} + \frac{\partial^2 \phi}{\partial x_1 \partial x_3} y_{12}, \\
\dot{y}_{11} &= \frac{\partial^2 \phi}{\partial x_1 \partial x_2} y_{10} + \frac{\partial^2 \phi}{\partial x_2^2} y_{11} + \frac{\partial^2 \phi}{\partial x_2 \partial x_3} y_{12}, \\
\dot{y}_{12} &= \frac{\partial^2 \phi}{\partial x_1 \partial x_3} y_{10} + \frac{\partial^2 \phi}{\partial x_2 \partial x_3} y_{11} + \frac{\partial^2 \phi}{\partial x_3^2} y_{12}.
\end{aligned}
\tag{B.5}
$$

As in the previous case, all partial derivatives of ϕ are evaluated at (y_1, y_2, y_3). Although equations (B.4) and (B.5) are costly to integrate, they only need to be solved once for each ray-surface intersection point. Their computational expense, therefore, is negligible when compared to integrating equations (B.1) or (B.2) for a large number of trial points along each individual ray.

APPENDIX C

Grid Rendering

GRID computing is a new computational model that exploits the availability of high performance computer nodes across a heterogeneous and geographically dispersed network. Many scientific and technological problems are known to have such a computational complexity that only the most powerful computers can solve them [Levin, 1989]. In the early years of computing, such powerful computers were built by individual companies using their own proprietary hardware and featuring thousands of processors inside a single computer. During subsequent years, a more flexible and cost-effective approach was taken whereby a large computing facility was assembled out of a collection of smaller computers that used commercially available processors, from companies like Intel or AMD, and that were connected together through a high speed network. Techniques for tapping the power of these computer clusters led to the development of High Performance Computing (HPC) as a new field of Computer Science. Libraries that implement the Message Passing Interface (MPI) protocol allow a programmer to build applications that can be distributed across several networked computers [Gropp et al., 1999]. For HPC installations that feature multi-processor computers, where groups of processors have access to a common memory space, the OpenMP standard can also be used to implement parallelism inside an application [Chandra et al., 2000]. At a higher level, job schedulers like the Sun Grid Engine or Condor are put in control of HPC clusters to mediate the access of users to the resources and to manage the load on the cluster [Gentzsch, 2001; Thain et al., 2005].

HPC installations are normally found at universities and government funded research laboratories due to the high cost of acquiring and maintaining this type of hardware. Traditionally, access to these HPC installations was restricted to students and staff of each university or laboratory. More recently, a new trend has emerged where a group of institutions agrees on a protocol that allows accredited researchers to have Internet access to their joint set of HPC computing facilities. This joint set of facilities is called a *grid* and features potentially dozens of HPC nodes from different institutions that may be spread across a large geographical area. The emergence of grids led to the development of *grid computing* as a new computational model that can be regarded as being one step above high performance computing. Issues that would not normally worry the user of a single HPC node become significant when dealing with grid applications. Such issues are related to the execution of common tasks like secure access or job management in a transparent manner across the grid. Ultimately, the issues that grid computing must solve arise as a consequence of the extreme level of heterogeneity that can be found in a grid. This is because each grid participating institution has its own type of HPC platform and implements its own management policies for users of that platform.

C.1 Grid Middleware

As part of the grid computing model, an intermediate layer of software, called *grid middleware*, must exist between a grid application and the grid nodes. Grid middleware addresses the problems created by the heterogeneity that is inherent to grids. It is responsible for providing a common access infrastructure to all grid services while, at the same time, hiding from the user the details of how these services are implemented in each grid node. The following list describes the most common services that are required from the grid:

Authentication and Secure Access Access of users to grid services must be properly authenticated. The communication between a grid application and the grid nodes must be encrypted, considering that it takes place over the Internet.

Resource Location A user must be able to query the grid to find diverse information such as what grid nodes are available, what services they offer, what is their hardware configuration and what is their load at any time.

Job Management With job management services, individual jobs or sequences of jobs can be issued to the grid. The state of these jobs can then be queried and jobs can be deleted, suspended or migrated to other nodes.

Data Retrieval Typically, after a job completes in a grid node, a file containing the output data is left on the node's file system. A grid application must be able to gather all the data files that were left on the grid nodes after a distributed computation has finished.

A standard called Open Grid Services Architecture (OGSA) is under development and specifies a set of requirements for grid services that compliant grid middleware is expected to obey [Foster et al., 2002]. One implementation of grid middleware that satisfies many of the requirements set forth in the OGSA is the Globus Toolkit [Foster and Kesselman, 1997]. The Globus Grid Security Infrastructure (GSI) manages user authentication and data encryption. User authentication in Globus is done with X509v3 digital certificates that are issued by a Globus recognised certification authority. When a user wishes to access grid services through Globus, he requests a grid proxy based on his certificate. Grid proxies have a limited validity. The user is given transparent and encrypted access to the grid while the proxy is valid. When a grid proxy expires a new proxy must be requested. Resource location in Globus is implemented by the Monitoring & Discovery System (MDS) that uses a Web Services interface. The Globus Grid Resource Allocation and Management module (GRAM) supports job management. GRAM exists as a collection of command line tools but it also provides language bindings for programming languages such as C/C++, Java and Python. Data retrieval in Globus is performed with its Global Access to Secondary Storage (GASS) module, which implements the GridFTP protocol.

In many applications, a grid user does not have to be aware of the individual HPC nodes that are part of the grid. The user should be able to launch distributed applications transparently across the grid as if he was working with a single virtual HPC installation. This degree of abstraction is achieved by having a *meta-scheduler* that uses the existing grid middleware to reroute the user jobs to individual grid nodes. The meta-scheduler knows which nodes are available at

every moment and what kind of services they are able to provide. When a user submits jobs for some particular computational task, the meta-scheduler selects the best nodes on the grid for that type of task and passes the jobs to the relevant job schedulers. Meta-schedulers have to be built with specific grid applications in mind because each application has its own scheduling policies. The Globus Alliance, the same software group that develops the Globus Toolkit, also offers the GridWay Toolkit to help in the implementation of meta-schedulers [Hudo et al., 2005]. GridWay is an extra layer of grid middleware that exists on top of the Globus Toolkit. It uses the GRAM module to provide adaptive scheduling of jobs in response to changing grid conditions.

C.2 The White Rose Grid

The White Rose Grid is a consortium of three universities in the Yorkshire area. These are the Universities of Sheffield, Leeds and York. The grid currently provides a total of four HPC nodes, one called Iceberg from the University of Sheffield, two called Everest and Snowdon from the University of Leeds[1] and one other node called Pascali from the University of York. The Iceberg node is a cluster of 40 Sun Fire V20Z servers from Sun Microsystems with two 2.4 GHz AMD Opteron processors each plus another 20 Sun Fire V40Z servers, which are similar but have four Opteron processors instead. All the machines are connected through a low latency, high bandwidth, Myrinet network. The Everest node consists of 87 V20Z servers with two dual core 2.2 GHz AMD Opteron processors each plus 7 V40Z servers that use four dual core 2.2 GHz AMD Opteron processors. The Snowdon node is a cluster of 128 Intel servers, using dual 2.2 GHz or 2.4 GHz Intel Xeon processors. Both the Everest and the Snowdon nodes use a Myrinet 2000 network as their backbone. A description of the Pascali node at York is not presented here as access to this node was not available for this work. In total there are 820 processors (including processors in dual core chips) in the White Rose Grid, available to perform image rendering tasks. All these processors have access to significant memory resources, varying between 2 Gb and 32 Gb in size. The size of main memory, however, is not a constraint for the type of rendering application that is described here. Procedural models of terrain have a very small memory footprint since they are essentially procedures in a computer program. There is no need to store large arrays of geometric data such as vertices or triangles.

All the available nodes in the White Rose Grid have a reasonably homogeneous configuration, compared to what is normally expected from a grid, and have the following characteristics in common:

- Scientific Linux as the operating system.
- GNU development tools.
- Bash shell interpreter.
- Single sign on access through SSH with public/private key pairs.
- Single sign on access with the Globus Toolkit.

[1]There is an extra node at Leeds called Maxima but maintenance of this node has recently been discontinued. The node is also not accessible through the Internet. It must be accessed indirectly through Everest or Snowdon.

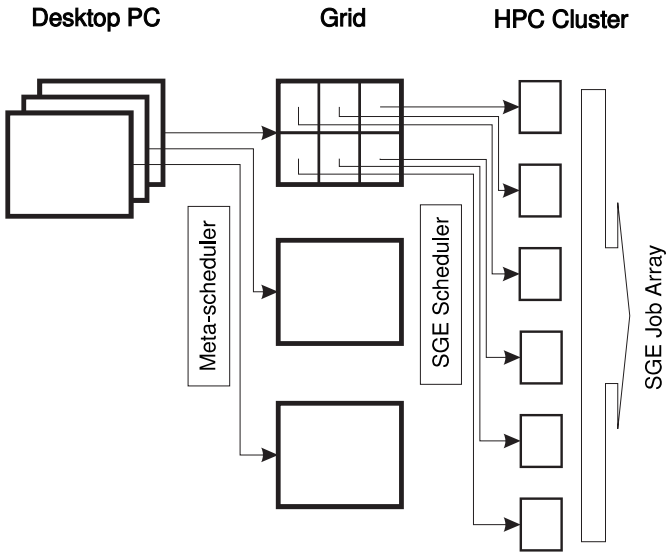


Figure C.1: The two tiered distribution architecture of the grid application. The larger rectangles represent images. The smaller rectangles represent image tiles.

- Job scheduling with the Sun Grid Engine.

All nodes run Linux and use the Sun Grid Engine to schedule jobs. Once a public RSA encryption key is copied to each node, it is possible to invoke any Unix command remotely through the SSH protocol. Because of the relative homogeneity and the ease in accessing grid nodes from a remote application, it did not seem necessary to use the more complex Globus Toolkit. When running remote Unix applications in the White Rose Grid, the `ssh` command can successfully fill the role of grid middleware. The situation may change in the future, however, if this work is later ported to a larger grid such as the National Grid Service.

C.3 A Grid Application for Rendering Synthetic Landscapes

This Appendix focuses on the development of a grid application on the White Rose Grid that implements the distributed rendering of sets of synthetically generated images. These images are then put together into a video sequence that can be played back with the appropriate video software. A meta-scheduler was written to coordinate the scheduling of the rendering jobs on all the available grid nodes. The meta-scheduler consists of a Python script that runs on a Linux desktop PC with Internet access to the grid. The Python scripting language was chosen because it allows the rapid development of new applications and also because it offers a rich set of system calls to interact with the operating system and the network.

As Figure C.1 shows, the grid application enforces a two tiered model of load distribution. In the first tier, images are individually distributed across the grid nodes. Each image is entirely rendered by the node to which it was assigned by the meta-scheduler. A load balancing scheme at the image level is the result of implementing a dynamic scheduling policy in the meta-

scheduler. Grid nodes that are faster receive more images to render. Whenever a grid node becomes idle, the meta-scheduler assigns the next available image to it. In the second tier, load distribution is implemented by splitting an image into smaller rectangular regions, called tiles, and assigning these tiles to different processors of the HPC cluster that is installed at the grid node. The set of rendering jobs for all the tiles of an image forms a *job array* in the Sun Grid Engine (SGE) scheduler at the node. So, load distribution at the image level is handled by the meta-scheduler while load distribution at the tile level is handled by the job schedulers at each individual grid node. The SGE schedulers will attempt to distribute the tile rendering jobs in a fair way between all the processors of a cluster, keeping in mind other users of the same cluster.

C.3.1 Application Deployment

The first step in setting up the grid is to deploy the ray tracing application to the grid nodes. Deployment of an application is often a source of problems in grid computing, especially when the application depends on third-party software libraries. It may happen that different grid nodes have slightly different versions of the same library. In this case, the user must tweak his code in different ways at each node to accommodate the idiosyncracies of the supporting software that resides there. This type of problem does not occur for the ray tracing application since it only requires support from the operating system libraries. Given that Scientific Linux is Posix compliant, the API calls that are required by the ray tracer can be found on all nodes.

The ray tracer is a C++ application that was written on a Windows laptop, inside the Cygwin emulating environment. It was written with the portability to Unix systems other than Cygwin in mind. The GNU set of portability tools (Autoconf, Automake and Libtool) was used to manage the software installation and configuration procedure. After unpacking the source code at each node, the command `./configure` was issued to generate the makefiles, followed by `make`. The source code compiled cleanly on all three nodes. The ray tracer is a command line tool that accepts several command line options. The line below shows an example of what a typical invocation of the ray tracer at one of the nodes might be:

```
gaea -r 800,600 -w 0,60,80,120 t=0.01 <outputfile>
```

This command starts the rendering of an image with a resolution of 800 × 600 pixels but it only renders a tile of this image where the pixels have coordinates (i, j) that are constrained by $0 \leqslant i < 80$ and $60 \leqslant j < 120$. The output is an image file, whose name is the last argument specified for the `gaea` command, with a dimension of $(80-0) \times (120-60) = 80 \times 60$ pixels. The command line variable `t=0.01` indicates to the ray tracer that it should render a snapshot of an animation corresponding to the time instant $t = 0.01s$. The setting up of the terrain functions and the lighting conditions is currently hardwired in the application. This is clearly a limitation as editing the landscape requires changing the source code in the Windows laptop and migrating copies of the source to all the grid nodes again. In the future, a grammar parser should be included to read the landscape settings from a text file, possibly written in XML format, and avoid having to recompile any code.

A set of six Bash scripts must also be installed on the grid nodes to ensure proper communication between the meta-scheduler and the local SGE schedulers. The list of the necessary scripts is as follows:

gaearun This is the script used by the Sun Grid Engine to start the rendering of a new tile at a processor. It invokes the `gaea` application, with the proper command line options.

gaeasub Submits a new job array that resulted from the subdivision of an image into tiles. This is basically a wrapper for the `qsub` SGE command. It passes the name of the `gaearun` script to the scheduler.

gaeadel Removes a job array, including all instances of that array that may be running at the time, from the scheduler. This is basically a wrapper for the `qdel` SGE command.

gaeals Returns a list of the image files that have been completed as part of the execution of a job array. This command can be invoked at any time, returning only the filenames for tiles that have finished rendering by that time.

gaeaget Returns the content of a given image file. The command basically sends the content of the file to its standard output, which is then piped back to the meta-scheduler.

gaearm Removes all image files associated with a given job array from the node's file system.

The scripts are quite portable since they rely only on the existence of a Bash shell interpreter (which all grid nodes have) and a minimal set of Unix command tools like `find`, `tail` and `rm`. Copies of the scripts are migrated to the grid nodes inside the source code package for the ray tracer. The original scripts are maintained at the Windows laptop, along with the Python script for the meta-scheduler, and are part of the ray tracer's development tree.

C.3.2 Grid Resource Location

Information about the grid nodes is read on startup from a local text file, called `grid.conf`. The text file lists the available nodes and further information, relative to each node, that is required by the meta-scheduler. The current contents of this static resource file are as follows:

```
# Node name              Username   Min_tsize  Max_jsize
iceberg.shef.ac.uk       acp04mog     2500       75000
everest.leeds.ac.uk      wrsmog       1600       75000
snowdon.leeds.ac.uk      wrsmog       5000       75000
```

The resource information required by the meta-scheduler, for each grid node, is the fully qualified address, the username of the account, the minimum size (in pixels) of a tile and the maximum size of a job array (correspondingly, the maximum number of tiles) that is allowed by the SGE scheduler at the node. The meta-scheduler uses the last two parameters during its initialisation to compute an optimal tile subdivision of the image for each node. The parameter `Max_jsize` is equal to the `max_aj_tasks` configuration parameter of the SGE schedulers. None of the grid nodes enforces a particular value for this parameter in their configuration files and it defaults to the value 75000. The `Min_tsize` parameter is chosen by the user and is loosely dependent on the maximum number of jobs that a user can run simultaneously on a node. This, in turn, is given by the `maxujobs` configuration parameter of the SGE schedulers. The rationale is that, if a user can only run a small number of jobs concurrently, there

Node	maxujobs
Iceberg	10
Everest	unlimited
Snowdon	4

Table C.1: The value of the `maxujobs` parameter at the three grid nodes.

is no advantage in having many small tiles since most of them will have to be kept waiting in the scheduling queue. If the SGE `maxujobs` parameter is small, it is slightly more efficient to have a few larger tiles as it decreases the overhead of launching many small jobs on the cluster. The `maxujobs` at the three grid nodes is shown in Table C.1. The content of this table motivated the values of the `Min_tsize` parameter that are used in the resource file.

The meta-scheduler performs a subdivision of the image into tiles for each grid node, based on the following list of criteria:

- The size of the tiles must be as small as possible but not smaller than `Min_tsize`.
- The total number of tiles must not exceed `Max_jsize`.
- The difference between the width and the height of the tiles must be as small as possible, i.e. the shape of the tiles must be as close as possible to a square.

The meta-scheduler computes all integer divisors of the image width and all integer divisors of the image height as part of its initialisation procedure. A list of all possible pairs of these two sets of divisors is then formed. Every element in this list is a potential candidate for the width and height of a tile. The list is sorted by increasing order of tile size and searched from the smallest to the largest size. The first element that obeys all criteria is the chosen tile size.

The data contained in the resource file is not likely to change frequently and, therefore, this static resource allocation scheme is adequate for our purposes. Situations where the data in the `grid.conf` file might change would be caused by one of the grid node system administrators reconfiguring the SGE scheduler by changing either its `maxujobs` or `max_aj_tasks` parameter. It could be possible to initialise the `Min_tsize` and `Max_jsize` parameters dynamically by querying the scheduler parameters of each node through a remote invocation of the SGE `qconf` command. This increased coding complexity does not seem warranted at this point, however.

C.3.3 Grid Access and Authentication

After installing an RSA public key at each grid node, it is possible to remotely invoke commands on that node through `ssh`, without having to interactively provide a password authentication every time. For example, to schedule an image rendering on the Iceberg node, a new job array must be submitted to that node's scheduler in the following way:

```
ssh acp04mog@iceberg.shef.ac.uk "~/gaeasub -t 1-192 \
  -v RESX=800,RESY=600,WINX=50,WINY=50,TIME=0.01 <filename>"
```

This specifies that a 800 × 600 resolution image is to be rendered after being split into 192 tiles, where each tile has a dimension of 50 × 50 pixels. The `-t` option specifies the range of tiles to be rendered, from tile 1 to tile 192. The `-v` option specifies a comma separated list of variables that are exported by the SGE scheduler as environment variables into the `gaearun` script. Remote invocation of the other shell scripts through the SSH protocol is performed in a similar manner.

The meta-scheduler launches these `ssh` commands through an `os.popen()` system call that is defined in the Python interpreter. The single argument to the `os.popen()` call is a string that contains a Unix command to be executed (a `ssh` command in this case). A Unix pipe is opened between the Python interpreter and the command. In particular, the standard output of the command is redirected back through the pipe and can be read by the Python script. For example, the remote invocation of the `gaeasub` script shown previously can be written in Python in the following way:

```
pipe = os.popen("<string for ssh gaeasub script>")
output = pipe.read()
pipe.close()
jobID = parse(output)
```

The Python variable `output` stores the standard output of the `gaeasub` command, which among other things indicates the job ID of the newly created SGE job array. If the `ssh` command executes without error, the job ID can then be parsed from the `output` variable. This ID may be required later, should the meta-scheduler wish to delete the job array.

C.3.4 Implementing a Polling Strategy

The initial idea at the start of this project was to have a client-server architecture between the meta-scheduler and the grid nodes. The meta-scheduler would keep a socket open, listening for incoming packets. Each grid node processor, once it finished rendering a tile, would open a connection to this socket and send over the computed tile data. The meta-scheduler would receive the rendering results in real time from all the processors in the grid and this would allow it to schedule new rendering jobs on the fly. There were, however, difficulties in implementing this architecture because of the firewalls at Sheffield and Leeds that do not allow machines from inside their HPC clusters to connect to arbitrary addresses on the outside. There was still the possibility of keeping a daemon proxy running inside each HPC cluster. The cluster processors would send their packets instead to this proxy and it would relay the packets back to the meta-scheduler through a SSH tunnel. The proxy would have to be left running outside the control of the Sun Grid Engine (otherwise it would be killed after a maximum allowable running time had elapsed) but this would represent a breach of policy of the HPC clusters, where every job must be supervised by the SGE in order to maintain fairness between users.

Rather than using a client-server architecture, the meta-scheduler implements a polling strategy instead and checks the state of the tile renderings on all grid nodes at regular intervals. The polling interval has a default duration of five minutes but this can be changed through a command line option when the meta-scheduler is launched. The rendering state of each grid node is remotely queried with the `gaeals` script, which returns a list of all the filenames for the tiles

that have been completed so far. This allows the meta-scheduler to detect new tiles that have been completed since the last poll to the same grid node was performed. When all the tiles of an image that is being rendered on a grid node have been completed, the meta-scheduler sends instructions through `gaeasub` for a new image to be scheduled on that node. This polling mechanism works well in situations where tiles take, on average, significantly longer to render than the duration of the polling interval since the rendering state of a grid node will undergo only small changes between polls.

C.3.5 Implementing a Schedule-Ahead Policy

The polling mechanism introduces only one source of inefficiency, which is related to the scheduling of new images. Consider the situation where a grid node finishes rendering an image shortly after a poll from the meta-scheduler was completed. For the remainder of the following five minutes the grid node is going to be idle with respect to the grid rendering application. The meta-scheduler will only detect the completion of the image and schedule a new image after the five minutes have elapsed. The problem is made worse by the fact that, during those five minutes of idle time, jobs from other users will occupy the processors that have since become available. When the new image is scheduled, it will have to wait for the processors to become available again. If the new image had already been present in the scheduling queue by the time the previous image finished it might have been able to recapture some of the same processors. This is because the SGE uses a dynamic priority strategy and the job array for the new image might have a higher priority that some of the other user jobs in the system. The rendering efficiency of the grid nodes would be improved in this way without violating the principles of fairness to other users that are always enforced by the SGE.

The inefficiency that stems from scheduling new images under control of the meta-scheduler's polling mechanism is alleviated by implementing a schedule-ahead policy. This basically means that the meta-scheduler speculatively assigns a new image in advance to a grid node while a previous image is still rendering at the same node. At any time there will always be two job arrays in a scheduling queue. One job array will be undergoing rendering (with some of the tiles in the array being rendered while the remaining tiles wait for available processors) and the other job array will stay in wait until the previous array finishes. The following partial output of the `qstat` command shows a typical scheduling state in the Snowdon node, in what concerns the grid rendering application:

```
job-ID  prior   name       user   state  ja-task-ID
-----------------------------------------------------
177620  0.51000 gaearun    wrsmog r      3
177620  0.51000 gaearun    wrsmog r      4
177620  0.51000 gaearun    wrsmog r      2
177620  0.51000 gaearun    wrsmog r      1
177620  0.00000 gaearun    wrsmog qw     5-48:1
177621  0.00000 gaearun    wrsmog qw     1-48:1
```

The job array with ID 177620 has image tiles 1 to 4 being rendered, all with a priority of 0.51, while the remaining tiles 5 to 48 remain waiting. There is another job array with ID 177621

that will start rendering once the last tiles from the previous job array begin to complete. The `maxujobs` parameter for Snowdon is 4 (recall Table C.1) and, therefore, only four tiles can be rendered concurrently on this node. There will be a brief transitional period when the four processor slots available to user `wrsmog` will be shared between the last tiles of job 177620 and the first tiles of job 177621. Once all the remaining tiles from job 17760 finish, a new job array will be placed on the queue by the meta-scheduler, sometime within a five minute period.

The schedule-ahead mechanism begins to break down for fast enough grid nodes that can render all the image tiles in an amount of time comparable to the meta-scheduler's polling interval. This is more likely to happen in the Everest node, since it does not impose any limit on the number of jobs running concurrently[2]. In this type of situation, the meta-scheduler may not have enough time to place another image on the scheduling queue before the previous image completes rendering. To solve this problem one must either decrease the polling interval or increase the number of simultaneous job arrays that must be kept in the queue.

C.3.6 Retrieving Computation Results

The outcome of a tile that has finished rendering on a grid node is an image data file, residing in the node's file system. The data file is written in the PPM format, a minimalistic image file format that basically contains a header, indicating the tile dimensions, followed by the raw pixel data [Murray and vanRyper, 1994]. Despite its simplicity, the PPM image format is accepted by virtually all Unix-based image viewers. The filename of the PPM tile data obeys the following convention, where the parameters between angle brackets signify fields with variable information:

```
<filename>_<imgnum>_<tilenum>.ppm
```

The `imgnum` field indicates the image number in the sequence of images that constitutes the computer animation. The `tilenum` field indicates the tile number inside the image `imgnum`. The `gaeals` script retrieves lists of filenames that obey this naming convention. The meta-scheduler then invokes the `gaeaget` script for individual files. The content of a file is sent from the standard output of `gaeaget`, through the `ssh` command and through the Unix pipe that was set up by the `os.popen()` call until it arrives at the meta-scheduler, where it is temporarily kept in a string variable. The meta-scheduler then stores the results of the tile rendering in the local image file:

```
<filename>_<imgnum>.ppm
```

The meta-scheduler opens the image file and uses the value of the `tilenum` field to move the file pointer to the correct position inside the image with a `file.seek()` Python call. The pixel data from the tile is then transferred to its correct place in the image[3].

[2]Even in grid nodes where the `maxujobs` parameter imposes no restriction on the number of simultaneous jobs from the same user, the load imposed by other users always constrains this number in practice. In the very best of situations, the number of active tile renderings will always be constrained by the `max_aj_instances` parameter, the maximum number of tasks from the same job array that can run simultaneously.

[3]The actual procedure of pasting a tile pixel data onto an image is a bit more complex. Because PPM images store the pixel array in a row major format, a `file.seek()` call must be made for every row of pixels in the tile.

C.4 Results

Figure C.2 shows four frames from a computer generated animation of a camera flyby over a procedural heightfield terrain. The camera is travelling forward with a constant speed of 60 km/h and at a constant altitude of 25 metres[4]. The animation has 500 frames, which, at a rate of 25 frames per second, corresponds to 20 seconds of playback time. At the speed the camera is travelling, the animation would need to have a duration of almost a month of continuous playback time for a complete circumnavigation of the planet to be achieved. In this impractical scenario, the first and the last frames of the animation would be equal and the animation could be looped indefinitely. The animation can be downloaded as a Quicktime movie file from the address[5]:

```
http://www.dcs.shef.ac.uk/~mag/flyby.mov
```

Anti-aliasing and motion blur for the animation of Figure C.2 were obtained with a cubic B-spline filter and with the shooting of 10^2 rays per pixel (refer to Chapter 6). The rendering was started on the 13th of June 2006 at 19:50 hours and it was complete by the 20th of June at 11:18 hours, which corresponds to 6 days and 15 hours. The rather large number of rays per pixel that was employed is largely responsible for the significant computational time that it took to render the 500 frames of animation. This large number of rays was considered necessary to obtain correct patterns of motion blur over the mountain peaks, given that the camera grazes those peaks at a very low altitude. A smaller number of rays per pixel would have caused temporal aliasing during some sections of the animation.

A problem of the proposed grid rendering tool that became apparent while the animation of Figure C.2 was being computed is that of *grid lockouts*. When scheduling an image to a node one must specify a parameter called `h_rt` that indicates to the scheduler how long each tile rendering is expected to run. The SGE will kill a tile rendering job once its running time exceeds `h_rt`. The meta-scheduler will then be left waiting indefinitely for the results from the tile that was killed and it will not be able to schedule a new image on that node. This situation can go on undetected for several hours. Once it is detected, it becomes necessary to re-initialise the grid application and, as a consequence, images that were being rendered at the time of re-initialisation have to be rendered again. Grid lockouts also contributed negatively to the total rendering time of the animation, with nodes that became locked out during the night having to wait until the following morning to be re-initialised.

C.5 Summary

The distributed rendering model presented in this report has the potential to significantly reduce the rendering time of complex computer animations. The computation of the animation shown in Figure C.2, expensive as it was, would have been totally unfeasible on a conventional desktop machine. Visualising terrains with overhangs and flow mapped details is also possible – each processor responsible for a given image tile keeps its local cache of connectivity information

[4] Although the mountains look imposing in Figure C.2, they are actually only a few metres high.

[5] A free Quicktime player, available from `www.apple.com`, must be installed before playing the animation.

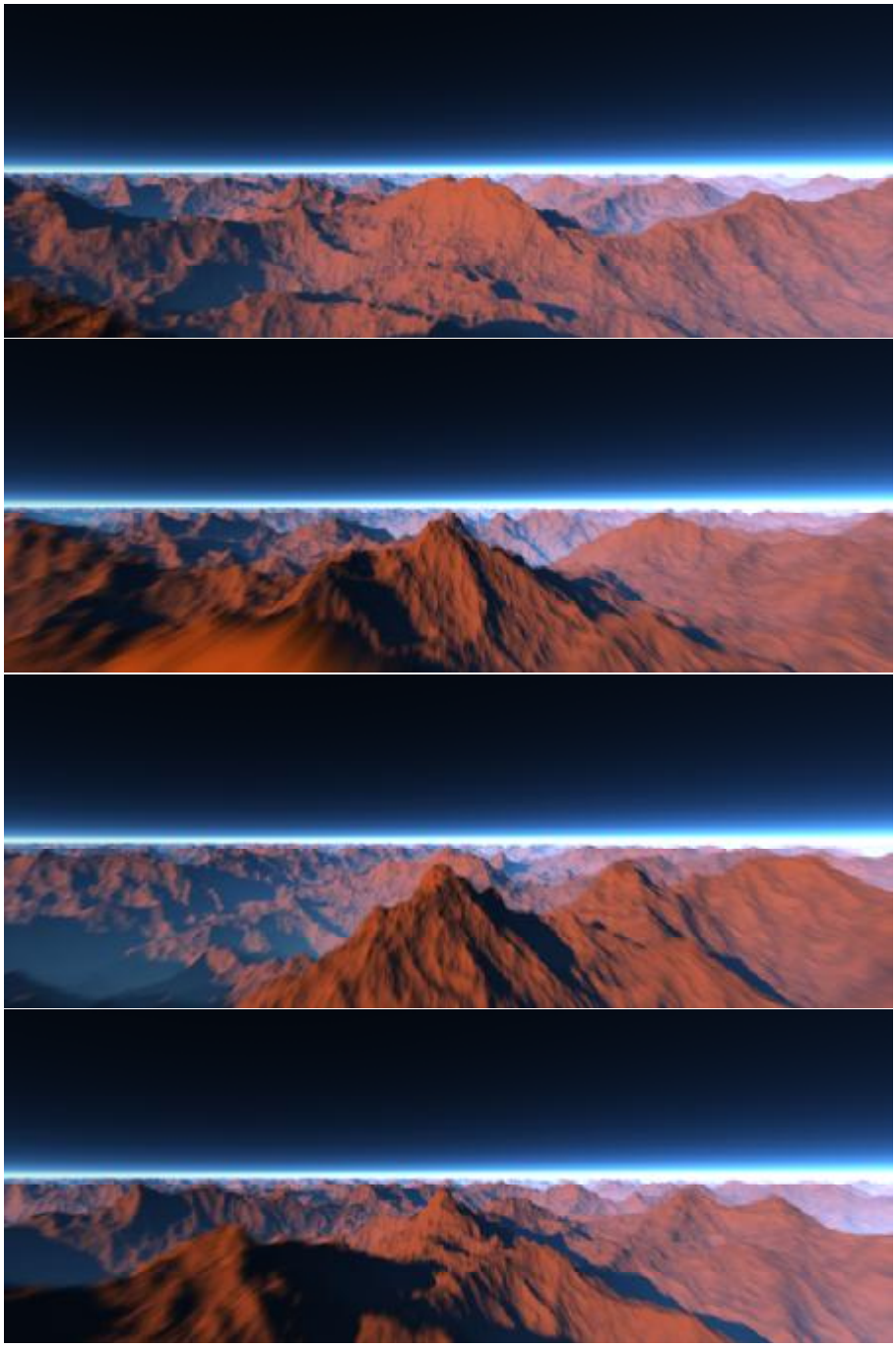


Figure C.2: A sequence of frames showing a camera flyby over a procedural planetary landscape at constant altitude and speed. The motion blur effect is visible in the lower part of the frames.

for the surface that is visible through that tile (refer to Chapter 8). This task remains as a future possibility but it is likely to require a compromise on the number of anti-aliased rays per pixel to compensate for the increased complexity of rendering terrains with overhangs.

The design of the current grid rendering tool can be considered as a pattern for similar computing problems. In the most general terms, the current design solves computational problems that can be split into smaller and independent tasks. The pattern favours a two-tiered distribution model, where tasks can themselves be split into smaller and still independent sub-tasks. Tasks are distributed across grid nodes while sub-tasks are distributed among the processors of a node. This two-tiered model, however, is not a requirement. If tasks cannot be further split they can still be arbitrarily grouped. The task groups are then distributed to the nodes. One example from Computer Graphics where this design model can be applied is in the implementation of a distributed version of the Reyes image rendering architecture [Cook et al., 1987]. A Reyes image renderer works by splitting the geometry (which can be made of polygon meshes, NURBS patches or subdivision surfaces) into progressively smaller fragments. When a fragment becomes much smaller than the size of a pixel it is called a *micropolygon*. It is then passed to the shading pipeline for rendering. Reyes has traditionally been used by major animation studios such as Pixar for the rendering of their computer animation feature films although a recent shift towards ray traced animations is beginning to occur.

The Reyes algorithm can be distributed by partitioning the image space into tiles, similarly to the ray tracing algorithm, and having each processor handle its own image tile[6]. Each processor will only split geometry data whose bounding box overlaps with the processor's tile. There are two additional steps in the implementation of a distributed Reyes renderer that were not necessary for the ray tracing of procedural surfaces. During the first step, all the geometry data must be transmitted to the grid nodes, possibly using a `scp` command. The second step is a pre-computation that must be performed before the rendering work at a grid node is distributed to its processors. This pre-computation step computes, among other things, hierarchies of bounding boxes for the geometry. These bounding boxes are required in order for each processor to know which geometry elements it should be concerned with. The two steps just described can be added with some extra effort to the distributed model that has been developed for ray tracing procedural landscapes.

An important aspect of the current grid rendering tool that needs improvement is the proper handling of grid lockouts by the meta-scheduler. Currently, the `h_rt` parameter must be estimated by the user before the meta-scheduler is launched. It is hardwired in the header of the `gaearun` shell script and is therefore the same for all image tiles. It is very difficult to predict what the value of `h_rt` should be because it depends on the complexity of the surface that is visible through a tile. Tiles that only see background sky will render much faster than tiles that focus on terrain features. Assigning an over-conservative estimate of the tile rendering time to `h_rt` would solve the problem since it would guarantee that even the slowest of the tile renderings would not be killed by the scheduler. This, however, is not a practical solution because it will make it more difficult for the tile jobs to become active. If the SGE scheduler is given a job with a long predicted run time, it will attempt to schedule faster jobs first. A better way to handle this situation is for the meta-scheduler to monitor the running jobs on the grid nodes

[6]Pixar has a render farm made of 1024 Intel servers with 2.8 GHz Xeon processors and is certain to have a distributed version of the Reyes rendering tool. Details of how this distribution is performed are not known.

and to detect jobs that were killed without producing a full tile rendering. The meta-scheduler would then examine the incomplete tile pixel data and re-issue the same tile jobs at the point where they stopped rendering.

BIBLIOGRAPHY

Agin, G. J. and Binford, T. O. (1976). Representation and description of curved objects. *IEEE Trans. on Computers*, 25(4):439–449.

Amanatides, J. (1984). Ray tracing with cones. In Christiansen, H., editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 129–135. ACM Press.

Anderson, J. and Anderson, B. (1993). The myth of persistence of vision revisited. *Journal of Film and Video*, 45(1):3–12.

Anjyo, K.-I. (1988). A simple spectral approach to stochastic modeling for natural objects. In Duce, D. A. and Jancene, P., editors, *Eurographics '88*, pages 285–296. North-Holland.

Anjyo, K.-I. (1991). Semi-globalization of stochastic spectral synthesis. *The Visual Computer*, 7(1):1–12.

Asirvatham, A. and Hoppe, H. (2005). Terrain rendering using GPU-based geometry clipmaps. In Pharr, M., editor, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 2, pages 27–46. Addison-Wesley.

Ausloos, M. and Berman, D. (1985). A multivariate Weierstrass-Mandelbrot function. *Proc. R. Soc. London A*, 400(1819):331–350.

Bærentzen, J. A. and Jørgen, N. (2002). Volume sculpting using the level-set method. In *Proceedings of the International Conference on Shape Modeling and Applications (SMI-02)*, pages 175–182. IEEE Computer Society.

Bajaj, C. L., Chen, J., and Xu, G. (1995a). Free-form modeling with C^2 quintic A-patches. In *Proc. 4th SIAM Conference on Geometric Design.*

Bajaj, C. L., Chen, J., and Xu, G. (1995b). Modeling with cubic A-patches. *ACM Transactions on Graphics*, 14(2):103–133.

Balsys, R. J., Suffern, K. G., and Jones, H. (2008). Point-based rendering of non-manifold surfaces. *Computer Graphics Forum*, 27(1):63–72.

Barr, A. H. (1981). Superquadrics and angle-preserving transformations. *IEEE Computer Graphics & Applications*, 1(1):11–23.

Barr, A. H. (1984). Global and local deformations of solid primitives. In Christiansen, H., editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 21–30. ACM Press.

Barr, A. H. (1986). Ray tracing deformed surfaces. In Evans, D. C. and Athay, R. J., editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, pages 287–296. ACM Press.

Barthe, L., Wyvill, B., and de Groot, E. (2004). Controllable binary CSG operators for soft objects. *International Journal of Shape Modeling*, 10(2):135–154.

Batchelor, G. K. (2000). *An Introduction to Fluid Dynamics*. Cambridge Mathematical Library Series. Cambridge University Press.

Bergman, L., Fuchs, H., Grant, E., and Spach, S. (1986). Image rendering by adaptive refinement. In Evans, D. C. and Athay, R. J., editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 29–37. ACM Press.

Berry, M. and Lewis, Z. (1980). On the Weierstrass-Mandelbrot fractal function. *Proc. R. Soc. London A*, 370(1743):459–484.

Blinn, J. F. (1978). Simulation of wrinkled surfaces. In Chasen, S. and Phillips, R., editors, *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 286–292. ACM Press.

Blinn, J. F. (1982a). A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256.

Blinn, J. F. (1982b). Light reflection functions for simulation of clouds and dusty surfaces. In Bergeron, R. D., editor, *Computer Graphics (SIGGRAPH '82 Proceedings)*, volume 16, pages 21–29. ACM Press.

Blinn, J. F. (1989a). Jim Blinn's Corner: Dirty pixels. *IEEE Computer Graphics and Applications*, 9(4):100–105.

Blinn, J. F. (1989b). Jim Blinn's Corner: What we need around here is more aliasing. *IEEE Computer Graphics and Applications*, 9(1):75–79.

Bloomenthal, J. (1997). Bulge elimination in convolution surfaces. *Computer Graphics Forum*, 16(1):31–41.

Bloomenthal, J., Bajaj, C., Blinn, J., Cani-Gascuel, M.-P., Rockwood, A., Wyvill, B., and Wyvill, G. (1997). *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

Bloomenthal, J. and Shoemake, K. (1991). Convolution surfaces. In Sederberg, T. W., editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 251–257. ACM Press.

Bloomenthal, J. and Wyvill, B. (1990). Interactive techniques for implicit modeling. In Zyda, M. J., editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 109–116. ACM Press.

Bracewell, R. (1999). *The Fourier Transform and Its Applications*. McGraw-Hill, 3rd edition.

Breen, D. E. and Whitaker, R. T. (2001). A level-set approach for the metamorphosis of solid models. *IEEE Trans. on Visualization and Computer Graphics*, 7(2):173–192.

Brigham, E. O. (1988). *The Fast Fourier Transform and its applications*. Prentice Hall.

Brosz, J., Samavati, F. F., and Sousa, M. C. (2006). Terrain synthesis by-example. In Braz, J., Jorge, J., Dias, M., and Marcos, A., editors, *Proceedings of the First International Conference on Computer Graphics Theory and Applications (GRAPP 2006)*, pages 122–133. INSTICC - Institute for Systems and Technologies of Information, Control and Communication.

Bruce, J. W. and Giblin, P. J. (1992). *Curves and Singularities*. Cambridge University Press, 2nd edition.

Bruneton, E. and Neyret, F. (2008). Precomputed atmospheric scattering. *Computer Graphics Forum*, 27(4):1079–1086.

Carlson, A. B., Crilly, P. B., and Rutledge, J. (2001). *Communication Systems: An Introduction to Signals and Noise in Electrical Communication*. McGraw-Hill, 4th edition.

Carpenter, L. (1980). Computer rendering of fractal curves and surfaces. In Thomas, J. J., Ellis, R. A., and Kriloff, H. Z., editors, *Computer Graphics (SIGGRAPH '80 Proceedings)*, page 109. ACM Press.

Carpenter, L. (1984). The A-buffer, an antialiased hidden surface method. In Christiansen, H., editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 103–108. ACM Press.

Catmull, E. (1974). *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Dept. of Computer Science, The University of Utah.

Catmull, E. (1978). A hidden-surface algorithm with anti-aliasing. In Chasen, S. and Phillips, R., editors, *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 6–11. ACM Press.

Catmull, E. and Clark, J. (1978). Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350–355.

Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., and McDonald, J. (2000). *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc.

Chang, Y.-C., Song, G.-S., and Hsu, S.-K. (1998). Automatic extraction of ridge and valley axes using the profile recognition and polygon-breaking algorithm. *Computers & Geosciences*, 24(1):83–93.

Chiba, N., Muraoka, K., and Fujita, K. (1998). An erosion model based on velocity fields for the visual simulation of mountain scenery. *The Journal of Visualization and Computer Animation*, 9(4):185–194.

Chiba, N., Muraoka, K., Yaegashi, K., and Miura, M. (1991). Terrain simulation based on the recursive refinement of ridge-lines. In *Proceedings of CAD/Graphics*, pages 19–24.

Chui, C. K. (1992). *An Introduction to Wavelets*, volume 1 of *Wavelet Analysis and its Applications*, chapter 4, pages 81–117. Academic Press.

Clasen, M. and Hege, H.-C. (2006). Terrain rendering using spherical clipmaps. In Santos, B. S., Ertl, T., and Joy, K. I., editors, *EUROVIS - Eurographics /IEEE VGTC Symposium on Visualization*, pages 91–98. Eurographics Association.

Cohen, L. (1995). *Time-Frequency Analysis*. Prentice Hall Signal Processing Series. Prentice Hall.

Cohen, M. F., Chen, S. E., Wallace, J. R., and Greenberg, D. P. (1988). A progressive refinement approach to fast radiosity image generation. In Dill, J., editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 75–84. ACM Press.

Comba, J. L. D. and Stolfi, J. (1993). Affine arithmetic and its applications to computer graphics. In *Proc. VI Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI '93)*, pages 9–18.

Cook, R. L. (1984). Shade trees. In Christiansen, H., editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 223–231. ACM Press.

Cook, R. L. (1986). Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72.

Cook, R. L., Carpenter, L., and Catmull, E. (1987). The Reyes image rendering architecture. In Stone, M. C., editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 95–102. ACM Press.

Cook, R. L. and DeRose, T. (2005). Wavelet noise. 24(3):803–811.

Cook, R. L., Porter, T., and Carpenter, L. (1984). Distributed ray tracing. In Christiansen, H., editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 137–45. ACM Press.

Corliss, G. F. (1977). Which root does the bisection algorithm find? *SIAM Reviews*, 19(2):325–327.

Crow, F. C. (1981). A comparison of antialiasing techniques. *IEEE Computer Graphics and Applications*, 1(1):40–48.

Danovaro, E., de Floriani, L., Vitali, M., and Magillo, P. (2007). Multi-scale dual morse complexes for representing terrain morphology. In Samet, H., Shahabi, C., and Schneider, M., editors, *International Symposium on Advances in Geographic Information Systems (Proc. ACM GIS '07)*, page 29. ACM Press.

de Cusatis Jr., A., de Figueiredo, L. H., and Gattas, M. (1999). Interval methods for raycasting implicit surfaces with affine arithmetic. In *Proc. XII Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI '99)*, pages 65–71.

de Figueiredo, L. H. and Stolfi, J. (1996). Adaptive enumeration of implicit surfaces with affine arithmetic. *Computer Graphics Forum*, 15(5):287–296.

de Figueiredo, L. H. and Stolfi, J. (2004). Affine arithmetic: Concepts and applications. *Numerical Algorithms*, 37(1–4):147–158.

de Toledo, R., Lévy, B., and Paul, J.-C. (2007). Iterative methods for visualization of implicit surfaces on GPU. In *Advances in Visual Computing (Proceedings of 3rd International Symposium ISVC 2007)*, volume 4841 of *Lecture Notes in Computer Science*, pages 598–609. springer.

Dixon, A. R., Kirby, G. H., and Wills, D. P. (1999). Artificial planets with fractal feature specification. *The Visual Computer*, 15(3):147–158.

Doob, J. L. (1953). *Stochastic Processes*. Wiley Classics Library. John Wiley and Sons.

Duff, T. (1992). Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. In Catmull, E. E., editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 131–138. ACM Press.

Dutré, P., Bala, K., and Bekaert, P. (2006). *Advanced Global Illumination*. AK Peters Ltd, Wellesley, MA, 2nd edition.

Eastlick, M. (2006). *Discrete Differential Geometry and an Application in Multiresolution Analysis*. PhD thesis, Department of Computer Science, The University of Sheffield.

Ebert, D. S., Musgrave, F. K., Peachey, D. R., Perlin, K., and Worley, S. P. (2003). *Texturing & Modeling: A Procedural Approach*. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann Publishers Inc., San Francisco, CA, 3rd edition.

Efros, A. A. and Freeman, W. T. (2001). Image quilting for texture synthesis and transfer. In Pocock, L., editor, *Computer Graphics (SIGGRAPH '01 Proceedings)*, pages 341–346. ACM Press.

Evertsz, C. J. G. and Mandelbrot, B. B. (1992). Multifractal measures. In Peitgen, H.-O., Jürgens, H., and Saupe, D., editors, *Chaos and Fractals: New Frontiers of Science*, appendix B, pages 921–953. Springer-Verlag.

Falconer, K. (1990). *Fractal Geometry: Mathematical Foundations and Applications*. John Wiley & Sons.

Farin, G. (2001). *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann Publishers Inc., San Francisco, CA, 5th edition.

Farrugia, J. P. and Peroche, B. (2004). A progressive rendering algorithm using an adaptive perceptually based image metric. *Computer Graphics Forum*, 23(3):605–614.

Ferley, E., Cani, M.-P., and Gascuel, J.-D. (2001). Resolution adaptive volume sculpting. *Graphics Models*, 63(6):459–478.

Fernando, R. and Kilgard, M. J. (2003). *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional.

Flandrin, P. (1987). On the spectrum of fractional brownian motions. *IEEE Transactions on Information Systems*, 35(1):197–199.

Flóres, J., Sbert, M., Ángel Sainz, M., and Vehí, J. (2006). Improving the interval ray tracing of implicit surfaces. In Nishita, T., Peng, Q., and Seidel, H.-P., editors, *Advances in Computer Graphics, 24th Computer Graphics Internacional Conference (Proc. CGI 2006)*, volume 4035 of *Lecture Notes in Computer Science*, pages 655–664. Springer.

Foley, J., van Dam, A., Feiner, S., and Hughes, J. (1990). *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Company, 2nd edition.

Foster, I. T. and Kesselman, C. (1997). Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128.

Foster, I. T., Kesselman, C., Nick, J. M., and Tuecke, S. (2002). Grid services for distributed system integration. *IEEE Computer*, 35(6):37–46.

Foster, N. and Fedkiw, R. (2001). Practical animation of liquids. In Pocock, L., editor, *Computer Graphics (SIGGRAPH '01 Proceedings)*, pages 23–30. ACM Press.

Fournier, A. (1980). *Stochastic Modelling in Computer Graphics*. PhD thesis, University of Texas at Dallas.

Fournier, A. (1989). The modelling of natural phenomena. In *Proceedings of Graphics Interface '89*, pages 191–202. Canadian Information Processing Society.

Fournier, A. and Fussell, D. (1980). Stochastic modeling in computer graphics. In Thomas, J. J., Ellis, R. A., and Kriloff, H. Z., editors, *Computer Graphics (SIGGRAPH '80 Proceedings)*, page 108. ACM Press.

Fournier, A., Fussell, D., and Carpenter, L. (1982). Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384.

Friedman, J. H., Bentley, J. L., and Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226.

Galin, E. and Akkouche, S. (1996a). Blob metamorphosis based on minkowsky sums. *Computer Graphics Forum (Eurographics 1996 Proceedings)*, 15(3):143–154.

Galin, E. and Akkouche, S. (1996b). Shape constrained blob metamorphosis. In *Proc. Implicit Surfaces '96*, pages 9–23.

Galin, E., Leclercq, A., and Akkouche, S. (1999). Blob-tree metamorphosis. In *Proc. Implicit Surfaces '99*.

Galyean, T. A. and Hughes, J. F. (1991). Sculpting: An interactive volumetric modeling technique. In Sederberg, T. W., editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 267–274. ACM Press.

Gamito, M. N. and Maddock, S. C. (2006a). Anti-aliasing with stratified B-spline filters of arbitrary degree. *Computer Graphics Forum*, 25(2):163–172.

Gamito, M. N. and Maddock, S. C. (2006b). A progressive refinement approach for the visualisation of implicit surfaces. In Braz, J., Jorge, J. A., Dias, M. S., and Marcos, A., editors, *International Conference on Computer Graphics Theory and Applications (GRAPP 2006 Proceedings)*, pages 26–33. INSTICC - Institute for Systems and Technologies of Information, Control and Communication.

Gamito, M. N. and Maddock, S. C. (2007a). Grid computing techniques for synthetic procedural planets. Memorandum CS – 07 – 01, Dept. of Comp. Science, The University of Sheffield.

Gamito, M. N. and Maddock, S. C. (2007b). *A Progressive Refinement Approach for the Visualisation of Implicit Surfaces*, volume 4 of *Communications in Computer and Information Science*, chapter 6, pages 93–108. Springer. Revised selected paper from the GRAPP 2006 Conference.

Gamito, M. N. and Maddock, S. C. (2007c). Progressive refinement rendering of implicit surfaces. *Computers & Graphics*, 31(5):698–709.

Gamito, M. N. and Maddock, S. C. (2007d). Ray casting implicit fractal surfaces with reduced affine arithmetic. *The Visual Computer*, 23(3):155–165.

Gamito, M. N. and Maddock, S. C. (2007e). Topological correction of hypertextured implicit surfaces for ray casting. In Galin, E., Pauly, M., and Wyvill, B., editors, *IEEE International Conference on Shape Modelling and Applications (Proc. SMI '07)*, pages 103–112. IEEE Press.

Gamito, M. N. and Maddock, S. C. (2008a). Localised topology correction for hypertextured terrains. In Lim, I. and Tang, W., editors, *Theory and Practice of Computer Graphics 2008*, Eurographics UK Chapter Proceedings, pages 91–98. The Eurographics Association.

Gamito, M. N. and Maddock, S. C. (2008b). Topological correction of hypertextured implicit surfaces for ray casting. *The Visual Computer*, 24(6):397–409. Revised selected paper from the SMI '07 Conference.

Gamito, M. N. and Musgrave, F. K. (2001). Procedural terrains with overhangs. In *Proceedings of the 10th Eurographics Portuguese Chapter Conference*, pages 33–42. Grupo Português de Computação Gráfica.

Genetti, J. D., Gordon, D., and Williams, G. (1998). Adaptive supersampling in object space using pyramidal rays. *Computer Graphics Forum*, 16(1):29–54.

Gentzsch, W. (2001). Sun Grid Engine: Towards creating a compute power grid. In *Cluster Computing and the Grid (CCGRID '01 Proceedings)*, pages 35–36. IEEE Computer Society.

Glassner, A. S., editor (1989). *An Introduction to Ray Tracing*. Academic Press, Boston, MA.

Glassner, A. S. (1995). *Principles of Digital Image Synthesis*. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann Publishers Inc., San Francisco, CA.

Govindaraju, N. K., Lloyd, B., Dotsenko, Y., Smith, B., and Manferdelli, J. (2008). High performance discrete fourier transforms on graphics processors. In *ACM/IEEE Supercomputing 2008 (SC08 Proceedings)*. ACM Press. to appear.

Grimm, C. M. (1999). Implicit generalized cylinders using profile curves. In Schlick, C., editor, *Proc. Implicit Surface '99*, pages 33–41. Eurographics Association.

Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI: Portable Parallel Programming with the Message–Passing Interface*. The MIT Press, 2nd edition.

Gross, M. and Pfister, H., editors (2007). *Point-Based Graphics*. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann Publishers Inc., San Francisco, CA.

Gyulassy, A., Natarajan, V., Pascucci, V., Bremer, P.-T., and Hamann, B. (2006). A topological approach to simplification of three-dimensional scalar functions. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):474–484.

Hanrahan, P. (1983). Ray tracing algebraic surfaces. In Tanner, P. P., editor, *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pages 83–90. ACM Press.

Hart, J. (1993). Ray tracing implicit surfaces. In *Modeling, Visualizing and Animating Implicit Surfaces*, pages 13.1–13.15. SIGGRAPH '93 Course Notes 25.

Hart, J. C. (1996). Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(9):527–545.

Hart, J. C. (1998). Morse theory for implicit surface modeling. In Hege, H.-C. and Polthier, K., editors, *Mathematical Visualization*, pages 257–268. Springer Verlag, Heidelberg.

Hart, J. C. (1999). Using the CW-complex to represent the topological structure of implicit surfaces and solids. In *Proc. Workshop on Implicit Surfaces*, pages 107–112. Eurographics/SIGGRAPH.

Hart, J. C., Durr, A., and Arsch, D. (1998). Critical points of polynomial metaballs. In *Proc. Workshop on Implicit Surfaces*, pages 69–76. Eurographics/SIGGRAPH.

Hart, J. C., Jarosz, W., and Fleury, T. (2002). Using particles to sample and control more complex implicit surfaces. In *Proceedings of the International Conference on Shape Modeling and Applications (SMI-02)*, pages 129–136. IEEE Computer Society.

Haruyama, S. and Barsky, B. A. (1984). Using stochastic modeling for texture generation. *IEEE Computer Graphics and Applications*, 4(3):7–19.

He, T., Wang, S. W., and Kaufman, A. E. (1994). Wavelet-based volume morphing. In Bergeron, R. D. and Kaufman, A. E., editors, *Proc. IEEE Visualization '94*, pages 85–92, Los Alamitos, CA. IEEE Computer Society Press.

Heidrich, W. and Seidel, H.-P. (1998). Ray-tracing procedural displacement shaders. In Davis, W., Booth, K., and Fournier, A., editors, *Proceedings of the 24th Conference on Graphics Interface (GI-98)*, pages 8–16, San Francisco. Morgan Kaufmann Publishers.

Heidrich, W., Slusallek, P., and Seidel, H. (1998). Sampling procedural shaders using affine arithmetic. *ACM Transactions on Graphics*, 17(3):158–176.

Helman, J. L. and Hesselink, L. (1991). Visualizing vector field topology in fluid flows. *IEEE Computer Graphics and Applications*, 11(3):36–46.

Hindmarsh, A. C. (1983). ODEPACK: A systematized collection of ODE solvers. In Stepleman, R. S., editor, *Scientific Computing*, pages 55–64. North-Holland, Amsterdam. Package available at www.netlib.org.

Hoffmann, C. and Hopcroft, J. (1987). The potential method for blending surfaces and corners. In Farin, G., editor, *Geometric Modeling: Algorithms and New Trends*, pages 347–365. SIAM, Philadelphia.

Hollasch, S. (1992). Progressive image refinement via gridded sampling. In Kirk, D., editor, *Graphics Gems III*, pages 358–361. Academic Press, San Diego.

Hudo, E., Montero, R. S., and Llorente, I. M. (2005). The GridWay framework for adaptive scheduling and execution on grids. *Scalable Computing: Practice and Experience*, 6(3):1–8.

Hughes, J. F. (1992). Scheduled fourier volume morphing. In Catmull, E. E., editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 43–46. ACM Press.

Ito, T., Fujimoto, T., Muraoka, K., and Chiba, N. (2003). Modeling rocky scenery taking into account joints. In *Computer Graphics International (Proc. CGI-03)*, pages 244–247. IEEE.

Jain, A. K. (1989). *Fundamentals of Digital Image Processing*. Prentice Hall, Englewood Cliffs, NJ.

Jin, X. and Tai, C.-L. (2002). Convolution surfaces for arcs and quadratic curves with a varying kernel. *The Visual Computer*, 18(8):530–546.

Kajiya, J. T. (1983). New techniques for ray tracing procedurally defined objects. In Tanner, P. P., editor, *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pages 91–102. ACM Press.

Kalra, D. and Barr, A. H. (1989). Guaranteed ray intersections with implicit surfaces. In Lane, J., editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 297–306. ACM Press.

Kanamori, Y., Szego, Z., and Nishita, T. (2008). GPU-based fast ray casting for a large number of metaballs. *Computer Graphics Forum (Eurographics 2008 Proceedings)*, 27(2):351–360.

Kelley, A. D., Malin, M. C., and Nielson, G. M. (1988). Terrain simulation using a model of stream erosion. In Dill, J., editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, pages 263–268. ACM Press.

Knoll, A., Hijazi, Y., Kensler, A., Schott, M., Hansen, C., and Hagen, H. (2009). Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic. *Computer Graphics Forum*, 28(1):26–40.

Knoll, A., Hijazi, Y., Wald, I., Hansen, C., and Hagen, H. (2007). Interactive ray tracing of arbitrary implicits with SIMD interval arithmetic. In *Proceedings of the 2nd IEEE/EG Symposium on Interactive Ray Tracing*, pages 11–18.

Kolb, C. E. (1992). Rayshade user's guide and reference manual. Draft 0.4.

Kumagai, S. (1980). An implicit function theorem: Comment. *Journal of Optimization Theory and Applications*, 31(2):285–288.

Kwatra, V., Schödl, A., Essa, I., Turk, G., and Bobick, A. (2003). Graphcut textures: Image and video synthesis using graph cuts. *ACM Transactions on Graphics (SIGGRAPH '03 Proceedings)*, 22(3):277–286.

Kwatra, V. and Wei, L.-Y. (2007). Example-based texture synthesis. In *SIGGRAPH 2007 Course Notes*. ACM Press. Course 15.

Lagae, A. and Dutré, P. (2007). A comparison of methods for generating Poisson disk distributions. *Computer Graphics Forum*, 27(1):114–129.

Lane, J. M., Carpenter, L. C., Whitted, T., and Blinn, J. F. (1980). Scan line methods for displaying parametrically defined surfaces. *Communications of the ACM*, 23(1):23–24.

Laur, D. and Hanrahan, P. (1991). Hierarchical splatting: A progressive refinement algorithm for volume rendering. In Sederberg, T. W., editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 285–288. ACM Press.

Levin, E. (1989). Grand challenges to computational science. *Communications of the ACM*, 32(12):1456–1457.

Levoy, M. (1988). Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37.

Lewis, J. P. (1987). Generalized stochastic subdivision. *ACM Transactions on Graphics*, 6(3):167–190.

Lewis, J.-P. (1989). Algorithms for solid noise synthesis. In Lane, J., editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 263–270. ACM Press.

Liktor, G. (2008). Ray tracing implicit surfaces on the GPU. In Wimmer, M. and Lipp, M., editors, *12th Central European Seminar on Computer Graphics (CESCG 2008 Proceedings)*. Technische Universität Wien, Institut für Computergraphik und Algorithmen.

Lippert, L. and Gross, M. H. (1995). Fast wavelet based volume rendering by accumulation of transparent texture maps. *Computer Graphics Forum*, 14(3):431–444.

Logie, J. R. and Patterson, J. (1995). Inverse displacement mapping in the general case. *Computer Graphics Forum*, 14(5):261–273.

Loop, C. (1987). Smooth subdivision surfaces based on triangles. Master's thesis, University of Utah, Deparment of Mathematics.

Loop, C. and Blinn, J. (2006). Real-time GPU rendering of piecewise algebraic surfaces. *ACM Transactions on Graphics (SIGGRAPH '06 Proceedings)*, 25(3):664–670.

Lorensen, W. E. and Cline, H. E. (1987). Marching cubes: A high resolution 3D surface construction algorithm. In Stone, M. C., editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 163–169. ACM Press.

Losasso, F. and Hoppe, H. (2004). Geometry clipmaps: Terrain rendering using nested regular grids. *ACM Transactions on Graphics (SIGGRAPH '04 Proceedings)*, 23(3):769–776.

Maillot, J., Carraro, L., and Peroche, B. (1992). Progressive ray tracing. In Chalmers, A., Paddon, D., and Sillion, F., editors, *Third Eurographics Workshop on Rendering*, Eurographics, pages 9–20. Consolidation Express Bristol.

Mandelbrot, B. B. (1975). On the geometry of homogeneous turbulence with stress on the fractal dimension of the isosurfaces of scalars. *Journal of Fluid Mechanics*, 72:401–416.

Mandelbrot, B. B. (1977). *Fractals: Form, Chance and Dimension.* W.H. Freeman.

Mandelbrot, B. B. (1982). Comment on computer rendering of fractal stochastic models. *Communications of the ACM*, 25(8):581–583.

Mandelbrot, B. B. (1983). *The Fractal Geometry of Nature*. W.H. Freeman.

Mandelbrot, B. B. (1988). Fractal landscapes without creases and with rivers. In Peitgen, H.-O. and Saupe, D., editors, *The Science of Fractal Images*, appendix A, pages 243–260. Springer-Verlag.

Mandelbrot, B. B. and van Ness, J. (1968). Fractional brownian motion, fractional noises and applications. *SIAM Review*, 10:422–437.

Mannersalo, P., Norros, I., and Riedi, R. H. (2002). Multifractal products of stochastic processes: Construction and some basic properties. *Advances in Applied Probability*, 34(4):888–903.

Marshall, R., Wilson, R., and Carlson, W. (1980). Procedure models for generating three-dimensional terrain. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, volume 14, pages 154–162. ACM Press.

Martin, R. R., Shou, H., Voiculescu, I., Bowyer, A., and Wang, G. (2002). Comparison of interval methods for plotting algebraic curves. *Computer Aided Geometric Design*, 19(7).

Mastin, G. A., Watterberg, P. A., and Mareda, J. F. (1987). Fourier synthesis of ocean scenes. *IEEE Computer Graphics and Applications*, 7(3):16–23.

McCool, M. and Fiume, E. (1992). Hierarchical Poisson disk sampling distributions. In *Proceedings of Graphics Interface '92*, pages 94–105. Canadian Information Processing Society.

Messine, F. (2002). Extensions to affine arithmetic: Application to unconstrained global optimization. *Journal of Universal Computer Science*, 8(11):992–1015.

Middleditch, A. E. and Sears, K. H. (1985). Blend surfaces for set theoretic volume modelling systems. volume 19, pages 161–170.

Miller, G. (1986). The definition and rendering of terrain maps. In Evans, D. C. and Athay, R. J., editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, pages 39–48. ACM Press.

Milliron, T., Jensen, R. J., and Barzel, R. (2002). A framework for geometric warps and deformations. *ACM Transactions on Graphics*, 21(1):20–51.

Milnor, J. (1963). *Morse Theory*, volume 51 of *Annals of mathematics studies*. Princeton University Press, Princeton.

Mitchell, D. P. (1990). Robust ray intersection with interval arithmetic. In MacKay, S. and Kidd, E. M., editors, *Proceedings of Graphics Interface '90*, pages 68–74. Canadian Information Processing Society.

Montiel, S. and Ros, A. (2005). *Curves and Surfaces*, volume 69 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI.

Moore, R. (1966). *Interval Arithmetic*. Prentice-Hall.

Moreland, K. and Angel, E. (2003). The FFT on a GPU. In Doggett, M., Heidrich, W., Mark, W., and Schilling, A., editors, *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware (Proceedings Graphics Hardware 2003)*, pages 112–119. Eurographics Association.

Morse, B. S., Yoo, T. S., Chen, D. T., Rheingans, P., and Subramanian, K. R. (2001). Interpolating implicit surfaces from scattered surface data using compactly supported radial basis functions. In Werner, B., editor, *Proceedings of the International Conference on Shape Modeling and Applications (SMI-01)*, pages 89–98. IEEE Computer Society.

Murakami, S.-I. and Ichihara, H. (1987). On a 3D display method by metaball technique. *Trans. IECE Japan, Part D*, J70-D(8):1607–1615. in Japanese.

Murray, J. D. and vanRyper, W. (1994). *Encyclopedia of Graphics File Formats*. O'Reilly & Associates, Inc.

Musgrave, F. K. (2003a). A brief introduction to fractals. In Ebert, D. S. and Musgrave, F. K., editors, *Texturing & Modeling: A Procedural Approach*, chapter 14, pages 429–445. Morgan Kauffman Publishers Inc., 3rd edition.

Musgrave, F. K. (2003b). Genetic textures. In Ebert, D. S. and Musgrave, F. K., editors, *Texturing & Modeling: A Procedural Approach*, chapter 19, pages 547–563. Morgan Kauffman Publishers Inc., 3rd edition.

Musgrave, F. K. (2003c). Mojoworld: Building procedural planets. In Ebert, D. S. and Musgrave, F. K., editors, *Texturing & Modeling: A Procedural Approach*, chapter 20, pages 565–615. Morgan Kauffman Publishers Inc., 3rd edition.

Musgrave, F. K. (2003d). Procedural fractal terrains. In Ebert, D. S. and Musgrave, F. K., editors, *Texturing & Modeling: A Procedural Approach*, chapter 16, pages 489–506. Morgan Kauffman Publishers Inc., 3rd edition.

Musgrave, F. K. (2003e). QAEB rendering for procedural models. In Ebert, D. S. and Musgrave, F. K., editors, *Texturing & Modeling: A Procedural Approach*, chapter 17, pages 509–526. Morgan Kauffman Publishers Inc., 3rd edition.

Musgrave, F. K., Kolb, C. E., and Mace, R. S. (1989). The synthesis and rendering of eroded fractal terrains. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, pages 41–50. ACM Press.

Nagashima, K. (1997). Computer generation of eroded valley and mountain terrains. *The Visual Computer*, 13(9–10):456–464.

Nedialkov, N. S., Kreinovich, V., and Starks, S. A. (2004). Interval arithmetic, affine arithmetic, Taylor series methods: Why, what next? *Numerical Algorithms*, 37(1-4):325–336.

Nishimura, H., Hirai, M., Kawai, T., Kawata, T., Shirakawa, I., and Omura, K. (1985). Object modeling by distribution function and a method of image generation. *Trans. IECE Japan, Part D*, J68-D(4):718–725. in Japanese.

Nishita, T., Sirai, T., Tadamura, K., and Nakamae, E. (1993). Display of the earth taking into account atmospheric scattering. In Kajiya, J. T., editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 175–182. ACM Press.

Notkin, I. and Gotsman, C. (1997). Parallel progressive ray-tracing. *Computer Graphics Forum*, 16(1):43–55. ISSN 0167-7055.

Ong, T. J., Saunders, R., Keyser, J., and Leggett, J. J. (2005). Terrain generation using genetic algorithms. In *GECCO '05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, pages 1463–1470. ACM SIGEVO (formerly ISGEC).

Osher, S. and Fedkiw, R. (2003). *Level Set Methods and Dynamic Implicit Surfaces*, volume 153 of *Applied Mathematical Sciences*. Springer-Verlag Inc., New York, NY.

Painter, J. and Sloan, K. (1989). Antialiased ray tracing by adaptive progressive refinement. In Lane, J., editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 281–288. ACM Press.

Papoulis, A. and Pillai, S. U. (2001). *Probability, Random Variables and Stochastic Processes*. McGraw-Hill, 4th edition.

Pasko, A. and Savchenko, V. (1995). Constructing functionally-defined surfaces. In Gascuel, M.-P. and Wyvill, B., editors, *Proc. Implicit Surfaces '95*, pages 97–106. Eurographics Association.

Peachey, D. R. (2003). Building procedural textures. In Ebert, D. S. and Musgrave, F. K., editors, *Texturing & Modeling: A Procedural Approach*, chapter 2, pages 7–94. Morgan Kauffman Publishers Inc., 3rd edition.

Pedersen, H. K. (1994). Displacement mapping using flow fields. In Glassner, A., editor, *Computer Graphics (SIGGRAPH '94 Proceedings)*, volume 28, pages 279–286. ACM Press.

Peitgen, H.-O., Jürgens, H., and Saupe, D. (1992). *Chaos and Fractals: New Frontiers of Science*. Springer-Verlag.

Perlin, K. (1985). An image synthesizer. In Barsky, B. A., editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 287–296. ACM Press.

Perlin, K. (2001). Noise hardware. In Olano, M., editor, *Real-Time Shading*. ACM Press. SIGGRAPH 2001 Course Notes.

Perlin, K. (2002). Improving noise. *ACM Transactions on Graphics (SIGGRAPH '02 Proceedings)*, 21(3):681–682.

Perlin, K. and Hoffert, E. M. (1989). Hypertexture. In Lane, J., editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 253–262. ACM Press.

Pharr, M. and Hanrahan, P. (1996). Geometry caching for ray-tracing displacement maps. In Pueyo, X. and Schröder, editors, *Procedings of the 1996 Eurographics Workshop on Rendering Techniques*, pages 31–40. Eurographics Association.

Porter, T. and Duff, T. (1984). Compositing digital images. In Christiansen, H., editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253–278. ACM Press.

Pratt, W. K. (1978). *Digital Image Processing*, chapter 10, pages 232–278. John Wiley & Sons.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992). *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition.

Prusinkiewicz, P. and Hammel, M. (1993). A fractal model of mountains and rivers. In *Proceedings of Graphics Interface '93*, pages 174–180. Canadian Information Processing Society.

Purves, D., Paydarfar, J. A., and Andrews, T. J. (1996). The wagon wheel illusion in movies and reality. *Proceedings of the National Academy of Sciences*, 93(8):3693–3697.

Reimers, M. and Seland, J. (2008). Ray casting algebraic surfaces using the frustum form. *Computer Graphics Forum (Eurographics 2008 Proceedings)*, 27(2):361–370.

Requicha, A. and Voelcker, H. (1982). Solid modeling: A historical summary and contemporary assessment. *IEEE Computer Graphics and Applications*, 2(2):9–24.

Ricci, A. (1973). A constructive geometry for computer graphics. *The Computer Journal*, 16(2):157–160.

Rockwood, A. and Owen, J. (1987). Blending surfaces in solid modeling. In Farin, G., editor, *Geometric Modeling: Algorithms and New Trends*, pages 367–383. SIAM, Philadelphia.

Roth, S. D. (1982). Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18(2):109–144.

Roudier, P., Peroche, B., and Perrin, M. (1993). Landscapes synthesis achieved through erosion and deposition process simulation. *Computer Graphics Forum*, 12(3):375–383.

Saupe, D. (1988). Algorithms for random fractals. In Peitgen, H.-O. and Saupe, D., editors, *The Science of Fractal Images*, chapter 2, pages 71–136. Springer-Verlag.

Saupe, D. (1989). Point evaluation of multi-variable random fractals. In Jüergens, H. and Saupe, D., editors, *Visualisierung in Mathematik und Naturissenschaften - Bremer Computergraphik Tage*, pages 114–126. Springer-Verlag.

Schaufler, G. and Priglinger, M. (1999). Efficient displacement mapping by image warping. In Lischinski, D. and Larson, G., editors, *Proceedings of the 1999 Eurographics Workshop on Rendering Techniques*, pages 175–186. Eurographics Association.

Sclaroff, S. and Pentland, A. (1991). Generalized implicit functions for computer graphics. In Sederberg, T. W., editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 247–250. ACM Press.

Sealy, G. and Wyvill, G. (1996). Smoothing of three-dimensional models by convolution. In *Proc. Computer Graphics International 1996*, pages 184–190. IEEE Computer Society Press.

Sederberg, T. W. (1985). Piecewise algebraic surface patches. *Computer Aided Geometric Design*, 2(1–3):53–60.

Sederberg, T. W. and Parry, S. R. (1986). Free-form deformation of solid geometric models. In Evans, D. C. and Athay, R. J., editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 151–160. ACM Press.

Sethian, J. A. (1999). *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science.* Cambridge University Press, 2nd edition.

Sherstyuk, A. (1999a). Fast ray tracing of implicit surfaces. *Computer Graphics Forum*, 18(2):139–147.

Sherstyuk, A. (1999b). Kernel functions in convolution surfaces: A comparative analysis. *The Visual Computer*, 15(4):171–182.

Shinagawa, Y., Kunii, T. L., and Kergosien, Y. L. (1991). Surface coding based on morse theory. *IEEE Computer Graphics and Applications*, 11(5):66–78.

Shou, H., Song, W., Shen, J., Martin, R., and Wang, G. (2006). A recursive taylor method for ray casting algebraic surfaces. In Arabnia, H. R., editor, *Proceedings of the 2006 International Conference on Computer Graphics & Virtual Reality, CGVR 2006, Las Vegas, Nevada, USA, June 26-29, 2006*, pages 196–204. CSREA Press.

Sims, K. (1993). Interactive evolution of equations for procedural models. *The Visual Computer*, 9:466–476.

Smith, A. R. (1984). Plants, fractals and formal languages. In Christiansen, H., editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 1–10. ACM Press.

Smits, B., Shirley, P., and Stark, M. M. (2000). Direct ray tracing of displacement mapped triangles. In Peroche, B. and Rushmeier, H., editors, *Proceedings of the 2000 Eurographics Workshop on Rendering Techniques*, pages 307–318. Eurographics Association.

Snyder, J. M. (1992). Interval analysis for computer graphics. In Catmull, E. E., editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 121–130. ACM Press.

Sproull, R. F. (1991). Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6:579–589.

Stander, B. T. and Hart, J. C. (1997). Guaranteeing the topology of an implicit surface polygonization for interactive modeling. In Whitted, T., editor, *Computer Graphics (SIGGRAPH '97 Proceedings)*, volume 31, pages 279–286. ACM Press.

Stark, M., Shirley, P., and Ashikmin, M. (2005). Generation of stratified samples for B-spline pixel filtering. *Journal of Graphics Tools*, 10(1):61–70.

Stolfi, J. and de Figueiredo, L. H. (1997). Self-validated numerical methods and applications. Course notes for the 21st Brazilian Mathematics Colloquium.

Strang, G. and Nguyen, T. (1996). *Wavelets and Filter Banks*. Wellesley-Cambridge Press, Wellesley, MA.

Sutherland, I. E., Sproull, R. F., and Schumacker, R. A. (1974). A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55.

Szirmay-Kalos, L. and Umenhoffer, T. (2008). Displacement mapping on the GPU – State of the art. *Computer Graphics Forum*, 27(6):1567–1592.

Tanner, C. C., Migdal, C. J., and Jones, M. T. (1998). The Clipmap: A virtual mipmap. In Cohen, M., editor, *Computer Graphics (SIGGRAPH '98 Proceedings)*, volume 22, pages 151–158. ACM Press.

Thain, D., Tannenbaum, T., and Livny, M. (2005). Distributed computing in practice: the condor experience. *Concurrency: Practice and Experience*, 17(2-4):323–356.

Thévenaz, P. and Unser, M. (2001). High-quality isosurface rendering with exact gradient. In *International Conference on Image Processing*, volume 1, pages 854–857.

Turk, G. and O'Brien, J. F. (1999). Shape transformation using variational implicit functions. In Rockwood, A., editor, *Computer Graphics (SIGGRAPH '99 Proceedings)*, volume 33, pages 335–342. ACM Press.

Ulichney, R. A. (1988). Dithering with blue noise. *Proceedings of the IEEE*, 76(1):56–79.

Upstill, S. (1990). *The Renderman Companion*. Addison-Wesley, Reading, MA.

van Wijk, J. J. (1985). Ray tracing objects defined by sweeping a sphere. *Computers & Graphics*, 9(3):283–290.

van Wijk, J. J. (1991). Spot noise. In Sederberg, T. W., editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 309–318. ACM Press.

VanRullen, R. and Koch, C. (2003). Is perception discrete or continuous? *TRENDS in Cognitive Science*, 7(5):207–213.

VanRullen, R., Reddy, L., and Koch, C. (2005). Attention-driven discrete sampling of motion perception. *Proceedings of the National Academy of Sciences*, 102(14):5291–5296.

Velho, L., Perlin, K., Ying, L., and Biermann, H. (2001). Procedural shape synthesis on subdivision surfaces. In *XIV Brazilian Symposium on Computer Graphics and Image Processing (Proc. SIBGRAPI 2001)*, pages 146–153. Sociedade Brasileira de Computação, IEEE Computer Society Press.

Voss, R. F. (1983). Fourier synthesis of Gaussian fractals: 1/f noises, landscapes and flakes. In *SIGGRAPH '83 Tutorial on State of the Art Image Synthesis*. ACM Siggraph.

Voss, R. F. (1985). Random fractal forgeries. In Earnshaw, R. A., editor, *Fundamental Algorithms for Computer Graphics*, pages 805–835. Springer-Verlag.

Voss, R. F. (1988). Fractals in nature: From characterization to simulation. In Peitgen, H.-O. and Saupe, D., editors, *The Science of Fractal Images*, chapter 1, pages 21–70. Springer-Verlag.

Wang, S. W. and Kaufman, A. E. (1995). Volume sculpting. In Hanrahan, P. and Winget, J., editors, *1995 Symposium on Interactive 3D Graphics*, pages 151–156. ACM SIGGRAPH.

Wang, X., Tong, X., Lin, S., Hu, S., Guo, B., and Shum, H.-Y. (2004). Generalized displacement maps. In Jensen, H. and Keller, A., editors, *Proceedings of the 2004 Eurographics Workshop on Rendering Techniques*, pages 227–234. Eurographics Association.

Ward, G. J., Rubinstein, F. M., and Clear, R. D. (1988). A ray tracing solution for diffuse interreflection. In Dill, J., editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 85–92. ACM Press.

Watt, A. (1989). *Fundamentals of Three-Dimensional Computer Graphics*. Addison-Wesley Publishing Company.

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

Wikipedia (2008). Embarrassingly parallel — wikipedia, the free encyclopedia. [Online; accessed 31-October-2008].

Williams, L. (1983). Pyramidal parametrics. In Tanner, P. P., editor, *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pages 1–11. ACM Press.

Witkin, A. P. and Heckbert, P. S. (1994). Using particles to sample and control implicit surfaces. In Glassner, A., editor, *Computer Graphics (SIGGRAPH '94 Proceedings)*, volume 28, pages 269–278. ACM Press.

Wolfe, J. M., Levi, D., Kluender, K., Bartoshuk, L., Herz, R., Klatzky, R. L., and Lederman, S. (2005). *Sensation and Perception*. Sinauer Associates Inc.

Woodwark, J. R. and Quinlan, K. M. (1982). Reducing the effect of complexity on volume model evaluation. *Computer Aided Design*, 14(2):89–95.

Worley, S. P. (1996). A cellular texture basis function. In Rushmeier, H., editor, *Computer Graphics (SIGGRAPH '96 Proceedings)*, volume 30, pages 291–294. ACM Press.

Worley, S. P. (2003). Practical methods for texture design. In Ebert, D. S. and Musgrave, F. K., editors, *Texturing & Modeling: A Procedural Approach*, chapter 6, pages 179–201. Morgan Kauffman Publishers Inc., 3rd edition.

Worley, S. P. and Hart, J. C. (1996). Hyper-rendering of hyper-textured surfaces. In *Proc. of Implicit Surfaces '96*, pages 99–104.

Wu, S.-T. and de Gomensoro Malheiros, M. (1999). On improving the search for critical points of implicit functions. In *Proc. Implicit Surfaces '99*, pages 73–80, Bordeaux. Eurographics/SIGGRAPH.

Wyvill, B. (1993). Metamorphosis of implicit surfaces. In *Modeling, Visualizing and Animating with Implicit Surfaces*, volume 25 of *SIGGRAPH 1993 Course Notes*. ACM Press.

Wyvill, B., Guy, A., and Gallin, E. (1999). Extending the CSG tree: Warping, blending and boolean operations in an implicit surface modeling system. *Computer Graphics Forum*, 18(2):149–158.

Wyvill, B., Leclercq, A., and Galin, E. (1998). Extending the CSG tree: Warping, blending and boolean operations in an implicit surface modeling system. In *Proc. Implicit Surfaces '98*, pages 113–121.

Wyvill, G., McPheeters, C., and Wyvill, B. (1986). Data structure for soft objects. *The Visual Computer*, 2(4):227–234.

Wyvill, G. and Trotman, A. (1990). Ray-tracing soft objects. In *Computer Graphics International'90*, pages 469–475.

Yellot, Jr, J. I. (1983). Spectral consequences of photoreceptor sampling in the rhesus retina. *Science*, 221:382–395.

Zhou, H., Sun, J., Turk, G., and Rehg, J. M. (2007). Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):834–848.